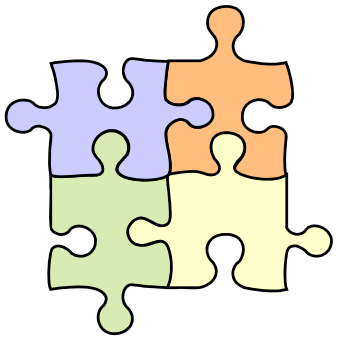
Luca Cabibbo



Architetture Software

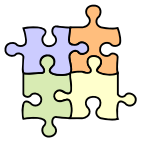
Programmazione di Web Services

Dispensa ASW 860

ottobre 2014

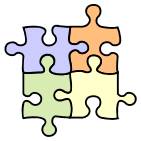
La scelta tra architetture
può ben dipendere
da quali sono gli svantaggi
che il cliente può gestire meglio.

Eberhardt Rechtin



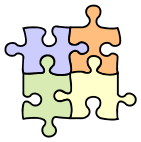
- Fonti

- Java Platform, Enterprise Edition
  http://www.oracle.com/technetwork/java/javaee/
- The Java EE 7 Tutorial
  - *http://docs.oracle.com/javaee/7/tutorial/doc/*
  - Chapter 27, Introduction to Web Services
  - Chapter 28, Building Web Services with JAX-WS
  - Chapter 29, Building RESTful Web Services with JAX-RS
  - Chapter 30, Accessing REST Resources with the JAX-RS Client API
- RESTful Web Services Developer's Guide
  http://docs.oracle.com/cd/E19776-01/820-4867/index.html
- [Papazoglou] Papazoglou, Web Services – Principles and Technology, 2008

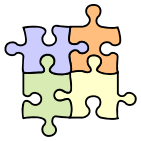


- Obiettivi e argomenti

- Obiettivi
  - introdurre alcuni aspetti relativi alla programmazione dei Web Services – nonché dei loro client – con riferimento a web services SOAP e REST
- Argomenti
  - introduzione
  - programmazione di web services
  - programmazione di web services SOAP
  - programmazione di web services REST

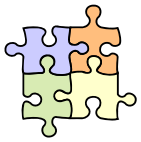


- Wordle



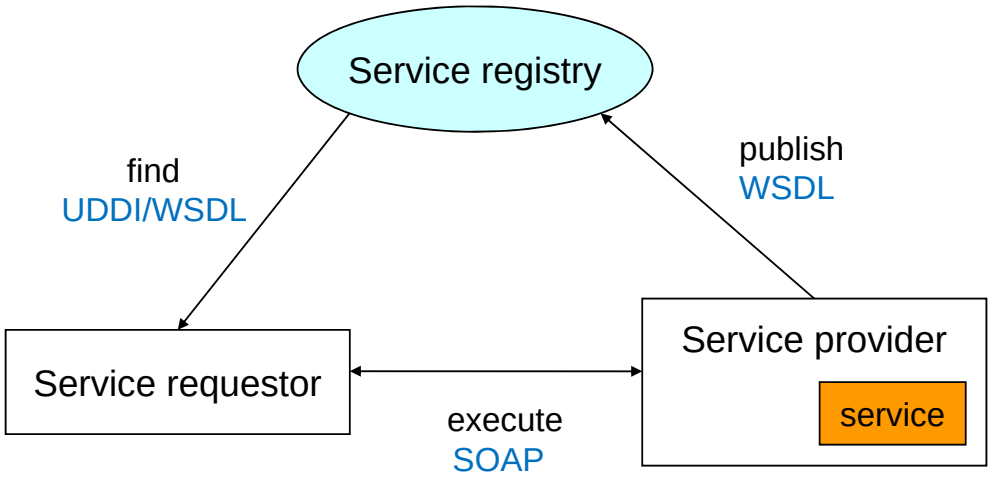
* Introduzione

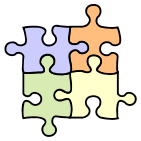
- La tecnologia a *servizi* – con il relativo paradigma di interazione e middleware – sostiene lo sviluppo e l'integrazione di applicazioni distribuite
  - evoluzione del middleware per architetture distribuite
  - generalità dei meccanismi di comunicazione – sia sincroni che asincroni
  - enfasi sull'interoperabilità tra componenti eterogenei, in esecuzione su piattaforme diverse, sulla base di protocolli standard aperti e universalmente accettati
- La tecnologia a servizi "dominante" è, attualmente, quella dei *Web Services* (WS)
  - due "versioni" principali
    - web services SOAP ("big")
    - web services REST ("lightweight")



