Serveis web amb Java EE 7

Josep Maria Camps i Riba

Desenvolupament web en entorn servidor

Índex

Introducció

Els **serveis web**, ja siguin **SOAP** o **REST**, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (**SOA**, per les sigles en anglès: Service Oriented Architecture) i, més recentment, per a la creació d'arquitectures basades en microserveis. La seva principal característica és la **interoperativitat**, ja que permeten que aplicacions escrites en llenguatges diferents i que s'executen en plataformes diferents puguin interactuar per construir aplicacions distribuïdes seguint arquitectures SOA.

Veurem els conceptes més rellevants dels serveis web SOAP amb Java EE 7 com a tecnologia. Mitjançant els exemples veureu les bases teòriques que regeixen els serveis web SOAP, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne, a fer-ne el desplegament i a codificar diferents tipus de clients capaços de consumir els serveis web creats.

Explicarem els conceptes més rellevants dels serveis web RESTful amb Java EE 7 com a tecnologia. Mitjançant els exemples veureu les bases teòriques que regeixen els serveis web RESTful, quins en són els components més rellevants i quina relació hi ha entre si. Aprendreu a crear-ne i a fer-ne el desplegament.

Crearem clients que consumeixin serveis web RESTful amb Java EE 7 com a tecnologia. Mitjançant els exemples, veureu com es codifiquen diferents tipus de client capaços de consumir serveis web RESTful.

La unitat descriu, des d'un vessant teòric i pràctic, els aspectes més essencials dels diferents tipus de serveis web que podem crear. Tots els apartats d'aquesta unitat s'han elaborat seguint exemples pràctics per introduir i aprofundir els conceptes abans esmentats. Es recomana que l'estudiant faci els exemples mentre els va llegint, així anirà aprenent mentre va practicant els conceptes exposats. Finalment, per treballar completament els continguts d'aquesta unitat és convenient anar fent les activitats i els exercicis d'autoavaluació.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa serveis web analitzant el seu funcionament i implantant l'estructura dels seus components.

- Identifica les característiques pròpies i l'àmbit d'aplicació dels serveis web.
- Reconeix els avantatges d'utilitzar serveis web per proporcionar accés a funcionalitats incorporades a la lògica de negoci d'una aplicació.
- Determina les tecnologies i els protocols implicats en la publicació i utilització de serveis web.
- Programa un servei web.
- Crea el document de descripció del servei web.
- Verifica el funcionament del servei web.
- Consumeix el servei web.

1. Serveis web SOAP amb Java EE 7

Veurem, mitjançant una aplicació d'exemple, els conceptes més rellevants dels serveis web SOAP. Aprendreu a crear-ne, a fer-ne el desplegament i a codificar diferents tipus de clients capaços de consumir els serveis web creats.

Els serveis web, ja siguin SOAP o REST, són una peça clau en el desenvolupament d'arquitectures de programari orientades a serveis (SOA per les sigles en anglès: Service Oriented Architecture) i, mes recentment, per a la creació d'arquitectures basades en microserveis.

> **SOAP** (de l'anglès *Simple Object Access Protocol*) és un protocol que permet la interacció de serveis web basats en XML. L'especificació de SOAP inclou la sintaxi amb la qual s'han de definir els missatges, les regles de codificació dels tipus de dades i les regles de codificació que regiran les comunicacions entre aquests serveis web.

Quan es descriu una arquitectura **SOAP** ens referim a arquitectures basades en un conjunt de serveis que es despleguen a Internet mitjançant **serveis web**.

> Un **servei web** és una tecnologia que fa servir un conjunt de protocols i estàndards per tal d'intercanviar dades entre aplicacions.

Podeu veure els serveis web com a components d'aplicacions distribuïdes que estan disponibles de forma externa i que es poden fer servir per integrar aplicacions escrites en diferents llenguatges (Java, .NET, PHP, etc.) i que s'executen en plataformes diferents (Windows, Linux, etc.). **La seva característica principal és, doncs, la interoperativitat**.

Un servei web publica una lògica de negoci exposada com a servei als clients. La diferencia més gran entre una lògica de negoci exposada com a servei web i, per exemple, una lògica de negoci exposada amb un mètode d'un EJB és que els serveis web SOAP proporcionen una interfície poc acoblada als clients. Això permet comunicar aplicacions que s'executin en diferents sistemes operatius, desenvolupades amb diferents tecnologies i amb diferents llenguatges de programació.

Els serveis web i, per tant, les aplicacions orientades a serveis es poden implementar amb diferents tecnologies. Les dues implementacions principals de serveis web que formen part de Java EE 7 són els serveis web **SOAP** i els serveis web **RESTful**.

Els serveis web SOAP són força més complexos que els serveis web RESTful, però proporcionen certes capacitats a nivell de seguretat i transaccionalitat que, a vegades, els fan l'única alternativa viable, sobretot en aplicacions empresarials.

En aquest capítol ens centrarem en la creació i el consum de serveis web SOAP amb Java EE 7.

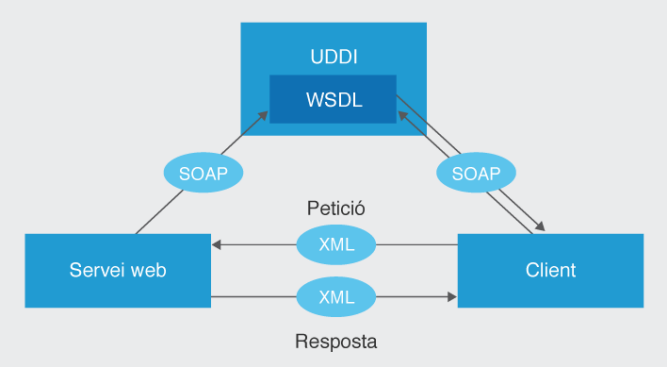
El proveïdor del servei web SOAP el dissenya, l'implementa i en publica la seva interfície i el format dels missatges amb **WSDL** (de l'anglès Web Services Description Language), i els clients consulten aquesta definició i es comuniquen amb el servei web seguint la interfície i el format de missatges que ha publicat el proveïdor del servei en el document WDSL de definició del servei web.

> **WSDL** descriu on es localitza un servei, quines operacions proporciona, el format dels missatges que han de ser intercanviats i com cal cridar el servei.

Opcionalment es pot publicar la definició WSDL del servei web en un registre **UDDI** (de l'anglès Universal Description, Dicovery and Integration) per tal que els clients puguin fer-hi cerques. Tingueu en compte que actualment no existeix un registre global que inclogui els diferents serveis web i, per això, UDDI ha passat a ser molt poc utilitzat a la pràctica. Normalment, el client coneix l'adreça del document WSDL de descripció del servei i, a partir d'aquest, invoca el servei web. Vegeu la relació entre els diferents components d'un servei web SOAP en la figura 1.1.

Quan parlem de serveis web, a les operacions se les sol anomenar *endpoints*. Al text farem servir indistintament qualsevol de les dues paraules.

Figura 1.1. Components i relacions d'un servei web SOAP



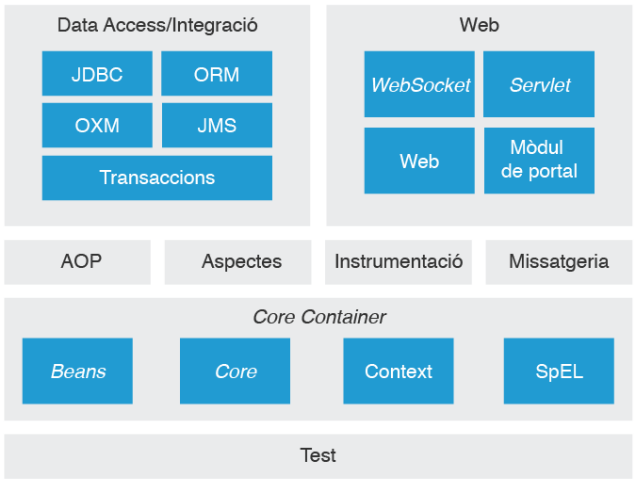
L'especificació de Java EE 7 inclou diverses especificacions que donen complet suport a la creació i el consum de serveis web SOAP; la més destacada és **JAX-WS** (de l'anglès Java API for XML-Based Web Services), que utilitza missatges XML seguint el protocol SOAP i oculta la complexitat d'aquest protocol proporcionant una API senzilla per al desenvolupament, el desplegament i el consum de serveis web SOAP amb Java EE.

> **JAX-WS** és l'API estàndard que defineix Java EE per desenvolupar i desplegar serveis web SOAP i permet ocultar la complexitat inherent a aquest protocol.

Tal com podeu veure en la figura 1.2, el *runtime* de JAX-WS converteix les crides a l'API en missatges SOAP i els missatges SOAP en crides a l'API i ho envia per HTTP. La implementació de JAX-WS que incorporen els servidors d'aplicacions

compatibles amb Java EE 7 també proporciona eines per generar els documents WSDL de definició dels serveis web i eines per generar els artefactes que faran d'intermediaris entre el client i el servei web. Aquests artefactes s'anomenen *stubs* a la part del client i *ties* a la part del servei web.

Figura 1.2. Esquema de comunicació entre un servei web SOAP i un client amb JAX-WS



1.1 Gestió de reserva de places a concerts com a servei web SOAP

Veurem els conceptes referents a la definició de serveis web SOAP amb Java EE 7 desenvolupant un servei web SOAP que permeti les següents operacions:

- Proporcionar una llistat de concerts i el nombre de places lliures a cada un d'ells.
- Fer la reserva d'una entrada per a un d'aquests concerts.
- Cancel·lar una reserva d'una entrada.

Estratègia 'top-down'

Tot i ser força més feixuc, també podríem haver seguit una estratègia *top-down*, creant primer el document de WSDL de definició i, a partit d'aquest, generar automàticament el codi Java que l'implementi.

Farem servir una aplicació web ja desenvolupada que segueixi una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió de reserves com a servei web SOAP.

Per a aquest exemple farem servir una aproximació *bottom-up*; crearem primer el codi Java que implementarà el servei web i, a partir d'aquest, es generarà automàticament el document WSDL que el descriurà.

1.1.1 Creació i configuració inicial del projecte

L'aplicació de la qual partirem s'anomena "Resentioc" i ens servirà per veure com podem exposar algunes funcionalitats de la capa de serveis d'una aplicació

mitjançant serveis web SOAP. Es tracta d'un projecte Spring MVC senzill que segueix una arquitectura típica per capes per tal d'aconseguir una alta reusabilitat, un baix acoblament i una alta cohesió a l'aplicació.

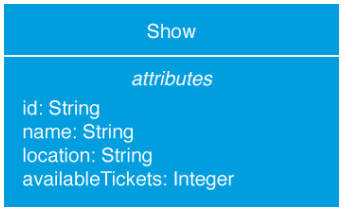
El projecte constarà de quatre capes:

- Capa de presentació
- Capa de domini
- Capa de serveis
- Capa de persistència

A l'exemple no farem servir la capa de presentació, ja que l'objectiu no és proporcionar una interfície d'usuari a l'aplicació, sinó publicar els mètodes de negoci com a serveis web.

El model de domini ja el teniu implementat i és molt senzill, només hi ha una entitat `Show` que teniu en la figura 1.3 i que representa els concerts que s'estan oferint amb el nombre d'entrades disponibles.

FIGURA 1.3. Entitat Show



La capa de serveis serà la que treballarem i haurà de proporcionar un **servei web SOAP** que permeti als clients fer les següents operacions:

- Llistat de concerts amb el nombre d'entrades disponibles per a cada un d'ells.
- Reservar una entrada.
- Anul·lar una reserva.

La capa de persistència també la teniu implementada i conté l'objecte repositori que permet mapar les dades de la font de dades amb l'objecte de domini. En el nostre cas, per simplificar, farem servir un repositori *in memory* que tindrà una llista precarregada de concerts.

1.1.2 Creació del servei web SOAP

L'objectiu d'aquest apartat és crear un servei web SOAP, seguint l'especificació JAX-WS, amb les tres operacions que corresponen a les tres funcionalitats que volem publicar:

- Llistat de concerts amb el nombre d'entrades disponibles per cada un d'ells.
- Reservar una entrada.
- Anul·lar una reserva.

Creació dels serveis web

Primer ho farem "a mà", sense gaire ajut de NetBeans, per tal que veieu i entengueu tot el procés. Veurem que NetBeans us pot ajudar força en el procés de creació dels serveis web, però és molt important que entengueu tot el procés abans de fer servir eines automatitzades.

Els serveis web SOAP amb Java EE 7 es poden implementar de dues maneres: amb una classe Java normal que s'executa en un contenidor de *servlets* o bé com un EJB de sessió sense estat o *singleton* que s'executa en un contenidor d'EJB. Veurem com s'implementa el servei web amb una classe Java normal.

Descarregueu el codi del projecte "Resentioc" en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans. Tot i que podeu descarregar-vos també el projecte en l'estat final des de un altre enllaç als annexos, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

Creeu una interfície que exposarà les tres operacions que volem; anomenarem la interfície `TicketServiceEndpoint` i la crearem al paquet `cat.xtec.ioc.service`.

A la interfície hi creeu els tres mètodes que volem exposar com a *endpoints* del servei web i els anoteu amb l'anotació `javax.jws.WebMethod`:

```
1  package cat.xtec.ioc.service;
2
3  import cat.xtec.ioc.domain.Show;
4  import java.util.ArrayList;
5  import javax.jws.WebMethod;
6  import javax.jws.WebService;
7  import javax.jws.soap.SOAPBinding;
8
9  @WebService
10 @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
11 public interface TicketServiceEndpoint {
12      @WebMethod ArrayList<Show> getAllShows();
13      @WebMethod Show makeReservation(String showId);
14      @WebMethod Show cancelReservation(String showId);
15 }
```

En les noves versions de Java EE, la creació explícita d'una interfície que caldrà que el servei web implementi és opcional.

Fixeu-vos que:

- Hem anotat la interfície amb l'anotació `@WebService` per indicar que la interfície correspondrà a un servei web.
- Hem anotat la interfície amb l'anotació `@SOAPBindig` per tal d'especificar l'estil de servei que volem crear. Per l'exemple, utilitzarem l'estil *Document*, que és l'opció per defecte.
- Tots els mètodes que volem exposar cal que els anoteu amb l'anotació `@WebMethod`.

