Serveis web amb Spring

Raúl Velaz Mayo

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) 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 em plataformes diferents puguin interactuar per construir aplicacions distribuïdes seguint arquitectures SOA.

Spring és un bastiment (*framework* en anglès) amb mòduls basat en Java Enterprise Edition. La principal característica del seu *core* és la utilització del patró de disseny inversió de control (de l'anglès *Inversion of Control*, IoC) i també la injecció de dependències (de l'anglès *Dependency Injection*), un tipus d'IoC. Spring ofereix també un conjunt de projectes/mòduls per desenvolupar serveis web SOAP i REST.

La unitat aborda els conceptes més rellevants dels serveis web SOAP amb Spring-WS com a tecnologia. Mitjançant els exemples, es veuran les bases teòriques que regeixen els serveis web SOAP i com es traslladen i apliquen en la construcció de serveis web SOAP amb Spring-WS, 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 clients Java capaços de consumir-los.

Així mateix, s'introduiran els conceptes més rellevants dels serveis web RESTful amb Spring MVC com a tecnologia. Mitjançant els exemples, coneixereu 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, a fer-ne el desplegament i a provar-los.

S'explicarà com codificar tests d'integració, clients Java amb Spring i clients AJAX capaços de consumir serveis web RESTful. Mitjançant els exemples, es veurà com es codifiquen aquests tipus de client.

Es descriuran, des d'un vessant teòric i pràctic, els aspectes més essencials dels diferents tipus de serveis web que podem crear amb Spring. 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.
- Identifica les tecnologies i els protocols implicats en la publicació i la 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.

2. Genera pàgines web dinàmiques analitzant i utilitzant tecnologies del servidor web que afegeixin codi al llenguatge de marques.

- Identifica les diferències entre l'execució de codi al servidor i al client web.
- Reconeix els avantatges d'unir les dues tecnologies en el procés de desenvolupament de programes.
- Identifica les llibreries i les tecnologies relacionades amb la generació per part del servidor de pàgines web amb guions embeguts.
- Utilitza aquestes tecnologies per generar pàgines web que incloguin interacció amb l'usuari en forma d'advertències i peticions de confirmació.
- Fa servir aquestes tecnologies per generar pàgines web que incloguin verificació de formularis.
- Empra aquestes tecnologies per generar pàgines web que incloguin modificació dinàmica del seu contingut i la seva estructura.
- Aplica aquestes tecnologies en la programació d'aplicacions web.

3. Desenvolupa aplicacions web híbrids seleccionant i utilitzant llibreries de codi i dipòsits heterogenis d'informació.

- Reconeix els avantatges que proporciona la reutilització de codi i l'aprofitament d'informació ja existent.
- Identifica llibreries de codi i tecnologies aplicables en la creació d'aplicacions web híbrides.
- Crea una aplicació web que recuperi i processi dipòsits d'informació ja existents.
- Crea dipòsits específics a partir d'informació existent a Internet i en magatzems d'informació.
- Utilitza llibreries de codi per incorporar funcionalitats específiques a una aplicació web.
- Programa serveis i aplicacions web utilitzant com a base informació i codi generats per tercers.
- Prova, depura i documenta les aplicacions generades.

1. Serveis web SOAP amb Spring Web Services

Mitjançant una aplicació d'exemple, veurem els conceptes més rellevants dels serveis web SOAP desenvolupats amb Spring Web Services (Spring-WS). 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) i, més recentment, per a la creació d'arquitectures basades en microserveis.

Quan es descriu una arquitectura SOA 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 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 interoperabilitat**.

Un servei web publica una lògica de negoci exposada com a servei als clients. La diferència 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.

Spring és un bastiment (*framework*, en anglès) amb mòduls basat en Java Enterprise Edition. La principal característica del seu *core* és la utilització del patró de disseny d'inversió de control (de l'anglès *Inversion of Control*) i també la injecció de dependències (de l'anglès *Dependency Injection*), un tipus d'IoC. Spring ofereix també un conjunt de projectes/mòduls per desenvolupar serveis web SOAP i RESTful.

Els serveis web i, per tant, les aplicacions orientades a serveis es poden implementar amb diferents tecnologies. Tant Java EE 7 com Spring proporcionen implementacions per treballar amb serveis web SOAP i serveis web RESTful.

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.

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.

Spring també proporciona un projecte anomenat Spring-WS (de l'anglès *Spring Web Services*) enfocat a la creació i el consum de serveis web SOAP. Spring-WS es basa completament en Spring i porta inherent al seu model molts dels conceptes clau a Spring, com poden ser la inversió del control i la injecció de dependències.

Spring-WS és el projecte que ofereix Spring per desenvolupar i desplegar serveis web SOAP.

JAX-WS no pressuposa un model de desenvolupament per als serveis web i el desenvolupador pot triar tant una estratègia *Code First* (també anomenada *Contract Last*) com una estratègia *Contract First*, tot i que el més habitual és utilitzar l'estratègia *Code First*.

Spring-WS, en canvi, només suporta la creació de serveis web SOAP seguint una estratègia *Contract First*.

Per què 'Contract First'?

Si voleu saber les motivacions que té Spring per suportar tan sols una estratègia *Contract First* ho podeu fer consultant aquesta URL: bit.ly/2niObVu.

WSDL és un document XML que descriu un servei web SOAP. Descriu on es localitza un servei, quines operacions proporciona, el format dels missatges que han de ser intercanviats i com cal cridar el servei.

Quan parlem de serveis web, una estratègia *Code First* vol dir crear primer les classes Java que implementaran el servei i, a partir d'aquestes, generar els documents WSDL de definició del servei.

Una estratègia *Contract First* vol dir exactament el contrari: fer primer la definició del servei abans d'implementar-lo. Això vol dir descriure els paràmetres i tipus de retorn del servei amb XSD (de l'anglès *XML Schema Definitions*), després utilitzar aquest XSD per generar el document WSDL que serà el contracte públic del servei i, finalment, generar les classes Java que implementaran el servei d'acord amb aquest contracte.

L'elecció entre JAX-WS i Spring-WS a l'hora de desenvolupar serveis web SOAP amb Java no és senzilla i no hi ha cap de les dues tecnologies que sigui una clara guanyadora; alguns dels motius per utilitzar Spring-WS són:

- Per disseny, una estratègia *Contract First* promou un baix acoblament entre el contracte i la implementació del servei web.
- Suport més ampli amb les llibreries de tractament XML.
- Suport més ampli i flexible en els mapatges entre XML i objectes Java.
- Permet reutilitzar tot el coneixement que tingueu en Spring.
- Suporta WS-Security i permet integrar-ho amb Spring Security.

1.1 Escrivint un servei web SOAP de consulta de dades d'empreses amb Spring-WS

Veurem els conceptes referents a la definició de serveis web SOAP amb Spring-WS desenvolupant un servei web SOAP amb Spring-WS que permeti consultar les dades d'una empresa per CIF.

Spring-WS ens obliga a utilitzar una estratègia *Contract First* per crear serveis web SOAP, això vol dir que primer descriurem els paràmetres i tipus de retorn del servei amb XSD, després utilitzarem aquest XSD per generar el document WSDL que serà el contracte públic del servei i, finalment, generarem les classes Java que implementaran el servei d'acord amb aquest contracte.

> Quan desenvolupeu un servei amb una estratègia *Contract First* us heu de centrar a **concretar el XML** que definirà el servei, el codi Java que ho implementarà passa a segon terme!

1.1.1 Creació i configuració inicial del projecte

El projecte ja té el pom.xml preparat per poder desenvolupar el servei web amb Spring-WS; concretament, s'hi ha afegit aquestes dues dependències:

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

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-ws</artifactId>
</dependency>
<dependency>
  <groupId>wsdl4j</groupId>
  <artifactId>wsdl4j</artifactId>
</dependency>
```

Farem el desplegament del servei web creant una aplicació executable. Per fer el desplegament creant una aplicació executable que s'executarà al contenidor de *servlets* que Spring porta incorporat cal crear una classe Java anotada amb `@SpringBootApplication` amb un mètode `main` que cridi el mètode `run` de `SpringApplication`. Per fer-ho ja us hem creat la classe `Application` al paquet `cat.xtec.ioc.service.impl` amb el següent codi:

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.

```java
package cat.xtec.ioc.service.impl;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

El fet d'anotar la classe amb `@SpringBootApplication` afegeix tota la configuració necessària per tal que l'aplicació pugui funcionar de forma autònoma.

1.1.2 Creació del servei web SOAP

El servei web SOAP que volem crear és molt senzill, tan sols ens ha de permetre consultar la informació d'una empresa d'un repositori d'empreses.

Com que crearem el servei web seguint una estratègia *Contract First*, el primer que ens cal fer és crear el document XSD que definirà el model de domini. El model de domini es defineix amb un fitxer d'esquema XML que després Spring-WS s'encarregarà d'exportar a WSDL.

La part principal del nostre model del domini serà l'entitat `Company`, que representarà les dades d'una empresa i la modelarem amb un document XSD. Per fer-ho creeu un fitxer anomenat company.xsd al directori */src/main/resources/* amb el següent contingut:

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cat.xtec.ioc/domain/company"
           targetNamespace="http://cat.xtec.ioc/domain/company"
               elementFormDefault="qualified">

    <xs:complexType name="company">
        <xs:sequence>
            <xs:element name="cif" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="employees" type="xs:int"/>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="web" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Fixeu-vos que es defineix un tipus XML complex per construir el model del domini; en aquest cas, un objecte `company` que conté el CIF, el nom, el nombre de treballadors, el correu i la pàgina web de l'empresa.

Modularitat en XSD

Es podria definir l'entitat del domini i les operacions en un únic document XSD, però és molt més modular fer-ho en documents separats i utilitzar la capacitat de fer importacions que proporciona XSD.

Un cop definida la nostra entitat del domini ens cal concretar els tipus de les peticions i les respostes del servei web que permetrà consultar la informació de l'empresa per CIF. Per fer-ho creeu un fitxer anomenat companyoperations.xsd al directori */src/main/resources/* amb el següent contingut:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cat.xtec.ioc/domain/company/services"
           targetNamespace="http://cat.xtec.ioc/domain/company/services"
               xmlns:company="http://cat.xtec.ioc/domain/company"
               elementFormDefault="qualified">
    <xs:import namespace="http://cat.xtec.ioc/domain/company" schemaLocation="company.xsd"/>
    <xs:element name="getCompanyRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="cif" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="getCompanyResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="company" type="company:company"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

Fixeu-vos que:

- S'importa la definició del tipus complex `company` que ha de tornar la resposta.
- Es defineix la petició com un tipus XML complex que conté la cadena on passarem el CIF de l'empresa de la qual volem consultar les dades.
- Es defineix la resposta també com un tipus XML complex que conté la informació de l'empresa consultada.