Standard per Web Services

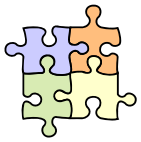
- I Web Services (SOAP) sono basati su un insieme di standard tecnologici – indipendenti dalle piattaforme e neutrali rispetto ad essi





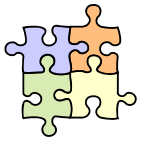
Standard per Web Services

- I web services SOAP (per semplicità chiamati solo WS in questa sezione) sono basati su un insieme di standard
  - *XML*
    - l'adozione di XML come "sintassi" e "linguaggio di trasporto" sostiene l'indipendenza dei vari standard per WS dalla piattaforma e dai linguaggi di programmazione
  - *SOAP*
    - definisce l'organizzazione per lo scambio di dati e messaggi strutturati
  - *WSDL – Web Services Description Language*
    - un linguaggio per la definizione dell'interfaccia di WS
  - *UDDI – Universal Description, Discovery and Integration*
    - standard per la ricerca di WS



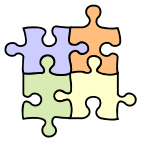
* Programmazione di Web Services

- Che cosa bisogna fare per realizzare un Web Service, oppure utilizzare i servizi di un Web Service esistenti? quanto è complicato utilizzare i diversi standard in gioco?
  - le diverse piattaforme di sviluppo (ad esempio, .NET e Java EE) semplificano lo sviluppo di WS – che sono interoperabili anche tra piattaforme diverse
  - (nei casi più semplici) lo sviluppatore non vede nessuno degli standard in gioco – né XML, né SOAP, né WSDL, ... – perché gli strumenti di sviluppo nascondono (in parte) questa complessità agli sviluppatori
    - in modo tale che gli sviluppatori possano concentrarsi sullo sviluppo di funzionalità, logica applicativa e servizi



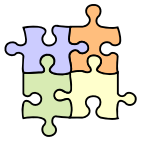
Java EE e Web Services

- La piattaforma Java EE (così come altre piattaforme per lo sviluppo del software) fornisce le API e gli strumenti per realizzare web services e loro client che sono completamente interoperabili, con altri web services e client, realizzati sia con tecnologie Java che con altre tecnologie
  - è possibile avere web services nello stile procedurale, con operazioni, parametri e valori restituiti – oppure web services orientati ai documenti, che consentono di inviare documenti che contengono dati
  - (nei casi più semplici) non è richiesta nessuna programmazione di basso livello, perché l'implementazione delle API si occupa della gestione di tutti i protocolli e i formati dagli standard per web services



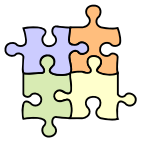
Java EE e Web Services

- Dal tutorial di Java EE
  - i Web Services sono applicazioni o componenti software (client e server), dotati di un punto di accesso in rete, che comunicano sulla base di protocolli standard web
  - i Web Services forniscono un meccanismo standard per l'interoperabilità tra componenti software in esecuzione su piattaforme e framework differenti – realizzati sia con tecnologie Java che con altre tecnologie
  - i Web Services sono caratterizzati da una grande interoperabilità ed estensibilità, grazie all'uso di XML e di protocolli basati su XML
  - i Web Services possono essere combinati, in modo debolmente accoppiato, per definire operazioni complesse – programmi che forniscono servizi semplici possono interagire tra loro per fornire servizi più sofisticati e a valore aggiunto



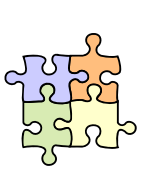
Tipi di web services

- In Java EE, ci sono due tipi principali di Web Services
  - *web services di tipo "big"* (o SOAP) – che usano XML e gli standard SOAP e WSDL
    - in Java, sostenuti dalla tecnologia *JAX-WS*
  - *web services di tipo REST* – che comunicano su HTTP, senza usare né SOAP né WSDL
    - in Java, sostenuti dalla tecnologia *JAX-RS*
    - sono adatti per gli scenari di integrazione più semplici, o per la realizzazione di applicazioni web
    - richiedono un'infrastruttura più leggera dei web services SOAP
    - per scambiare i dati, possono usare XML, JSON o altri formati