Un cop fet això cal que creeu la implementació del servei web; per fer-ho, creeu una nova classe anomenada `TicketServiceEndpointImpl` al paquet `cat.xtec.ioc.service.impl`. En aquesta classe cal que hi implementeu els tres mètodes que volem exposar com a *endpoints* del servei web:

```
1  package cat.xtec.ioc.service.impl;
2
3  import cat.xtec.ioc.domain.Show;
4  import cat.xtec.ioc.domain.repository.ShowRepository;
5  import cat.xtec.ioc.domain.repository.impl.InMemoryShowRepository;
6  import javax.jws.WebService;
```

```
import cat.xtec.ioc.service.TicketServiceEndpoint;
import java.util.ArrayList;


@WebService(serviceName = "TicketService",
        endpointInterface = "cat.xtec.ioc.service.TicketServiceEndpoint")
public class TicketServiceEndpointImpl implements TicketServiceEndpoint {

    private final ShowRepository showRepository = new InMemoryShowRepository();

    @Override
    public ArrayList<Show> getAllShows() {
        return new ArrayList<Show>(this.showRepository.getAllShows());
    }

    @Override
    public Show makeReservation(String showId) {
        return this.showRepository.makeReservation(showId);
    }

    @Override
    public Show cancelReservation(String showId) {
        return this.showRepository.cancelReservation(showId);
    }
}
```

Hem anotat la classe amb l'anotació `@WebService` i li hem indicat amb l'atribut `name` que els clients faran referència al servei web amb el nom `TicketService`. També li hem indicat quina és la interfície que implementa el servei web amb la ruta completa de la interfície `TicketServiceEndpoint`:

```
@WebService(serviceName = "TicketService",
        endpointInterface = "cat.xtec.ioc.service.TicketServiceEndpoint")
public class TicketServiceEndpointImpl implements TicketServiceEndpoint {
```

La classe implementa la interfície que defineix el servei web i proporciona una implementació per a cada un dels tres mètodes que implementaran les operacions que publica el servei web cridant mètodes existents del repositori de concerts. A aquests mètodes no cal afegir-hi cap anotació especial; el fet d'implementar una interfície amb mètodes anotats amb `@WebMethod` ja indica quins mètodes de la classe corresponen als *endpoints* del servei web.

I aquesta és tota la feina que ens cal fer per implementar el servei web de reserva d'entrades, ara tan sols ens manca fer-ne el desplegament al servidor d'aplicacions i provar-lo.

1.1.3 Desplegament del servei web SOAP

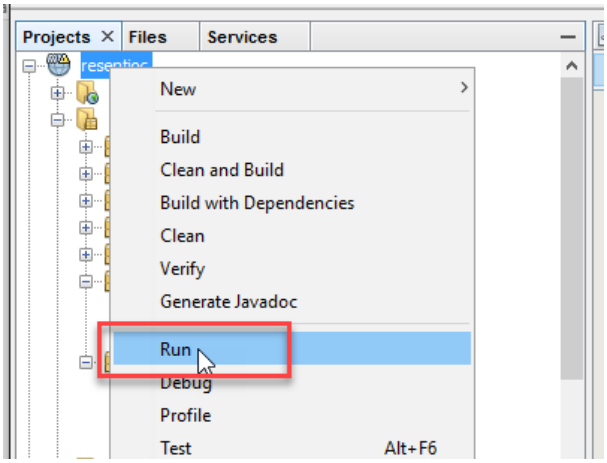
Un cop creat el servei web cal que el desplegueu per tal de fer-lo accessible als clients. El procés de desplegament del servei web engega un conjunt de processos definits a JAX-WS per tal de generar el document WSDL de descripció del servei i publicar els *endpoints* del servei web.

El desplegament del servei web es pot fer de diverses maneres, entre elles:

- Desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.
- Creant una aplicació Java que s'encarregui de publicar el servei web a un URL determinat.

El desplegament del servei web com a part de l'aplicació Java EE que el conté és molt senzill, simplement cal que feu *Run* a NetBeans (vegeu la figura 1.4) i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte:

Figura 1.4. 'Run' a NetBeans



En fer el desplegament, la implementació de JAX-WS que té el servidor d'aplicacions ja genera automàticament el document WSDL de definició del servei i publica els *endpoints* que heu definit amb el nom de servei que li heu donat (en el cas que ens ocupa l'hem anomenat `TicketService`). La figura 1.5 mostra la sortida de la consola de NetBeans amb el desplegament del servei web:

Figura 1.5. Desplegament a NetBeans

```
Información:   Webservice Endpoint deployed TicketServiceEndpointImpl
 listening at address at http://BCN-CLI-P113:8080/resentioc/TicketService.
```

Podeu provar que el desplegament des de NetBeans ha anat bé consultant el document WSDL generat en el següent enllaç: localhost:8080/resentioc/TicketService?wsdl.

El desplegament mitjançant una aplicació Java que s'encarregui de publicar el servei web és una mica més complexa, i ens caldrà crear una classe Java que tingui un mètode `main` que cridi el mètode `publish` de l'objecte `javax.xml.ws.Endpoint`. Per fer-ho creem una classe `TicketServicePublisher` al paquet `cat.xtec.ioc.publisher` amb el següent codi:

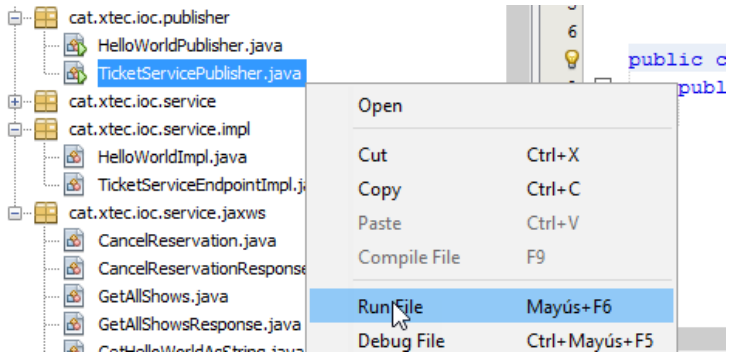
```
1  package cat.xtec.ioc.publisher;
2
3  import cat.xtec.ioc.service.impl.TicketServiceEndpointImpl;
4  import javax.xml.ws.Endpoint;
5
6  public class TicketServicePublisher {
```

```
7
8       public static void main(String[] args) {
9           Endpoint.publish("http://localhost:9999/publisher/TicketService", new
                TicketServiceEndpointImpl());
10      }
11  }
```

Fixeu-vos que al mètode `publish` de l'objecte `javax.xml.Endpoint` cal que li passeu l'URL on voleu publicar el servei web i una instància de la classe que implementa el servei web.

Amb això ja podeu publicar el servei web de reserva d'entrades executant aquesta aplicació Java; per fer-ho, poseu-vos damunt de la classe a NetBeans i feu *Run File* (vegeu la figura 1.6).

FIGURA 1.6. 'Run File' a NetBeans



Si ho feu, veureu el següent error a la consola de NetBeans:

```
1  Exception in thread "main" com.sun.xml.ws.model.RuntimeModelerException:
2  runtime modeler error: Wrapper class cat.xtec.ioc.service.jaxws.GetAllShows is
       not found. Have you run APT to generate them?
```

Generar artefactes

Quan heu fet el desplegament amb NetBeans és el contenidor del servidor d'aplicacions qui s'encarrega de generar automàticament aquests artefactes, per això no ho heu hagut de fer manualment.

El problema és que ens cal executar una utilitat del JDK anomenada `wsgen` que generarà tots els artefactes portables que necessita el servei web per ser publicat i consumit.

Per fer-ho obriu un *command-prompt*, situeu-vos al directori *\target\classes* on tingueu el projecte "Resentioc" i executeu la següent línia (cal que tingueu el directori *<JDK_HOME>/bin* al *classpath* per tal que us trobi l'executable `wsgen`):

Amb Windows:

```
1  wsgen –verbose  –keep –cp . –s ..\..\src\main\java cat.xtec.ioc.service.impl.
       TicketServiceEndpointImpl
```

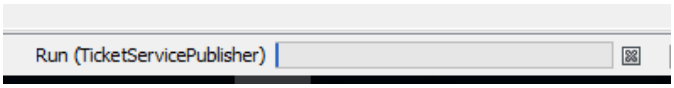
Amb Linux:

```
1  wsgen –verbose  –keep –cp . –s ../../src/main/java cat.xtec.ioc.service.impl.
       TicketServiceEndpointImpl
```

Aquesta instrucció us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per fer la publicació des de l'aplicació Java i els col·locarà al paquet `cat.xtec.ioc.service.jaxws` del projecte "Resentioc".

Ara sí que podeu publicar el servei web de reserva d'entrades executant el *Publisher*; si tot va bé, veureu que a la part inferior esquerra de NetBeans (vegeu la figura 1.7) us apareix una finestra indicant que el servei web està corrent a l'URL que heu especificat.

Figura 1.7. Servei web publicat a NetBeans



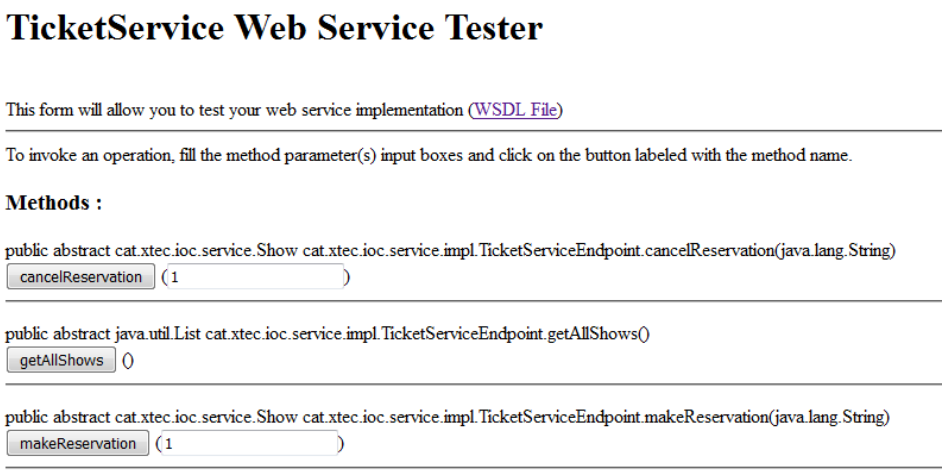
Podeu provar que el desplegament des de l'aplicació Java ha anat bé consultant el document WSDL generat en el següent enllaç: localhost:9999/publisher/TicketService?wsdl.

1.1.4 Provant el servei web

Si tot ha anat bé ja teniu el servei web desplegat i a punt per provar-lo. Però com ho fem? Com el provem? Fixeu-vos que fins al moment no hem codificat cap client que faci peticions al servei web desenvolupat; com podem, doncs, provar-lo?

Per tal de provar el servei web que heu desenvolupat, desplegueu l'aplicació al servidor d'aplicacions Glassfish i accediu al següent URL: localhost:8080/resentioc/TicketService?Tester, on veureu una pàgina (vegeu la figura 1.8) amb un botó per a cada una de les operacions que conté el servei web i un enllaç al fitxer WSDL de definició de servei que s'ha generat:

Figura 1.8. Pàgina de prova del servei web



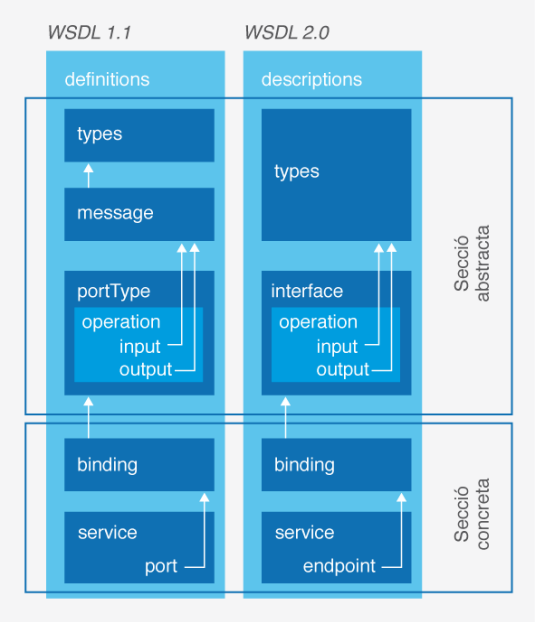
Quan es desplega un servei web SOAP es genera un fitxer WSDL de definició del servei web. El fitxer WSDL està forma per elements XML que **descriuen completament el servei web** i com s'ha de consumir. El podeu consultar en el següent URL: localhost:8080/resentioc/TicketService?WSDL.

```
1 <definitions targetNamespace="http://impl.service.ioc.xtec.cat/" name="
      TicketService">
```

```
    <import namespace="http://service.ioc.xtec.cat/" location="http://
        localhost:8080/resentioc/TicketService?wsdl=1"/>
    <binding name="TicketServiceEndpointImplPortBinding" type="
        ns1:TicketServiceEndpoint"></binding>
    <service name="TicketService">
        <port name="TicketServiceEndpointImplPort" binding="
            tns:TicketServiceEndpointImplPortBinding">
            <soap:address location="http://localhost:8080/resentioc/
                TicketService"/>
        </port>
    </service>
</definitions>
```

El fitxer WSDL és un document XML que té una secció abstracta per definir els ports, els missatges i els tipus de dades de les operacions i una part concreta que defineix la instància concreta on hi ha aquestes operacions (vegeu la figura 1.9). Aquesta estructura permet reutilitzar la part abstracta del document.

