chapter 1 Tehnologii de dezvoltare web

1.1 Concepte de bază

1.1.1 Aplicații distribuite

O **aplicație distribuită** este o aplicație a cărei componente sunt localizate și executate în spații de memorie (calculatoare, platforme interconectate) diferite.

Componentele aplicației comunică între ele prin schimburi de mesaje.

Raționalitatea utilizării sistemelor și aplicațiilor distribuite ține de câțiva factori dintre care menționăm:

- utilizarea mai eficientă a resurselor disponibile
- dezvoltarea modulară a aplicațiilor
- robustețe sporită
- scalabilitate
- portabilitate
- administrare simplificată în raport cu complexitatea aplicației

Componentele unei aplicații web sunt, în general, independente, comunicarea dintre ele realizându-se pe baza unui contract (interfață) prestabilit.

Modificările interne aduse unei componente nu (ar trebui să) se reflectă(e) în interfețele componentei, ele sunt transparente pentru dezvoltatorii celorlalte componente.

Aplicațiile distribuite sunt dominante la ora actuală, menționăm doar câteva exemple dintre ele:

- jocuri online cu mai mulți participanți
- baze de date distribuite
- aplicații de administrare a rețelelor de telecomunicații
- aplicații de chat
- aplicații de prognoză climatică

Din punct de vedere arhitectural, distingem câteva modele de bază, și anume:

- client-server – clienții trimit cereri unei aplicații server și primesc un răspuns de la aceasta
- 3-tier (pe 3 nivele) – arhitectură în care partea de logică funcțională (business logic) a aplicației este mutată pe un nivel intermediar
- n-tier (pe n nivele) – arhitecturi în care aplicațiile web transmit mai departe cererile lor la alte servicii ale companiei
- peer-to-peer – arhitecturi în care responsabilitățile sunt împărțite „egal” între toate sistemele participante

1.1.2 Aplicații web (web applications)

Aplicațiile web sunt aplicații distribuite care au un profil specific, și anume:

- implementează paradigma client-server, în varianta ei primară sau 3(n)-tier
- utilizează tehnologiile deschise ale World Wide Web (www)

Partea de client a unei aplicații web are ca mediu de execuție, în majoritatea cazurilor, un browser.

1.2 tehnologii de programare front-end

Tehnologiile de programare a părții de front-end (client, în genere) sunt diverse și constant avem parte de noi apariții care rezolvă probleme mai vechi sau mai noi sau care contribuie la eficientizarea activității de programare web.

Site-ul tutorialspoint.com oferă o lista extinsă a acestor tehnologii, dintre care menționăm unele de interes, în ordine alfabetică. Nu au fost incluse tehnologiile Java EE, care sunt prezentate separat.

Link-ul este https://www.tutorialspoint.com/web_development_tutorials.htm

- Ajax – un set de tehnologii care permit comunicarea asincronă a clientului cu serverul având ca rezultat pagini web dinamice
- AngularJS – un mediu de dezvoltare JavaScript
- Bootstrap – cel mai popular cadru de dezvoltare a front-end-ului paginilor și aplicaților web
- CSS – folosit pentru controlul stilistic al paginilor web
- Drupal – un sistem de administrare al conținutului (CMS) open source, care permite organizarea, administrarea și publicare unui site web
- HTML5 – versiunea curentă și cea mai avansată a limbajului de „programare” web HTML
- HTTP – protocol la nivel de aplicație, fundamental pentru comunicarea pe internet
- JavaScript – un limbaj de programare interpretat, utilizat pentru crearea de aplicații pentru rețele
- Joomla – un alt sistem de administrare al conținutului (CMS) open source, mai sofisticat
- jQuery – o bibliotecă JavaScript care simplifică procesarea evenimentelor, interacțiuni Ajax precum și procesarea documentelor
- Less – un pre-procesor CSS care extinde programatic CSS-ul
- Ruby on Rails – un mediu de dezvoltare pentru aplicații web bazat pe limbajul Ruby
- WebGL – Web Graphics Library – folosit pentru afișare de grafică 2D și grafică 3D interactivă
- Wordpress – un CMS (sistem de administrare al conținutului) de categorie mai ușoară, open source

Unele din aceste tehnologii vor fi detaliate pe parcursul acestui curs.

1.3 Soluții la nivel de companie (enterprise solutions)

În materie de infrastructuri dominante distingem două categorii de soluții majore pentru aplicații la nivel de server:

- soluții bazate pe Php
- soluții bazate pe Java EE (Java Enterprise Edition)

1.4 Php versus Java EE

În această secțiune încercăm să răspundem unei întrebări legitime – de ce să ne încurcăm cu servlete, java server pages or java server faces când putem obține aceleași rezultate și mai rapid utilizând un limbaj versatil precum Php? Fără a ignora calitățile majore ale acestui limbaj și cadru de programare, trebuie amintite câteva din slăbiciuni, care sunt inerente naturii sale.

În primul rând, scripturile Php sunt legate de un anumit server web iar pentru generarea de conținut dinamic trebuie scrise interogări SQL explicite. Chiar și aceste interogări sunt scrise într-un mod care este specific unui anumit serviciu de baze de date. În PHP, programatorul aplicației scrie interogări SQL pe care le încorporează direct în script, amestecând prezentarea și logica aplicației în acest proces. Nici un sprijin direct

nu este asigurat pentru gestionarea setului de componente, de administrare a ciclului de viață sau a sesiunii client, a conexiunilor cu baza de date, persistență, gestiunea tranzacțiilor, autentificare și control al accesului. Pe de altă parte, toate acestea pot fi realizate printr-un server de aplicații.

Desigur, că în timp, specificația și implementarea Php-ului evoluează și soluționează multe din aceste slăbiciuni ale sale.

Pe de altă parte, Php e simplu de învățat, documentația este bună și există o cantitate imensă de scripturi Php disponibile.

Pentru cei preocupați de orientare profesională, vestea bună este ambele seturi de competențe, fie în domeniu Php sau Java EE sunt la mare căutare.

Dacă doriți o înțelegere mai profundă a acestei dezbateri, următoarele link-uri pot ajuta.

- http://stiri.rol.ro/dezbatere-forum-la-link-academy-php-vs-java-820468.html
- http://www.cs.montana.edu/~tosun/phpvsjava.pdf
- http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=java&lang2=php
- http://raibledesigns.com/rd/entry/php_vs_java_which_is
- http://brand-maestro.com/php-vs-java-programming-language-better-future/

1.5 Structura cursului

Cursul este axat pe tehnologiile principale care fac parte din specificația Java EE precum și pe tehnologiile web adiacente, care nu sunt parte a acestei specificații, dar dar care completează structural arhitectura aplicațiilor web bazate pe tehnologiile și interfețele de programare Java EE.

În **prima parte** este prezentată specificația Java EE, modelul arhitectural al aplicațiilor Java EE, precum și tehnologiile și API-urile care sunt parte a acestei specificații.

În a **doua parte** sunt prezentate tehnologiile de dezvoltare web de ordin general precum și limbajele de programare specifice, și anume:

- protocolul de comunicare HTTP
- limbajele HTML(5), CSS, LESS, JavaScript
- specificația DOM, biblioteca jQuery, tehnologiile Ajax

În a **treia parte** sun prezentate tehnologiile majore ale specificației Java EE, și anume:

- Servlete
- Java DataBase Connectivity
- Java Server Pages
- Java Server Faces
- Java Naming and Directory Interface
- Java Message Service
- Enterprise Java Beans
- Web Sockets
- JavaScript Object Notation - JSON
- Web Services

chapter 2 Java EE

Conținutul acestui capitol se bazează în mare măsură pe tutorialul Java EE, versiunea 7. Acest tutorial poate fi vizualizat/descărcat de la adresa - https://docs.oracle.com/javaee/7/tutorial/ .

Pentru noua versiune (cea curentă), Java EE 8, a cărei specificație a fost finalizată pe 31.08.2017, tutorialul poate fi văzut la link-ul – https://javaee.github.io/tutorial/toc.html.

Java EE este o platformă pentru aplicații la nivel de companie (enterprise level). Ea oferă interfețe de programare a aplicațiilor (API) precum și un mediu de execuție al acestora.

Java EE extinde platforma standard Java (Java SE), oferind interfețe de programare pentru ahitecturi distribuite și pe mai multe nivele, pentru servicii web și pentru asociere relațională a obiectelor.

Aplicațiile pentru Java EE sunt dezvoltate în marea majoritate a cazurilor în limbajul de programare Java.

Specificațiile Java EE sunt create și administrate în cadrul Java Community Process, rezultatul acestor activități fiind materializat în specificații denumite JSR – Java Specification Request.

2.1 Scurt istoric

Platforma Java EE a fost cunoscută inițial sub numele Java 2 Platform, Enterprise Edition sau J2EE, corespunzând ediției standard (SE) 1.2. Începând cu versiunea (1.)5, numele a fost schimbat în Java EE 5. Versiunea curentă este Java EE 8

Prima specificație (J2EE 1.2) datează din decembrie 1999, fiind urmată de specificația J2EE 1.3 din septembrie 2001. J2EE 1.4 apare în noiembrie 2003 iar Java EE 5 în mai 2006.

Următoarea specificație (Java EE 6) apare în decembrie 2009, urmată în august 2013, de versiunea Java EE 7.

Versiunea curentă este Java EE 8 iar specificația sa (JSR 366) - https://www.jcp.org/en/jsr/detail?id=366 – a apărut pe 18.09.2017.

2.2 Starea versiunii curente

Versiunea curentă (în materie de implementare) a specificației Java EE este JAVA EE 8 și se bazează pe SDK-ul (Software Development Kit) Java EE 8.

- link pentru versiunea finală - https://github.com/javaee/javaee-spec/blob/master/download/JavaEE8_Platform_Spec_FinalRelease.pdf

Pentru a înțelege mai bine modul în care a evoluat această specificație, puteți consulta link-urile:

- https://javaee8.zeef.com/arjan.tijms
- https://dzone.com/articles/java-ee-8-current-status-case-study-for-completed
- https://www.javacodegeeks.com/2014/12/whats-up-with-java-ee-8.html

2.3 Modelul aplicațiilor Java EE

Fundamentul aplicațiilor Java EE este reprezentat de către **limbajul de programare Java** și de către **Mașina Virtuală Java** (Java Virtual machine – JVM). Această combinație oferă un nivel înalt de portabilitate, scalabilitate și eficiență în programare.

Scopul Java EE este de a oferi suport pentru aplicații la nivel de companie care implementează servicii pentru clienți, angajați, furnizori, parteneri sau pentru alte entități care au o contribuție la dezvoltarea

companiei.

Astfel de aplicații sunt prin natura lor complexe și deseori au de accesat date de pe platforme diferite iar rezultatele procesării pot fi distribuite la o varietate de clienți. Pentru a administra mai eficient aceste aplicații, parte de logică a aplicațiilor este localizată în nivelul de mijloc. Acesta beneficiază de hardware dedicat și are acces complet la resursele și datele companiei.

Modelul Java EE al aplicațiilor definește o arhitectură pentru implementarea serviciilor ca aplicații pe mai multe nivele care oferă scalabilitatea, accesibilitatea și administrabilitatea cerută de aplicații la nivel de companie. Acest model partiționează munca necesară pentru implementarea unui serviciu pe mai multe nivele în două părți:

- logica și prezentarea aplicației care sunt implementate de către programator
- serviciile standard ale sistemului care care sunt oferite de către platforma Java EE

2.4 Aplicații distribuite pe mai multe nivele (sectoare, tiers)

Platforma Java EE utilizează un model de aplicații pe mai multe nivele (tiers). O aplicație Java EE este alcătuită din mai multe componente care au o funcționalitate distinctă și care sunt localizate pe sisteme diferite, în funcție de specificul componentei.

Figura de mai jos descrie o aplicație Java EE generică împreună cu diferitele combinații de componente precum și distribuția lor în nivele diferite.

- nivelul **client** – conține componente care sunt executate pe sistemul clientului
- nivelul **web** – conține componente care sunt executate pe serverul Java EE
- nivelul **business** – conține componente care implementează logica aplicației
- nivelul **EIS** (Enterprise Information System) – conține software executat pe serverul EIS

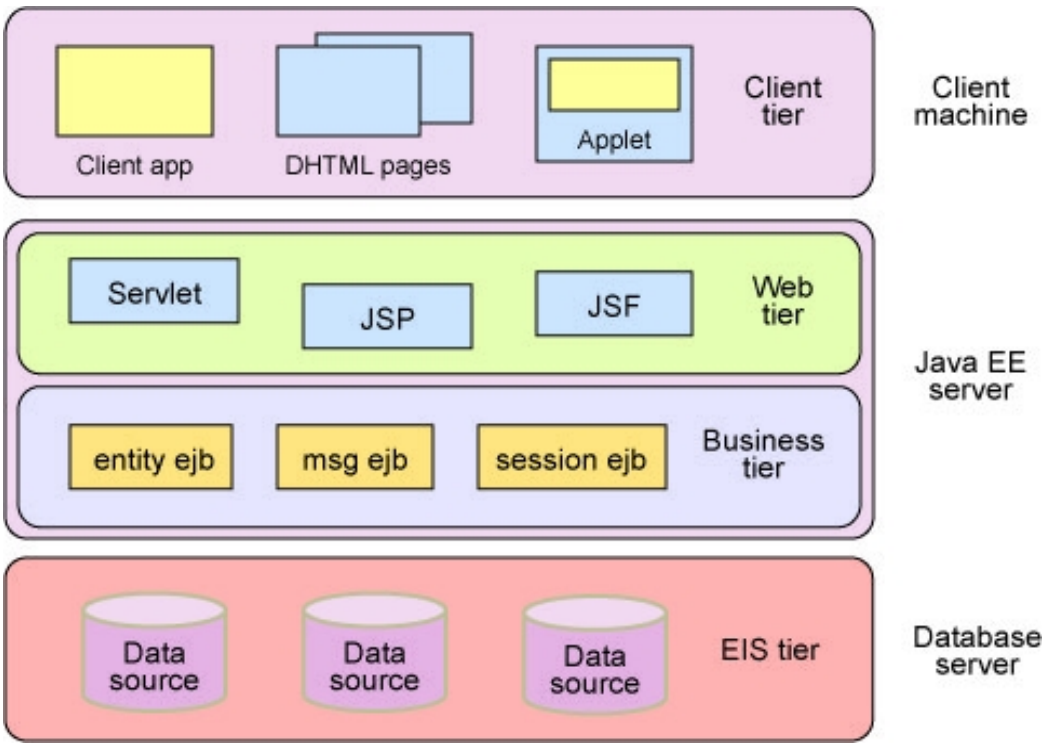


Figura 2.1 - Aplicații distribuite pe mai multe nivele, Java EE

2.4.1 Componente Java EE

O aplicație Java EE este constituită din **componente**. O componentă Java EE este o unitate funcțională de sine stătătoare, care este asmblată într-o aplcație Java EE prin clasele și fișierele sale și care comunică cu celelalte componente ale aplicației.

Specificația Java EE precizează următoarele componente:

- aplicațiile client și paginile HTML sunt componentele care se execută la nivelul client
- componentele care se execută pe server sunt servletele Java, paginile JSP precum și componentele Java Server Faces
- componentele de business (de logica aplicației) care se execută pe server sunt EJB-urile (Emterprise Java Beans)

Componentele Java EE sunt scrise în limbajul Java și sunt compilate ca (aproape) orice alt program scris într-un limbaj de programare (cu compilare).

Diferența dintre componentele Java EE și clasele Java standard este căcă aceste componente sunt asamblate într-o aplicație Java EE, sunt verificate pentru a avea un format corect și conform specificației Java EE, sunt desfășurate în producție unde sunt administrate și executate de către un server Java EE.

2.4.2 Nivelul client

Un client Java EE poate fi un client web, o aplicație client sau un applet. Ne vom concentra asupra primelor două categorii, deoarece applet-urile sunt utilizate într-o măsură mult mai mică în ultima vreme.

Client web

Un client web costă din două părți:

- pagini web dinamice, conținând diferite tipuri de limbaje de markup, precum HTML sau XML
- un browser web, care afișează paginile trimise ca răspuns de către server

Un client web este denumit, în general, un client subțire. Aceștia nu interacționează, în general, cu serviciile de baze de date, nu conțin o parte seminificativă a logicii aplicației și nu interacționeaza cu elemente legacy ale sistemului informatic.

Aplicațiile client

O aplicație client este executată pe o stație client și oferă utilizatorilor posibilitatea de a executa sarcini care implică o interfață mai complexă, creată, în general, prin intermediul API-ului oferit de Swing sau AWT (Abstract Window Toolkit).

O aplicație client poate comunica cu un servlet sau cu o altă componentă web localizată pe server, prin intermediul protocolului HTTP. Aplicațiile client pot fi scrise și în alte limbaje decât Java.

2.4.3 Comunicarea cu serverul Java EE

Figura de mai jos ilustrează modalitățile de comunicare ale clienților cu nivelul business. Această comunicare se face fie direct, sau, în cazul unui client executat în browser, prin intermediul paginilor JSP sau al servletelor din nivelul web

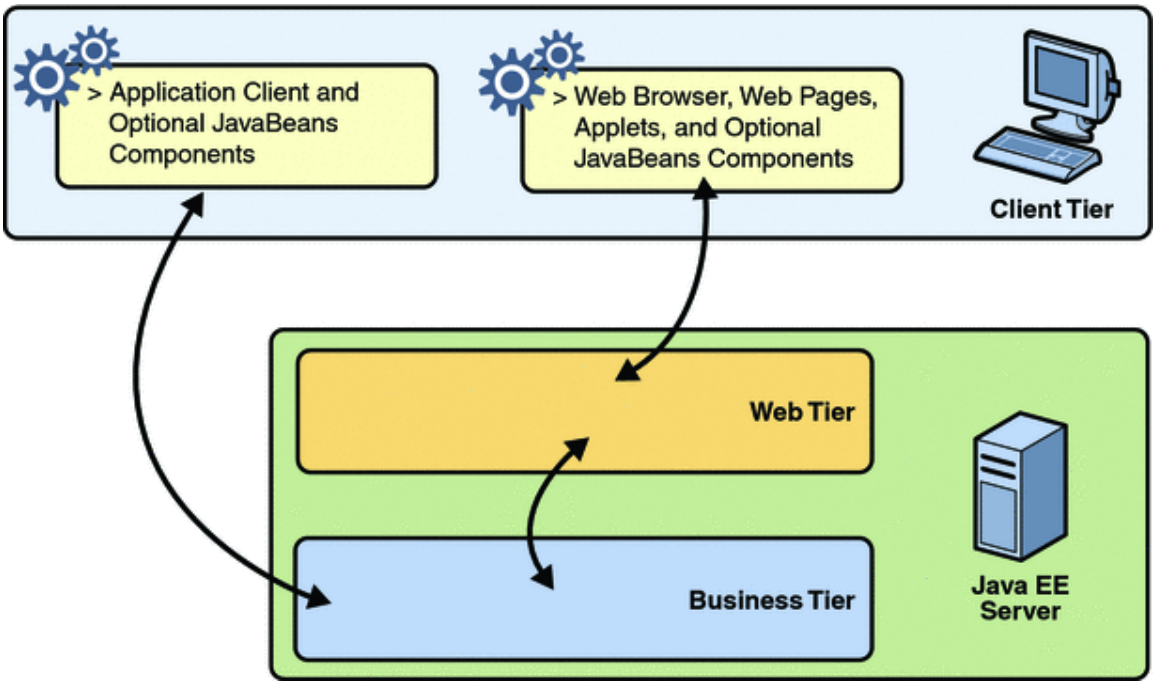


Figura 2.2 – Comunicarea cu serverul

2.4.4 Componente web

Componentele web sunt localizate în nivelul web al serverului. Acestea pot fi servlete, pagini JSP sau componente bazate pe tehnologia Java Server Faces (JSF).

Servletele sunt clase Java care procesează dinamic cererile clienților și care generează răspunsuri, trimise ulterior clienților.

Paginile **JSP** sunt documente de tip text, care sunt convertite și executate ca servlete și oferă o abordare mai naturală a procesului de creare a unui conținut static.

Tehnologia **JSF** extinde tehnologiile servlet și JSP și oferă un cadru pentru construirea de componente pentru interfața utilizator.

2.4.5 Componente business (logica aplicației)

Logica aplicației este, în general, implementată prin intermediul **EJB** (Enterprise Java Beans), figura de mai jos arată modalitatea prin care un EJB primește date de la client, le procesează și, dacă este necesar, le transmite nivelului EIS pentru a fi stocate. Totodată, un EJB poate primi date (în general, prin interogări ale bazelor de date) de la nivelul EIS, le procesează și apoi le trimite clientului.

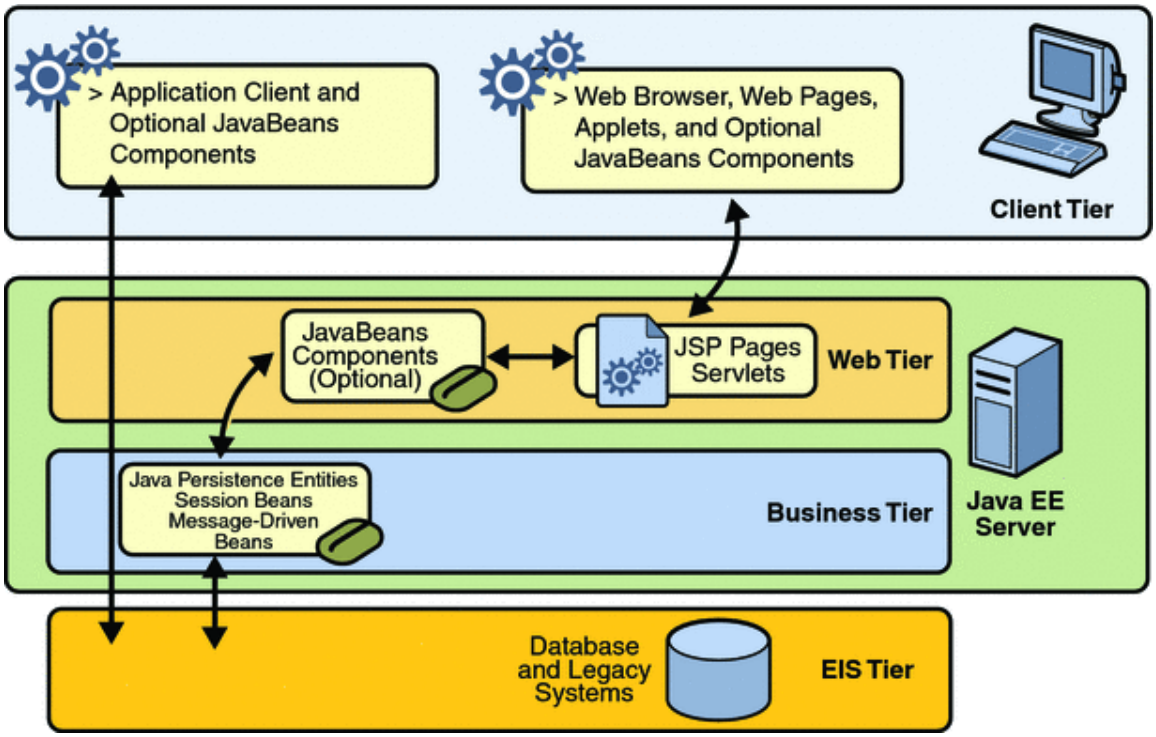


Figura 2.3 - Nivelele web, business, EIS

2.4.6 Nivelul EIS (Enterprise Information Systems)

Nivelul Sistemului de Informații al Companiei (EIS) administrează sistemul de programe EIS și include sisteme de infrastructură ale companiei precum ERP (Enterprise Resource Planning) (planificarea resurselor companiei), procesarea tranzacțiilor mainframe, servicii de baze de date sau alte sisteme informatice pre-existente (legacy systems).

În cele mai multe scenarii, nivelul EIS poate oferi informații despre modalitatea de conectare a aplicațiilor la serviciile de baze de date.

2.5 Conatinere Java EE

Compnentele unei aplicații distribuite nu sunt, în general, unități de execuție (executabile) de sine stătătoare. Aceste componente își desfășoară ciclul de viață într-un mediu controlat, oferit de către containerele Java EE.

Rațiunea de-a fi a acestor containere ține de complexitatea crescută a aplicațiilor distribuite într-un mediu dinamic, de cerințele de portabilitate, scalabilitate, robustețe sau de administrare care sunt impuse aplicațiilor dintr-un sistem de nivel de companie (enterprise).

2.5.1 Servicii oferite de containere

Containerele reprezintă mediul de execuție al componentelor și sunt interfața dintre acestea și funcțiile de bază care sunt implementate de platforma de suport. Înainte de a putea fi executată, componenta trebuie asamblată și desfășurată în containeriul adecvat.

Procesul de asamblare implică specificarea elementelor de configurare a containerului pentru toate componentele unei aplicații precum și pentru aplicația însăși. Aceste elemente de configurare particularizează suportul oferit de către serverul de aplicații Java EE, incluzând servicii precum securitate,

administrarea tranzacțiilor, căutări JNDI, conectare de la distanță, etc.

Iată câteva dintre aceste servicii:

- servicii de securitate – componentele web sau EJB-urile pot fi configurate pentru a permite accesul în funcție de credențialele entității care cere acest acces
- suport pentru tranzacțiilo – modelul de tranzacționare al Java EE permite specificarea de relații dintre metodele care alcătuiesc o tranzacție, acestea fiind tratate ca o singură entitate
- servicii de director și de căutare – aceste servicii sunt oferite de către JNDI, care permite integrarea unor servicii de director și de căutare implementate pe platforme diferite și distribuite
- servicii de comunicare – permit comunicarea dintre diferitele componente ale unei aplicații web, dintre acestea și containerele care le oferă mediul de execuție precum și apeluri de la distanță între componente localizate pe alte platforme.

Containerele permit de asemenea administrarea serviciilor care nu sunt configurabile, precum administrarea ciclului de viață al EJB-urilor și al servletelor, gruparea resurselor de conectare la serviciile de baze de date, serializarea datelor, administrarea alocării și dealocării variabilelor în memorie, asigurarea accesului la API-urile platformei Java EE.

2.5.2 Tipuri de containere

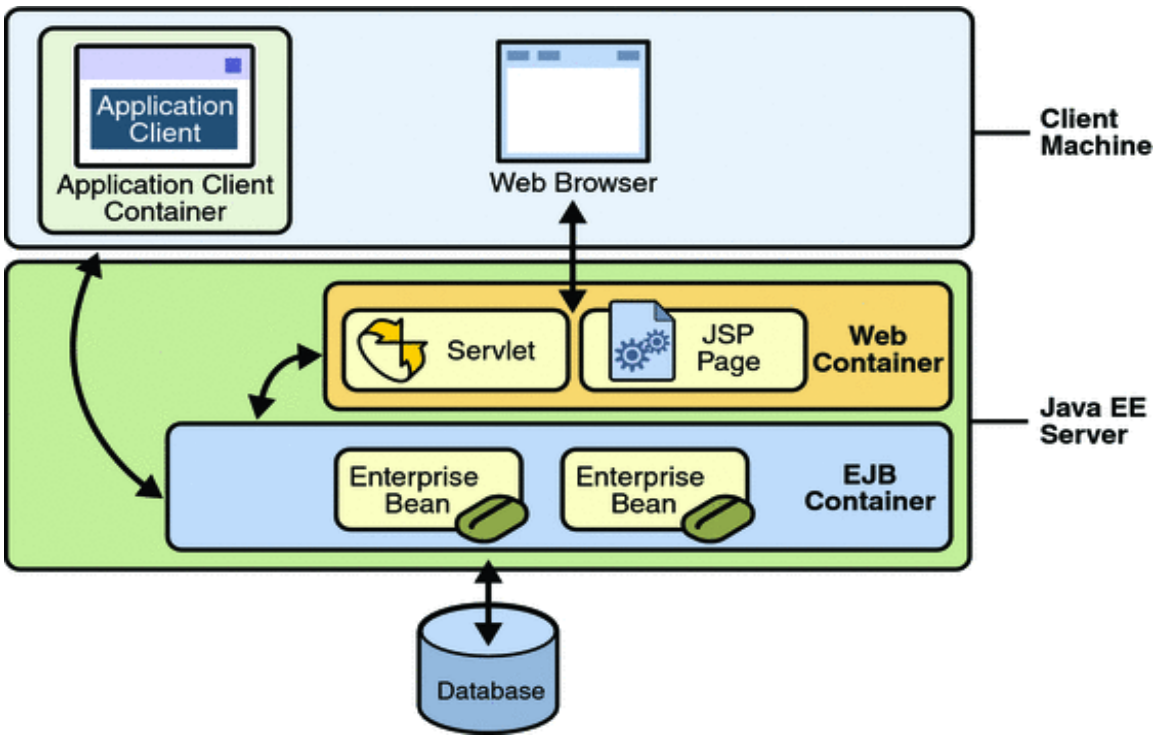


Figura 2.4 – Tipuri de containere

- container EJB – administrează execuția EJB-urilor pentru aplicațiile Java EE. EJB-urile și containerele lor sunt localizate în serverul Java EE
- container Web – administrează execuția paginilor JSP și a servletelor pentru aplicațiile Java EE. Componentele web și containerele lor sunt localizate în serverul Java EE
- container pentru aplicațiile client – acestea sunt executate la nivelul client

- container pentru applet-uri – administrează execuția applet-urilor. Constă dintr-un browser web și un plugin Java

2.6 Suport pentru servicii web

Serviciile Web sunt aplicaţii la nivel de întreprindere, bazate pe web, care folosesc standarde deschise și protocoale de transport bazate pe XML, pentru schimbul de date cu aplicațiile client. Platforma Java EE oferă API-uri XML şi instrumentele de care e nevoie rapid pentru proiectarea, dezvoltarea, testarea şi implementarea serviciilor web şi a clienţilor care să coopereze cu alte servicii web şi clienţi care rulează pe platforme bazate pe Java sau pe platforme care nu sunt bazate pe Java.

2.6.1 XML

XML – Extensible Markup Language - este un standard extensibil, bazat pe text, definit pe mai multe platforme, pentru reprezentarea datelor. Când datele XML sunt schimbate între părţi, părţile sunt libere să creeze propriile tag-uri pentru a descrie datele, stabilite pentru aceste liste pentru a specifica care etichete pot fi utilizate pentru un anumit tip de document XML. Meta-limbajul XML este o simplificare a limbajului SGML, din care se trage și limbajul HTML, fiind proiectat pentru transferul de date peste internet.

2.6.2 SOAP – Simple Object Access Protocol

SOAP este o specificație de protocol pentru schimbul de informații structurate pentru servicii web. Raționalitatea acetui protocol ține de extensibiltatea, neutralitatea și independența sa.

SOAP permite comunicarea dintre procese care sunt executate pe platforme diferite utilizând XML.

Un mesaj SOAP este un fișier XML care conține următoartele elemente:

- o anvelopă – identifică documentul XML ca mesaj SOAP - obligatoriu
- header – conține informații de tip header
- conținut – conține cereri și răspunsuri - obligatoriu
- informații de eroare – conține informații despre erorile survenite în procesarea mesajului

2.6.3 Formatul standard WSDL – Web Services Description Language

WSDL este un limbaj de definere a interfețelor care descrie funcționalitatea oferită de serviciile web.

Un fișier WSDL oferă o descriere care poate fi citită de o mașină care oferă informații despre modul de apelare a serviciului, parametrii necesari și structurile de date pe care le returnează.

2.7 Tehnologii majore și interfețe de programare (API) în Java EE

În această secțiune trecem în revistă tehnologiil majore precum și API-urile (Application Programming Interface) care sunt relevante pentru acest curs. Pentru o listă exhaustivă a acestor tehnologii precum și a API-urilor ce fac parte din specificația JAVA EE versiunea 7 cititorul poate consulta paragrafele 1.8 din tutorialul pentru Java EE 8.

Figura de mai jos ilustrează API-urile definite conform specificației Java EE 6.

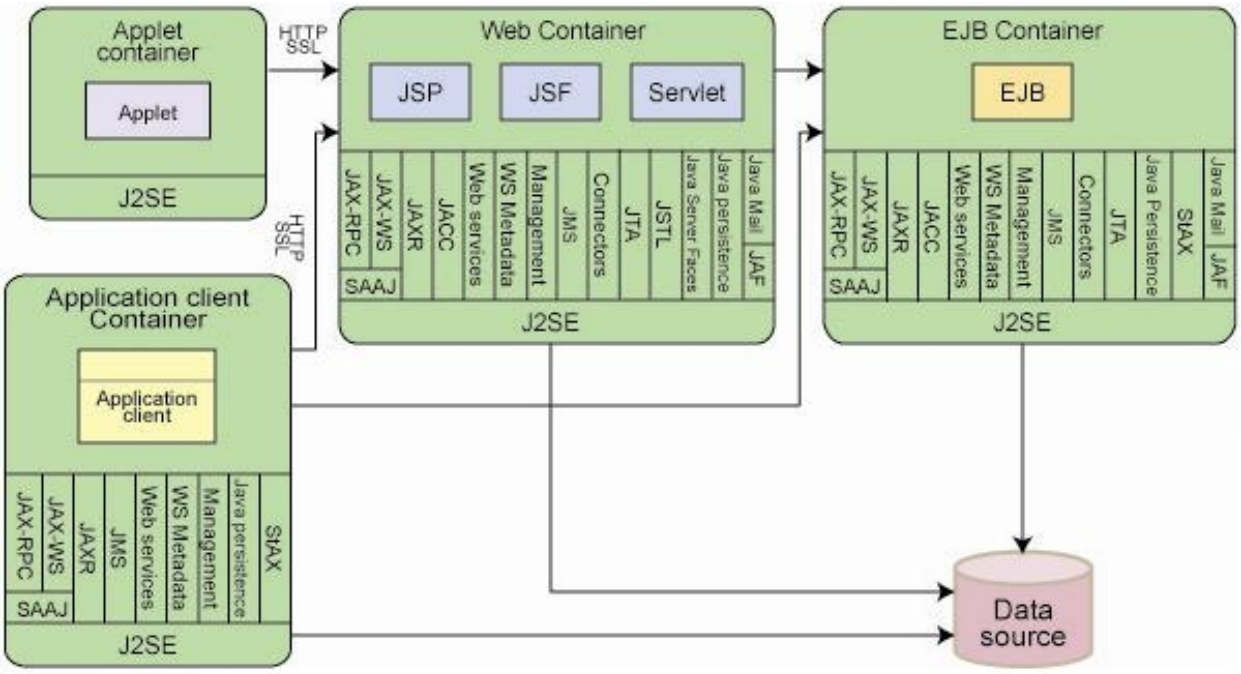


Figura 2.2 - API-urile specifice Java EE 6

2.7.1 Java Servlet

Din punct de vedere tehnic, un **servlet** este o clasă Java care implementează interfața Servlet, sau, în marea majoriatate a cazurilor, extinde clasa abstractă HttpServlet.

Din punct de vedere practic, servletele au rolul dea procesa cererile clienților, utilizând funcționalitatea oferită de către nivelele de server sau de către serviciile de baze de date și de a genera un răspuns care este apoi trimis acestora.

Două din noutățile oferite de specicația Java Servlet în Java EE 7:

- operații asincrone de I/O
- adaptare la noua versiune de HTTP

Specificația Java EE 7/8 utilizează specificația Servlet 3.1/4.0.

2.7.2 Java Server Pages

O pagină JSP este, în linii mari, o pagină HTML care conține elemente (tag-uri) noi, sub forma de directive sau acțiuni standard, precum și cod Java.

Ca funcționalitate, JSP este similar cu Php și ASP, diferența primară fiind aceea că utilizează limbajul Java.

Pentru execuția unei pagini JSP este nevoie de un server web compatibil cu un container de servlete, precum Apache Tomcat sau Jetty. Toate serverele de aplicații Java EE oferă această facilitate.

De fapt, paginile JSP sunt convertite în servlete înainte de procesarea cererilor care le sunt adresate.

Java EE 7/8 utilizează specificația JSP 2.3, dar, totdată, recomandă utilizarea Facelet-urilor (parte a tehnologiei JSF) ca tehnologie pentru interfața cu utilizatorul în aplicații noi.

2.7.3 Java Server Pages Standard Tag Library

Biblioteca standard de tag-uri (elemente) JSP (JSTL) încapsulează funcționalitatea de bază, comună

marii majorități a aplicațiilor JSP. Standardizarea acestor elemente evită utilizarea unor seturi de elemente specifice unei companii sau alteia.

Totodată, această standardizare permite desfășurarea aplicațiilor în orice conainer JSP care oferă suport pentru JSTL.

JSTL are elemente de iterare, condiționale (pentru controlul execuției), elemente pentru procesarea de documente, elemente de localizare, elemente pentru accesul bazelor de date, precum și elemente pentru funcțiile uzuale.

Java EE 7/8 utilizează specificația JSTL 1.2.

2.7.4 Java Server Faces

Tehnologia JSF oferă un cadru pentru construirea de aplicații web. Principalele sale componente sunt:

- un cadru pentru dezvoltarea de componente de interfață vizuală (GUI)
- un model flexibil de afișare a componentelor în versiuni diferite de HTML sau în alte limbaje de markup sau în alte tehnologii.
- un RenderKit standard pentru generarea de markup specific pentru HTML
- support pentru validarea introducerilor
- procesarea evenimentelor
- conversii de date între obiecte și componente
- configurarea navigației între pagini

Specificația Java EE 7 aduce și alte facilități, precum:

- Markup pentru HTML5
- fluxuri pentru Faces
- contracte pentru biblioteci de resurse

Specificația Java EE 8 aduce unele noutăți, dintre care menționăm:

- Suport direct pentru WebSockets, prin noul element <f:websocket>
- validare de bean-uri la nivel de clasă prin elementul <f:validateWholeBean>

Java EE 7/8 utilizează specificația JSF 2.2/2.3 precum și Expression Language 3.0.

Pentru un sumar al noutăților din JSF 2.3, consultați link-ul - https://javaserverfaces.github.io/users.html

2.7.5 Enterprise Java Beans

O componentă EJB (numită și enterprise bean) este o clasă Java care are variabile și metode ce permit implementarea logicii aplicației (business logic). Acestei clase i se adaugă o interfață home, o interfață remote precum și un descriptor de desfășurare.

Există trei tipuri de EJBs:

- **session beans** – reprezintă o conversație (sesiune) cu un client. La terminarea execuției de către client, instanța bean-ului și datele pe care le conține sunt șterse
- **entity beans** – reprezintă date, date care de obicei sunt stocate ca articole în baza de date. Acest tip de EJB-uri sunt înlocuite recent cu **persistence entities**
- **message driven beans** – combină capabilitățile unui session bean și a unui message listener, permițând unei componente de business să primească mesaje de o manieră asincronă. Aceste mesaje sunt de obicei mesaje JMS (Java Message Service)

Principala noutate adusă de specificația 7 a Java EE în acest domeniu este reprezentată de operații ne-blocante de intrare/ieșire.

Java EE 7/8 utilizează specificațiile EJB 3.2 și Interceptors 1.2.

2.7.6 Java persistence API

API-ul pentru Java persistence oferă programatorilor Java o asociere între obiecte (clase Java) și entități relaționale (de obicei, articole din baze de date relaționale). Java persistence include următoarele arii:

- Java Persistence API
- Limbajul de interogare
- Java Persistence Criteria API
- Metadate de asociere obiect/relațional

O **entitate** este un obiect de persistență ușor. În principiu, o entitate reprezintă o tabelă dintr-o bază de date relațională, iar o instanță a unei asemenea entități este un articol din acea tabelă.

Caracterul persistent al unei entități este reprezentat prin câmpuri persistente sau proprietăți persistente.

Java EE 8 utilizează Java Persistence API 2.1.

2.7.7 Managed Beans

Un **managed bean** este o clasă Java înregistrată la o JSF. Aceste clase sunt administrate în cadrul JSF.

Tipic pentru aceste Java beans sunt metodele obligatorii de set și get pentru proprietățile bean-ului (variabilele) precum și implementarea unui mecanism de detecție a schimbărilor aduse acestor proprietăți.

Java EE 8 utilizează specificația Managed Beans 1.0.

2.7.8 Java Message Service - JMS

JMS este un API Java care permite aplicațiilor să creeze, trimită, primească și să citească mesaje. Acest API definește un set comun de interfețe și semanticile asociate care permit unui program scris în limbajul Java să comunice cu alte platforme de schimburi de mesaje.

JMS permite comunicare asincronă și de încredere:

- **asincronă** – clientul care primește mesajul nu trebuie să primească mesajul imediat după transmiterea acestuia. Totodată, clientul care trimite mesajul poate trece imediat la alte sarcini, imediat după trimitere.
- **de încredere** (reliable) – un generator de mesaje poate avea certitudinea că un mesaj este trimis exact o singură dată. Nivele mai reduse de încredere pot fi asigurate unor entități care își pot permite pierderea unora dintre mesaje

Versiunea curentă a JMS este 2.0.

2.7.9 Java Database Connectivity – JDBC

JDBC este un API al limbajului de programare Java care definește o interfață uniformă de accesare a a serviciilor de baze de date, indiferent de natura acestui serviciu.

Adaptarea apelurilor JDBC la un serviciu specific de date de baze se face prin intermediul unor conectori (drivere) JDBC.

API-ul JDBC oferă un mecanism de încărcare dinamică a pachetelor Java corespunzătoare și înregistrarea lor cu DriverManager-ul JDBC. Clasa DriverManager este apoi utilizată pentru realizarea de conexiuni cu serviciile de baze de date.

O conexiune JDBC permite crearea de obiecte de tip Statement, care emulează funcționalitatea interogărilor SQL. Obiectele de tip Statement pot fi apoi executate, rezultatul fiind, de regulă un set de articole din baza de date (ResultSet) sau o operație de modificare a conținutului bazei de date.

Articolele din obiectul de tip ResultSet pot fi parcurse și procesate individual prin intermediul unor metode specifice JDBC.

Un bridge JDBC-ODBC permite translatarea acestor apeluri către orice baza de date care este accesibilă prin intermediul ODBC (Open DataBase Connectivity).

2.7.10 Java EE Connector Architecture

Utilizată de firmele care oferă utilitare și de către integratorii de sisteme pentru a crea adaptori de resurse.

Java EE 8 utilizează specificația Java EE Connector Architecture 1.7.

2.7.11 Java Naming and Directory Interface - JNDI

JNDI este un API pentru serviciile de director care permite aplicațiilor scrise în Java să caute și să descopere diferite resurse (clase, fișiere, arhive, etc.) prin intermediul unui nume și a unor valori ale atributelor asociate.

Ca și celelalte API-uri, JNDI este independent de platforma pe care sunt executate aplicațiile ce folosesc facilitățile JNDI.

Principalele facilități oferite JNDI sunt:

- un mecanism care leagă un obiect de un nume (binding)
- o interfață de căutare care permite interogări flexibile
- o interfață pentru evenimente care permite detectarea modificărul obiectelor și atributelor lor
- extensii LDAP (Lightweight Directory Access protocol)

Interfața **SPI** (Service Provider Interface) permite conectarea cu diferite servicii de director

2.7.12 Java API for WebSocket

Acest API oferă suport pentru crearea de aplicații **WebSocket**. WebSocket este un protocol la nivel de aplicație care oferă comunicare full-duplex între două entități de nivel egal prin intermediul protocolului de transport TCP.

În modelul tradițional cerere-răspuns utilizat în HTTP, clientul trimite o cerere serverului care îi trimite un răspuns. Schimbul de mesaje este inițiat în toate cazurile de către client, serverul nu poate trimite nimic în absența unei cereri explicite.

Acest model este convenabil pentru Worlld Wide Web, atunci când resursele erau relativ statice și predictibile, dar limitările acestei abordări devin tot mai problematice în contextul în care conținutul se schimbă rapid și utilizatorii așteaptă o experiență mai interactivă pe web.

Protocolul WebSocket adresează aceste limitări oferind un canal de comunicare full-duplex în care comunicarea poate fi inițiată și continuată de către oricare din cele două părți.

Atunci când este combinat cu alte tehnologii client, precum JavaScript sau HTML5, WebSocket permite aplicațiilor web livrarea unui conținut bogat și dinamic.

Protocolul WebSocket este standardizat de către IETF (Internet Engineering Task Force) prin RFC (Request For Comments) 6455 din 2011.

Java EE 8 utilizează specificația Java API for WebSocket 1.0.

2.7.13 Java API pentru procesare JSON

JSON (JavaScript Object Notation) este un format bazat pe text pentru schimbul de date.

JSON este o alternativă mai prietenoasă la XML.

JSON definește doar două tipuri de date:

- obiecte – un set de perechi nume-valoare
- tablouri (arrays) – o listă de valori

JSON este utilizat în principal pentru serializarea și deserializarea datelor (obiectelor) în aplicații care comunică între ele peste internet. Aceste aplicații sunt create utilizând diferite limbaje de programare și sunt executate pe platforme diferite. În calitate de standard deschis, JSON este bine echipat pentru aceste scenarii, este ușor de scris și este mai compact decât alte standarde.

Java EE include suport pentru JSR 353, care oferă un API pentru parcurgerea, transformarea și interogarea datelor JSON utilizând unul din cele două modele – modelul obiect sau modelul șir (stream).

API-ul pentru procesare JSON conține două pachete: `javax.json` și `javax.json.stream`.

Java EE 8 utilizează specificația JSON-P 1.1.

2.8 Asamblarea și desfășurarea aplicațiilor Java EE

O aplicație Java este împachetată în una sau mai multe unități standard pentru desfășurare pe orice platformă Java EE. Fiecare unitate conține:

- una sau mai multe componente funcționale – precum servlete, EJB-uri, pagina JSP, etc.
- un descriptor de desfășurare (opțional) care descrie conținutul aplicației

Desfășurarea are loc de obicei prin invocarea unui set de unelte care este specific unei anumite platforme.

O aplicație Java EE este livrată sub forma unei arhive de tip EAR (Enterprise Archive), care este o simplă arhivă JAR dar cu extensia .ear.

Un fișier EAR conține module Java EE și descriptori de desfășurare. Un **descriptor de desfășurare** (deployment descriptor) este un document de tip XML care descrie modalitățile de desfășurare ale unei aplicații, modul sau componentă. Datorită caracterului său declarativ, conținutul uni descriptor de desfășurare poate fi modificat fără a fi necesară modificarea codului sursă.

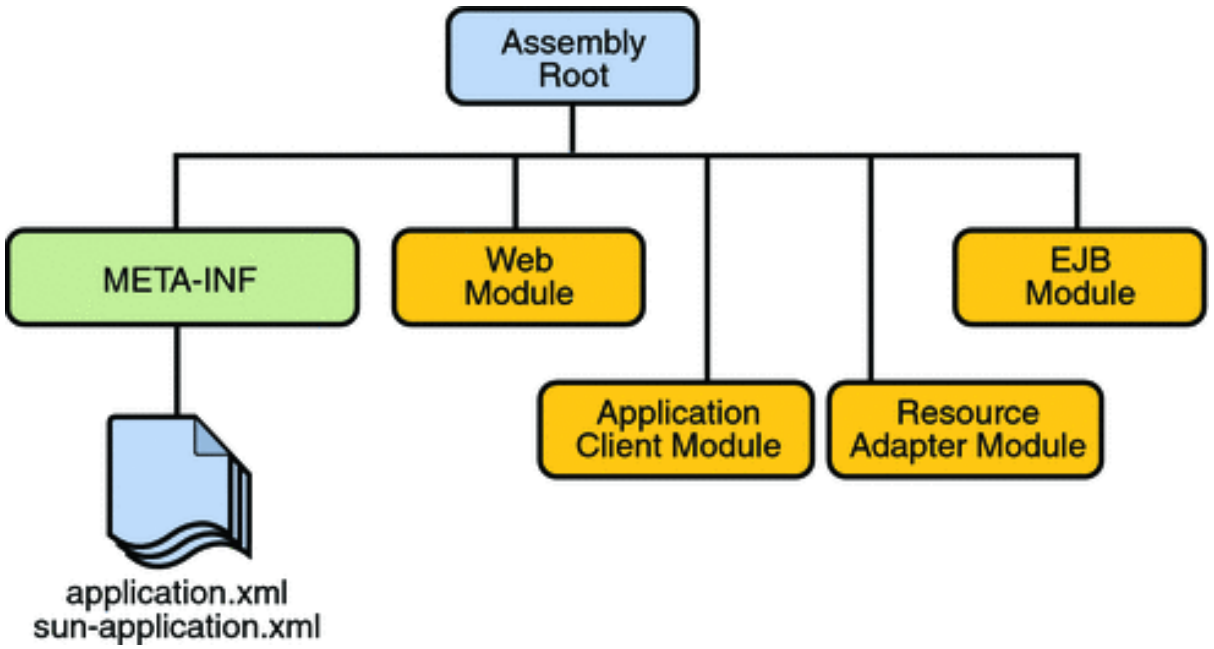


Figura 2.5 – Structura unui fișier EAR

chapter 2

Un modul Java EE constă din una sau mai multe componente care țin de același tip de container. Acesta poate avea sau nu un descriptor de desfășurare, în cel de-al doilea caz având de-a face cu un modul de sine stătător.