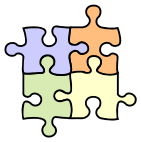
Tipi di web services

- In pratica, può essere preferibile usare
  - WS di tipo REST, per l'integrazione di applicazioni sul web
  - WS di tipo SOAP, in scenari di EAI che hanno requisiti avanzati di qualità del servizio (QoS)
- Infatti, confrontando le due tecnologie
  - JAX-WS affronta meglio i requisiti di qualità comuni delle applicazioni di tipo enterprise – poiché è possibile utilizzare gli standard WS-* che, tra l'altro, sostengono qualità come sicurezza e disponibilità, nonché interoperabilità con altri servizi conformi a WS-*
  - JAX-RS rende più semplice la scrittura di applicazioni web con le caratteristiche dello stile architetturale REST – sostenendo anche alcune qualità come accoppiamento debole, scalabilità e semplicità architetturale – inoltre, per alcuni tipi di client è più semplice consumare servizi REST che non servizi SOAP



* Programmazione di Web Services SOAP

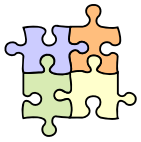
- Viene ora esemplificato un Web Service realizzato con *JAX-WS*
  - *JAX-WS – Java API for XML-Web Services*
    - per implementare sia servizi che client di servizi
    - per realizzare WS che comunicano sia nello stile RPC che nello stile orientato ai messaggi



- Il Web service CalculatorService

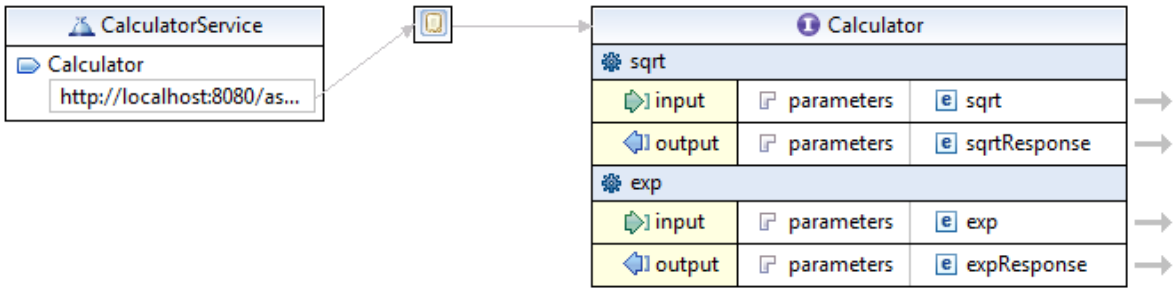
```
package asw.asw860.calculator;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Calculator {

   @WebMethod
   public double sqrt(double x) { return Math.sqrt(x); }

   @WebMethod
   public double exp(double x) { return Math.exp(x); }

}
```

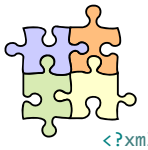
un WS di tipo bottom up – definito nell'ambito di un dynamic web project ***CalculatorWS***



CalculatorService - WSDL (1)

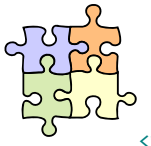
generato automaticamente





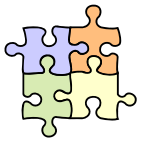
CalculatorService - WSDL (2)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://calculator.asw860.asw" ... >
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://calculator.asw860.asw" ... >
   <element name="sqrt">
    <complexType>
     <sequence>
      <element name="x" type="xsd:double"/>
     </sequence>
    </complexType>
   </element>
   <element name="sqrtResponse">
    <complexType>
     <sequence>
      <element name="sqrtReturn" type="xsd:double"/>
     </sequence>
    </complexType>
   </element>
   <element name="exp">
    <complexType>
     <sequence>
      <element name="x" type="xsd:double"/>
     </sequence>
    </complexType>
   </element>
   <element name="expResponse">
    <complexType>
     <sequence>
      <element name="expReturn" type="xsd:double"/>
     </sequence>
    </complexType>
   </element>
  </schema>
 </wsdl:types>
```



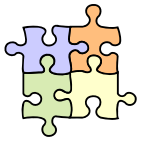
CalculatorService – WSDL (3)

