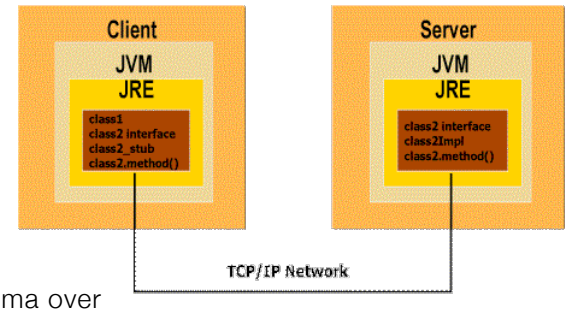
Hanzehogeschool Groningen
Instituut voor Informatie
en Communicatietechnologie

Thema 4.2I Gedistribueerde applicaties
Practicumopgaven week 2

inleiding



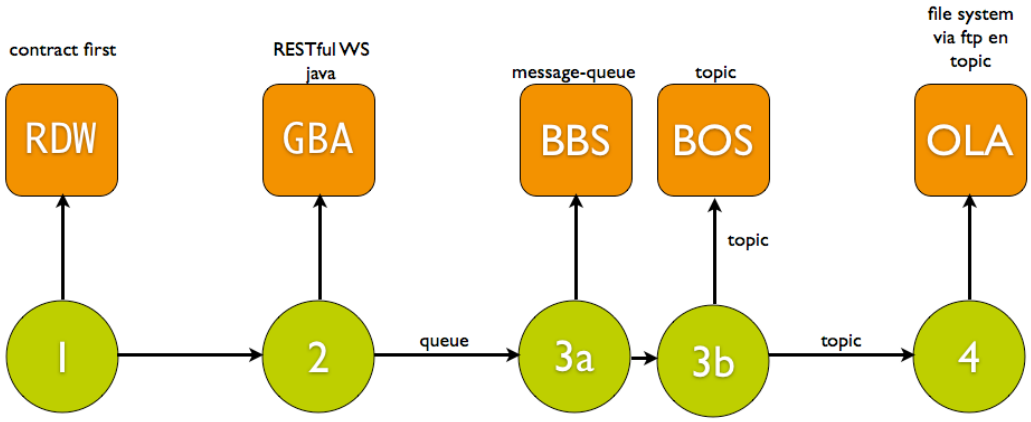
Een groot nadeel van het gebruik van RPC technieken als RMI of WCF is dat het in principe platformafhankelijk is. Dat betekent bijvoorbeeld dat een Java component dat via RMI beschikbaar wordt gesteld niet te benaderen is via een .NET client omdat het RMI protocol niet beschikbaar is in .NET.

Webservices vormen een veel meer gestandaardiseerde manier om verschillende applicaties te koppelen, onafhankelijk van de gebruikte platformen. Het zal je niet verbazen dat een groot deel van dit thema over webservices in allerlei varianten zal gaan.

De boete voor te hard rijden

Deze en komende week werken we aan de implementatie van één service: de verwerking van een boete voor te hard rijden door het CJIB. Wanneer een weggebruiker te hard rijdt, is het mogelijk dat zij door een camera van de Rijksdienst voor het Wegverkeer (RDW) geflitst wordt. Op basis van het kenteken worden vervolgens bij de Gemeentelijke Basisadministratie (GBA) de gegevens van de persoon op wiens of wier naam het voertuig is ingeschreven opgehaald. Daarna wordt via een derde systeem aan de hand van de overtreding en de historie de hoogte van de boete bepaald (BBS, Boeteberekeningsysteem) en de historie bijgewerkt (BOS, Boeteopslagsysteem). Ten slotte wordt een acceptgiro gegenereerd (OLA, optisch leesbare acceptgiro) die naar de overtreder wordt verzonden.

Het is de bedoeling van elk van deze systemen een implementatie te maken, waarbij in elk systeem een andere programmeertaal of –methode gebruikt wordt. De vier systemen en de werking van de service worden hieronder schematisch weergegeven.



In deze week beginnen we met een basale opdracht om een beeld te krijgen van webservices, hoe je daarmee werkt en hoe je ze kunt programmeren. Vervolgens maken we een eerste aanzet voor een banksysteem, waarbij we uitgaan van twee rekeningen die op dezelfde database draaien. Deze opdracht werken we verder uit met twee verschillende databases, zodat je rekening moet houden met gedistribueerde integriteit van de data. Hiervoor gebruiken we een clearing-house en de mogelijkheden die Glassfish biedt. Ten slotte starten we met de eerste service in de CJIB casus: het RDW systeem.