FIGURA 1.9. Estructura general d'un fitxer WSDL (versions 1.1 i 2.0)



A la part abstracta hi tenim els següents elements:

- `types`
- `message`
- `portType`

Els tipus de dades, tant d'entrada com de sortida, de les operacions es defineixen amb XSD a l'etiqueta `<types>`. En el nostre cas veiem que ho tenim definit amb un import de localhost:8080/resentioc/TicketService?wsdl=1; si accediu a aquest fitxer veureu que referencia un document d'esquema definit amb XSD:

```
<definitions targetNamespace="http://service.ioc.xtec.cat/">
    <types>
        <xsd:schema>
```

```
            <xsd:import namespace="http://service.ioc.xtec.cat/" schemaLocation
                ="http://localhost:8080/resentioc/TicketService?xsd=1"/>
        </xsd:schema>
    </types>
    <message name="makeReservation"></message>
    <message name="makeReservationResponse"></message>
    <message name="getAllShows"></message>
    <message name="getAllShowsResponse"></message>
    <message name="cancelReservation"></message>
    <message name="cancelReservationResponse"></message>
    <portType name="TicketServiceEndpoint"></portType>
</definitions>
```

I en aquest document XSD localhost:8080/resentioc/TicketService?xsd=1 és on trobem la definició de les operacions i els tipus de dades, tant d'entrada com de sortida, que fan servir les operacions:

```
<xs:schema version="1.0" targetNamespace="http://service.ioc.xtec.cat/">
    <xs:element name="cancelReservation" type="tns:cancelReservation"/>
    <xs:element name="cancelReservationResponse" type="
        tns:cancelReservationResponse"/>
    <xs:element name="getAllShows" type="tns:getAllShows"/>
    <xs:element name="getAllShowsResponse" type="tns:getAllShowsResponse"/>
    <xs:element name="makeReservation" type="tns:makeReservation"/>
    <xs:element name="makeReservationResponse" type="
        tns:makeReservationResponse"/>
    <xs:complexType name="cancelReservation">
        <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="cancelReservationResponse">
        <xs:sequence>
            <xs:element name="return" type="tns:show" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="show">
        <xs:sequence>
            <xs:element name="availableTickets" type="xs:int" minOccurs="0"/>
            <xs:element name="id" type="xs:string" minOccurs="0"/>
            <xs:element name="location" type="xs:string" minOccurs="0"/>
            <xs:element name="name" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="makeReservation">
        <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="makeReservationResponse">
        <xs:sequence>
            <xs:element name="return" type="tns:show" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="getAllShows">
        <xs:sequence/>
    </xs:complexType>
    <xs:complexType name="getAllShowsResponse">
        <xs:sequence>
            <xs:element name="return" type="tns:show" minOccurs="0" maxOccurs="
                unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Aconseguir interoperativitat

El *runtime* de JAX-WS serà l'encarregat de fer les transformacions dels missatges SOAP a crides a l'API Java, fent els mapatges de tipus adient per aconseguir, per exemple, tornar als clients objectes Java complexos de tipus *Show*. Aquesta és la clau per aconseguir interoperativitat, ja que un client que no sigui Java i que tingui un *runtime* que permeti fer les transformacions pertinents també podrà cridar el servei de gestió de reserves que heu creat.

Per exemple, si us hi fixeu, defineix que les peticions a l'operació `cancelReservation` reben un paràmetre de tipus *string* i a les respostes

es torna un tipus de dades complex anomenat *show* que està format per un *int* i tres *string* (`availableTickets`, `id`, `location` i `name`)

Després de la definició dels tipus de dades trobem els elements `<message>` amb la definició dels missatges que es poden intercanviar les operacions del servei web, amb una entrada per a la petició i una altra per a la resposta:

```
<definitions targetNamespace="http://service.ioc.xtec.cat/">
    <types></types>
    <message name="makeReservation"></message>
    <message name="makeReservationResponse"></message>
    <message name="getAllShows"></message>
    <message name="getAllShowsResponse"></message>
    <message name="cancelReservation"></message>
    <message name="cancelReservationResponse"></message>
    <portType name="TicketServiceEndpoint"></portType>
</definitions>
```

I després, els elements `<portType>` ens defineixen les operacions que es poden fer al servei web i els seus paràmetres:

```
<definitions targetNamespace="http://service.ioc.xtec.cat/">
    <types></types>
    <message name="makeReservation"></message>
    <message name="makeReservationResponse"></message>
    <message name="getAllShows"></message>
    <message name="getAllShowsResponse"></message>
    <message name="cancelReservation"></message>
    <message name="cancelReservationResponse"></message>
    <portType name="TicketServiceEndpoint">
        <operation name="makeReservation">
            <input ns1:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/makeReservationRequest" message="
                tns:makeReservation"/>
            <output ns2:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/makeReservationResponse" message="
                tns:makeReservationResponse"/>
        </operation>
        <operation name="getAllShows">
            <input ns3:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/getAllShowsRequest" message="
                tns:getAllShows"/>
            <output ns4:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/getAllShowsResponse" message="
                tns:getAllShowsResponse"/>
        </operation>
        <operation name="cancelReservation">
            <input ns5:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/cancelReservationRequest" message="
                tns:cancelReservation"/>
            <output ns6:Action="http://service.ioc.xtec.cat/
                TicketServiceEndpoint/cancelReservationResponse" message="
                tns:cancelReservationResponse"/>
        </operation>
    </portType>
</definitions>
```

Vegem, per exemple, que l'operació `cancelReservation` rep com a paràmetre d'entrada un `tns:cancelReservation` i que retorna un `tns:cancelReservationResponse`. Aquests tipus de dades han estat definides completament en el document XSD (vegeu la figura 1.10).

FIGURA 1.10. Tipus de dades de cancelReservation

A la part concreta hi tenim els següents elements:

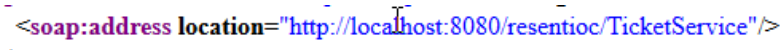
- `binding`
- `service`

L'element `<binding>` (vegeu la figura 1.11) defineix el protocol i el format en què es poden fer les operacions; en el nostre cas, es faran per **HTTP** i amb un estil de *Document*.

FIGURA 1.11. Element binding

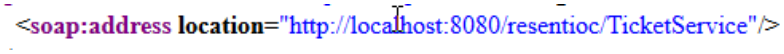
L'element `<service>` (vegeu la figura 1.12), per la seva part, defineix el lloc físic on hi ha el servei web (adreça URL) mitjançant una col·lecció de punts de connexió `<binding>`. En el nostre cas, el servei web es cridarà amb el nom `TicketService` i està desplegat a l'adreça localhost:8080/resentioc/TicketService.

FIGURA 1.12. Element service



Si voleu provar alguna de les operacions que proporciona el servei web, per exemple, `getAllShows`, polseu el botó corresponent (vegeu la figura 1.13) i es fa una petició a l'operació del servei web que torna la llista de tots els concerts amb les entrades disponibles.

FIGURA 1.13. Botó per provar l'operació getAllShows



Aquesta petició genera aquest missatge SOAP de petició:

```
<?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:getAllShows xmlns:ns2="http://service.ioc.xtec.cat/"/>
    </S:Body>
</S:Envelope>
```

I aquest missatge SOAP de resposta:

```
<?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:getAllShowsResponse xmlns:ns2="http://service.ioc.xtec.cat/">
            <return>
                <availableTickets>5</availableTickets>
                <id>1</id>
                <location>Palau Sant Jordi</location>
                <name>U2 360 Tour</name>
            </return>
            <return>
                <availableTickets>3</availableTickets>
                <id>2</id>
                <location>Palau de la musica</location>
                <name>Carmina Burana</name>
            </return>
        </ns2:getAllShowsResponse>
    </S:Body>
</S:Envelope>
```

1.2 Fent servir la gestió de reserva de places a concerts des d'una aplicació Java 'stand-alone'

Hi ha dues maneres de cridar serveis web des de Java amb l'API JAX-WS:

- Creant un *stub* estàtic.
- Fent servir una interfície d'invocació dinàmica.

Anem a crear una aplicació Java *stand-alone* que consumeixi el servei web de gestió de reserves de places a concerts, i ho farem desenvolupant un **client Java *stand-alone*** que accedirà al servei web amb JAX-WS **utilitzant *stubs* estàtics**.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Els passos generals per fer un client que faci servir *stubs* estàtics són els següents:

1. Codificar la classe que farà de client.
2. Generar els artefactes necessaris per poder consumir el servei web des d'aquest client amb la utilitat `wsimport`.
3. Compilar i executar el client.

Descarregueu el codi del projecte "Resentclientioc" en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Creeu la classe Java que farà de client al paquet `cat.xtec.ioc.client` i l'anomeneu `TicketServiceClient`.

Servidor i WSDL

Assegureu-vos que teniu el servidor d'aplicacions arrencat i el document WSDL de descripció del servei web de gestió de reserves de places a concerts accessible a l'URL que especifiqueu.

Genereu els artefactes per tal de poder consumir el servei web amb la utilitat `wsimport`. A `wsimport` cal especificar l'URL del document WSDL de descripció del servei web de gestió de reserves i on voleu que us generi els artefactes. Per fer-ho, obriu un *command-prompt*, situeu-vos al directori *\target\classes* on tingueu el projecte "Resentclientioc" i executeu la següent línia (cal que tingueu el directori *<JDK_HOME>/bin* al *classpath* per tal que us trobi l'executable `wsimport` i que hagueu fet *Clean and Build* del projecte):

Amb Windows:

```
wsimport -s ..\..\src\main\java -p cat.xtec.ioc.service.client.jaxws http://
    localhost:8080/resentioc/TicketService?wsdl
```

Amb Linux:

```
wsimport -s ../../src/main\java -p cat.xtec.ioc.service.client.jaxws http://
    localhost:8080/resentioc/TicketService?wsdl
```

Això us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per cridar el servei web des de l'aplicació Java i els col·locarà al paquet *cat.xtec.ioc.service.client.jaxws* del projecte "Resentclientioc".

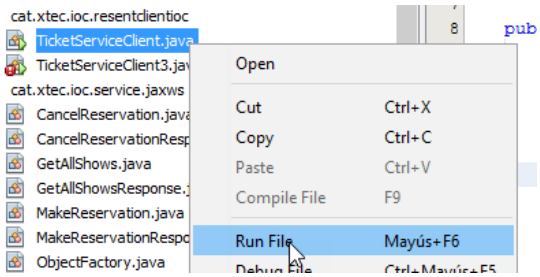
Ara ja podem codificar la classe Java `TicketServiceClient` creada utilitzant els artefactes generats per `wsimport`. A les classes Java generades n'hi haurà una que s'anomenarà `TicketService` i que heretarà de `Service`; cal que instancieu aquesta classe i, a partir d'ella, obtingueu l'*stub* que us permetrà cridar les operacions del servei web.

El codi de la classe `TicketServiceClient` amb una crida a l'operació que mostra tots els concerts amb el nombre d'entrades disponibles és el següent:

```
package cat.xtec.ioc.client;

import cat.xtec.ioc.service.client.jaxws.Show;
import cat.xtec.ioc.service.client.jaxws.TicketService;
import cat.xtec.ioc.service.client.jaxws.TicketServiceEndpoint;
import java.util.List;

public class TicketServiceClient {

    public static void main(String[] args) {
        TicketService service = new TicketService();
        TicketServiceEndpoint port = service.getTicketServiceEndpointImplPort()
            ;
        List<Show> list = port.getAllShows();
        printAllShows(list);
    }

    public static void printAllShows(List<Show> shows) {
        shows.stream().forEach((show) -> {
            System.out.println("Show [id=" + show.getId() + ", name=" + show.
                getName() + ", available tickets=" + show.getAvailableTickets
                () + "]");
        });
    }
}
```

Ara ja podem executar el client; per fer-ho, poseu-vos damunt de la classe `TicketServiceClient` i feu *Run File* a NetBeans (vegeu la figura 1.14).

FIGURA 1.14. Execució del client Java a NetBeans



I obtindreu per consola un llistat dels concerts i de les entrades disponibles per a cada un d'ells:

```
Show [id=1, name=U2 360 Tour, available tickets=5]
Show [id=2, name=Carmina Burana, available tickets=3]
```

1.3 Fent servir la gestió de reserva de places a concerts des d'una aplicació web

Anem a crear una aplicació web que consumeixi el servei web de gestió de reserves de places a concerts.

Per fer-ho crearem una aplicació web molt senzilla que farà servir el servei web SOAP de gestió de reserva de places a concerts.

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Per desplegar el servei web de gestió de reserva de places a concerts al servidor, descarregueu el codi del projecte resentioc.zip en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Per implementar l'aplicació web que accedirà com a client al servei web, descarregueu el codi del projecte resentwebclientioc.zip en l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

1.3.1 Creació i configuració inicial del projecte

Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

El primer que cal fer és desplegar el servei web de gestió de reserva de places a concerts al servidor.

Comproveu que el servei web està desplegat correctament accedint a l'URL localhost:8080/resentioc/TicketService?WSDL amb un navegador.

Un cop desplegat el servei web ens cal un altre projecte, que serà l'aplicació web que accedirà com a client al servei web.

No entrarem en detalls, però el projecte que tenim com a punt de partida és una senzilla aplicació web Spring MVC que ha de mostrar un llistat dels concerts disponibles i, per a cada concert, ha de proporcionar una acció que permeti reservar una entrada i una acció que permeti cancel·lar una reserva. Per simplicitat, a l'exemple no seguirem estrictament una arquitectura per capes on els serveis es cridin des de la capa de serveis. Si ho féssim així crearíem un servei propi en

aquesta capa i aquest servei seria l'encarregat de cridar el servei web SOAP de reserva de places a concerts.

L'aplicació web utilitzarà el servei web SOAP de gestió de reserva de places a concerts per proporcionar aquestes funcionalitats.

1.3.2 Creació i prova del client web

Referències a l'"endpoint' del servei

A l'exemple creem directament una instància de l'*endpoint* del servei (invocació programàtica); també es pot utilitzar l'anotació `@WebServiceRef` per tal que el contenidor injecti automàticament la instància del *proxy* (o *stub*) del client.

Un cop tingueu el projecte "Resentwebclientioc" carregat a NetBeans, el primer que farem serà llistar tots els concerts disponibles. Per fer-ho heu d'accedir a la classe `ShowController` del paquet `cat.xtec.ioc.controller` que fa de controlador i crear la referència al servei web SOAP que ens permetrà obtenir la llista de concerts:

```
1 private final TicketService showsWebService=new TicketService();
```

Si ho feu, veureu que NetBeans indica que la classe no compila. Això és degut al fet que ens falta generar els artefactes de client per tal de poder consumir el servei web amb la utilitat `wsimport`. A `wsimport` cal especificar l'URL del document WSDL de descripció del servei web de gestió de reserves i on voleu que us generi els artefactes.

Per fer-ho, obriu un *command-prompt*, situeu-vos al directori *\target\classes* on tingueu el projecte "Resentwebclientioc" i executeu la següent línia (cal que tingueu el directori *<JDK_HOME>/bin* al *classpath* per tal que us trobi l'executable `wsimport` i que hagueu fet *Clean and Build* del projecte):

Amb Windows:

```
1 wsimport -s ..\..\src\main\java -p cat.xtec.ioc.service.client.jaxws http://
      localhost:8080/resentioc/TicketService?wsdl
```

Amb Linux:

```
1 wsimport -s ../../src/main\java -p cat.xtec.ioc.service.client.jaxws http://
      localhost:8080/resentioc/TicketService?wsdl
```

Això us generarà un conjunt de fitxers .java amb tots els artefactes portables necessaris per cridar el servei web des de l'aplicació i els col·locarà al paquet `cat.xtec.ioc.service.client.jaxws` del projecte "Resentwebclientioc". Un cop fet això, i l'import corresponent al controlador, la classe ja us compilarà.

A l'exemple hem vist que sempre ens cal generar els artefactes de client per poder cridar el servei web SOAP, i hem vist com fer-ho amb la utilitat del JDK `wsimport`. Si feu servir Maven al projecte també ho podeu fer amb un *plugin* per a JAX-WS que configura un *goal* de Maven anomenat `wsimport` i que s'executa a la fase `generate-sources` de generació del codi font.

Objectiu de l'exemple

Tingueu en compte que l'objectiu de l'exemple és veure com podem accedir al servei web SOAP des d'una aplicació web i no la construcció de l'aplicació web en si. Per això hi haurà molts detalls propis de l'anatomia de l'aplicació que elidirem o explicarem amb poc detall.

Ja tenim la referència al servei web SOAP; ara crearem l'acció `MVC` que torni la llista de concerts. Per fer-ho, creeu un mètode anomenat `shows` amb el següent codi:

```
@RequestMapping(value = "/shows", method = RequestMethod.GET)
    public ModelAndView shows(HttpServletRequest request, HttpServletResponse
         response)
             throws ServletException, IOException {
         ModelAndView modelview = new ModelAndView("shows");
         modelview.getModelMap().addAttribute("shows", showsWebService.
              getTicketServiceEndpointImplPort().getAllShows());
         return modelview;
    }
```

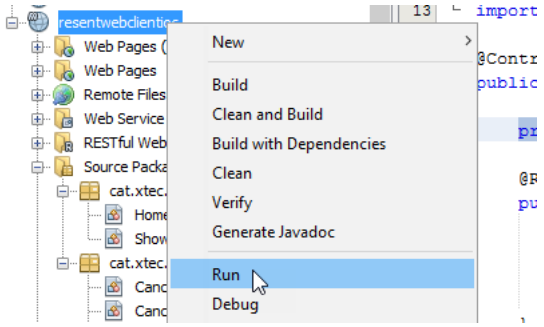
La part important d'aquest mètode és la crida al servei web SOAP:

```
showsWebService.getTicketServiceEndpointImplPort().getAllShows();
```

Obtenim una referència al servei web i invoquem el mètode `getAllShows`, que ens tornarà la llista de concerts.

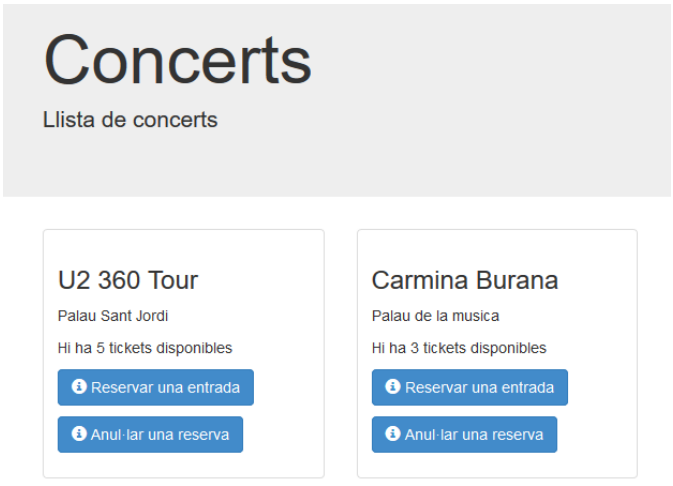
Per provar-ho cal desplegar l'aplicació "Resentwebclientioc" a Glassfish. Per fer-ho, feu *Run* a NetBeans i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte (vegeu la figura 1.15).

Figura 1.15. 'Run' a NetBeans



Si tot ha anat bé i accediu a l'URL localhost:8080/resentwebclientioc/shows veureu la pàgina que mostra el llistat de concerts disponibles (vegeu la figura 1.16).

Figura 1.16. Llistat de concerts



El procés per crear la funcionalitat que permeti fer una reserva i cancel·lar-la és el mateix: us cal crear l'acció corresponent al controlador i fer ús dels mètodes que proporciona el servei web SOAP de gestió de reserva de places a concerts per implementar-la.

El codi per a aquestes dues accions és el següent:

```
@RequestMapping("/makeReservation")
public ModelAndView makeReservation(@RequestParam("id") String id,
    HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    showsWebService.getTicketServiceEndpointImplPort().makeReservation(id);
    ModelAndView modelview = new ModelAndView("shows");
    modelview.getModelMap().addAttribute("shows", showsWebService.
        getTicketServiceEndpointImplPort().getAllShows());
    return modelview;
}

@RequestMapping("/cancelReservation")
public ModelAndView cancelReservation(@RequestParam("id") String id,
    HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    showsWebService.getTicketServiceEndpointImplPort().cancelReservation(id);
    ModelAndView modelview = new ModelAndView("shows");
    modelview.getModelMap().addAttribute("shows", showsWebService.
        getTicketServiceEndpointImplPort().getAllShows());
    return modelview;
}
```

Si desplegueu la nova versió de l'aplicació fent *Run* a NetBeans ja podreu provar les funcionalitat que permeten fer una reserva i cancel·lar-la.

Si ho proveu i aneu fent reserves per al concert del grup **U2** veureu que, quan ja no quedin entrades disponibles, el servidor torna un error HTTP 500 (vegeu la figura 1.17).

Figura 1.17. Error 500 - No more tickets available!

Això és degut al fet que des del codi del servei web, quan volem fer una reserva per a un concert que no té localitats disponibles, s'està llençant una excepció:

```
public void makeTicketReservation() {
    if(this.availableTickets > 0) {
        this.availableTickets--;
    } else {
        throw new IllegalArgumentException("No more tickets available!");
    }
}
```

Si mirem el *log* del servidor Glassfish veurem aquesta excepció:

```
Grave:   No more tickets available!
java.lang.IllegalArgumentException: No more tickets available!
  at cat.xtec.ioc.domain.Show.makeTicketReservation(Show.java:64)
  at cat.xtec.ioc.domain.repository.impl.InMemoryShowRepository.makeReservation
      (InMemoryShowRepository.java:33)
  at cat.xtec.ioc.service.impl.TicketServiceEndpointImpl.makeReservation(
      TicketServiceEndpointImpl.java:24)
```

JAX-WS converteix **automàticament** les excepcions de Java en una SOAPFault en format XML que viatjarà al missatge SOAP de resposta.

> **SOAPFault** és el mecanisme que proporciona SOAP per indicar que el servei web cridat ha tingut algun problema.

1.4 Què s'ha après?

Heu vist les bases per al desenvolupament dels serveis web SOAP amb Java EE 7 i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web SOAP amb Java EE.
- A desenvolupar, desplegar i provar un servei web SOAP amb Java EE.
- A desenvolupar i provar un client Java que consulti un servei web SOAP.
- A desenvolupar i provar una aplicació web que consulti un servei web SOAP.

Per aprofundir en aquests conceptes i veure com us pot ajudar NetBeans en la creació i el consum de serveis web SOAP us recomanem que feu les activitats associades a aquest apartat.

2. Serveis web RESTful amb Java EE7. Escrivint serveis web

Explicarem, mitjançant exemples, els conceptes més rellevants dels serveis web RESTful (de l'anglès REpresentational State Transfer), aprendreu a crear-ne i a fer-ne el desplegament mitjançant exemples.

REST no és un protocol, sinó un conjunt de regles i principis que permeten desenvolupar serveis web fent servir HTTP com a protocol de comunicacions entre el client i el servei web, i es basa a definir **accions sobre recursos** mitjançant l'ús dels mètodes `GET`, `POST`, `PUT` i `DELETE` inherents d'HTTP.

Per a REST, qualsevol cosa que es pugui identificar amb un URI (de l'anglès Uniform Resource Identifier) es considera un recurs, i, per tant, es pot manipular mitjançant accions (també anomenades verbs) especificades a la capçalera HTTP de les peticions seguint el següent conjunt de regles i principis que regeixen REST:

- `POST`: crea un recurs nou.
- `GET`: consulta el recurs i n'obté la representació.
- `DELETE`: esborra un recurs.
- `PUT`: modifica un recurs.
- `HEAD`: obté metainformació del recurs.

REST es basa en l'ús d'estàndards oberts en totes les seves parts; així, fa servir URI per a la localització de recursos, HTTP com a protocol de transport, els verbs HTTP per especificar les accions sobre els recursos i els tipus MIME per a la representació dels recursos (XML, JSON, XHTML, HTML, PDF, GIF, JPG, PNG, etc.).

Format JSON

JSON (de l'anglès Java Script Object Notation) és un format lleuger d'intercanvi de dades. És fàcil de llegir i escriure per als éssers humans i, per a les màquines, d'analitzar i generar. Això el fa ideal per representar els recursos en arquitectures REST. Un dels principals problemes dels serveis web basats en SOAP és la mida dels missatges d'intercanvi; l'ús de JSON permet minimitzar la informació a enviar.

En una comparació ràpida entre REST i SOAP veiem que REST fa servir gairebé sempre HTTP com a mecanisme de comunicació i XML o JSON per intercanviar dades. Un servei REST no té estat i cada URI representa un recurs sobre el qual s'opera amb verbs HTTP. Un servei web REST no requereix de missatges SOAP/XML ni de descripcions del servei amb documents WDSL de definició de servei.

Als serveis web SOAP tota la infraestructura es basa en XML i les operacions cal que les defineixi el desenvolupador del servei; són força més complexos, però proporcionen certes capacitats a nivell de seguretat i transaccionalitat que, a vegades, els fan l'única alternativa viable, sobretot en aplicacions empresarials. Tot això fa que REST sigui molt més lleuger i fàcil d'utilitzar que SOAP i, per a determinades arquitectures, sigui una millor opció.

Ens centrarem en la creació i el desplegament de serveis web RESTful amb Java EE 7 i ho farem mitjançant l'API **JAX-RS**. Noteu que per escriure un servei web RESTful tan sols us caldria un client i un servidor que es puguin comunicar per HTTP, però hauríeu de fer a mà tota la configuració, el parseig de les peticions segons el verb HTTP emprat, el mapatge entre el format de dades de la petició i els objectes Java que representen el domini i enviar les respostes amb el tipus MIME que s'especifiqui a la capçalera de la petició; JAX-RS us estalviarà tota aquesta feina proporcionant-vos un petit conjunt d'anotacions que us permetran desenvolupar còmodament serveis web RESTful.

JAX-RS (de l'anglès Java API for RESTful Web Services) és l'API que inclou l'especificació de Java EE 7 per crear i consumir serveis web basats en REST.

2.1 Un servei web RESTful que contesta "Hello World!!!"

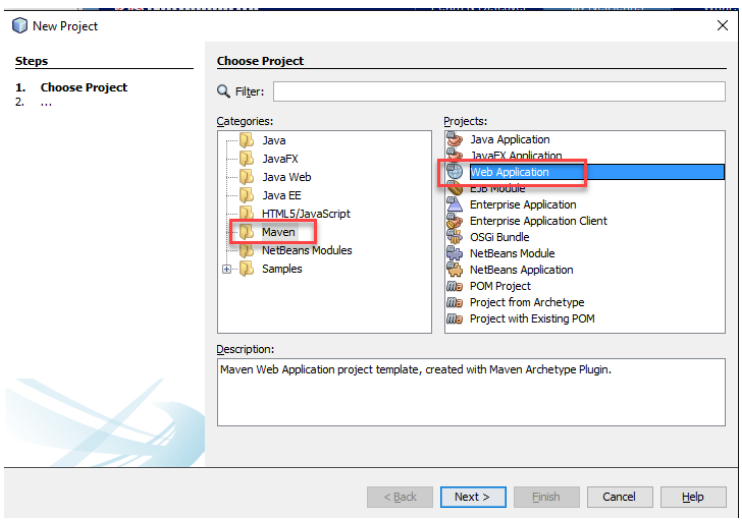
Veurem els diferents conceptes bàsics dels serveis web RESTful amb el típic exemple que sempre trobeu als manuals: farem un "*Hello World!!!*" i el publicarem com a servei web RESTful utilitzant l'API que JAX-RS que proporciona Java EE 7 per fer-ho.

2.1.1 Creació i configuració inicial del projecte

Hem triat per a l'exemple un projecte web amb Maven, però és perfectament vàlid fer-ho amb qualsevol altre tipus de projecte web.

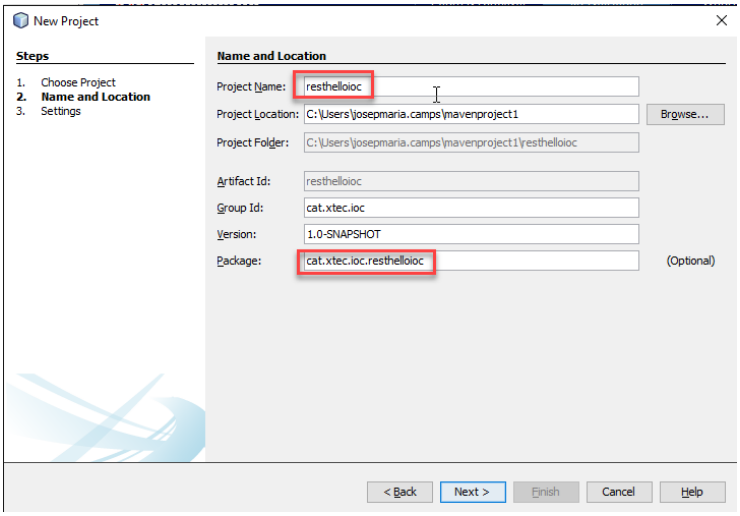
Com que es tracta d'un exemple molt senzill, no ens cal cap projecte de partida; simplement, creeu a NetBeans un nou projecte Maven de tipus *Web Application*. Per fer-ho, feu *File / New Project* i us apareixerà l'assistent de creació de projectes. A l'assistent, seleccioneu *Maven* i *Web Application*, tal com es veu en la figura 2.1.

Figura 2.1. Creació de projectes a NetBeans



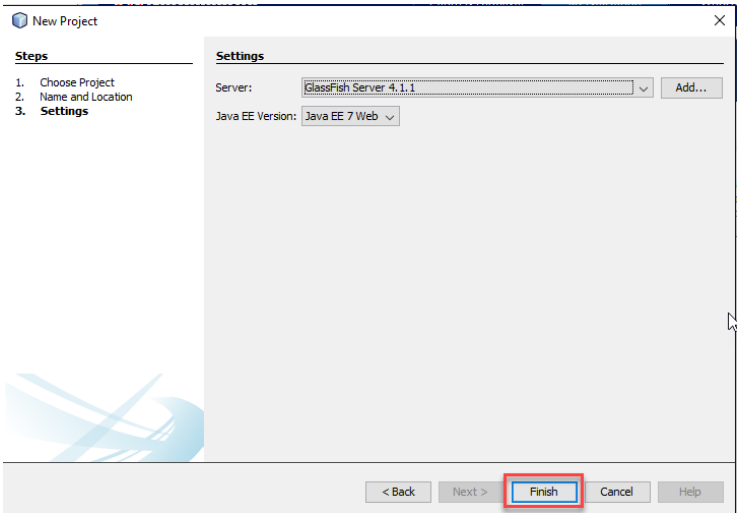
En la següent pantalla (vegeu la figura 2.2) triarem el nom del projecte i el paquet per defecte on anirà el codi font. El podeu anomenar "Resthelloioc" i posar el codi font a `cat.xtec.ioc.resthelloioc`.

FIGURA 2.2. Nom del nou projecte



En la següent pantalla (vegeu la figura 2.3) de l'assistent deixeu els valors per defecte i polseu *Finish*.

FIGURA 2.3. Configuració del nou projecte



2.1.2 Creació del servei web RESTful

Ja tenim el projecte que ens servirà de base creat. Ara cal que codifiqueu el servei web RESTful que ens ha de tornar la salutació "*Hello World!!!*"; per fer-ho ens cal decidir primer dues coses: amb quin dels verbs HTTP ha de respondre el servei web i quin URI farem servir per cridar-lo.

Aquest servei web el que ha de fer és **obtenir una representació del recurs** en format HTML; per tant, haurà de respondre al verb `GET`.

La decisió sobre quin URI utilitzar és força arbitrària en aquest cas; utilitzarem, per exemple, */hello*.

Per tant, a les peticions d'aquest tipus:

```
GET http://localhost:8080/resthelloioc/hello
```

Ha de respondre amb:

```
Hello World!!!
```

Un cop decidit el funcionament del nostre servei, el següent que us cal fer és codificar una classe Java que implementi la funcionalitat demanada.

> Els serveis web RESTful són **POJO** (de l'anglès Plain Old Java Object) que tenen almenys un mètode anotat amb l'anotació `@Path`.

Creeu una classe Java que implementarà aquesta funcionalitat, anomeneu-la `HelloWorldService` i la creeu al paquet `cat.xtec.ioc.resthelloioc.service`.

Anoteu la classe amb l'anotació `@Path(''/hello'')` i hi creeu un mètode anomenat, per exemple, `sayHello`, i l'anoteu amb `@GET` i `@Produces(''text/html'')`.