```
<wsdl:message name="expRequest">
   <wsdl:part element="impl:exp" name="parameters">
   </wsdl:part>
</wsdl:message>
<wsdl:message name="expResponse">
   <wsdl:part element="impl:expResponse" name="parameters">
   </wsdl:part>
</wsdl:message>
<wsdl:message name="sqrtResponse">
   <wsdl:part element="impl:sqrtResponse" name="parameters">
   </wsdl:part>
</wsdl:message>
<wsdl:message name="sqrtRequest">
   <wsdl:part element="impl:sqrt" name="parameters">
   </wsdl:part>
</wsdl:message>

<wsdl:portType name="Calculator">
   <wsdl:operation name="sqrt">
      <wsdl:input message="impl:sqrtRequest" name="sqrtRequest">
    </wsdl:input>
      <wsdl:output message="impl:sqrtResponse" name="sqrtResponse">
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="exp">
      <wsdl:input message="impl:expRequest" name="expRequest">
    </wsdl:input>
      <wsdl:output message="impl:expResponse" name="expResponse">
    </wsdl:output>
   </wsdl:operation>
</wsdl:portType>
```

CalculatorService – WSDL (4)

```
<wsdl:binding name="CalculatorSoapBinding" type="impl:Calculator">
   <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="sqrt">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="sqrtRequest">
         <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="sqrtResponse">
         <wsdlsoap:body use="literal"/>
      </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="exp">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="expRequest">
         <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="expResponse">
         <wsdlsoap:body use="literal"/>
      </wsdl:output>
   </wsdl:operation>
</wsdl:binding>

<wsdl:service name="CalculatorService">
   <wsdl:port binding="impl:CalculatorSoapBinding" name="Calculator">
      <wsdlsoap:address location="http://localhost:8080/asw-860a-CalculatorWS/services/Calculator"/>
   </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```



- Client per CalculatorService

- In generale, nella realizzazione di un *client* per un WS, è possibile utilizzare un compilatore di specifiche WSDL
  - si tratta di un'applicazione di solito fornita dagli ambienti di sviluppo (SDK) per WS – spesso direttamente accessibile dagli IDE
  - un compilatore WSDL genera, a partire da una specifica WSDL (o dalla sua URI), un client proxy (stub) composto da un certo numero di classi e interfacce
    - l'applicazione client viene poi realizzata facendo esplicito utilizzo di questo stub – oppure definendo un ulteriore proxy o adattatore
  - sono spesso possibili diverse modalità di realizzazione (binding) dello stub

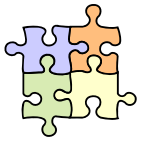
. Client Java per CalculatorService (AXIS)

```
package asw.asw860.client;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

import asw.asw860.calculator.*;

public class CalculatorProxy {

    private Calculator calculator;

    public CalculatorProxy() throws ServiceException {
        CalculatorService locator = new CalculatorServiceLocator();
        this.calculator = locator.getCalculatorPort ();
    }

    public double sqrt(double x) throws RemoteException {
        return calculator.sqrt(x);
    }

}
```

attenzione: anche se il nome del package è lo stesso, in realtà contiene classi e interfacce diverse da quelle dell'implementazione del servizio

tipi generati automaticamente come proxy al servizio

casualmente ha lo stesso nome dell'implementazione del servizio – ma è un'interfaccia RMI per il servizio



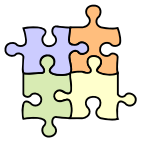
Un'interazione con CalculatorService

- Richiesta

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <S:Body>
        <ns2:sqrt xmlns:ns2="http://calculator.asw860.asw/">
            <arg0>100.0</arg0>
        </ns2:sqrt>
    </S:Body>
</S:Envelope>
```

- Risposta

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <S:Body>
        <ns2:sqrtResponse xmlns:ns2="http://calculator.asw860.asw/">
            <return>10.0</return>
        </ns2:sqrtResponse>
    </S:Body>
</S:Envelope>
```