Există patru tipuri de module Java EE:

- **module EJB** – conțin fișiere de tip .class spentru EJB-uri
- **module web –** conțin servlete, pagini JSP, clase de suport, fișiere de tipHTML precum și un descriptor de desfășurare pentru aplicația web
- **module aplicație client** – care conțin fișiere de tip .class și un descriptor de desfășurare
- **module de adaptare a resurselor** – conțin interfețe și clase Java, precum și biblioteci native. Acestea implementează arhitectura Connector pentru un anumit EIS.

chapter 3 HyperText Transfer Protocol (HTTP)

3.1 ce este HTTP?

HTTP este un acronim pentru HyperText Transfer Protocol iar HyperText înseamnă text care conține link-uri către un alt text. Thenic vorbind, HTTP este un protocol de comunicare, la nivel de aplicație, de tip text, fiind protocolul implicit de comunicare pe www (World Wide Web).

Un mesaj care se conformează specificației HTTP are o linie inițială, urmată (opțional sau nu) de o zonă de header-e precum și (opțional) de un conținut al mesajului – de obicei un fișier de tip html sau alta resursă care constituie obiectul schimbului de mesaje.

3.2 istoric și versiuni

HTTP a fost creat de către Tim Berners Lee în anul 1990 la CERN (Conseil Européen pour la Recherche Nucléaire)(nume curent – European Organization for Nuclear Research) ca modalitate de schimb și de stocare a informațiilor științifice.

Prima versiune oficială – HTTP 1.0 – dateaza din 05.1995 și constituie obiectul RFC 1945 (https://tools.ietf.org/html/rfc1945). RFC înseamnă Request For Comments și desemnează un tip de publicații controlat de către IETF (Internet Engineering Task Force) și de către ISOC (the Internet Society).

RFC 1945 are ca autori pe Tim Berners-Lee, Roy Fielding și Henrik Nielsen.

A doua versiune, și anume HTTP 1.1 a constituit obiectul mai multor RFC-uri, dintre care menționăm RFC 2068 (01/97), RFC 2616 (06/99), RFC 2617 (06/99) și RFC 2774 (02/00).

Pentru o listă completă a versiunilor specificațiilor HTTP precum și a conținutului acestora, puteți consulta site-ul ofocial al HTTP - www.w3.org/Protocols. Pentrua înțelege modul îl care HTTP operează, sugerăm un alt site - www.jmarshall.com/easy/http.

Versiunea curentă (deși nu și cea mai utilizată) a protocolului HTTP este HTTP 2.0 (sau HTTP/2), care a fost publicată ca RFC 7540 în mai 2015 – https://tools.ietf.org/html/rfc7540. Lucrul la această specificație a început în 2012, când a fost lansat un Call for proposals pentru HTTP 2.0.

Conform cu W3Techs, în spetembrie 2018 30.0% din cele mai importante 10 milioane de site-uri ofertă suport pentru HTTP/2. În mai 2017, nivelul de suport era de doar 13.7%.

Grupul de lucru pentru HTTP/2 a oferit soluții pentru următoarele scopuri.

- Un mecanism de negociere care să permită clienților și serverelor să poată alege versiunea dorită a protocolului HTTP sau să utilizeze un alt protocol.
- Menținerea unui înalt nivel de similaritate cu HTTP 1.1, cel puțin în materie de metode, coduri de stare, câmpuri de header precum și URI-uri.
- Reducerea timpilor de așteptare și îmbunătățirea timpilor de încărcare în browsere prin:
  - Compresia datelor din hader-ele HTTP
  - Utilizarea tehnologiilor Server Push – serverul trimite resurse înainte de a-i fi cerute
  - Rezolvarea problemei head-of-line blocking din HTTP 1
  - Încărcarea elementelor unei pagini în paralel utilizând o singură conexiune TCP
- Suport pentru cazurile de utilizare comune din HTTP, precum browsere pentru desktop-uri, browsere pentru dispozitive mobile, web API, servere web, servere proxy, firewalls precum și pentru rețele de livrare a conținutului.

3.3 Structura tranzacțiilor HTTP - cerere

Protocolul HTTP implementează modelul de comunicare client-server. Clientul trimite o cerere (request) către server și acesta răspunde, de regulă, cu un mejaj de tip răspuns (respone). Aceste două tipuri de mesaje pot fi diferite în materie de conținut, dar au câteva elemente structurale în comun, după cum urmează:

1. O linie inițială
2. zero sau mai multe linii de header
3. o linie goală (CR/LF)
4. un corp al mesajului (opțional)

Prezentăm mai jos un mesaj tipic de cerere HTTP:

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/61.0.3163.100 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/a
png,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8,ro;q=0.6
```

Vom trece în revistă structura unei cerei HTTP, plecând de la acest exemplu.

3.3.1 Linia inițială a unei cereri HTTP

Conține 3 elemente:

- un nume de comandă (metodă) HTTP – precum `GET`, `POST`, `HEAD`, etc.
- o specificație de fișier – partea din URL de după numele serverului web – în cazul nostru, doar „/”
- versiune HTTP utilizată – în majoritatea cazurilor – `HTTP/1.1`

Un exemplu mai generic: `GET /path/to/the/file/index.html HTTP/1.1`

3.3.2 Liniile de header

O linie de header (în cazul nostru sunt 8 asemenea headere) conține un nume de header (parametru) – precum `Host`, `Accept`, `Connection` sau `Accept-Language`, urmat de „:” și de valoarea (valorile) header-ului (parametrului), separate prin virgulă sau spații.

În specificația HTTP/1.1 sunt precizate 46 de headere, dintre care unul singur – `Host` – este obligatoriu.

3.3.3 O linie goală

Aceasta este vizibilă și în exemplul nostru, chiar dacă nu este urmată de un corp al mesajului HTTP.

3.3.4 Corpul mesajului

Opțional, mai multe despre acesta în unul din paragrafele următoare.

3.4 Comenzi HTTP

În variantele ce au dus la definirea HTTP/1.1 sunt precizate 14 comenzi, dintre care 8 se bucură de suport din partea tuturor browserelor majore. Acestea sunt:

1. GET
2. HEAD
3. POST
4. CONNECT
5. DELETE
6. OPTIONS
7. PUT
8. TRACE

Celelalte 6 comenzi apărute în diferite specificații, și anume:

- COPY
- LINK
- MOVE
- PATCH
- UNLINK
- WRAPPED

nu sunt prea des utilizate, datorită funcționalității lor exotice și lipsei de suport generalizat.

3.5 Metodele GET și POST

Cele mai utilizate metode HTTP sunt GET și POST. Prin natura sa, protocolul HTTP urmează paradigma cerere-răspuns. Ambele metode GET și POST pot fi utilizate pentru a transmite o cerere din partea clientului către server; cu toate acestea, cele două metode au destule particularități care le diferențiază.

GET – cere date (răspuns) din partea unei resurse specificate

POST – trimite date care urmează a fi procesate de o anumită resursă

Într-un scenariu tipic, datele (parametrii) unei forme HTTP sunt trimiși unei resurse specificate în atributul ACTION al formei.

În cazul invocării metodei GET, perechile nume_parametru=valoare_parametru sunt conținute în URL-ul cererii – de exemplu:

```
/ccards/registration_form.php?nume=Popescu&prenume=Vasile
```

În cazul invocării metodei POST, perechile nume_parametru=valoare_parametru sunt conținute în corpul cererii, care este de obicei codificat, permițând protejarea unor date senzitive.

Alte diferențe: cererile GET au restricții de lungime, cele POST nu, cererile GET pot fi reținute în cache sau în bookmarks, nu se poate face acest lucru cu cererile POST.

Pentru detalii asupra diferențelor dintre cele două metode, consultați - https://www.w3schools.com/TAgs/ref_httpmethods.asp

3.6 Header HTTP

O linie de header constă din două părți: numele header-ului și valoarea sa, separate prin două puncte „:”. Versiunea HTTP 1.0 precizează 16 headere, niciunul nefiind obligatoriu, în timp ce versiune HTTP 1.1 specifică 46, dintre care unul singur – host – este obligatoriu.

HTTP/2 nu aduce headere noi, dar maniera de prezentare este complet diferită, utilzând metoda HPACK de codificare (compresie) a headerelor.

Numele header-ului nu este case sensitive, dar valoarea/valorile sale sunt. De obicei, perechea header: valoare_header se scrie pe un singur rând, dar dacă o linie începe cu spații sau tab-uri atunci se consideră a fi o continuare a liniei precedente.

Două exemple:

```
User-agent: Mozilla/3.0Gold
```

```
Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT
```

3.7 Corpul unui mesaj – tipuri MIME

Corpul unui mesaj HTTP este opțional, iar atunci când este prezent conține informații trimise de către client sau de către server. Dacă acesta este prezent, trebuie precizate valorile a două headere:

- Content-Type – precizează tipul MIME al corpului mesajului
- Content-Length – precizează lungimea în bytes a corpului mesajului

MIME - Multipurpose Internet Mail Extensions – este un standard de Internet care precizează extensiile (tipurile) email-urilor precum al atașamentelor asociate – audio, video, imagini, aplicații, text, etc.

Deși definite inițial pentru SMTP – Simple Mail Transfer Protocol – tipurile MIME sunt utilizate și de unele protocoale de comunicații, precum HTTP. Serverele inserează tipul MIME ca header iar clienții selectează programul de redare a conținutului în baza valorii acestui header.

Tipurile MIME sunt precizate într-o suită de RFC-uri, și anume: RFC 2045, RFC 2046, RFC 2047, RFC 4288, RFC 4289 și RFC 2049. Tipurile MIME inițiale se refereau la protocolul SMTP și au fost definite în RFC 1521 și RFC 1522.

O extensie MIME are două părți – un tip și un sub-tip: de exemplu: text/html, image/jpeg, video/mp4, etc.

Pentru a avea o idee mai precisă asupra acestor tipuri, reamintim cele 7 tipuri definite în RFC 1521 precum și unele din subtipurile asociate.

1. **text**, cu sub-tipul plain
2. **multipart**, cu sub-tipurile mixed, alternative, digest, parallel
3. **message**, cu sub-tipurile rfc822, partial, external-body
4. **application**, cu sub-tipurile octet-stream, postscript
5. **image**, cu sub-tipurile jpeg, gif
6. **audio**, cu sub-tipul basic
7. **video**, cu sub-tipul mpeg

3.8 Structura tranzacțiilor HTTP - răspuns

Structura unui răspuns HTTP este identică, în general, cu structura cererii HTTP. Diferența apare în conținutul liniei inițiale a răspunsului HTTP, care are următoarea configurație:

- **versiunea** HTTP a răspunsului

- un **cod de stare** a răspunsului (o valoare numerică)
- un **enunț** (șir de caractere) care explicitează codul de stare

Iată mai jos un exemplu de linie inițială a unui răspuns HTTP:

```
HTTP/1.1 404 Not Found
```

3.9 Coduri de stare

Codul de stare conținut într-un răspuns HTTP este un întreg cu trei cifre, dintre care prima cifră desemnează categoria răspunsului:

- **1xx** indică un mesaj de tip informațional
- **2xx** indică succes de o formă sau alta
- **3xx** specifică redirectare
- **4xx** indică o eroare din partea clientului
- **5xx** indică o eroare din partea serverului

Cele mai comune coduri de stare sunt:

- **200 OK** – cererea a reușit iar resursa solicitată (de exemplu – un fișier html sau un script) este returnată în corpul răspunsului.
- **404 Not Found** – resursa solicitată nu există
- **301 Moved Permanently –** mutat permanent
- **302 Moved Temporarily** – mutat temporar
- **303 See Other** (doar în HTTP 1.1) - resursa a fost mutată la un alt URL, specificat în header-ul **Location** și poate fi regăsită automat de către client. Utilizat de obicei de către un script CGI pentru redirectarea browser-ului către un fișier existent.
- **500 Server Error** - o eroare neașteptată din partea serverului. De obicei, cauzată de o eroare într-un script – eroare de sintaxă, resursă neexistentă sau altă condiție imputabilă serverului.

O listă completă a codurilor de stare poate fi găsită în specificația HTTP/1.0 (paragraful 9) sau în paragraful 10 al specificației HTTP/1.1.

3.10 Exemplu de tranzacție HTTP

Pentru a afișa conținutul fișierului precizat de URL-ul http://web.info.uvt.ro/path/file.html, clientul deschide un socket pentru comunicarea cu serviciul web al serverului domeniului **web.info.uvt.ro**, la portul 80 (portul implicit pentru protoolul HTTP). Cererea HTTP este creată și trimisă prin intermediul socket-ului:

```
GET /path/file.html HTTP/1.1
From: someuser@yahoo.com
User-Agent: HTTPTool/1.1
[o linie goală aici]
```

Răspunsul serverului poate fi unul ca cel listat mai jos și este trimis la același socket al clientului:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354
```

```
<html>
<body>
<h1>Hello valued customer!</h1>
(more file contents)
.
.
</body>
</html>
```

După trimiterea răspunsului, serverul poate închide socket-ul..

3.11 Noua versiune – HTTP 2.0

Noutățile aduse de noua versiune HTTP/2 (sau HTTP 2.0) țin nu atît de conținutul mesajelor HTTP ci de modalitatea de formatare și livrare a mesajelor. Un sumar al acestor noutăți.

HTTP/2 oferă transport optimizat pentru mesajele HTTP. HTTP/2 oferă suport pentru facilitățile principale ale protocolului HTTP/1.1 dar încearcă să fie mai eficient în mai multe privințe.

Unitatea de bază a protocolului HTTP/2 este **frame**-ul. Fiecare frame servește unor scopuri predefinite. De exemplu, frame-urile de tip HEADERS sau DATA formează fundamentul schimbului de cereri și răspunsuri HTTP; pe de altă parte, frame-uri precum SETTINGS, WINDOW_UPDATE sau PUSH_PROMISE sunt utilizate ca suport pentru alte facilități HTTP/2.

Multiplexarea cererilor este realizată prin atribuirea unui stream separat fiecărui schimb de cerere/răspuns HTTP. Stream-urile sunt independente, de aceea blocarea sau întârzierea unuia dintre ele nu le afectează pe celelalte.

Controlul fluxului și mecanismele de **prioritizare** asigură posibilitatea utilizării eficiente a stream-urilor multiplexate. Controlul fluxului asigură transmiterea către destinatar doar a volumului de date ce poate fi procesat de către acesta.

Push responses (răspunsuri predictive) – acest mecanism permite trimiterew anticipată a datelor de către server în baza unei predicții care poate fi confirmată sau nu. Acest lucru este realizat prin crearea de către server a unei cereri fictive, care este trimisă ca un frame de tip PUSH_PROMISE. Ulterior, serverul trimite pe un stream diferit un răspuns la această cerere fictivă.

Compresia header-elor. Deoarece câmpurile de header conțin deseori o mare cantitate de informații redundante, frame-urile care le conțin sun comprimate (codificate). Acest lucru este avantajos în cazul cererilor multiple, permițând trimiterea frame-urilor comprimate într-un singur pachet.

chapter 4 HTML, CSS și altele

4.1 despre HTML

HTML – Hyper Text Markup Language - este un limbaj de marcare (markup) care este utilizat pentru crearea de pagini ce pot fi încărcate și vizualizate într-un browser web. HTML este un limbaj de prezentare a informațiilor (fonturi, culori, paragrafe, diviziuni) și mai puțin un limbaj de descriere a semanticii informațiilor.

Specificațiile acestui limbaj sunt produse de către consorțiul World Wide Web (sau W3C).

Prima versiune a HTML-ului (HTML 1.0) a apărut în vara lui 1991 și avea suport din partea primului browser cu succes la public – Mosaic. Prima versiune oficială – HTML 2.0 a fost admisă ca standard în septembrie 1995 ca RFC (Request For Comments) 1866 și a avut suport larg.

Următorul standard (HTML 3.0) nu a a vut prea mult succes, dar versiunea minoră HTML 3.2, apărută în ianuarie 1997, a normalizat lucrurile în materie de acceptare și suport.

Versiunea 4.0 introduce Cascade Style Sheets (CSS). O revizuire a versiunii 4.0, denumită 4.01 a fost acceptată în decembrie 1997. Aceasta este ultim versiune a HTML ca dialect SGML.

Din anul 1999, HTML este parte a unei noi specificații – XHTML.

În ianuarie 2008 a fost publicat un document de lucru (working draft) pentru o nouă versiune – HTML5. Specificația sa a fost finalizată în octombrie 2014.

4.2 definiția limbajului HTML

Din punct de vedere formal, HTML este o versiune specializată a SGML (Standard Generalized Markup Language – un standard ISO (ISO 8879)). Toate limbajele de markup definite în cadrul SGML sunt denumite **aplicații SGML**. Acest lucru nu mai este însă valabil pentru HTML5, care va fi descris ceva mai târziu.

Acestea, în cazul HTML 4(.01), sunt caracterizate prin:

1. o declarație SGML – specifică ce caractere și delimitatori sunt specifice limbajului. Aceasta poate fi văzuta la acest link - https://www.w3.org/TR/html401/sgml/sgmldecl.html
2. trei fișiere de tip DTD (Document Type Definition), și anume:
   - HTML 4.01 Strict DTD – conține toate elementele și atributele curente
   - HTML 4.01 Transitional DTD – include toate elementele și atributele din Strict DTD plus cele deprecate
   - HTML 4.01 Frameset DTD – conține în plus și frames
3. referințe de caractere (character references) – dacă sunteți curioși ce vrea să însemne acest lucru, câteva exemple - “&lt”, “&quot”, "水" (în hexazecimal) – ideograma chineză pentru apă.
4. exemple (instances) de documente ce conțin date și markup. Fiecare din acestea conține o referință la DTD-ul utilizat pentru interpretare.

Rezumând, specificația majoră HTML 4 conține o declarație SGML, trei DTD-uri și o listă de referințe de caractere.

4.3 elemente HTML

Elementele din HTML 4 pot fi împărțite în 10 categorii ad-hoc:

- elemente top-level – HTML, HEAD, BODY, FRAMESET

- elemente head – 7 în total, gen BASE, SCRIPT, STYLE
- elemente generice la nivel de block – 17 în total, precum – DIV, H2, P
- elemete de tip listă – 9 în total, precum – DL, LI, UL
- pentru tabele – 10 elemente, precum – COL, TR, TD
- pentru forme – 10 elemente, precum – FORM, SELECT, INPUT
- elemente speciale inline – 17 în total, precum – A, BR, IMG, SPAN
- elemente de formatare – 12 în total, precum – ABBR, CODE, STRONG, VAR
- elemente de stil pentru fonturi – 8 în total, precum – SMALL, STRIKE, U
- elemente tip frame – FRAMESET, FRAME, NOFRAMES

În total, specificația HTML 4 conține 91 de elemente. (unele elemente figurează în mai multe categorii).

Lista completă poate fi văzută la - https://www.w3.org/TR/1999/REC-html401-19991224/index/elements.html

4.4 HTML5

HTML5 e versiunea curentă a specifiicației HTML.

În decembrie 2012 specificația trece în starea candidat la recomandare a consorțiului W3C (World Wide Web Consortium).

Din start, HTML se dorește a subsuma nu doar capabilitățile HTML 4, ci și cele ale XHTML 1 precum și ale specificației HTML corespunzătoare nivelui 2 al DOM (Document Object Model). (DOM level 2 HTML).

Specificația HTML5 a fost finalizată în octombrie 2014 – https://www.w3.org/TR/html5/.

Aceasta include modele detaliate de procesare pentru a încuraja implementări interoperabile. Extinde, îmbunătățește și și raționalizează elementele de markup disponibile pentru documente și introduce un API pentru aplicații web complexe.

4.4.1 elemente noi

HTML5 introduce elemente noi în diferite categorii.

- elemente structurale/semantice (22 la număr), precum - <article>, <nav>, <menuitem>, <nav>, <progress>
- elemente pentru forme (3 la număr) - <datalist>, <keygen>, <output>
- noi tipuri de input (precum color, datetime, range, tel, url, week) și noi atribute pentru elementul input (precum autocomplete, list, min and max, pattern (regexp), placeholder, required)
- elemente de grafică - <canvas>, <svg>
- elemente media - <audio>, <embed>, <source>, <track>, <video>

Pentru o listă exhaustivă cu explicații, consultați - http://www.w3schools.com/html/html5_new_elements.asp

4.4.2 elementul <form>

Elementul HTML <form> definește o formă care este utilizată pentru colectarea, procesarea (indirectă) și transmiterea (indirectă) a introducerilor utilizatorului.

<form> are două atribute obligatorii:

- method – precizează modalitatea (comanda HTTP) utilizată pentru a trimite datele și poate avea valorile POST sau GET

- action – specifică URL-ul resursei care urmează să proceseze datele din formă, atunci când forma este transmisă (submitted)

4.4.3 elemente asociate unei forme

Elementele (definite în HTML 4) asociate unei forme HTML sunt:

- <input> -
- <select> -
- <option> -
- <textarea> -
- <button> -

Elementele noi, introduse în HTML5, sunt:

- <datalist> -
- <keygen> -
- <output> -

4.4.4 tipuri de input

Tipurile de input (valorile posibile ale atributului type ale elementului <input>), definite în HTML 4 sunt:

- text
- password
- submit
- reset
- radio
- checkbox
- button
- image
- hidden
- file

HTML5 introduce câteva tipuri noi de input, și anume: color, date, datetime, datetime-local, email, month, number, range, search, tel, time, url, week.

4.4.5 Atributele elementului <input>

Atributele elementului <input> din HTML 4 erau:

- readonly
- disabled
- size
- maxlength

HTML5 avea să adauge o întreagă listă la cele existente, și anume: autocomplete, autofocus, form, formaction, formenctype, formmethod, formnovalidate, formtarget, height and width, list, min and max, multiple, pattern (regexp), placeholder, required, step.

Elementul <form> primește și el două noi atribute în această versiune: autocomplete și novalidate.

Detalii despre aceste atribute pot fi găsite la - http://www.w3schools.com/html/html_form_attributes.asp .

4.5 CSS

CSS sau Cascade Style Sheet este un standard pentru formatarea documentelor HTML. Acest lucru se face prin intermediul fișierelor de tip .css externe sau al codului CSS introdus prin elementul <style> și/sau atributul style al unui alt element.

Un set de reguli CSS constă dintr-un selector și un bloc de declarații.

4.5.1 selectori CSS

Selectorii CSS selectează elementele HTML în funcție de numele, id-ul, clasa unui element HTML, atribut, etc.

CSS are următoarele tipuri de selectori:

- element – numele elementului
- id-ul elementului
- clasa – selectează un element cu un atribut de clasa specific.

4.5.2 comentarii CSS

Comentariile sunt delimitate de /* … */

4.6 LESS

LESS este un limbaj de stilizare dinamic care poate fi compilat în CSS și executat pe aplicațiile client sau de server. Este open source, prima sa versiune fiind scrisă în Ruby iar versiunile ulterioare într-un limbaj mai răspândit, JavaScript. Ca limbaj, a fost influențat de către Sass, fiind la rândul său o influență pentru noua varietate a Sass-ului, numită SCSS.

LESS oferă următoarele facilități programatice:

- variabilele
- nesting
- mixins
- operatori
- funcții

4.6.1 variabile

Variabilele sunt declarate cu semnul (@) urmat de numele variabilei. Operatorul de atribuire este (:) iar instrucțiunile de atribuire sunt terminate cu punct și virgula (;). Un exemplu:

```
@my-blue: #5C13AD;
@your-blue: @my-blue + #111;
#header {
   color: @your-blue;
}
```

4.6.2 mixins

Permite propagarea tuturor proprietăților unei clase într-o clasă prin includerea numelui clasei ca fiind una din proprietăți, comportându-se ca o clasă sau o variabilă.

CSS nu oferă această facilitate. Această facilitate permite cod mai eficient și mai ușor de menținut.

4.6.3 nesting (imbricare)

CSS oferă suport pentru imbricare logică, dar blocurile de cod înseși nu pot fi imbricate. LESS pemite imbricarea selectorilor în alți selectori.

4.6.4 operații

În materie de operații, avem – adunare, scădere, înmulțire și împărțire. Aplicabile, în special, culorilor și valorilor numerice.

4.6.5 funcții

Funcțiile au un format similar cu cel din JavaScript.

4.6.6 exemplu de cod LESS

Exemplu din - https://codemyviews.com/blog/10-less-css-examples-you-should-steal-for-your-projects

```
/* Mixin */
.border-radius (@radius: 5px) {
   -webkit-border-radius: @radius;
   -moz-border-radius: @radius;
   border-radius: @radius;
}

/* Implementation */
#somediv {
   .border-radius(20px);
}
```

După compilare, codul CSS arată ca mai jos:

```
/* Compiled CSS */
#somediv {
   -webkit-border-radius: 20px;
   -moz-border-radius: 20px;
   border-radius: 20px;
}
```

4.6.7 utilizare LESS

Există mai multe modalități de utilizare a codului LESS într-un site. Iată două dintre ele:

- includerea scriptului less.js care permite conversia în CSS on-the-fly
- conversia a-priori a codului LESS în CSS și încărcarea acestuia pe site

Prima variantă nu e și cea mai eficientă pentru producție, fișierul less.js având peste 140 KB.

Deoarece LESS nu este un standard (W3C sau alt tip), browser-ele nu oferă suport nativ pentru LESS (sau alte preprocesoare CSS, precum Sass sau SCSS).

Abordarea entităților de standardizare este de a introduce în standarde (CSS, în special) acele facilități care își dovedesc utilitatea prin intermediul LESS (SCSS).

4.7 Ruby on rails

4.7.1 Limbajul ruby

Limbajul Ruby este un limbaj de programare orientat obiect, creat în 1993 de către japonezul Yukihiro Matsumoto. Este open source, gratuit.

Este un limbaj de scriptare interpretat, similar în multe privințe cu limbaje de scriptare precum Smalltalk, Perl sau Python. Din punct de vedere al localizării, Ruby este un limbaj pentru server-side, putând fi utilizat pentru crearea de scripturi CGI (Common Gateway Interface).

Elementele principale de sintaxă sunt similare cu cele din C++ sau Perl.

În materie de utilizare, Ruby este folosit, în mare măsură, pentru aplicații Internet și intranet. Poate fi conecta cu ușurință la servicii de baze de date, precum DB2, MySQL, Oracle sau Sybase.

Ruby este situat în Top 20 în TIOBE Index (poziția 12, mai exact, în sept. 2018) în materie de popularitate în rândul limbajelor de programare - https://www.tiobe.com/tiobe-index/ . Top 5, dacă sunteți curioși, este fromat de Java, C, Python, C++ și Visual Basic .NET.

Versiunea curentă a limbajului este 2.5.1. Următoarea versiune – 2.6.0 – este în stare de previzionare.

Site-ul oficial este - https://www.ruby-lang.org/en/ .

4.7.2 mediul Rails

Rails este un cadru (platformă, framework) pentru dezvolatarea aplicațiilor web, cadru scris în Ruby și orientat spre dezvoltare aplicațiilor în limbajul Ruby.

Ruby on Rails a fost creat de către David Hansson din proiectul său Basecamp, o unealtă destinată administrării proiectelor. Rails a devenit open source în iulie 2004, fiind deschis pentru contribuții exterioare din februarie 2005.

Precum multe alte platforme de dezvoltare web, Rails folosește arhitectura Model-View-Controller (MVC) pentru structurarea procesului de programare a aplicațiilor.

Ruby on Rails conține mai multe pachete, și anume:

- ActiveRecord – facilitează accesul la sisteme de baze de date
- ActiveResource – servicii web
- ActionPack
- ActiveSupport
- ActionMailer

Totodată, Ruby on Rails conține elemente de facilitare a dezvoltării, prin crearea de schelete de aplicații, un server web integrat (WEBrick) precum și un sistem de construire și desfășurare a aplicațiilor – Rake.

Pentru execuția aplicațiilor e nevoie de un server web, pe lângă soluția standard (Mongrel, ulterior Passenger), pot fi utilizate și servere web clasice, precum Lighthttpd, Abyss sau Apache.

4.7.3 Link-uri utile

- https://www.railstutorial.org/book/frontmatter
- https://www.codecademy.com/learn/learn-rails
- http://railsforzombies.org/

4.8 Bootstrap

4.8.1 despre

Booststrap este un cadru (framework) pentru dezvoltarea front-end-ului aplicațiilor web. Inițial, a fost un proiect intern al companiei Twitter, pentru a asigura coerența internă a platformei.

Așa-numitul „Twitter Blueprint for Bootstrap” a fost făcut public ca proiect open source în august 2011.

Link-uri utile:

- https://www.tutorialspoint.com/bootstrap/index.htm
- https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)
- http://www.w3schools.com/bootstrap/default.asp
- https://www.taniarascia.com/what-is-bootstrap-and-how-do-i-use-it/

Bootstrap include șabloane HTML și CSS pentru elemente tipografice, fonturi, butoane, tabele, navigație, carusele de imagini, etc., precum și scripturi JavaScript (opționale).

Principalele carcteristici:

- ușor de utilizat
- flexibil (responsive) – CSS-ul flexibil se ajustează la telefoane, tablete sau desktop-uri
- abordare mobile-first

4.8.2 cum funcționează

Bootstrap se bazează pe 3 fișiere principale:

- bootstrap.css – un cadru (framework) CSS
- bootstrap.js – un cadru (framework) JavaScript/jQuery
- glyphicons – un set de fonturi (icon font set)

Totodată, Bootstrap are nevoie de biblioteca jQuery pentru a funcționa.

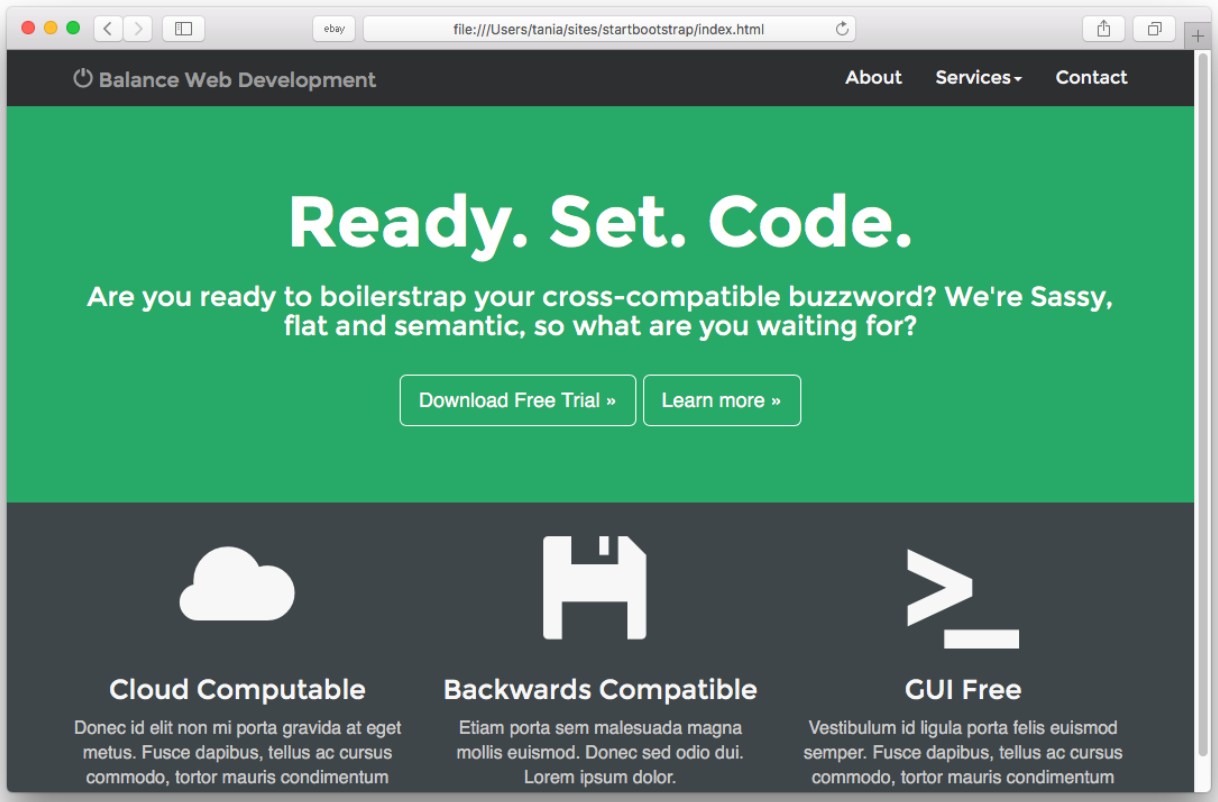
4.8.3 lucruri la care ajută

- reduce reundanța proiectelor
- folosește un design flexibil pentru adaptarea la diferitele mărimi ale ecranelor
- adaugă consistență la stilul site-urilor
- permite crearea rapidă și ușoară a noi variante de design
- asigură compatibilitate cu browserele principale

4.8.4 un exemplu

Luat de pe blogul - https://www.taniarascia.com/what-is-bootstrap-and-how-do-i-use-it/

Mai jos e rezultatul final al mini-tutorialului de pe blog.



4.9 sisteme de administrare a conținutului - CMS

Un sistem de administrare a conținutului (CMS – Content Management System) este „un sistem software creat pentru automatizarea cât mai deplină a gestiunii conținutului, în special a site-urilor web. Scopul este de a reduce sau elimina intervenția programatorilor la editarea și administrarea site-urilor lor. CMS-ul facilitează organizarea, controlul și publicarea de documente sau alt tip de conținut, cum ar fi imagini și resurse multimedia. Un CMS facilitează adesea crearea in comun de documente. Un "CMS web" este un CMS cu facilități adiționale pentru ușurarea publicării de conținut pe diversele site-uri.” - https://ro.wikipedia.org/wiki/Sistem_de_administrare_a_con%C8%9Binutului

Cele mai răspândite CMS-uri sunt, în ordinea complexității: wordpress, drupal, joomla.

Aceste CMS-uri oferă facilități de dezvoltare front-end (în special, sub forma unor șabloane predefinite) precum și de administrare – back-end.

4.9.1 wordpress

Wordpress - https://wordpress.org/ - este un CMS gratuit și open source, bazat pe Php și MySQL. Prima sa versiune a apărut în mai 2003. Versiunea curentă este 4.7.

Ultimile date statistice (din dec. 2017) atribuie wordpress 60% din totalul de site-uri cu CMS cunoscut,

reprezentând 29.2% din toate site-urile.

Wordpress are un sistem de șabloane (themes) și utilizează un procesor de șabloane. Utilizatorii wordpress pot instala și comuta între diferite șabloane (teme).

Totodată, arhitectura plugin a wordpress-ului permite extinderea funcționalității cu plugin-uri. La această oră sunt peste 40000 de asemenea plugin-uri disponibile.

4.9.2 Joomla!

Joomla – www.joomla.org - este „un sistem de management al conținutului Open Source, scris în PHP, destinat publicării de conținut pe inter și intra net prin intermediul bazelor de date SQL. Joomla! include funcționalități precum cache-ingul paginilor pentru îmbunătățirea performanțelor, RSS, opțiune de printare a paginilor web, știri de ultimă oră, bloguri, sondaje, căutare web și localizare internațională.

Numele reprezintă transcrierea fonetică al cuvântului din limba Swahili "jumla", însemnând "împreună" sau "tot unitar". A fost ales pentru a reprezenta nivelul de angajament al echipei de dezvoltare și a comunității, vis-a-vis de proiect. Prima versiune de Joomla! (Joomla! 1.0.0) a fost anunțată pe 16 septembrie 2005. Aceasta a fost o lansare Mambo 4.5.2.3 sub un alt brand, combinat cu unele îmbunătățiri de securitate și performanță” - https://ro.wikipedia.org/wiki/Joomla!

Versiunea curentă este Joomla 4.0 (sept. 2018)

Crearea site-urilor Joomla începe (de obicei) prin instalarea unui șablon (template) Joomla, care formează scheletul funcțional al noului site.

Pe lângă șablonul inițial, funcționalitatea site-ului poate fi extinsă cu extensiile Joomla, de următoarele tipuri:

- componente
- pugin-uri
- module
- suport pentru alte limbi

Cota de piață a Joomla este de aproximativ 5.8%, repectiv 3.1% din totalul site-urilor web, datorită complexității sale. Un șablon cu suport pentru Joomla 3.X poate avea peste 10000 de fișiere.

4.9.3 drupal

Drupal – www.drupal.org – este „un sistem modular open source de gestionare a conținutului, un cadru de dezvoltare pentru aplicații web și motor de blogging. A fost fondat și dezvoltat de Dries Buytaert. După anul 2000 Drupal a câștigat în popularitate mulțumită flexibilității, adaptabilității, ușurinței în administrare și exploatare, precum și a unei comunități online foarte active.

Drupal este scris în limbajul PHP, însă instalarea, dezvoltarea și întreținerea unui site web Drupal nu necesită (de obicei) cunoștințe de programare PHP.

Drupal funcționează pe o platformă variată de sisteme de operare cum ar fi Unix, Linux,BSD, Solaris, Windows, sau Mac OS X. În afara PHP, Drupal are nevoie pentru a funcționa și de un server de web cum este Apache sau IIS precum și de un motor de baze de date cum este MySQL.” - https://ro.wikipedia.org/wiki/Drupal .

Versiunea curentă este 8.6.1. (sept. 2018) Versiunea 9.0 este în lucru. În ianuarie 2015 a fost creată și o derivație, numită Backdrop.

Cota de piață a Drupal-ului este de aproximativ 3.7%, resectiv 2.0% din totalul site-urilor web. Acest număr este suficient pentru poziția a 3-a în calsamentul CMS-urilor.

chapter 5 javascript

5.1 ce este JavaScript?

JavaScript este un limbaj de scriptare care oferă elemente de interactivitate paginilor HTML.

- Un limbaj de scriptare este un limbaj de programare de categorie ușoară
- O sursă JavaScript constă din linii de cod executabile
- Codul JavaScript este de obicei inclus direct într-o pagină HTML
- JavaScript este un limbaj interpretat (această înseamnă că scripturile pot fi executate fără compilare prealabilă)

Numele oficial inițial al acestui limbaj este ECMAScript. ECMA înseamnă - European Computer Manufacturers Association – desemnează o organizație înființată în 1961 și al cărei scop era standardizarea sistemelor de calcul din Europa.

Limbajul însuși a apărut în 1995, fiind dezvoltat inițial de către Brendan Eich de la compania Netscape, sub numele de Mocha, nume schimbat ulterior în LiveScript, apoi în JavaScript.

JavaScript a fost standardizat de către ECMA în iunie 1997 sub numele de ECMAScript. Cu toate acestea, publicul larg îl cunoaște sub numele dat de către creatorul său – JavaScript.

Standardul ECMA a fost adaptat ulterior la alte limbaje, precum QtScript sau ActionScript.

Prima versiune standardizată a limbajului ECMAScript este versiunea 1.0 care datează din 03.1996, urmată de versiunea 1.1 din 08.1996, ..., versiunea 1.5 (numită v 5, cu cel mai bun suport din partea browser-elor) din 11.2000 și, în final, ultima versiune menționată este versiunea 9, cunoscută ca și ECMAScript 2018.

5.2 ce poate face JavaScript?

- **JavaScript oferă dezvoltatorilor HTML o unealtă de programare** – autorii HTML nu sunt în mod normal programatori, dar JavaScript este un limbaj de scriptare cu o sintaxă foarte simplă
- **JavaScript poate pune text dinamic într-o pagină HTML** – o instrucțiune JavaScript precum: document.write("<h1>" + name + "</h1>") poate scrie un text variabil într-o pagină HTML
- **JavaScript poate reacționa la evenimente -** un script JavaScript poate fi lansat în execuție atunci când are loc un eveniment, precum încărcarea unei pagini sau când utilizatorul face click pe un element HTML
- **JavaScript poate citi și modifica elemente HTML –** un script JavaScript poate citi sau modifica conținutul unui element HTML
- **JavaScript poate fi utilizat pentru validarea datelor –** un script JavaScriptpoate fi utilizat pentru validarea datelor unei forme înainte ca acestea să fie trimise pentru prelucrare către un server. Aceasta duce la economisirea resurselor serverului
- **JavaScript poate detecta tipul și versiunea browser-ului** - JavaScript poate fi utilizat pentru detectarea tipului și versiunii browser-ului clientului și, depinzând de browser, poate adapta conținul paginii sau încărca alta pagină, specific creatră pentru acel browser
- **JavaScript poate crea cookies** - un script JavaScript poate fi utilizat pentru a stoca și regăsi informații pe calculatorul clientului

5.3 unde și cum?

Codul JavaScript integrat într-o pagină HTML va fi executat imediat ce pagina este încărcată în browser.

Nu întotdeauna dorim acest lucru. Uneori dorim ca executarea unui script să aibă loc la încărcarea paginii, alteori, ca efect al apariției unui eveniment. Din aceste motive, momentul execuției unui cod JavaScript depinde de locația acestuia.

5.3.1 scripturi în secțiunea HEAD a paginii

Scripturile care urmează a fi executate atunci când sunt apelate sau atunci când are loc un eveniment, trebuie puse în secțiunea HEAD a paginii HTML. Acest lucru ne asigură că scriptul este încărcat înainte de a fi executat de cineva/ceva. Iată un mic exemplu:

```
<html>
<head>
   <script type="text/javascript">
   ...
   </script>
</head>
```

5.3.2 scripturi în secțiunea BODY

Scripturile care trebuie executate la încărcarea paginii sunt puse în secțiune BODY a paginii HTML. Un astfel de script va genera parte a conținutului acelei pagini. Un exemplu:

```
<html>
<head>
</head>
<body>
   <script type="text/javascript">
   ...
   </script>
</body>
```

5.3.3 utilizare unui script extern

Scripturile externe sunt foarte utile atunci când sunt utilizate repetat. Funcțiile implementate astfel pot fi parte a unei biblioteci de scripturi sau putem avea o implementare a unei singure funcții într-un fișier extern. Fișierele care conțin scripturi JavaScript au extensia .js.

Notă: Un script extern nu poate conține elemntul (tag-ul) <script>!

Pentru utilizarea unui script extern este suficient să precizăm specificația sa de fișier ca valoare a atributului `src` din elementul (tag-ul) <script>:

```
<html>
  <head>
    <script src="myScript.js">
    </script>
  </head>
  <body>
  </body>
</html>
```

5.4 variabile și expresii JavaScript

O variabilă este un container pentru o informație a cărei valoare se poate modifica pe durata executării

scriptului.

5.4.1 numele variabilelor

Reguli pentru numele variabilelor:

- Numele variabilelor sunt case sensitive
- Trebuie să înceapă cu o literă sau cu caracterul bară jos (underscore).

5.4.2 declararea variabilelor

O variabilă poate fi declarată sau creată prin utilizarea cuvântului cheie `var`:

```
var strnum = "2157 Sunrise Blvd";
```

sau

```
strnum = "2157 Sunrise Blvd";
```

5.4.3 atribuirea de valori variabilelor

Atribuirea unei valori unei variabile poate avea loc la declararea acesteia:

```
var strnum = "Pobeda 771"
```

Sau doar utilizând o instrucțiune de atribuire, fără declarare prealabilă:

```
strname = "Pobeda 771"
```

5.4.4 tipuri de variabile

Declararea unei variabile în JavaScript nu conține și o precizare a tipului acesteia. Tipul unei variabile este de fapt determinat de ultima instrucțiune de atribuire care privește această variabilă. Aceasta înseamnă că tipul unei variabile se poate modifica pe durata executării scriptului.

5.5 instrucțiuni de control

Pe lângă intrucțiunile de control uzuale, întâlnite în limbajul C, și anume: – if ... else, switch(), for(), while(), break, continue, while() se cuvine să menționăm două tipuri noi, și anume: `for ... in` și `try ... catch`.

5.5.1 instrucțiunea for...in

Instrucțiunea `for...in` este utilizată pentru parcurgerea (iterarea) elementelor unei matrici (tablou) prin intermediul proprietăților unui obiect.

Codul din blocul de instrucțiuni este executat o singură dată pentru fiecare element/proprietate a matricii (tabloului).

Sintaxa

```
for (variable in object) {
    code to be executed
}
```

Argumentul variabil (de iterare) poate fi o variabilă, un element al tabloului sau o proprietate a unui obiect.

Exemplu

Utilizarea `for...in` pentru parcurgerea unui tablou:

```
<html>
<body>
   <script type="text/javascript">
   var x;
   var mycars = new Array();
   mycars[0] = "Saab";
   mycars[1] = "Volvo";
   mycars[2] = "BMW";

   for (x in mycars) {
   document.write(mycars[x] + "<br />");
   }
   </script>
</body>
</html>
```

5.5.2 prinderea erorilor/excepțiilor

Există două modalități de prindere a erorilor/excepțiilor în paginile web:

- utilizând construcția `try...catch`
- utilizând evenimentul `onerror`. Aceasta este modalitatea veche de prindere a erorilor:

5.5.3 instrucțiunea `try...catch`

Instrucțiunea (construcția) `try...catch` permite testarea unui block de instrucțiuni pentru eventuale erori. Blocul `try {...}` conține codul ce urmează a fi executat, iar blocul `catch {...}` conține codul ce urmează a fi executat în caz de eroare.

Sintaxa

```
try {
```

```
      // run some code here
   }
   catch(err) {
      // handle errors here
   }
```

Exemplu:

```
<html>
<head>
<script type="text/javascript">
var txt=""
function message() {
   try {
      adddlert("Welcome guest!");
   }
   catch(err) {
      txt="There was an error on this page.\n\n";
      txt+="Error description: " + err.description + "\n\n";
      txt+="Click OK to continue.\n\n";
      alert(txt);
   }
}
</script>
</head>

<body>
<input type="button" value="View message" onclick="message()" />
</body>
</html>
```

5.6 operatori

Trei operatori noi: „===”, „!==”, „**”. Pentru „===”, rezultatul este TRUE dacă cei doi operanzi au aceeași valoare și același tip. Operatorul „**” este operatorul de exponențiere (power). În plus față de limbajul C (dar prezent în Java, de exemplu) este operatorul + de adunare (concatenare) a șirurilor.

5.7 popup boxes (casuță de mesaje)

Există trei tipuri de căsuțe de mesaje (popup boxes) în Java Script.

5.7.1 căsuță de alertă (alert Box)

Este utilizată pentru afișarea unui mesaj informațional către utilizator. Utilizatorul are apoi posibilitatea de a îndepărta căsuța prin apăsarea butonului de OK.

Sintaxă:

```
alert("sometext");
```

5.7.2 căsuță de confirmare (confirm Box)

O căsuță de confirmare este de obicei utilizată atunci când se dorește verificarea sau acceptarea de către utilizator a unei variante. Utilizatorul poate apăsa fie butonul de Cancel sau de OK. În primul caz, valoarea de retur este FALSE, în al doilea caz, se returnează TRUE.

Sintaxă:

```
confirm("sometext")
```

De remarcat absența simbolului „;” la sfârșitul instrucțiunii de mai sus. Nu este o eroare, instrucțiunile JavaScript nu se termină obligatoriu (ca în C) cu acest simbol.

5.7.3 căsuță de introducere (prompt Box)

O căsuță de introducere este utilizată, în general, pentru introducerea unei valori într-un câmp de text. După introducere (sau chiar înainte), utilizatorul poate apăsa pe unul din butoanele Ok, sau Cancel. În primul caz, funcția returnează valoarea (șirul) introdusă. În al doilea caz, se returnează NULL.

Sintaxă:

```
prompt("sometext","defaultvalue")
```

5.8 funcții

5.8.1 definirea funcțiilor

O funcție conține un bloc de cod care va fi executat atunci când este apelată. O funcție poate fi apelată de oriunde din interiorul paginii sau chiar din exterior, atunci când funcția este definită într-un fișier extern. În mod uzual, funcțiile se definesc în zona de HEAD a paginii, dar acest lucru nu este obligatoriu.