Opgave 1 – eenvoudige webservices

Zoals tijdens het inleidende college is gedemonstreerd, is het vrij eenvoudig om in een IDE als Eclipse, Netbeans of Visual Studio een webservice aan te maken, te deployen en client stubs te genereren. In deze opgave gaan we daar wat verder op in.

a. Download de klasse Student van Blackboard. Dit is een POJO die een student representeert. Zoals je ziet maakt deze klasse gebruikt van JAXB voor serialisatie. Implementeer de methode toXML.

b. Maak een tweede klasse in Eclipse die een methode `String getStudent(String naam)` heeft. Maak in deze methode een nieuwe student aan en geef die de naam die meegestuurd wordt. Geef wat (hard gecodeerde) waarden aan het geslacht en de leeftijd en return de geserialiseerde versie van de student.

c. Bestudeer http://javaeenotes.blogspot.nl/2010/10/web-service-with-jax-ws-in-eclipse.html. Hier wordt duidelijk gemaakt hoe je van een geannoteerde klasse een wsdl en een webservice kunt maken. Volg deze aanwijzingen om van de klasse van opgave 1b een webservice te maken.

d. Deploy de webservice op Glassfish en test hem in een browser (bijvoorbeeld via de Glassfish management dropdown)

e. Maak een C# en een php console client die van deze service gebruik maken. Demonstreer het resultaat aan je practicumdocent.

f. Leg uit waarom hier een klasse Student en een aparte klasse voor de webservice gebruikt wordt. Wat is het voordeel van deze manier van werken?

Opgave 2 – Javabank en lokale transacties

Het is uiteraard ook mogelijk om methoden aan te roepen die een zelfgemaakt type object verwachten en/of retourneren. Het is bedoeling van deze opgave om daar wat mee te experimenteren.

We willen graag de volgende methoden en typen gebruiken in een JavaBank webservice:

- ✓ `public void openAccount(Account account)`: het openen van een rekening.
- ✓ `public boolean transfer(float amount, String reknummerFrom, String reknummerTo)`: het overmaken van geld, rekening houdend met limiet. De boolean geeft aan of de transactie is gelukt.
- ✓ `public Account getAccount(String reknummer)`: het opvragen van rekeninggegevens.
- ✓ `public void alterAccount(Account account)`: het wijzigen van een account. Het moet onmogelijk zijn om het rekeningnummer van een account te wijzigen.
- ✓ `public Transaction[] getTransactions(String reknummer, String date)`: het opvragen van (geld)transacties, vanaf een bepaalde datum voor een specifiek rekeningnummer.
  In een Account worden NAW gegevens (naam, adres, woonplaats) en rekeninggegevens (rekeningnummer, saldo en limiet) opgeslagen. In een `Transaction` worden de verzender, ontvanger, bedrag en datum opgeslagen.

a. Maak de onderliggende database aan in MySQL.

b. Implementeer de webservice in Java. Hou hierbij rekening dat overmaken van geld alleen lukt als:

- ✓ verzender en ontvanger bestaan en
- ✓ de verzender voldoende saldo heeft.

- ✓ Het bijwerken van de saldi van de verzender en de ontvanger atomair moet zijn: beide lukken of beide mislukken. Gebruik hiervoor de transactiemogelijkheid in de `java.sql.Connection` interface.

c. Beschrijf hoe je kunt testen dat overmaken inderdaad atomair is.

d. Maak een php-pagina aan waarmee de webservice kan worden aangeroepen. Demonstreer de werking aan je practicumdocent en lever de programmacode in via Blackboard.

Opgave 3 – gedistribueerde transacties (theorie)

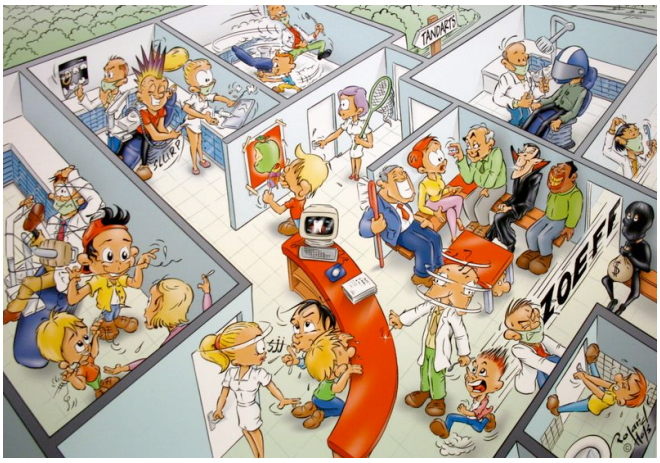
a. Leg de betekenis van ACID uit.

b. Leg uit waarom je niet met een gewone JDBC connectie of datasource zonder meer een transactie over twee of meer databases kunt regelen.

c. Om een gedistribueerde transactie voor elkaar te krijgen wordt vaak gebruik gemaakt van een two phase commit. Leg dit concept uit en gebruik in je uitleg de XA en TX interface.