Per tal d'utilitzar els tipus que hem definit als fitxers XSD ens cal generar les classes Java per a aquests tipus. Spring-WS serà capaç de generar la classe Java `Company` que representa el model de domini i les classes Java que modelaran la petició (`GetCompanyRequest`) i la resposta (`GetCompanyResponse`) del servei web.

JAXB (de l'anglès *Java Architecture for XML Binding*) és un bastiment que permet fer la conversió entre documents XML i objectes Java, i a l'inrevés.

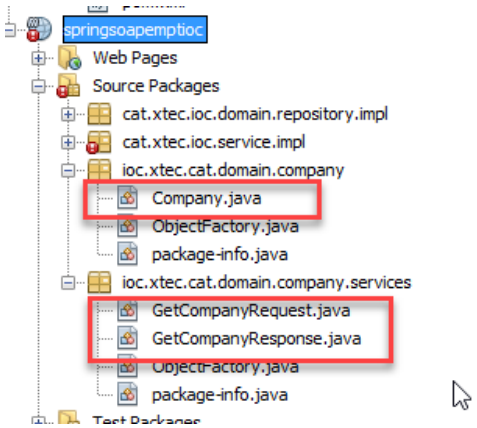
Aquesta és una part fonamental als serveis web SOAP: la conversió dels missatges SOAP d'XML cap a Java i a l'inrevés. Aquesta tasca no seria senzilla si l'haguéssiu de fer a mà, però Spring ho fa fàcil utilitzant el bastiment (*framework*, en anglès) JAXB, i ho farem en temps de construcció del projecte.

Això es pot fer de diverses formes; una manera senzilla és que ho faci el *plugin* de Maven `jaxb-maven-plugin` en temps de construcció del projecte. Afegiu les següents línies al fitxer pom.xml:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <version>1.6</version>
    <executions>
        <execution>
            <id>xjc</id>
            <goals>
                <goal>xjc</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <schemaDirectory>${project.basedir}/src/main/resources/</
            schemaDirectory>
        <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
        <clearOutputDir>false</clearOutputDir>
    </configuration>
</plugin>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual i després feu *Clean and Build*, també al menú contextual de NetBeans, veureu com a la fase de generació el *plugin* `xjc` ha generat, a partir dels documents XSD de definició de servei, la classe Java que modela el domini i les classes Java que modelen la petició i la resposta del servei web (vegeu la figura 3.1).

FIGURA 1.1. Classes Java generades




```
--- jaxb2-maven-plugin:1.6:xjc (xjc) @ springsoapemptioc ---
Generating source...
Analizando un esquema...
Compilando un esquema...
ioc\xtec\cat\domain\company\services\GetCompanyRequest.java
ioc\xtec\cat\domain\company\services\GetCompanyResponse.java
ioc\xtec\cat\domain\company\services\ObjectFactory.java
ioc\xtec\cat\domain\company\services\package-info.java
ioc\xtec\cat\domain\company\Company.java
ioc\xtec\cat\domain\company\ObjectFactory.java
ioc\xtec\cat\domain\company\package-info.java
```

Ara toca crear el repositori d'on el servei ha de consultar la informació de les empreses. En el nostre cas, per simplicitat, farem servir un repositori *in memory*

que tindrà una llista precarregada d'empreses i un mètode `findCompany` que permet consultar una de les empreses per CIF. Òbviament, en un projecte real la definició del servei serà molt més complexa!

Per fer-ho creeu una classe Java anomenada `InMemoryCompanyRepository` al paquet `cat.xtec.ioc.domain.repository.impl` amb el següent codi:

```
package cat.xtec.ioc.domain.repository.impl;

import ioc.xtec.cat.domain.company.Company;
import java.util.ArrayList;
import java.util.List;
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Repository;
import org.springframework.util.Assert;

@Repository
public class InMemoryCompanyRepository {

    private static final List<Company> companies = new ArrayList<Company>();

    @PostConstruct
    public void initData() {
        Company oracle = new Company();
        oracle.setCif("XXXXXXXX");
        oracle.setName("Oracle");
        oracle.setEmployees(5000);
        oracle.setEmail("oracle@oracle.com");
        oracle.setWeb("http://www.oracle.com");

        companies.add(oracle);

        Company ms = new Company();
        ms.setCif("YYYYYYYY");
        ms.setName("Microsoft");
        ms.setEmployees(10000);
        ms.setEmail("microsoft@microsoft.com");
        ms.setWeb("http://www.microsoft.com");

        companies.add(ms);

        Company redhat = new Company();
        redhat.setCif("ZZZZZZZZ");
        redhat.setName("Red Hat");
        redhat.setEmployees(2000);
        redhat.setEmail("redhat@redhat.com");
        redhat.setWeb("http://www.redhat.com");
        companies.add(redhat);
    }

    public Company findCompany(String cif) {
        Assert.notNull(cif);
        Company result = null;
        for (Company company : companies) {
            if (cif.equals(company.getCif())) {
                result = company;
            }
        }

        return result;
    }
}
```

Amb tot això ja estem preparats per crear l'*endpoint* del servei web SOAP que tornarà la informació de les empreses. Noteu que, per sota, hi ha un *servlet* que s'encarrega de recollir les peticions SOAP que vagin al servei i les envia a l'*endpoint* per tal de processar-les.

Creeu una classe Java anomenada `CompanyEndpoint` al paquet `cat.xtec.ioc.service.impl` amb el següent codi:

```java
package cat.xtec.ioc.service.impl;

import cat.xtec.ioc.domain.repository.impl.InMemoryCompanyRepository;
import ioc.xtec.cat.domain.company.services.GetCompanyRequest;
import ioc.xtec.cat.domain.company.services.GetCompanyResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;

@Endpoint
public class CompanyEndpoint {

    private static final String NAMESPACE_URI = "http://cat.xtec.ioc/domain/
        company/services";

    private InMemoryCompanyRepository companyRepository;

    @Autowired
    public CompanyEndpoint(InMemoryCompanyRepository companyRepository) {
        this.companyRepository = companyRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCompanyRequest")
    @ResponsePayload
    public GetCompanyResponse getCompany(@RequestPayload GetCompanyRequest
        request) {
        GetCompanyResponse response = new GetCompanyResponse();
        response.setCompany(companyRepository.findCompany(request.getCif()));

        return response;
    }
}
```

Fixeu-vos que:

- Hem injectat el repositori d'empreses al constructor del servei amb Spring.
- Hem anotat la classe amb `@Endpoint` per indicar a Spring que aquesta classe correspon a un *endpoint* d'un servei web SOAP i tindrà mètodes que serviran peticions SOAP. `@Endpoint` és una versió especialitzada de l'anotació `@Component`.
- TARGET_NAMESPACE representa l'espai de noms que heu definit anteriorment al document XSD. Es fa servir per mapar les peticions a mètodes específics de l'*endpoint*.
- Hem anotat el mètode `getCompany` amb l'anotació `@PayloadRoot` on es defineix l'espai de noms i el mètode que servirà les peticions de cerca d'empreses per CIF.
- El missatge SOAP que consulti una empresa per CIF haurà de fer referència al `PayloadRoot` especificat.
- Hem anotat el mètode amb `@ResponsePayload` per indicar que el mètode tornarà un objecte de tipus `GetCompanyResponse` amb la informació de l'empresa. Spring s'encarregarà de convertir aquest objecte a XML.

- El paràmetre del mètode l'hem anotat amb `@RequestPayload` per indicar que el missatge d'entrada serà de tipus `GetCompanyRequest`. Spring s'encarregarà de convertir el missatge XML d'entrada a un objecte d'aquest tipus.
- Tant el tipus de retorn com el paràmetre corresponen a classes generades automàticament en el procés de construcció a partir de la definició del fitxer d'esquema XML companyoperations.xsd.

Un cop creat l'*endpoint* del servei web SOAP tan sols ens queda configurar Spring per carregar tota la configuració. Això ho podeu fer de diverses maneres, ja sigui amb fitxers de configuració o amb anotacions a classes Java de configuració. A l'exemple utilitzarem aquesta darrera tècnica.

Creeu una classe Java anomenada `WebServiceConfig` al paquet `cat.xtec.ioc.service.impl` amb el següent codi:

```
1  package cat.xtec.ioc.service.impl;
2
3  import org.springframework.boot.web.servlet.ServletRegistrationBean;
4  import org.springframework.context.ApplicationContext;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.ComponentScan;
7  import org.springframework.context.annotation.Configuration;
8  import org.springframework.core.io.ClassPathResource;
9  import org.springframework.ws.config.annotation.EnableWs;
10 import org.springframework.ws.config.annotation.WsConfigurerAdapter;
11 import org.springframework.ws.transport.http.MessageDispatcherServlet;
12 import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
13 import org.springframework.xml.xsd.SimpleXsdSchema;
14 import org.springframework.xml.xsd.XsdSchema;
15
16 @EnableWs
17 @Configuration
18 @ComponentScan("cat.xtec.ioc.service.imp, cat.xtec.ioc.domain.repository.impl")
19 public class WebServiceConfig extends WsConfigurerAdapter {
20
21     @Bean
22     public ServletRegistrationBean messageDispatcherServlet(ApplicationContext
           applicationContext) {
23         MessageDispatcherServlet servlet = new MessageDispatcherServlet();
24         servlet.setApplicationContext(applicationContext);
25         servlet.setTransformWsdlLocations(true);
26         return new ServletRegistrationBean(servlet, "/soapws/*");
27     }
28
29     @Bean(name = "companyoperations")
30     public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema
           companiesSchema) {
31         DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition
               ();
32         wsdl11Definition.setPortTypeName("CompaniesPort");
33         wsdl11Definition.setLocationUri("/soapws");
34         wsdl11Definition.setTargetNamespace("http://cat.xtec.ioc/domain/company
               /services");
35         wsdl11Definition.setSchema(companiesSchema);
36         return wsdl11Definition;
37     }
38
39     @Bean
40     public XsdSchema companiesSchema() {
41         return new SimpleXsdSchema(new ClassPathResource("companyoperations.xsd
               "));
42     }
```

```
43
44      @Bean(name = "company")
45      public XsdSchema companySchema() {
46          return new SimpleXsdSchema(new ClassPathResource("company.xsd"));
47      }
48  }
```

Fixeu-vos que:

- Spring fa servir un *servlet* especial de tipus `MessageDispatcherServlet` per servir peticions SOAP, i cal que el registreu al `ApplicationContext` de l'aplicació.
- El mètode `defaultWsdl11Definition` s'encarrega de configurar el document WSDL de definició del servei web a partir del document XSD especificat.
- L'ús de `DefaultWsdl11Definition` permet la generació automàtica del document WSDL.
- El mètode `defaultWsdl11Definition` també defineix l'URI i l'espai de noms que caldrà que utilitzeu per cridar el servei web i que estarà disponible en el document WSDL de definició del servei.

