

Anwendungsintegration
mit REST-Services

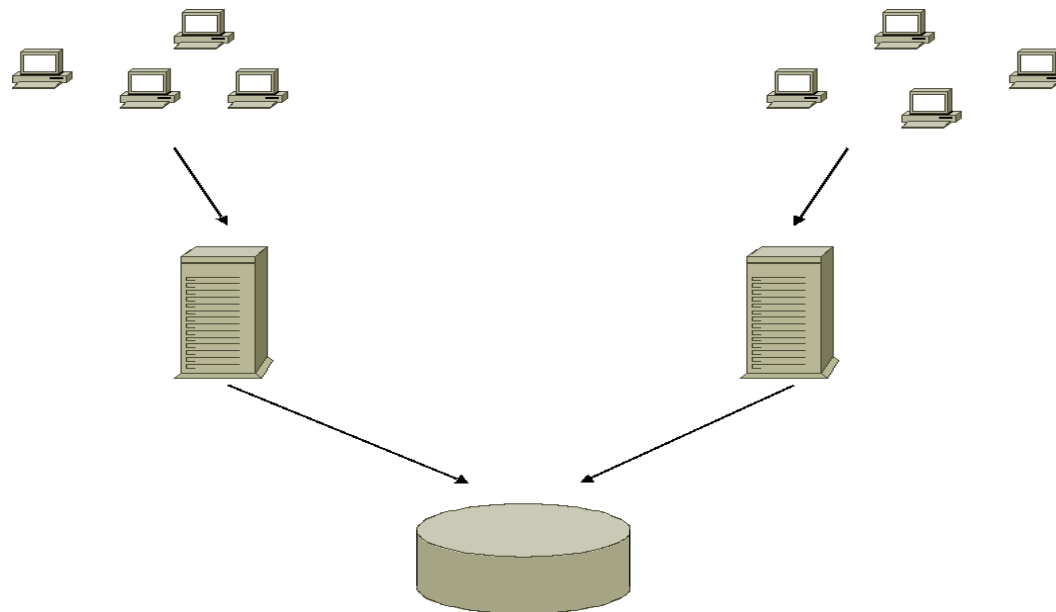
ex|Xcellent
solutions

JavaForum Stuttgart - 02.07.2009
Dr. Ralph Guderlei

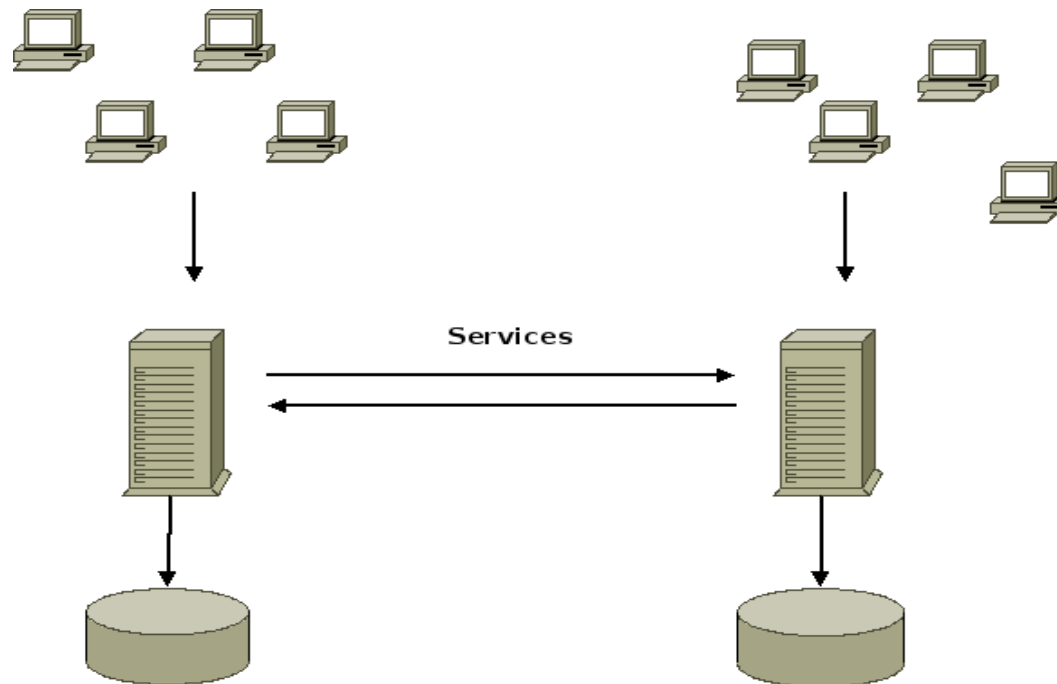


- x| Das Problem
- x| REST
- x| JAX-RS und Jersey
- x| Die Lösung
- x| Fazit

- x| 2 autonome Systeme
„die Inventar-Datenbank“ und
„die Workflow-Plattform“
- x| Teilweise gemeinsame Datenhaltung (z.B. Benutzerverwaltung)
(Shared Database Pattern)

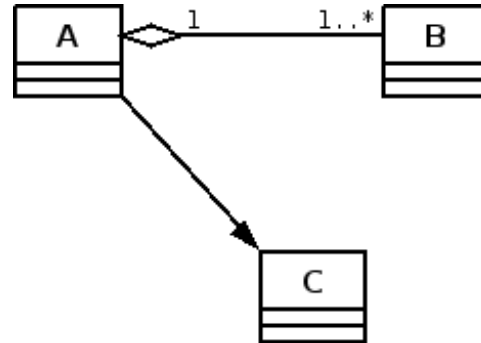


- x| Getrennte Datenhaltung
- x| Integration über Services (→ Remote Procedure Invocation Pattern)