Opgave 4 – gedistribueerde transactie (praktijk)

Download DisTransExample.zip van Blackboard en importeer dit in Eclipse of Netbeans.



a. Maak een nieuwe aparte database CJIBBank aan met JPA met een vergelijkbare tabelstructuur zoals die door de JavaBank uit opgave 2 wordt gebruikt.

b. Maak op basis van het DisTransExample een webservice ClearingHouse die geld kan overmaken van de Javabank naar de CJIBBank en vice versa, met behulp van een container managed transactie, sessionbean en JPA. Let op: in DisTransExample wordt een transactie niet uitgevoerd als er technisch iets mis gaat. Je wilt ook om een functionele reden dat een transactie kan mislukken, namelijk:

- ✓ wanneer de verzender of ontvanger niet bestaat of
- ✓ wanneer de verzender onvoldoende saldo heeft.

Hint: zoek eens naar een relevante methode in javax.ejb.SessionContext om dat mogelijk te maken.

Opgave 5 – contract first

Bij de webservices die je toe nu toe hebt gemaakt heb je vanuit code een wsdl-bestand gegenereerd. Dit is een voorbeeld van een *contract last* aanpak: eerst wordt de service geïmplementeerd, pas daarna is duidelijk wat de formele definitie is van de gebruikte typen.

In de praktijk wordt vaak voor een *contract first* aanpak gekozen: eerst overeenstemming bereiken over het formaat van de gegevens die worden gebruikt bij de afhandeling van een webservice aanroep voordat deze wordt geïmplementeerd. Zo kun je precies afspreken wat een client verstuurt en wat een server antwoordt. Dit gebeurt aan de hand van een XML Schema Definition (xsd).

In deze opgave gaan we aan de hand van een dergelijk xsd-bestand skelettons genereren die we vervolgens gaan implementeren. Je kunt zelf kiezen of je de implementatie in C# of in Java wilt doen: in het eerste geval kun je gebruik maken het script xsd, in het andere geval van xjc.

Download rdw.xsd van Blackboard. Hierin staat beschreven welke informatie is op te vragen bij de Rijksdienst voor het Wegverkeer op basis van een kenteken. Bestudeer het bestand.

Genereer de programmacode. Maak vervolgens een webservice *Thesaurus* met een methode `GetInfo` dat op basis van een kenteken een Info bericht terugstuurt. Dit Info bericht bestaat uit de volgende velden:

- ✓ statusInfoAanvraag: type String, “OK” of “NOT FOUND”. Als een kenteken bestaat en alle gegevens zijn gevonden wordt “OK” gehanteerd. Als een kenteken niet gevonden wordt, wordt “NOT FOUND” gehanteerd en zijn alle andere velden, behalve kenteken, niet met zinnige informatie gevuld;
- ✓ bsn: type long, de BSN van de kentekenbewijshouder;
- ✓ chassisNummer: type String, uniek identificatienummer van het voertuig;
- ✓ kenteken: type String, het opgevraagde kenteken, ook als deze niet bestaat;
- ✓ kleur: type String, de kleur van het voertuig;
- ✓ model: type Model, de generieke voertuiggegevens.

Het type Model bestaat uit:

- ✓ merkNaam: type String, merk van het voertuig, bijvoorbeeld “Citroën”;
- ✓ motorInhoud: type float, motorinhoud in cc van het voertuig, bijvoorbeeld 2940.0;
- ✓ typeAanduiding: type String, specifiek typeaanduiding van het merk, bijvoorbeeld “2CV”;
- ✓ typeMotor: type String, het brandstoftype, kan alleen de waarden “B”, “D” of “L” bevatten.

a. Implementeer de programmacode en toon het resultaat aan je practicumdocent. Je hoeft geen achterliggende database te maken. Het volstaat om in de implementatie hardcoded drie voertuigen op te nemen.