Iată un exemplu:

```
<html>
<head>
   <script type="text/javascript">
      function displaymessage() { alert("Hello World!") }
   </script>
</head>
<body>
   <form>
      <input type="button" value="Click me!"
         onclick="displaymessage()" >
   </form>
</body>
</html>
```

Dacă instrucțiunea: `alert("Hello world!")`, din exemplul de mai sus nu ar fi fost scrisă în interiorul unei funcții, ea ar fi fost executată la încărcarea paginii. Dar în modalitatea implementată, instrucțiunea nu va fi executată decât la apăsarea butonului „Click me!”.

Legătura dintre apăsarea butonului și afișarea căsuței cu mesajul este făcută prin adăugarea evenimentului `onClick` ca atribut al butonului, cu valoarea `"displaymessage()"`. În urma apăsării butonului, se apelează funcția `displaymessage()` care afișează mesajul de alertare.

Sintaxa pentru crearea unei funcții este:

```
function functionname(var1,var2,...,varX) { some code }
```

var1, var2, etc. sunt variabile sau valori care sunt utilizate ca argumente ale funcției. Corpul funcției este conținut între acolade. Argumentele nu sunt obligatorii, ca în exemplul de mai jos:

```
function functionname() { some code }
```

Notă: Nu uitați de importanța capitalizării în JavaScript. Cuvântul cheie `function` trebuie scris exact așa, orice variație de capitalizare la definire sau la apelarea funcției duce la erori.

5.8.2 instrucțiunea return

Instrucțiunea `return` este utilizată pentru specificarea valorii returnate de către funcție. Funcțiile care returnează ceva trebuie să utilizeze instrucțiunea `return`. Exemplul de mai jos prezintă o funcție care returnează produsul celor două argumente ale sale:

```
function prod(a,b) { x=a*b return x }
```

La apelarea funcției trebuie precizate două argumente reale, fie sub forma unor constante (în cazul nostru), fie ca variabile inițializate.

```
product=prod(2,3)
```

Valoarea returnată de către funcția `prod()` este 6, iar valoarea va fi atribuită variabilei `product`.

5.9 obiecte JavaScript

5.9.1 limbaj orientat spre obiecte

JavaScript este un limbaj orientat spre obiecte (Object Oriented Programming (OOP) language). Un astfel de limbaj permite definirea propriilor obiecte (clase) precum și crearea propriilor tipuri de variabile.

JavaScript vine cu propriile sale obiecte (clase) predefinite, obiecte care vor fi trecute în revistă în următoarele paragrafe.

5.9.2 proprietăți (variabile membru)

Proprietățile sunt valori asociate unui obiect (clase).

În exemplul de mai jos, vom utiliza propritatea `length` a unui obiect de tip `String` pentru a obține numărul de caractere al unui șir:

```
<script type="text/javascript">
   var txt="Hello World!";
   document.write(txt.length);
</script>
```

Valorea afișată de codul de mai sus va fi 12.

5.9.3 metode

Metodele sunt acțiuni care pot fi efectuate de către obiecte sau asupra obiectelor.

În exemplul de mai jos vom utiliza metoda `toUpperCase()` a unui obiect de tip `String` pentru a afișa un șir capitalizat:

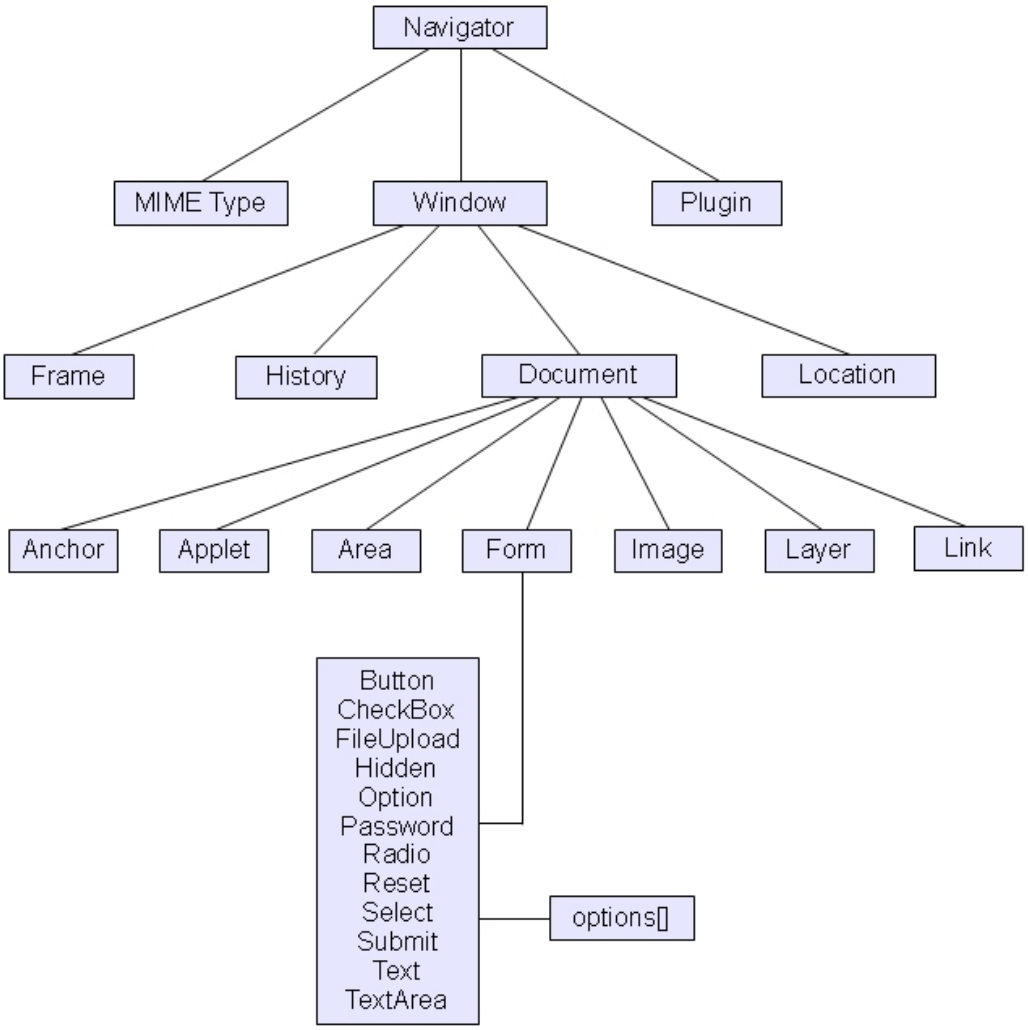
```
<script type="text/javascript">
   var str="Hello world!";
   document.write(str.toUpperCase());
</script>
```

5.10 ierarhia obiectelor browser în JavaScript

Există două clase majore de obiecte implicite în limbajul JavaScript. Prima clasă constă din obiecte specifice browser-elor. Cealaltă clasă conta din obiectele specifice limbajului, obiecte care vor fi detaliate în paragraful următor.

O pagină web poate fi considerată ca o colecție de elemente individuale, denumite generic Obiecte. De exemplu, fiecare imagine este un obiect, fiecare link este de asemenea un obiect. Pagina însăși este un obiect, care servește drept container pentru majoritatea obiectelor paginii.

Java Script oferă instrumentele care permit controlul aupra acestor obiecte.



Obiectele, în acest context, pe lângă proprietăți și metode, mai au și **evenimente** care le sunt asociate. Evenimentele sunt semnale sau mesage care au loc atunci când se întâmplă anumite acțiuni predefinite în browser sau când un utilizator interacționează cu o pagină web. Atunci când un mesaj de eveniment este generat, este nevoie de un mecanism de **interceptare** și procesare a acestuia. Acest lucru poate fi realizat prin intermediul unui **Event Handler**.

Pentru o listă completă a obiectelor implicite de tip browser din JavaScript, puteți consulta link-ul: http://www.w3schools.com/jsref/default.asp .

5.11 obiecte implicite în JavaScript

Obiectele standard pentru care JavaScript oferă suport pot fi plasate în mai multe categorii, conform referinței complete a obiectelor predefinite ale limbajului - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference .

- **Obiecte fundamentale**, precum: Object, Function, Boolean, Symbol, Error, EvalError, InternalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError
- **Numerice și dată** – Number, Math, Date
- **Procesare text** – String, RegExp

- **Colecții indexate** – Array, Int8Array, Uint8Array, Uint8ClampedArray, Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, Float64Array
- **Colecții asociative** – Map, Set, WeakMap, WeakSet
- **Colecții vectoriale** – SIMD, SIMD.Float32x4, etc. (date aranjate în benzi (lanes))
- **Date structurate** – ArrayBuffer, SharedArrayBuffer, Atomics, DataView, JSON
- **Obiecte pentru abstractizarea controlului** – Promise, Generator, GeneratorFunction, AsyncFunction
- **Reflecție** – Reflect, Proxy
- **Internaționalizare** – Intl, Intl.Collator, Intl.DateTimeFormat, Intl.NumberFormat
- **WebAssembly** – WebAssembly, WebAssembly.Module, etc.
- **Obiecte non-standard** – Iterator, ParallelArray, StopIteration
- **Altele** – arguments

5.11.1 Un exemplu, obiectul String

Obiectul **String** este utilizat pentru procesarea textelor (șirurilor de caractere).

Proprietăți

Proprietate	***Descriere***
constructor	O referință a funcției care a creat obiectul
length	Conține numărul de caractere din șir
prototype	Permite adăugarea de proprietăți și metode la obiectul curent

Metode

Metodă	***Descriere***
anchor()	Crează o ancoră (anchor) HTML
big()	Afișează un caracter într-un font mare
blink()	Afișează un șir intermitent
bold()	Afișează un șir îngroșat
charAt()	Returnează caracterul de la poziția specificată
charCodeAt()	Returnează caracterul Unicode de la poziția specificată
concat()	Combină două sau mai multe șiruri
fixed()	Afișează un string în format fix
fontcolor()	Afișează un șir în culoarea specificată
fontsize()	Afișează un șir în font de mărimea specificată
fromCharCode()	Găsește într-un șir un caracter Unicode, returnează un șir
indexOf()	Returnează poziția primei apariții a subșirului specificat
italics()	Afișează un șir înclinat (italic)
lastIndexOf()	Returnează poziția ultimei apariții a subșirului specificat
link()	Afișează un string în format fix
match()	Caută valoarea specificată în șir
replace()	Înlocuiește anumite caractere cu alte caractere în șir
search()	Caută valoarea specificată în șir

slice()	Extrage o parte a unui șir și o returnează ca nou șir
small()	Afișează un string în cu font mic
split()	Despart un șir înt-un tablou de subșiruri
strike()	Afișează un string cu caracterele tăiate
sub()	Afișează un string în format fix
substr()	Extrage un anumit număr de caractere dintr-un șir, de la poziția specificată
substring()	Extrage un număr de caractere dintr-un șir, între doi indici
sup()	Afișează un string în format superscript
toLowerCase()	Afișează un string cu litere mici
toUpperCase()	Afișează un string cu litere mari
toSource()	Returnează codul sursă al unui obiect
valueOf()	Returnează valoarea primitivă a unui obiect de tip String

Cele mai multe din proprietăți și metode sunt ușor de înțeles. Câteva cuvinte însă despre proprietatea `prototype`.

Prototype este un mecanism de extindere (cu noi proprietăți sau metode) a unui obiect existent. Toate obiectele standard din JavaScript au această proprietate. Un exemplu din https://www.w3schools.com/js/js_object_prototypes.asp de utilizare a acestui mecanism.

```
    function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";
```

5.12 construirea de obiecte noi

Există trei metode de a crea obiecte în JavaScript, și anume:

- definește și crează o instanță a unui obiect utilizând un nou nume
- definește și crează o instanță a unui obiect utilizând cuvântul cheie `new`
- definește un constructor, crează apoi noi obiecte de acel tip

5.13 procesarea evenimentelor

Evenimentele pot fi clasificate în două categorii: evenimente generate de utilizator și evenimente specifice HTML, precum încărcarea unei pagini sau primirea unui răspuns.

Exemple de evenimente:

- mouse click
- încărcarea unei pagini web sau a unei imagini
- trecerea cu mouse-ul peste un element al paginii
- selectarea unui câmp de introducere într-o formă HTML
- trimiterea unei forme HTML

- apăsarea unei taste pe tastatură

Lista evenimentelor este în mare măsură dependentă de suportul oferit de un browser sau altul. Pentru o listă aproape exhaustivă, puteți consulta link-ul - https://developer.mozilla.org/en-US/docs/Web/Events .

Parte din aceste evenimente sunt prezentate în tabelul de mai jos:

Eveniment	***Evenimentul are loc când ...***
onabort	Încărcarea unei imagini este întreruptă
onblur	Un element își pierde focus-ul
onchange	Utilizatorul modifică conținutul unui câmp
onclick	Mouse click pe un obiect
ondblclick	Mouse double-click pe un obiect
onerror	O eroare este detectată la încărcarea unui document sau a unei imagini
onfocus	Un element primește focus-ul
oninput	Un element primește input
onkeydown	O tastă este apăsată
onkeypress	O tastă este apăsată sau ținută apăsată
onkeyup	O tastă este eliberată
onload	O pagină sau o imagine își termină încărcarea
onmousedown	Un buton al mouse-ului este apăsat
onmousemove	Mouse-ul este mișcat
onmouseout	Mouse-ul este deplast de pe un element
onmouseover	Mouse-ul este deplasat peste un element
onmouseup	Un buton al mouse-ului este eliberat
onreset	Butonul de reset este apăsat
onresize	O fereastră sau un frame este redimensionat
onscroll	Are loc atunci când scrollbar-ul este acționat
onselect	Un text este selectat
onsubmit	Butonul de submit este apăsat
onunload	Utilizatorul părăsește pagina

Evenimentele specifice unui element pot fi procesate prin adaugarea evenimentului ca atribut al acelui element, iar valoarea atributului este setată cu acțiunea dorită, în general, invocarea unei funcții sau a unui cod JavaScript explicit.

Un scurt exemplu:

```
<button onclick="document.getElementById('timpul').innerHTML =
Date()">Afișează data și ora</button>
```

Ca urmare a apăsării butonului (elementul curent) este invocată funcția Date() care va afișa (în zona de conținut a elementului din pagină identificat prin ID-ul „timpul”, data și timpul curent, în formatul standard (ceva de genul - `Sat Oct 07 2017 10:48:22 GMT+0300 (GTB Daylight Time)`).

5.14 biblioteci JavaScript

Bibliotecile JavaScript sunt multe și diverse. Prezentăm doar o listă a celor mai importante categorii de biblioteci și bibliotecile majore din fiecare categorie.

chapter 5

1. Suport pentru DOM – Document Object Model
    - Dojo Toolkit
    - jQuery
    - MooTools
    - React
2. Vizualizare (suport grafic)
    - Highcharts
    - AnyChart
    - JavaScript InfoVis Toolkit
    - Velocity.js
3. Interfață utilzator (GUI widgets)
    - AngularJS
    - Bootstrap
    - jQWidgets
    - Webix
4. Suport Ajax
    - Joose
    - JsPHP
    - Microsoft's Ajax library
    - Rico
5. Suport pentru aplicații web
    - AngularJS
    - Ember.js
    - Google web toolkit
    - MooTools
    - Node.js
6. Testare
    - Jasmine
    - Mocha
    - QUnit

chapter 6 jQuery

6.1 ce este?

jQuery este o bibliotecă JavaScript, destinată în special programării pe partea de client a paginilor HTML. Metodele implementate în această bibliotecă au ca obiect task-urile cu înalt nivel de utilizare de către programatorii JavaScript. Acestea sunt ambalate sub forma unei metode care poate fi apelată foarte simplu de către programator.

jQuery acoperă, prin metodele implementate, următoarele domenii:

- procesare obiecte HTML/DOM
- procesare CSS
- procesare evenimente HTML
- efecte și animație
- Ajax
- utilitare
- procesare JSON
- extensibilitate, prin plugin-uri

jQuery este gratis, open source, subiect al licențierii permisive MIT.

6.2 utilizare

6.2.1 includere

Întreaga bibliotecă jQuery este un singur fișier de tip JavaScript care conține implementarea tuturor metodelor sale.

Poate fi inclusă într-o pagină web prin includerea unei copii locale, codul de includere fiind foarte simplu:

```
<script src="jquery.js"></script>
```

Totodată, biblioteca jQuery poate fi accesată printr-o rețea de de livrare a conținutului – CDN – Content Delivery Network. Printre CDN-urile care oferă jQuery menționăm MaxCDN (devenită recent StackPath) – www.maxcdn.com, Google sau Microsoft.

Includerea prin intermediul unui CDN este și ea simplă, de exemplu:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
```

6.2.2 stiluri de utilizare

jQuery are două moduri de utilizare:

1. prin intermediul funcției **$**, care este o metodă „factory” pentru obiectul jQuery. Aceste funcții pot fi utilizate în cascadă, deoarece toate returnează obiecte jQuery
2. prin intermediul funcțiilor cu prefixul **$.** . Acestea sunt funcții utilitare care nu acționează direct asupra obiectului jQuery

Un exemplu pentru primul mod:

```
$(document).ready(function(){
```

```
  $("p").click(function(){
    $(this).hide();
  });
});
```

Un exemplu pentru al doilea mod de utilizare (din - https://en.wikipedia.org/wiki/Jquery).

```
$.each([1,2,3], function() {
  console.log(this + 1);
});
```

Efectul executării acestui cod este listarea valorilor 2, 3, 4 la consolă.

În jQuery, `each()` este o funcție de iterare (parcurgere, traversare) a obiectelor sau listelor.

6.3 sintaxa jQuery

În general, avem de selectat elemente HTML, asupra cărora efectuăm un anumit tip de acțiune.

Sintaxa de bază este:

```
$(selector).actiune()
```

care se traduce prin:

- semnul $ e pentru a accesa jQuery
- un (selector) pentru a găsi elemente HTML
- o acțiune() jQuery asupra elementelor HTML

Exemple (din http://www.w3schools.com/jquery/jquery_syntax.asp):

`$(this).hide()` - ascunde elementul curent

`$("p").hide()` - ascunde toate elementele de tip paragraf (<p>)

`$(".test").hide()` - ascunde toate elementele cu clasa="test"

`$("#test").hide()` - ascunde elementul cu id="test"

Pentru cei familiarizați cu CSS, recunoaștem sintaxa CSS pentru selectarea elementelor.

6.4 evenimentul „document.ready”

În primul exemplu (cel din paragraful 6.2), funcțiile jQuery sunt incluse într-un eveniment „$(document).ready()”. Acest lucru este necesar pentru a preveni execuția vreunui cod jQuery înainte de încărcarea completă a documentului.

Dacă documentul nu ar fi complet încăcat, acțiuni precum cele listate mai jos ar putea să eșueze:

- ascunderea unui element care nu a fost încă creat
- obținerea mărimii unei imagini care nu a fost complet încărcată

6.5 selectori jQuery

Acești selectori sunt aceiași ca în CSS. Iată-i:

- selectorul element – selectarea are loc în baza numelui elementului
- selectorul via ID - #id_element – utilizează valoarea atributului id al unui element (id_element în cazul nostru) pentru selecția elementului afectat
- selectorul .numeClasa – permite selectarea elementelor care au clasa specificată (mai bine zis, valoarea atributului class e cea specificată)

Exemple pentru toți acești selectori. Începem cu un exemplu pentru selectorul cu nume element.

```
$(document).ready(function(){
  $("button").click(function(){
    $(".test").hide();
  });
});
```

Pentru a selecta elementul cu id=”myId” utilizăm:

```
$("#myId")
```

Pentru a selecta elementele de clasa=”myClass” utilizăm:

```
$(".myClass")
```

Alte exemple, mai puțin evidente:

- `$(this)`
- `$("p.intro")` – selectează toate elementele de tip <p> și clasă=intro
- `$("ul li:first")` – selectează primul element <li> al primului
- `$("[href]")` – selectează toate elementele cu un atribut href
- `$("a[target='_blank']")` – selectează toate elementele de tip <a> și cu valoarea atributului target='_blank'

6.6 metode pentru evenimente

Evenimentele asociate unei pagini sunt, în general, generate de acțiuni ale utilizatorilor. Desigur că și alte evenimente, precum sosirea unui mesaj asincron sau o eroare neprogramată, sunt evenimente, chiar dacă nu sunt generate (direct) de către un utilizator.

Exemple de evenimente:

- terminarea încărcării paginii
- mișcarea mouse-ului peste un element HTML
- selectarea unui buton radio
- click pe un element

Specificația DOM (Document Object Model) prevede mai multe categorii de evenimente. Iată câteva dintre ele:

- generate de acțiune mouse
- generate de tastatură
- generate de forme
- generate de ferestre/documente

Exemplu de utilizare:

```
$("p").click(function(){
```

```
  // action goes here!!
});
```

Funcția definește ce se întâmplă atunci când se face click pe un element de tip paragraf.

6.7 efecte jQuery

Iată o listă a efectelor mai importante jQuery. Pentru o listă completă, consultați link-ul - http://www.w3schools.com/jquery/jquery_ref_effects.asp

- hide() - ascunde elementul HTML pentru care e invocată funcția
- show() - arată elementul HTML pentru care e invocată funcția
- toggle() - trecere de la hide() la show() și invers
- fade() - trecere graduală a unui element la starea de invizibil
- animate() - utilizată pentru a crea animații. Are ca argumente o listă de parametri cu valorile lor, viteza (exprimată în unități de timp) precum și o metodă de callback (apelată la sfârșitul animației)
- slideUp() - slides up elementul pentru care e apelată metoda
- delay() - funcția pasată funcției delay() este executată cu întârzierea cerută

Iată un exemplu de slideDown():

```
$("#flip").click(function(){
  $("#panel").slideDown();
});
```

Atunci când se pface click pe elementul cu id-ul `flip`, are loc slideDown al elementului cu id-ul `panel`.

6.8 procesare elemente și atribute HTML

6.8.1 funcții de obținere (get)

Aceste funcții sunt:

- text() - returnează conținutul de text (text content) al elementului
- html() - returnează conținutul elementului (include marcaj HTML)
- val() - returnează valoarea câmpului unei forme

6.8.2 funcții de setare

Aceste funcții sunt aceleași ca mai sus, dar în acest caz au ca argument valoarea care va fi atribuită:

- text() - setează conținutul elementelor selectate
- html() - setează conținutul elementelor selectate (include marcaj HTML)
- val() - setează valoarea câmpului unei forme

6.8.3 funcții de adăugare

Aceste funcții sunt:

- append() - adaugă conținut la sfârșitul elementului

- prepend() - inserează conținut la începutul elementului
- after() - inserează conținut după elementele selectate
- before() - inserează conținut înainte de elementele selectate

6.8.4 funcții de eliminare

Aceste funcții sunt:

- remove() - elimină elementul selectat și copiii săi
- empty() - elimină urmașii (copiii) săi

6.8.5 funcții pentru clase CSS

Unele dintre funcții:

- addClass() - adaugă una sau mai multe clase la elementele selectate
- removeClass() - elimină una sau mai multe clase din elementele selectate
- toggleClass() - comută între adăugarea și eliminarea claselor din elementele selectate
- css() - setează sau returnează atributul stil

6.8.6 mărimi

jQuery permite dimensionarea elementelor precum și a ferestrei browser-ului. Metode specifice:

- width() - setează sau returnează lățimea unui element (exclude padding, border, margin)
- height() - setează sau returnează înălțimea unui element (exclude padding, border, margin)
- innerWidth() - returnează lățimea unui element (include padding)
- innerHeight() - returnează înălțimea unui element (include padding)
- outerWidth() - returnează lățimea unui element (include padding și border)
- outerHeight() - returnează înălțimea unui element (include padding și border)

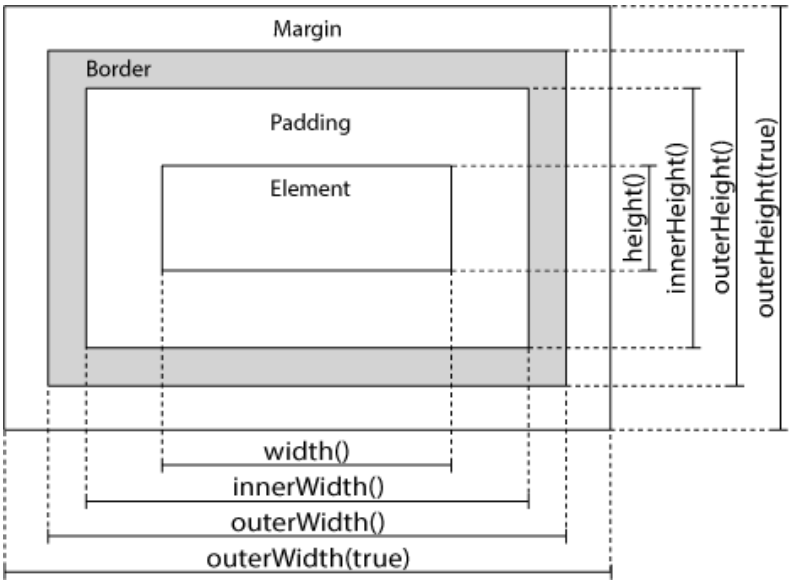


Figura 6.1 – dimensiuni (mărimi) jQuery

Figura de mai sus (din - http://www.w3schools.com/jquery/jquery_dimensions.asp) ilustrează semnificația acestor mărimi (dimensiuni).

6.9 traversare cu jQuery

Conceptul de traversare (în acest caz) se referă la parcurgerea obiectelor din specificația DOM (care au, implicit, și suport în jQuery) . Aceste obiecte pot fi reprezentate sub forma unui arbore cu rădăcină (rooted tree). Conceptele sunt cele din teoria arborilor:

- părinte (parent)
- urmaș (copil) (child)
- frate (soră) (sibling)

Metodele care facilitează această traversare (parcurgere, iterare) sunt listate la - http://www.w3schools.com/jquery/jquery_ref_traversing.asp

Cele mai importante sunt:

- children() - returnează toți descendenții elementului curent
- end() - returnează setul de elemente filtrate la starea anterioară
- filter() - reduce setul de elemente conform criteriilor de selectare
- find() - returnează descendenții elementului curent
- first() - returnează primul din elementele selectate
- next() - retunează următorul element de același nivel (sibling)
- parent() - returnează toți predecesorii elementului curent
- siblings() - returnează toate elementele care au acelați părinte cu cel curent

6.10 suport pentru Ajax

Tehnologiile Ajax (un set mic, dar eclectic) permit schimbul de date asincron cu un server, fără a fi nevoie de reîncărcarea întregii pagini. De obcei, doar un câmp al paginii este modificat.

jQuery oferă câteva metode ca suport pentru funcționalitatea Ajax.

Cererile trimise sunt fie de tip GET sau POST, iar răspunsul primit (în manieră asincronă, fără blocarea paginii curente) este fie de tip text, HTML, XML sau JSON.

6.10.1 metoda get()

Metoda get`()` cere date de la server prin intermediul unei cereri de tip HTTP GET.

Sintaxa acestei metode este:

```
$.get(URL, callback);
```

unde:

- url – specifică URL-ul pe care dorim să-l cerem (unde trimitem cererea)
- callback – un parametru opțional, indică metoda ce urmează a fi apelată dacă cererea reușește

Exemplu (din - http://www.w3schools.com/jquery/jquery_ajax_get_post.asp)

```
$("button").click(function(){
```

```
  $.get("demo_test.asp", function(data, status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```

6.10.2 metoda post()

Metoda post`()` cere date de la server prin intermediul unei cereri de tip HTTP POST.

Sintaxa acestei metode este:

```
$.post(URL,data,callback);
```

unde:

- URL – specifică URL-ul pe care dorim să-l cerem
- data – specifică datele care sunt trimise odată cu cererea (opțional)
- callback – un parametru opțional, indică metoda ce urmează a fi apelată dacă cererea reușește

Exemplu (din - http://www.w3schools.com/jquery/jquery_ajax_get_post.asp)

```
$("button").click(function(){
  $.post("demo_test_post.asp",
  {
    name: "Donald Duck",
    city: "Duckburg"
  },
  function(data, status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```

6.10.3 metoda load()

Metoda **`load()`** încarcă datele primite de la server și le pune în elementul selectat.

Sintaxa acestei metode este:

```
$(selector).load(URL,data,callback);
```

unde:

- URL – specifică URL-ul pe care dorim să-l încărcăm
- data – un set de perechi parametru/valoare care este trimis odată cu cererea
- callback – un parametru opțional, care indică metoda ce urmează a fi apelată după execuția metodei load()

Un exemplu din - http://www.w3schools.com/jquery/jquery_ajax_load.asp :

```
$("button").click(function(){
  $("#div1").load("demo_test.txt", function(responseTxt, statusTxt, xhr){
    if(statusTxt == "success")
      alert("External content loaded successfully!");
```

```
    if(statusTxt == "error")
      alert("Error: " + xhr.status + ": " + xhr.statusText);
  });
});
```

6.10.4 alte metode

- #.ajax() - execută o cerere ajax asincronă
- $.ajaxSetup() - setează valorile implicite pentru viitoare cereri Ajax
- $.ajaxTransport() - crează un obiect ce transmite datele Ajax
- $.getJSON() - încarcă date în format JSON utilizând o cerere HTTP GET
- ajaxSend() - specifică o funcție ce este apelată înainte de trimiterea cererii
- serialize() - codifică elemente ale unei forme ca șir pentru trimitere

6.11 metoda noConflict()

Această metodă are rolul de a preveni confuzii de namespace (spațiu de numire). jQuery utilizează prefixul $ pentru metodele sale dar alte medii de dezvoltare, precum Angular, Ember sau Knockout ar putea face același lucru.

Invocarea metodei noConflict() duce la eliberarea identificatorului $, care poate fi utilizat de către alte scripturi.

jQuery poate fi utilizat apoi, însă doar cu menționarea numelui întreg. Un exemplu:

```
$.noConflict();
jQuery(document).ready(function(){
  jQuery("button").click(function(){
    jQuery("p").text("jQuery is still working!");
  });
});
```

Este posibilă utilizarea unui prefix la alegere, care este de fapt returmat de către metoda noConflict().

Un exemplu luat din – http://www.w3schools.com/jquery/jquery_noconflict.asp.

```
var jq = $.noConflict();
jq(document).ready(function(){
  jq("button").click(function(){
    jq("p").text("jQuery is still working!");
  });
});
```

chapter 7 DOM – modelul obiectelor documentului

7.1 ce este DOM?

DOM, acronim pentru Document Object Model, este o interfață, neutră din punct de vedere al platformei sau al limbajului, care permite programelor și scripturilor să acceseze și modifice conținutul, structura sau stilul unui document.

Specificația DOM, controlată de W3C (World Wide Web Consortium), oferă un set standardizat de obiecte pentru documente HTML și XML precum și o interfață standardizată pentru accesarea și utilizarea acestora.

Specificația DOM este bidimensională, având trei părți componente (Core, XML și HTML), structurate pe patru (cinci) nivele (Nivele DOM (0)/1/2/3/4).

- Core DOM - definește un set standard de obiecte pentru orice document structurat
- XML DOM - definește un set standard de obiecte pentru documente XML
- HTML DOM definește un set standard de obiecte pentru documente HTML

Un browser web nu este obligat să utilizeze DOM pentru a reda o pagină web. Pe de altă parte, interpretorul JavaScript are nevoie de DOM pentru a parcurge, interpreta și modifica dinamic o pagină web. Cu alte cuvinte, JavaScript vede conținutul paginii HTML precum și elementele de stare a browserului prin intermediul DOM.

Deoarece DOM oferă suport pentru navigarea în ambele sensuri și permite modificări de tot felul, o implementare a specificației DOM trebuie să ofere suport pentru memorarea (stocarea) documentului ce a fost deja citit. Din acest motiv, DOM este potrivit pentru aplicații care accesează documentele în mod repetat sau în ordine aleatoare.

În cazul accesării documentelor într-o singură direcție, este mai potrivită (și mai rapidă) utilizarea SAX (Simple API for XML). SAX oferă un mecanism de citire a datelor dintr-un fișier XML și este o alternativă viabilă pentru DOM, în acest caz.

7.2 scurt istoric

WC3 DOM a fost creat de către World Wide Web Consortium ca răspuns la proliferarea unor modele specifice fiecărui tip de browser. Aceste modele specifice, care au precedat formalizarea DOM au primit denumirea generică de DOM-uri intermediare.

Dezvoltarea specificației DOM a început la mijlocul anilor 90, un model precursor fiind inclus în specificația HTML 4.

Prima specificație formală, DOM 1, a apărut în octombrie 1998, urmată de specificația DOM 2 în noiembrie 2000. Specificația 2 aduce (printre altele) noutăți referitoare la elementele de stil. Specificația DOM 3 a fost publicată în aprilie 2004 și este versiunea pentru care are suport generalizat din partea browserelor.

Versiunea curentă este DOM 4, a apărut în noiembrie 2015.

Începând cu ianuarie 2008 activitatea specifică pentru DOM încetează. Grupul de lucru pentru DOM a fost închis în primăvara 2004, după terminarea lucrului la recomandările prezente în specificația DOM 3. Activitatea acestuia este continuată prin intermediul altor grupuri de lucru în domeniul web, printre acestea menționăm cele pentru HTML, SVG, CSS sau WebAPI.

Grupul de lucru pentru aplicații web (**Web Applications WG**) și-a încetat activitatea în octombrie 2015, componentele activității sale fiind transferate grupului de lucru pentru platforma web (**Web Platform WG**).

Grupul de lucru pentru platforma web are ca sarcini continuarea dezvoltării limbajului HTML, prin crearea de specificații care permit îmbunătățirea dezvoltării de aplicații pentru Web, incluzând API-uri pentru partea de client precum și elemente de vocabular pentru descrierea și controlul comportamentului aplicațiilor client.

7.3 nivele

Nivel 0

Desemnează support pentru o specificație care exista înainte de crearea specificației DOM 1. Exemple includ *DHTML Object Model* sau Netscape intermediate DOM. Nivelul 0 nu este o specificație formală, ci desemnează ceva ce exista înainte de începerea procesului de standardizare.

Nivel 1

Specifică elemente de navigare a unui document DOM (HTML și XML) (o structură de tip arbore) precum și de procesare a conținutului (include și adăugarea de elemente). Elementele specifice HTML sunt incluse și ele.

Nivel 2

Aduce suport pentru spațiul de numire (namespace) XML, vederi filtrate și evenimente.

Nivel 3

Constă din 6 specificații diferite:

1. DOM Level 3 Core;
2. DOM Level 3 Load and Save;
3. DOM Level 3 XPath;
4. DOM Level 3 Views and Formatting;
5. DOM Level 3 Requirements; precum și
6. DOM Level 3 Validation

Nivel 4

Apărut ca Recomandare W3C în noiembrie 2015. Pentru detalii, folosiți link-ul - https://www.w3.org/TR/2015/REC-dom-20151119/ .

Procesul adoptării și modificării DOM este considerat acum ca parte WHATWG - Web Hypertext Application Technology Working Group - o comunitate de persoane interesate în evoluția MTHL și a tehnologiilor adiacente, creată ca reacție la ritmul lent a proceselor de standardizare W3C.

7.4 DOM pentru HTML

DOM pentru HTML definește un set standard de obiecte pentru HTML precum și modalități standard de accesare și procesare a documentelor HTML.

Toate elementele (tags) HTML, precum și textul conținut sau atributele lor, pot fi accesate prin DOM. Conținutul lor poate fi modificat sau șters, iar noi elemente pot fi create.

DOM pentru HTML nu depinde de platformă sau de limbajul în care este utilizat. Poate fi folosit în limbaje diferite, precum Java, JavaScript sau VBScript.

Obiecte HTML DOM – o listă selectivă

Object	***Description***
Document	Reprezintă întregul document HTML ți poate fi utilizat pentru accesarea tutror elementelor din pagină
Anchor	Reprezintă un element <a>
Area	Reprezintă un element <area> dintr-un image-map

Base	Reprezintă un element <base> (specifică o adresă implicită sau o țintă implicită pentru toate link-urile dintr-o pagină)
Body	Reprezintă elementul <body>
Button	Reprezintă un element <button>
Event	Reprezintă starea unui eveniment
Form	Reprezintă un element <form>
Frame	Reprezintă un element <frame>
Frameset	Reprezintă un element <frameset>
Iframe	Reprezintă un element <iframe>
Image	Reprezintă un element <img>
Input button	Reprezintă un buton dintr-o formă HTML
Input checkbox	Reprezintă un checkbox dintr-o formă HTML
Input file	Reprezintă un element fileupload in an HTML form
Input hidden	Reprezintă un câmp ascuns dintr-o formă HTML
Input password	Reprezintă un câmp de parolă dintr-o formă HTML
Input radio	Reprezintă un buton radio dintr-o formă HTML
Input reset	Reprezintă un buton de reset dintr-o formă HTML
Input submit	Reprezintă un buton de submit dintr-o formă HTML
Input text	Reprezintă un câmp de introducere a unui text dintr-o formă HTML
Link	Reprezintă un element <link>
Meta	Reprezintă un element <meta>
Option	Reprezintă un element <option>
Select	Reprezintă o listă de selecție dintr-o formă HTML
Style	Reprezintă o instrucțiune de stil individual
Table	Reprezintă un element <table>
TableData	Reprezintă un element <td>
TableRow	Reprezintă un element <tr>
Textarea	Reprezintă un element <textarea>

7.5 noduri DOM

Conform specificației DOM, orice într-un document HTML este un nod. Astfel:

- Întreg documentul este un nod de tip document
- Orice element (tag) HTML este un nod de tip element
- Textul dintr-un element HTML este un nod de tip text
- Orice atribut HTML este un nod de tip atribut
- Comentariile sunt noduri de tip comentariu

În DOM pentru HTML există un obiect generic, numit **Element**, care reprezintă un element HTML. Obiectele de tip Element pot avea scuccesori (copii) de tip element, text sau comentariu. Elementele pot avea atribute, care sunt noduri de tip atribut.

Un obiect **NodeList** reprezintă o listă de noduri, precum mulțimea de succesori ai unui nod de tip element.

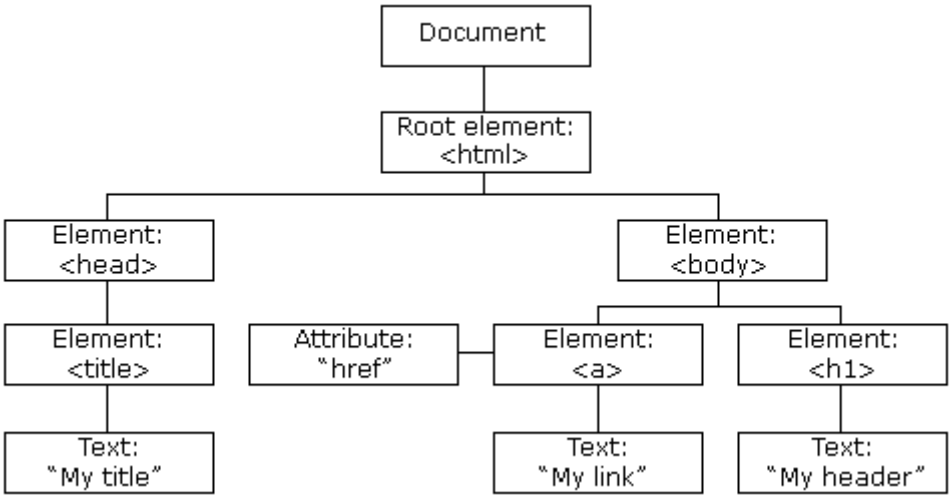
7.6 arborele nodurilor în DOM pentru HTML

7.6.1 arborele Document

DOM pentru HTML vede un document HTML ca o structură de tip arbore. Această structură este numită **node-tree** (**arborele nodurilor**).

Toate nodurile pot fi accesate prin intermediul acestui arbore. Conținutul acestora poate fi modificat sau șters, iar noduri noi pot fi adăugate.

Arborele nodurilor prezentat mai jos arată o mulțime de noduri și legăturile dintre ele. Arborele începe de la rădăcina sa și se extinde până la nodurile de tip text de la cel mai de jos nivel al arborelui.



7.6.2 părinți, copii și frați/surori

Nodurile din arbore au o relație de tip ierarhic.

Termenii de părinte, copil (succesor) și de frate/soră sunt utilizați pentru a descrie aceste relații. Nodurile părinte au copii (succesori, descendenți). Copiii de la același nivel sunt frați/surori.

- Într-un arbore de noduri, nodul din vârf se numește rădăcină
- Fiecare nod, cu excepția rădăcinii, are exact un nod părinte
- Un node poate avea orice număr de copii (descendenți)
- O frunză este un nod fără nici un copil
- Frații/surorile sunt noduri care au același părinte

7.6.3 accesarea nodurilor

Nodurile pot fi accesate în 3 moduri:

1. Utilizând metoda `getElementById()`

2. Utilizând metoda `getElementsByTagName()`

3. Navigând arborele nodurilor, utilizând relațiile dintre noduri

Următorul exemplu returnează o listă de noduri (nodeList) a tuturor elementelor de tip <p> care succesorii elementului cu atributul id="main":

```
document.getElementById('main').getElementsByTagName("p");
```

Proprietatea `length` definește lungimea unei liste de noduri (numărul nodurilor). Putem parcurge lista nodurilor utilizând această proprietate.