Aquesta és tota la feina que ens cal fer per implementar el servei web de consulta de dades d'empreses; ara tan sols ens manca fer el desplegament 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 desplegament del servei web es pot fer de diverses maneres, entre les quals:

- Desplegant l'aplicació Java EE que el conté mitjançant els mecanismes normals de desplegament de qualsevol aplicació Java EE.
- Creant una aplicació executable que s'executarà amb un contenidor de *servlets* Tomcat que Spring porta incorporat.

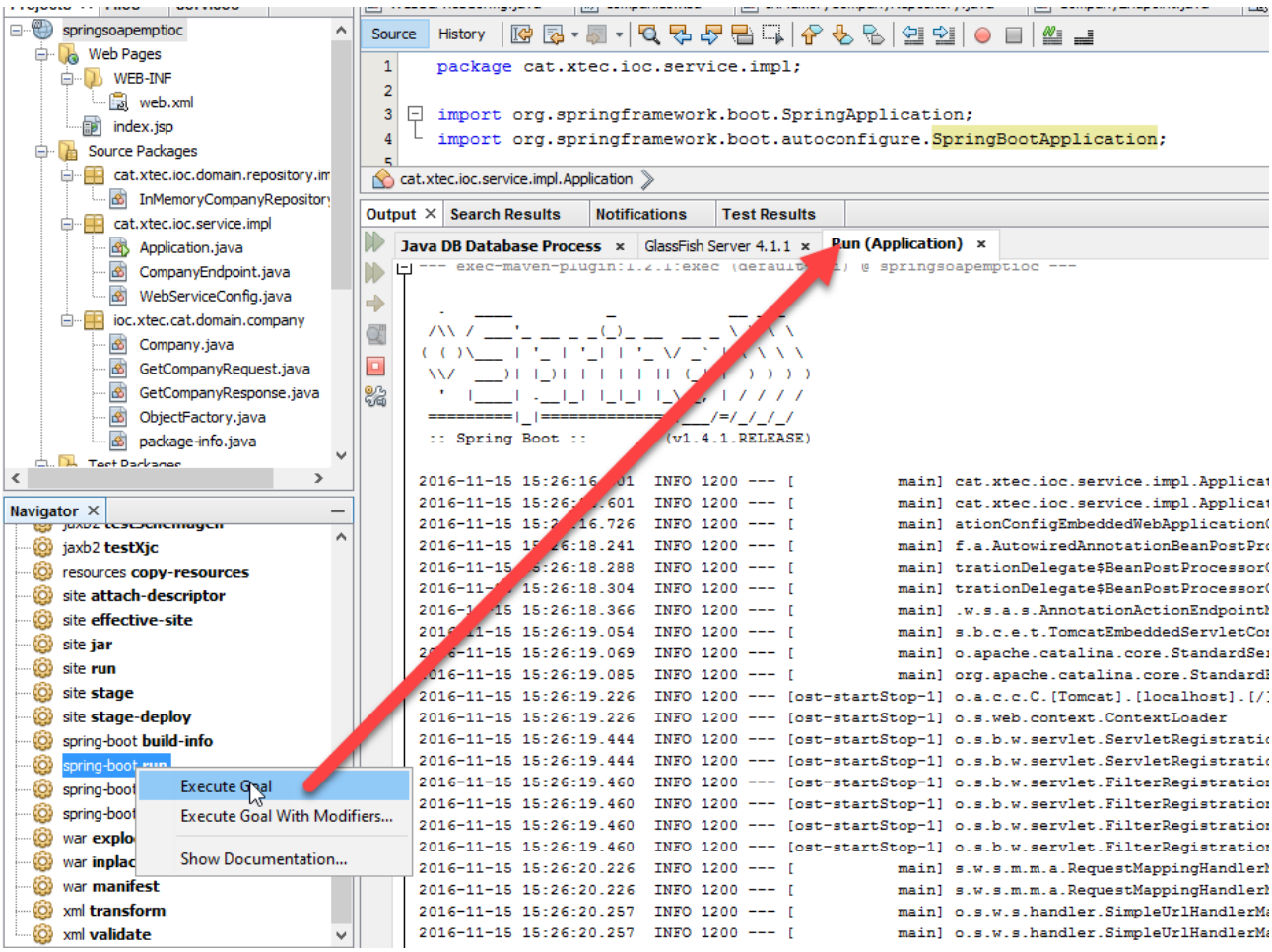
Prerequisits per a l'execució

Abans d'executar la classe cal que us assegureu de tenir el servidor Glassfish parat, ja que, si no, no podrà arrencar el servei Tomcat on desplegarem l'aplicació per un conflicte amb els ports. Tant Glassfish com Tomcat estan configurats per utilitzar el port 8080 per defecte.

Farem el desplegament del servei web creant una aplicació executable. Per fer el desplegament creant una aplicació executable que s'executarà al contenidor de *servlets* que Spring porta incorporat tan sols ens cal una classe Java anotada amb *@SpringBootApplication* amb un mètode `main` que cridi el mètode `run` de `SpringApplication`. Al projecte inicial ja teniu una classe `Application` al paquet `cat.xtec.ioc.service.impl` que permet que l'aplicació pugui funcionar de forma autònoma.

Poseu-vos damunt la classe `Application`, feu clic amb el botó dret del ratolí i seleccioneu l'opció *Run* (també podeu executar el *goal* de Maven `spring-boot:run`). Veureu que apareix una finestra a NetBeans amb l'execució de l'aplicació amb Spring (vegeu la figura 3.2).

FIGURA 1.2. Execució de l'aplicació amb el servei web



Podeu provar que el desplegament ha anat bé consultant el document WSDL generat a l'enllaç localhost:8080/soapws/companyoperations.wsdl amb qualsevol navegador, i heu de veure el document WSDL de definició del servei generat:

```
<wsdl:definitions targetNamespace="http://cat.xtec.ioc/domain/company/services"
    >
    <wsdl:types></wsdl:types>
    <wsdl:message name="getCompanyRequest"></wsdl:message>
    <wsdl:message name="getCompanyResponse"></wsdl:message>
    <wsdl:portType name="CompaniesPort"></wsdl:portType>
    <wsdl:binding name="CompaniesPortSoap11" type="tns:CompaniesPort"></
        wsdl:binding>
    <wsdl:service name="CompaniesPortService"></wsdl:service>
</wsdl:definitions>
```

1.1.4 Prova del servei web

Si tot ha anat bé ja teniu el servei web desplegat i a punt per provar-lo. Però com ho fem? Quin format han de tenir els missatges?

Quan es desplega el 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 fitxer WSDL 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 on hi ha aquestes operacions. Aquesta estructura permet reutilitzar la part abstracta del document.

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>`:

```
<wsdl:types>
    <xs:schema elementFormDefault="qualified" targetNamespace="http://cat.xtec.
        ioc/domain/company/services">
        <xs:import namespace="http://cat.xtec.ioc/domain/company"
            schemaLocation="company.xsd"/>
        <xs:element name="getCompanyRequest">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="cif" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="getCompanyResponse">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="company" type="company:company"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:schema>
</wsdl:types>
```

Per exemple, si us hi fixeu, defineix que les peticions a l'operació `getCompanyRequest` reben un paràmetre de tipus `string` i a les respostes es torna un tipus de dades complex anomenat `company` que està format per un *int* i quatre *string* (*cif*, *name*, *employees*, *email* i *web*).

La definició del tipus complex `company` s'importa d'un fitxer XSD separat que té la definició del model del domini.

```
<xs:schema elementFormDefault="qualified" targetNamespace="http://cat.xtec.ioc/
    domain/company">
    <xs:complexType name="company">
        <xs:sequence>
            <xs:element name="cif" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="employees" type="xs:int"/>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="web" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

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:

```
<wsdl:message name="getCompanyRequest">
    <wsdl:part element="tns:getCompanyRequest" name="getCompanyRequest">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="getCompanyResponse">
    <wsdl:part element="tns:getCompanyResponse" name="getCompanyResponse">
    </wsdl:part>
</wsdl:message>
```

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

```
<wsdl:portType name="CompaniesPort">
    <wsdl:operation name="getCompany">
        <wsdl:input message="tns:getCompanyRequest" name="getCompanyRequest">
        </wsdl:input>
        <wsdl:output message="tns:getCompanyResponse" name="getCompanyResponse"
            >
        </wsdl:output>
    </wsdl:operation>
</wsdl:portType>
```

Veiem, per exemple, que l'operació `getCompanyRequest` rep com a paràmetre d'entrada un `tns:getCompanyRequest` i que retorna un `tns:getCompanyResponse`. Aquests tipus de dades han estat definides completament en el document XSD que hem creat per definir el servei.

A la part concreta hi tenim els següents elements:

- `binding`
- `service`

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

```
<wsdl:binding name="CompaniesPortSoap11" type="tns:CompaniesPort">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
        http"/>
    <wsdl:operation name="getCompany">
```

```
        <soap:operation soapAction=""/>
        <wsdl:input name="getCompanyRequest">
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="getCompanyResponse">
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

L'element `<service>`, per la seva part, defineix el lloc físic on hi ha el servei web (adreça URL).

```
<wsdl:service name="CompaniesPortService">
    <wsdl:port binding="tns:CompaniesPortSoap11" name="CompaniesPortSoap11">
        <soap:address location="http://localhost:8080/soapws"/>
    </wsdl:port>
</wsdl:service>
```

En el nostre cas, el servei web es a l'URL localhost:8080/soapws.

Amb tot això ja sabem el format dels missatges SOAP que cal enviar per accedir al servei web i on cal enviar-los; creeu un fitxer anomenat request.xml, que representarà el missatge SOAP de petició, amb el següent contingut:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:gs="http://cat.xtec.ioc/domain/company/services">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:getCompanyRequest>
         <gs:cif>XXXXXXXX</gs:cif>
      </gs:getCompanyRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

I ja tan sols ens queda fer una petició al servei web per provar-ho. La nostra proposta és que feu servir cURL, que és una eina molt útil per fer peticions HTTP de diferents tipus i que és multiplataforma. Si feu servir Linux possiblement ja la tingueu instal·lada al sistema, i en cas que utilitzeu Windows la podeu descarregar del següent enllaç: curl.haxx.se/download.html.