```
package cat.xtec.ioc.resthelloioc.service;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/hello")
public class HelloWorldService {
    @GET
    @Produces("text/html")
    public String sayHello() {
        return "Hello World!!!";
    }
}
```

Fixeu-vos que:

- Hem anotat la classe amb `@Path(''/hello'')` per indicar que respondrà a les peticions que arribin a l'URL localhost:8080/resthelloworld/hello.
- Hem anotat el mètode `sayHello` amb *@GET* per indicar que respondrà a les peticions HTTP que es facin mitjançant el verb `GET`.
- Hem anotat el mètode `sayHello` amb `@Produces(''text/html'')` per indicar que respondrà a les peticions HTTP en les quals a la capçalera s'indiqui "text/html" com a tipus MIME, i ho farà produint HTML com a representació del recurs.

I aquesta és tota la feina que ens cal fer per implementar el servei web, JAX-RS farà la resta. Ara tan sols ens manca fer-ne el desplegament al servidor d'aplicacions i provar-lo.

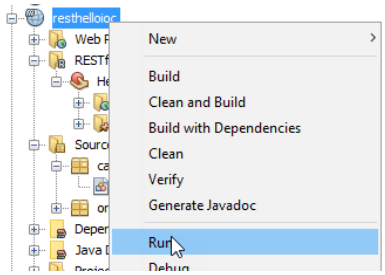
2.1.3 Desplegament del servei web RESTful

Un cop creat el servei web cal que el desplegueu per tal de fer-lo accessible als clients i poder-lo provar.

El procés de desplegament del servei web es fa desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.

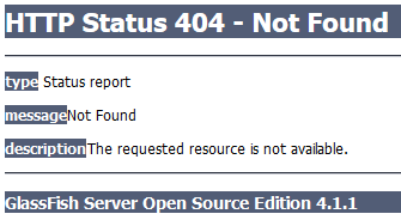
El desplegament del servei web com a part de l'aplicació Java EE que el conté és molt senzill: simplement cal que feu *Clean and Build* i després *Run* a NetBeans (vegeu la figura 2.4) i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte.

FIGURA 2.4. 'Run' a NetBeans



Si ho feu i el proveu accedint a l'URL que hem configurat: localhost:8080/resthelloioc/hello veureu que el desplegament falla (vegeu la figura 2.5).

FIGURA 2.5. Error 404 en desplegar



Això és degut al fet que ens cal indicar quines són les classes que ha de tractar com a recursos REST. Això es pot fer de moltes maneres, i una d'elles és crear una classe de configuració anomenada `ApplicationConfig` al paquet `cat.xtec.ioc.resthelloioc.service` amb el següent codi:

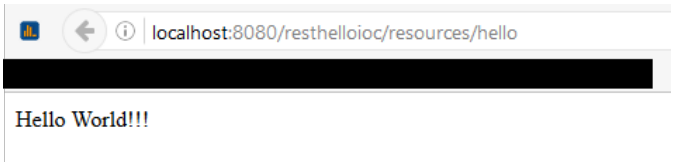
Si la classe `ApplicationConfig` no es crea al paquet on hi ha les classes que formen els serveis web ens cladrà afegir cada una de les classes a una col·leccio que té la classe `Application` de la qual hereta.

```
1  package cat.xtec.ioc.resthelloioc.service;
2
3  import javax.ws.rs.core.Application;
4
5  @javax.ws.rs.ApplicationPath("rest")
6  public class ApplicationConfig extends Application {
7
8  }
```

On s'indica que cal afegir "/rest" (o el que vulgueu posar) abans del nom del servei web per cridar-lo.

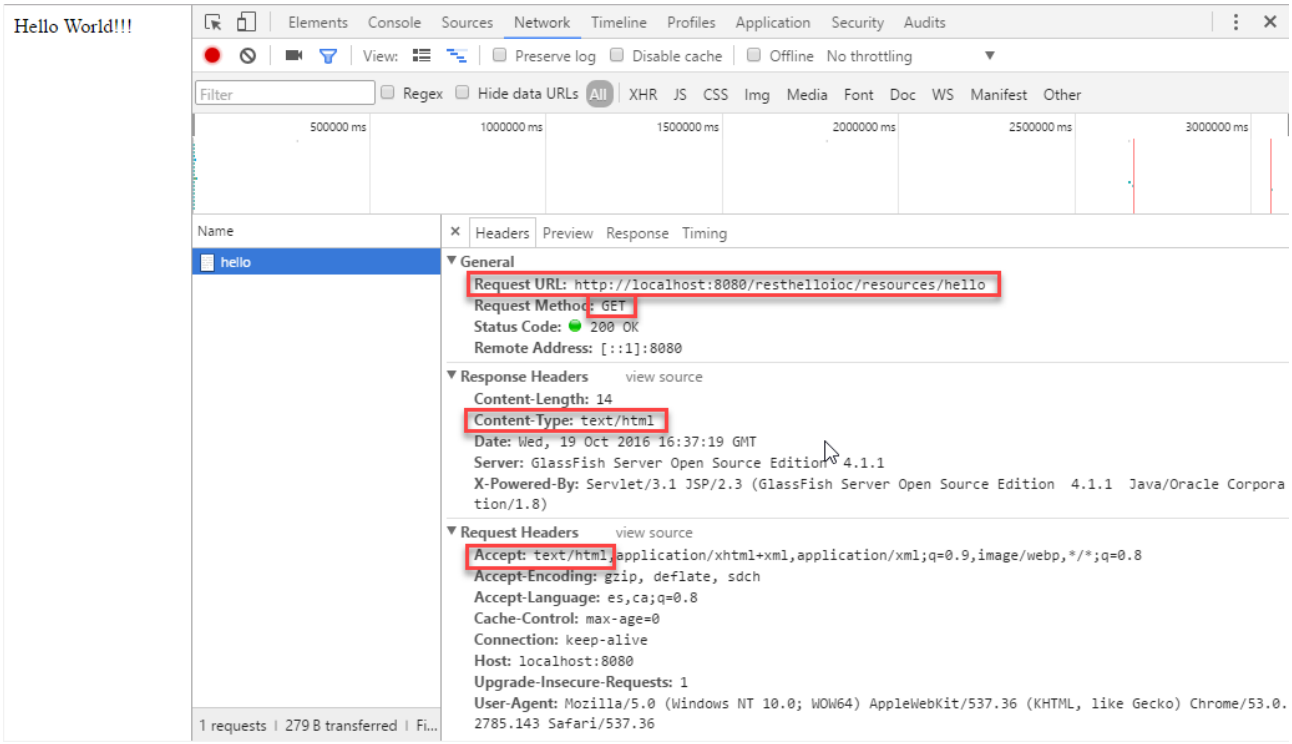
Si proveu ara accedint a l'URL localhost:8080/resthelloioc/rest/hello veureu que el servei web us torna la salutació (vegeu la figura 2.6).

FIGURA 2.6. Servei web desplegat



Si utilitzeu les eines de *debug* de Google Chrome, per exemple, podreu veure la petició HTTP feta i la resposta rebuda (vegeu la figura 2.7).

FIGURA 2.7. Petició i resposta HTTP



2.2 El servei web de dades de llibres. Operacions CRUD

CRUD és l'acrònim en anglès per a les operacions de creació (Create), lectura (Read), actualització (Update) i esborrat (Delete).

Veurem els conceptes referents a la creació de serveis web RETSful amb Java EE 7 desenvolupant un servei web RESTful que permeti treballar sobre un catàleg de llibres. Hi farem les operacions típiques CRUD i dues operacions de cerca: la que ens tornarà tots els llibres i una que ens permetrà fer cerques per títol.

Concretament, el servei web que farem tindrà les següents operacions:

- Consulta d'un llibre mitjançant l'ISBN (operació `Read` CRUD).
- Creació d'un llibre al catàleg (operació `Create` CRUD).

- Actualització de les dades d'un llibre (operació `Update` CRUD).
- Esborrar un llibre del catàleg (operació `Delete` CRUD).
- Llistar tots els llibres del catàleg.
- Cercar llibres per títol.

Per fer-ho farem servir una aplicació web ja desenvolupada que segueixi una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió del catàleg com a servei web RESTful. La representació que farem servir per als llibres en tot l'exemple serà JSON.

2.2.1 Creació i configuració inicial del projecte

L'aplicació de la qual partirem s'anomena "Restbooksioc" i ens servirà per veure com podem exposar algunes funcionalitats de la capa de serveis d'una aplicació mitjançant serveis web RESTful. Es tracta d'un projecte Spring MVC senzill que segueix una arquitectura típica per capes per tal d'aconseguir una alta reusabilitat, un baix acoblament i una alta cohesió en l'aplicació.

El projecte consta de quatre capes:

- capa de presentació
- capa de domini
- capa de serveis
- capa de persistència

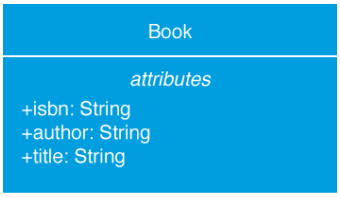
En l'exemple no farem servir la capa de presentació, ja que l'objectiu no és proporcionar una interfície d'usuari a l'aplicació sinó publicar els mètodes de negoci com a serveis web.

El model de domini ja el teniu implementat i és molt senzill, només hi ha una entitat `Book` que representa els llibres del catàleg (vegeu la figura 2.8).

Format JSON

A l'exemple us demanem que treballeu amb una representació JSON. La simplicitat, la lleugeresa i la facilitat de lectura fan ideal aquesta representació per treballar amb aplicacions i dispositius que tenen restriccions pel que fa al volum de dades a intercanviar. Les aplicacions mòbils que consumeixin dades de serveis web RESTful són un molt bon exemple en aquest sentit; la quantitat d'informació que s'intercanviaran client i servidor per fer les operacions és molt menor en una aproximació JSON + RESTful que en una aproximació XML + SOAP.

Figura 2.8. Entitat Book



La representació en JSON d'un recurs llibre és la següent:

```
{
  "isbn":"9788425343537",
  "author":"Ildefonso Falcones",
  "title":"La catedral del mar"
}
```

Per fer la transformació entre objectes Java i JSON i a l'inrevés cal que importeu `Jersey` i `Jackson` com a artefactes al pom.xml. Aquesta tasca ja l'hem fet per vosaltres a la configuració inicial del projecte; les línies afegides al pom.xml són aquestes:

```
<dependency>
  <groupId>org.glassfish.jersey.core</groupId>
  <artifactId>jersey-server</artifactId>
  <version>2.22.1</version>
</dependency>
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-json</artifactId>
  <version>1.19</version>
  </dependency>
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId>
  <version>2.22.1</version>
</dependency>
```

La capa de serveis serà la que treballarem i haurà de proporcionar **un servei web RESTful** que permeti als clients fer les següents operacions sobre un catàleg de llibres:

- Llistar tots els llibres del catàleg.
- Consulta d'un llibre mitjançant l'ISBN.
- Creació d'un llibre al catàleg.
- Actualització de les dades d'un llibre.
- Esborrar un llibre del catàleg.
- Cercar llibres per títol.

Repository 'in memory'

Tot i que podríem haver optat per fer l'exemple amb un repositori connectat a una font de dades persistent com una base de dades relacional i utilitzar l'API JPA per definir les entitats del projecte, s'ha considerat que això afegeix "soroll" a l'exemple i us faria portar a terme algunes tasques de configuració que no són pròpies de l'objectiu principal. Per aquest motiu, farem servir un repositori *in memory*.

La capa de persistència també la teniu implementada i conté l'objecte repositori que permet mapar les dades de la font de dades amb l'objecte de domini. En el nostre cas, per simplicitat, farem servir un repositori *in memory* que tindrà una llista precarregada amb els llibres del catàleg.

'Bug' a la versió 4.1.1 de Glassfish

Si teniu la versió 4.1.1 de Glassfish sembla que hi ha un error en treballar amb JAX-RS que fa que quan crideu operacions que involucren la transformació entre JSON i les representacions com a objecte Java de les entitats del domini obtingueu la següent excepció:

```
java.lang.NoClassDefFoundError:
Could not initialize class org.eclipse.persistence.jaxb.
    BeanValidationHelper
```

Per solucionar-ho cal que descarregueu el fitxer org.eclipse.persistence.moxy.jar que trobareu enllaçat als annexos de la unitat, li tragueu l'extensió .zip, substituïu el que teniu amb el mateix nom al Glassfish, a la carpeta *<GLASSFISH_HOME>\glassfish\modules*, i reinicieu el servidor.

2.2.2 Creació i prova del servei web RESTful

Descarregueu el codi del projecte "Restbooksioc" de l'enllaç que trobareu als annexos de la unitat, i importeu-lo a NetBeans. Tot i que també podeu descarregar-vos el projecte en l'estat final des dels annexos, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

Primer ens cal indicar quines són les classes que ha de tractar com a recursos REST. Això es pot fer de moltes maneres, una d'elles és crear una classe de configuració anomenada `ApplicationConfig` al paquet `cat.xtec.ioc.service` amb el següent codi:

```
1  package cat.xtec.ioc.service;
2
3  import javax.ws.rs.core.Application;
4
5  @javax.ws.rs.ApplicationPath("rest")
6  public class ApplicationConfig extends Application {
7
8  }
```

On s'indica que cal afegir "/rest" (o el que vulgueu posar) abans del nom del servei web per cridar-lo.

Un cop fet, crearem una classe que exposarà les operacions que volem; anomenarem la classe `BooksRestService`, la crearem al paquet `cat.xtec.ioc.service` i l'anotarem amb `@Path(''/books'')` i amb `@Singleton`.