```
x=document.getElementsByTagName("p");
for (i=0;i<x.length;i++){
   document.write(x[i].innerHTML);
   document.write("<br />");
}
```

7.6.4 proprietățile nodurilor

În DOM pentru HTML, fiecare nod este un **obiect**.

Obiectele au metode (funcții) și proprietăți (variabile membre), care pot fi accesate și procesate prin intermediul limbajelor de scriptare, precum JavaScript.

Principalele proprietăți ale unui nod din DOM pentru HTML:

- nodeName
- nodeValue
- nodeType

proprietatea nodeName

Proprietatea **nodeName** specifică numele nodului:

- nodeName al unui nod de tip element este același cu numele elementului (tag-ului)
- nodeName al unui atribut este același cu numele atributului
- nodeName al unui nod de text este întotdeauna #text
- nodeName al nodului document este întotdeauna #document

proprietatea nodeValue

Proprietatea **nodeValue** specifică valoarea unui nod

- nodeValue pentru un nod de tip element este nedefinit
- nodeValue pentru noduri de tip text este textul însuși
- nodeValue pentru noduri de tip atribut este valoarea atributului

proprietatea nodeType

Proprietatea **nodeType** returnează tipul nodului și nu poate fi modificată. Cele mai importante sunt:

Tipul nodului	***nodeType***
Element	1
Atribut	2
Text	3
Comentariu	8
Document	9

7.7 evenimente HTML

Evenimentele HTML pot fi clasificate în mai multe categorii. Acestea sunt asociate elementelor HTML, ca atribute ale acestor elemente. Iată unele din aceste evenimente (atribute) și afiliația lor:

1. Evenimente fereastră (window)
    - onerror
    - onload
    - onmessage
    - onresize
2. Evenimente formă
    - onblur
    - oninput
    - onreset
    - onsubmit
3. Evenimente tastatură
    - onkeydown, onkeypress, onkeyup
4. Evenimente mouse
    - onclick, ondblclick, onmousemove, onmouseout, onmouseover, onmouseup, onwheel
5. Evenimente de agățare (drag)
    - ondrag, ondragend, ondrop, onscroll
6. Evenimente clipboard
    - oncopy, oncut, onpaste
7. Evenimente media
    - onabort, oncanplay, onpause, onplaying, onprogress, onseeking, onvolumechange, onwaiting
8. Alte evenimente
    - onshow, ontoggle

chapter 8 AJAX

8.1 ce este Ajax?

Ajax se traduce prin: Asynchronous JavaScript And XML. Nu este o thehnologie în sine, ci mai degrabă o colecție de tehnologii existente, reunite sub un singur nume prin intermediul limbajului JavaScript. Iată aceste tehnologii:

- HTML și CSS pentru partea de prezentare
- JavaScript (ECMAScript) pentru procesare locală (pe partea de client) precum și DOM (Document Object Model) pentru accesarea elementelor din pagină sau al informațiilor conținute în fișierul XML primit ca răspuns din partea serverului
- Clasa `XMLHttpRequest` pentru citirea sau trimiterea datelor către server, într-o manieră asincronă

Opțional:

- clasa `DomParser` poate fi utilizată
- PHP sau un alt limbaj de scriptare poate fi folosit pe partea de server
- XML și XSLT pentru procesarea datelor, dacă răspunsul de la server vine ca fișier XML
- SOAP poate fi utilizat pentru dialogul cu serverul

XSL se traduce prin eXtensible Stylesheet Language în timp ce XSLT înseamnă XSL Transformations.

Cuvântul "Asynchronous" (asincron) înseamnă că răspunsul serverului va fi procesat atunci când va fi disponibil, într-un fir de execuție (thread) separat, fără a fi nevoie să întrerupem lucrul cu pagina curentă.

8.2 arhitectura Ajax

Aplicațiile web clasice funcționează cam așa: o mare parte din acțiunile utilizatorului în interfață generează o cerere HTTP care este trimisă serverului. Serverul procesează într-un fel sau altul – regăsește date, efectuează calcule, comunică cu sisteme legacy – apoi returnează o pagină HTML către client. Este un model adaptat din începuturile utilizării web-ului ca mediu hypertext, dar ceea ce este potrivit pentru hypertext nu este neapărat potrivit pentru aplicații complexe.

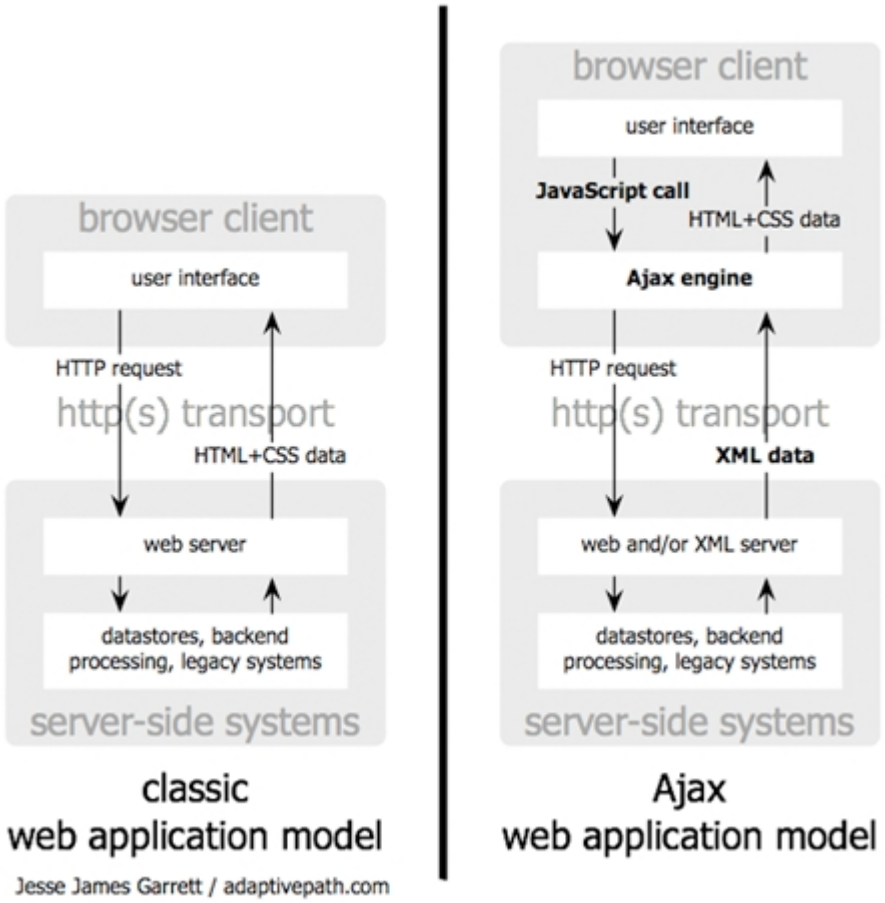
Această abordare are sens din punct de vedere tehnic, dar nu reprezintă o experiență convenabilă pentru utilizator. Ce face utilizatorul în timp ce serverul își face treaba? Exact, așteaptă. Și la fiecare pas important pentru îndeplinirea scopului aplicației, se așteaptă din nou.

Dacă ar fi să re-fundamentăm web-ul având în vedere aplicațiile web, nu am obliga utilizatorul să aibă parte de o doză lungă de așteptare. Din momentul ce interfața este încărcată, de ce ar trebui să oprim interacțiunea utilizatorului de fiecare dată când aplicația are nevoie de ceva din partea serverului?

O aplicație Ajax elimină natura start-stop-start-stop a interacțiunii pe web prin introducerea unui nivel intermediar – o mașinărie Ajax – între utilizator și server. Aparent, adăugarea unui alt nivel al face aplicația mai lentă, dar se întâplă exact opusul.

În locul încărcării unei pagini web, la începutul sesiunii, browser-ul încarcă mașinăria Ajax – scrisă în JavaScript. Această mașinărie este responsabilă atît pentru afișarea interfeței utilizator cât și pentru comunicarea cu serverul în beneficiul utilizatorului. Mașinăria Ajax permite ca interacțiunea utilizatorului cu aplicația să se desfășoare asincron – independent de comunuicarea cu serverul. Astfel, utilizatorul nu va mai fi nevoit să privească o clepsidră într-o pagină web înghețată, așteptând să se întâmple ceva.

Imaginea de mai jos (preluată de pe site-ul www.adaptivepath.com) prezintă comparativ modelul clasic pentru aplicații web precum și cel bazat pe Ajax.

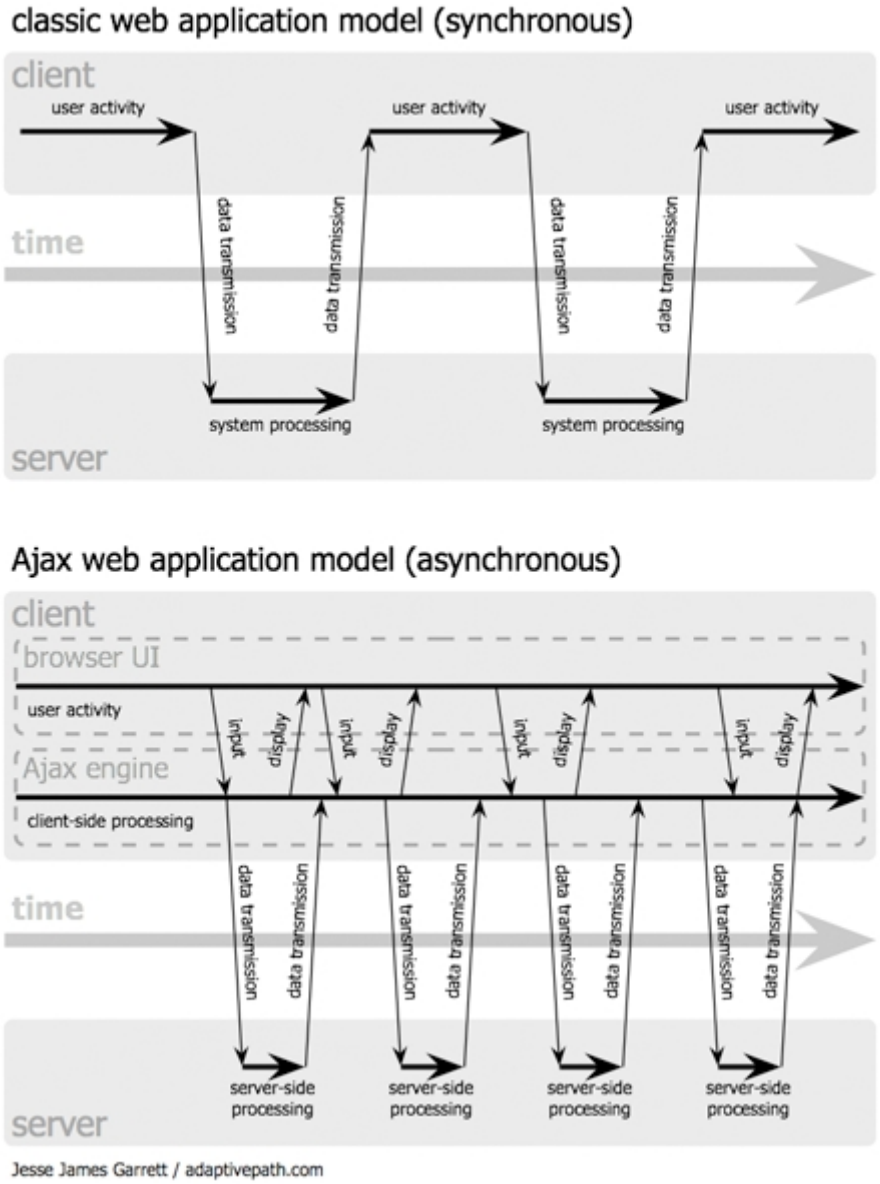


Modelul tradițional comparat cu modelul Ajax

8.3 comunicare sincronă versus asincronă

Fiecare acțiune a unui utilizator, care în mod normal generează o cerere HTTP, devine în schimb un apel JavaScript către mașinăria Ajax. Orice răspuns la acțiunea unui utilizator, care nu cere un drum dus-întors către server – precum validarea unor date, editarea datelor din memorie sau chiar unele cazuri de navigare – este administrat de către server. Dacă Ajax are nevoie de ceva din partea serverului pentru a răspunde – fie trimiterea de date pentru procesare, încărcarea de cod suplimentar pentru afișare sau pentru obținerea de noi date – Ajax face aceste cereri într-o manieră asincronă, de obicei utilizând XML, fără a deranja interacțiunea utilizatorului cu aplicația.

Imaginea de mai jos (luată de pe site-ul www.adaptivepath.com) prezintă modelele de interacțiune pentru o aplicație web tradițională și una cu support Ajax.



Modelul de interacțiune sincron al unei aplicații web tradiționale, comparat cu modelul asincron oferit de o aplicație Ajax

8.4 cum funcționează

Ajax utilizează un model de programare cu afișaj și evenimente. Aceste evenimente sunt în cea mai mare parte acțiuni ale utilizatorului, ele au drept consecință invocarea unor funcții asociate elementelor paginii.

Interactivitatea este obținută prin forme și butoane. Implementarea DOM permite asocierea elementelor paginii cu acțiuni și permite extragerea datelor din fișierele XML trimise de către server.

Pentru a obține date de la server, mașinăria Ajax utilizează obiectul **XMLHttpRequest**. Acest obiect oferă (printre altele) două metode pentru comunicarea cu serverul:

- **open**: pentru crearea unei conexiuni

- **send**: pentru a trimite cereri către server

Datele trimise de către server pot fi găsite în următoarele atribute ale obiectului XMLHttpRequest:

- **responseXml** – pentru un fișier Xml sau

- **responseText** – pentru un text simplu

Odată ce o cerere a fost trimisă, trebuie să așteptăm ca datele de răspuns să fie disponibile. Nivelul de disponibilitate este menținut în atributul **readyState** al obiectului XMLHttpRequest.

readyState poate avea următoarele coduri de stare (ultimul este și cel mai important):

0: ne-inițializat

1: conexiunea a fost stabilită

2: cererea a fost primită

3: răspunsul este în curs

4: terminat

8.5 clasa XMLHttpRequest

câteva detalii clasa despre clasa XMLHttpRequest. Ea permite interacțiunea cu serverul, datorită atributelor și metodelor sale. Iată mai jos o listă cu cele mai importante; pentru o listă exhaustivă, puteți consulta link-ul - https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest .

Atribute

readyState	- indică codul de stare al cererii, valoirile sale merg de la 0 la 4
status	- codul de stare HTTP returnat de server: 200 e Ok, 404 - page not found
responseText	- conține datele trimise de server sub forma unui șir de caractere
responseXml	- conține răspunsul de la server sub forma unui fișier XML
onreadystatechange	- numele funcției care va fi invocată la schimbarea stării cererii

Metode

open(mode, url, boolean)	- mode: tipul cererii - GET sau POST
	- url: locația fișierului care va procesa cererea
	- boolean: true (asincron) / false (sincron)
send("string")	- null pentru o comandă GET

8.6 construcția unei cereri, pas cu pas

Primul pas: crearea unei instanțe a clasei XMLHttpRequest

Pentru a asigura compatibilitatea cu cele două tipuri majore de browsere, codul trebuie să acopere ambele cazuri:

```
if (window.XMLHttpRequest) { // Object of the current windows
request = new XMLHttpRequest();  // Firefox, Safari, ...
}
else if (window.ActiveXObject) { // ActiveX version
   request = new ActiveXObject("Microsoft.XMLHTTP");      // IE
```

```
}
```

Al doilea pas: definirea unei funcții de procesare a răspunsului

Codul conține declararea și implementarea unei funcții JavaScript, un exemplu arată astfel:

```
function myFunction() {
   // urmează implementarea funcției
}
```

Al treilea pas – așteptarea răspunsului

Primirea răspunsului și procesarea sa ulterioară vafi făcută de de către funcția precizată ca valoare a atributului **onreadystatechange** al obiectului creat anterior.

```
request.onreadystatechange = myFunction();
if (request.readyState == 4) {
   // received, OK
}
else {
   // wait...
}
```

Al patrulea pas – crearea și trimiterea cererii

Sunt utilizate (în principal) două metode ale clasei XMLHttpRequest:

- **open**: are ca argumente numele comenzii - GET sau POST, URL-ul documentului, true pentru procesare asincronă.

- **send**: fără argument pentru GET, pentru POST, datele ce sunt trimise serverului

Cererea din exemplul de mai jos permite citirea unui document de pe server.

```
http_request.open('GET', 'http://www.xul.fr/somefile.xml', true);
http_request.send(null);
```

8.7 exemple

8.7.1 cum să obții un text

```
<html>
<head>
<script>
function submitForm()
{
  var req = null;
```

```
    if(window.XMLHttpRequest)  req = new XMLHttpRequest();
    else if (window.ActiveXObject)
      req = new ActiveXObject(Microsoft.XMLHTTP);

    req.onreadystatechange = function()
    {
      if(req.readyState == 4)
        if(req.status == 200)
         document.ajax.dyn="Received:" + req.responseText;
        else
          document.ajax.dyn="Error code " + req.status;
    };

    req.open("GET", "data.xml", true);
    req.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    req.send(null);
  }
  </script>
  </head>

  <body>
  <FORM method="POST" name="ajax" action="">
    <INPUT type="BUTTON" value="Submit" ONCLICK="submitForm()">
    <INPUT type="text" name="dyn" value="">
  </FORM>
  </body>
  </html>
```

8.7.2 răspuns cu conținut xml

Pentru a obține date dintr-un fișier XML trebuie înlocuită (în exemplul precedent) linia:

```
document.ajax.dyn=""Received:" + req.responseText;
```

cu codul:

```
var doc = req.responseXML;       // assign the Xml file to a var
var element = doc.getElementsByTagName('root').item(0);  // read the
first element with a dom's method
document.ajax.dyn.value= element.firstChild.data;  // assign the
   content of the element to the form
```

8.7.3 cum să trimiți un text

Se trimite serverului un text, care urmează a fi scris într-un fișier. Apelul către funcția `open()` este modificat, comanda HTTP utilizată este POST, iar metoda `send()` are în acest caz un argument.

```
req.open("POST", "ajax-post.xml", true);
req.setRequestHeader("Content-Type",
   "application/x-www-form-urlencoded");
req.send(document.getElementById("dyn".value));
```

8.7.4 cum să scrii în corpul unei pagini HTML

În acest exemplu, textul primit de la server este scris în corpul paginii. Codul de mai jos înlocuiește câmpul de text al elementului cu id-ul “zone” cu textul primit ca răspuns al unei cereri Ajax.

```
<div id="zone">
   ... some text to replace ...
</div>
document.getElementById("zone").innerHTML = "Received:" +
xhr.responseText;
```

8.8 ajax via jQuery

Toate metodele referitoare la Ajax din jQuery utilizează metoda $.ajax(). Reamintim sintaxa sa:

```
$.ajax({name:value, name:value, ... })
```

Iată un exemplu de utilizare a bibliotecii jQuery pentru suport Ajax:

```
$.ajax({
     // request type ( GET or POST )
   type: "GET",

     // the URL to which the request is sent
   url: mw.util.wikiScript('api'),

     // data to be sent to the server
   data: { action:'query', format:'json', lgname:'foo', lgpass:'fbar' },

     // The type of data that you're expecting back from the server
   dataType: 'json',

     // Function to be called if the request succeeds
   success: function( jsondata ){
         alert( jsondata.result );
   }
   });
```

chapter 9 aplicații web

9.1 tipuri de aplicații web

O aplicație web este o aplicație desfășurată pe un server web sau pe un server de aplicații și care este accesată printr-un browser. Altfel spus, o aplicație web este o extensie dinamică a unui server web sau de aplicații.

Aplicațiile web pot fi clasificate în două tipuri majore, în funcție de orientarea lor principală:

- **tip prezentare (presentation oriented)** – o astfel de aplicație generează pagini web interactive care conțin elemente de limbaje de markup, precum HTML, XHTML, XML, etc., precum și conținut dinamic generat ca răspuns la cererile clientului.
- **tip serviciu (service-oriented)**: o astfel de aplicație implementează punctul de acces al unui serviciu web (web service). Aplicațiile de tip prezentare sunt deseori clienți ai aplicațiilor web de tip serviciu.

În materie de implementare a unei aplicații web pe platforma Java EE, pe partea de server, aceasta este realizată prin intermediul componentelor web (web components). Acestea oferă capacitățile de extindere dinamică a unui server web. Aceste componente pot fi:

- servlete
- Java Server Faces (mai exact, pagini web implementate prin intermediul tehnologiei JSF)
- pagini JSP (Java Server Pages)

Servletele sunt clase Java care procesează în mod dinamic cererile clienților și construiesc răspunsuri la aceste cereri. Tehnologii Java, precum JSF sau Facelets, sunt utilizate pentru a construi aplicații web interactive.

Deși servletele și paginile Java Server Faces sau Facelets sunt utilizate pentru a atinge țeluri similare, fiecare din aceste tehnologii are puncte forte specifice.

Servletele sunt mai potrivite pentru aplicații orientate spre servicii (punctele de acces pentru servicii web pot fi implementate ca servlete) precum și ca funcții de control ale unei aplicații de tip prezentare, precum distribuirea cererilor și procesarea datelor ne-textuale.

Paginile JavaServer Faces sunt mai potrivite pentru generarea te fișiere de markup bazate pe text, precum XHTML.

Componenentele web beneficiază de suportul serviciilor oferite de către o platformă de execuție numită **container web**. Un container web oferă servicii precum distribuirea cererilor, securitate, acces concurent precum și administrarea ciclului de viață a componentelor. Containerul web oferă de asemenea acces la API-uri precum cele de numire, tranzacții sau pentru schimb de mesaje.

Anumite aspecte ale comportamentului unei aplicații web pot fi configurate utilizând adnotări Java sau prin intermediul unui fișier text de tip XML, numit descriptorul de desfășurare (deployment descriptor (DD)). Formatul unui descriptor de desfășurare trebuie să se conformeze schemei precizate în specificația Java pentru servlete.

9.2 ciclul de dezvoltare al unei aplicații web

O aplicație web constă din componente web, fișiere statice de resurse, precum imagini sau fișiere CSS, alte clase ajutătoare precum și biblioteci (de exemplu, scripturi JavaScript sau clase Java).

Containerul web oferă multe din serviciile de suport care suplimentează capabilitățile componentelor web și fac dezvoltarea acestora mai simplă. Totodată, deoarece o aplicație web trebuie să ia în considerare aceste servicii, procesul creării și execuției unei aplicații web este diferit de cel al unei aplicații Java de sine stătătoare.

Procesul creării, desfășurării și execuției unei aplicații web poate fi descris astfel:

1. dezvoltarea codului componentelor web și al resurselor statice
2. dezvoltarea descriptorului de desfășurare, dacă este necesar acest lucru
3. compilarea componentelor aplicației web și a claselor ajutătoare utilizate de componente
4. opțional, împachetarea aplicației într-o unitate desfășurabilă
5. desfășurarea (deployment) aplicației într-un container web
6. pentru execuție, accesează un URL care conține referința la aplicația web

9.3 structura unei aplicații web

O aplicație web este o colecție de servlete Java, pagini JSP, Java Server Faces, alte clase ajutătoare sau provenite din biblioteci, resurse statice (pagini HTML, imagini, etc.) precum și un fișier XML, descriptorul de desfășurare (deploynment descriptor).

O aplicație web constă (în general) din patru părți:

1. un director (folder) public – acesta conține fișiere html, jsp, precum și alte resurse. Acesta este directorul de bază al aplicației.
2. un fișier WEB-INF/web.xml – descriptorul de desfășurare (the deployment descriptor).
3. un director WEB-INF/classes – conține clase specifice aplicației.
4. Un director WEB-INF/lib – conține clase generice, din biblioteci proprii sau externe.

Pe lângă aceste părți obligatorii, o aplicație web mai poate conține și alte fișiere generice sau specifice serverului de aplicații.

Deseori, folderul WEB-INF poate conține un fișier numit **application.xml** sau un alt fișier specific platformei (serverului de aplicații). De exemplu, pe un server de aplicații Weblogic, folderul WEB-INF conține un descriptor de desfășurare numit **weblogic.xml**.

Pe lângă folderul WEB-INF, o aplicație web mai poate conține și un alt folder: META-INF, care conține de obicei un fișier de context, numit **context.xml**, un substitut pentru un element de tip <context> din fișierul de configurare la nivel de server (de aplicații), fișier numit **server.xml**.

Exemplu:

Să presupunem că utilzăm serverul web Tomcat și că variabila de mediu (environment variable) %TOMCAT_HOME% are valoarea C:\TW\Tomcat. În acest caz, directorul de bază al unei aplicații web **ccards** al firmei **bank11** poate fi:

```
C:\TW\Tomcat\webapps\bank11\ccards
```

iar folderele obligatorii sunt:

```
C:\TW\Tomcat\webapps\bank11\ccards\WEB-INF\classes
C:\TW\Tomcat\webapps\bank11\ccards\WEB-INF\lib
```

Începând cu Java EE 5, conținutul descriptorului de desfășurare poate fi înlocuit de așa-numitele admotări în fișierele sursă ale aplicațiilor.

Adnotările sunt o formă de metadate sintactice (comentarii inteligente) care pot fi adăugate fișierelor sursă Java. Pot fi adnotate clasele, metodele, variabilele, parametrii precum și pachetele Java. Spre deosebire de etichetele Javadoc, adnotările Java pot fi reflective, în sensul că ele pot fi încorporate în fișierele de tip class generate de către compiler și pot fi reținute în mașina virtuală Java pentru a fi utilizate în momentul execuției.

9.4 containere web

Un **container web** este o unitate de execuție Java care oferă o implementare a API-ului Java pentru servlete precum și unele facilități pentru paginile JSP și JSF. Containerul web este responsabil pentru inițializarea, invocarea și administrarea ciclului de viață al servletelor, paginilor JSP și JSF.

Un container web poate implementa serviciile de bază HTTP sau le poate delega unui server web extern.

Conatinerele web pot fi parte a unui server de aplicație, a unui server web sau pot fi o unitate de execuție separată. Descriem mai jos aceste situații.

- **container web într-un server de aplicații J2EE**. Implementări comerciale ale specificației J2EE, precum WebLogic (BEA Systems, acum Oracle), Enterprise Application Server (Borland) sau WebSphere de la IBM includ containere web.
- **container inclus într-un server web**. Cele mai cunoscute cazuri sunt Java WebServer de la Sun (Oracle) și serverul web Tomcat.
- **container web ca unitate de execuție separată**. Unele servere web, precum Apache sau IIS au nevoie de o unitate de execuție separată pentru a invoca servlete precum și un plugin pentru serverul web pentru a integra această unitate de execuție Java cu serverul web. Cazuri tipice pentru o astfel de integrare sunt Tomcat cu Apache și JRun (de la Allaire) cu cele mai multe servere de aplicații J2EE.

9.5 implementări ale containerelor web

9.5.1 containere web gratuite (non-commercial)

- Apache Tomcat (nume vechi - Jakarta Tomcat) este un container web (și nu numai) open source disponibil sub licența Apache Software License.
- Apache Geronimo este o implementare de către Apache a întregii specificații Java EE.
- GlassFish (server de aplicații open source), de la Oracle
- JBoss Application Server (open source) este o o implementare de către RedHat Inc. a întregii specificații Java EE.
- Jetty (open source) de la Eclipse Foundation. Suportă, în plus, protocoalele SPDY și WebSocket.
- Jaminid conține o implemntare la un nivel mai ridicat de abstractizare a servletelor.
- Enhydra Winstone, concentrat pe adaptarea configurării și a funcționalității containerului la minimul necesar.
- Tiny Java Web Server (TJWS) 2.5 – design modular, resurse minimale
- Eclipse Virgo oferă containere web modulare, bazate pe OSGi, implementate utilizând embedded Tomcat și Jetty. Virgo este open source și este disponibil sub licența Eclipse Public License.

9.5.2 containere web comerciale (pe bani)

- Borland Enterprise Server – important în epoca de pionerat a Java EE
- Sun GlassFish Server, de la Sun Microsystems (achiziționat de către Oracle)
- Sun Java System Web Server, de la Sun Microsystems (acum Oracle)
- Sun Java System Application Server (server de aplicații, include și un container web)
- JBoss Enterprise Application Platform (open source)

- JRun, de la Adobe Systems (dezvoltat inițial de către Allaire Corporation)
- LiteWebServer (open source)
- WebLogic Application Server, de la Oracle Corporation (dezvoltat inițial de către BEA Systems)
- Orion Application Server, de la IronFlare
- Caucho's Resin Server (open source)
- ServletExec, de la New Atlanta Communications
- WebSphere Application Server, produs IBM
- NetWeaver, de la SAP
- tc Server (SpringSource)

9.6 descriptorul de desfășurare

Descriptorul de desfășurare (**deployment descriptor**) este un fișier XML (numit, în general - web.xml) care permite particularizarea aplicației web la momentul defășurării. Informații extra pentru desfășurare poate fi conținută în fișier(e) XML extra (precum application.xml) sau sunt conținute în adnotări din fișerele sursă Java.

Scopurile utilizării descriptorului de desfășurare pot fi rezumate astfel:

1. Inițializarea parametrilor pentru servlete, paginin JSP sau Java Server Faces.
2. Declararea servletelor, pagininlor JSP sau Java Server Faces, precizarea claselor de implementare a servletelor, entități JSP precompilate (nume, clase de implementarea, descriere).
3. Mapare (relații) pentru servlete, pagini JSP sau Java Server Faces.
4. Tipurile MIME utilizate de către aplicația web.
5. Elemente de securitate – se pot preciza paginile care cer autentificare precum și diferitele roluri pe care le poate avea un utilizator.
6. Altele, precum paginile în caz de eroare (error pages), paginile de intrare (welcome pages), elemente de configurare a sesiunii.

Prezentăm mai jos un exempleu mic, dar tipic de fișier **web.xml**:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
<web-app>
<!--  Define the Bank 11 ccards Servlets  -->
<servlet>
  <servlet-name>Login</servlet-name>
  <servlet-class>com.bank11.ccards.servlets.LoginServlet
</servlet-class>
</servlet>
</web-app>
```

9.7 accesarea (invocarea) unei aplicații web

Cum accesăm o aplicație web (cum o invocăm)? În principiu, utilizăm un browser. Dar care URL?

URL-ul utilizat de către un client pentru a accesa o aplicație web (implementată ca o componentă web de tip servlet, să zicem) urmează modelul de mai jos:

```
http://hoststring/ContextPath/servletPath/pathInfo
```

1. `hoststring` este de obicei un nume de domeniu sau o adresă web și (în anumite cazuri) un număr de port (port number). De exemplu – `http://localhost:8080`
2. `ContextPath` – numele (logic) al aplicației web. De exemplu, dacă utilizăm Tomcat ca server web, directorul de bază al aplicației este un subdirector al folderului "%TOMCAT_HOME%\webapps". Acest subdirector are (în general) același nume ca și aplicația însăși. Pentru flexibilitate însă, locația directorului de bază poate fi orice (sub)director al folderului "%TOMCAT_HOME%\webapps". Asocierea dintre numele (cunoscut extern) aplicației web și locația directorului de bază al aplicației este făcută într-un element de tip `<context>` în fișierul "%TOMCAT_HOME%\conf\server.xml" file. De exemplu, dacă directorul de bază al aplicației web "/ccards" este "%TOMCAT_HOME%\webapps\bank11\cc", elementul `<context>` corespunzător în fișierul "%TOMCAT_HOME%\conf\server.xml" arată astfel:

   ```
   <context path="/ccards" docbase="bank11/cc" />
   ```

3. `servletPath` – o specificație (url-pattern) asociată unui (nume de) servlet. Acest lucru este făcut într-un element `<servlet-mapping>` din descriptorul de desfășurare web.xml. Un exemplu:

   ```
   <servlet-mapping>
     <servlet-name>watermelon</servlet-name>
     <url-pattern>/fruit/summer/*</url-pattern>
   </servlet-mapping>
   ```

   În acest exemplu, un URL precum:

   ```
   http://host:port/mywebapp/fruit/summer/index.abc
   ```

duce la invocarea servletului **watermelon**. Clasa care implementează acest servlet este specificată într-un element <servlet> din același fișier – web.xml. Ceva de genul:

```
<servlet>
  <servlet-name>watermelon</servlet-name>
  <servlet-class>myservlets.watermelon</servlet-class>
</servlet>
```

4. `pathInfo` – porțiunea rămasă din URL, de obicei un nume de fișier

9.8 servere de aplicații

Un **server de aplicații** (conform https://en.wikipedia.org/wiki/Application_server) este o platformă software care oferă atît facilități pentru crearea de aplicații web cât și un mediu pentru execuția acestora pe un server.

Serverele de aplicații constau din conectori (adaptori) pentru servere web, limbaje de programare, biblioteci pentru suportul execuției, conectori cu servicii de baze de date precum și facilități de administrare necesare pentru desfășurarea, configurarea, administrarea și conectarea acestor componente.

În cazul serverelor de aplicații Java, serverul se comportă ca o mașină virtuală extinsă pentru execuția aplicațiilor, facilitarea accesului la servicii de baze de date precum și conexiuni cu clienții web.

Java EE definește un set de API-uri și funcționalități (features) pentru serverele de aplicații Java.

Infrastructura Java EE este partiționată în containere.

- container web
- container EJB

- container JCA (Java EE Connector Architecture)
- provider JMS (Java Message Service)

O listă cu principalele servere de aplicații Java comerciale:

- Weblogic – de la Oracle, dezvoltat inițial de Bea Systems
- WebSphere – de la IBM
- NetWeaver – de la SAP AG
- Jetty – an Eclipse project
- Jboss Enterprise Application Platform – de la Red Hat
- ColdFusion – de la Adobe Systems

O listă cu principalele servere de aplicații Java ne-comerciale:

- Glassfish – menținut de comunitatea Glassfish, dezvoltat inițial de Sun, apoi – Oracle
- Payara – o derivație Glassfish, de la C2B2
- Geronimo – de la Apache
- WildFly – o derivație JBoss, o divizie Red Hat
- JOnAS – de la Object Web
- Tomcat

Cota de piață pentru fiecare din aceste servere de aplicații Java poate fi consultată, de exemplu, la link-ul - https://plumbr.io/blog/java/most-popular-java-application-servers-2017-edition .

Primele poziții sunt ocupate, în ordine, de:

- Tomcat – 63.8% (dacă e gratis, cu plăcere)
- Jboss/WildFly – 13.8%
- Jetty – 9.0%
- GlassFish – 5.6%
- WebLogic – 4.5%

chapter 10 servlete

Servletele sunt programe mici, independente de platformă, care extind funcționalitatea unui server web sau a unui server de aplicații. Un servlet nu comunică direct cu clientul, ci prin intermediul unui container web. Servletul există doar în interiorul acestui container, care îi oferă un mediu de execuție. Cotainerele web sunt implementate de diferite firme, în general, ca parte a unui server web sau de aplicații.

10.1 specificații

Din punct de vedere tehnic, un servlet este o clasă Java care extinde clasa `GenericServlet` sau (în marea majoritate a cazurilor) clasa `HttpServlet`. În teorie, servletele pot comunica prin intermediul oricărui protocol client-server, dar în marea majoritate a cazurilor, o fac prin intermediul HTTP.

API-ul pentru servlete, definit prin intermediul specificației Servlet, oferă un cadru simplu pentru dezvoltarea de aplicații web pe servere web sau de aplicații.

Prima specificație Servlet (versiunea 1.0) a fost finalizată de către Sun Microsystems în iunie 1997. A continuat cu versiunile 2.0, 2.1, 2.2 în următorii doi ani. Începând cu versiunea 2.3 (apărută în august 2001), specificația Servlet este parte a Java Community Process.

Versiunea 3.1 (apărută în mai 2013 ca JSR 340) este parte a SDK-ului pentru Java EE 7.

Versiunea 3.1 beneficiază de suportul Tomcat 8.0, Glassfish 4.0 și Netbeans 7.3.1, continuând cu versiunile ulterioare ale acestor produse.

Versiunea curentă (4.0 - parte a Java EE 8, JSR 369) este în curs de finalizare, o primă versiune apărând în sept. 2017.

10.2 pachete și clase din API-ul pentru servlete

API-ul pentru servlete constă din 2 pachete, care sunt parte a Java Platform SDK, Enterprise Edition.

Aceste pachete sunt:

- javax.servlet
- javax.servlet.http

Clasele și interfețele definite în pachetul `javax.servlet` sunt independente de protocolul de comunicare, în timp ce cel de-al doilea - `javax.servlet.http`, conține clase și interfețe specifice protocolului HTTP.

Clasele și interfețele ale API-ului pentru servlete Java pot fi împărțite în mai multe categorii, și anume:

- implementarea servletelor
- configurarea servletelor
- excepții pentru servlete
- cereri și răspunsuri
- urmărirea sesiunilor
- contextul servletelor
- colaborarea între servlete
- diverse

10.3 interfața Servlet

Interfața `Servlet` face parte din pachetul `javax.servlet`. Această interfață declară următoarele metode:

```
public void init(ServletConfig config) throws ServletException;
public void service(ServletRequest req, ServletResponse resp) throws
   ServletException, IOException;
public void destroy() throws ServletException;
public ServletConfig getServletConfig();
public String getServletInfo();
```

După instanțierea servletului, containerul web apelează metod `init()` a servletului. Metoda execută instrucțiuni care țin de inițializarea servletului, precum încărcarea de drivere JDBC, conectarea la serviciul de baze de date, etc. Specificația pentru servlete asigură că această metodă este apelată o singură dată.

În momentul sosirii unei cereri din partea unui client, containerul apelează metoda `service()` a servletului, al cărei scop este procesarea cererii. Metoda (sau una din derivatele sale) are două argumente, argumente care implementează interfețele `ServletRequest` și, respectiv, `ServletResponse`.

După procesarea cererii (cererilor) și atunci când containerul web nu mai consideră necesară această instanță a servletului (conform logicii configurate), containerul apelează metoda `destroy()` a servletului și îl dezinstanțiază.

Mai multe despre ciclul de viață al unui servlet, într-un alt paragraf.

10.4 clasa GenericServlet

```
public abstract class GenericServlet implements
   Servlet, ServletConfig, Serializable
```

Această clasă oferă o implementare elementară a interfeței `Servlet`. Deoarece această clasă implementează și interfața `ServletConfig`, programatorul poate apela metodele interfeței `ServletConfig` direct, fără a fi nevoie să obțină mai întâi un obiect de tip `ServletConfig`. Toate clasele care extind clasa `GenericServlet` trebuie să ofere o implementare pentru metoda `service()`.

Methode specifice acestei clase:

```
public void init()
public void log(String msg)
public void log(String msg, Throwable t)
```

10.5 clasa HttpServlet

Este foarte probabil că va fi singura implementare a interfeței `Servlet` pe care o veți folosi vreodată pentru a procesa o cerere a clientului pentru aplicația web. API-ul pentru servlete oferă o astfel de clasă, și anume clasa `HttpServlet`.

```
public abstract class HttpServlet extends GenericServlet implements
Serializable
```

Clasa `HttpServlet` poferă o implementare specifică pentru protocolul HTTP a interfeței `Servlet`. Această clasă abstractă specifică următoarele metode:

```
public void service(ServletRequest req, ServletResponse resp)
public void service(HttpServletRequest req, HttpServletResponse resp)
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
protected void doPost(HttpServletRequest req,
   HttpServletResponse resp)
protected void doDelete(HttpServletRequest req,
   HttpServletResponse resp)
protected void doOptions(HttpServletRequest req,
   HttpServletResponse resp)
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
```

10.6 interfața ServletConfig

Această interfață abstractizează informații de configurare a servletului, precum:

- parametri de inițializare (ca perechi nume_parametru-valoare_parametru)
- numele servletului
- un obiect de tip ServletContext, conținând informații despre containerul web

Interfața `ServletConfig` specifică următoarele metode:

```
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public ServletContext getServletContext()
public String getServletName()
```

10.7 interfața ServletContext

Contextul unui servlet definește imaginea pe care o are servletul despre aplicația web și oferă acces la resursele comune tuturor servletelor aplicației web. Fiecare context de servlet este localizat într-un anume folder de pe serverul web. Desfășurarea unei aplicații web implică adăugarea unui element de tip `<context>` specific aplicației, element care asociază numelui aplicației directorul său de bază. Acest lucru are loc într-un fișier de configurare la nivel de server, numit `server.xml`.

Există un singur `ServletContext` pentru întreaga aplicație web și toate componentele aplicației web au acces la acest obiect. Contextul servletului este creat de către containerul în care servletul este desfășurat.

Interfața `ServletContext` abstractizează contextul unei aplicații web. Pentru a obține o referință la un obiect de acest tip, poate fi obținută prin invocarea metodei `getServletContext()` a obiectului

`HttpServlet`.

Metode principale:

```
public String getMIMEType(String fileName)
public String getResource(String path)
public ServletContext getContext(String urlPath)
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public Object getAttribute(String name)
public Enumeration getAttributeNames()
public void setAttribute(String name, Object attr)
public String removeAttribute(String name)
```

10.8 interfața HttpServletRequest

```
public interface HttpServletRequest extends ServletRequest
```

Această interfață conține metode specifice HTTP. Dacă cunoaștem structura unui cereri HTTP, putem înțelege semnificația principalelor metode ale acestei interfețe. Iată unele din ele:

```
public Cookie[] getCookies()
public long getDateHeader()
public String getHeader(String name)
public Enumeration getHeaders(String name)
public Enumeration getHeaderNames()
public String getContextPath()
public String getPathInfo()
public String getQueryString()
public String getRemoteUser()
```

10.9 interfața HttpServletResponse

Această interfață definește metode ce permit servletului să construiască răspunsuri în format HTTP la cererile clientului. Iată cele mai importante dintre ele:

```
public ServletOutputStream getOutputStream()
public PrintWriter getWriter()
public void setContentLength(int len)
public void setContentType(String type)
public void setBufferSize(int size)
```

```
public int getBufferSize()
public void flushBuffer()
```

Scrierea efectivă a conținutului răspunsului este făcută de obiectul de tip `PrintWriter`.

10.10 ciclul de viață al unui servlet

În general, o instanță a unui servlet trece prin următoarele faze:

- instanțiere
- inițializare
- serviciu
- ștergere
- ne-disponibil

Containerul web crează o instanță a servletului ca prim răapuns la o cerere adresată acestuia sau la pornirea containerului. În mod normal, containerul crează o singură instanță care procesează toate cererile. Dacă servletul nu implementează interfața `javax.servlet.SingleThreadModel`, cererile concurente (simultane) sunt procesate în mai multe fire de execuție.

O altă variantă, mai ales în cazul previzionării unui număr mare de cereri, este cea în care containerul web crează la pornire un set (pool) de instanțe ale servletului.

După instanțiere, containerul apelează metoda `init()` a servletului, metodă care realizează inițializarea servletului. Deseori, această metodă constă în obținerea parametrilor de inițializare, încărcarea driverelor JDBC, conectarea la servicii de baze de date, etc.

Metoda `service()` sau una din variantele sale (`processRequest()`, `doGet()`, `doPost()`, etc.) este cea care implementează funcționalitatea primară a servletului. Într-un scenariu tipic, în această fază servletul preia parametrii trimiși prin cererea clientului, crează interogări SQL, le execută și, în baza datelor returnate crează un răspuns (sub forma unui fișier HTML, XML, etc.) care este trimis clientului.

Containerul web se asigură că metoda `service()` nu va fi apelată înainte de completarea metodei `init()`. Totodată, metoda `destroy()` va fi apelată doar înainte de dezinstanțierea servletului.

10.11 o aplicație web ce utilizează un servlet

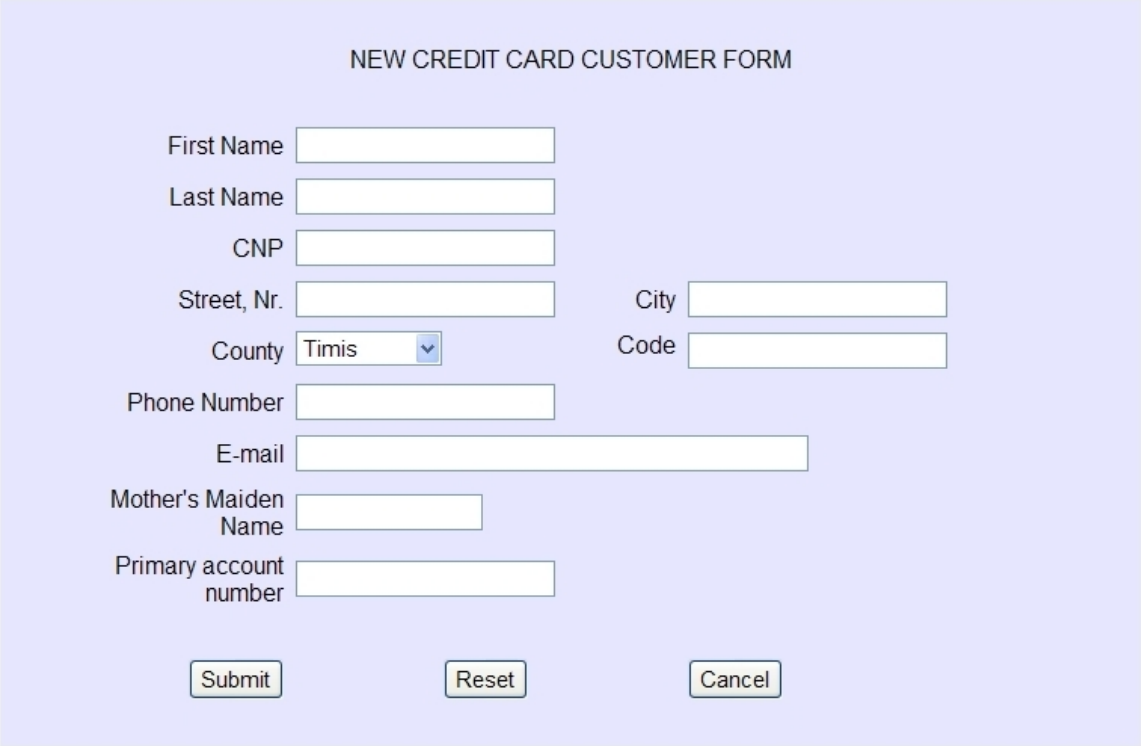
Această aplicație, constă, în principiu, din:

- un fișier html – OAM.html, care este și punctul de intrare al aplicației
- un servlet – OAMServlet, care procesează datele trimisa prin forma OAM.html
- descriptorul de desfășurare – web.xml

10.11.1 forma OAM.html

Acest fișier este punctul de intrare al aplicației web. El poate fi accesat prin specificarea adresei aplicației web în browser. Aceasta constă din adresa serverului web, urmată (dacă portul utilizat nu este cel implicit – 80) de numărul de port, apoi de numele (identificatorul literal) aplicației web (așa cum este specificat în elementul `<context>` corespunzător din fișierul `sever.xml`).

Fișierul OAM.html este primul afișat, deoarece este specificat ca `welcome-file` în descriptorul de desfășurare `web.xml`.



După ce datele au fost introduse, clientul apasă butonul de Submit, un script de validare este executat și în cazul în care datele sunt validate, browserul crează o cerere HTTP, conținând parametrii introduși și care este transmisă pentru procesare servletului OAMServlet, deoarece adresa servletului este specificată ca valoare a atributului ACTION al formei.

10.11.2 servletul OAM

Servletul OAMServlet procesează cererea trimisă de către browser-ul web. Prezentăm mai jos o variantă simplificată a fișierului Java OAMServlet.java care implementează servletul:

```
package com.bank11.ccards.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class OAMServlet extends HttpServlet {
```

```
        /**
         * Processes requests for both HTTP GET and POST methods.
         * @param request servlet request
         * @param response servlet response
         * @throws ServletException if a servlet-specific error occurs
         * @throws IOException if an I/O error occurs
         */
        Connection myConn=null;
        boolean isThere;

        @Override
        public void init(ServletConfig config) throws ServletException{
            super.init(config);
            String driverName = getInitParameter("jdbcDriver");
            String connURL = getInitParameter("dbServer");
            try {
                Class.forName(driverName).newInstance();
                myConn = DriverManager.getConnection
             (connURL,"tw2017","tw2017");
            }catch(Exception e){
                e.printStackTrace();
            }
        }

        protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
            isThere=true;
            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out = response.getWriter();
            String outMess = "Connection established";

            if(myConn == null){
                outMess="Nu exista conexiune ";
            }else{
                String fName=request.getParameter("fname");
                String lName=request.getParameter("lname");
                String cnp=request.getParameter("cnp");
                String street=request.getParameter("street");
```

```
                String city=request.getParameter("city");
                String county=request.getParameter("county");
                String code=request.getParameter("code");
                String phoneNumber=request.getParameter("phoneNo");
                String email=request.getParameter("email");
                String mName=request.getParameter("mname");
                String pryAccount=request.getParameter("primaryacc");
                try{
                   String query="select * from TW2010.customers WHERE cnp="+cnp;
                   System.out.println(query);
                   java.sql.Statement stmtQuery = myConn.createStatement();
                   ResultSet resultQuery = stmtQuery.executeQuery(query);
                   if(!resultQuery.next()){
                       isThere=false;
                   }
                   if(isThere==false){
                        String sql="INSERT INTO TW2010.CUSTOMERS
(FIRST_NAME,LAST_NAME,CNP,STR_NUM,CITY,COUNTY,CODE,PHONE_NUM,E_MAIL,MM_NAME,PRIM
ARY_ACCT) VALUES
('"+fName+"','"+lName+"',"+cnp+",'"+street+"','"+city+"','"+county+"',"+code+",'
"+phoneNumber+"','"+email+"','"+mName+"',"+pryAccount+")";

                       java.sql.Statement stmt = myConn.createStatement();
                       int result = stmt.executeUpdate(sql);
                       if(result!=-1){
                           System.out.println(outMess+="<br />Record added");
                       }else{
                           System.out.println(outMess+="<br />The record
couldn't be added");
                       }
                    }
                }catch(Exception e){
                    e.printStackTrace();
            }
        }

            try {
                 // output your page here
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Servlet OAM</title>");
```

```
                out.println("</head>");
                out.println("<body>");
                out.println("<h1>Servlet OAM at " + request.getContextPath () +
"</h1>");
                if(isThere==false){
                   out.println(outMess);
                }else{
                    out.println(outMess+="<br />The CNP already exists in the
database");
                }
                out.println("</body>");
                out.println("</html>");

            }catch(Exception e){e.printStackTrace();
            }finally {
                out.close();
            }
        }
```

În metoda `init()` pare loc încărcarea driverului jdbc și se deschide o conexiune cu baza de date. Atît numele driverului jdbc pentru sistemul de baze de date Derby cât și URL-ul conexiunii sunt parametri specificați în descriptorul de desfășurare web.xml.

În partea de procesare a cererii, parametrii cererii (care vin din câmpurile formei OAM.html) sunt recuperați și sunt utilizați pentru crearea unei interogări a bazei de date.

Această interogare este executată și, dacă este cazul, se efectuează inserarea unui articol în tabela CUSTOMERS a bazei de date.

În final, servletul generează un răspuns HTTP care are în partea de conținut un fișier HTML în care se comunică clientului starea cererii sale.

10.11.3 descriptorul de desfășurare

Descriptorul de desfășurare (fișierul web.xml) este arătat mai jos.

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
        <servlet>
          <servlet-name>OAM</servlet-name>
          <servlet-class>com.bank11.ccards.servlets.OAMServlet</servlet-class>
          <init-param>
            <param-name>jdbcDriver</param-name>
            <param-value>org.apache.derby.jdbc.ClientDriver</param-value>
```

```
      </init-param>
      <init-param>
        <param-name>dbServer</param-name>
        <param-value>jdbc:derby://localhost:1527/ccards</param-value>
      </init-param>
    </servlet>
    <servlet-mapping>
      <servlet-name>OAM</servlet-name>
      <url-pattern>/OAM</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>
        30
      </session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>OAM.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Descriptorul de desfășurare conține câteva elemente de interes.

- Elementele `<servlet-name>` și `<servlet-class>` asociază servletului identificat prin numele său (OAM) clasa java care îl implementează: `om.bank11.ccards.servlets.OAMServlet`
- Avem doi parametri de inițializare: `jdbcDriver`, care identifică driver-ul jdbc utilizat, precum și `dbServer`, care identifică URL-ul serviciului de baze de date:
- `welcome-file-list`, care specifică punctul (punctele) de intrare al aplicației web, în cazul nostru – fișierul static OAM.html

chapter 11 JDBC

11.1 ce este JDBC?

JDBC este un acronim pentru Java Data Base Connectivity și este versiunea Java a ODBC (Open Data Base Connectivity). Oferă un API pentru accesul bazelor de date relaționale care oferă suport pentru limbajul de interogare SQL. Abstractizează detaliile specifice de acces pentru diferiții producători de servicii de baze de date și oferă suport pentru cele mai comune funcții de acces la bazele de date. Și, mai ales, codul Java astfel obținut este independent de serviciul de baze de date utilizat.

Prima versiune a specificației JDBC datează din feb. 1997, ca parte a JDK (Java Development Kit) 1.1. După aceasta, specificația JDBC a fost parte a JSE (Java Standard Edition). Începând cu versiunea 3.1, evoluția JDBC este parte a Java Community Process. Versiunea majoră curentă (4.0) a JDBC este definită în JSR (Java Specification Request) 221.

Versiunile minore de la 4.1 la 4.3 (cea curentă în oct. 2018) sunt specificate în versiuni de mentenanță (maintenance releases) ale aceluiași JSR 221. Versiunea 4.1 e inclusă în Java SE 7, versiunea 4.2 e inclusă în Java SE 8 iar versiunea 4.3 e inclusă în Java SE 9.

API-ul pentru JDBC 4.3 (ca și toate celelalte versiuni, de altfel) constă din 2 pachete:

1. pachetul `java.sql`
2. pachetul `javax.sql`, care oferă facilități în special pentru partea de server

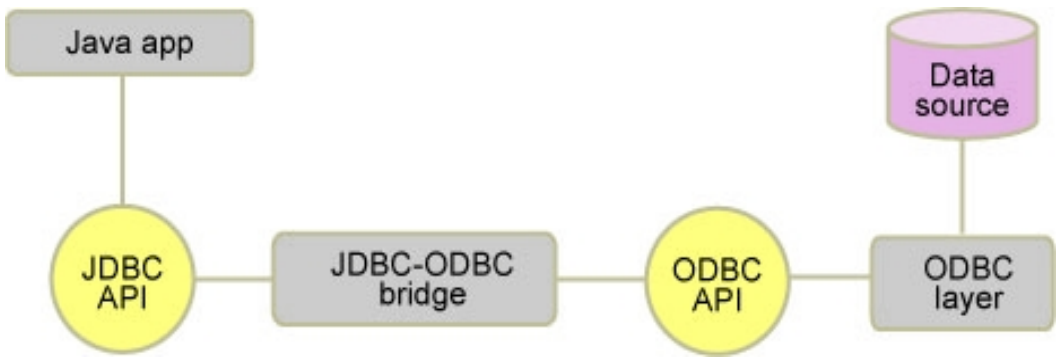
API-ul pentru JDBC oferă acces programatic la bazele de date din aplicații scrise în Java. Acest API permite accesul la o mare varietate de sisteme de baze de date, de surse de date sau sisteme legacy.

11.2 drivere jdbc

Fiecare producător de sisteme de baze de date are propria sa versiune de API pentru accesarea bazelor de date. Un driver (conector, adaptor) JDBC este un produs soft (middleware) care translatează apelurile JDBC în apeluri specifice unui anumit sistem. Aceste drivere pot fi clasificate în patru categorii majore. O nouă categorie de drivere cu performanțe superioare poate fi și ea luată în considerare.