La sintaxi de cURL per fer peticions és molt senzilla; per exemple, per consultar les dades de l'empresa que té per CIF XXXXXXXX feu un `command prompt`:

```
curl --header "content-type: text/xml" -d @request.xml http://localhost:8080/
    soapws/
```

Indiquem que la petició s'envia al fitxer request.xml amb el tipus MIME *text/xml*. Aquest fitxer és el que acabeu de crear amb el missatge SOAP de petició, i l'URL és l'URL on escolta el servei web.

Si executeu la comanda anterior i tot ha anat correctament, veureu la resposta del servei web amb les dades de l'empresa que té per CIF XXXXXXXX:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns3:getCompanyResponse xmlns:ns2="http://cat.xtec.ioc/domain/company"
            xmlns:ns3="http://cat.xtec.ioc/domain/company/services">
```

```
            <ns3:company>
                <ns2:cif>XXXXXXXX</ns2:cif>
                <ns2:name>Oracle</ns2:name>
                <ns2:employees>5000</ns2:employees>
                <ns2:email>oracle@oracle.com</ns2:email>
                <ns2:web>http://www.oracle.com</ns2:web>
            </ns3:company>
        </ns3:getCompanyResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

I això és tot! Hem vist els passos que us cal fer per desenvolupar, desplegar i provar un servei web SOAP amb Spring-WS seguint una estratègia *Contract First*.

1.2 Fent servir la consulta de dades d'empreses des d'una aplicació Java 'stand-alone'

Crearem una aplicació Java que consumeixi el servei web de consulta de dades d'empreses i ho farem desenvolupant un client Java *stand-alone* que accedirà al servei web amb Spring-WS.

Els passos generals per fer un client amb Spring-WS són els següents:

1. Generar els artefactes necessaris per poder consumir el servei web a partir de la definició del servei en format WSDL.

2. Codificar la classe que farà la crida al servei web heretant de la classe `WebServiceGatewaySupport` que proporciona Spring-WS.

3. Compilar i executar el client.

El procediment que expliquem **és genèric i serveix per a qualsevol servei web SOAP**, estigui o no codificat amb Spring-WS. Només ens cal la definició WSDL del servei.

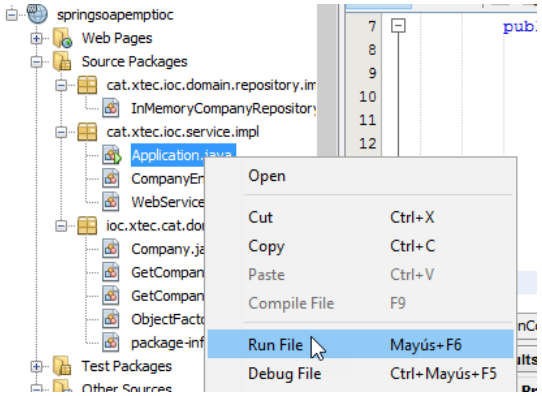
El client que farem consultarà les dades de les empreses d'un servei web SOAP desenvolupat amb Spring-WS. Descarregueu el codi del servei web que consultarem en el següent i importeu-lo a NetBeans.

Quan tingueu el projecte importat a NetBeans cal que desplegueu el servei web que conté; per exemple, creant una aplicació executable que s'executarà amb un contenidor de *servlets* Tomcat que Spring porta incorporat. Ho podeu fer de diverses maneres, una és fent clic amb el botó dret damunt de la classe `Application` del paquet `cat.xtec.ioc.client` i seleccionant *Run File* (vegeu la figura 1.3).

Prerequisits per a l'execució

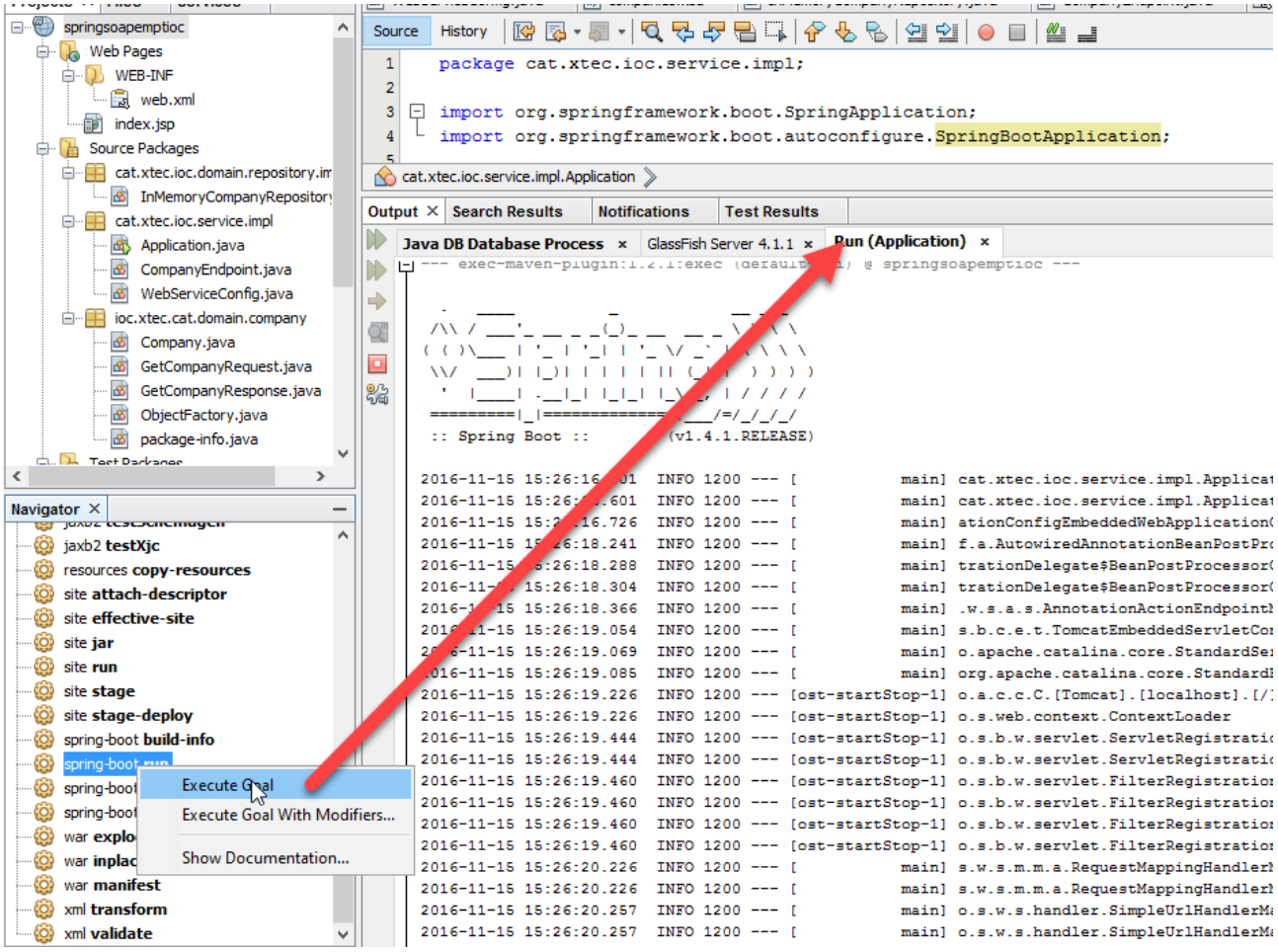
Abans d'executar la classe cal que us assegureu de tenir el servidor Glassfish parat, ja que, si no, no podrà arrencar el servei Tomcat on desplegarem l'aplicació per un conflicte amb els ports, ja que tant Glassfish com Tomcat estan configurats per utilitzar el port 8080 per defecte.

Figura 1.3. Execució del servei web SOAP de consulta d'empreses



També ho podeu fer executant el *goal* de Maven `spring-boot:run`. Amb qualsevol de les dues opcions veureu que apareix una finestra a NetBeans amb l'execució de l'aplicació amb Spring (vegeu la figura 1.4).

Figura 1.4. Finestra d'execució del servei web SOAP de consulta d'empreses



Podeu provar que el desplegament ha anat bé consultant el document WSDL generat a l'enllaç localhost:8080/soapws/companyoperations.wsdl amb qualsevol navegador, i heu de veure el document WSDL de definició del servei.

Després de desplegar el servei web que volem consultar ens cal crear el client Java que el consultarà amb Spring-WS.

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

El primer que cal fer és generar els artefactes necessaris per poder consumir el servei web des d'aquest client a partir de la seva definició en format WSDL. Això es pot fer de diverses formes, i una manera senzilla és que ho faci un *plugin* de Maven en temps de construcció del projecte.

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.

Afegiu les següents línies al fitxer pom.xml i feu un *Reload POM* del projecte:

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <version>0.13.1</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <schemaLanguage>WSDL</schemaLanguage>
        <generatePackage>cat.xtec.ioc.domain.companies.wsdl</generatePackage>
        <schemas>
            <schema>
                <url>http://localhost:8080/soapws/companyoperations.wsdl</url>
            </schema>
        </schemas>
    </configuration>
</plugin>
```

Aquest *plugin* generarà automàticament les classes necessàries per consultar el servei web de consulta de dades d'empreses en temps de compilació i les deixarà al paquet `cat.xtec.ioc.domain.companies.wsdl`. Fixeu-vos també que estem especificant en la configuració del *plugin* la localització del document WSDL amb la definició del servei.

Per tal que el projecte funcioni cal que l'executeu amb JDK 1.8; podeu canviar el JDK a les propietats del projecte, a l'apartat *Build* / *Compile* / *Java Plattform* .

Si feu clic amb el botó dret damunt el projecte "Springsoapempclientioc" i feu *Clean and Build* veureu que el *plugin* genera, entre d'altres, les següents classes a partir del document WSDL de definició de servei:

- `Company.java`
- `GetCompanyRequest.java`
- `GetCompanyResponse.java`

Un cop generats els artefactes necessaris cal que creeu la classe Java que farà la crida al servei web. Creeu-la, per exemple, al paquet `cat.xtec.ioc.client` i anomeneu-la `CompaniesClient`:

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

import cat.xtec.ioc.domain.companies.wsdl.GetCompanyRequest;
import cat.xtec.ioc.domain.companies.wsdl.GetCompanyResponse;
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;
import org.springframework.ws.soap.client.core.SoapActionCallback;

public class CompaniesClient extends WebServiceGatewaySupport {

    public GetCompanyResponse getCompanyInformation(String cif) {
        GetCompanyRequest request = new GetCompanyRequest();
```

```
        request.setCif(cif);
        GetCompanyResponse response = (GetCompanyResponse)
            getWebServiceTemplate()
                .marshalSendAndReceive(request,
                        new SoapActionCallback("http://localhost:8080/soapws/
                            getCompanyResponse"));

        return response;
    }

}
```

Aquesta classe hereta d'una classe que proporciona Spring anomenada `WebServiceGatewaySupport` i utilitza les classes generades per fer una crida al servei web mitjançant un *template* que proporciona la seva classe base.