. Client C# per CalculatorService

stub generato da Visual Studio a partire dall'URI del servizio

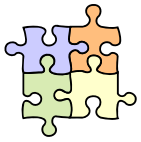
```
using System;
using System.Collections.Generic;
using System.Text;

using calculatorservice.CalculatorService;

namespace Asw.Asw860.CalculatorServiceClient
{
   class Program
   {
      static void Main(string[] args)
      {
         CalculatorService calculator = new CalculatorService();

         System.Console.WriteLine( calculator.sqrt(144.0) );
         System.Console.WriteLine( calculator.exp(3.0) );
      }
   }
}
```

. Client Java per CalculatorService (AXIS2)

```
package asw.asw860.calculator.client;

import java.rmi.RemoteException;

import asw.asw860.calculator.CalculatorServiceStub;
import asw.asw860.calculator.CalculatorServiceStub.*;

public class CalculatorProxy {
    private CalculatorServiceStub stub;

    public CalculatorProxy() throws RemoteException {
        this.stub = new CalculatorServiceStub();
    }

    public double sqrt(double x) throws RemoteException {
        Sqrt request = new Sqrt();
        request.setX(x);
        SqrtResponse response = stub.sqrt(request);
        return response.get_return();
    }
}
```

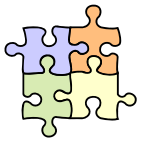

Client asincrono con callback (AXIS2)

```
public class CalculatorProxyAsincrono {
    ...

    public void startSqrt(double x) throws RemoteException {
        Sqrt request = new Sqrt();
        request.setX(x);
        CalculatorServiceCallbackHandler callback =
                    new CalculatorServiceCallbackHandler(this) {
                public void receiveResultsqrt(SqrtResponse response) {
                    double radiceX = response.get_return();
                    CalculatorProxyAsincrono proxy =
                                (CalculatorProxyAsincrono) getClientData();
                    proxy.callbackSqrt(radiceX);
                }
                 public void receiveErrorecho(Exception e) { ... gestisci l'eccezione e ... }
        };
        stub.startsqrt(request, callback);
    }

    public void callbackSqrt(double radiceX) { ... fa qualcosa con radiceX ... }

}
```



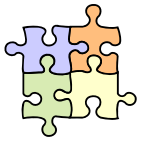
Discussione

- Alcune funzionalità offerte da JAX-WS
  - generazione bottom up di WS a partire da classi Java (POJO) – oppure da enterprise bean di tipo stateless
  - generazione top down dello skeleton dell'implementazione di un WS a partire da una specifica WSDL – consente un uso più flessibile dei diversi MEP
  - il ciclo di vita delle istanze dei servizi è gestito da un contenitore – con la possibilità di definire metodi di callback associati alla gestione del ciclo di vita dei servizi
  - diverse modalità di generazione dello stub per i client di un WS a partire da una specifica WSDL – con riferimento a diverse modalità di binding dei dati – di solito maggior flessibilità nell'utilizzo di un servizio comporta maggior complessità del codice da scrivere (e viceversa)



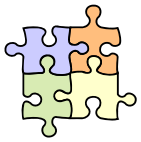
* Programmazione di Web Services REST

- Viene ora esemplificato un Web Service realizzato con *JAX-RS*
  - *JAX-RS – Java API for RESTful Web Services*
    - per implementare sia servizi che client di servizi
    - per realizzare semplici WS stateless che comunicano tramite HTTP
    - per favorire il consumo di questi servizi nell'ambito di applicazioni web
  - le operazioni offerte da un servizio REST sono definite in corrispondenza con le operazioni HTTP GET, PUT, POST e DELETE
  - la rappresentazione restituita dall'operazione GET è espressa in un formato di interscambio opportuno – ad es., testo, HTML, XML oppure JSON – ma anche PDF, JPEG, …



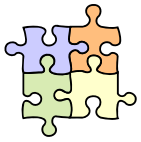
Servizi nello stile REST

- Un esempio di servizio nello stile REST
  - il servizio gestisce una collezione di risorse
    - ad esempio, un insieme di prodotti
  - il servizio ha un'URI di base – chiamata *collection URI*
    - ad es., *http://localhost:8080/GestioneProdotti/prodotti*
  - ogni istanza di risorsa ha un'URI – chiamata *element URI*
    - ad es., *http://localhost:8080/GestioneProdotti/prodotti/123*
  - le operazioni del servizio sono messe in corrispondenza con le operazioni HTTP GET, PUT, POST e DELETE – riferite alla collection URI e/o all'element URI