Tip 1. **Podul (bridge) JDBC – ODBC**

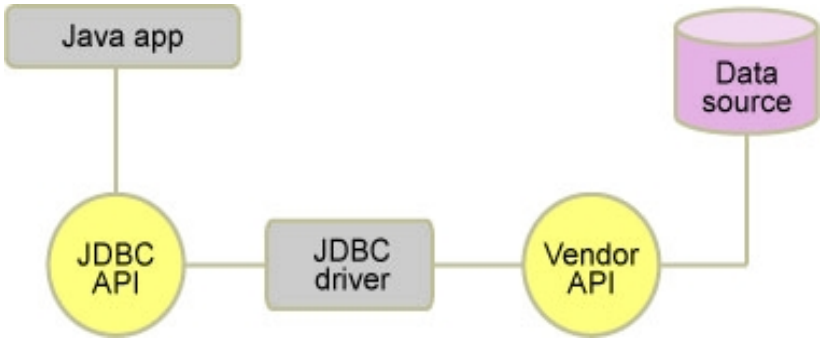
Acets driver translatează apeluri JDBC în apelurile ODBC corespunzătoare. Această soluție este ineficientă, datorită nivelelor multiple de indirectare precum și datorită limitărilor impuse nivelului JDBC de către cadrul ODBC.



JDK-ul standard include toate clasele pentru acest bridge - `sun.jdbc.odbc.JdbcOdbcDriver` .

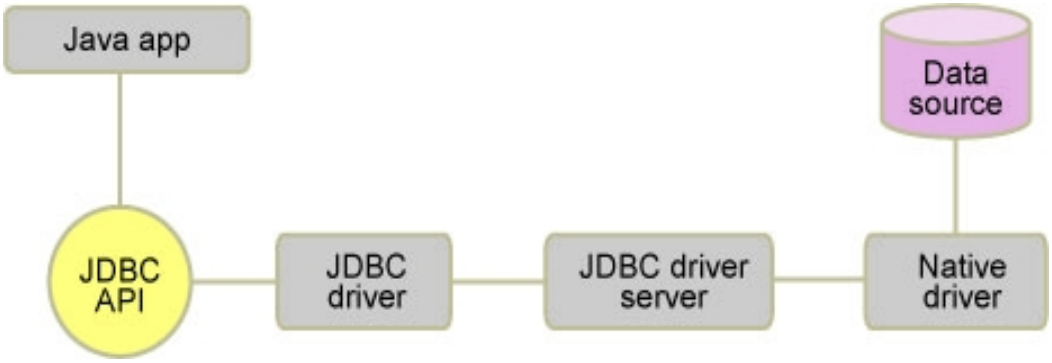
Tip 2. **Driver Parțial Java, Parțial Nativ**

Driverele din această categorie folosesc o combinație între o implementare Java și un API specific unui anumit sistem de baze de date. Driverul translatează apelurile specifice JDBC în apeluri specifice sistemului de baze de date. Serviciul de baze de date returnează rezultatul apelării căatre acest API, care le direcționează către driverul JDBC. Este mult mai rapid ca un driver de tip 1, fiindcă elimină un nivel de indirectare.



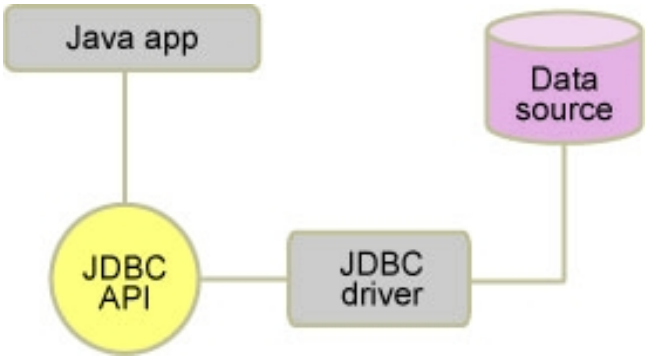
Tip 3. **Server intermediar de acces la baze de date (translatare Java + Middleware)**

Driverele de tip 3 sunt drivere Java pentru JDBC care sunt utilizate de către serverele de baze de date care acționează ca nivel intermediar între clienți multipli și servere de baze de date multiple. Aplicația client trimite un apel JDBC printr-un driver JDBC către serverele intermediare. Aceste servere translatează apelul în apeluri native care administrează conexiunea propriu-zisă. Acest tip de drivere este implementat de câteva servere de aplicații, precum WebLogic sau Inprise Application Server.



Tip 4. **Drivere Java native (pure)**

Acestea sunt cele mai eficiente drivere. Apelurile JDBC sunt convertite în apeluri directe de rețea utilizând protocoalele oferite de către producător (vendor). Toate firmele majore în domeniul sistemelor de baze de date oferă drivere de tip 4 pentru sistemele lor.



Type 5. **drivere de înaltă funcționalitate cu performanțe superioare**

Un tip mai recent, include drivere precum **DataDirect Connect for JDBC drivers,** oferind funcționalitate avansată și performanțe superioare celorlalte tipuri de drivere.

11.3 pachetul java.sql

Acest pachet con'ine API-ul de bază JDBC. O listă completă a claselor și interfețelor ale acestui pachet pot fi găsite în cea mai recentă specificație JDBC (4.2 pentru Java SE 8 și 4.3 pentru Java SE 9). Documentele conținând aceste specificații sunt versiunile de mentenanță 2 și respectiv 3 ale JSR 221 și pot fi găsite la link-ul http://jcp.org/en/jsr/detail?id=221.

Dintre cele peste 80 de clase și interfețe ale specificației JDBC 3.x, să le amintim pe cele mai importante:

```
java.sql.Array
java.sql.Blob
java.sql.CallableStatement
java.sql.Clob
java.sql.Connection
java.sql.Date
java.sql.Driver
java.sql.DriverManager
java.sql.PreparedStatement
java.sql.ResultSet
java.sql.ResultSetMetaData
java.sql.SQLData
java.sql.SQLDataException
java.sql.SQLException
java.sql.SQLInput
java.sql.SQLOutput
java.sql.SQLPermission
java.sql.SQLXML
java.sql.SQLWarning
java.sql.Statement
```

```
java.sql.Struct
java.sql.Time
java.sql.Timestamp
java.sql.Types
java.sql.Wrapper
```

Lista de mai jos conține toate clasele și interfețele noi sau modificate în versiunile 4.x.

```
java.sql.Blob
java.sql.CallableStatement
java.sql.Clob
java.sql.ClientinfoStatus
java.sql.Connection
java.sql.DatabaseMetaData
java.sql.NClob
java.sql.PreparedStatement
java.sql.ResultSet
java.sql.RowId
java.sql.RowIdLifeTime
java.sql.SQLClientInfoException
java.sql.SQLDataException
java.sql.SQLException
java.sql.SQLFeatureNotSupportedException
java.sql.SQLInput
java.sql.SQLIntegrityConstraintViolationException
java.sql.SQLInvalidAuthorizationSpecException
java.sql.SQLNonTransientConnectionException
java.sql.SQLNonTransientException
java.sql.SQLOutput
java.sql.SQLSyntaxErrorException
java.sql.SQLTimeoutException
java.sql.SQLTransactionRollbackException
java.sql.SQLTransientConnectionException
java.sql.SQLTransientException
java.sql.SQLXML
java.sql.SQLWarning
java.sql.Statement
java.sql.Types
java.sql.Wrapper
javax.sql.CommonDataSource
javax.sql.StatementEvent
```

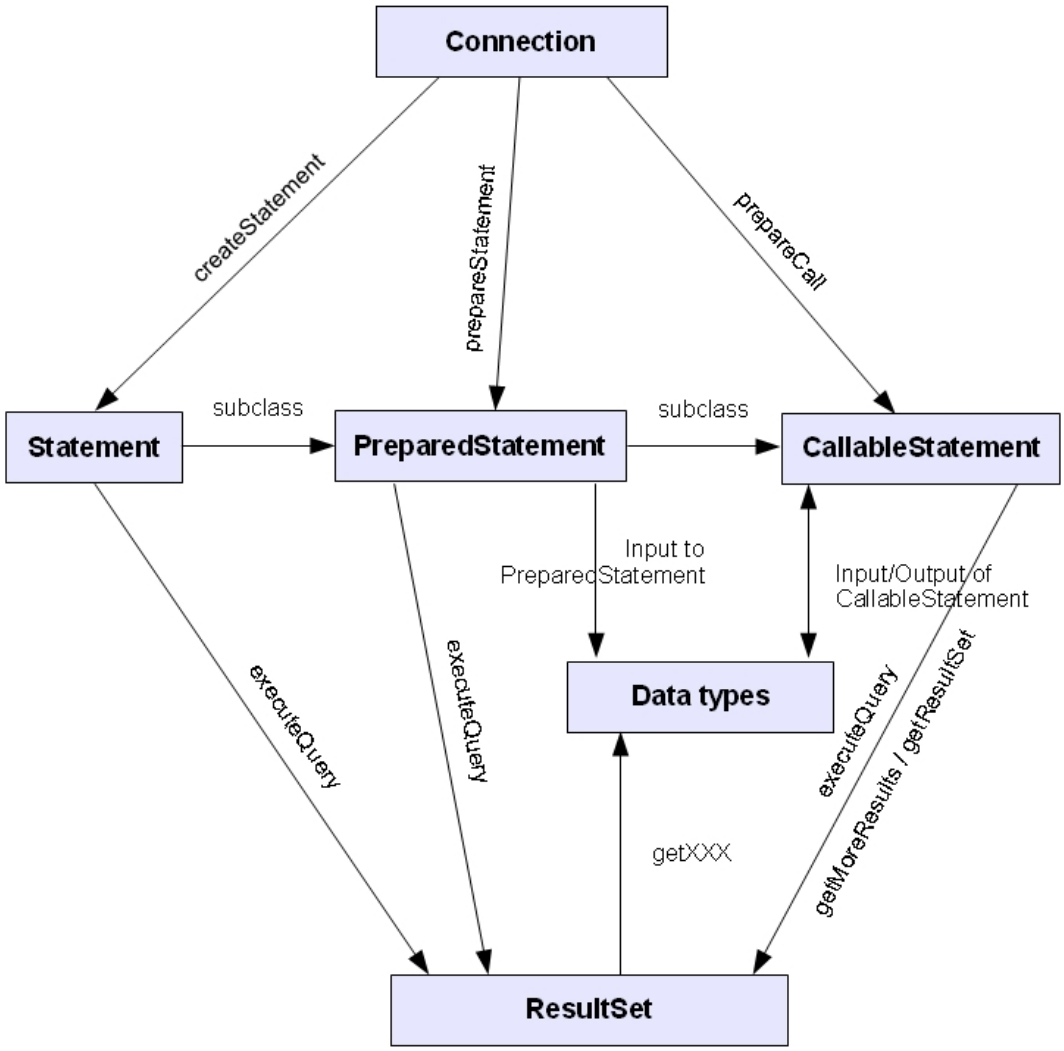
`javax.sql.StatementEventListener`

11.4 schema de interacțiune în pachetul java.sql

Figura de mai jos ne arată interacțiunile și relațiile dintre clasele și interfețele majore din pachetul java.sql.

Pașii principali în accesarea unei baze de date sunt:

1. încărcarea driverului JDBC corespunzător bazei de date
2. stabilirea unei conexiuni cu baza de date
3. interogarea bazei de date
4. procesarea setului de date (result set) returnat



11.5 încărcarea driver-ului și conectarea la baza de date

Sunt doi pași principali în conectarea la un serviciu de baze de date. Primul din acești pași este:

Încărcarea driverului JDBC pentru baza de date.

Un driver (conector) JDBC este specificat prin numele driverului. câteva exemple de nume de drivere JDBC:

- `com.ibm.db2.jdbc.app.DB2Driver`
- `oracle.jdbc.driver.OracleDriver`
- `com.mysql.jdbc`
- `org.gjt.mm.mysql`
- `com.sybase.jdbc.SybDriver`
- `sun.jdbc.odbc.JdbcOdbcDriver`
- `weblogic.jdbc.mssqlserver4.Driver`
- `org.postgresql.Driver`

Prezentăm acum prima metodă (oarecum obscură) de încărcare a driverului JDBC:

```
import java.sql.*;
import java.util.*;

try {
   Class.forName("org.gjt.mm.mysql.Driver").newInstance();
} catch (Exception e) {
   // driver not found
   e.printStackTrace();
}
```

Notă. Fiecare obiect în Java are o clasă (aparține unei clase) precum și un obiect de tip Class corespunzător, care conține metadate despre el, date accesibile în timpul execuției.

Locația serviciului de baze de date este specificată de URL-ul său (cunoscut sub numele URL-connection). URL-ul are trei părți, separate prin „:”

```
jdbc:<subprotocol>:subname
```

- jdbc este numele **protocolului** (de fapt, singurul nume acceptat în acest context).
- **sub-protocolul** este utilizat pentru identificarea driverului JDBC, conform specificației producătorului
- **subname** – sintaxa acestui câmp este specifică producătorului și ajută la identificare

câteva exemple de URL-uri pentru servicii de baze de date prin JDBC:

- jdbc:sybase:localhost:2025?ServiceName=<databaseName>
- jdbc:derby:net://<host>:1527/<databaseName>
- jdbc:db2://db2.bank11.com:50002/ccards
- jdbc:oracle:thin:@loclahost:1521:ORCL
- jdbc:postgresql://<host>:5432/<databaseName>

Al doilea pas în conectarea la o bază de date este **deschiderea conexiunii**, utilizând URL-ul conexiunii:

Exemplu de cod care arată cum se realizează acest lucru:

```
String connURL = "jdbc:mysql://localhost:3306/ccards";
String user = "root";
String passwd = "root"
Connection conn = DriverManager.getConnection(connURL,
   user, passwd);
```

Fiindcă tocmai am utilizat clasa `DriverManager`, să avem o privire mai detaliată asupra sa în paragraful următor.

11.6 clasa DriverManager

Această clasă aparține pachetului `javax.sql` și oferă un nivel de acces comun peste diferitele drivere JDBC. Fiecare asemenea driver utilizat de către aplicație trebuie să fie înregistrat (încărcat) înainte ca `DriverManager` să încerce obținerea unei conexiuni.

O conexiune poate fi obținută invocând metoda `getConnection()` a acestei clase. Există trei versiuni ale metodei `getConnection()`. Le prezentăm mai jos:

```
public static Connection getConnection(String connURL)
   throws SQLException
public static Connection getConnection(String connURL, String user,
String passwd) throws SQLException
public static Connection getConnection(String connURL,
java.util.Properties info) throws SQLException
```

Primele două dintre aceste versiuni nu necesită alte explicații. Pentru ultima versiune, e mai simplu să prezentăm un exemplu:

```
Properties prp = new Properties();
prp.put("autocommit", "true");
prp.put("create", "true");
Connection conn = DriverManager.getConnection(connURL, prp);
```

(Un obiect de tip `Properties` este o colecție de perechi parametru – valoare_parametru.)

11.7 interfața Connection

Interfața `Connection` este parte a pachetului `javax.sql`. Principala sa funcție, din punctul de vedere al programatorului, este de a crea obiecte de tip `Statement`, obiecte care abstractizează interogări SQL.

Odată ce obținem un obiect de tip `Connection`, acesta poate fi utilizat în multipe scopuri, dar noi ne vom interesa în special de acela de a crea interogări (instrucțiuni) SQL. Cele mai importante metode pentru a crea astfel de obiecte:

```
Statement createStatement() throws SQLException

Statement createStatement(int resultSetType, int resultSetConcurrency)
   throws SQLException

Statement createStatement(int resultSetType, int resultSetConcurrency,
int resultSetHoldability)

PreparedStatement prepareStatement(String sql)

   throws SQLException

CallableStatement prepareCall(String sql)

   throws SQLException
```

11.8 interfețe de tip statement

Obiectele pe care le-am întâlnit în secțiunea precedentă, și anume: `Statement`, `PreparedStatement` and `CallableStatement` abstractizează interogări SQL normale, interogări pregătite (prepared statements) și, respectiv, proceduri stocate.

Interfața `Statement` are (printre altele) următoarele metode:

1. metode pentru execuția interogărilor:
   - `execute()`
   - `executeQuery()`
   - `executeUpdate()`

2. metode pentru batch-uri:
   - `addBatch()`
   - `executeBatch()`
   - `clearBatch()`

3. metode pentru mărimea și direcția achizițiilor (fetches):
   - `setFetchSize()`
   - `getFetchSize()`
   - `setFetchDirection()`

- `getFetchDirection()`

4. metode pentru obținerea setului curent de rezultate (result set):
   - `getResultSet()`

5. metode pentru tipul setului de rezultate și de acces concurent al acestuia:
   - `getResultSetConcurrency()`
   - `getResultSetType()`

6. alte metode:
   - `setQueryTimeout()`
   - `getQueryTimeout()`
   - `setMaxFieldSize()`
   - `getMaxFieldSize()`
   - `cancel()`
   - `getConnection()`

Iată un exemplu tipic de utilizare a obiectelor de tip `Statement`:

```
Statement stmt = conn.createStatement();
String sqlString = "CREATE TABLE customer ...";
stmt.executeUpdate(sqlString);
```

11.9 interfața ResultSet

Rezultatul unei interogări prin intermediul unui obiect de tip `Statement` este un obiect de tip `java.sql.ResultSet`, obiect ce este disponibil programatorului pentru accesul la datele returnate și procesarea acestora. Un `ResultSet` este în principiu un set de articole (sau părți ale acestora) din baza de date. Interfața `ResultSet` ieste implementată de producătorii de drivere.

Metode pentru obținerea de date:

- `getAsciiStream()`
- `getBoolean()`
- `getDate()`
- `getInt()`
- `getShort()`
- `getTimeStamp()`
- `getBinaryStream()`
- `getBytes()`
- `getFloat()`

- `getObject()`
- `getTime()`
- `getString()`
- `getByte()`
- `getDouble()`
- `getLong()`
- `getBigDecimal()`
- `getMetaData()`
- `getClob()`
- `getWarnings()`
- `getBlob()`

Cele mai multe din aceste metode cer ca argument indexul coloanei de interes din tabelă (care începe de la 1, nu de la 0) sau numele coloanei.

Utilizarea acestor metode de obținere presupun cunoașterea a priori a tipului și a indexului unei anume coloane din tabelă. Dar dacă nu avem aceste cunoștințe? Din fericire, toate datele relevante privind schema bazei de date sunt accesibile prin intermediul interfeței `ResultSetMetaData`. Invocarea metodei `getMetaData()` a unui obiect `ResultSet` returnează un obiect de tip `ResultSetMetaData`.

Iată cele mai importante metode specificate de interfața `ResultSetMetaData`:

- `getCatalogName()`
- `getTableName()`
- `getSchemaName()`
- `getColumnCount()`
- `getColumnName()`
- `getColumnLabel()`
- `getColumnType()`
- `getColumnTypeName()`
- `getColumnClassName()`
- `getColumnDisplaySize()`
- `getScale()`
- `getPrecision()`
- `isNullable()`
- `isCurrency()`
- `isSearchable()`
- `isCaseSensitive()`
- `isSigned()`
- `isAutoIncrement()`
- `isReadOnly()`
- `isDefinitelyWritable()`

11.10 exemple de selectare date

```
// DisplayServlet.java
package com.bank11.ccards.servlets;

import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class DisplayServlet extends HttpServlet {
  Connection conn;

  // Initializes the servlet
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    String driverName = "com.mysql.jdbc.Driver";
    try {
      Class.forName(driverName).newInstance();
    }
    catch(ClassNotFoundException e) {
      e.printStackTrace();
    }

    String connURL="jdbc:mysql://localhost:3306/ccards";
    try {
      conn=DriverManager.getConnection(connURL,"root","root");
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
  }

  // Destroys the servlet.
  public void destroy() {
  }

  // Processes requests for both HTTP GET and POST methods.
  protected void processRequest(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, java.io.IOException {
   String theCode = req.getParameter("CODE");
```

```
      String sql = "SELECT FIRST_NAME, LAST_NAME, ACCOUNT_NUM from CUSTOMERS
where CNP="+theCode+";";
       try {
         Statement stmt = conn.getStatement();
         ResultSet rs = stmt.executeQuery(sql);

         while(rs.next()) {
           String firstName = rs.getString("FIRST_NAME");
           String lastName = rs.getString("LAST_NAME");
           BigDecimal accountNum = rs.getBigDecimal("ACCOUNT_NUM");
         }
       } catch (SQLException sqle) {
         sqle.printStackTrace();
       } catch (Exception e) {
         e.printStackTrace();
       }

       resp.setContentType("text/html");
       java.io.PrintWriter out = resp.getWriter();
       // output your page here
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet</title>");
       out.println("</head>");
       out.println("<body>");
       ...
       out.println("</body>");
       out.println("</html>");
       out.close();
     }

     // Handles the HTTP GET method.
     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, java.io.IOException {
       processRequest(req, resp);
     }

     // Handles the HTTP POST method.
     protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, java.io.IOException {
```

```
    processRequest(req, resp);
  }

  // Returns a short description of the servlet.
  public String getServletInfo() {
    return "Short description";
  }
}
```

11.11 exemplu de inserare date

```
Connection conn = null;
Statement stmt = null;
try {
  Class.forName("org.postgresql.Driver").newInstance();
}
catch(ClassNotFoundException e) {
  System.out.println(e.toString());
}
String connUrl = "jdbc:mysql://localhost:3306/ccards";
try {
  conn = DriverManager.getConnection(connUrl, "root", "root");
  stmt = conn.createStatement();
  String query = "INSERT INTO STUDENTS VALUES(1009, 'Vasile',
   'Abrudan', 'IE3')";
  // Update the table
  stmt.executeUpdate(query);
} catch(Exception e) {
  System.out.println(e.toString());
} finally {
  // close the connection
  stmt.close();
  conn.close();
}
```

11.12 interfața PreparedStatement

Dacă o instrucțiune SQL este executată de mai multe ori și dacă formele sale diferă doar în privința datelor concrete conținute, o alegere mai bună este reprezentată de către interfața `PreparedStatement`. Obiectele de acest tip sunt parametrizate și fiecare parametru (de obicei, valoarea unei coloane) este reprezentat de semnul întrebării „?” sau de un asterisc “*”.

Următoarele linii de cod Java oferă un exemplu de utilizare a interfeței `PreparedStatement`. Se presupune că deja avem o conexiune cu baza de date.

```
Statement stmt = conn.createStatement();
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO customer
   VALUES (?, ?, ?)");
stmt.executeUpdate("CREATE TABLE customer (id int, firstName
varchar(32) lastName varchar(24))");
// set parameters for preparedStatement
pstmt.setInt(1, 1021);
pstmt.setString(2, "Vasile");
pstmt.setString(3, "Dumitrascu");
int count = pstmt.executeUpdate();
```

11.13 un alt exemplu de inserare

```
import java.sql.*;

public class InsertRows {
  public static void main(String args[]) {
    String url = "jdbc:mySubprotocol:myDataSource";
    Connection con;
    Statement stmt;
    try {
      Class.forName("myDriver.ClassName");
    } catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }
    try {
      con = DriverManager.getConnection(url, "myLogin",
            "myPassword");
      stmt = con.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
             ResultSet.CONCUR_UPDATABLE);
      ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");

      uprs.moveToInsertRow();
      uprs.updateString("COF_NAME", "Kona");
      uprs.updateInt("SUP_ID", 150);
      uprs.updateFloat("PRICE", 10.99f);
```

```
        uprs.updateInt("SALES", 0);
        uprs.updateInt("TOTAL", 0);
        uprs.insertRow();
        uprs.updateString("COF_NAME", "Kona_Decaf");
        uprs.updateInt("SUP_ID", 150);
        uprs.updateFloat("PRICE", 11.99f);
        uprs.updateInt("SALES", 0);
        uprs.updateInt("TOTAL", 0);
        uprs.insertRow();
        uprs.beforeFirst();

        System.out.println("Table COFFEES after insertion:");
        while (uprs.next()) {
          String name = uprs.getString("COF_NAME");
          int id = uprs.getInt("SUP_ID");
          float price = uprs.getFloat("PRICE");
          int sales = uprs.getInt("SALES");
          int total = uprs.getInt("TOTAL");
          System.out.print(name + "    " + id + "    " + price);
          System.out.println("    " + sales + "    " + total);
        }
        uprs.close();
        stmt.close();
        con.close();
      } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
      }
    }
}
```

11.14 tipuri de date jdbc și sql și clasele java corespondente

Tip JDBC	*Scop*	*Tip SQL*	*Tip Java*
ARRAY	SQL array	ARRAY	java.sql.Array
BIGINT	64 bit integer	BIGINT	long
BINARY	binary value	none	byte[]
BIT	one bit value	BIT	boolean
BLOB	binary large object	BLOB	java.sql.Blob
CHAR	char string	CHAR	String

Tip JDBC	*Scop*	*Tip SQL*	*Tip Java*
CLOB	character large object	CLOB	java.sql.Clob
DATE	day, month, year	DATE	java.sql.Date
DECIMAL	decimal value	DECIMAL	java.math.BigDecimal
DISTINCT	distinct	DISTINCT	none
DOUBLE	double precision	DOUBLE PRECISION	double
FLOAT	double precision	FLOAT	double
INTEGER	32 bit integer	INTEGER	int
JAVA_OBJECT	stores Java objects	none	Object
LONGVARBINARY	variable length binary val	none	byte[]
LONGVARCHAR	variable length char string	none	String
NULL	null values	NULL	null
NUMERIC	decimal value	NUMERIC	java.math.BigDecimal
OTHER	db specific types	none	Object
REAL	single precision	REAL	float
REF			
SMALLINT	16 bit integer	SMALLINT	short
STRUCT			
TIME	hrs, mins, secs	TIME	java.sql.Time
TIMESTAMP	date, time, nanoseconds	TIMESTAMP	java.sql.Timestamp
TINYINT	8 bit integer	TINYINT	short
VARBINARY	variable length binary value	none	byte[]
VARCHAR	variable length char string	VARCHAR	String

11.15 JDBC Data Sources

În pachetul opțional din JDBC 2.0, interfața `DriverManager` este înlocuită de către interfața `DataSource` ca metodă principală de obținere a conexiunilor la baze de date.

În timp ce interfața `DriverManager` este utilizată în momentul execuției pentru a încărca în mod explicit un driver JDBC, noul mecanism utilizează un serviciu centralizat JNDI (Java Naming and Directory Interface) pentru a localiza un obiect de tip `javax.sql.DataSource`.

Această interfață este o fabrică (factory) pentru crearea de conexiuni la baze de date. Este parte a pachetului `javax.sql`.

Există trei tipuri de implementări pentru această interfață:

1. Implementare de bază – produce un obiect standard de tip `Connection`.
2. Implementare de bazin (pool) de obiecte `Connection` – produce un obiect de tip `Connection`, object care va participa automat într-un bazin (pool) de conexiuni.
3. Implementare de tranzacții distribuite – produce un obiect de tip `Connection` ocare poate fi utilizat

pentru tranzacții distribuite și care aproape întotdeauna participă la un bazin de conexiuni.

Principalele metode:

```
public Connection getConnection() throws SQLException
public Connection getConnection(String user, String pwd) throws SQLException
```

Un exemplu de servlet care utilizează interfața `DataSource`:

```
package com.bank11.ccards.servlets;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.sql.*;

public class TestDataSource extends HttpServlet
{
  private final static String DATASOURCE_NAME = "jdbc/ccards";
  private DataSource theDataSource;
  public void setDataSource(DataSource dataSource) {
    theDataSource = dataSource;
  }
  public DataSource getDataSource() {
    return theDataSource;
  }
  public void init() throws ServletException {
    if (theDataSource == null) {
      try {
        Context env =
         (Context) new InitialContext().lookup("java:comp/env");
        theDataSource = (DataSource) env.lookup(DATASOURCE_NAME);
        if (theDataSource == null)
          throw new ServletException("`" + DATASOURCE_NAME +
               "' is an unknown DataSource");
      } catch (NamingException e) {
        throw new ServletException(e);
      }
```

```
        }
      }
      public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
        ...
      }
    }
```

chapter 12 JSP

12.1 Java Server Pages ca parte a unei aplicații web

O pagină JSP (Java Server Pages) este un fișier standard HTML sau XML care conține noi elemente de scriptare, care permit, în special, inserarea de cod Java.

Atunci când o pagină JSP este încărcată de către containerul web, ea este convertită automat într-un **servlet**. Dacă pagina JSP este modificată, servletul este re-generat.

Specificația curentă (nov. 2018) este 2.3 (din mai 2013) și este versiunea de mentenanță 3 a JSR 245 (care definește versiunea majora JSP 2.0).

Această specificație definește interfețe, clase și excepții, grupate în două pachete: **javax.servlet.jsp** și **javax.servlet.jsp.tagext.**

Pachetul **javax.servlet.jsp** conține un număr de clase și interfețe care definesc și descriu contractul dintre o pagină JSP și mediul său de execuție, oferit de către un container web.

Pachetul **javax.servlet.jsp** definește două interfețe - `JspPage` și `HttpJspPage`. Interfața `HttpJspPage` este interfața pe care trebuie să o satisfacă o clasă generată de procesorul JSP pentru protocolul HTTP. Interfața `JspPage` este o interfață pe care trebuie să o satisfacă o clasă generată de procesorul JSP.

Pachetul **javax.servlet.jsp.tagext** conține clasele și interfețele utilizate pentru definirea bibliotecilor de elemente (Tag Libraries) JSP.

12.2 interfața java.servlet.jsp.JspPage

Această interfață are 2 metode:

```
public void jspInit()
public void jspDestroy()
```

Interfața `javax.servlet.HttpJspPage` are o singură metodă:

```
public void jspService(HttpServletRequest req,
   HttpServletResponse resp) throws ServletException, IOException
```

Implementarea acestei metode este generată de către container și nu de către programator.

12.3 un exemplu de servlet generat din pagină JSP

Ca orice obiect generat automat, așteptați-vă la ceva complex. Așa se întâmplă și în acest caz, când avem o pagină JSP care doar afișează un text simplu.

Iată fișierul .jsp de start.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-
8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Servletul generat arată astfel:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class index_jsp extends
org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
  private static final JspFactory _jspxFactory =
JspFactory.getDefaultFactory();
  private static java.util.Vector _jspx_dependants;
  private org.glassfish.jsp.api.ResourceInjector _jspx_resourceInjector;

  public Object getDependants() {
    return _jspx_dependants;
  }
  public void _jspService(HttpServletRequest request,
HttpServletResponse response)
        throws java.io.IOException, ServletException {

    PageContext pageContext = null;
    HttpSession session = null;
```

```
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
        try {
          response.setContentType("text/html;charset=UTF-8");
          response.setHeader("X-Powered-By", "JSP/2.1");
          pageContext = _jspxFactory.getPageContext(this, request, response,
null, true, 8192, true);
          _jspx_page_context = pageContext;
          application = pageContext.getServletContext();
          config = pageContext.getServletConfig();
          session = pageContext.getSession();
          out = pageContext.getOut();
          _jspx_out = out;
          _jspx_resourceInjector = (org.glassfish.jsp.api.ResourceInjector)
application.getAttribute("com.sun.appserv.jsp.resource.injector");
          out.write("\n");
          out.write("\n");
          out.write("\n");
          out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\"\n");
          out.write("   \"http://www.w3.org/TR/html4/loose.dtd\">\n");
          out.write("\n");
          out.write("<html>\n");
          out.write("    <head>\n");
          out.write("        <meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\">\n");
          out.write("        <title>JSP Page</title>\n");
          out.write("    </head>\n");
          out.write("    <body>\n");
          out.write("        <h1>Hello World!</h1>\n");
          out.write("    </body>\n");
          out.write("</html>\n");
        } catch (Throwable t) {
          if (!(t instanceof SkipPageException)){
            out = _jspx_out;
```

```
            if (out != null && out.getBufferSize() != 0)
              out.clearBuffer();
            if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
            else throw new ServletException(t);
          }
        } finally {
          _jspxFactory.releasePageContext(_jspx_page_context);
        }
      }
```

Un scurt comentariu: clasa HttpJspBase este o clasă implementată de către vendor, a cărei declarare clarifică relația dintre clase și interfețe JSP standard.

```
public abstract class HttpJspBase extends javax.servlet.http.HttpServlet
implements javax.servlet.jsp.HttpJspPage
```

12.4 bean-uri java simple (ordinary java beans)

Este vorba de clase Java care abstractizează, în principal, articole din tabelele unei baze de date. Asta spre deosebire de EJBs (Enterprise Java Beans) care sunt, mai degrabă, o colecție de clase Java distribuite.

Un bean Java este o clasă care:

- implementează interfața **java.io.Serializable**
- oferă un constructor fără argumente
- pentru fiecare din proprietățile sale, oferă metode de get și de set
- implementează un mecanism de detectare a modificării proprietăților

câteva cuvinte despre **serializare**. Serializarea unui obiect (instanță a unei clase) înseamnă conversia stării acestuia într-un șir de octeți (bytes stream), conversie care să permită conversia în sens invers într-o copie a obiectului. Un obiect Java este **serializable** dacă clasa sa sau oricare din super-clasele sale implementează fie interfața `java.io.Serializable` sau sub-interfața sa, `java.io.Externalizable`. **Deserializarea** este procesul conversiei formei serializate a obiectului într-o copie a obiectului original.

Iată un exemplu tipic de cod pentru un bean simplu.

```
import java.beans.*;
import java.io.Serializable;

public class NewBean extends Object implements Serializable {

    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";
    private String sampleProperty;
```

```
        private PropertyChangeSupport propertySupport;

        public NewBean() {
            propertySupport = new PropertyChangeSupport(this);
        }

        public String getSampleProperty() {
            return sampleProperty;
        }

        public void setSampleProperty(String value) {
            String oldValue = sampleProperty;
            sampleProperty = value;
            propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,
oldValue, sampleProperty);
        }

        public void addPropertyChangeListener(PropertyChangeListener
listener) {
            propertySupport.addPropertyChangeListener(listener);
        }

        public void removePropertyChangeListener(PropertyChangeListener
listener) {
            propertySupport.removePropertyChangeListener(listener);
        }
    }
```

12.5 elemente (tags) JSP

Există 3 categorii de elemente JSP:

1. **directive** – afectează structura întregii pagini JSP
2. **elemente de scriptare** – cod java inserat într-o pagină JSP
3. **acțiuni** – elemente speciale care afectează comportamentul paginii în timpul execuției

12.6 directive JSP

Directivele JSP sunt mesaje trimise de către pagina JSP către containerul JSP. Aceste directive nu produc nici un efect asupra paginii afișate la client dar afectează întreg fișierul JSP. Pentru mai multe

informații, accesați site-ul: http://beginnersbook.com/2013/05/jsp-tutorial-directives/.

Forma generală a unei directive JSP este astfel:

```
<%@directive_name attr1="val1" ... attrn="valn" %>
```

Există trei directive JSP: **page**, **include** și **taglib**.

Formatul directivei **page**:

```
<%@page attr1="val1" ... %>
```

atribute:

- **language** – valori: "java"
- **extends** – superclasa clasei generate
- **import** – lista pachetelor de clase importate
- **session** – "true" sau "false", obiectul implicit `session` este disponibil
- **buffer** – modelul de buffer pentru output stream
- **autoflush** – dacă are valoarea "true", buffer-ul este golit automat, dacă este plin
- **isThreadSafe** – "true" sau "false"
- **isErrorPage** – "true" sau "false"
- **contentType** – tipul MIME al răspunsului
- **info**
- **errorPage** – URL-ul unei pagini de eroare, afișată în cazul unei erori

Directiva **include** cere containerului să includă inline conținutul resursei identificate prin "`fileName`". Formatul acestei directive este :

```
<%@include file="fileName" %>
```

Directiva **taglib** permite utilizarea unor elemente (tag-uri) proprii, definite întro bibliotecă de elemente (tag-uri). Are următorul format:

```
<%@taglib uri="tagLibUri" prefix="tagPrefix" %>
```

Noile elemente sunt identificate prin atributul `tagPrefix`, care specifică o zonă de numire (name scope), care permite identificarea corectă a noilor elemente, în caz de conflict de numire.

12.7 elemente de scriptare

Elementele de scriptare (în esență, cod Java) sunt de trei tipuri: declarații, scriptlete și expresii.

12.7.1 declarații

```
<%! variabile java și declarații de metode %>
```

În principiu, un bloc de cod Java folosit pentru a declara variabile și metode la nivel de clasă, în servletul generat din pagina JSP.

12.7.2 scriptlete

```
<% instrucțiuni java %>
```

Bloc de cod java care este executat atunci când este procesată cererea clientului. În Tomcat, acest cod ajunge în metoda `service()`.

12.7.3 expresii

```
<%= java expressions to be evaluated %>
```

Un scriptlet care trimite valoarea unei expresii Java înapoi către client. În momentul procesării, expresia este evaluată și rezultatul este convertit într-un șir de caractete care este apoi afișat.

12.8 acțiuni standard (standard actions)

Aceste elemente afectează comportamentul paginii JSP în momentul execuției precum și răspunsul trimis clientului. Acțiunile stndard vor fi detaliate în paragrafele următoare.

12.8.1 acțiunea standard useBean

```
<jsp:useBean>
```

Utilizată pentru instanțierea unui bean sau pentru localizarea unei instanțe. Totodată, îi atribuie un nume sau un ID.

Sintaxa acestei acțiuni este:

```
<jsp:useBean  id="beanName" scope="sName" beandetails />
```

unde `beandetails` poate fi unul din următoarele lucruri:

- class="className"
- class="className" type="typeName"
- beanName="beanName" type="typeName"
- type="typeName"

Pentru a clarifica distincția dintre numele clasei și numele bean-ului (beanName), se poate citi secțiunea de mai jos:

Linia de cod:

```
class="package.class" type="package.class"
```

Instanțiază un bean din clasa specificată de atributul `class` și atribuie bean-ului tipul de dată specificat în `type`. Valoarea acestui tip (type) poate fi aceeași cu a clasei, o superclasă a clasei sau o interfață implementată de către clasă.

Clasa specificată ca valoare a atributului class nu poate fi abstractă și trebuie să aibă un constructor

public, fărăr argumente. Capitalizarea numelui pachetului și clasei contează.

Această linie de cod:

```
beanName="{package.class | <%= expression %>}" type="package.class"
```

Instanțiază un bean dintr-o clasă, un șablon serializat sau dintr-o expresie care este evaluată ca și clasă sau șablon serializat. Atunci când se utilizează `beanName`, bean-ul este instanțiat de către metoda `java.beans.Beans.instantiate()`. Metoda `Beans.instantiate()` verifică dacă pachetul sau clasa specificată reprezintă o clasă sau un șablon serializat. Dacă reprezintă un șablon serializat, `Beans.instantiate()` citește forma serializată (care are un nume de genul `package.class.ser`) utilizând un încărcător de clasă (class loader).

12.8.2 acțiunea standard setProperty

`<jsp:setProperty>`

Utilizată cu acțiunea `<jsp:useBean>` pentru a seta valoarea unei proprietăți a bean-ului.

Sintaxa pentru această acțiune:

```
<jsp:setProperty name="beanName" propertydetails />
```

unde `propertydetails` este una din următoarele variante:

- property="*"
- property="propertyName"
- property="propertyName" param="parameterName"
- property="propertyName" value="propertyValue"

`propertyValue` fiind un șir de caractere sau un scriptlet.

Descrierea atributelor:

- **name** - numele instanței bean-ului, definit deja într-un `<jsp:useBean>`
- **property** - specifică relația dintre parametrii cererii și proprietățile corespunzătoare ale bean-ului
- **property="*"** - stochează toate valorile parametrilor cererii în proprietățile corespunzatoare ale bean-ului. Numele acestor proprietăți trebuie să corespundă cu numele parametrilor cererii.
- **property="*propertyName*" [param="*parameterName*"]** - setează valoarea unei proprietăți a bean-ului cu valoarea proprietății specificate a cererii. Aceasta poate avea un nume diferit de cel al proprietății bean-ului, dar, în acest caz, trebuie specificat explicit acest nume prin `param`.
- **property="*propertyName*" value="{ *string* | <%= *expression* %> }"** - Setează o proprietate a bean-ului cu valoarea specificată. Aceasta poate fi un `String` sau o expresie.

12.8.3 acțiunea standard getProperty

`<jsp:getProperty>`

Utilizată pentru accesarea proprietăților unui bean și pentru conversia acestora într-un `String` care este afișat clientului.

Sintaxa acestei acțiuni este:

```
<jsp:getProperty name="beanName" property="propName" />
```

Descrierea atributelor:

- **name** - numele unei instanțe de bean ale cărei proprietăți urmează a fi obținute
- **property** – numele proprietății care va fi obținută

12.8.4 acțiunea standard param

`<jsp:param>`

Utilizată în interiorul altor elemente pentru a le oferi informații adiționale sub forma unor perechi nume:valoare. Etse utilizată în contextul unor elemente precum acțiunile <jsp:include>, <jsp:forward>, <jsp:plugin>.

Sintaxa acestei acțiuni este:

```
<jsp:param name="paramName" value="paramValue" />
```

12.8.5 acțiunea standard include

`<jsp:include>`

Utilizată pentru includerea unei resurse statice sau dinamice în pagina JSP curentă în momentul procesării cererii. O resursă astfel inclusă are acces doar la obiectul `JspWriter` și nu poate seta headere sau cookies. În timp ce directiva `<%@include>` este executată în momentul compilării și are conținut static, acțiunea `<jsp:include>` aeste executată în momentul procesării cererii și poate avea conținut static sau dinamic.

Sintaxa pentru această acțiune este:

```
<jsp:include page="pageURL" flush="true" />
```

Descrierea atributelor:

- **page** – URL-ul paginii, are același format ca și directiva `<%@include>`.
- **flush** – doar valoarea "true" este acceptată.

12.8.6 acțiunea standard forward

`<jsp:forward>`

Utilizată pentru direcționarea cererii către un alt JSP, servlet sau o resursă statică.

Sintaxa pentru această acțiune este:

```
<jsp:forward page="pageURL" />
```

Acțiunea poate include și câteva elemente `<jsp:param>`. Este utilizată, în principal, atunci când vrem

să separăm aplicația în mai multe părți, în funcție de cerere și conținutul său.

12.8.7 acțiunea standard plugin

`<jsp:plugin>`

Utilizat în pagini pentru a genera elemente specifice browser-ului clientului (`<OBJECT>` sau `<EMBED>`) care necesită (dacă este cazul) descărcarea unui plugin Java, urmată de excuția unui applet sau a unei componente de tip bean Java specificat(ă) de către element.

Sintaxa pentru această acțiune este:

```
<jsp:plugin type="bean|applet" code="objCode" codeBase="objCodeBase"
align="align" archive="archiveList" height="height" hspace="hSpace"
jreversion="jreVersion" name="componentName" vspace="vSpace"      width="width"
nspluginurl="netscapeURL" iepluginurl="IEURL">
    <jsp:params>
      <jsp:param name="paramName" value="paramValue" />
          ...
    </jsp:params>
</jsp:plugin>
```

Descrierea atributelor:

- **name** - numele instanței bean-ului, așa cum este definit în elementul `<jsp:useBean>`
- **type="bean|applet"** – tipul obiectului executat de către plugin. Valorile posibile sunt `bean` sau `applet`, deoarece acest atribut nu are o valoare implicită.
- **code="*classFileName*"** - numele fișierului de clasă Java care va fi executat de către plugin. Extensia .class trebuie inclusă în numele care urmează atributului `code`. Numele fișierului este relativ la directorul specificat în atributul `codebase`.
- **codebase="*classFileDirectoryName*"** – apecificația relativă sau absolută către directorul ce conține codul applet-ului. Dacă nu este specificată vreo valoare, este utilizată specificația directorului care conține fișierul JSP care conține elementul `<jsp:plugin>`.
- **name="*instanceName*"** – numele Bean-ului sau al instanței applet-ului, ceea ce permite ca applet-uri sau bean-uri apelate de către același JSP să comunice între ele.
- **archive="*URIToArchive*, ..."** - o listă de specificații, separate prin virgule, care precizează locațiile fișierelor de tip arhivă care urmează să fie încărcate de către un class loader în directorul specificat de `codebase`.
- **align="bottom|top|middle|left|right"** – poziționarea imaginii afișate de către applet sau Bean, relativ la linia din fișierul JSP care conține elementul `<jsp:plugin>`.
- **height="*displayPixels*" width="*displayPixels*"** – înălțimea și lățimea inițială, în pixeli, a imaginii afișate de către applet sau Bean, fără a lua în considerare orice altă fereastră sau cutie de dialog afișată de către applet sau Bean.
- **hspace="*leftRightPixels*" vspace="*topBottomPixels*"** – distanța, în pixeli, la stânga sau dreapta (sau sus și jos) imaginii afișate de către applet sau Bean. Trebuie să fie valori nenule.
- **jreversion="*JREVersionNumber*|1.1"** - versiunea Java Runtime Environment (JRE) cerută de către applet sau Bean. Valoarea implicită este 1.1.
- **nspluginurl="*URLToPlugin*"** - URL link-ului de unde utilizatorul poate descărca plugin-ul pentru

Netscape Navigator. Valoarea poate fi un URL întreg, cu nume de protocol, un nume de domeniu, precum și un număr de port (opțional).

- **iepluginurl="*URLToPlugin*"** - URL link-ului de unde utilizatorul poate descărca plugin-ul pentru Internet Explorer.

12.9 obiecte implicite

JSP oferă câteva obiecte implicite, specificate de către API-ul pentru servlete, obiecte care sunt disponibile pentru programator.

1. **request** – reprezintă obiectul care a dus la invocarea metodei `service()`, are tipul **HttpServletRequest** și scopul **request**
2. **response** – reprezintă răspunsul serverului la cerere, are tipul **HttpServletResponse** și scopul **page**
3. **pageContext** – oferă un punct unic de acces la atribute și la datele partajate ale paginii, are tipul **PageContext** și scopul **page**
4. **session** – are tipul **HttpSession** și scopul **session**
5. **application** – reprezintă contextul servletului generat, are tipul **ServletContext** și scopul **application**
6. **out** - reprezintă o versiune cu buffer a obiectului (clasei) `java.io.PrintWriter` și scrie în stream-ul de ieșire pentru răspunsul către client și are tipul **javax.servlet.jsp.JspWriter** iar scopul e **page**
7. **config** – este obiectul gen `SevletConfig` al paginii curente JSP, are tipul **ServletConfig** și scopul **page**
8. **page** - este o instanță a implementării clasei de servlet, are tipul **java.lang.Object** și scopul **page.**

12.10 domenii de vizibilitate (scopes)

Scopul (scope) sau domeniul de vizibilitate al unui obiect este pur și simplu obiectul (implicit JSP) care îl conține (ca atribut, variabilă membru sau cum vreți să-l numiți).

1. **request** - un obiect cu scopul request este legat de (aparține) obiectul(ui) `HttpServletRequest`; obiectul poate fi accesat invocând metoda `getAttribute()` a obiectului implicit **request**; servletul generat leagă obiectul de `HttpServletRequest` utilizând metoda `setAttribute(String key, Object value)`
2. **session** - un obiect cu scopul session este legat de (aparține) obiectul(ui) `HttpSession`; obiectul poate fi accesat invocând metoda `getValue()` a obiectului implicit **session**; servletul generat leagă obiectul de `HttpSession` utilizând metoda `setAttribute(String key, Object value)`
3. **application** - un obiect cu scopul application este legat de (aparține) obiectul(ui) `ServletContext`; obiectul poate fi accesat invocând metoda `getAttribute()` a obiectului implicit **application**; servletul generat leagă obiectul de `ServletContext` utilizând metoda `setAttribute(String key, Object value)`
4. **page** - un obiect cu scopul page este legat de (aparține) obiectul(ui) `PageContext`; obiectul poate fi accesat invocând metoda `getAttribute()` a obiectului implicit **pageContext**; servletul generat leagă obiectul de `PageContext` utilizând metoda `setAttribute(String key, Object value)`

12.11 un scurt exemplu JSP

Prezentăm fișierul Enroll.jsp.