També ens cal crear una classe de configuració per indicar a Spring que faci servir JAXB per fer les transformacions de missatge SOAP a objecte Java, i a la inversa. Creeu aquesta classe Java també al paquet `cat.xtec.ioc.client` i anomeneu-la `ClientAppConfig`:

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

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;

@Configuration
public class ClientAppConfig {

    @Bean
    public Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("cat.xtec.ioc.domain.companies.wsdl");
        return marshaller;
    }

    @Bean
    public CompaniesClient companiesClient(Jaxb2Marshaller marshaller) {
        CompaniesClient client = new CompaniesClient();
        client.setDefaultUri("http://localhost:8080/soapws/companies.wsdl");
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
    }
}
```

Finalment, creeu una classe Java, també al paquet `cat.xtec.ioc.client`, i l'anomeneu `RunCompaniesClient`, amb el següent codi:

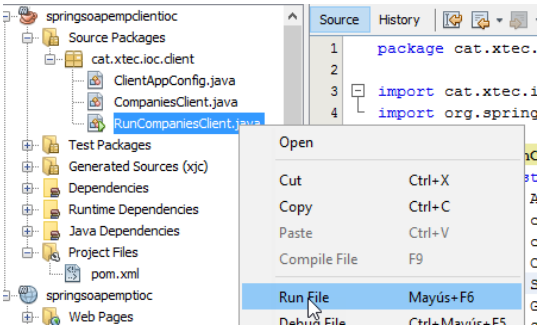
```
package cat.xtec.ioc.client;

import cat.xtec.ioc.domain.companies.wsdl.GetCompanyResponse;
import org.springframework.context.annotation.
    AnnotationConfigApplicationContext;

public class RunCompaniesClient {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
            AnnotationConfigApplicationContext();
        ctx.register(ClientAppConfig.class);
        ctx.refresh();
        CompaniesClient companiesClient = ctx.getBean(CompaniesClient.class);
```

```
13          System.out.println("For Company XXXXXXXX");
14          GetCompanyResponse response = companiesClient.getCompanyInformation("
                XXXXXXXX");
15          System.out.println("CIF: " + response.getCompany().getCif());
16          System.out.println("Name: " + response.getCompany().getName());
17          System.out.println("Num. employees: " + response.getCompany().
                getEmployees());
18          System.out.println("Email: " + response.getCompany().getEmail());
19          System.out.println("Web: " + response.getCompany().getWeb());
20      }
21  }
```

Aquesta classe té un mètode `main` que s'encarrega de crear un objecte de tipus `CompaniesClient` amb la configuració especificada a `ClientAppConfig`, fer una petició al servei web de consulta de dades d'empreses per recuperar les dades de l'empresa que té el CIF XXXXXXXX amb aquesta classe i mostrar els resultats per a la sortida estàndard.

Ara ja podem executar el client; per fer-ho, poseu-vos damunt de la classe `RunCompaniesClient` i feu *Run File* a NetBeans (vegeu la figura 2.4).

FIGURA 1.5. Execució del client del servei web SOAP de consulta d'empreses



I obtindreu per consola les dades de l'empresa que té el CIF XXXXXXXX:

```
CIF: XXXXXXXX
Name: Oracle
Num. employees: 5000
Email: oracle@oracle.com
Web: http://www.oracle.com
```

1.3 Què s'ha après?

Heu vist les bases per al desenvolupament dels serveis web SOAP amb Spring-WS i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web SOAP amb Spring-WS.
- La diferència entre *Contract First* i *Code First*.

- Les diferències bàsiques entre JAX-WS i Spring-WS.
- Desenvolupar, desplegar i provar un servei web SOAP amb Spring-WS.
- Desenvolupar i provar un client Java que consulti un servei web SOAP mitjançant Spring-WS.

2. Serveis web RESTful amb Spring. Escrivint serveis web

Explicarem, mitjançant exemples, els conceptes mes rellevants dels serveis web RESTful (de l'anglès *REpresentational State Transfer*) amb Spring MVC; aprendreu a crear-ne, a fer-ne el desplegament i a provar-los 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, 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.

L'especificació de Java EE 7 inclou l'API de JAX-RS (de l'anglès *Java API for RESTful Web Services*), que fa servir un conjunt d'anotacions per simplificar el desenvolupament, el desplegament i el consum de 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.

Spring, per la seva banda, no proporciona un projecte ni un mòdul específic per a la creació i el consum de serveis web RESTful, sinó que ho engloba com una capacitat més dins el projecte Spring MVC.

Spring MVC és el mòdul del projecte Spring que proporciona suport per crear i consumir serveis web basats en REST.

JAX-RS és una API centrada únicament i exclusivament en el desenvolupament de serveis web RESTful. Spring MVC és un mòdul del projecte Spring centrat en el desenvolupament web en general, i és en aquest marc que dóna complet suport tant al desenvolupament d'aplicacions web com al desenvolupament de serveis web RESTful.

Les capacitats que ofereix Spring MVC per desenvolupar serveis web RESTful són una continuació del model més general de programació que té Spring MVC. No hi ha, doncs, un *framework* o mòdul específic a Spring per desenvolupar serveis web RESTful.

Per escriure un servei web RESTful, tant amb JAX-RS com amb Spring, 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. Tant JAX-RS com Spring us estalvien tota aquesta feina proporcionant-vos un petit conjunt d'anotacions que us permetran desenvolupar còmodament serveis web RESTful.

Ens centrarem en les capacitats que proporciona Spring, i més concretament el mòdul Spring MCV, per donar suport a la creació i el consum de serveis web RESTful.

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 Spring MVC per fer-ho.

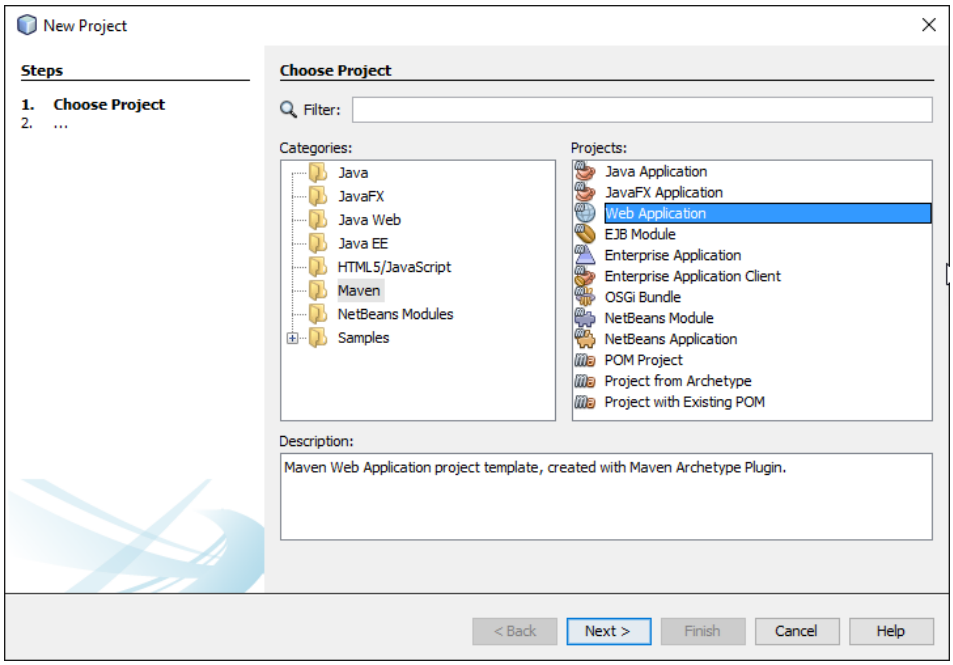
Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en l'enllaç que trobareu als annexos de la unitat, sempre és millor que aneu fent vosaltres tots els passos partint del projecte en l'estat inicial de l'apartat. Recordeu que podeu utilitzar la funció d'importar per carregar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

El servei web que farem tornarà *"Hello, World!!!"* si no li passem cap paràmetre; si li passem un paràmetre anomenat `name` ens tornarà una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

2.1.1 Creació i configuració inicial del projecte

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 3.1.

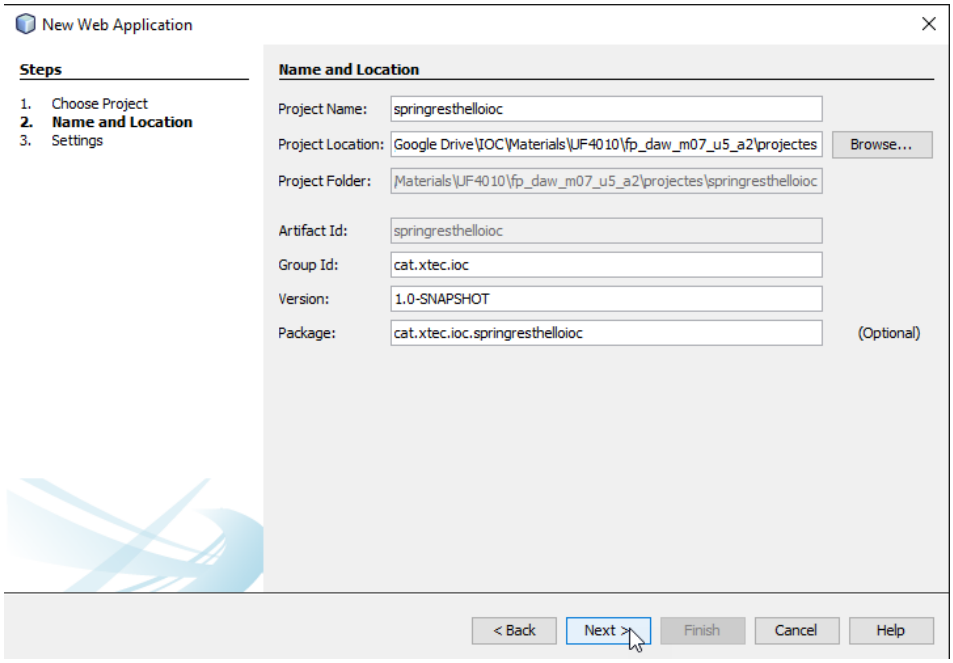
Figura 2.1. Creació de projectes a NetBeans



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

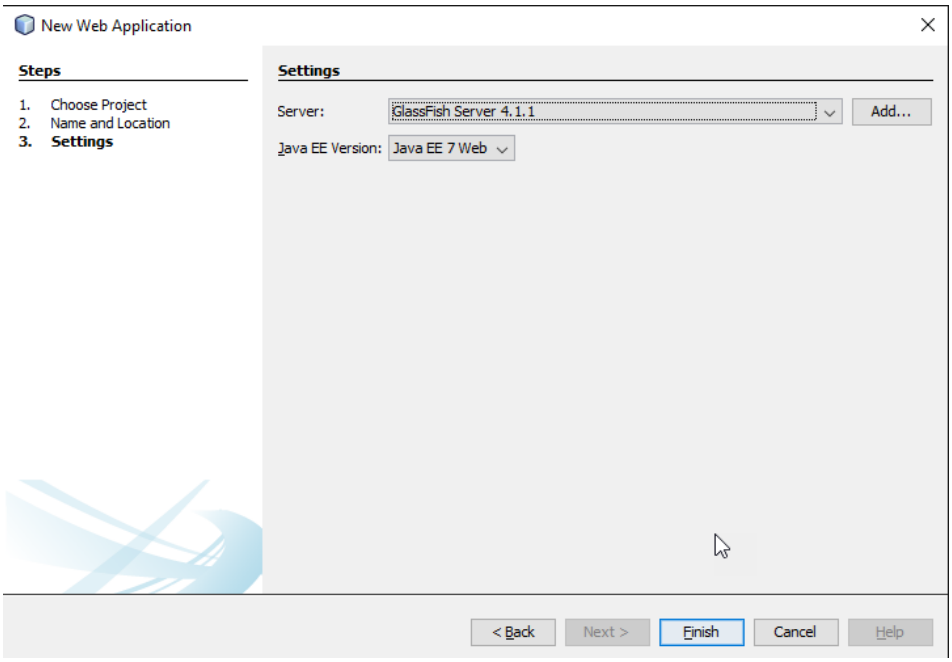
En la següent pantalla (vegeu la figura 3.2) triarem el nom del projecte (el podeu anomenar "Springresthelloioc") i el paquet per defecte on anirà el codi font; per exemple, `cat.xtec.ioc.springresthelloioc`.

Figura 2.2. Nom del nou projecte



En la següent pantalla (vegeu la figura 1.3) de l'assistent deixeu els valors per defecte i polseu *Finish*.

FIGURA 2.3. Configuració del nou projecte



Un cop creat el projecte cal que modifiqueu el pom.xml per afegir les dependències cap a Spring; afegiu les següent línies al fitxer pom.xml:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.0.3.RELEASE</version>
</dependency>
```