```
1  @Path("/books")
2  @Singleton
3  public class BooksRestService {
4
5  }
```

Quan NetBeans us demani quins imports voleu afegir especifiqueu els del paquet `javax.ws.rs`.

Fixeu-vos que:

- L'anotació `@Path(''/books'')` indica que el recurs estarà accessible a l'URI `/books`. Accedirem, doncs, als recursos amb l'URL localhost:8080/restbooksioc/rest/books.
- L'anotació `@Singleton` ens cal, pel fet que estem utilitzant una repositori *in memory* i necessitem que el servei web no s'inicialitzi cada vegada que es fa una petició, per tal de no perdre els canvis que anem fent al catàleg de llibres. Si féssiu servir una font de dades persistent no us caldria aquesta anotació, ja que els canvis persistirien a cada petició.

Un cop creada la classe cal que hi afegiu una referència al repositori del catàleg de llibres amb el següent codi:

```
1  private BookRepository bookRepository = new InMemoryBookRepository();
```

I ja podem començar a codificar el primer servei que volem oferir; per exemple, la consulta de tots els llibres del catàleg. Per fer-ho creeu un mètode anomenat `getAll` amb el següent codi:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Book> getAll() {
    return this.bookRepository.getAll();
}
```

Fixeu-vos que:

- Hem anotat el mètode amb l'anotació `@GET` per indicar que aquest mètode respondrà a peticions HTTP de tipus `GET`.
- Hem anotat el mètode amb `@Produces(MediaType.APPLICATION_JSON)` per indicar que el resultat del mètode serà del tipus MIME *"application/json"*, ja que tornarem la llista de llibres en format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/restbooksioc/rest/books; si tot ha anat bé veureu la representació JSON del catàleg de llibres al navegador:

```
[{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
     mar"},
{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
    el misterio de las catedrales"}]
```

Passem a crear ara el servei de consulta de llibres per ISBN (l'operació `Read` de CRUD); per fer-ho, creeu un mètode anomenat `find` amb el següent codi:

```
@GET
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)
public Book find(@PathParam("isbn") String isbn) {
    return this.bookRepository.get(isbn);
}
```

Hem anotat el mètode amb l'anotació `@GET` per indicar que aquest mètode respondrà a peticions HTTP de tipus `GET`.

Hem anotat el mètode amb *@Produces(MediaType.APPLICATION_JSON)* per indicar que el resultat del mètode serà del tipus MIME "application/json", ja que tornarem la informació del llibre en format JSON.

Hem anotat el mètode amb l'anotació `@Path(''{isbn}'')` per indicar que la informació del llibre la tornarem quan ens arribin peticions `GET` a l'URI `/books/{isbn}`; és a dir, el que posem darrere de `/books` serà l'ISBN del llibre que volem consultar. Les anotacions `@Path` dels mètodes sempre s'afegeixen a l'anotació de l'arrel del recurs que heu definit a la classe.

El paràmetre que rep el mètode s'ha anotat amb l'anotació `@PathParam(''isbn'')`. Aquesta és la forma que proporciona JAX-RS per extreure els paràmetres d'una petició. En aquest cas estem extraient un paràmetre del *path*, de l'URI, i el passem com a *String* al mètode `find`.

JAX-RS proporciona un ampli conjunt d'anotacions per fer aquesta tasca fàcil, entre elles destaquem `@PathParam`, `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@HeaderParam` i `@FormParam`.

Per exemple, podríem utilitzar `@QueryParam("year")` per extreure un paràmetre que vingués en una petició d'aquesta forma: localhost:8080/restbooksioc/rest/books?year=2016.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/restbooksioc/rest/books/9788425343537; si tot ha anat bé, veureu la representació JSON del llibre consultat al navegador:

```
[{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
    mar"}]
```

Creem ara el servei que permeti donar d'alta un llibre al catàleg (l'operació `Create` de CRUD); per fer-ho, creeu un mètode anomenat `create` amb el següent codi:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public void create(Book book) {
    this.bookRepository.add(book);
}
```

Hem anotat el mètode amb l'anotació `@POST` per indicar que aquest mètode respondrà a peticions HTTP de tipus `POST`.

Hem anotat el mètode amb `@Consumes(MediaType.APPLICATION_JSON)` per indicar que la informació del llibre que es vol donar d'alta vindrà en format JSON.

Fixeu-vos que tan sols amb aquesta informació JAX-RS és capaç, quan rep una petició `POST` a l' URI `/books` amb una representació JSON d'un llibre, de crear un objecte de tipus `Book` i passar-lo al mètode `create`. No és fantàstic?

Swagger i Postman

Per documentar i provar API RESTful teniu moltes alternatives, entre elles destaquem Swagger swagger.io i Postman www.getpostman.com, que és una extensió per al navegador Google Chrome. </note>

Ara tocaria provar aquesta funcionalitat, però tenim un problema: com podem enviar peticions `POST` sense fer una aplicació web amb un formulari? Per fer una petició `GET` tan sols hem de posar l'URL al navegador i ja ho tenim, però per fer peticions `POST` ens caldrà alguna utilitat extra. La nostra proposta és que feu servir **cURL**, que és una eina molt útil per fer peticions HTTP de diferents tipus i és multiplataforma. Si feu servir Linux possiblement ja la tingueu instal·lada al sistema; en cas que utilitzeu, Windows la podeu descarregar en el següent enllaç: curl.haxx.se/download.html, o bé aconseguir un .msi que us faciliti la instal·lació.

Afegiu el paràmete `-v` (*verbose*) a les crides a cURL si voleu veure més informació sobre les peticions i respostes que feu.

La sintaxi de cURL per fer peticions és molt senzilla; per exemple, per consultar tots els llibres del catàleg feu un *command prompt*:

```
curl localhost:8080/restbooksioc/rest/books
```

Per provar la funcionalitat que permet afegir un llibre al catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició `POST` a l'URL localhost:8080/restbooksioc/rest/books especificant la informació del llibre amb JSON; això ho podeu fer amb cURL i la següent comanda:

```
curl –H "Content–Type: application/json" –X POST –d "{\"isbn
    \":\"9788499301518\",\"author\":\"J.K. Rowling\",\"title\":\"Harry Potter
    y la piedra filosofal\"}" http://localhost:8080/restbooksioc/rest/books
```

Fixeu-vos-hi: li diem que farem una petició `POST` amb el paràmetre `-X`, li especifiquem el format JSON del llibre amb el paràmetre `-d` i indiquem que el format és JSON especificant-ho a la capçalera amb el paràmetre `-H`.

Executeu la comanda anterior i després consulteu el llistat de llibres amb:

```
curl localhost:8080/restbooksioc/rest/books
```

Si tot ha anat bé, veureu que el nou llibre s'ha afegit al catàleg de llibres:

```
[{"isbn":"9788499301518","author":"J.K. Rowling","title":"Harry Potter y la
    piedra filosofal"},
{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
    mar"},
{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
    el misterio de las catedrales"}]
```

Creem ara el servei que permeti modificar la informació d'un llibre del catàleg (l'operació `Update` de CRUD); per fer-ho, creeu un mètode anomenat `update` amb el següent codi:

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
public void edit(Book book) {
    this.bookRepository.update(book);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la creació de llibres, l'únic que canvia és que farem servir el verb `PUT` en lloc del verb `POST`.

Per provar la funcionalitat que permet modificar la informació d'un llibre del catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició `PUT` a l'URL localhost:8080/restbooksioc/rest/books especificant la nova informació del llibre amb JSON; això ho podeu fer amb cURL i la següent comanda:

```
curl –H "Content–Type: application/json" –X PUT –d "{\"isbn
    \":\"9788425343537\",\"author\":\" Ildefonso Falcones\",\"title\":\" LA
    CATEDRAL DEL MAR \"}" http://localhost:8080/restbooksioc/rest/books
```

Executeu la comanda anterior i després consulteu la informació del llibre que heu modificat amb:

```
curl localhost:8080/restbooksioc/rest/books/9788425343537
```

Si tot ha anat bé, veureu que el títol del llibre ha canviat a majúscules:

```
{"isbn":"9788425343537","author":" Ildefonso Falcones","title":" LA CATEDRAL
    DEL MAR "}
```

Creem ara el servei que permeti esborrar un llibre del catàleg (l'operació `Delete` de CRUD); per fer-ho, creeu un mètode anomenat `remove` amb el següent codi:

```
@DELETE
@Path("{isbn}")
public void remove(@PathParam("isbn") String isbn) {
    this.bookRepository.delete(isbn);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la consulta de llibres per ISBN, l'únic que canvia és que farem servir el verb `DELETE` en lloc del verb `GET`.

Per provar la funcionalitat que permet esborrar un llibre del catàleg desplegueu el projecte fent *Run* a NetBeans i feu una petició `DELETE` a l'URL localhost:8080/restbooksioc/rest/books especificant l'ISBN del llibre que volem esborrar; això ho podeu fer amb cURL i la següent comanda:

```
curl –X DELETE http://localhost:8080/restbooksioc/rest/books/9788425343537
```

Executeu la comanda anterior i després consulteu el llistat de llibres amb:

```
curl localhost:8080/restbooksioc/rest/books
```

Si tot ha anat bé, veureu que el llibre ja no el teniu al catàleg de llibres:

```
[{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
    el misterio de las catedrales"}]
```

Finalment, codificarem una operació que ens permeti cercat llibres per títol; per fer-ho, creeu un mètode anomenat `findByTitle` amb el següent codi:

```
@GET
@Path("findByTitle/{title}")
@Produces(MediaType.APPLICATION_JSON)
public List<Book> findByTitle(@PathParam("title") String title) {
    return this.bookRepository.findByTitle(title);
}
```

Si voleu cercar títols de llibre que continguin espais en blanc amb cURL ho haureu de fer posant %20 enlloc de l'espai en blanc.

Les consideracions en aquest cas són les mateixes que hem vist per al cas d'un llibre per ISBN, l'únic que canvia és que l'anotació `@Path` ara comença amb `findByTitle` i després té el paràmetre `{title}` per indicar que la consulta de llibres per títol la tornarem quan ens arribin peticions `GET` a l'URI `/books/findByTitle/{title}`, és a dir, el que posarem darrere del `/books/findByTitle/` serà el títol que volem cercar.

Per provar la funcionalitat que permet cercar un llibre del catàleg per títol desplegueu el projecte fent *Run* a NetBeans i feu una petició `GET` a l'URL localhost:8080/restbooksioc/rest/books/findByTitle especificant el títol que volem cercar; això ho podeu fer amb cURL i la següent comanda (cercarem tots els llibres que tenen la paraula "catedral" al títol):

```
curl http://localhost:8080/restbooksioc/rest/books/findByTitle/catedral
```

Executeu la comanda anterior i, si tot ha anat bé, veureu que us torna els dos llibres del catàleg que tenen la paraula “catedral” al títol:

```
[{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y el misterio de las catedrales"},
{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del mar"}]
```

I amb això ja heu codificat i provat el servei web RESTful que us permet treballar amb el catàleg de llibres.

El codi final de la classe `BooksRestService` és el següent:

```
@Path("/books")
@Singleton
public class BooksRestService {

    private BookRepository bookRepository = new InMemoryBookRepository();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> getAll() {
        return this.bookRepository.getAll();
    }

    @GET
    @Path("{isbn}")
    @Produces(MediaType.APPLICATION_JSON)
    public Book find(@PathParam("isbn") String isbn) {
        return this.bookRepository.get(isbn);
    }

    @GET
    @Path("findByTitle/{title}")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> findByTitle(@PathParam("title") String title) {
        return this.bookRepository.findByTitle(title);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void create(Book book) {
        this.bookRepository.add(book);
    }

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void edit(Book book) {
        this.bookRepository.update(book);
    }

    @DELETE
    @Path("{isbn}")
    public void remove(@PathParam("isbn") String isbn) {
        this.bookRepository.delete(isbn);
    }
}
```

2.3 Què s'ha après?

En aquest apartat heu vist les bases pel desenvolupament dels serveis web RESTful amb Java EE 7 i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web RESTful amb Java EE.
- A desenvolupar, desplegar i provar un servei web RESTful senzill amb Java EE.
- A desenvolupar, desplegar i provar un servei web RESTful complex amb Java EE que inclou operacions CRUD sobre un recurs.

Per aprofundir en aquests conceptes i veure com us pot ajudar NetBeans en la creació i el consum de serveis web RESTful us recomanem la realització de les activitats associades a aquest apartat.

3. Serveis web RESTful amb Java EE7. Consumint serveis web

Explicarem, mitjançant exemples, com podem consumir serveis web RESTful remots amb Java.

Un cop codificat el servei web RESTful, el següent que ens cal fer és accedir-hi, i per fer-ho l'únic que ens cal és fer les diferents peticions HTTP a l'URI del servidor que tingui els recursos als quals volem accedir.

Es poden provar els serveis RESTful **amb qualsevol eina que permeti fer peticions HTTP**.

La primera eina en la qual pensaríem tots és un navegador web (Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, etc.). El problema és que els navegadors, per defecte, tan sols poden fer peticions `GET` i peticions `POST`. Si voleu fer peticions d'un altre tipus (`PUT`, `DELETE`, `HEAD`) us caldrà instal·lar algun *plugin* al navegador que us ho permeti (Postman és un possible *plugin* per a Google Chrome, però n'hi ha molts).

Una altra opció és utilitzar la utilitat **cURL** que us permet fer tot tipus de peticions HTTP per línia de comandes.

Si el que volem és consumir serveis web RESTful des de Java, l'única opció que hi havia abans de JAX-RS 2.0 era fer servir l'API de baix nivell `java.net.HttpURLConnection` o alguna llibreria propietària que us fes la vida una mica més fàcil.