```
<%@page contentType="text/html" errorPage="error.jsp"%>

<jsp:useBean id="enrollBean" scope="session"
class="com.bank11.ccards.beans.EnrollBean" />
<jsp:setProperty name="enrollBean"  property="*" />

<%  enrollBean.init();

    if (enrollBean.invalidAcct())
    { %>
  <jsp:forward page="retry.jsp">
      <jsp:param name="resolution" value="invalidAcct"/>
  </jsp:forward>
    <% }
    else if (enrollBean.registeredAcct())
    { %>
  <jsp:forward page="response.jsp">
      <jsp:param name="resolution" value="registeredAcct"/>
  </jsp:forward>
    <% }
    else if (enrollBean.userExists())
    { %>
  <jsp:forward page="retry.jsp">
      <jsp:param name="resolution" value="userExists"/>
  </jsp:forward>
    <% }
    else
    {
  enrollBean.register(); %>
  <jsp:forward page="response.jsp">
      <jsp:param name="resolution" value="userEnrolled"/>
  </jsp:forward>
    <% }
%>
```

12.12 un exemplu JSP extins

Acest exemplu este oferit de către **Devsphere,** o companie de consultanță și de dezvoltare de soft.

12.12.1 bean-uri de date

SimpleBean este un bean Java care conține câteva proprietăți standard (un String, un float, un int, un boolean și un alt String), două proprietăți standard indexate (un String[] și un int[]) precum și un alt bean (un SimpleSubBean). Clasa SimpleBean este declarată ca public, are un constructor fără argumente și oferă metode de accesare (get & set) pentru proprietățile sale. Constructorul public poate fi omis deoarece compilatorul Java generează automat un astfel de constructor, în absența oricărui constructor.

SimpleBean.java:

```
package com.devsphere.examples.mapping.simple;

// Simple bean

public class SimpleBean implements java.io.Serializable {
    private String string;
    private float number;
    private int integer;
    private boolean flag;
    private String colors[];
    private int list[];
    private String optional;
    private SimpleSubBean subBean;

   // No-arg constructor
    public SimpleBean() {
    }

   // Gets the string property
    public String getString() {
        return this.string;
    }

   // Sets the string property
    public void setString(String value) {
        this.string = value;
    }

   // Gets the number property
    public float getNumber() {
        return this.number;
    }
```

```
// Sets the number property
 public void setNumber(float value) {
     this.number = value;
 }

// Gets the integer property
 public int getInteger() {
     return this.integer;
 }

// Sets the integer property
 public void setInteger(int value) {
     this.integer = value;
 }

// Gets the flag property
 public boolean getFlag() {
     return this.flag;
 }

// Sets the flag property
 public void setFlag(boolean value) {
     this.flag = value;
 }

// Gets the colors property
 public String[] getColors() {
     return this.colors;
 }

// Sets the colors property
 public void setColors(String values[]) {
     this.colors = values;
 }

// Gets an element of the colors property
 public String getColors(int index) {
     return this.colors[index];
 }
```

```
// Sets an element of the colors property
 public void setColors(int index, String value) {
     this.colors[index] = value;
 }

// Gets the list property
 public int[] getList() {
     return this.list;
 }

// Sets the list property
 public void setList(int values[]) {
     this.list = values;
 }

// Gets an element of the list property
 public int getList(int index) {
     return this.list[index];
 }

// Sets an element of the list property
 public void setList(int index, int value) {
     this.list[index] = value;
 }

// Gets the optional property
 public String getOptional() {
     return this.optional;
 }

// Sets the optional property
 public void setOptional(String value) {
     this.optional = value;
 }

// Gets the subBean property
 public SimpleSubBean getSubBean() {
     return this.subBean;
```

```
    }

   // Sets the subBean property
    public void setSubBean(SimpleSubBean value) {
        this.subBean = value;
    }
}
```

SimpleSubBean conține doar două proprirtăți standard (un String și un float).

SimpleSubBean.java:

```
package com.devsphere.examples.mapping.simple;

// Simple sub-bean
public class SimpleSubBean implements java.io.Serializable {
    private String string;
    private float number;

   // No-arg constructor
    public SimpleSubBean() {
    }

   // Gets the string property
    public String getString() {
        return this.string;
    }

   // Sets the string property
    public void setString(String value) {
        this.string = value;
    }

   // Gets the number property
    public float getNumber() {
        return this.number;
    }

   // Sets the number property
    public void setNumber(float value) {
        this.number = value;
```

```
        }
    }
```

12.12.2 forma HTML

Proprietățile clasei of SimpleBean sunt asociate elementelor formei SimpleForm.html:

Nume	Tipul proprietății	Tipul elementului
string	String	text
number	float	text
integer	int	radio[]
flag	boolean	checkbox
colors	String[]	checkbox[]
list	int[]	select
optional	String	text
subBean.string	String	text
subBean.number	float	text

SimpleForm.html:

```
<HTML>
<HEAD><TITLE>Simple form</TITLE></HEAD>
<BODY>
<H3>Simple Example</H3>

<FORM METHOD="POST">
<P> String <BR>
<INPUT TYPE="TEXT" NAME="string" SIZE="20">

<P> Number <BR>
<INPUT TYPE="TEXT" NAME="number" SIZE="20">

<P> Integer <BR>
<INPUT TYPE="RADIO" NAME="integer" VALUE="1">Option 1
<INPUT TYPE="RADIO" NAME="integer" VALUE="2">Option 2
<INPUT TYPE="RADIO" NAME="integer" VALUE="3">Option 3

<P> Flag <BR>
```

```
<INPUT TYPE="CHECKBOX" NAME="flag">Flag

<P> Colors <BR>
<INPUT TYPE="CHECKBOX" NAME="colors" VALUE="red">Red
<INPUT TYPE="CHECKBOX" NAME="colors" VALUE="green">Green
<INPUT TYPE="CHECKBOX" NAME="colors" VALUE="blue">Blue

<P> List <BR>
<SELECT NAME="list" SIZE="3" MULTIPLE>
<OPTION VALUE="1">Item 1</OPTION>
<OPTION VALUE="2">Item 2</OPTION>
<OPTION VALUE="3">Item 3</OPTION>
</SELECT>

<P> Optional <BR>
<INPUT TYPE="TEXT" NAME="optional" SIZE="20">

<P> String (subBean) <BR>
<INPUT TYPE="TEXT" NAME="subBean.string" SIZE="20">

<P> Number (subBean) <BR>
<INPUT TYPE="TEXT" NAME="subBean.number" SIZE="20">

<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

12.12.3 resursele bean-ului

Clasa SimpleBeanResources class este un pachet de resurse (resource bundle) care conține informații opționale despre setarea parametrilor - valori implicite, mesaje de eroare, lista proprietăților opționale, ordinea de procesare, numele formei și al procesorului.

Valorile implicite sunt definite pentru un String, un float, un boolean și un int []. Valorile primitive trebuie să fie înfășurate de un Float și un Boolean pentru a fi stocate ca resurse. Valorile implicite pentru proprietățile bean-ului conținute ar fi putut fi definite în alt pachet de resurse numit SimpleSubBeanResources.

Există trei mesaje de eroare. Rolul lor este de a ajuta utilizatorii să corecteze erorile de intrare. Cadrul de mapare conține mesaje de eroare implicite pentru fiecare tip de element al formei.

Lista proprietăților opționale are un singur element. Nu se semnalează nici o eroare dacă utilizatorul nu furnizează o valoare pentru această proprietate.

Ordinea de procesare nu este necesară pentru acest exemplu. A fost inclus aici doar pentru scopuri demonstrative.

Numele formularului și numele procesorului sunt utilizate de operatorul JSP descris în secțiunea următoare. Aceste două resurse nu sunt accesate de utilitățile de mapare.

SimpleBeanResources.java:

```
package com.devsphere.examples.mapping.simple;

public class SimpleBeanResources extends java.util.ListResourceBundle {
    private static final Object[][] contents = {
        { "[DEFAULT_VALUE.string]", "abc" },
        { "[DEFAULT_VALUE.number]", new Float(0.123) },
        { "[DEFAULT_VALUE.flag]", new Boolean(true) },
        { "[DEFAULT_VALUE.list]", new int[] { 2, 3 } },
        { "[ERROR_MESSAGE.integer]", "An option must be selected" },
        { "[ERROR_MESSAGE.colors]", "One or more colors must be selected" },
        { "[ERROR_MESSAGE.list]", "One or more items must be selected" },
        {
            "[OPTIONAL_PROPERTIES]",
            new String[] {
                "optional"
            }
        },
        {
            "[PROCESSING_ORDER]",
            new String[] {
                "string",
                "number",
                "integer",
                "flag",
                "colors",
                "list",
                "optional",
                "subBean"
            }
        },
        { "[FORM_NAME]", "SimpleForm.html" },
        { "[PROC_NAME]", "SimpleProc.jsp" }
    };

    public Object[][] getContents() {
```

```
            return contents;
        }
    }
```

12.12.4 operatorul (handler) JSP

Operatorul SimpleHndl.jsp se bazează pe un șablon descris într-un capitol anterior.

Metoda formToBean () din com.devsphere.mapping.FormUtils stabilește proprietățile bean-ului la valorile parametrilor de cerere (datele formei). Dacă este necesar, valorile șirului sunt convertite în numere. O proprietate booleană este setată la true dacă parametrul de cerere este prezent indiferent de valoarea sa (cu excepția "false"). Mesajele de eroare care apar în timpul procesului de mapare sunt stocate într-un Hashtable.

Metoda beanToForm () din com.devsphere.mapping.FormUtils introduce datele bean-ului și mesajele de eroare în formularul HTML. Se inserează un atribut VALUE pentru elementele de text, un atribut CHECKED pentru casetele de selectare și butoanele radio care trebuie selectate și un atribut SELECTED pentru elementele listate care trebuie evidențiate.

Pentru o mai bună înțelegere a acestui exemplu, o secțiune ulterioară a acestui capitol prezintă două JSP-uri care realizează maparea și construiesc formularul HTML fără a utiliza acest cadru.

SimpleHndl.jsp:

```
<%@ page language="java" %>
<%@ page import="com.devsphere.mapping.*, com.devsphere.logging.*" %>
<jsp:useBean id="simpleBean" scope="request"
    class="com.devsphere.examples.mapping.simple.SimpleBean"/>
<%
    // Get the bean resources
    java.util.ResourceBundle beanRes
        = HandlerUtils.getBeanResources(simpleBean.getClass());

    // Construct the base path
    String basePath = request.getServletPath();
    int slashIndex = basePath.lastIndexOf('/');
    basePath = slashIndex != -1 ? basePath.substring(0, slashIndex+1) : "";

    // Determine the HTTP method
    boolean isPostMethod = request.getMethod().equals("POST");

    // Create a logger that wraps the servlet context
    ServletLogger logger = new ServletLogger(application);

    // Wrap the form data
    FormData formData = new ServletFormData(request);

    // Form-to-bean mapping: request parameters are mapped to bean
```

```
properties
        java.util.Hashtable errorTable
            = FormUtils.formToBean(formData, simpleBean, logger);


        if (isPostMethod && errorTable == null) {
            // Construct the processor's path
            String procPath = basePath +
beanRes.getString("[PROC_NAME]").trim();


            // Process the valid data bean instance
            application.getRequestDispatcher(procPath).forward(request,
response);
        } else {
            if (!isPostMethod)
                // Ignore the user errors if the form is requested with GET.
                errorTable = HandlerUtils.removeUserErrors(errorTable);


            // Construct the form's path
            String formPath = basePath +
beanRes.getString("[FORM_NAME]").trim();
            formPath = application.getRealPath(formPath);


            // Get the form template
            FormTemplate template = FormUtils.getTemplate(new
java.io.File(formPath));


            // Get a new document
            FormDocument document = template.getDocument();


            // Bean-to-form mapping: bean properties are mapped to form elements
            FormUtils.beanToForm(simpleBean, errorTable, document, logger);


            // Send the form document
            document.send(out);
        }
    %>
```

12.12.5 procesorul JSP

Procesorul SimpleProc.jsp primește bean-urile care au fost validate de operatorul JSP și imprimă valorile proprietăților lor.

SimpleProc.jsp:

```
<%@ page language="java"%>
<jsp:useBean id="simpleBean" scope="request"
    class="com.devsphere.examples.mapping.simple.SimpleBean"/>
<HTML>
<HEAD><TITLE>Simple bean</TITLE></HEAD>
<BODY>
<H3>Simple Example</H3>
<P><B> SimpleBean properties: </B>
    <P> string = <jsp:getProperty name="simpleBean" property="string"/>
    <P> number = <jsp:getProperty name="simpleBean" property="number"/>
    <P> integer = <jsp:getProperty name="simpleBean" property="integer"/>
    <P> flag = <jsp:getProperty name="simpleBean" property="flag"/>
    <P> colors = <%= toString(simpleBean.getColors()) %>
    <P> list = <%= toString(simpleBean.getList()) %>
    <P> optional = <jsp:getProperty name="simpleBean" property="optional"/>
    <P> subBean.string = <%= simpleBean.getSubBean().getString() %>
    <P> subBean.number = <%= simpleBean.getSubBean().getNumber() %>
</BODY>
</HTML>

<%!
    public static String toString(String list[]) {
        if (list == null || list.length == 0)
            return "";
        if (list.length == 1 && list[0] != null)
            return list[0];
        StringBuffer strbuf = new StringBuffer();
        strbuf.append("{  ");
        for (int i = 0; i < list.length; i++)
            if (list[i] != null) {
                strbuf.append(list[i]);
                strbuf.append("  ");
            }
        strbuf.append("}");
        return strbuf.toString();
    }

    public static String toString(int list[]) {
        if (list == null || list.length == 0)
```

```
            return "";
        if (list.length == 1)
            return Integer.toString(list[0]);
        StringBuffer strbuf = new StringBuffer();
        strbuf.append("{  ");
        for (int i = 0; i < list.length; i++) {
            strbuf.append(list[i]);
            strbuf.append("  ");
        }
        strbuf.append("}");
        return strbuf.toString();
    }
%>
```

12.12.6 fără utilizarea cadrului devsphere

ComplexForm.jsp generează dinamic formularul HTML și introduce valori implicite și mesaje de eroare. Utilizează 120 de linii de cod Java-JSP-HTML pentru a genera un formular HTML de 40 linii. Un singur apel la FormUtils.beanToForm () poate face același lucru folosind un fișier HTML pur. În plus, beanToForm () gestionează și înregistrează multe tipuri de erori ale aplicației, făcând mai ușoară testarea și depanarea.

ComplexHndl.jsp utilizează 150 de linii de cod Java-JSP pentru a seta proprietățile unui obiect al bean-ului la valorile parametrilor de solicitare. Acesta este echivalentul unui singur apel FormUtils.formToBean ().

Adăugarea / eliminarea unei proprietăți a bean-ului necesită modificări în ambele fișiere Complex*.jsp. Folosind cadrul, trebuie doar să adăugați / să eliminați un element de formă în / din un fișier HTML pur.

Localizarea fișierelor Complex*.jsp în alte limbi necesită multă muncă și ar putea face întreținerea foarte dificilă. Folosind cadrul, separați codul HTML de codul Java / JSP. În plus, valorile implicite și mesajele de eroare sunt păstrate în pachete de resurse localizabile. Un capitol ulterior arată cum se construiesc aplicații internaționalizate utilizând cadrul.

ComplexForm.jsp:

```
<%@ page language="java" %>
<jsp:useBean id="simpleBean" scope="request"
    class="com.devsphere.examples.mapping.simple.SimpleBean"/>
<jsp:useBean id="errorTable" scope="request"
    class="java.util.Hashtable"/>
<HTML>
<HEAD><TITLE>Without using the framework</TITLE></HEAD>
<BODY>
<H3>Equivalent of Simple Example</H3>
<FORM METHOD=POST>

<P> String <BR>
<%= getErrorMessage(errorTable, "string") %>
<INPUT TYPE="TEXT" NAME="string"
```

```
    VALUE="<jsp:getProperty name="simpleBean" property="string"/>">

<P> Number <BR>
<%= getErrorMessage(errorTable, "number") %>
<INPUT TYPE="TEXT" NAME="number"
    VALUE="<jsp:getProperty name="simpleBean" property="number"/>">

<P> Integer <BR>
<%= getErrorMessage(errorTable, "integer") %>
<%
    String integerLabels[] = { "Option 1", "Option 2", "Option 3" };
    for (int i = 0; i < integerLabels.length; i++) {
        int value = i+1;
        boolean checked = simpleBean.getInteger() == value;
%>
        <INPUT TYPE="RADIO" NAME="integer" VALUE="<%= value %>"
            <%= checked ? "CHECKED" : "" %>> <%= integerLabels[i] %>
<%
    }
%>

<P> Flag <BR>
<%= getErrorMessage(errorTable, "flag") %>
<INPUT TYPE="CHECKBOX" NAME="flag"
    <%= simpleBean.getFlag() ? "CHECKED" : "" %>> Flag

<P> Colors <BR>
<%= getErrorMessage(errorTable, "colors") %>
<%
    String colors[] = simpleBean.getColors();
    if (colors == null)
        colors = new String[0];
    String colorLabels[] = { "Red", "Green", "Blue" };
    String colorValues[] = { "red", "green", "blue" };
    for (int i = 0; i < colorValues.length; i++) {
        boolean checked = false;
        if (colors != null)
            for (int j = 0; j < colors.length; j++)
                if (colors[j].equals(colorValues[i])) {
```

```
                    checked = true;
                    break;
                }
%>
        <INPUT TYPE="CHECKBOX" NAME="colors" VALUE="<%= colorValues[i] %>"
            <%= checked ? "CHECKED" : "" %>> <%= colorLabels[i] %>
<%
    }
%>

<P> List <BR>
<%= getErrorMessage(errorTable, "list") %>
<SELECT NAME="list" SIZE="3" MULTIPLE>
<%
    int list[] = simpleBean.getList();
    if (list == null)
        list = new int[0];
    String listItems[] = { "Item 1", "Item 2", "Item 3" };
    for (int i = 0; i < listItems.length; i++) {
        int value = i+1;
        boolean selected = false;
        if (list != null)
            for (int j = 0; j < list.length; j++)
                if (list[j] == value) {
                    selected = true;
                    break;
                }
%>
        <OPTION VALUE = "<%= value %>"
            <%= selected ? "SELECTED" : "" %>> <%= listItems[i] %>
<%
    }
%>
</SELECT>

<P> Optional <BR>
<%= getErrorMessage(errorTable, "optional") %>
<INPUT TYPE="TEXT" NAME="optional"
    VALUE="<jsp:getProperty name="simpleBean" property="optional"/>">
```

```
<% if (simpleBean.getSubBean() == null) simpleBean.setSubBean(
new com.devsphere.examples.mapping.simple.SimpleSubBean()); %>

<P> String (subBean) <BR>
<%= getErrorMessage(errorTable, "subBean.string") %>
<INPUT TYPE="TEXT" NAME="subBean.string"
    VALUE="<%= simpleBean.getSubBean().getString() %>">

<P> Number (subBean) <BR>
<%= getErrorMessage(errorTable, "subBean.number") %>
<INPUT TYPE="TEXT" NAME="subBean.number"
    VALUE="<%= simpleBean.getSubBean().getNumber() %>">

<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Reset">

</FORM>
</BODY>
</HTML>

<%!
    String getErrorMessage(java.util.Hashtable errorTable, String property)
{
        String message = (String) errorTable.get(property);
        if (message == null)
            message = "";
        return message;
    }
%>
```

ComplexHndl.jsp:

```
<%@ page language="java" %>
<jsp:useBean id="simpleBean" scope="request"
    class="com.devsphere.examples.mapping.simple.SimpleBean"/>
<jsp:useBean id="simpleSubBean" scope="page"
    class="com.devsphere.examples.mapping.simple.SimpleSubBean"/>
<jsp:useBean id="errorTable" scope="request"
    class="java.util.Hashtable"/>
```

```
    <%
        simpleBean.setSubBean(simpleSubBean);

        boolean isPostMethod = request.getMethod().equals("POST");
        if (isPostMethod) {

    //* string : text

    %>
            <jsp:setProperty name="simpleBean" property="string"/>
    <%
            if (simpleBean.getString() == null
                || simpleBean.getString().length() == 0) {
                simpleBean.setString("abc");
                setErrorMessage(errorTable, "string", "Must be filled");
            }

    //* number : text

            try {
                String numberValue = request.getParameter("number");
                if (numberValue != null && numberValue.length() != 0)
                    simpleBean.setNumber(new Float(numberValue).floatValue());
                else {
                    simpleBean.setNumber(0.123f);
                    setErrorMessage(errorTable, "number", "Must be filled");
                }
            } catch (NumberFormatException e) {
                simpleBean.setNumber(0.123f);
                setErrorMessage(errorTable, "number", "Must be a number");
            }

    //* integer : radio group

    %>
            <jsp:setProperty name="simpleBean" property="integer"/>
    <%
            if (simpleBean.getInteger() == 0) {
                setErrorMessage(errorTable, "integer", "An option must be
selected");
```

```
        }

//* flag : checkbox

        String flagValue = request.getParameter("flag");
        if (flagValue != null) {
            flagValue = flagValue.trim();
            if (flagValue.length() == 0 || flagValue.equals("false"))
                flagValue = null;
        }
        simpleBean.setFlag(flagValue != null);

//* color : checkbox group

%>
        <jsp:setProperty name="simpleBean" property="colors"/>
<%
        if (simpleBean.getColors() == null
            || simpleBean.getColors().length == 0) {
            setErrorMessage(errorTable, "colors",
                "One or more colors must be selected");
        }

//* list : select

%>
        <jsp:setProperty name="simpleBean" property="list"/>
<%
        if (simpleBean.getList() == null
            || simpleBean.getList().length == 0) {
            simpleBean.setList(new int[] { 2, 3 });
            setErrorMessage(errorTable, "list",
                "One or more items must be selected");
        }

//* optional : text

%>
        <jsp:setProperty name="simpleBean" property="optional"/>
```

```
    <%
            if (simpleBean.getOptional() == null)
                simpleBean.setOptional("");

    //* subBean.string : text

    %>
            <jsp:setProperty name="simpleSubBean" property="string"
                param="subBean.string"/>
    <%
            if (simpleSubBean.getString() == null
                || simpleSubBean.getString().length() == 0) {
                simpleSubBean.setString("");
                setErrorMessage(errorTable, "subBean.string", "Must be filled");
            }

    //* subBean.number : text

            try {
                String numberValue = request.getParameter("subBean.number");
                if (numberValue != null && numberValue.length() != 0)
                    simpleSubBean.setNumber(new
Float(numberValue).floatValue());
                else {
                    setErrorMessage(errorTable, "subBean.number", "Must be
filled");
                }
            } catch (NumberFormatException e) {
                setErrorMessage(errorTable, "subBean.number", "Must be a
number");
            }
        } else {
            simpleBean.setString("abc");
            simpleBean.setNumber(0.123f);
            simpleBean.setFlag(true);
            simpleBean.setList(new int[] { 2, 3 });
            simpleBean.setOptional("");
            simpleSubBean.setString("");
        }
```

```
        if (isPostMethod && errorTable.isEmpty()) {
    %>
            <jsp:forward page="SimpleProc.jsp"/>
    <%
        } else {
    %>
            <jsp:forward page="ComplexForm.jsp"/>
    <%
        }
    %>


    <%!
        void setErrorMessage(java.util.Hashtable errorTable,
            String property, String message) {
            message = "<FONT COLOR=\"#FF0000\">" + message + "</FONT><BR>";
            errorTable.put(property, message);
        }
    %>
```

12.12.7 utilizarea cadrului cu servlete și JSP-uri

Operatorul SimpleHndl.jsp este în esență un scriptlet Java. Acesta a fost un mod simplu și compact de prezentare a unui operator. Codul Java ar putea fi ușor mutat într-o clasă de utilități. O soluție mai elegantă este înlocuirea unui operator JSP cu un servlet Java general.

Pachetul com.devsphere.helpers.mapping conține o clasă abstractă numită GenericHandler. Această clasă este extinsă de BeanDispatcher, care este echivalentul de independent de bean SimpleHndl.jsp. Operatorul (handler) JSP poate fi înlocuit cu doar câteva linii care sunt adăugate la servlets.properties sau web.xml:

```
    SimpleHndl.code=com.devsphere.helpers.mapping.BeanDispatcher
    SimpleHndl.initparams=\
        BEAN_NAME=com.devsphere.examples.mapping.simple.SimpleBean,\
        BEAN_ID=simpleBean,\
        BASE_PATH=/simple
    or
    <servlet>
        <servlet-name>SimpleHndl</servlet-name>
        <servlet-class>com.devsphere.helpers.mapping.BeanDispatcher</servlet-
class>
        <init-param>
            <param-name>BEAN_NAME</param-name>

<param-value>com.devsphere.examples.mapping.simple.SimpleBean</param-value>
```

```
        </init-param>
        <init-param>
            <param-name>BEAN_ID</param-name>
            <param-value>simpleBean</param-value>
        </init-param>
        <init-param>
            <param-name>BASE_PATH</param-name>
            <param-value>/simple</param-value>
        </init-param>
    </servlet>
```

chapter 13 JSF

13.1 ce este JavaServer Faces?

Thehnologia JavaServer Faces este o platformă (cadru) pentru dezvoltarea (pe partea de server) a componentelor de interfață utilizator (UI) ale aplicațiilor web scrise în Java. Această tehnologie include:

1. Un set de API-uri pentru:
   - reprezentarea componentelor de UI, precum câmpuri de introducere, butoane, link-uri
   - administrarea componentelor de UI
   - procesarea evenimentelor
   - validarea introducerilor
   - procesarea erorilor
   - specificarea regulilor de navigare între pagini
   - suport pentru internaționalizare și accesibilitate
2. O bibliotecă de elemente JavaServer Pages (JSP) pentru prezentarea unei interfețe JavaServer Faces într-o pagină JSP.

JSF este o platformă MVC (Model View Controller) controlată de cereri, bazată pe componente UI, utilizând fișiere XML, numite șabloane de vedere sau vederi Facelet. Cererile sunt procesate de către FacesServlet, care încarcă șablonul de vedere corespunzător, construiește un arbore al componentelor, procesează evenimente și afișează răspunsul (de obicei, un fișier HTML) la client.

Pentru o discuție asupra diferențelor dintre pagini JSP, servlets, JSF-uri, puteți consulta link-ul http://stackoverflow.com/questions/2095397/what-is-the-difference-between-jsf-servlet-and-jsp .

13.2 specificații pentru tehnologia JavaServer Faces

Prima versiune JSP a apărut ca urmare a dezvoltării pachetului `javax.servlet.ui`, în anul 2001.

Ultima versiune oficială (pentru noiembrie 2018) a tehnologiei JavaServer Faces este versiunea 2.3 (apărută în martie 2017), ca JSR 372 și este parte a Java EE 8.

Versiunea 2.3 introduce ca noutăți: Expresii de căutare, URL-uri fără extensii, validare de bean-uri pentru clase complete, comunicare prin WebSocket-uri, integrare mai strânsă cu CDI (Contexts and Dependency Injection).

Versiunea 2.0 aste parte a Java Enterprise Edition 6 în timp ce versiunea 2.2 este parte a Java EE 7.

Versiunea 2.0 înlocuiește versiunea 1.2 și aduce ca noutăți suport obligatoriu pentru Facelets ca tehnologie de vedere pentru JavaServer Faces, suport integrat pentru Ajax și pentru marcaje și încărcare de pagină.

Versiunea 2.2, la rândul ei, introduce concepte noi precum vederi fără stare (stateless views), controlul afișării paginilor (page flow) precum și posibilitatea de a crea contracte pentru resurse portabile.

Există cinci biblioteci de elemente specifice pentru JSF în specificațiile curente, și anume:

- biblioteca JSF HTML
- biblioteca JSF Core
- biblioteca JSTL Core

- biblioteca JSTL Functions
- biblioteca JSF Facelets

13.3 facelets

Facelets este o tehnologie de vedere pentru JavaServer Faces care permite costruirea mai rapidă și mai simplă de vederi compuse decât JSP, care este tehnologia de vederi implicită pentru JSF.

Paginile JSP snt compilate în servlete, ceea ce nu este cazul cu Facelets, deoarece paginile Facelet sunt pagini XML iar cadrul lor de dezvoltare utilizează un compilator rapid bazat pe tehnologia SAX pentru construirea de vederi. Facelets pot modifica imediat paginile astfel încât dezvoltarea de aplicații JSP devine pur și simplu mai rapidă.

13.4 elementele JSF HTML

Această bibliotecă ce elemente conține elemente JavaServer pentru toate combinațiile UIComponent + HTML RenderKit Renderer definite în specificația JSF. Numărul acestor elemente este de 30 în specificația JSF 2.0.

Elementele JSF HTML pot fi grupate în mai multe categorii:

- input
- output
- comenzi
- selecție
- aranjare (layout)
- tabele de date
- erori și mesaje

13.4.1 lista elementelor JSF HTML

Pentru a ne face o idee asupra elementelor din această bibliotecă, iată o listă exhaustivă a elementelor din JSF HTML:

- button
- column
- commandButton
- commandLink
- dataTable
- form
- graphicImage
- head
- inputHidden
- inputSecret
- inputText
- inputTextArea
- link

- message
- messages
- outputFormat
- outputLabel
- outputLink
- outputScript
- outputStyleSheet
- outputText
- panelGrid
- panelGroup
- selectBooleanCheckbox
- selectManyCheckbox
- selectManyListbox
- selectManyMenu
- selectOneListbox
- selectOneMenu
- selectOneRadio

În paragrafele următoare ne vom uita la unele din aceste elemente mai în detaliu.

13.4.2 h:dataTable

Elementul **dataTable** afișează o tabelă conformă cu specificația HTML 4 care poate fi asociată cu un bean pentru a obține datele acestuia precum și pentru procesarea evenimentelor adiacente.

Tabela poate fi formatată extensiv utilizând clase și definiții CSS (cascading style sheets) pentru a adecva modalitatea de prezentare a capetelor de tabel, a liniilor, coloanelor sau a subsolului. Elemente de efect, precum alternarea culorii rândurilor pot fi obținute destul de ușor utilizând acest element.

Elementul **dataTable** conține de obicei unul sau mai multe elemente de tip coloană care definesc coloanele tabelei. O componentă a unei coloane este redată ca simplu element **td**. Pentru mai multe informații despre coloane, consultați documentatția despre elementul **column**.

Un element **dataTable** poate conține de asemenea facet-uri pentru header și footer. Acestea sunt redate ca un singur element **th** dintr-o linie din partea superioară a tabelului și, respectiv, ca un singur element de tip **th** din partea inferioară a tabelului.

Exemplu:

```
<h:dataTable id="table1" value="#{shoppingCartBean.items}" var="item">
  <f:facet name="header">
    <h:outputText value="Your Shopping Cart" />
  </f:facet>
  <h:column>
  <f:facet name="header">
    <h:outputText value="Item Description" />
  </f:facet>
  <h:outputText value="#{item.description}" />
```

```
    </h:column>
    <h:column>
    <f:facet name="header">
      <h:outputText value="Price" />
    </f:facet>
      <h:outputText value="#{item.price}" />
    </h:column>
    <f:facet name="footer">
      <h:outputText value="Total: #{shoppingCartBean.total}" />
    </f:facet>
  </h:dataTable>
```

HTML Output

```
<table id="table1">
  <thead>
    <tr><th scope="colgroup" colspan="2">Your Shopping Cart</th></tr>
    <tr><th>Item Description</th><th>Price</th></tr>
  </thead>
  <tbody>
    <tr><td>Delicious Apple</td><td>$5.00</td></tr>
    <tr><td>Juicy Orange</td><td>$5.00</td></tr>
    <tr><td>Tasty Melon</td><td>$5.00</td></tr>
  </tbody>
  <tfoot>
    <tr><td colspan="2">Total: $15.00</td></tr>
  </tfoot>
</table>
```

13.4.3 h:form

Elementul **form** redă un element HTML de tip **form**. Formele JSF utilizează tehnica "post-back" pentru a submite datele formei către pagina care conține forma. De asemenea, este obligatorie utilizarea metodei POST și nu este posibilă utilizarea metodei GET pentru formele generate de acest element.

Dacă aplicația cere metoda GET ca metodă de submitere a datelor, există posibilitatea utilizării elementului **form** din HTML, legând parametrii cererii de proprietățile bean-ului și utilizând elementul **outputLink** pentru a genera hyper-link-uri dinamice.

Exemplu:

```
<h:form id="form1"></h:form>
```

HTML Output

```
<form id="form1" name="form1" method="post" action="/demo/form.jsp"
enctype="application/x-www-form-urlencoded"></form>
```

13.4.4 h:commandButton

Elementul **commandButton** redă un boton HTML de submit, care poate fi asociat unui bean de suport sau unei clase **ActionListener** pentru procesarea evenimentelor. Valoarea afișată de către buton poate fi obținută dintr-o legătură (bundle) de mesaje pentru a obține suport pentru internaționalizare.

Exemplu:

```
<h:commandButton id="button1" value="#{bundle.checkoutLabel}"
action="#{shoppingCartBean.checkout}" />
```

HTML Output

```
<input id="form:button1" name="form:button1" type="submit"
value="Check Out" onclick="someEvent();" />
```

13.4.5 h:inputText

Elementul **inputText** redă un element HTML de input de tipul "text".

Example:

```
<h:inputText id="username" value="#{userBean.user.username}" />
```

HTML Output

```
<input id="form:username" name="form:username" type="text" />
```

13.4.6 elementul message

Elementul **message** afișează un mesaj pentru o anume componentă. Mesajul generat pentru această componentă poate fi particularizat aplicând diferite stiluri **CSS** mesajului, depinzând de severitatea sa (de ex. roșu pentru eroare, verde pentru informații) precum și nivelul de detaliu al mesajului. De asemenea, pot fi modificate și mesajele standard de eroare prin modificarea unor proprietăți JSF din legătura (bundle) de mesaje.

Example:

```
<h:inputText id="username" required="#{true}"
             value="#{userBean.user.username}"
             errorStyle="color:red"/>
<h:message for="username" />
```

HTML Output

```
<input type="text" id="form:username" name="form:username" value=""/>
<span style="color:red">"username": Value is required.</span>
```

13.5 elemente JSF din nucleu (core)

Elementele core ale JavaServer Faces definesc acțiuni care sunt independente de orice RenderKit

particular.

13.5.1 lista elementelor JSF Core

Iată o listă exhaustivă a elementelor JSF core:

- actionListener
- attribute
- convertDateTime
- converter
- convertNumber
- facet
- loadBundle
- param
- selectItem
- selectItems
- subview
- validateDoubleRange
- validateLength
- validateLongRange
- validator
- valueChangeListener
- verbatim
- view

Unele din aceste elemente vor fi detaliate în continuare.

13.5.2 f:facet

Un **facet** reprezintă o secțiune numită (delimitată) în cadrul unei componente.

Elementul **facet** înregistrează un facet numit componentei asociate cu elementul în care **f:facet** e inclus. De exemplu, se poate create un facet de header sau de subsol (footer) pentru o componentă de tip **dataTable**.

Example:

```
<h:dataTable id="reportTable" value="#{reportBean.dailyReport}" var="item">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Daily Report" />
    </f:facet>
    <h:outputText value="#{item}" />
  </h:column>
</h:dataTable>
```

HTML Output

```
<table id="reportTable">
  <thead>
    <tr><th>Daily Report</th></tr>
  </thead>
  <tbody>
    <tr><td>Item 1</td></tr>
    <tr><td>Item 2</td></tr>
    <tr><td>Item 3</td></tr>
  </tbody>
</table>
```

13.5.3 f:validator

Elementul **Validator** înregistrează o instanță de Validator pentru componenta asociată cu elementul care îl include. Platforma JavaServer Faces include trei validatori standard (a se vedea elementele **validateDoubleRange**, **validateLength** și **validateLongRange**), dar intefața Validator poate fi implementată de clase care efectuează validări specifice pentru aplicație. Acest element acceptă o valoare care coincide cu ID-ul atribuit clasei de validator în fișierul de configurare pentru Faces. Conținutul acestui element trebuie să fie gol.

Examplu:

```
<h:inputText id="emailAddress"
value="#{customerBean.customer.emailAddress}">
  <f:validator validatorId="emailAddressValidator" />
</h:inputText>
<h:message for="emailAddress" />
```

HTML Output

```
<input id="form:emailAddress" name="form:emailAddress" type="text"
value="fake@email"/>
Invalid email address.
```

13.5.4 f:valueChangeListener

Elementul **ValueChangeListener** înregistrează o instanță de ValueChangeListener pe componenta asociată cu elementul ce o conține. Interfața ValueChangeListener trebuie implementată de către clasele care trebuie înregistrate cu componentele care publică evenimente de schimbare a valorii.

Orice componentă care primește input de la utilizator, precum componentele HTML select sau de introducere text, pot publica evenimente de schimbare a valorii. O componentă crează un eveniment de schimbare a va lorii atunci când introducerea se schimbă, însă doar în cazul în care noua valoare e validată.

Se pot înregistra mai multe instanțe de ValueChangeListeners cu o componentă și ele vor fi invocate în ordinea în care au fost înregistrate. O aternativă la acest element ar putea fi utilizarea unei expresii de legare la metodă, expresie care conține o referință la o metodă de ascultarea a schimbării valorii a unui bean asociat.

De remarcat în exemplul de mai jos utilizarea evenimentului JavaScript `onchange()` care are ca efect submiterea formei atunci când selecția din listă se schimbă. Fără acest eveniment JavaScript, utilizatorul ar trebui să submită forma manual pentru a invoca ValueChangeListener.

Example:

```
<h:selectOneMenu id="optionMenu" value="#{optionBean.selectedOption}"
```

```
onchange="submit()">
      <f:selectItems value="#{optionBean.optionList}" />
      <f:valueChangeListener
type="com.mycompany.MyValueChangeListenerImpl" />
    </h:selectOneMenu>
```

HTML Output

```
<select name="form:optionMenu" size="1" onchange="submit()">
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
</select>
```

13.5.5 f:view

Elementul **View** este containerul pentru toate elementele JavaServer Faces pentru componente utilizate într-o pagină. Puteți împacheta elementul rădăcină al documentului vostru pentru a vă asigura că toate elementele descendente sunt parte a aceleiași vederi.

Acest element este de ajutor pentru scopuri de internaționalizare. Elementul oferă câteva opțiuni pentru a prezenta utilizatorului o versiune localizată a aplicației. Implicit, platforma JSF va încerca să selecteze cea mai bună vedere în baza header-ului Accept-Language trimis către server de către browser-ul utilizatorului ca parte a cererii HTTP pentru pagina voastră.

Dacă localizarea cerută de către utilizator nu este implementată de către aplicația voastră, platforma JSF va utiliza implicit localizarea specificată în fișierul de configurare pentru Faces. Dacă această localizare implicită nu este specificată, JSF va utiliza localizarea pentru mașina virtuală Java care deservește aplicația.

Dacă aplicația oferă suport pentru localizarea cerută de către utilizator, JSF va seeta această localizare pentru vedere și va afișa mesajele pentru această localizare definite în legătura de mesaje (message bundle) pentru localizare.

Puteți de asemenea specifica localizarea în care vederea va fi afișată precizând explicit atributul **locale** al elementului **view**. Aceasta permite proiectarea de versiuni localizate pentru fiecare pagină, incluzând imagini și stiluri, pentru fiecare localizare implementată.

O altă opțiune este de a obține localizarea dorită în mod dinamic, prin interacțiunea utilizatorului. Aceată alegere ar putea fi stocată ca un cookie sau într-o bază de date pentru a identifica localizarea preferată a unui utilizator. Atributul **locale** acceptă și o valoare sub forma unei expresii.

Example:

welcome_en.jsp (English)

```
<f:view locale="en">
  <f:loadBundle basename="com.mycompany.MessageBundle" var="bundle" />
  <h:outputText value="#{bundle.welcomeMessage}" />
</f:view>
```

welcome_fr.jsp (French)

```
<f:view locale="fr">
  <f:loadBundle basename="com.mycompany.MessageBundle" var="bundle" />
  <h:outputText value="#{bundle.welcomeMessage}" />
</f:view>
```

HTML Output

welcome_en.jsp (English)

```
Welcome to our site!
```

welcome_fr.jsp (French)

```
Bienvenue à notre site!
```

13.6 structura unei aplicații JSF

Iată o structură de directori tipică pentru o aplicație JSF. Folderul **myJSFapp** este directorul de bază al aplicației.

```
myJSFapp
   /ant
      build.xml
   /JavaSource
   /WebContent
      /WEB-INF
         /classes
         /lib
            jsf-impl.jar
            jsf-api.jar
         faces-config.xml
         web.xml
      /pages
```

Comentarii asupra acestor structuri:

- **myJSFapp** – directorul de bază al aplicației cu numele aplicației
- /**ant** – folder ce conține scripturi de build Ant cu un fișier implicit **build.xml**
- /**JavaSource** – surse Java pentru clase și fișiere de proprietăți specifice aplicației
- /**WebContent** – conține fișierele aplicației utilizate de către serverul de aplicații sau de către containerul web
- /**WEB-INF** – conține fișiere utilizate ca parte a zonei de execuție a aplicației web
- /**classes** – clase Java compilate precum și fișiere de proprietăți copiate din folderul /JavaSource
- /**lib** – conține biblioteci cerute de către aplicație, de obicei ca arhive jar externe
- **jsf-impl.jar**, **jsf-api.jar** – fișiere incluse în folderul /lib, obligatorii pentru orice aplicație JSF
- **web.xml** – descriptorul de desfășurare, inclus în folderul /WEB-INF
- **faces-config.xml** – fișierul de configurare JSF, inclus în folderul /WEB-INF
- /**pages** – folder conținând fișiere JSP și HTML de prezentare

13.7 cum lucrează JSF? un exemplu

Exemplu preluat de la http://www.exadel.com/tutorial/jsf/jsftutorial-kickstart.html.

O aplicație JSF nu este nimic altceva decât o aplicație servlet/JSP. Are un descriptor de desfășurare, pagini JSP, biblioteci de elemente particularizate, resurse statice, etc. Ceea ce o distinge de ele este faptul că o aplicație JSF este condusă (driven) de evenimente. Comportamentul aplicației este dictat de către o clasă de ascultare a evenimentelor (event listener).

Prezentăm acum pașii principali în construirea unei aplicații JSF:

1. Crearea paginilor JSP
2. Definirea regulilor de navigare
3. Crearea bean-urilor asociate
4. Crearea fișierelor de proprietăți
5. Editarea paginilor JSP
6. Crearea unui fișier index.jsp
7. Compilarea aplicației
8. Desfășurarea și executarea aplicației

13.7.1 crearea paginilor JSP

În exemplul nostru, creăm fișierele **inputname.jsp** și **greeting.jsp** în folderul **WebContent/pages/**. Trebuie create doar fișierele, deoarece structura de directori e deja creată.

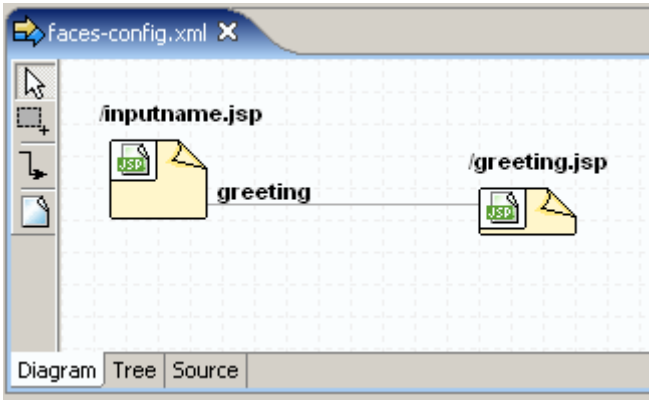
Aceste fișiere sunt goale, deocamdată. Vom completa conținutul lor ceva mai târziu.

Acum că avem cele două pagini JSP, putem crea o regulă de navigare.

13.7.2 navigare

Conceptul de navigare este în centrul tehnologiei JavaServer Faces. Regulile de navigare pentru această aplicație (și nu numai) sunt descrise în fișierul **faces-config.xml**. Acest fișier există deja în structura de directori creată automat. Trebuie doar creat conținutul fișierului

În aplicația noastră, vrem să navigăm doar de la pagina **inputname.jsp** la pagina **greeting.jsp**. Ca diagramă, ar arăta cam așa:



Imagine din Exadel Studio Pro

Regula de navigare arătată mai sus este definită mai jos. Regula spune că sursa regulii de navigare este pagina **inputname.jsp** iar destinația este **greeting.jsp**, dacă rezultatul (outcome) execuției **inputname.jsp** este **greeting**. Cam asta este totul.

```
<navigation-rule>
  <from-view-id>/pages/inputname.jsp</from-view-id>
  <navigation-case>
    <from-outcome>greeting</from-outcome>
    <to-view-id>/pages/greeting.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Desigur că aceasta este o regulă de navigare foarte simplă. Reguli mai complexe pot fi create, însă, cu ușurință. Pentru mai multe infoemații despre regulile de navigare, citiți în forumul JSP Navigation Example.

13.7.3 crearea bean-ului administrat (managed Bean)

Vom crea acum un folder **myJFSapp** în interiorul folderului **JavaSource**. În folderul **myJFSapp** vom crea un fișier **PersonBean.java**. Această clasă java este foarte simplă. Este un bean Java cu un singur atribut și metodele de get și set corespunzătoare. Rolul bean-ului este a captura numele introdus de către utilizator, după ce acesta apasă pe butonul de submit. În acest fel, bean-ul oferă o punte de legătură între pagina JSP și logica aplicației, De remarcat că numele câmpului din fișierul JSP trebuie să coincidă cu numele atributului din bean.

13.7.3.1 PersonBean.java

Scrieți codul de mai jos în fișier:

```
package myJFSapp;


public class PersonBean {
   String personName;
   /**
   * @return Person Name
   */
   public String getPersonName() {
      return personName;
   }
   /**
   * @param Person Name
   */
   public void setPersonName(String name) {
      personName = name;
   }
}
```

Mai târziu vom vedea cum conectăm acest bean cu pagina JSP.

13.7.3.2 declararea bean-ului în fișierul faces-config.xml

În partea a 2-a a fișierului **faces-config.xml** vom descrie bean-ul nostru, creat anterior. Această secțiune definește un nume de bean: **PersonBean**. Următoarea linie este numele complet al clasei: **myJFSapp.PersonBean**. Următoarea linie setează **request** ca fiind scopul (obiectul de care este atașat) bean-ului în aplicație.

```
<managed-bean>
  <managed-bean-name>personBean</managed-bean-name>
  <managed-bean-class>myJFSapp.PersonBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

13.7.3.3 fișierul faces-config.xml

Varianta finală a fișierului **faces-config.xml** arată astfel:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
  <navigation-rule>
   <from-view-id>/pages/inputname.jsp</from-view-id>
    <navigation-case>
     <from-outcome>greeting</from-outcome>
     <to-view-id>/pages/greeting.jsp</to-view-id>
   </navigation-case>
  </navigation-rule>
  <managed-bean>
    <managed-bean-name>personBean</managed-bean-name>
    <managed-bean-class>myJFSapp.PersonBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</faces-config>
```

13.7.4 crearea unui fișier de proprietăți (Resource Bundle)

Un fișier de proprietăți este pur și simplu un set de perechi parametru=valoare. Mesajele ce vor fi afișate în paginile noastre vor fi stocate în fișierul de proprietăți. Ținându-le separat de pagina JSP permite modificarea rapidă a acestora fără a fi nevoie să editâm pagina JSP.

Să cream un folder **bundle** în folderul **JavaSource/myJFSapp** și apoi un fișier **messages.properties** în folderul **bundle**. Va trebui apoi repoziționat în folderul **JavaSource** pentru ca la compilarea proiectului acest fișier de proprietăți să fie copiat în folderul **classes** unde va fi găsit în momentul execuției.

13.7.4.1 messages.properties

Puneți acest text în fișierul de proprietăți.

```
inputname_header=JSF KickStart
prompt=Tell us your name:
greeting_text=Welcome to JSF
button_text=Say Hello
sign=!
```

Avem acum tot ce ne trebuie pentru a crea paginile JSP.

13.7.5 Editarea paginilor JSP

La această oră două pagini (goale) ar trebui deja să fie create în **myJFSapp/WebContent/pages**.

13.7.5.1 inputname.jsp

Puneți următorul cod în acest fișier:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="myJFSapp.bundle.messages" var="msg"/>

<html>
 <head>
  <title>enter your name page</title>
 </head>
 <body>
   <f:view>
     <h1>
      <h:outputText value="#{msg.inputname_header}"/>
     </h1>
     <h:form id="helloForm">
      <h:outputText value="#{msg.prompt}"/>
      <h:inputText value="#{personBean.personName}" required=”true”>
        <f:validateLength minimum="2" maximum="10"/>
      </h:inputText>
      <h:commandButton action="greeting" value="#{msg.button_text}" />
     </h:form>
   </f:view>
 </body>
</html>
```

Să trecem acum în revistă principalele secțiuni din acest fișier, începând cu începutul.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="myJFSapp.bundle.messages" var="msg"/>
```

Prima din aceste trei linii este o directivă JSP care ne spune unde se găsesc elementele JSF care definesc elemente HTML, iar cea de-a doua directivă ne spune unde se găsesc elementele JSF care definesc elemente core (din nucleul) JSF. A treia linie servește la încărcarea fișierului de proprietăți (resource bundle) care conține mesajele ce urmează a fi afișate în paginile JSP.