També heu de crear el fitxer web.xml i configurar el *servlet* `DispatcherServlet` per tal que Spring MVC serveixi les peticions web; creeu el fitxer web.xml a la carpeta *Web Pages/WEB-INF* amb el següent contingut:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/web-app_3_0.
                                 xsd">

    <display-name>Spring RESTful Hello World</display-name>

    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</
            servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/DispatcherServlet-servlet.xml</param-
                value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Ens cal també crear el fitxer de configuració per a Spring; li hem indicat que el fitxer s'anomenarà DispatcherServlet-servlet.xml i serà a la ruta */WEB-INF/spring/*. Creeu primer la carpeta */WEB-INF/spring/* i després el fitxer DispatcherServlet-servlet.xml amb el següent contingut:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
                               beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring
                               -context-4.0.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc
                               -4.0.xsd">
    <mvc:annotation-driven />
    <context:component-scan base-package="cat.xtec.ioc" />

    <bean class="org.springframework.web.servlet.view.
        InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el servei web RESTful amb Spring.

2.1.2 Creació del servei web RESTful

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

Aquest servei web ha d'**obtenir una representació del recurs** en format JSON, i per tant haurà de respondre el 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:

```
{"id":1,"content":" Hello, World!!!"}
```

I a les peticions d'aquest tipus:

```
GET http://localhost:8080/resthelloioc/hello?name=User
```

ha de respondre amb:

```
{"id":1,"content":"Hello, User!"}
```

JSON

A l'exemple us demanem que treballeu amb una representació JSON. La simplicitat, lleugeresa i facilitat de lectura fan ideals 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.

El servei web tornarà la informació en format JSON. A la sortida, el camp *id* és un identificador únic per a la salutació, i el camp *content* representa la salutació.

Un cop decidit el funcionament del nostre servei, el següent serà implementar la classe que modelarà la representació de la salutació, és a dir, la classe que modelarà el que torna el servei web.

Creeu una classe Java anomenada `Greeting` al paquet `cat.xtec.ioc.springresthelloioc.domain` amb el següent codi:

```
public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

Spring fa servir la llibreria Jackson per fer les transformacions entre el format JSON i la seva representació Java, i a l'inrevés.

El següent pas és crear el controlador Spring, que s'encarregarà de servir les peticions; per fer-ho, creeu una classe Java anomenada `GreetingController` al paquet `cat.xtec.ioc.springresthelloioc.controller` amb el següent codi:

```
package cat.xtec.ioc.springresthelloioc.controller;

import cat.xtec.ioc.springresthelloioc.domain.Greeting;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping(method =  RequestMethod.GET, value = "/hello")
```

```
17      public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!
            ") String name) {
18          return new Greeting(counter.incrementAndGet(),
19                              String.format(template, name));
20      }
21  }
```

El codi del controlador és molt simple, però porta algunes particularitats que cal remarcar; fixeu-vos que:

- Hem anotat la classe amb `@RestController` per marcar la classe com un controlador `REST` on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista. Aquesta és la diferència més gran entre un controlador MVC típic i un controlador per serveis web RESTful: el cos de la resposta HTTP es crea transformant un objecte del domini a JSON, i no es torna en format HTML.
- L'objecte `Greeting` que torna el mètode `greeting` cal que sigui transformat a JSON; Spring s'encarrega de fer això automàticament.
- Hem anotat el mètode `greeting` amb `@RequestMapping` indicant que les peticions a */hello* les servirem amb aquest mètode.
- Hem anotat el mètode `greeting` amb `RequestMethod.GET` per indicar que respondrà a les peticions HTTP que es facin mitjançant el verb `GET`. Si no ho haguéssim fet, el mètode respondria a qualsevol dels verbs HTTP.
- Hem anotat el paràmetre del mètode `greeting` amb `@RequestParam` per indicar que el paràmetre `name` de la petició HTTP s'ha de vincular amb el paràmetre `name` del mètode. El paràmetre és opcional i, si no ve a la petició HTTP, farem servir *"World!!!"* com a valor per defecte.
- En la implementació simplement creem i retornem l'objecte `Greeting`, que modela la salutació.
- Com que hi ha la llibreria Jackson al *classpath*, Spring tria la conversió de l'objecte `Greeting` a JSON per defecte.

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

2.1.3 Desplegament i prova 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.

Aquest procés és molt senzill, simplement cal que feu *Clean and Build* i després *Run* a NetBeans, i es farà el desplegament al servidor d'aplicacions que tingueu configurat per al projecte.

Si proveu ara accedint a l'URL localhost:8080/springresthelloioc/hello amb qualsevol navegador veureu que el servei web us torna la salutació en format JSON:

```
{"id":1,"content":"Hello, World!!!!"}
```

Proveu també especificant el paràmetre `name`, accedint a localhost:8080/springresthelloioc/hello?name=User, per exemple, i veureu que la salutació canvia:

```
{"id":2,"content":"Hello, User!"}
```

Amb això ja heu creat, configurat, desplegat i provat un servei web RESTful molt senzill amb Spring.

2.2 Testejant el servei web '"Hello, World!!!"'

Aprofitarem les capacitats que proporcions Spring, i més concretament Spring MVC, per crear un conjunt de tests que van a cavall entre un test unitari i un test d'integració. Aquests tests us permetran provar els serveis web sense necessitat de tenir-los desplegats i executant-se a un servidor d'aplicacions. Testejarem amb Spring el típic exemple de "*Hello, World!!!*".

2.2.1 Creació i configuració inicial del projecte

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

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

```
package cat.xtec.ioc.springresthelloioc.controller;

import cat.xtec.ioc.springresthelloioc.domain.Greeting;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping(method =  RequestMethod.GET, value = "/hello")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!
        ") String name) {
        return new Greeting(counter.incrementAndGet(),
```

```
19                                String.format(template, name));
20        }
21    }
```

Aquest servei web és un servei web RESTful desenvolupat amb Spring que torna "*Hello, World!!!*" si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

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.

El servei web torna la salutació en format JSON i respon a les peticions `GET` a */hello* amb:

```
1  {"id":1,"content":" Hello, World!!!"}
```

I a les peticions `GET` a */hello?name=User* amb:

```
1  {"id":1,"content":"Hello, User!"}
```

El test que escriurem utilitzarà les capacitats que té Spring MVC per fer test unitari dels *endpoints* configurats a un controlador RESTful. La principal diferència entre aquest tipus de tests i els tests d'integració és que en aquest cas **no ens cal tenir el servei web que volem provar desplegat i executant-se** a un servidor d'aplicacions.

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

Els tests unitaris amb el bastiment proporcionat per Spring MVC es basen en JUnit 4. 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 una *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.

2.2.2 Creació i execució dels tests unitaris

Ara toca començar a crear els tests d'integració per provar el servei web. Per fer-ho, creeu un nou paquet dins de *Test Packages* anomenat, per exemple, `cat.xtec.ioc.springresthelloioc.controller`, i una classe Java anomenada `GreetingControllerTest`:

```
1  package cat.xtec.ioc.springresthelloioc.controller;
2
3  import org.junit.Before;
4  import org.junit.Test;
```