JAX-RS 2.0 proporciona una API client estàndard que permet fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

L'API client de JAX-RS és molt senzilla d'utilitzar; moltes vegades tan sols us caldrà utilitzar tres classes: `Client`, `WebTarget` i `Response`.

Amb SOAP, l'API JAX-WS forma part del mateix JDK, mentre que amb JAX-RS cal que incloguem JAX-RS al *classpath* del client. JAX-WS generava un conjunt d'artefactes automàticament per poder connectar amb els serveis web, mentre que JAX-RS no en genera cap; tan sols cal tenir el .jar de la implementació de l'API al *classpath*.

3.1 Un client per al servei web RESTful que contesta "Hola"

Veurem com consumir serveis web RESTful des d'una aplicació Java *stand-alone* utilitzant l'API que proporciona JAX-RS.

3.1.1 Creació i configuració inicial del projecte

Descarregueu el codi del projecte "Resthelloioc" en l'estat inicial d'aquest apartat des de l'enllaç que trobareu als annexos de la unitat, i importeu-lo a NetBeans.
Tot i que també podeu descarregar-vos el projecte en l'estat final des del annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat.

El projecte és un senzill servei web RESTful que respon a peticions `GET` a l'URI localhost:8080/resthelloioc/rest/hello amb la salutació "*Hello World!!!*".

3.1.2 Creació del client Java 'stand-alone'

Quan NetBeans us demani quins imports voleu afegir especifiqueu els del paquet `javax.ws.rs`.

Creeu la classe Java que farà de client al paquet `cat.xtec.ioc.resthelloioc.client` i l'anomeneu `HelloWorldClient` amb el següent codi:

```
1  public class HelloWorldClient {
2
3      public static void main(String[] args) {
4          Client client = ClientBuilder.newClient();
5          WebTarget target = client.target("http://localhost:8080/resthelloioc/
               rest/hello");
6          Invocation invocation = target.request(MediaType.TEXT_HTML).buildGet();
7          Response res = invocation.invoke();
8          System.out.println(res.readEntity(String.class));
9      }
10 }
```

Fem servir la interfície `Client` per construir l'objecte `WebTarget`.

```
1  Client client = ClientBuilder.newClient();
```

L'objecte `WebTarget` representa l'URI on enviarem les peticions; en el nostre cas, a localhost:8080/resthelloioc/rest/hello.

```
1  WebTarget target = client.target("http://localhost:8080/resthelloioc/rest/hello
       ");
```

Un cop tenim l'URI on enviarem les peticions ens cal construir la petició HTTP. Ho fem amb la interfície `Invocation`, que permet, entre moltes altres coses, especificar el tipus MIME de la petició:

```
1  Invocation invocation = target.request(MediaType.TEXT_HTML).buildGet();
```

En aquest punt, quan construïm la petició, podrem especificar paràmetres, especificar el *path*, els objectes i el format d'aquests per a les peticions `POST` i `PUT`, etc. Tot això ho podrem fer mitjançant els mètodes de la interfície `Invocation`.

La creació de l'objecte `Invocation` no executa encara la petició, i per fer-ho cal que executeu el mètode `invoke`:

```
1 Response res = invocation.invoke()
```

Aquesta crida, en el nostre cas, fa una petició `GET` a un servei web RESTful que hi ha a localhost:8080/resthelloioc/rest/hello i ens torna el resultat en text/HTML.

API fluent

Una API fluent és un patró de disseny que permet encadenar les crides als mètodes d'un objecte amb l'objectiu de fer un codi més elegant, concís i comprensible.

Noteu que, pel fet que l'API client de JAX-RS és una API *fluent*, podríem escriure tot el codi anterior d'una forma molt més concisa:

```
1 Response response = ClientBuilder.newClient()
2                     .target("http://localhost:8080/resthelloioc/rest/hello")
3                     .request(MediaType.TEXT_HTML).get();
```

L'objecte `Response` representa la resposta del servei web i conté, a part del "*Hello World!!!*", informació de la resposta HTTP rebuda del servei web. Amb `Response` podreu verificar els codis HTTP de retorn, accedir a les capçaleres, a les *cookies* i al valor de retorn.

Accedirem al valor retornat pel servei web amb el mètode `readEntity`; a aquest mètode li heu de passar un objecte que permeti fer la transformació entre la representació del recurs que envia el servidor (en el nostre cas, un senzill *String*, però poden ser tipus complexes) i el tipus de dades que volem. JAX-RS s'encarregarà de fer aquestes transformacions.

```
1 res.readEntity(String.class)
```

Ara ja tan sols ens queda desplegar el servei web i executar el client per veure si es comporta com volem.

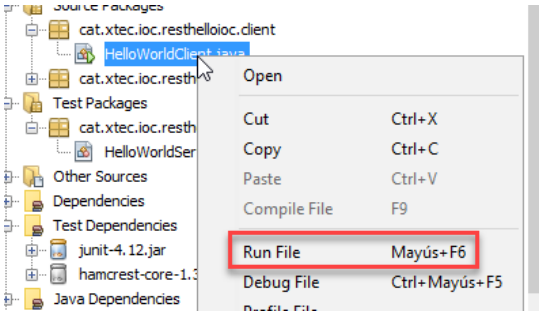
3.1.3 Desplegament del servei web i prova amb el client Java

Desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web està desplegat correctament accedint a l'URL localhost:8080/resthelloioc/rest/hello amb un navegador. El servei web us ha de tornar la salutació "*Hello World!!!*".

Si tot és correcte ja podeu executar el client; per fer-ho, poseu-vos damunt de la classe `HelloWorldClient` i feu *Run File* a NetBeans (vegeu la figura 3.1).

FIGURA 3.1. Execució d'una classe Java a NetBeans



I veureu la salutació "*Hello World!!!*" a la consola de sortida de NetBeans:

```
Hello World!!!

BUILD SUCCESS
```

3.2 El servei web de dades de llibres. Consum i testeig

Veurem com consumir i fer un test d'integració d'un servei web que permet als clients fer operacions sobre un catàleg de llibres des d'un conjunt de test d'integració amb JUnit i l'API client que proporciona JAX-RS.

El primer que fareu serà desplegar a Glassfish el servei web REST de gestió del catàleg de llibres i després creareu el conjunt de tests d'integració que faran peticions HTTP al servei web amb l'API client de JAX-RS.

Els **tests d'integració** difereixen dels tests unitaris en el fet que no testegen el codi de forma aïllada, sinó que requereixen que el codi a testejar estigui desplegat al servidor d'aplicacions per funcionar.

3.2.1 Creació i configuració inicial del projecte

Descarregueu el codi del projecte "Restbooksioc" des de l'enllaç disponible als annexos de la unitat i importeu-lo a NetBeans. Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'altre enllaç disponible, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar els projectes a NetBeans.

Aquest projecte correspon al servei web RESTful de gestió d'un catàleg de llibres i ja té codificades les següents operacions:

- Llistar tots els llibres del catàleg.
- Consulta d'un llibre mitjançant l'ISBN.
- Creació d'un llibre al catàleg.
- Actualització de les dades d'un llibre.

- Esborrar un llibre del catàleg.

- Cercar llibres per títol.

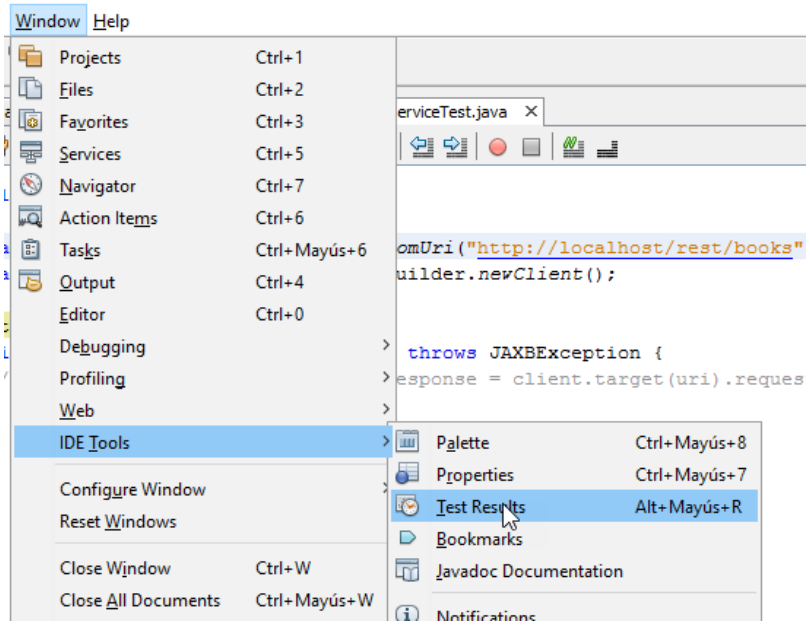
Crearem els tests d'integració dins el mateix projecte que conté el codi del servei web que volem provar. Una altra opció, igualment vàlida, seria crear un nou projecte i posar-hi només el test.

JUnit és un *framework* de test que utilitza anotacions per identificar els mètodes que especifiquen un test. A JUnit, un test, ja sigui unitari o d'integració, és un mètode que s'especifica en una classe que només s'utilitza per al test. Això s'anomena *classe de test*. Un mètode de test amb JUnit 4 es defineix amb l'anotació `@org.junit.Test`. En aquest mètode s'utilitza un mètode d'asserció en el qual es comprova el resultat esperat de l'execució de codi en comparació del resultat real.

Per tal de fer els tests d'integració farem servir JUnit 4; per fer-ho ens cal tenir la dependència a JUnit al pom.xml del projecte. Al projecte de partida ja hi teniu aquesta dependència afegida.

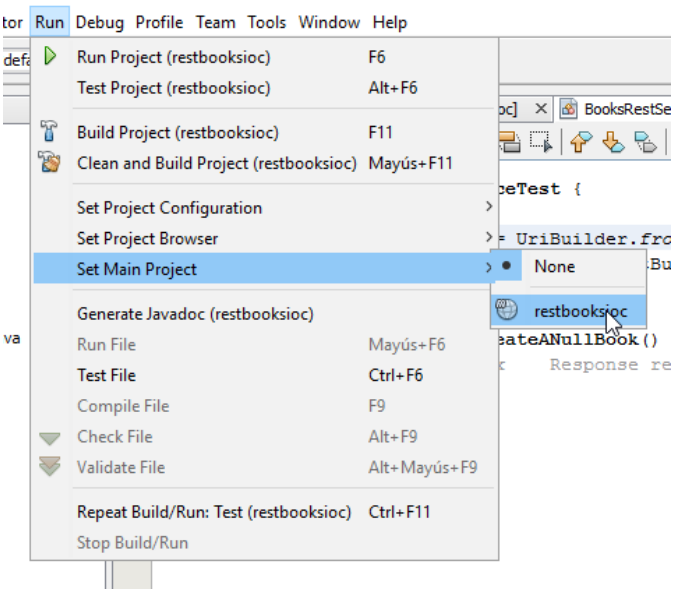
Tot i que el resultat de l'execució dels tests es pot veure a la finestra de sortida de NetBeans, és molt més còmode visualitzar-ho a la finestra de resultats de test que proporciona també NetBeans. Per mostrar aquesta finestra aneu al menú *Window / IDE Tools / Test Results* (vegeu la figura 3.2).

Figura 3.2. Finestra de resultats dels tests



A Netbeans, els tests es poden executar de forma individual, és a dir, classe per classe o tots els del projecte. Si voleu executar tots els tests d'un projecte primer heu de designar com a projecte principal el projecte "Restbooksioc", tal com podeu veure en la figura 3.3.

FIGURA 3.3. Assignació del projecte com a projecte principal



Un cop fet això, desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans.

Comproveu que el servei web s'ha desplegat correctament accedint a l'URL localhost:8080/restbooksioc/rest/books amb un navegador. El servei web us ha de tornar el llistat de llibres que té el catàleg en format JSON:

```
[{"isbn":"9788425343537","author":"Ildefonso Falcones","title":"La catedral del
     mar"},
{"isbn":"9788467009477","author":"Jose Maria Peridis Perez","title":"La luz y
    el misterio de las catedrales"}]
```

3.2.2 Creació i execució dels tests d'integració

Quan NetBeans us demani quins imports voleu afegir, especifiqueu els del paquet `javax.ws.rs`.

Ara toca començar a crear els tests d'integració per provar el servei web RESTful de gestió del catàleg.

Per fer-ho, creeu un nou paquet dins de `Test Packages` anomenat, per exemple, *cat.xtec.ioc.test*, i creeu-hi una classe Java anomenada `BooksRestServiceTest` amb el següent codi:

```
package cat.xtec.ioc.test;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

public class BooksRestServiceTest {

    private static final Client client = ClientBuilder.newClient();

}
```

Estructura dels tests

Un dels patrons més utilitzats a l'hora d'estructurar el codi d'un test és l'anomenat **AAA** (de l'anglès **A**rrange-**A**ct-**A**ssert); amb aquest, els tests sempre tindran una fase de **preparació** (*Arrange*), una d'**execució** (*Act*) i una de **verificació** de resultats (*Assert*).

El primer test que fareu serà un test que **comprovi que la consulta de tots els llibres del catàleg us torna la llista sencera de llibres**; per fer-ho, creeu un mètode

anomenat `shouldReturnAllBooks` dins la classe `BooksRestServiceTest` amb el següent codi:

```
@Test
public void shouldReturnAllBooks() {
    // Arrange
    URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
        port(8080).build();
    WebTarget target = client.target(uri);
    Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet
        ();
    Book first = new Book("9788425343537", "Ildefonso Falcones", "La catedral
        del mar");
    Book second = new Book("9788467009477", "Jose Maria Peridis Perez", "La luz
         y el misterio de las catedrales");

    // Act
    Response res = invocation.invoke();
    List<Book> returnedBooks = res.readEntity(new GenericType<List<Book>>() {})
        ;

    // Assert
    assertTrue(returnedBooks.contains(first));
    assertTrue(returnedBooks.contains(second));
}
```

Si analitzeu el codi veureu diverses coses importants: la primera és que heu anotat el mètode `shouldReturnAllBooks` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test.