- x| Representational State Transfer
- x| Menge an Bedingungen für eine Architektur eines verteilten Systems
 - | Client-Server
 - | Stateless
 - Aufrufe müssen alle Informationen beinhalten, die zum Verarbeiten des Aufrufs notwendig sind
 - | Cache
 - | Uniform Interface
 - URIs
 - Manipulation von Ressourcen durch Repräsentationen
 - sich selbst beschreibende (standardisierte) Methoden
 - HATEOAS
 - | Layered System
 - | Code on Demand

- x| XML over HTTP
- x| Alle Parameter eines Aufrufs werden in die URI codiert
- x| Simples CRUD auf Daten



- x| Relationen in Objektmodellen können abgebildet werden durch
- x| Den URI
`http://example.com/a/{identifizier}/b/`
- x| Links in Repräsentationen

- x| → gewisse Ähnlichkeiten erkennbar

- x| Wiederverwendung von HTTP-Features
 - | Authentifizierung (HTTP Auth)
 - | Caching
 - | Verschlüsselte Kommunikation (HTTPS)

- x| Beschreibung von REST-Service mit WADL

- x| Standardisiert in JSR-311 (final seit Oktober 2008)
- x| Implementierungen: Jersey, RESTEasy, Apache CXF, Restlet
- x| Programmiermodell: POJO + Annotations
- x| Deklaration von Services über standardisierte Annotations
- x| Automatisches Marshalling/Unmarshalling entsprechend dem definierten MIME-Type

@GET, @POST, @PUT, @DELETE, ...	Definieren die HTTP-Methode
@Produces, @Consumes	Definieren den MIME-Type des Requests bzw. der Response
@Path	Definiert die URI der Ressource
@PathParam, @HeaderParam	Erlauben das Auslesen von URI- Parametern bzw. Parametern aus dem HTTP-Header

- x| HTTP Status Codes ermöglichen einen standardisierten Austausch von Informationen über Erfolg/Misserfolg eines Aufrufs
- x| Frameworks verwenden Server- und Clientseitig „normale“ Java-Exceptions

```
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.PathParam;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.WebApplicationException;
8 import javax.ws.rs.core.MediaType;
9 import javax.ws.rs.core.Response;
10
11 @Path("/myservice")
12 public class MyServiceRessource {
13     @GET
14     @Path("/{identifier}")
15     @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
16     public MyClass getObject(@PathParam("identifier") String identifier){
17         MyClass myObject = MyDao.findById(identifier);
18         if(myObject == null){
19             throw new WebApplicationException(Response.Status.NOT_FOUND);
20         } else {
21             return myObject;
22         }
23     }
24
25 }
```

- x| Minimale Anforderungen an Infrastruktur (nur Servlet-Container)
- x| Standardisierte API, mehrere Implementierungen
- x| Einfache Technologie
- x| Wenig Abhängigkeiten
- x| Keine Probleme mit Firewalls, da HTTP/HTTPS
- x| Strukturierung von URIs analog zum Objektmodell möglich

- x| Entscheidung für Jersey als JAX-RS Implementierung
- x| Integration in bestehende Anwendungen unter Java 1.5 + WebSphere, Tomcat 6 und WebStart-Clients
- x| Java – only – Ansatz: POJOs für Datenaustausch, keine XSD für Repräsentationen o.ä.

x| Vorgehensweise:

- | Identifikation der benötigten Informationen
- | Entwurf der URIs für die benötigten Informationen:
Strukturierung analog zum bestehenden Objektmodell
- | Entwurf der Repräsentationen:
Auswahl der benötigten Attribute/Relationen, Erstellung von DTOs
Wahl der Darstellung (XML bzw. PDF)

- x| Problem: kein Zugriff auf Servlet-Container möglich (insbesondere keine Konfiguration der Zugriffskontrolle)

- x| Lösungsansatz:
 - | Analog zu HTTP Auth werden (eigene) Authentifizierungsinformationen in die Request-Header eingefügt

 - | Auf Serverseite: Wiederverwendung der bestehenden Authentifizierungs-/Authorisierungs-Infrastruktur

 - | Nachteil: API der Service-Methoden wird unnötig aufgebläht

x| Strukturierung

http://example.com/a/{identifizier-a}/b/{identifizier-b}

```
@Path("/a/{identifizier-a}")
public class RessourceForA {
    @Path("/b")
    public RessourceForB delegate(){
        return new RessourceForB();
    }

    @GET
    public A getA(@PathParam("identifizier-a") String id){
        return new A();
    }

    // ...
}

public class RessourceForB {
    @GET
    @Path("/{identifizier-b}")
    public B getB(@PathParam("identifizier-b") String id){
        return new B();
    }

    // ...
}
```

- x| ~ 100 ms./Request (ohne Caching)
- x| Kein signifikanter Unterschied zwischen XML und JSON als Repräsentationsformat

- x| Einfache Integration in bestehende (Web-) Anwendungen
- x| Einfache, reibungslose Verwendung von JAX-RS/Jersey
- x| Flexibel: z.B. einfache Verwendung eines eigenen Authentifizierungsmechanismus
- x| Ein bestehendes Objektmodell lässt sich einfach als Ressourcen abbilden

x| Richardson, L.; Ruby, S.(2007), RESTful Web Services, O'Reilly

x| Sun Jersey : <https://jersey.dev.java.net/>

Vielen Dank für Ihre Aufmerksamkeit!

Fragen??

Gerne auch
am Stand im Ausstellungsbereich
oder per Mail
r.guderlei@excellent.de