Servizi nello stile REST

- Un servizio nello stile REST
  - operazioni riferite a una *collection URI*
    - GET – restituisce tutti gli elementi della collezione
    - PUT – sostituisce la collezione con un'altra collezione
    - POST – crea un nuovo elemento della collezione e gli assegna una nuova URI
    - DELETE – cancella l'intera collezione
  - operazioni riferite a un'*element URI*
    - GET – restituisce uno specifico elemento della collezione
    - PUT – crea un nuovo elemento della collezione, oppure lo aggiorna
    - POST – considera l'elemento della collezione come un'altra collezione, e ne aggiunge un elemento
    - DELETE – cancella l'elemento della collezione



- Classe (Java bean) per la risorsa

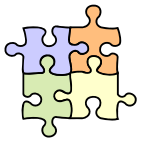
```
package asw.asw865.prodotti;

public class Prodotto {

    private String id;
    private String descrizione;
    private int prezzo;

    public Prodotto() { }

    public Prodotto(String id, String descrizione, int prezzo) {
        this.id = id;
        this.descrizione = descrizione;
        this.prezzo = prezzo;
    }

    ... metodi set e get ...

}
```


Repository per la risorsa

```
package asw.asw865.prodotti;

import java.util.*;

public class GestioneProdottiRepository {

    ... variabile e metodi per singleton ...

    private Map<String,Prodotto> prodotti;

    private GestioneProdottiRepository() {
        this.prodotti = new HashMap<String,Prodotto>();
    }

    public void aggiungiProdotto(Prodotto p) { prodotti.put(p.getId(), p); }

    public Prodotto trovaProdotto(String id) { return prodotti.get(id); }

    public Prodotto cancellaProdotto(String id) { return prodotti.remove(id); }

    public Collection<Prodotto> tuttiProdotti() { return prodotti.values(); }

}
```

Classe container – per la collection URI (1)

```
package asw.asw865.prodotti;

import ...

@Path("/prodotti")
@Produces( { MediaType.APPLICATION_XML,
             MediaType.APPLICATION_JSON,
             MediaType.TEXT_XML } )
@Consumes( MediaType.APPLICATION_FORM_URLENCODED )
public class ProdottoContainer {
    @Context
    private UriInfo context;

    private GestioneProdottiRepository repository;

    public ProdottoContainer() {
        this.repository = GestioneProdottiRepository.getInstance();
    }

    ... segue ...
```

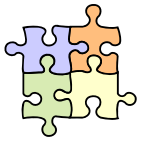



Classe container – per la collection URI (2)

```
    /* GET: Restituisce l'elenco di tutti i prodotti */
    @GET
    public Collection<Prodotto> getProdotti() {
        Collection<Prodotto> prodotti = new ArrayList<Prodotto>();
        prodotti.addAll( repository.tuttiProdotti() );
        return prodotti;
    }

    ... segue ...
```

il valore restituito dall'operazione GET viene concretamente restituito al client in uno dei formati di scambio specificati, sulla base dei formati accettati dal client

questa conversione di formato viene effettuata dal contenitore



Classe container – per la collection URI (3)

```
    /* POST: Aggiunge un nuovo prodotto */
    @POST
    public Response postProdotto(
                @FormParam("id") String id,
                @FormParam("descrizione") String descrizione,
                @FormParam("prezzo") int prezzo,
                @Context HttpServletResponse servletResponse
        ) {
        Prodotto p = new Prodotto(id, descrizione, prezzo);
        repository.aggiungiProdotto(p);

        URI uri = context.getAbsolutePathBuilder().path(id).build();
        return Response.created(uri).build();
    }

}
```

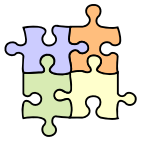
i parametri dell'operazione POST forniti dal client vengono legati con i parametri del metodo

anche questa attività viene effettuata dal contenitore

Classe resource – per l'element URI (1)

```
package asw.asw865.prodotti;

import ...

@Path("/prodotti/{id}")
public class ProdottoResource {
    @Context
    private UriInfo uriInfo;

    private GestioneProdottiRepository repository;

    public ProdottoResource() {
        this.repository = GestioneProdottiRepository.getInstance();
    }

  ... segue ...
```

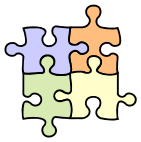
{id} nel path viene comunicato alle operazioni del servizio come un parametro