A la fase de **preparació** feu servir la interfície `Client` per construir l'objecte `WebTarget`, que representa l'URI on enviareu les peticions; en el vostre cas, a localhost:8080/restbooksioc/rest/books.

```
URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").port
    (8080).build();
WebTarget target = client.target(uri);
```

Un cop teniu l'URI on enviareu les peticions cal construir la petició HTTP. Ho feu amb la interfície `Invocation`, que permet, entre moltes altres coses, especificar el tipus MIME de la petició:

```
Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet();
```

Tot l'anterior ha servit per preparar la petició que voleu enviar al servei web; el següent que fareu serà l'**execució**, cridant el mètode `invoke`:

```
Response res = invocation.invoke();
```

Aquesta crida, en el vostre cas, fa una petició `GET` a un servei web RESTful que hi ha a localhost:8080/restbooksioc/rest/books i torna el resultat en format `application/json`.

L'objecte `Response` representa la resposta del servei web que us permetrà verificar que el resultat és correcte. Per fer-ho, accedireu al valor retornat pel servei web amb el mètode `readEntity`; a aquest mètode li heu de passar un objecte que permeti fer la transformació entre la representació del recurs que envia el servidor

(en el vostre cas, la llista de llibres en format JSON) i el tipus de dades que voleu. JAX-RS s'encarregarà de fer aquestes transformacions.

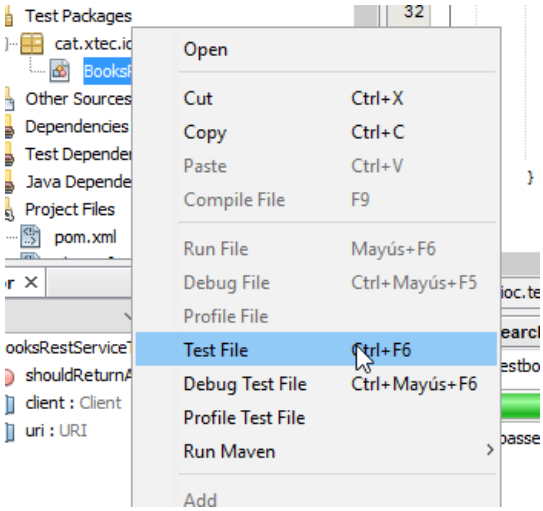
```
List<Book> returnedBooks = res.readEntity(new GenericType<List<Book>>() {});
```

En la fase de **verificació** comproveu, per exemple, que el títol dels dos llibres coincideix amb l'esperat:

```
assertTrue(returnedBooks.contains(first));
assertTrue(returnedBooks.contains(second));
```

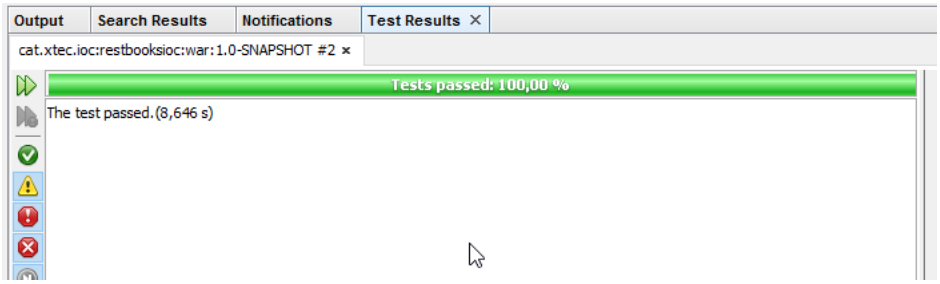
Ja podeu executar el test que heu creat fent *Test File* (vegeu la figura 3.4) al menú contextual de la classe de `Test`.

Figura 3.4. Execució del test



Si tot ha anat bé veureu el resultat a la finestra de `Test` (vegeu la figura 3.5).

Figura 3.5. Resultat de l'execució del test



El segon test que fareu serà un test que **comprovarà que si consulteu un llibre per ISBN que no existeix al catàleg de llibres el servei web us torna el codi HTTP 404 – Not Found**.

Per fer-ho, creeu un mètode anomenat `nonExistentBookShouldReturn404` dins la classe `BooksRestServiceTest` amb el següent codi:

```
@Test
public void nonExistentBookShouldReturn404() {
    // Arrange
```

```
    URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
        port(8080).build();
    WebTarget target = client.target(uri).path("unknownISBN");
    Invocation invocation = target.request(MediaType.APPLICATION_JSON).buildGet
        ();

    // Act
    Response res = invocation.invoke();

    // Assert
    assertEquals(Response.Status.NOT_FOUND.toString(), res.getStatusInfo().
        toString());
}
```

Fixeu-vos que ara l'URI on enviareu les peticions `GET` és localhost:8080/restbooksioc/rest/books, i que li afegiu el *path* */unknownISBN*. Aquesta crida correspon al mètode `find` del servei web que té el següent codi:

```
@GET
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)
public Book find(@PathParam("isbn") String isbn) {
  return this.bookRepository.get(isbn);
}
```

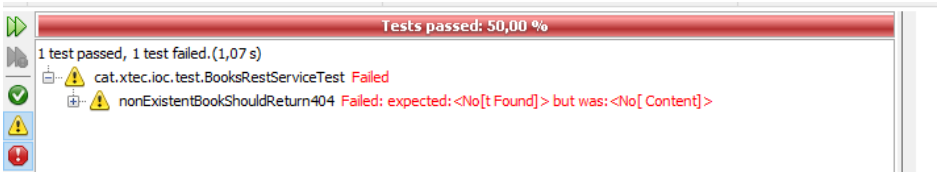
A la fase d'execució feu el mateix que per cercar tots els llibres del catàleg, i a la verificació ara comprovareu el codi HTTP retornat per la crida:

```
assertEquals(Response.Status.NOT_FOUND.toString(), res.getStatusInfo().toString
    ());
```

L'objecte `Response` representa la resposta del servei web i conté, a part del contingut, molta informació de la resposta HTTP rebuda. Amb `Response` podeu verificar els codis HTTP de retorn i accedir a les capçaleres, a les *cookies* i al valor de retorn.

Si executeu els tests veureu que el test que comprova que el servei web torni un *404 – Not Found* si consulteu un llibre per ISBN que no existeix falla perquè torna un *204 – No content* enlloc de l'esperat *404 – Not Found* (vegeu la figura 3.6).

FIGURA 3.6. Resultat erroni de l'execució del test



El problema és la codificació del mètode que fa la cerca per ISBN al servei web; us tocarà modificar-lo si voleu que torni un error 404 enlloc d'un 204. Per fer-ho obriu la classe `BooksRestService` del paquet `cat.xtec.ioc.service` i canvieu el codi del mètode `find` pel següent:

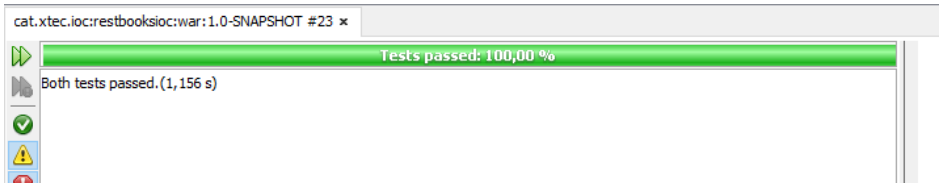
```
@GET
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)
public Response find(@PathParam("isbn") String isbn) {
    Book book = this.bookRepository.get(isbn);
```

```
6       if(book == null) {
7           throw new NotFoundException();
8       }
9       return Response.ok(book).build();
10  }
```

Hem canviat el tipus de retorn a `Response`, i en aquest objecte hi afegim el llibre en cas que el trobem. En cas que no trobem el llibre llencem una excepció de tipus `NotFoundException` que el *runtime* de JAX-RS s'encarregarà de transformar en un error *HTTP 404 – Not Found*.

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els dos tests (vegeu la figura 3.7).

Figura 3.7. Resultat correcte de l'execució dels tests



El tercer test que mostrarem és un test que **comprovarà que no es puguin afegir llibres amb contingut nul**. El que volem és que si la informació del llibre que es vol afegir és nul·la, el servei web ens torni el codi *HTTP 400 – Bad Request*.

Per fer-ho, creeu un mètode anomenat `attemptsToCreateNullBooksShouldReturn400` dins la classe `BooksRestServiceTest` amb el següent codi:

```
1   @Test
2   public void attemptsToCreateNullBooksShouldReturn400() {
3       // Arrange
4       URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
            port(8080).build();
5       WebTarget target = client.target(uri);
6       Invocation invocation = target.request(MediaType.APPLICATION_JSON).
            buildPost(null);
7
8       // Act
9       Response res = invocation.invoke();
10
11      // Assert
12      assertEquals(Response.Status.BAD_REQUEST.toString(), res.getStatusInfo().
            toString());
13  }
```

Fixeu-vos que ara enviarem peticions `POST` a l'URI localhost:8080/restbooksioc/rest/books especificant un objecte nul. Aquesta crida correspon al mètode `create` del servei web:

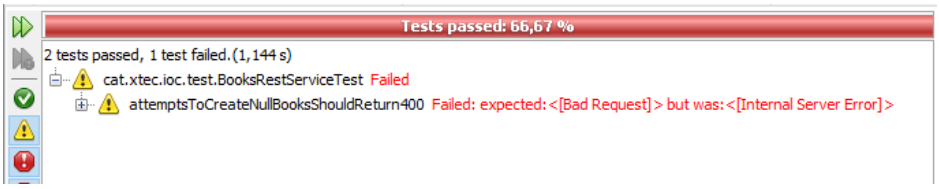
```
1   @POST
2   @Consumes(MediaType.APPLICATION_JSON)
3   public void create(Book book) {
4       this.bookRepository.add(book);
5   }
```

A la fase d'execució feu el mateix que per cercar tots els llibres del catàleg i a la verificació tornareu a comprovar el codi HTTP retornat per la crida:

```
// Assert
assertEquals(Response.Status.BAD_REQUEST.toString(), res.getStatusInfo().
    toString());
```

Si executeu els tests veureu que el test que comprova que no es puguin afegir llibres amb contingut nul falla perquè torna un *500 – Internal Server Error* enlloc del *400 – Bad Request*(vegeu la figura 3.8).

FIGURA 3.8. Resultat erroni de l'execució del test



El problema és la codificació del mètode que fa la creació de llibres al servei web; us tocarà modificar-lo si voleu que torni un error 400 enlloc d'un 500. Per fer-ho, obriu la classe `BooksRestService` del paquet `cat.xtec.ioc.service` i canvieu el codi del mètode `create` pel següent:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response create(Book book) {
    if(book == null) {
        throw new BadRequestException();
    }
    this.bookRepository.add(book);

    return Response.ok(book).build();
}
```

Heu canviat el tipus de retorn a `Response`, i si us passen un llibre nul llenceu una excepció de tipus `BadRequestException` que el *runtime* de JAX-RS s'encarregarà de transformar en un error *HTTP 400 – Bad Request*.

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els tres tests (vegeu la figura 3.9).

FIGURA 3.9. Resultat correcte de l'execució dels tests

El quart i últim test que mostrarem és un test que **comprovarà que la creació d'un llibre ens torni l'URL amb la qual es podrà consultar el llibre**. Per fer-ho, creeu un mètode anomenat `createBookShouldReturnTheURLToGetTheBook` dins la classe `BooksRestServiceTest` amb el següent codi:

```
@Test
public void createBookShouldReturnTheURLToGetTheBook() {
```

```
3       // Arrange
4       Book book = new Book("9788423342518", "Clara Sanchez", "Lo que esconde tu
            nombre");
5       URI uri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/books").
            port(8080).build();
6       WebTarget target = client.target(uri);
7       Invocation invocation = target.request(MediaType.APPLICATION_JSON).
            buildPost(Entity.entity(book, MediaType.APPLICATION_JSON));
8
9       // Act
10      Response res = invocation.invoke();
11      URI returnedUri = res.readEntity(URI.class);
12
13      // Assert
14      URI expectedUri = UriBuilder.fromUri("http://localhost/restbooksioc/rest/
            books").port(8080).path("9788423342518").build();
15      assertEquals(expectedUri, returnedUri);
16  }
```

El codi del test és el mateix que el codi del test que comprovava que no es poguessin afegir llibres nul; tan sols canvia la comprovació final i el llibre a afegir. Evidentment, si executeu aquest test fallarà, ja que ara el mètode `create` del servei web està tornant la representació del llibre en format JSON i no l'URL on es pot consultar el llibre.

Per fer que torni l'URL cal que modifiqueu un altre cop el codi del servei web. Per fer-ho, obriu la classe `BooksRestService` del paquet `cat.xtec.ioc.service` i canvieu el codi del mètode `create` pel següent:

```
1   @POST
2   @Consumes(MediaType.APPLICATION_JSON)
3   public Response create(Book book) {
4       if(book == null) {
5           throw new BadRequestException();
6       }
7       this.bookRepository.add(book);
8
9       URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getIsbn()).build()
            ;
10      return Response.ok(bookUri).build();
11  }
```

També cal que afegiu el següent atribut a la classe:

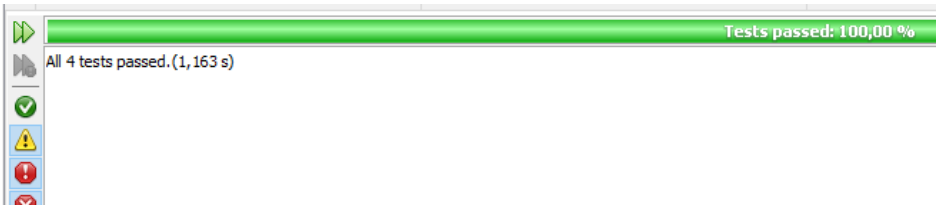
```
1   @Context  private UriInfo uriInfo;
```

L'objecte `UriInfo` us permet accedir a informació sobre la petició, i amb això podreu construir l'URL amb la qual es podrà consultar el llibre creat i tornar-la a l'objecte `Response`:

```
1   URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getIsbn()).build();
```

Desplegueu el servei web fent *Run* a NetBeans i torneu a executar el test. Si tot ha anat bé veureu que s'han executat correctament els quatre tests (vegeu la figura 3.10).

FIGURA 3.10. Resultat correcte de l'execució dels tests



Aquests quatre exemples us donen la base per tal que pugueu fer tots els tests d'integració que se us acudeixin i així tenir el servei web de gestió del catàleg de llibres totalment provat!

3.3 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament de clients Java que accedeixin a serveis web RESTful amb Java EE 7 i els heu treballat de manera pràctica mitjançant exemples.

Concretament, heu après:

- Les bases de l'API Client de JAX-RS.
- A desenvolupar i provar un client Java *stand-alone* que consulti un servei web RESTful mitjançant l'API Client de JAX-RS.
- A fer tests d'integració que us permeten provar els serveis web RESTful.

Per aprofundir en algun d'aquests conceptes i veure com podeu treballar de forma asíncrona amb els serveis web RESTful us recomanem la realització de l'activitat associada a aquest apartat.