```
<h:inputText value="#{msg.inputname_header}" required=”true”>
```

Acest element ne spune să căutăm în fișierul de proprietăți (resource bundle) pe care l-am definit la începutul paginii. Atributul **required** al elementului **h:inputText** ne asigură că un nume gol nu va fi trimis. Se mai poate adăuga o linie precum:

```
<f:validateLength minimum="2" maximum="10"/>
```

pentru a ne asigura că lungimea acestui câmp este rezonail de mare.

Apoi, căutați valoarea parametrului **inputname_header** în acest fișier și afișati-l aici:

```
1 <h:form id="helloForm">
2   <h:outputText value="#{msg.prompt}"/>
3   <h:inputText value="#{personBean.personName}" required=”true”>
4     <f:validateLength minimum="2" maximum="10"/>
5   </h:inputText>
6   <h:commandButton action="greeting" value="#{msg.button_text}" />
7 </h:form>
```

Linia 1. Crează o formă HTML utilizând elemente JSF.

Linia 2. Afișează un mesaj din fișierul de proprietăți utilizând valoarea parametrului **prompt**.

Liniile 3-5. Crează o câmp HTML de introducere text. Prin atributul **value**, conectăm acest câmp cu bean-ului administrat creat mai înainte.

Linia 6. Elemente JSF pentru butonul HTML de submit. Textul afișat pe buton este luat din fișierul de proprietăți, iar valoarea atributului **action** este setată ca **greeting**, cea ce coincide cu valoarea parametrului **outcome** din fișierul **faces-config.xml** file. Așa află JSF unde să meargă la pasul următor.

13.7.5.2 greeting.jsp

Scrieți acest cod în al doilea fișier JSP.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="myJFSapp.bundle.messages" var="msg"/>
```

```
<html>
  <head>
   <title>greeting page</title>
  </head>
  <body>
     <f:view>
        <h3>
         <h:outputText value="#{msg.greeting_text}" />,
         <h:outputText value="#{personBean.personName}" />
         <h:outputText value="#{msg.sign}" />
        </h3>
     </f:view>
 </body>
</html>
```

Această pagină este foarte simplă. Primele trei linii sunt identice cu cele ale primei pagini. Aceste linii importă bibliotecile de elemente JSF precum și fișierul de proprietăți ce conține mesaje.

Partea de cod ce ne interesează este cuprinsă între elementele **<h3>...</h3>**. Prima linie va lua un mesaj din fișierul de proprietăți și-l va afișa pe pagină. Al doilea va accesa atributul **personName** al bean-ului și îi va afișa conținutul în pagină.

Atunci când această pagină va fi afișată în browser, veți vedea ceva de genul:

```
Welcome to JSF, Vasile Popescu!
```

13.7.6 crearea fișierului index.jsp

Vom crea acum un al treilea fișier JSP care nu va mai funcționa ca o pagină de prezentare. Utilizează o acțiune standard JSP care ne direcționează (forwards) la pagina **inputname.jsp.**

Creați fișierul **index.jsp** în folderul **WebContent**. De remarcat că acest fișier nu este creat în folderul **pages** precum celelalte fișiere JSP.

Având un fișier implicit **index.jsp** ne va permite să pornim aplicația din browser astfel:

```
http://localhost:8080/myJFSapp/
```

Acest fișier va conține doar o acțiune standard JSP de forward:

```
<html>
 <body>
  <jsp:forward page="/pages/inputname.jsf" />
 </body>
</html>
```

Dacă ne uităm la numele fișierului către care se face redirectarea, vom observa că extensia sa este **.jsf** și nu **.jsp**. Aceasta deoarece în descriptorul de desfășurare `web.xml`, pentru aplicație, `*.jsf` este modelul (pattern) URL utilizat pentru a semnala că pagina de redirecționare va trebui procesată către servletul JavaServer Faces din cadrul Tomcat.

Suntem aproape de terminarea acestui exemplu.

13.7.7 compilare

Aceasta este realizată prin execuția unui script de build de tip Ant. Pentru a construi aplicația, se execută scriptul **build.xml** din folderul **ant**:

```
ant build
```

13.7.8 desfășurare

Înainte de executarea acestei aplicații în containerul web, trebuie ca ea să fie desfășurată. Pentru aceasta trebuie să înregistrăm un element de context în fișierul de configurare la nivel de server, numit, în acest caz: **{TomcatHome}\conf\server.xml** file.

Codul arată astfel:

```
<Context debug="0"
docBase="Path_to_WebContent"
path="/myJFSapp" reloadable="true"/>
```

și este localizat pe la sfârșitul fișierului **server.xml**, în interiorul elementului **Host**, chiar înainte de eticheta de încheiere **</Host>**. Desigur, **Path_to_WebContent** trebuie să fie înlocuit cu specificația exactă de folder pentru sistemul curent, indicând locația folderului **WebContent**, în interiorul folderului **myJFSapp**. (de examplu, `C:/examples/myJFSapp/WebContent`).

13.7.9 execuție

Se pornește serverul Tomcat (probabil executând scriptul **startup.bat** din folderul **bin** al Tomcat-ului). După ce Tomcat-ul este încărcat, porniți un browser web cu adresa: **http://localhost:8080/myJFSapp**. (Portul 8080 este portul implicit în Tomcat. Posibil, însă, că unii dintre utilizatori să aibă o configurație (port) diferit(ă)).

13.8 pachete din API-ul pentru JavaServer Faces

Clasele și interfețele din API-ul pentru JavaServer Faces sunt grupate în câteva pachete, și anume:

- javax.faces
- javax.faces.application
- javax.faces.component
- javax.faces.component.html
- javax.faces.context
- javax.faces.convert
- javax.faces.el
- javax.faces.event

- javax.faces.lifecycle
- javax.faces.model
- javax.faces.render
- javax.faces.validator
- javax.faces.webapp

Vom trece în revistă doar trei din aceste pachete, și anume: javax.faces.convert, javax.faces.validator and javax.faces.webapp

13.9 pachetul java.faces.convert

13.9.1 interfața Converter

Converter este o interfață care descrie o clasă Java care poate efectua conversii Object-to-String și String-to-Object între modele de obecte de date și reprezentarea ca String a acestor obiecte, reprezentare care e potrivită pentru redare.

Clasele care implementează interfața în acest pachet sunt:

- **BigDecimalConverter**
- **BigIntegerConverter**
- **BooleanConverter**
- **ByteConverter**
- **CharacterConverter**
- **DateTimeConverter**
- **DoubleConverter**
- **EnumConverter**
- **FLoatConverter**
- **IntegerConverter**
- **LongConverter**
- **NumberConverter**
- **ShortConverter**

Pachetul conține și o excepție:

- **ConverterException** - o excepție aruncată de către metoda `getAsObject()` sau `getAsText()` a unui `Converter`, pentru a indica că metoda cerută pentru conversie nu opate fi efectuată

13.10 pachetul java.faces.validator

Interfață care definește modelul de validare precum și clase concrete de validare.

O implementare a interfeței **Validator** este o clasă care poate efectua validări pe un `EditableValueHolder`.

Clase de implementare:

- **DoubleRangeValidator** - un `Validator` care verifică dacă valoarea componentei corespunzătoare este în domeniul specificat (valoare minimă și maximă)
- **LengthValidator** - un `Validator` care verifică numărul de caractere în reprezentarea ca String a componentei asociate
- **LongRangeValidator** - un `Validator` care verifică dacă valoarea componentei corespunzătoare este în intervalul de minim și maxim specificat

Pachetul conține, de asemenea, și o excepție.

O excepție **ValidatorException** este aruncată de către metoda `validate()` a unui `Validator` pentru a indica nereușita validării.

13.11 pachetul java.faces.webapp

Conține clase cerute pentru integrarea JavaServer Faces în aplicații web, incluzând un servlet standard, clase de bază pentru elemente de componente JSP precum și implementări concrete pentru elementele din nucleu (core).

- **AttributeTag** – implementare de element care adaugă un atribut cu numele specificat și o valoare de tip String pentru componenta al cărei element este conținut în interior, dacă componenta nu conține deja un atribut cu același nume.
- **ConverterTag** – o clasă de bază pentru toate acțiunile particularizate (custom) JSP care crează și înregistrează o instanță de `Converter` pe un `ValueHolder` asociat cu vecinătatea imediată a unei instanțe a unui element a cărui clasă de implementare este o subclasă a `UIComponentTag`.
- **FacesServlet** – un servlet care administrează ciclul de viață al procesării cererilor pentru aplicații web care utilizează JavaServer Faces pentru a construi interfața utilizator.
- **FacetTag** – mecamismul JSP pentru a specifica că o `UIComponent` urmează să fie adăugată ca `facet` la componenta asociată cu părintele ei.
- **UIComponentBodyTag** – o clasă de bază pentru acțiuni JSP, referitoare la componente UI, care trebuie să proceseze conținutul elementelor.
- **UIComponentTag** – clasa de bază pentru toate acțiunile JSP care corespund componentelor de interfață utilizator (UI) dintr-o pagină care easte redată (afișată) de către JavaServer Faces.
- **ValidatorTag** - o clasă de bază pentru acțiuni JSP care crează și înregistrează o instanță de `Validator` pe un `EditableValueHolder` asociat cu cea mai apropiată instanță a unui element a cărui clasă de implementare este o subclasă a `UIComponentTag`.

13.12 ciclul de viață JSF

Indiferent de faptul că utilizați JSF cu JSP, cu servlete sau cu o altă tehnologie, fiecare curs cerere/răspuns care implică JSF urmează un anumit ciclu de viață. Câteva variante de cerere/răspuns pot avea loc într-o aplicație JSF. Puteți avea o cerere care vine de la o pagină JSF redată anterior (o cerere JSF) sau o cerere care vine de la o pagină non-JSF (o cerere non-JSF). La fel, putem avea un răspuns JSF sau un răspuns non-JSF. La fel, putem avea de a face cu un răspuns JSF sau cu un răspuns a non-JSF. Suntem interesați de următoarele combinații:

- cerere non-JSF care generează un răspuns JSF
- cerere JSF care generează un răspuns JSF
- cerere JSF care generează un răspuns non-JSF

Desigur, putem avea și cazul unei cereri non-JSF care generează un răspuns non-JSF. Dar fiindcă un astfel de scenariu nu implică în nici un fel JSF, ciclul de viață al JSF-ului nu se aplică.

Paginile JSP au un ciclu de viață relativ simplu. O sursă JSP este compilată într-o clasă de implementare

a paginii. Când un server web primește o cerere, această cerere este trimisă containerului web, care o trimite mai departe către clasa de implementare a paginii. Clasa paginii procesează cererea și apoi crează un răspuns care este trimis înapoi clientului. când sunt implicate alte pagini sau cererea este redirecționată, sau când are loc o excepție, procesarea implică mai multe componente sau pagini, dar, în principiu, un număr mic de clase procesează cererea și trimit înapoi un răspuns.

Atunci când utilizăm JSF, lucrurile sunt mai complicate. Aceasta deoarece nucleul JSF este bazat pe pe modelul MVC (Model View Controller), ceea ce are câteva implicații. Acțiunile utilizatorului în vederi generate de JSF au loc în zona clientului care nu a re o conexiune permanentă cu serverul. Trimiterea acțiunilor clientului sau a evenimentelor la nivel de pagină este amânată pînă la (re)stabilirea conexiunii. Ciclul de viață al unui JSF trebuie să ia în considerare întârzierea dintre evenment și procesarea acestuia. De asemenea, ciclul de viață la JSF trebuie să asigure că vederea este corectă înainte de a fi redată (afișată). Pentru a se asigura că starea curentă este totdeauna validă, sistemul JSF include o fază de validare pentru introduceri și o alta pentru adaptarea modelului după validarea tuturor introducerilor.

În MVC, prezentarea datelor (vederea) este separată de reprezentarea lor în sistem (modelul). când modelul este modificat, controller-ul trimite un mesaj către vedere, spunându-i să-și modifice prezentarea. când utilizatorul modifică prezentarea, controller-ul trimite un mesaj model-ului, spunându-i să-și aducă la zi datele. În JSF, modelul este compus din obiecte business care sunt de obicei implementate ca bean-uri Java, controller-ul este chiar implementarea JSF iar vederea este compusă din componentele de interfață utilizator (UI).

Ciclul de viață al JSF constă din șase faze, conform specificației JSF.

• **Restaurarea vederii:** În această fază, implementarea JSF restaurează obiectele și structurile de date care reprezintă vederea cererii. Dacă aceasta este prima vizită a clientului la pagină, implementarea JSF trebuie să creeze vederea. Atunci când implementarea JSF crează și redă o pagină JSF, ea crează obiecte de UI pentru fiecare componentă a vederii. Componentele sunt stocate într-un arbore de componente, iar starea vederii UI este salvată pentru cererile ulterioare. Dacă aceasta este o cerere ulterioară, vederea UI salvată anterior este restaurată pentru procesarea cererii curente.

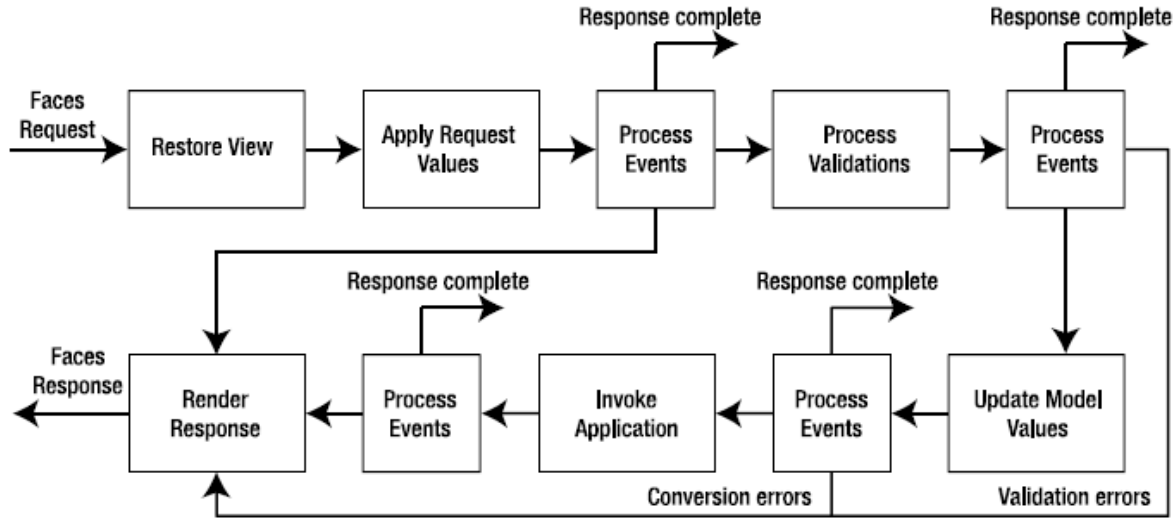
• **Aplicarea valorilor cererii:** Toate datele care au trimise ca parte a cererii sunt trimise către obiectele UI corespunzătoare care compun vederea. Aceste obiecte sunt actualizate cu valorile din aceste date. Datele pot proveni din câmpurile de introducere ale unei forme web, din cookies trimise ca parte a cererii, sau din header-ele cererii. Datele pentru anumite componente, precum componentele care crează câmpuri de introducere HTML, sunt validate în acest moment. Acest lucru încă nu actualizează obiectele de business care fac parte din model. Are loc doar actualizarea componentelor de interfață utilizator (UI) cu noile date.

• **Procesul de validare: D**atele care au trimise de către formă sunt validate acum (dacă nu au fost deja validate). La fel ca în faza precedentă, aceasta nu duce încă la actualizarea obiectelor de business (modelul) din aplicație. Aceasta deoarece dacă implementarea JSF ar fi început deja actualizarea modelului odată cu validarea datelor, ar fi posibil ca unele date să nu fie validate iar datele din model să fie într-o stare invalidă.

• **Actualizarea valorilor modelului:** După terminarea validărilor, obiectele de business din aplicație sunt actualizate cu datele validate din cerere. În plus, dacă unele din date trebuie convertite în alt format pentru actualizarea modelului (de exemplu, conversia unui String într-un obiect de tip Date), conversia are loc acum. Conversia este necesară atunci când tipul datei nu este un String sau un tip Java primitiv.

• **Invocarea aplicației:** În această fază, se apelează resursa specificată de atributul action sau de către un link. În plus, orice alt event generat în una din fazele anterioare care încă nu a fost procesat este trimis aplicației web pentru a finaliza orice altă procesare cerută.

• **Redarea (afișarea) răspunsului: C**omponentele UI ale răspunsului sunt redate iar răspunsul este trimis clientului. Starea componentelor UI este salvată pentru ca arborele componentelor să poată fi restaurat atunci când clientul trimite altă cerere. Pentru o aplicație JSF, un fir de execuție (thread) pentru un ciclu cerere/răspuns poate trece prin fiecare fază, în ordinea afișată în figura de mai jos. Totuși, depinzând de cerere și de ceea ce se întâmplă în timpul procesării cererii și trimiterii răspunsului, nu orice cerere va trece prin toate cele șase faze.



În figura de mai sus se pot vedea un număr de pași opționali prin ciclul de viață. De exemplu, dacă au loc erori în oricare din faze, cursul de execuție este transferat imediat către faza Redare răspuns (Render Response), sărind peste fazele rămase. Acest lucru se poate întâmpla, de exemplu, dacă datele introduse sunt incorecte sau ne-valide. Dacă validarea datelor nu are loc fie în faza Aplicarea valorilor cererii (Apply Request Values) sau în Procesul de validare (Process Validations), informațiile despre erori sunt salvate și procesarea continuă direct cu faza Redare răspuns. De asemenea, dacă în orice moment al ciclului de viață procesarea răspunsului este completă și un răspuns non-JSF este trimis către client, cursul de execuție poate ieși din ciclul de viață fără completarea fazelor ulterioare.

chapter 14 Java Message Service (JMS)

14.1 elemente JMS

API-ul pentru **Java Message Service** (**JMS**) este un Java Message Oriented Middleware (MOM) API pentru trimiterea mesajelor între doi sau mai mulți clienți. JMS este parte a Java Platform, Enterprise Edition, și este definit printr-o specificație din cadrul Java Community Process, și anume JSR 914 - https://jcp.org/en/jsr/detail?id=914

JMS itself is specified (the latest major version – 2.0) în JSR 343 – https://jcp.org/en/jsr/detail?id=343.

Elementele (conceptele) de bază în JMS sunt:

- **provider JMS –** o implementare a interfeței JMS pentru un Message Oriented Middleware (MOM). Providerii sunt implementați fie ca implementare JMS Java, sau ca un adaptor la un MOM non-Java.
- **client JMS** – o aplicație sau un proces care produce sau consumă mesaje
- **producător JMS** – un client JMS care crează și trimite mesaje
- **consumator JMS –** un client JMS care primește mesaje
- **mesaj JMS** – un obiect care conține datele ce sunt transferate între clienții JMS
- **coadă JMS –** o zonă de asamblare care conține mesajele care au fost trimise și care așteaptă să fie citite. După cum numele de *coadă* sugerează, mesajele sunt citite în ordinea în care au fost trimise. Un mesaj este scos din coadă după ce a fost citit.
- **topic JMS –** un mecanism de distribuție pentru publicarea mesajelor ce sunt trimise la mai mulți abonați

14.2 modele JMS

API-ul pentru JMS oferă suport pentru două modele:

- modelul **de la punct la punct** (point-to-point) sou coadă
- modelul **publicare și abonare** (publish and subscribe)

În modelul de la punct la punct, un *producător* afișează mesaje pe o anume coadă iar un *consumator* citește mesajele de pe coadă. În acest caz, producătoru cunoaște destinația mesajului și afișează (postează) mesajul direct pe coada consumatorului. Acest model are următoarele caracteristici:

- Un singur consumator va obține mesajul
- Producătorul mesajului nu trebuie să fie în stare de execuție în momentul în care consumatorul citește mesajul, la fel, nici consumatorul nu trebuie să fie în stare de execuție atunci când mesajul este trimis
- Fiecare mesaj procesat cu succes este confirmat (acknowledged) de către consumator

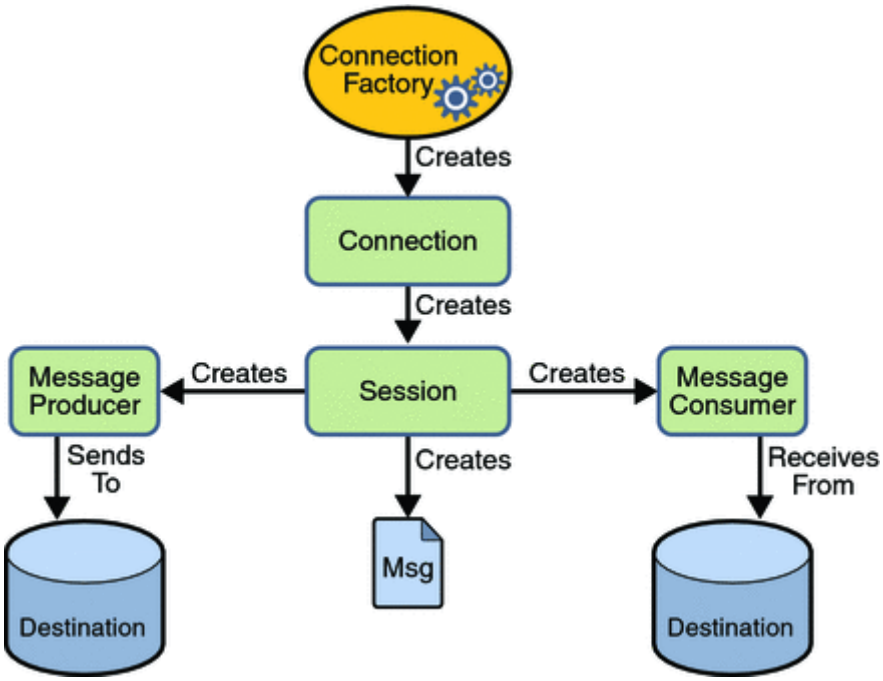
Modelul publicare/abonare oferă suport pentru afișarea mesajelor pentru un anumit topic (subiect). Zero sau mai mulți abonați își pot înregistra interesul în a primi mesaje despre un anume topic (subiect). În acest model, nici producătorul mesajului și nici abonatul nu știu unul de celălalt. O bună comparație ar fi cu un panou de afișaj (bulletin board). Acest model are următoarele caracteristici:

- Mai mulți consumatori primesc mesajul
- Există o dependență temporală între producători (publishers) și abonați. Producătorul trebuie să creeze un sistem de abonare (subscription) pentru ca clienții să se poată abona. Abonatul trebuie să fie activ în mod continuu pentru a putea primi mesajele, cu excepția cazului în care abonamentul său este durabil. În acest caz, mesajele publicate atunci când abonatul nu este conectat vor fi redistribuite atunci când acesta se reconectează.

JMS oferă o modalitate de separare a aplicației față de nivelul de transport care aduce datele. Aceleași

clase Java pot fi utilizate pentru comunicarea cu provideri diferiți de JMS, utilizând informațiile JNDI pentru alegerea unui provider convenabil. Clasele utilizează o fabrică de conexiuni (connection factory) pentru a se conecta la o coadă sau la un topic, iar apoi utilizează metodele `populate()` și `send()` pentru a publica sau trimite mesajele. Pe parte de recepție a mesajelor, clientul primește sau se abonează la mesaje.

14.3 modelul de programare pentru API-ul JMS



14.4 API-ul pentru JMS

API-ul pentru JMS este cuprins în pachetul `javax.jms`.

14.4.1 interfața ConnectionFactory

Este un obiect administrat care este utilzat de către clienți pentru a crea o conexiune cu un provider de JMS. Clienții JMS accesează fabrica de conexiuni prin interfețe portabile astfel încât la modificarea implementării, codul nu trebuie schimbat. Administratorii configurează fabrica de conexiuni în spațiul de numire JNDI pentru ca clienții să le poată găsi. În funcție de tipul mesajelor, utilizatorii vor folosi fie o fabrică de conexiuni pentru cozi sau pentru topicuri.

La începutul unui program client pentru JMS se efectuează, de obicei, căutarea unei fabrici de conexiuni prin intermediul JNDI, apoi se efectuează un cast atribuit unui obiect de tip `ConnectionFactory`.

De exemplu, codul ce urmează obține un obiect de tip `InitialContext` care este utilizat apoi pentru a căuta după nume, un obiect de tip `ConnectionFactory`. Acesta este apoi atribuit unui obiect `ConnectionFactory`:

```
Context ctx = new InitialContext();
```

```
ConnectionFactory connectionFactory = (ConnectionFactory)
   ctx.lookup("jms/ConnectionFactory");
```

Într-o aplicație J2EE, obiectele JMS administrate sunt puse de obicei în subcontextul `jms`.

14.4.2 interfața Connection

După ce obținem o fabrică de conexiuni, putem crea o conexiune la un provider JMS. O conexiune reprezintă o legătură de comunicare între o aplicație și serverul de mesaje. În funcție de tipul conexiunii, conexiunile permit utilizatorilor să creeze sesiuni pentru trimiterea și primirea de mesaje, (de) la o coadă sau un topic.

Conexiunile implementează interfața `Connection`. Atunci când aveți un obiect de tip `ConnectionFactory`, îl puteți utiliza pentru a crea un obiect de tip `Connection`:

```
Connection connection = connectionFactory.createConnection();
```

Înainte de a termina o aplicație, trebuie închise toate conexiunile create anterior. Dacă nu faceți acest lucru, resursele utilizate nu vor putea fi eliberate de către provider-ul JMS. Închiderea unei conexiuni duce și la închiderea sesiunilor asociate, precum și a producătorilor și consumatorilor de mesaje.

```
connection.close();
```

Înainte ca aplicația voastră să poată consuma mesaje, trebuie apelată metoda `start()` a conexiunii. Dacă doriți să opriți temporar livrarea mesajelor fără a închide conexiunea, se poate apela metoda `stop()`.

14.4.3 interfața Destination

Un obiect administrat, care encapsulează identitatea unei destinații de mesaj (locul în care mesajul este livrat și consumat). Poate fi o coadă sau un topic. Administratorul JMS crează aceste obiecte iar utilizatorii le descoperă utilizând JNDI. La fel ca și fabrica de conexiuni, administratorul poate crea două tipuri de destinații: cozi pentru modelul de la punct la punct și topicuri pentru modelul publicare/abonare.

De exemplu, linia de cod de mai jos realizează o căutare JNDI a unui topic creat anterior: `jms/MyTopic` pe care îl convertește (cast) și îl atribuie unui obiect de tip `Destination`:

```
Destination myDest = (Destination) ctx.lookup("jms/MyTopic");
```

Următoarea linie de cod caută o coadă numită `jms/MyQueue` pe care o convertește și o atribuie unui obiect de tip `Queue`:

```
Queue myQueue = (Queue) ctx.lookup("jms/MyQueue");
```

14.4.4 interfața MessageConsumer

Un obiect creat de către o sesiune. Primește mesaje trimise la o destinație. Consumatorul poate primi mesajele sincron (cu blocare) sau asincron (fără blocare) atît pentru cozi cât și pentru topicuri.

De exemplu, puteți utiliza un `Session` pentru a crea un `MessageConsumer` pentru o coadă sau pentru un topic:

```
MessageConsumer consumer = session.createConsumer(myQueue);
MessageConsumer consumer = session.createConsumer(myTopic);
```

Puteți utiliza metoda `Session.createDurableSubscriber()` pentru a crea un abonat la un topic

durabil. Metoda este validă doar dacă utilizați un topic.

După crearea unui consumator de mesaje, acesta devine activ și poate fi utilizat pentru primirea de mesaje. Puteți utiliza metoda `close()` a unui `MessageConsumer` pentru a inactiva consumatorul de mesaje. Livrarea mesajelor nu începe pînă nu porniți conexiunea creată prin apelarea metodei sale `start()`.

Pentru a consuma sincron un mesaj, se folosește metoda `receive()`. Metoda poate fi utilizată oricând după apelarea metodei `start()`:

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a second
```

Pentru consumarea asincronă a unui mesaj, trebuie utilizat un obiect de ascultare a mesajelor.

14.4.5 interfața MessageListener

Un **ascultător de mesaje** (**message listener**) este un obiect care acționează ca un procesator de evenimente asincron (asynchronous event handler) pentru mesaje. Acest obiect implementează interfața `MessageListener` care conține o singură metodă: `onMessage()`. În metoda `onMessage()` se definesc acțiunile luate atunci când sosește un mesaj.

Înregistrarea unui ascultător de mesaje (message listener) cu un anume `MessageConsumer` se face prin utilizarea metodei `setMessageListener()`. De exemplu, dacă definiți o clasă numită `Listener` care implementează interfața `MessageListener`, puteți înregistra ascultătorul de mesaje astfel:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

După ce înregistrați ascultătorul de mesaje, puteți apela metoda `start()` a obiectului de tip `Connection` pentru a începe livrarea mesajului. (Dacă apelați `start()` înainte de înregistrarea ascultătorului de mesaje, e foarte probabil că veți rata mesaje).

Atunci când începe livrarea mesajelor, provider-ul JMS apelează automat metoda `onMessage()` a ascultătorului de mesaje. Metoda `onMessage()` are un singur argument, de tip `Message`, care este convertit (cast) la tipul corespunzător de mesaj de către implementarea voastră a metodei.

Un ascultător de mesaje nu este specific pentru un anumit tip de destinație. Același ascultător poate obține mesaje atît din partea unei cozi cât și din partea unui topic, depinzând de tipul destinației pentru care a fost creat consumatorul de mesaje. Un ascultător de mesaje așteaptă, totuși, un anume tip de mesaj și un anumit format. Mai mult decât atît, dacă trebuie să răspundă mesajelor, un ascultător de mesaje fie trebuie să presupună un anume tip de destinație sau să obțină tipul destinației mesajului și să creeze un producător pentru acel tip de destinație.

14.4.6 interfața MessageProducer

Un obiect creat de către o sesiune și care trimite mesaje la o destinație. Utilizatorul poate crea un producător pentru o destinație specifică sau poate crea un producător generic care precizează tipul destinației în momentul trimiterii mesajului.

Se poate utiliza un obiect de tip `Session` pentru a crea un `MessageProducer` pentru o destinație. În primul exemplu de mai jos, se crează un producător pentru destinația `myQueue`, iar în cel de-al doilea pentru destinația `myTopic`:

```
MessageProducer producer = session.createProducer(myQueue);
```

```
MessageProducer producer = session.createProducer(myTopic);
```

Se poate crea un producător generic specificând valoarea `null` ca argument pentru metoda `createProducer()`. Cu un producător generic, destinația poate fi specificată doar atunci când trimiți mesajul.

După crearea producătorului de mesaje, acesta poate fi utilizat pentru pentru trimiterea de mesaje cu ajutorul metodei `send()`:

```
producer.send(message);
```

În cazul unui producător generic, mai întâi trebuie creat mesajul, apoi trebuie utilizată o metodă `send()` supraîncărcată (overloaded) care specifică destinația ca prim parametru. De exemplu:

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(myQueue, message);
```

14.4.7 interfața Message

Un obiect care constituie obiectul schimburilor dintre consumatori și producători, adică de la o aplicație la alta. Un mesaj are trei părți principale:

1. Un **antet** (**header**) (obligatoriu): conține setări operaționale pentru identificarea și rutarea mesajelor
2. Un set de **proprietăți** ale mesajelor (opțional): conține proprietăți adiționale pentru menținerea compatibilității cu alți provideri sau utilizatori. Poate fi utilizat pentru a crea câmpuri particularizate sau filtre (selectori).
3. Un corp (**body**) al mesajului (opțional): permite utilizatorilor să creeze cinci tipuri de mesaje (mesaj **text,** mesaj **map**, mesaj **bytes**, mesaj **stream** și mesaj **object**).

Interfața **Message** este extrem de flexibilă și oferă numeroase posibilități de adaptare a conținutului mesajului.

API-ul pentru JMS oferă metode pentru crearea mesajelor de toate tipurile și pentru completarea conținutului lor. De exemplu, pentru a crea și trimite un `TextMessage`, puteți folosi următoarele instrucțiuni:

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);      // msg_text is a String
producer.send(message);
```

La capătul consumatorului, un mesaj sosește ca un `Message` generic și trebuie prezentat (cast) la tipul corepunzător. Pentru a extrage conținutul mesajului, se pot utiliza metode de tip get. Fragmentul de cod de mai jos utilizează metoda `getText()`:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage message = (TextMessage) m;
   System.out.println("Reading message: " + message.getText());
} else {
   // Handle error
}
```

14.4.8 interfața Session

Reprezintă un context pentru trimiterea și primirea mesajelor. O sesiune are un singur fir de execuție

(single-threaded) pentru ca mesajele să fie serializate, aceasta însemnând că mesajele sunt primite unul după altul, în ordinea în care au fost trimise. Beneficiul unei sesiuni este că oferă suport pentru tranzacții. Dacă utilizatorul selectează suport pentru tranzacții, contextul sesiunii reține un grup de mesaje pînă când tranzacția este finalizată (committed), apoi livrează mesajele. Înainte de finalizarea tranzacției, utilizatorul poate anula mesajele utilizând o operație de rostogolire (rollback). O sesiune permite utilizatorilor să creeze producători de mesaje, care transmit mesaje și consumatori de mesaje, care le primesc.

Sesiunile implementează interfața `Session`. După crearea unui obiect de tip `Connection`, acesta poate fi utilizat pentru crearea unuia de tip `Session`:

```
Session session = connection.createSession(false,
   Session.AUTO_ACKNOWLEDGE);
```

Primul argument înseamnă că sesiunea nu este tranzacționată, al doilea înseamnă că sesiunea confirmă automat mesajele care au fost primite cu succes.

Pentru a crea o sesiune cu suport pentru tranzacții, utilizați codul:

```
Session session = connection.createSession(true, 0);
```

Aici, primul argument înseamnă că sesiunea este tranzacționată, al doilea indică faptul că confirmarea mesajelor nu este specificată pentru sesiuni tranzacționate.

chapter 15 Enterprise Java Beans

15.1 enterprise java beans versus bean-uri java ordinare

Bean-urile Java studiate în contextul tehnologiei JSP oferă un format pentru componente de uz general, în timp ce arhitectura EJB (Enterprise Java Beans) oferă un format pentru componente de logica aplicației cu un înalt nivel de specializare.

Ce sunt EJB-urile? În principiu, o colecție de clase Java, împreună cu un fișier XML, legate într-o singură unitate. Clasele Java trebuie să se supună anumitor reguli și să ofere anumite metode de callback.

EJB-urile au ca mediu de execuție un container EJB care este (în general) parte a unui server de aplicații.

Versiunea 1.1 a specificației EJB preciza două tipuri de EJB-uri:

- **bean-uri de sesiune (session beans)** – utilizate de către un singur client (o extensie a clientului pe server), ciclul de viață al unui astfel de bean nu-l poate depăși pe cel al clientului
- **bean-uri de entitate (entity beans)** – reprezentări orientate obiect ale datelor dintr-o bază de date, mai mulți clienți pot să le acceseze simultan iar durata lor de viață coincide cu cea a datelor pe care le reprezintă. Bean-urile de entitate sunt înlocuite de un nou concept, entități de persistență (**Java Persistence API**), începând cu specificația EJB 3.0.

Specificația EJB 2.0 aduce un tip nou de bean-uri:

- **beanuri conduse de mesaje (message-driven beans)**

Specificația curentă (dec. 2018) este 3.2 și constituie obiectul JSR 345, a cărui versiune finală datează din mai 2013. Noutățile aduse de versiunea majoră 3.0 încearcă să facă procesul de dezvoltare al EJB-urilor mai ușor. Oferă un mecanism de adnotare pentru toate tipurile de metadate care erau furnizate anterior de către descriptorul de desfășurare, astfel încât furnizarea unui fișier de tip XML nu mai este necesară la desfășurarea bean-urilor, toate informațiile pertinente putând fi incluse acum în interiorul fișierelor de tip .jar care sunt desfășurate pe serverul de aplicații.

15.2 containerul EJB și serviciile sale

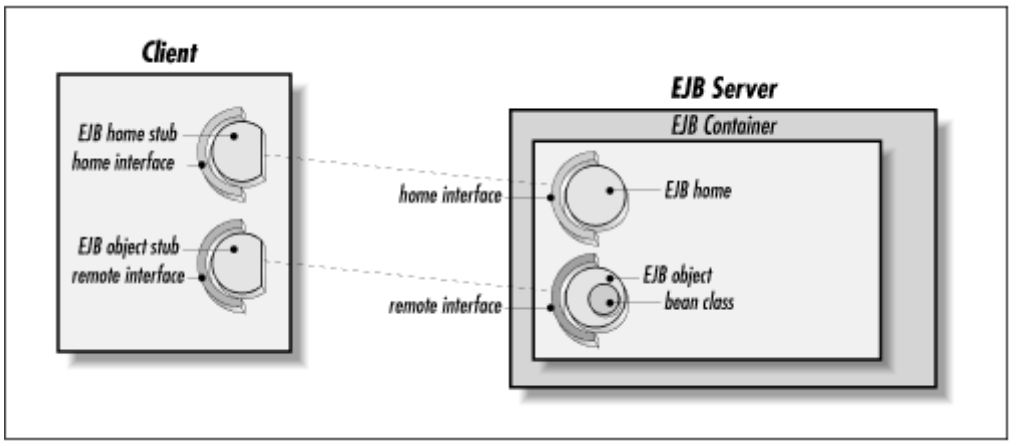
Containerul EJB oferă un mediu de execuție pentru componente de tip EJB. Componentele există în interiorul containerelor, care le oferă diferite servicii. Pe de altă parte, containerele există (în general) într-un server de aplicații, server care oferă un mediu de excuție pentru containere.

Principala rațiune a utilizării EJB-urilor ține de mulțimea și complexitatea serviciilor oferite de aceste containere.

Aceste servicii sunt:

- persistență – interacțiune cu bazele de date
- suport pentru tranzacții – administrarea tranzacțiilor poate fi complexă, în special dacă avem mai multe baze de date și mai multe componente care au nevoie de acces la ele
- cache pentru date – nu necesită cod creat de programator, performanțe ridicate
- securitate – accesul la EJB-uri poate fi reglementat fără cod specializat
- peocesarea erorilor – un cadru coerent pentru procesarea erorilor – notificare, recuperarea stării componentelor
- scalabilitate
- portabilitate
- suport pentru administrare

15.3 arhitectura enterprise java beans



Un EJB constă din (cel puțin) 3 clase și un fișier xml. Este sarcina programatorului de a le crea (cu ajutor din partea mediului de dezvoltare, în general) precum urmează:

1. bean-ul însuși (clasa care implementează logica aplicației)
2. interfața **home** a bean-ului
3. interfața **remote** a bean-ului
4. descriptorul de desfășurare, care e un fișier xml, numit `ejb-jar.xml`

15.4 interfața home

Interfața **home** a unui ejb este o interfață care extinde interfața `EJBHome`. Oferă metode numite `create()` cu argumente specifice pentru o anumită aplicație, returnând interfața **remote** și aruncând excepțiile `CreateException` și `RemoteException`. Folsește ca argumente numai tipuri de date permise de standardul RMI (Remote Method Invocation).

Handle – acest concept desemnează o referință de rețea (network reference) la un EJB.

Metodele specificate de către interfața `EJBHome` (dar ne-implementate (în general) de către programator) sunt următoarele:

```
public void remove(Handle han) throws RemoteException, RemoveException
public void remove(Object primaryKey) throws RemoteException,
RemoveException
public EJBMetaData getEJBMetaData() throws RemoteException
public HomeHandle getHomeHandle() throws RemoteException
```

Exemplu de cod pentru o interfață home, numită MyBeanHome:

```
package myBeans;
import.javax.ejb.*;
```

```
import java.rmi.RemoteException;
public interface MyBeanHome extends EJBHome
{
   MyBeanObject create() throws CreateException,
         RemoteException;
}
```

15.5 interfața remote

Interfața remote a unui ejb este o interfață Java standard care extinde interfețele `EJBObject` și `Remote` și declară metodele care implementează logica aplicației (business logic methods). Programatorul nu implementează această interfață.

Interfața `Remote` nu declară nici o metodă, dar interfața `EJBObject` declară următoarele metode:

```
public EJBHome getEJBHome() throws RemoteException
public Object getPrimaryKey() throws RemoteException
public Handle getHandle() throws RemoteException
public boolean isIdentical(EJBObject obj) throws
   RemoteException
public void remove() throws RemoteException, RemoveException
```

Exemplu de cod pentru o interfață numită `MyBeanObject`:

```
package myBeans;
import.javax.ejb.*;
import java.rmi.RemoteException;
public interface MyBeanObject extends EJBObject
{
   // assume that we have two business logic methods
   void processEntry(String firstName, String lastName, int custId)
   throws RemoteException;
   void deleteEntry(int custId) throws RemoteException;
}
```

15.6 programarea clientului

Pentru o aplicație client EJB, trebuie să cunoaștem:

1. cum să creăm sau cum să găsim bean-ul
2. ce metode putem utiliza (să îi cunoaștem interfața)
3. cum să îi eliberăm resursele

Clientul poate crea un EJB printr-un obiect care implementează interfața EJBHome. Acest obiect acționează ca o fabrică (factory) pentru EJB-uri, creându-le pentru aplicații client.

Clientul are access la EJB (care este executat pe server) prin intermediul unei interfețe **remote**, implementată de către un obiect construit de către serverul EJB în procesul de desfășurare al aplicației.

Iată pașii principali în scrierea codului client:

partea de autentificare

Autentificarea clientului se poate face printr-o modalitate care este specifică serverului. În cazul unei aplicații web, autentificarea poate fi făcută (de exemplu) prin protocolul TLS/SSL.

obținerea unui context inițial

- dacă clientul este un alt EJB care are ca mediu de execuție același container și bean-ul utilizat este declarat ca o resursă în descriptorul de defășurare, obiectul de tip `InitialContext` este deja disponibil.

```
Context ctx = new InitialContext();
```

- dacă clientul este executat în afara containerului, obținerea unui obiect de tip `InitialContext` se poate face prin utilizarea unor proprietăți de pe server. Iată un exemplu:

```
try
{
   Properties prop = new Properties();
   prop.put(Context.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory";
   prop.put(Context.PROVIDER_URL,
         "localhost:1099");
   Context ctx = new InitialContext(prop);
}
```

găsirea interfeței home a bean-ului

- pentru un client executat în interiorul containerului, codul poate arăta astfel:

```
Object homeRef = ctx.lookup("java:comp/env/ejb/MyBean");
```

- dacă clentul este executat în afara containerului, bean-ul poate fi asociat unui nume din spațiul de

numire JNDI. Este sarcina JNDI să identifice resursa asociată numelui pentru care se face căutarea:

```
Object homeRef = ctx.lookup("MyBean");
```

cast-ul referinței la interfața home

Pentru a ne asigura că clientul lucrează corect cu protocolul de comunicație dintre client și server, clientul trebuie să utilizeze metoda `narrow()` a obiectului `javax.rmi.PortableRemoteObject`:

```
MyBeanHome myHome = (MyBeanHome)PortableRemoteObject.narrow(homeRef,
   MyBeanHome.class);
```

De ce trebuie utilizată metoda `narrow()`? De obicei, atunci când facem o căutare utilizând metoda `lookup()` a unui obiect de tip `Context`, metoda va returna un `Object` vag, care are nevoie de un cast la interfața home pe care o căutăm. Problema este că acest lucru nu se poate face utilizând un cast normal, precum:

```
MyBeanHome myHome = (MyBeanHome)returnedObject;
```

Motivația acestui lucru ține de CORBA (Common Object Request Broker Architecture). De ce? Pentru EJB-uri, comunicarea dintre client și server este bazată pe RMI (Remote Method Invocation)(de fapt, atît interfața locală cât și cea Remote implementează interfața `java.rmi.Remote`).

Pe de altă parte, protocolul de comunicare utilizat de CORBA este IIOP (Internet Inter ORB Protocol), care este parte a standardului CORBA.

IIOP nu a fost creat pentru Java, ci pentru o întreagă familie de limbaje generice, și aceasta înseamnă că are câteva limitări. Unele limbaje, de exemplu, nu au conceptul de casting. Java RMI-IIOP oferă un mecanism pentru a îngusta (narrow) obiectul primit în urma căutării, la tipul de obiect căutat. Această îngustare se face prin intermediul clasei `javax.rmi.PortableRemoteObject`, mai exact, prin metoda sa `narrow()`.

crearea unei instanțe a EJB-ului

Această instanță este creată pe server. Clientul are acces doar la interfața **remote** a acestei instanțe. Altfel spus, clientul are acces doar la un stub.

Iată codul:

```
MyBeanObject myObject = myHome.create();
```

apelarea metodelor de logica business-ului a bean-ului

```
myObject.processEntry("Dumitrascu", "Vasile", 1102);
```

ștergerea instanței bean-ului

```
myObject.remove();
```

15.7 programarea bean-ului

Deoarece interfața home și interfața remote au fost deja detaliate, ne concentrăm acum pe clasa bean-ului propriu-zis. Pe lângă implementarea metodelor de logică a business-ului (care au fost declarate în interfața remote), clasa bean-ului trebuie să implementeze (deși implementarea propriu-zisă poate fi goală) un anumit set de metode, set care este specific pentru fiecare tip major de EJB-uri (sesiune, entitate, conduse de mesaje).

Presupunând că bean-ul nostru (numit MyBean) este unul de tip sesiune, codul de implementare a clasei poate arăta astfel:

```
package com.bank11.ccards.ejbeans;
import javax.ejb.SessionContext;
public class MyBean implements javax.ejb.SessionBean
{
   public void processEntry(String firstName, String lastName, int
   custId)
   {
         // method implementation
         ...
   }
   public void deleteEntry(int custId)
   {
         // method implementation
         ...
   }
   // mandatory methods for session beans
   // method implementations may be empty
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext ctx) {}
}
```

15.8 descriptorul de desfășurare

Descriptorul de desfășurare al uni EJB conține informații despre bean în relație cu aplicația de care aparține.

Aceste informații pot fi împărțite în două mari categorii:

- infromații structurale despre un anumit EJB
- informații de asamblare a aplicației

Deși lista de mai jos nu este exhaustivă, iată o listă tipică de intrări dintr-un descriptor de desfășurare:

1. elemente de control al accesului – elemente de securitate, ce utilizatori pot accesa un bean sau anumite metode ale bean-ului
2. numele bean-ului – numele sub care este înregistrată sub JNDI
3. descriptori de control – specifică atribute de control pentru tranzacții
4. numele clasei EJB
5. proprietăți de mediu (environment)
6. numele intefeței home
7. numele intefeței remote
8. elemente specifice pentru bean-uri de tip sesiune
9. elemente specifice pentru bean-uri de tip entitate
10. atribute – precum tranzacții, nivel de izolare, securitate

Ținând în minte că mai trebuie adăugat și asamblorul de aplicație, iată un exemplu tipic de descriptor de desfășurare:

```
<?xnm version="1.1"?>
<ejb-jar>
  <entrprise-beans>

  <session>
    <ejb-name>CCEnroll</ejb-name>
    <home>com.bank11.ccards.ejb.CCEnrollHome</home>
    <remote>com.bank11.ccards.CCEnrollObject</remote>
    <ejb-class>com.bank11.ccards.CCEnroll</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container<transaction-type>
    <ejb-ref>
      <ejb-ref-name>ejb/CCAccount</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>com.bank11.ccards.ejb.AccountHome</home>
      <remote>com.bank11.ccards.ejb.AccountObj</remote>
    </ejb-ref>
    <security-role-ref>
      <description>
```

```
          This role relates to cash advances from ATMs
        </description>
        <role-name>CashAdvATM</role-name>
      <security-role-ref>
    </session>

    <entity>
      <ejb-name>Account</ejb-name>
      <home>com.bank11.ccards.ejb.AccountHome</home>
      <remote>com.bank11.ccards.Accountbject</remote>
      <ejb-class>com.bank11.ccards.Account</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
        <field-name>accountNumber</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>userName</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>customerID</field-name>
      </cmp-field>
      <cmp-field>
        <prim-key-field>accountNumber</prim-key-field>
      </cmp-field>
      <env-entry>
        <env-entry-name>env/minPaymentPerc</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>2.5</env-entry-value>
      </env-entry>
    </entity>

    </enterprise-beans>
</ejb-jar>
```

Descriptorul de asamblare combină EJB-urile într-o aplicație desfășurabilă. Iată un astfel de asamblor:

```
</ejb-jar>
```

```
  <enterprise-beans>

  ...
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
     <ejb-name>CCEnroll</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

chapter 16 Bean-uri de sesiune (session beans)

Există două tipuri de bean-uri de sesiune, și anume: cu stare (**stateful**) și fără stare (**stateless**).

Un bean de sesiune cu stare păstrează datele între perioadele de acces de către client. Un bean fără stare nu face acest lucru.

Atunci când un server (container) de EJB-uri trebuie să-și economisească resursele, acesta poate scoate din memorie bean-urile de sesiune cu stare . Acest lucru reduce numărul de instanțe administrate de către server. Pentru pasivarea (**passivate**) bean-ului și conservarea stării sale de la un moment dat, starea bean-ului este serializată și stocată pe un mediu secundar. Atunci când clientul invocă o metodă a EJB-ului, obiectul este activat (**activated**), adică o nouă instanță este creată și conținutul ei este completat cu informațiile stocate anterior.

16.1 metode de callback pentru bean-urile de sesiune

Există 5 metode obligatorii de callback pentru clasele ce implementează interfața SessionBean.