Classe resource – per l'element URI (2)

```
/* GET: Cerca un prodotto */
@GET
@Produces( { MediaType.APPLICATION_XML,
                MediaType.APPLICATION_JSON,
                MediaType.TEXT_XML } )
public Prodotto getProdotto(@PathParam("id") String id) {
    Prodotto p = repository.trovaProdotto(id);
    if (p==null) {
        String errorMessage = "Prodotto with id: " + id +  " not found";
        throw new WebApplicationException(
            Response.status(404).entity(errorMessage).type("text/plain").build()
        );
    }
    return p;
}
```

... segue ...



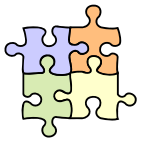
Classe resource – per l'element URI (3)

```
/* DELETE: Cancella un prodotto */
@DELETE
public Prodotto deleteProdotto(@PathParam("id") String id) {
    Prodotto p = repository.cancellaProdotto(id);
    if (p==null) {
        String errorMessage = "Prodotto with id: " + id +  " not found";
        throw new WebApplicationException(
            Response.status(404).entity(errorMessage).type("text/plain").build()
        );
    }
    return p;
}

}
```

Classe application – per il servizio

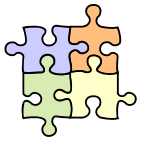
```
package asw.asw865.prodotti;

import …

@ApplicationPath("/services")
public class GestioneProdottiApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        /* registra le risorse che definisco il servizio */
        classes.add(ProdottoContainer.class);
        classes.add(ProdottoResource.class);
        return classes;
    }

}
```

definisce il path di base per il servizio – complessivamente l'URI base sarà services/prodotti



- Client HTTP di servizi REST

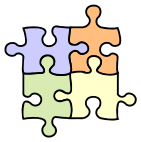
- Un web service di tipo REST può essere acceduto da una molteplicità di client
  - direttamente tramite richieste HTTP, da un browser – nell'ambito di un'applicazione web
  - in modo programmatico
    - tramite richieste HTTP
    - scambiando dati XML
    - scambiando oggetti JSON – un formato di interscambio, che può essere utilizzato da molti linguaggi di programmazione



Client HTTP di servizi REST

- Esempi di uso del web service GestioneProdotti
  - crea un nuovo prodotto di id 123

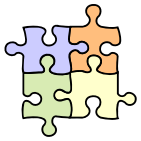
```
POST /GestioneProdotti/services/prodotti
User-Agent: ...
Host: localhost:8080
Accept: */*
Content-Type: application/x-www-form-urlencoded
id=123&descrizione=prodotto123&prezzo=10

HTTP/1.1 201 Created
Location: http://localhost:8080/GestioneProdotti/services/prodotti/123
Content-Type: application/xml
Content-Length: 0
```

Client HTTP di servizi REST

- Esempi di uso del web service GestioneProdotti
  - restituisce la rappresentazione del prodotto di id 123

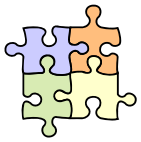
```
GET /GestioneProdotti/services/prodotti/123
User-Agent: ...
Host: localhost:8080
Accept: */*

HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<prodotto>
    <descrizione>prodotto123</descrizione>
    <id>123</id>
    <prezzo>10</prezzo>
</prodotto>
```



Client HTTP di servizi REST

- Esempi di uso del web service GestioneProdotti
  - restituisce l'elenco di tutti i prodotti

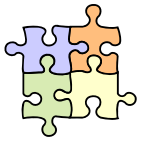
```
GET /GestioneProdotti/services/prodotti
User-Agent: ...
Host: localhost:8080
Accept: */*

HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<prodottoes>
    <prodotto>
        <descrizione>prodotto123</descrizione>
        <id>123</id>
        <prezzo>10</prezzo>
    </prodotto>
    <prodotto>...</prodotto>
    <prodotto>...</prodotto>
</prodottoes>
```

Client HTTP di servizi REST

- Esempi di uso del web service GestioneProdotti
  - cancella il prodotto di id 123

```
DELETE /GestioneProdotti/services/prodotti/123
User-Agent: ...
Host: localhost:8080
Accept: */*

HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<prodotto>
    <descrizione>prodotto123</descrizione>
    <id>123</id>
    <prezzo>10</prezzo>
</prodotto>
```