```
public void ejbActivate()
public void ejbPassivate()
public void ejbCreate()
public void ejbRemove()
public void setSessionContext(SessionContext ctx)
```

Primele două metode nu vor fi apelate niciodată pentru bean-uri de sesiune fără stare, deoarece containerul nu va pasiva/activa niciodată un bean de sesiune fără stare.

16.2 ciclul de viață al unui bean de sesiune cu stare

Figura 16.1 ilustrează stadiile prin care trece un bean de sesiune cu stare pe durata vieții sale. Clientul inițiază ciclul de viață invocând metoda `create()`. Containerul EJB instanțiază bean-ul și apoi invocă metodele `setSessionContext()` și `ejbCreate()` ale bean-ului de sesiune. Bean-ul este acum gata pentru a executa metodele de logica aplicației. (business methods).

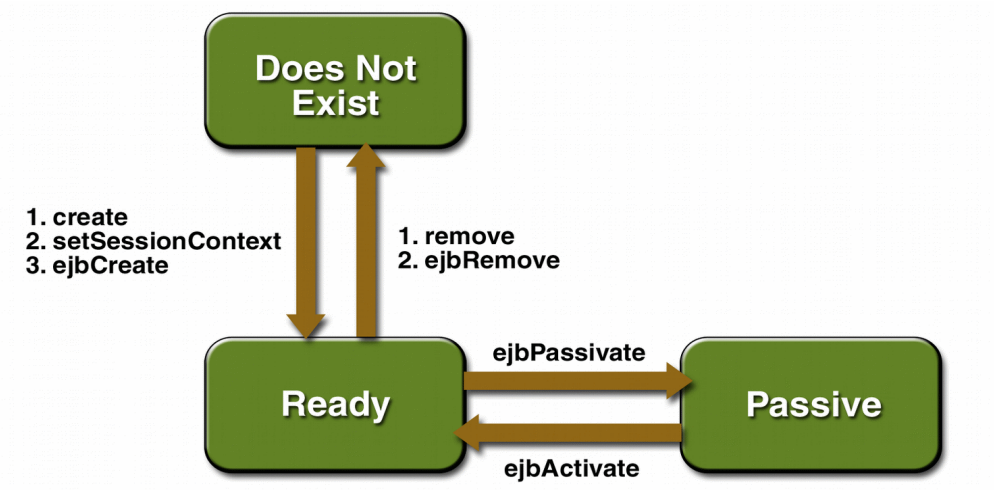


Figura 16.1 Ciclul de viață al unui bean de sesiune cu stare

Atunci când bean-ul este în starea de ready, containerul EJB poate decide deactivarea (sau pasivarea) bean-ului, mutându-l din memorie în stocare secundară. (Într-un scenariu tipic, containerul EJB pasivează bean-ul care are cea mai lungă perioadă de inactivitate). Containerul EJB invocă metoda `ejbPassivate()` a bean-ului imediat înainte de pasivarea sa. Dacă un client invocă o metodă de business a bean-ului atunci când acesta este pasivat, containerul EJB activează bean-ul, apelează metoda `ejbActivate()` a bean-ului și acesta trece în stadiul de ready.

La sfârșitul ciclului de viață, clientul invocă metoda `remove()` iar containerul EJB apelează metoda `ejbRemove()` a bean-ului. Instanța bean-ului e acum gata pentru curățenie (garbage collection).

Codul clientului controlează doar invocarea a două metode din ciclul de viață: metodele `create()` și `remove()`. Toate celelalte metode din Figure 16.1 sunt invocate de către containerul EJB. Metoda `ejbCreate()`, de exemplu, aparține clasei bean-ului, permițând efectuarea unor operații imediat după instanțierea bean-ului. De exemplu, se poate crea o conexiune cu o bază de date în metoda `ejbCreate()`.

16.3 ciclul de viață al unui bean de sesiune fără stare

Deoarece un bean de sesiune fără stare nu este niciodată pasivat, ciclul său de viață constă din doar două stadii: ne-existent și gata (ready) pentru invocarea metodelor de business. Figura 16.2 ilustrează aceste stadii pentru un bean de sesiune fără stare.

Figure 16.2 Ciclul de viață al unui bean de sesiune fără stare

chapter 17 Bean-uri de entitate (entity beans)

Bean-urile de entitate reprezintă date concrete (stocate, de obicei, într-o bază de date).

Containerul EJB oferă programatorului câteva servicii de persistență:

1. funcții container pentru administrarea cache-ului într-o tranzacție
2. suport pentru acces concurent (simultan)
3. menținerea unui cache între tranzacții
4. oferă tot codul de administrare a comunicării cu bazele de date (nu este nevoie de cod SQL)

Există două tipuri principale de bean-uri de entitate:

- **CMP**s (Container Managed Persistence) – persistență administrată de către container
- **BMP**s (Bean Managed Persistence) – persistență administrată de către bean – programatorul bean-ului trebuie să creeze codul Sql de comunicare cu bazele de date

17.1 chei primare

Fiecare bean de entitate are o cheie primară. Cheia primară trebuie reprezentată de o clasă de cheie primară. Cerințele care trebuie satisfăcute de către cheia primară sunt diferite pentru cele două tipuri principale de bean-uri de entitate:

Pentru BMP-uri:

- cheia primară poate fi orice tip RMI / IIOP legal
- trebuie să ofere implementări conforme pentru metodele `hashCode(), equals()`
- trebuie să aibă o valoare unică printre bean-urile de un anumit tip

Pentru CMP-uri:

- containerul trebuie să fie capabil de a crea o cheie primară
- clasa cheii primare trebuie să aibă un constructor fără argumente

Numele complet (incluzând pachetul) al clasei de cheie primară este întotdeauna specificat în descriptorul de desfășurare (eceptând cazul în care nu este cunoscută pînă la desfășurare).

Un exemplu:

```
<prim-key-class>com.bank11.ccards.CustomerID</prim-key-class>
```

sau

```
<prim-key-class>java.lang.String</prim-key-class>
```

În cazul unui CMP care utilizează un câmp de tip primitiv ca și cheie primară, acel câmp trebuie specificat:

```
<prim-key-field>sportsTeamID</prim-key-field>
```

17.2 metode oblligatorii de callback pentru bean-uri de entitate

Pe lîngă metodele CRUD de callback care vor fi discutate ceva mai târziu în această secțiune, un bean de entitate trebuie să implementeze (deși implementarea poate fi și goală) următoarele metode:

```
public void ejbActivate()
public void ejbPassivate()
public void setEntityContext(EntityContext ctx)
public void unsetEntityContext()
```

CRUD înseamnă: Create, Read, Update și Delete. Metodele sunt obligatorii pentru bean-urile de entitate.

17.3 create

Atunci când un client apelează o metodă `create()` pe interfața home a unui bean de sesiune, este creată o instanță a acelui bean (interfața remote, de fapt). Pe de altă parte, atunci când un client apelează metoda `create()` pe interfața home a unui bean de entitate, date de stare sunt stocate (de obicei, într-o bază de date) (inserăm de fapt un articol în baza de date). Acestea sunt date tranzacționale care sunt accesibile mai multor clienți. Putem avea mai multe metode `create()`, toate aruncând excepțiile `RemoteException`, `CreateException`.

Fiecare metodă `create()` a interfeței home a bean-ului are două metode corespondente în clasa de implementare a bean-ului, șia nume: `ejbCreate()` și `ejbPostCreate()`, metode care au aceiași parametri, în aceeași ordine, ca și parametrii din metoda `create()` originală.

- Tipul de retur al metodei `ejbCreate()` este același cu al cheii primare, dar pentru CMP valoarea de retur este null.
- pentru BMP, `ejbCreate()` trebuie să conțnă cod SQL de inserare și returnează o instanță a cheii primare, nenulă.

17.4 read

- `ejbLoad()`, lăsată goală în majoritatea cazurilor pentru CMP, dar necesită cod SQL în BMP
- implementarea persistenței baen-ului poate amîna înc ărcarea pînă când este utilizat
- `ejbLoad()` poate conține cod de procesare

17.5 update

- `ejbStore()` în CMP; metoda poate fi utilizată pentru preprocesarea datelor ce urmează a fi stocate, dar în general este goală
- în BMP, cod SQL de update, datele modificate trebuie stocate imediat

17.6 delete

- metoda corespunzătoare din clasa de implementare a bean-ului este `ejbRemove()`
- datele sunt șterse din baza de date (în cazul CMP), pentru BMP, programatorul va scrie codul SQL corespunzător

17.7 ciclul de viață al unui bean de entitate

Figura 17.1 arată stadiile prin care trece un bean de entitate pe durata vieții sale. După ce containerul EJB crează o instanță a bean-ului, apelează metoda `setEntityContext()` a bean-ului.

După instanțiere, bean-ul de entitate este mutat într-un bazin (pool) de instnațe disponibile. Atâta timp cât este în bazin, instanța nu este asociată cu o anume entitate EJB. Containerul EJB îi atribuie o identitate atunci când este trecut în stadiul de ready (gata).

Există două trasee din stadiul de bazin în stadiul de ready. În prima variantă, clientul invocă metoda `create()`, care are ca efect invocarea metodelor `ejbCreate()` și `ejbPostCreate()` de către containerul EJB. În a doua variantă, containerul EJB invocă metoda `ejbActivate()`. Atunci când un bean de entitate este în stadiul de ready, metodele sale de business pot fi apelate.

Există de asemenea două trasee din stadiul de ready în cel de bazin. În prima variantă, clientul invocă metoda `remove()`, care are ca efect ivocarea metodei `ejbRemove()` de către containerul EJB. În a doua variantă, containerul poate invoca metoda `ejbPassivate()`.

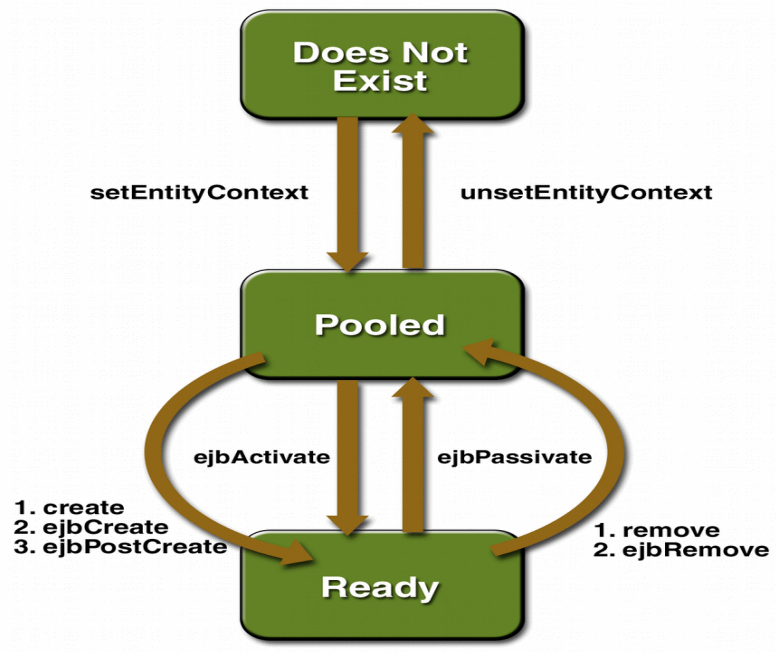


Figura 17.1 Ciclul de viață al unui bean de entitate

La sfârșitul ciclului, containerul EJB elimină instanța și invocă metoda `unsetEntityContext()`.

În starea de stocare în bazin, o instanță nu este asociată cu vreo identitate particulară de obiect EJB. În cazul BMP (bean managed persistence), atunci când containerul EJB mută instanța din stadiul de bazin în stadiul de ready, acesta nu setează automat cheia primară. Din acest motiv, metodele `ejbCreate()` și `ejbActivate()` trebuie să atribuie o valoare cheii primare. Dacă cheia primară este incorectă, metodele

`ejbLoad()` și `ejbStore()` nu pot sincroniza variabilele instanței cu datele din baza de date. Metoda `ejbActivate()` setează cheia primară (`id`) astfel:

```
id = (String)context.getPrimaryKey();
```

Când sunt stocate în bazin, valorile variabilelor instanței nu sunt necesare. Aceste variabile pot fi făcute disponibile pentru curățenie prin setarea lor cu valoarea null în metoda `ejbPassivate()`.

chapter 18 entități de persistență (persistence ent.)

18.1 cum au apărut?

Înainte de introducerea specificației EJB 3.0, mulți dezvoltatori de Java foloseau obiecte persistente ușoare, furnizate fie de cadre de persistență (de exemplu Hibernate), fie de obiecte de acces la date în loc de bean-uri de entitate. Acest lucru se datorează faptului că bean-urile de entitate, în specificațiile anterioare ale EJB, au avut o amprentă prea mare a resurselor, un cod complex și acestea ar putea fi utilizate numai în serverele de aplicații Java EE din cauza interconexiunilor și a dependențelor din codul sursă între bean-uri și obiectele DAO sau cadrul de persistență.

Astfel, multe dintre caracteristicile prezentate inițial în cadrele de persistență ale unor terțe părți au fost încorporate în API-ul Java Persistence și, începând cu 2006, proiecte precum Hibernate (versiunea 3.2) și TopLink Essentials au devenit ele însele implementări ale specificației API Java Persistence.

18.2 domenii pentru persistența java

Există patru domenii principale care sunt acoperite sub termenul generic de persistență Java:

1. **Java persistence Entities**
2. **Java Persistence API**.
3. **Java Persistence Query Language**
4. **Java Persistence Criteria API**

18.3 entități de persistență java

O entitate este o clasă Java ușoară. De obicei, o entitate reprezintă un tabel într-o bază de date relațională, iar fiecare instanță entitate corespunde unui rând din acel tabel. Artefactul de programare primar al unei entități este clasa entității, deși entitățile pot folosi clase de ajutor.

Entitățile au de obicei relații cu alte entități, iar aceste relații sunt exprimate prin metadate obiect / relaționale.

Starea persistentă a unei entități este reprezentată fie prin câmpuri persistente, fie prin proprietăți persistente. Aceste câmpuri sau proprietăți utilizează adnotări de mapare obiect / relație pentru a cartografia relațiile entităților și entităților cu datele relaționale din memoria de bază a datelor.

18.4 API pentru persistența java (Java Persistence API)

Java Persistence API este o specificație de interfață de programare a limbajului de programare Java care descrie gestionarea datelor relaționale în aplicații folosind Platforma Java, Standard Edition și Platforma Java, Enterprise Edition.

API-ul Java Persistence a apărut ca parte a activității Grupului JSR 220 Expert al Procesului comunitar Java. Versiunea finală a specificației JPA 1.0 datează din mai 2006.

JPA 2.0 a fost lucrarea Grupului de experți JSR 317 și a fost lansat în decembrie 2009. Specificația actuală este JPA 2.1 și a fost lansată în aprilie 2013 ca JSR 338.

Lucrul la versiunea JPA 2.2 a început în 2017 ca versiune de mentenanță a JSR 338. în iunie 2017 a apărut un prim draft al acestei versiuni.

Principalele caracteristici incluse în versiunea 2.1 au fost:

- Convertoare - permite conversii de cod personalizate între tipurile de bază de date și obiecte.
- Actualizarea / ștergerea criteriilor - permite actualizări în masă și șterge prin criteriul API.
- Proceduri stocate - permite definirea interogărilor pentru procedurile stocate în baza de date.
- Generarea schemei
- Grafuri pentru entități - permite preluarea parțială sau specificată a fuzionării obiectelor.
- Îmbunătățiri JPQL / Criterii - sub-interogări aritmetice, funcții generice de baze de date, clauză ON, opțiune TREAT.

Furnizori care oferă suport pentru JPA 2.1

- DataNucleus
- EclipseLink
- Hibernate

API-ul Java Persistence a fost dezvoltat în parte pentru a unifica API-ul Java Data Objects și API-ul EJB 2.0 Container Managed Persistence (CMP). Începând cu 2009, majoritatea produselor care susțin fiecare dintre aceste API-uri suportă API Java Persistence.

API-ul Java Persistence specifică persistența numai pentru sistemele de gestionare a bazelor de date relaționale. Cu alte cuvinte, JPA se concentrează pe maparea obiectual-relațională (ORM) (rețineți că există furnizori APP care susțin alte modele de baze de date în afara bazelor de date relaționale, dar acest lucru este în afara scopului pentru care a fost proiectat APP). Consultați secțiunea 1 a introducerii JPA 2 pentru clarificarea rolului JPA, care afirmă foarte clar că "Obiectivul tehnic al acestei lucrări este de a oferi o facilitate de mapare obiect / relațional pentru dezvoltatorul de aplicații Java folosind un model de domeniu Java pentru a gestiona o relațională Bază de date."

Specificația Java Data Objects oferă suport pentru ORM, precum și persistența față de alte tipuri de modele de baze de date, de exemplu baze de date cu fișiere plate și baze de date NoSQL, inclusiv baze de date de documente, baze de date grafice, precum și orice alt tip de date care se poate concepe.

18.5 entități de persistență în detaliu

18.5.1 cerințe pentru clasele de entitate

O clasă de entitate trebuie să respecte aceste cerințe.

- Clasa trebuie adnotată cu adnotarea javax.persistence.Entity.
- Clasa trebuie să aibă un constructor public sau protejat, fără argument. Clasa poate avea și alți constructori.
- Clasa nu trebuie declarată finală. Nu trebuie declarate definitive metode sau variabile de instanță persistente.
- Dacă o instanță a entității este trecută prin valoare ca obiect detașat, cum ar fi printr-o interfață de afaceri la distanță a fasolei sesiunii, clasa trebuie să implementeze interfața Serializabilă.
- Entitățile pot extinde atât clasele entității, cât și clasele non-entitate, iar clasele non-entitate pot extinde clasele de entități.
- Variabilele de instanță persistente trebuie declarate private, protejate sau pachete private și pot fi accesate direct numai prin metodele clasei entității. Clienții trebuie să acceseze statutul entității prin metode accesoriale sau de afaceri.

18.5.2 câmpuri persistente și proprietăți în clase de entitate

Starea persistentă a unei entități poate fi accesată prin variabilele sau proprietățile instanței entității. Câmpurile sau proprietățile trebuie să fie din următoarele tipuri ale limbajului Java:

- Tipuri primitive Java
- java.lang.String
- Alte tipuri serializabile, inclusiv:
  - ambalare a tipurlor Java primitive
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - tipuri serializabile definite de utilizator
  - byte[]
  - Byte[]
  - char[]
  - Character[]
  - tipuri enumerate
  - alte entități și/sau colecții de entități
  - clase scufundate

Entitățile pot folosi câmpuri persistente, proprietăți persistente sau o combinație a celor două. Dacă adnotările de mapare sunt aplicate variabilelor de instanță ale entității, entitatea folosește câmpuri persistente. Dacă adnotările de mapare sunt aplicate metodelor getter ale entității pentru proprietățile stilului JavaBeans, entitatea utilizează proprietăți persistente.

18.5.3 chei primare în entități

Fiecare entitate are un identificator de obiect unic. O entitate client, de exemplu, ar putea fi identificată de un număr de client. Identificatorul unic sau cheia primară permite clienților să localizeze o anumită instanță a unei entități. Fiecare entitate trebuie să aibă o cheie primară. O entitate poate avea o cheie primară simplă sau compusă.

Cheile primare simple utilizează adnotarea javax.persistence.Id pentru a denumi proprietatea sau câmpul cheie primară.

Cheile primare compuse sunt utilizate atunci când o cheie primară constă din mai multe atribute, care corespund unui set de proprietăți sau câmpuri persistente unice. Tastele primare compuse trebuie să fie definite într-o clasă de chei primare. Tastele primare compuse sunt notate folosind adnotările javax.persistence.EmbeddedId și javax.persistence.IdClass.

Cheia primară sau proprietatea sau câmpul unei chei primare compuse trebuie să fie unul dintre următoarele tipuri de limbi Java:

- Tipuri primitive Java
- Tipuri de wrapuri primitive Java
- java.lang.String
- java.util.Date (tipul temporal ar trebui să fie DATE)
- java.sql.Date
- java.math.BigDecimal

- java.math.BigInteger

Tipurile în virgulă flotantă nu trebuie folosite niciodată în cheile primare. Dacă utilizați o cheie primară generată, numai tipurile integrale vor fi portabile.

O clasă de chei primare trebuie să îndeplinească aceste cerințe.

- Modificatorul de control al accesului din clasă trebuie să fie public.
- Proprietățile clasei cheie primară trebuie să fie publice sau protejate dacă se utilizează accesul bazat pe proprietăți.
- Clasa trebuie să aibă un constructor public implicit.
- Clasa trebuie să implementeze metodele hashCode () și equals (Object others).
- Clasa trebuie să fie serializabilă.
- O cheie primară compusă trebuie reprezentată și cartografiată în mai multe câmpuri sau proprietăți ale clasei entității sau trebuie reprezentată și cartografiată ca o clasă care poate fi încorporată.
- Dacă clasa este mapată în mai multe câmpuri sau proprietăți ale clasei entității, numele și tipurile câmpurilor cheie sau a proprietăților din clasa cheii primare trebuie să se potrivească cu cele ale clasei entității.

Următoarea clasă de chei primare este o cheie compusă și câmpurile orderId și itemId împreună identifică în mod unic o entitate:

```
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return (
                    (orderId==null?other.orderId==null:orderId.equals
                    (other.orderId)
                    )
```

```
                    &&
                    (itemId == other.itemId)
                );
        }

        public int hashCode() {
            return (
                    (orderId==null?0:orderId.hashCode())
                    ^
                    ((int) itemId)
                );
        }

        public String toString() {
            return "" + orderId + "-" + itemId;
        }
    }
```

18.5.4 multiplicitate în relațiile dintre entități

Multiplicitățile sunt de tip: unul la unul, unul la mulți, mulți la unul și mulți la mulți:

- unul-la-unul: Fiecare instanță entitate este legată de o singură instanță a unei alte entități. De exemplu, pentru a modela un depozit fizic în care fiecare bin de stocare conține un singur widget, StorageBin și Widget ar avea o relație unu-la-unu. Relațiile one-to-one utilizează adnotarea javax.persistence.OneToOne pe proprietatea sau câmpul persistent corespunzător.
- unul-la-mulți: O instanță entitate poate fi asociată cu mai multe instanțe ale celorlalte entități. Un ordin de vânzări, de exemplu, poate avea mai multe elemente rând. În aplicația de comandă, Ordinul ar avea o relație una cu mai multe cu LineItem. Relațiile one-to-many utilizează adnotarea javax.persistence.OneToMany pe proprietatea sau câmpul persistent corespunzător.
- mulți-la-unul: instanțe multiple ale unei entități pot fi asociate cu o singură instanță a celeilalte entități. Această multiplicitate este opusul unei relații unu-la-mulți. În exemplul menționat, relația cu Ordinul din perspectiva lui LineItem este mult-la-unu. Relațiile multi-la-unu folosesc javax.persistence.ManyToOneannotation pe proprietatea sau câmpul persistent corespunzător.
- mulți-la-mulți: instanțele entității pot fi asociate cu mai multe instanțe una de cealaltă. De exemplu, fiecare curs de colegiu are mulți studenți, iar fiecare student poate lua mai multe cursuri. Prin urmare, într-o aplicație de înscriere, Cursul și Studentul ar avea o relație numeroasă. Relațiile multi-multi utilizează adnotarea javax.persistence.ManyToMany pe proprietatea sau câmpul persistent corespunzător.

18.5.5 direction in entity relationships

Direcția unei relații poate fi bidirecțională sau unidirecțională. O relație bidirecțională are atât o parte proprie, cât și o parte inversă. O relație unidirecțională are doar o parte proprie. Partea proprie a unei relații determină modul în care rutina de persistență face actualizări relației în baza de date.

18.6 entități de gestionare (managing entities)

Entitățile sunt gestionate de către managerul entității, care este reprezentat de instanțe ale `javax.persistence.EntityManager`. Fiecare instanță `EntityManager` este asociată cu un context de persistență: un set de instanțe entitate gestionate care există într-un anumit stoc de date. Un context de persistență definește domeniul în care sunt create, persistente și eliminate anumite instanțe ale entității. Interfața `EntityManager` definește metodele utilizate pentru a interacționa cu contextul persistenței.

18.6.1 interfața EntityManager

API-ul `EntityManager` creează și elimină instanțe ale entității persistente, găsește entități după cheia primară a entității și permite ca interogările să fie difuzate pe entități.

18.6.2 manageri de entități gestionate de container

Cu un manager de entități gestionate de un container, contextul de persistență al unei instanțe `EntityManager` este propagat automat de către container către toate componentele aplicației care utilizează instanța `EntityManager` într-o singură tranzacție API (Java Transaction API).

Operațiunile JTA implică de obicei apeluri prin intermediul componentelor aplicației. Pentru a finaliza o tranzacție JTA, aceste componente necesită, de obicei, acces la un singur context de persistență. Acest lucru se întâmplă atunci când un `EntityManager` este injectat în componentele aplicației prin intermediul adnotării `javax.persistence.PersistenceContext`. Contextul de persistență este propagat automat cu tranzacția JTA curentă, iar referințele `EntityManager` care sunt mapate la aceeași unitate de persistență oferă acces la contextul de persistență din acea tranzacție. Prin propagarea automată a contextului persistenței, componentele aplicațiilor nu trebuie să transmită reciproc referințe la instanțe `EntityManager` pentru a efectua modificări într-o singură tranzacție. Containerul Java EE gestionează ciclul de viață al administratorilor entităților gestionate de containere.

Pentru a obține o instanță `EntityManager`, injectați managerul entității în componenta aplicației:

```
@PersistenceContext
EntityManager em;
```

18.6.3 manageri de entități gestionate de aplicație

Cu un manager de entități gestionate de aplicație, pe de altă parte, contextul de persistență nu este propagat în componentele aplicației, iar ciclul de viață al instanțelor `EntityManager` este gestionat de aplicație.

Managerii de entități gestionați de aplicații sunt utilizați atunci când aplicațiile trebuie să acceseze un context de persistență care nu este propagat cu tranzacția JTA în cadrul instanțelor `EntityManager` într-o anumită unitate de persistență. În acest caz, fiecare `EntityManager` creează un nou context de persistență izolat. `EntityManager` și contextul asociat de persistență sunt create și distruse explicit de aplicație. Ele sunt, de asemenea, folosite atunci când injectarea directă de instanțe `EntityManager` nu poate fi făcută, deoarece instanțele `EntityManager` nu sunt thread-safe. `EntityManagerFactory` instanțele sunt thread-safe.

Aplicațiile creează instanțe `EntityManager` în acest caz, utilizând metoda `createEntityManager` a `javax.persistence.EntityManagerFactory`.

Pentru a obține o instanță `EntityManager`, mai întâi trebuie să obțineți o instanță `EntityManagerFactory` prin injectarea acesteia în componenta aplicației prin intermediul adnotării `javax.persistence.PersistenceUnit`:

```
@PersistenceUnit
```

```
EntityManagerFactory emf;
```

Apoi, obțineți un `EntityManager` din instanța `EntityManagerFactory`:

```
EntityManager em = emf.createEntityManager();
```

Managerii de entități gestionați de aplicații nu propagă automat contextul tranzacțiilor JTA. Aceste aplicații trebuie să obțină manual accesul la managerul de tranzacții JTA și să adauge informații privind delimitarea tranzacțiilor atunci când efectuează operațiuni ale entității. Interfața javax.transaction.`UserTransaction` definește metode pentru a începe, a comite și a întoarce tranzacțiile. Injectați o instanță a `UserTransaction` creând o variabilă de instanță adnotată cu @`Resource`:

```
@Resource
UserTransaction utx;
```

Pentru a începe o tranzacție, apelați metoda `UserTransaction.begin`. Când toate operațiile entității sunt complete, apelați metoda `UserTransaction.commit` pentru a angaja tranzacția. Metoda `UserTransaction.rollback` este utilizată pentru a redirecționa tranzacția curentă.

Următorul exemplu arată modul de gestionare a tranzacțiilor într-o aplicație care utilizează un manager de entități gestionate de aplicații:

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
  utx.begin();
  em.persist(SomeEntity);
  em.merge(AnotherEntity);
  em.remove(ThirdEntity);
  utx.commit();
} catch (Exception e) {
  utx.rollback();
}
```

18.6.4 găsirea de entități care utilizează EntityManager

Metoda `EntityManager.find` este utilizată pentru a căuta entități din depozitul de date prin cheia primară a entității:

```
@PersistenceContext
EntityManager em;
```

```
public void enterOrder(int custID, Order newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

18.6.5 gestionarea ciclului de viață a unei instanțe de entitate

Administrezi instanțele entității invocând operații asupra entității prin intermediul unei instanțe EntityManager. Entitățile se află într-una din cele patru stări: noi, gestionate, detașate sau eliminate.

- Instanțele de entitate noi nu au identitate persistentă și nu sunt încă asociate cu un context de persistență.
- instanțele entității gestionate au o identitate persistentă și sunt asociate cu un context de persistență.
- Instanțele entității separate au o identitate persistentă și nu sunt asociate în prezent cu un context de persistență.
- Instanțele eliminate din entitate au o identitate persistentă, sunt asociate cu un context persistent și sunt programate pentru a fi eliminate din magazinul de date.

18.7 interogarea entităților

API-ul Java Persistence oferă următoarele metode pentru interogarea entităților.

- Limbajul de interogare Java Persistence (JPQL) este un limbaj simplu, bazat pe șir, similar cu SQL folosit pentru interogarea entităților și a relațiilor acestora.
- API-ul Criteria este utilizat pentru a crea interogări de tipul tipafe folosind API-urile de limbaj de programare Java pentru a interoga entitățile și relațiile acestora.

Atât JPQL, cât și Criteria API au avantaje și dezavantaje.

Doar câteva rânduri lungi, interogările JPQL sunt de obicei mai concise și mai ușor de citit decât criteriile de interogare. Dezvoltatorii familiarizați cu SQL vor găsi ușor să învețe sintaxa JPQL. JPQL numitele interogări pot fi definite în clasa entității utilizând o adnotare a limbajului de programare Java sau în descriptorul de implementare al aplicației. Întrebările JPQL nu sunt totuși, însă, și necesită o distribuție la preluarea rezultatului interogării de la managerul de entități. Acest lucru înseamnă că erorile de tip casting nu pot fi prinse la momentul compilării. Întrebările JPQL nu acceptă parametrii finali.

Criteriile interogări vă permit să definiți interogarea în nivelul de activitate al aplicației. Deși acest lucru este posibil și utilizând interogări dinamice JPQL, interogările criteriilor oferă performanțe mai bune deoarece interogările dinamice JPQL trebuie să fie analizate de fiecare dată când sunt chemați. Criteriile de interogare sunt tipice și, prin urmare, nu necesită turnare, așa cum fac JPQL. Criteriul API este doar un alt limbaj de programare Java API și nu necesită dezvoltatorii să învețe sintaxa unui alt limbaj de interogare. Interogările criteriilor sunt de obicei mai verbose decât interogările JPQL și solicită dezvoltatorului să creeze mai multe obiecte și să efectueze operații asupra acestor obiecte înainte de a trimite interogarea către managerul de entități.

18.8 un exemplu

Verificați link-ul – https://docs.oracle.com/javaee/6/tutorial/doc/gigst.html – pentru un inventar simplu și o cerere de comandă pentru menținerea unui catalog de piese și plasarea unei ordini detaliate a acelor părți. Aplicația are entități care reprezintă părți, furnizori, comenzi și elemente rând. Aceste entități sunt accesate utilizând o fascină de sesiune statală care ține logica de afaceri a aplicației. O fasole simplu de sesiune singleton creează entitățile inițiale în implementarea aplicațiilor. O aplicație Web Facelets manipulează datele și afișează date din catalog.

chapter 19 Bean-uri conduse de mesaje

19.1 ce sunt bean-urile conduse de mesaje (message driven beans)?

Un bean condus de mesaje (***message-driven bean***) este un EJB care permite aplicațiilor J2EE să proceseze asincron mesaje. Acționează ca un ascultător (listener) de mesaje JMS, similar cu un ascultător de evenimente, cu excepția faptului că primește mesaje în loc de evenimente. Mesajele pot fi trimise de către orice componentă J2EE – o aplicație client, alt EJB, o componentă web, de către o aplicație JMS sau de către un sistem care nu utilizează tehnologii J2EE.

La ora actuală, bean-urile conduse de mesaje procesează doar mesaje JMS, dar în viitor vor putea fi utilizate și pentru procesarea altor tipuri de mesaje.

Bean-urile de sesiune și de entitate permit trimiterea și primirea sincronă a mesajelor, dar nu și în mod asincron. Pentru a evita blocarea resurselor serverului pe durata așteptării sincrone a mesajelor, există acum posibilitatea utilizării bean-urilor conduse de mesaje, care permit procesarea asincronă a mesajelor.

19.2 diferențe dintre bean-uri conduse de mesaje și alte bean-uri

Cea mai vizibilă diferență dintre bean-urile conduse de mesaje și bean-urile de sesiune sau de entitate este aceea că clienții nu accesează bean-urile conduse de mesaje prin intermediul unei interfețe. Spre deosebire de bean-urile de sesiune sau de entitate, un bean condus de mesaje constă dintr-o singură clasă.

În multe privințe, un bean condus de mesaje seamănă cu un bean de sesiune fără stare.

- o instanță a unui bean condus de mesaje nu reține date de stare pentru un anumit client.
- toate instanțele unui bean condus de mesaje sunt echivalente, permițând containerului EJB să asocieze mesaje oricărei instanțe de bean condus de mesaje. Containerul poate stoca aceste instanțe într-un bazin, ceea ce permite procesarea concurentă a șirurilor de mesaje.
- un singur bean condus de mesaje poate procesa mesaje de la mai mulți clienți.

Variabilele la nivel de instanță ale unui bean condus de mesaje poate conserva anumite date de stare pe parcursul procesării mesajelor clienților – de exemplu, o conexiune cu un sistem de baze de date, o referință la un obiect sau la un EJB.

Atunci când sosește un mesaj, containerul apelează metoda `onMessage()` a bean-ului pentru procesarea mesajului. În nod normal, metoda `onMessage()` efectuează un cast al mesajului la unul din tipurile standard de mesaje JMS și îl procesează în conformitate cu logica aplicației. Metoda `onMessage()` poate apela alte metode de ajutor sau poate invoca un bean de sesiune sau un bean de entitate pentru procesarea informației din mesaj sau pentru a-l stoca în baza de date.

Un mesaj poate fi livrat unui bean condus de mesaje într-un context de tranzacție, astfel încât toate operațiile din metoda `onMessage()` să fie parte a unei singure tranzacții. Dacă procesarea mesajului este reversată (rolled back), mesajul va fi livrat din nou.

19.3 diferențe dintre bean-uri conduse de mesaje și bean-urile de sesiune fără stare

Deși dinamica creării și alocării instanțelor de bean-uri conduse de mesaje este similară cu comportamentul instanțelor de bean-uri de sesiune fără stare, bean-urile conduse de mesaje sunt diferite de bean-urile de sesiune fără stare (precum și față de alte tipuri de bean-uri) în câteva aspecte semnificative:

- bean-urile conduse de mesaje procesează mesaje JMS multiple în mod asincron, în loc de a le procesa într-o secvență serializată de apeluri de metode

- bean-urile conduse de mesaje nu au interfață home sau remote și din acest motiv nu pot fi accesate direct de către clienți interni sau externi. Clienții interacționează cu bean-urile conduse de mesaje doar indirect, prin trimiterea/primirea unui mesaj (de) la o coadă sau (de) la un topic.

19.4 suport de acces concurent pentru bean-uri conduse de mesaje

Bean-urile conduse mesaje oferă suport pentru procesare concurentă atît pentru cozi cât și pentru topicuri. Mai înainte, era disponibilă doar procesarea concurentă pentru cozi.

Pentru asigurarea procesării concurente, trebuie modificat fișierul de desfășurare `ejb-jar.xml`, prin setarea elementului `max-beans-in-free-pool` la o valoare >1. Dacă acest element este setat la o asemenea valoare, containerul va genera atîtea fir-uri de execuție (threads) câte au fost specificate. Pentru mai multe detalii, a se consulta link-ul max-beans-in-free-pool.

19.5 invocarea unui bean condus de mesaje

Atunci când un topic sau o coadă JMS primește un mesaj, utilizarea bean-ului condus de mesaje asociat se face astfel:

1. Obține o nouă instanță a bean-ului, fie dintr-un bazin de instanțe, dacă acesta există, sau crează o nouă instanță, conform procedurii din acest link: Creating and Removing Bean Instances.
2. Dacă bean-ul nu poate fi localizat în bazin și o nouă instanță trebuie creată, se apelează metoda `setMessageDrivenContext()` a bean-ului pentru a asocia instanța cu un context de container. Acest bean va putea utiliza elementele acestui context așa cum este specificat la acest link: Using the Message-Driven Bean Context.
3. Apelează metoda `onMessage()` a bean-ului pentru a efectua logica aplicației. A se vedea: Implementing Business Logic with onMessage().

19.6 programarea bean-urilor conduse de mesaje

Pentru a crea bean-uri conduse de mesaje, trebuie urmate anumite convenții care sunt descrise în specificația JavaSoft EJB 2.0, precum și anumite practici care au ca rezultat un comportament adecvat al bean-urilor.

Specificația EJB 2.0 oferă indicații detaliate pentru definirea metodelor într-un bean condus de mesaje. Codul de mai jos arată componentele de bază ale unei clase de bean condus de mesaje. Clasele, metodele și declarațiile de metode obligatorii sunt specificate în bold (îngroșate):

```
public class MessageTraderBean implements javax.ejb.MessageDrivenBean {
  public MessageTraderBean() {...};
    // An EJB constructor is required, and it must not
    // accept parameters. The constructor must not be declared as
    // final or abstract.
  public void onMessage(javax.jms.Message MessageName) {...}
    // onMessage() is required, and must take a single parameter of
    // type javax.jms.Message. The throws clause (if used) must not
    // include an application exception. onMessage() must not be
    // declared as final or static.
  public void ejbRemove() {...}
```

```
        // ejbRemove() is required and must not accept parameters.
        // The throws clause (if used) must not include an application
        //exception. ejbRemove() must not be declared as final or static.
      finalize{};
      // The EJB class cannot define a finalize() method
    }
```

19.7 crearea și ștergerea instanțelor de bean-uri

Containerul EJB apelează metodele `ejbCreate()` și `ejbRemove()` ale bean-ului condus de mesaje la crearea și ștergerea unei instanțe a clasei bean-ului. Ca și în cazul altor tipuri de EJB-uri, metoda `ejbCreate()` a clasei bean-ului pregătește toate resursele care sunt necesare pentru operațiile bean-ului. Metoda `ejbRemove()` are scopul de a elibera aceste resurse, pentru ca acestea să fie eliberate înainte ca containerul EJB să șteargă instanța.

Bean-urile conduse de mesaje trebuie, de asemenea, să efectueze anumite rutine de curățenie în afara metodei `ejbRemove()`, deoarece bean-urile nu pot conta pe faptul că metoda `ejbRemove()` va fi apelată în toate circumstanțele, de exemplu, dacă EJB-ul aruncă o excepție de runtime.

19.8 utilizarea contextului unui bean condus de mesaje

Containerul EJB apelează metoda `setMessageDrivenContext()` pentru a asocia bean-ul condus de mesaje cu contextul containerului. Acesta nu este un context al unui client, contextul clientului nu este transmis împreună cu un mesaj JMS. Containerul EJB oferă bean-ului un context al containerului, ale cărui proprietăți pot fi accesate din interiorul instanței utilizînd următoarele metode ale interfeței `MessageDrivenContext`:

- `getCallerPrincipal()`
- `isCallerInRole()`
- `setRollbackOnly()` – EJB-ul poate utiliza această metodă doar dacă utilizează demarcare pentru tranzacții administrate de către container.
- `getRollbackOnly()` – EJB-ul poate utiliza această metodă doar dacă utilizează demarcare pentru tranzacții administrate de către container.
- `GetUserTransaction()` – EJB-ul poate utiliza această metodă doar dacă utilizează demarcare pentru tranzacții administrate de către bean.

  Notă: Deși `getEJBHome()` este moștenită și ca parte a interfeței `MessageDrivenContext`, bean-urile conduse de mesaje nu au o interfață home. Apelarea metodei `getEJBHome()` dintr-o instanță a unui bean condus de mesaje va duce la aruncarea unei excepții `IllegalStateException`.

19.9 implementarea logicii aplicației cu metoda onMessage()

Metoda `onMessage()` efectuează întreaga logică de aplicație pentru bean. Containerul EJB apelează metoda `onMessage()` atunci când coada sau topicul asociat primește un mesaj, având obiectul de tip mesaj JMS ca argument. Este responsabilitatea bean-ului condus de mesaje să parcurgă mesajul și să efectueze logica de aplicație în metoda `onMessage()`.

Trebuie să vă asigurați că logica aplicației ține seama de caracterul asincron al procesării mesajelor. De exemplu, nu se poate presupune că EJB-ul va primi mesajele în ordinea în care au fost trimise de către client. Faptul că instanțele de bean-uri sunt parte a unui bazin (pool) înseamnă că mesajele nu sunt primite

sau procesate în ordine secvențială, deși apelurile individuale ale metodei `onMessage()` ale unui bean precizat sunt serializate.

A se vedea link-ul javax.jms.MessageListener.onMessage() pentru mai multe informații.

19.10 servicii de tranzacționare cu bean-uri conduse de mesaje

La fel ca și alte tipuri de EJB-uri, bean-urile conduse de mesaje pot demarca limitele tranzacțiilor fie prin ele însele (utilizând tranzacții administrate de bean-uri) sau lăsând acest lucru containerului EJB (în cazul tranzacțiilor administrate de către container). În ambele cazuri, un bean condus de mesaje nu primește un context de tranzacție de la clientul care trimite mesajul. Containerul EJB apelează întotdeauna metoda `onMessage()` a bean-ului utilizând contextul de tranzacție specificat în descriptorul de desfășurare al bean-ului, conform specificației EJB 2.0.

Deoarece nici un client nu oferă un context de tranzacție pentru apeluri ale metodelor unui bean condus de mesaje, bean-urile care utilizează tranzacții administrate de container trebuie să fie desfășurate utilizând atributul `Required` sau `NotSupported` în `ejb-jar.xml`. Atributele de tranzacționare sunt definite în `ejb-jar.xml` astfel:

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>MyMessageDrivenBeanQueueTx</ejb-name>
      <method-name>*</method-name>
    </method>
  <trans-attribute>NotSupported</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

19.11 primirea mesajelor

Primirea unui mesaj JMS care duce la apelarea metodei `onMessage()` a bean-ului, nu este inclusă (în general) în scopul unei tranzacții. Pentru EJB-urile care utilizează tranzacții administrate de bean-uri, primirea mesajului este întotdeauna în afara scopului tranzacției, precum este precizat în apecificația EJB 2.0. Pentru EJB-uri care utilizează demarcare de tranzacții administrate de container, containerul EJB include primirea mesajului ca parte a tranzacției doar dacă atributul de tranzacționare al bean-ului este setat ca `Required`.

19.12 confirmarea mesajelor

Pentru bean-urile conduse de mesaje care utilizează tranzacții administrate de container, containerul EJB confirmă automat un mesaj atunci când tranzacția EJB este finalizată (committed). Dacă EJB-ul utilizează tranzacții administrate de către bean, atît primirea cât și confirmarea mesajului au loc în afara contextului tranzacției. Containerul EJB confirmă automat mesajele pentru bean-uri cu tranzacții administrate de bean-uri, iar desfășurătorul poate configura semantica confirmării utilizând parametrul de desfășurare `jms-acknowledge-mode`.

19.13 descriptorul de desfășurare

Pentru a desfășura un bean condus de mesaje într-un container EJB, trebuie editat un fișier XML –

descriptorul de desfășurare, care asociază bean-ul cu o destinație JMS configurată.

Descritporul de desfășurare pentru bean-urile conduse de mesaje specifică de asemenea:

- Dacă bean-ul este asociat unei cozi sau unui topic
- Dacă topicul asociat este durabil sau ne-durabil
- Atributele de tranzacționare pentru EJB
- Semantica JMS de confirmare care trebuie utilizată pentru bean-uri care își demarchează propriile tranzacții

Specificația EJB 2.0 adaugă următoarele noi elemente de desfășurare pentru bean-urile conduse de mesaje:

- `message-driven-destination` - specifică dacă EJB-ul eset asociat unui topic sau enei cozi JMS
- `subscription-durability` - specifică dacă topicul asociat este durabil sau ne-durabil
- `jms-acknowledge-mode` - specifică semantica JMS de confirmare care trebuie utilizată pentru bean-uri care își demarchează propriile tranzacții. Acest element poate avea două valori: `AUTO_ACKNOWLEDGE` (valoarea implicită) sau `DUPS_OK_ACKNOWLEDGE`.

Aceste elemente sunt definite în fișierul de desfășurare `ejb-jar.xml`, conform specificației EJB 2.0. Codul de mai jos arată un exemplu pentru definirea unui bean condus de mesaje:

```
<enterprise-beans>
  <message-driven>
    <ejb-name>exampleMessageDriven1</ejb-name>
    <ejb-class>examples.ejb20.message.MessageTraderBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <jms-destination-type>
        javax.jms.Topic
      </jms-destination-type>
    </message-driven-destination>
    ...
  </message-driven>
  ...
</enterprise-beans>
```

Pe lîngă adugarea de noi elemente de către fișierul `ejb-jar.xml`, fișierul `weblogic-ejb-jar.xml` include un nou cadru de tip message-driven-descriptor care asociază bean-ului condus de mesaje o destinație în containerul EJB.

19.14 ciclul de viață al unui bean condus de mesaje

Figura 19.1 iluztrează stadiile ciclului de viață al unui bean condus de mesaje.

De obicei, containerul EJB crează un bazin (pool) de instanțe de bean-uri conduse de mesaje. Pentru fiecare instanță, containerul EJB instanțiază bean-ul și efectuează aceste operații:

1. apelează metoda `setMessageDrivenContext()` pentru a seta contextul instanței

2. apelează metoda `ejbCreate()` a instanței

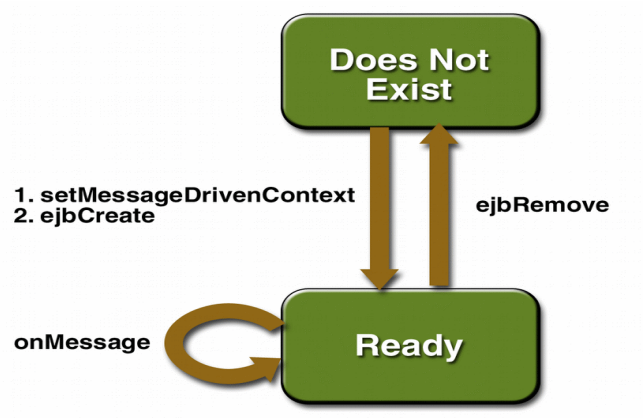


Figure 19.1 Ciclul de viață al unui bean condus de mesaje

La fel ca și un bean de sesiune fără stare, un bean condus de mesaje nu este niciodată pasivat și are doar două stări: inexistent sau gata (ready) de a primi mesaje.

La sfârșitul ciclului de viață, containerul apelează metoda `ejbRemove()`. Acum instanța bean-ului este gata pentru curățenie (garbage collection).