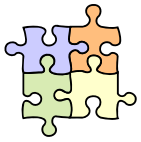
- Client Java di servizi REST

- Un web service di tipo REST può essere acceduto anche da client Java – usando *JAX-RS Client API*
  - per accedere servizi REST di qualunque tipo – e non solo a servizi REST implementati in Java con JAX-RS
  - in modo programmatico
    - utilizzando un oggetto Client
    - tramite il Client, è possibile specificare
      - l'URI della richiesta
      - il tipo di richiesta (GET, POST, PUT, DELETE) con i relativi parametri
      - invocare la richiesta
      - ottenere una risposta – nel formato desiderato

Proxy al servizio REST (1)

```
package asw.asw865.prodotti.client;

import javax.ws.rs.*;
import javax.ws.rs.client.*;
import javax.ws.rs.core.*;
import …

/* Proxy al servizio REST per la gestione dei prodotti. */
public class GestioneProdottiServiceProxy {

    private Client client;    // il client REST delle JAX-RS Client API
    private final String SERVICE_URL =
            "http://localhost:8080/GestioneProdotti-REST/services/prodotti";

    /* Crea il proxy per il servizio di gestione prodotti. */
    public GestioneProdottiServiceProxy() {
        /* crea il client, e lo abilita all'utilizzo di JSON */
        client = ClientBuilder.newClient();
        client.register( new JacksonFeature() );
    }

    /* Chiude il client REST. */
    public void close() {
        if (client!=null) { client.close(); }
    }
```

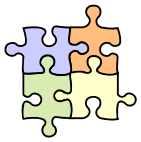
Client è l'interfaccia principale delle JAX-RS Client API



Proxy al servizio REST (2)

```
/* Richiede la collezione di tutti i prodotti (come una stringa JSON). */
public String getProdottiAsJsonString() {
    String result = client.target(SERVICE_URL)
                    .request().
                    .accept(MediaType.APPLICATION_JSON)
                    .get()
                    .readEntity(String.class);
    return result;
}
```

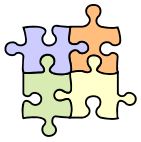
- target() consente di specificare l'URI (base) del servizio
- request() inizia la costruzione di una richiesta
- accept() specifica i tipi di risposte accettate
- get() invoca l'operazione HTTP GET
- readEntity() estrae il risultato dalla risposta



Proxy al servizio REST (3)

```
/* Richiede la collezione di tutti i prodotti. */
public Collection<Prodotto> getProdotti() {
    Collection<Prodotto> result = client.target(SERVICE_URL)
                    .request(MediaType.APPLICATION_JSON)
                    .get(new GenericType<Collection<Prodotto>>() {});
    return result;
}
```

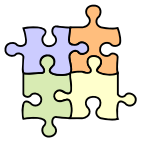
- in pratica, alcune delle operazioni possono essere combinate – in particolare, accept() con request() e readEntity() con get()
- l'API accede al servizio REST e richiede una rappresentazione JSON di tutti i prodotti – l'API si occupa anche di convertire questa rappresentazione JSON a una rappresentazione in Java dei prodotti



Proxy al servizio REST (4)

```
/* Richiede un prodotto. */
public Prodotto getProdotto(String id) {
    try {
        Prodotto result = client.target(SERVICE_URL)
            .path(id)
            .request(MediaType.APPLICATION_JSON)
            .get(Prodotto.class);
        return result;
    } catch(NotFoundException e) {
        return null;
    }
}
```

- path() consente di dettagliare ulteriormente l'URI della risorsa desiderata
- anche qui l'API si occupa della conversione del prodotto restituito da JSON a Java



Proxy al servizio REST (5)

```
/* Aggiunge un prodotto. */
public void addProdotto(Prodotto p) {
    Form form = new Form()
            .param("id", p.getId())
            .param("descrizione", p.getDescrizione())
            .param("prezzo", String.valueOf(p.getPrezzo()));
    client.target(SERVICE_URL)
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .post(Entity.form(form));
}

/* Cancella un prodotto. */
public void deleteProdotto(String id) {
    client.target(SERVICE_URL)
            .path((id))
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .delete();
}
}
```

- post() e delete() corrispondono alle operazioni HTTP POST e DELETE