```
5  import org.junit.runner.RunWith;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.http.MediaType;
8  import org.springframework.http.converter.HttpMessageConverter;
9  import org.springframework.http.converter.json.
       MappingJackson2HttpMessageConverter;
10 import org.springframework.mock.http.MockHttpOutputMessage;
11 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
12 import org.springframework.test.context.web.WebAppConfiguration;
13 import org.springframework.test.web.servlet.MockMvc;
14 import org.springframework.web.context.WebApplicationContext;
15
16 import java.io.IOException;
17 import java.nio.charset.Charset;
18 import java.util.Arrays;
19 import static junit.framework.Assert.assertNotNull;
20 import static org.hamcrest.CoreMatchers.is;
21
22 import org.springframework.boot.test.SpringApplicationConfiguration;
23 import static org.springframework.test.web.servlet.request.
       MockMvcRequestBuilders.*;
24 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
       .*;
25 import static org.springframework.test.web.servlet.setup.MockMvcBuilders.*;
26
27
28 @RunWith(SpringJUnit4ClassRunner.class)
29 @SpringApplicationConfiguration(classes = Application.class)
30 @WebAppConfiguration
31 public class GreetingControllerTest {
32     private MediaType contentType = new MediaType(MediaType.APPLICATION_JSON.
           getType(),
33             MediaType.APPLICATION_JSON.getSubtype(),
34             Charset.forName("utf8"));
35
36     private MockMvc mockMvc;
37
38     private HttpMessageConverter mappingJackson2HttpMessageConverter;
39
40     @Autowired
41     private WebApplicationContext webApplicationContext;
42
43     @Autowired
44     void setConverters(HttpMessageConverter<?>[] converters) {
45         this.mappingJackson2HttpMessageConverter = Arrays.asList(converters).
               stream()
46             .filter(hmc -> hmc instanceof MappingJackson2HttpMessageConverter)
47             .findAny()
48             .orElse(null);
49
50         assertNotNull("the JSON message converter must not be null",
51                 this.mappingJackson2HttpMessageConverter);
52     }
53
54     @Before
55     public void setup() throws Exception {
56         this.mockMvc = webAppContextSetup(webApplicationContext).build();
57
58     }
59
60
61     protected String json(Object o) throws IOException {
62         MockHttpOutputMessage mockHttpOutputMessage = new MockHttpOutputMessage
               ();
63         this.mappingJackson2HttpMessageConverter.write(
64                 o, MediaType.APPLICATION_JSON, mockHttpOutputMessage);
65         return mockHttpOutputMessage.getBodyAsString();
66     }
67 }
```

Fixeu-vos que:

- Anotem la classe amb `@RunWith(SpringJUnit4ClassRunner.class)` per indicar que es tracta d'un test unitari que s'executarà amb l'executor de JUnit que porta Spring.
- Anotem la classe amb `@SpringApplicationConfiguration(classes = Application.class)` per indicar que crearem una classe que carregui la configuració Spring. Aquesta és una de les diverses maneres de fer-ho, també es podria carregar la configuració amb fitxers XML.
- Anotem la classe amb `@WebAppConfiguration` per indicar a JUnit que es tracta d'un test unitari per a Spring MVC i que s'ha d'executar amb un context d'aplicació web i no d'aplicació *stand-alone*.
- Anotem un mètode anomenat `setup` amb `@Before` per indicar a JUnit que aquest mètode s'executi cada cop que es creï la classe de test. En aquest mètode creem un objecte de tipus `MockMvc` que serà el que ens proporcionarà tota la infraestructura necessària per fer els tests sense haver de desplegar l'aplicació a cap servidor d'aplicacions.
- El mètode `json` és un mètode que ens torna la representació en format JSON d'un objecte.
- S'han configurat tots els conversors que es requereixen per parsejar les peticions i les respostes.

Per tal que Spring pugui carregar la configuració cal una classe anomenada `Application` al paquet `cat.xtec.ioc.springresthelloioc.controller` amb el següent codi:

```
1  package cat.xtec.ioc.springresthelloioc.controller;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class Application {
8
9      public static void main(String[] args) {
10         SpringApplication.run(Application.class, args);
11     }
12 }
```

Aquesta classe ja la teniu creada al projecte de partida.

El primer test que farem serà un test que **comprovi que la petició *`/hello`* sense cap paràmetre torna la representació JSON de la salutació "*Hello, World!!!*"**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloWorldWithoutName` dins la classe `GreetingControllerTest` amb el següent codi:

```
1  @Test
2  public void greetingShouldReturnHelloWorldWithoutName() throws Exception {
3      mockMvc.perform(get("/hello"))
```

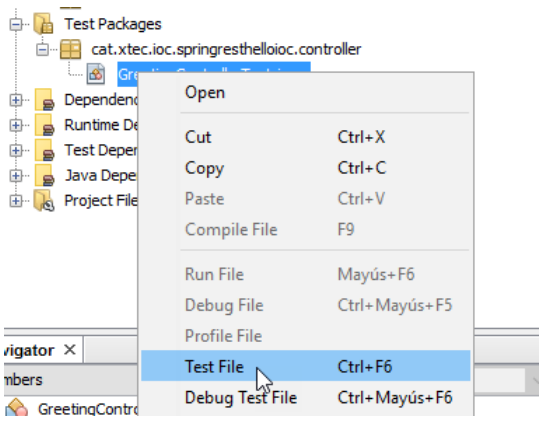
```
4               .andExpect(status().isOk())
5               .andExpect(content().contentType(contentType))
6               .andExpect(jsonPath("$.content", is("Hello, World!!!")));
7   }
```

Si analitzem el codi veiem diverses coses importants: la primera és que hem anotat el mètode `greetingShouldReturnHelloWorldWithoutName` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test.

També que fa servir l'objecte `MockMVC` que heu creat per fer una petició de tipus `GET` a l' URI */hello*; comproveu que el resultat de la petició sigui un 200 (OK), que el *content type* sigui JSON i que tingui al camp *content* la salutació per defecte "*Hello, World!!!*".

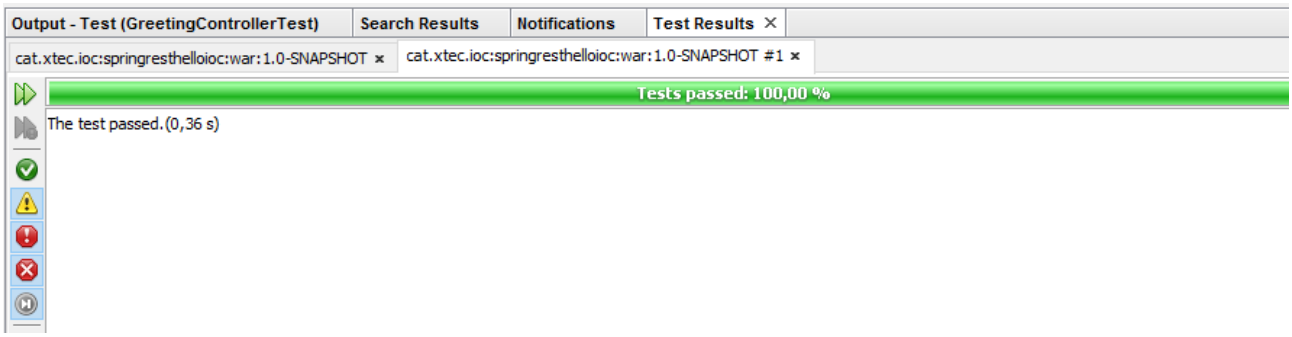
Ja podem executar el test que hem creat fent *Test File* (vegeu la figura 2.4) al menú contextual de la classe de test.

Figura 2.4. Execució del test



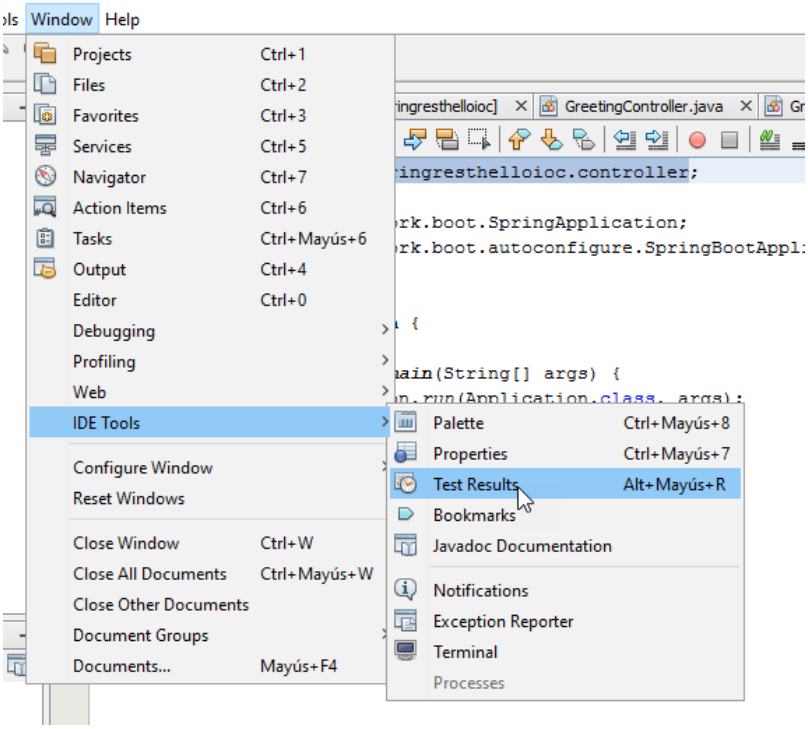
Si tot ha anat bé veureu el resultat a la finestra de *Test* (vegeu la figura 2.5).

Figura 2.5. Resultat de l'execució del test



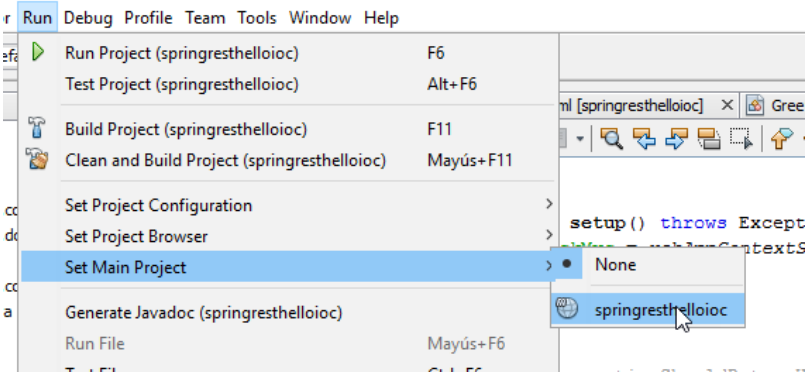
Tot i que el resultat de l'execució dels test es pot veure a la finestra de sortida de NetBeans, és molt més còmode visualitzar-ho a la finestra de resultats de tests que proporciona també NetBeans; per mostrar aquesta finestra aneu al menú *Window / IDE Tools / Test Results* (vegeu la figura 2.6).

Figura 2.6. 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 "Springtestresthelloioc", tal com podeu veure en la figura 2.7.

Figura 2.7. Assignació del projecte com a projecte principal

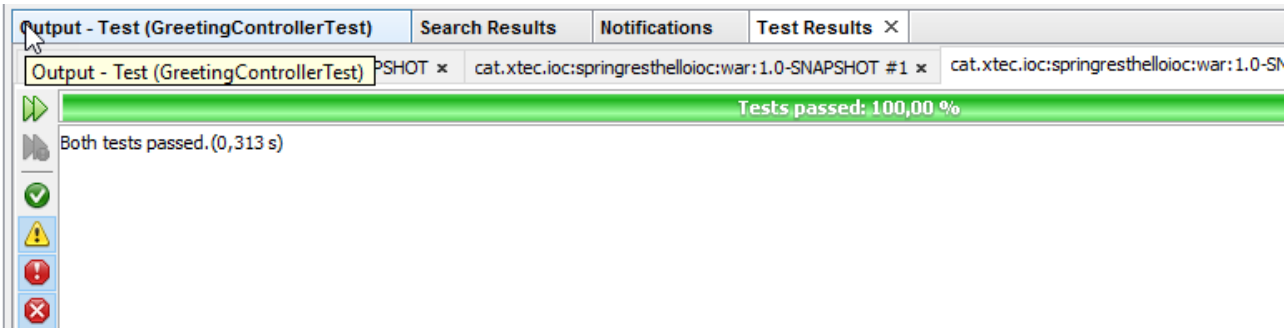


El segon test que farem serà un test que **comprovi que la petició *'/hello?name=User* torna la representació JSON de la salutació "*Hello, User*"** ; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloNameWithName` dins la classe `GreetingControllerTest` amb el següent codi:

```
@Test
public void greetingShouldReturnHelloNameWithName() throws Exception {
    mockMvc.perform(get("/hello?name=User"))
            .andExpect(status().isOk())
            .andExpect(content().contentType(contentType))
            .andExpect(jsonPath("$.content", is("Hello, User")));
}
```

El codi és molt similar al primer test; executeu-lo de la mateixa manera i comproveu que el test s'executa correctament (vegeu la figura 2.8).

FIGURA 2.8. Resultat de l'execució dels dos tests



Aquest dos exemples us donen la base per tal que pugueu fer tots els tests unitaris que se us acudeixin per als serveis web RESTful que desenvolupeu amb Spring i així tenir-los totalment provats.

2.3 El servei web de gestió d'equips de futbol. Operacions CRUD

Veurem els conceptes referents a la creació de serveis web RETSful amb Spring desenvolupant un servei web RESTful que permeti fer operacions sobre equips de futbol i els jugadors que els integren. Farem les operacions típiques CRUD amb els jugadors, una operació que mostra el llistat d'equips, una que mostra els jugadors d'un equip i una altra que mostra les dades d'un equip.

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

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

- llistar tots els equips
- consulta de les dades d'un equip
- llistar tots els jugadors d'un equip
- consulta d'un jugador d'un equip (operació `Read` CRUD)
- creació d'un jugador en un equip (operació `Create` CRUD)
- actualització de les dades d'un jugador d'un equip (operació `Update` CRUD)
- esborrar un jugador d'un equip (operació `Delete` CRUD)

Per fer-ho emprareu una aplicació web ja desenvolupada que segueix una arquitectura per capes i hi afegireu una capa de serveis on creareu i publicareu el servei web de gestió d'equips catàleg com a servei web RESTful. La representació dels recursos que fareu servir en tot l'exemple serà JSON.

2.3.1 Creació i configuració inicial del projecte

L'aplicació de la qual partireu s'anomena "Springrestteamsioc" i servirà per veure com podeu 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 a l'aplicació.

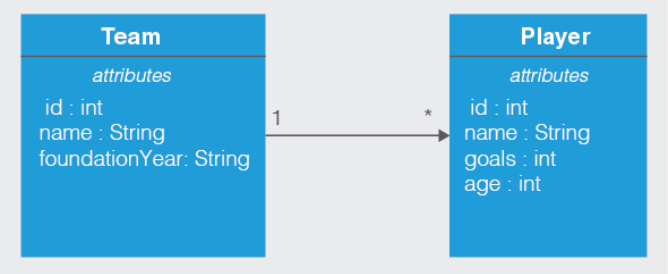
El projecte consta de quatre capes:

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

Com que l'aplicació d'exemple no ha de proporcionar interfície gràfica, fareu servir la capa de presentació com la capa on publicareu els serveis RESTful; seria igualment vàlid fer-ho a una capa de serveis independent.

El model de domini ja el teniu implementat i és molt senzill: hi ha una entitat `Team` que representa un equip i una entitat `Player` que representa els jugadors d'un equip. Entre `Team` i `Player` hi ha una associació *1:N* per indicar que un jugador pertany a un equip i que un equip està format per *N* jugadors (vegeu la figura 2.9).

FIGURA 2.9. Entitats Team i Player



La representació en JSON d'un equip és la següent:

```
{
  "id":1,
  "name":"F.C. Barcelona",
  "foundationYear":"1899"
}
```

I la representació JSON d'un jugador és la següent:

```
{
  "id":1,
  "name":"Lionel Messi",
```

Repositoris '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 fer algunes tasques de configuració que no són pròpies de l'objectiu principal. Per aquest motiu fareu servir repositoris *in memory* tant per als equips com per als jugadors.

```
  "goals":472,
  "age":29,
  "teamId":1
}
```

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 els equips:

- llistar tots els equips
- consulta de les dades d'un equip
- llistar tots els jugadors d'un equip
- consulta d'un jugador d'un equip
- creació d'un jugador en un equip
- actualització de les dades d'un jugador d'un equip
- esborrar un jugador d'un equip

La capa de persistència també la teniu implementada i conté dos objectes repositori que permeten mapar les dades de les fonts de dades amb els objectes del domini. En el nostre cas, per simplicitat, farem servir repositoris *in memory* que tindran una llista precarregada amb els equips i els jugadors de cada equip.

2.3.2 Creació i prova del servei web RESTful

Descarregueu el codi del projecte "Springrestteamsioc" de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

Un cop fet, creareu una classe que exposarà les operacions que voleu realitzar sobre els equips; anomenareu la classe `TeamsController`, la creareu al paquet `cat.xtec.ioc.controller` i l'anotareu amb `@RestController`:

```
package cat.xtec.ioc.controller;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class TeamsController {
}
```

Tot i que podeu descarregar-vos el projecte en l'estat final d'aquest apartat en 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.

Simplement heu anotat la classe amb `@RestController` per marcar la classe com un controlador `REST` on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista. Aquesta és la diferència més gran entre un controlador MVC típic i un controlador per a serveis web RESTful: el cos de la resposta HTTP es crea transformant un objecte del domini a JSON i no es torna en format HTML.

Un cop creada la classe cal que hi afegiu una referència al repositori dels equips fent que Spring el carregui amb el seu mòdul d'injecció de dependències:

```
@Autowired
private TeamRepository teamRepository;

public TeamsController() {
}

public TeamsController(TeamRepository teamRepository) {
    this.teamRepository = teamRepository;
}
```

Quan NetBeans us demani quins imports voleu afegir especifiqueu, a la major part de casos, els del paquet `org.springframework.web.bind.annotation`.

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

```
@RequestMapping(method =  RequestMethod.GET, value = "/teams")
public @ResponseBody List<Team> getAll() {
    return this.teamRepository.getAll();
}
```

Hem anotat el mètode amb l'anotació `@RequestMapping(method = RequestMethod.GET, value = ''/teams'')` per indicar que aquest mètode respondrà a peticions HTTP de tipus `GET` a l'URL */teams*.

Hem anotat el mètode amb `@ResponseBody` per indicar a Spring que torni el resultat en un format que sigui entenedor per al client a partir de les capçaleres HTTP que envia amb la petició. Per defecte, es torna en format JSON.

Tornem la llista d'equips com una llista d'objectes Java de tipus `Team` i Spring s'encarregarà de transformar-los a format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams; si tot ha anat bé, veureu la representació JSON dels equips al navegador:

```
[
{"id":1,"name":"F.C.Barcelona","foundationYear":"1899"},
{"id":2,"name":"Real Madrid","foundationYear":"1902"}
]
```

Passem a crear ara el servei de consulta de les dades d'un equip; per fer-ho, creeu un mètode anomenat `getById` amb el següent codi:

```
@RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
 public @ResponseBody Team getById(@PathVariable int id) {
    return this.teamRepository.get(id);
}
```

El codi és molt similar a la consulta d'equips, però com a particularitat cal comentar que ara tornem un objecte Java de tipus `Team` que Spring convertirà a JSON abans de tornar-lo al client i que hem anotat el mètode amb l'anotació `@RequestMapping(value = ''/teams/{id}'', method = RequestMethod.GET)` per indicar que la informació de l'equip la tornarem quan ens arribin peticions `GET` a l'URI */teams/{id}*; és a dir, el que posem darrere de */teams* serà l'identificador de l'equip que volem consultar.

El paràmetre que rep el mètode s'ha anotat amb l'anotació `@PathVariable int id`. Aquesta és la forma que proporciona Spring per extreure els paràmetres d'una petició. En aquest cas estem extraient un paràmetre del *path* i el passem com a *int* al mètode `getById`.

Spring MVC proporciona un ampli conjunt d'anotacions per fer aquesta tasca fàcil, entre elles `@PathParam`, `@RequestParam`, `@CookieValue`, `@RequestHeader`, etc.

Per exemple, podríem utilitzar `@RequestParam("id")` per extreure un paràmetre que vingués en una petició d'aquesta forma: localhost:8080/springrestteamsioc/teams?id=2.

Per provar si funciona desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams/1; si tot ha anat bé, veureu la representació JSON de l'equip consultat al navegador:

```
[{
  "id":1,
  "name":"F.C. Barcelona",
  "foundationYear":"1899"
}]
```

Ja heu creat les dues operacions que volíeu per als equips, ara començareu amb les operacions que voleu oferir per als jugadors d'un equip. Aquestes operacions les podríeu posar el mateix controlador on heu posat les operacions per als equips, però per fer un codi més net i que respecti al màxim el principi de disseny SRP (de l'anglès *Single Responsability Principle*) ho fareu en un controlador diferent.

Creeu una classe que exposarà les operacions que voleu realitzar sobre els jugadors dels equips; anomenareu la classe `PlayerController`, la creareu al paquet `cat.xtec.ioc.controller` i l'anotareu amb `@RestController`:

```
package cat.xtec.ioc.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@RequestMapping("/teams/{teamId}/players")
public class PlayerController {
}
```

Simplement heu anotat la classe amb `@RestController` per marcar la classe com un controlador `REST` on cada un dels seus mètodes tornarà un objecte del domini enlloc d'una vista.

També heu anotat la classe amb `@RequestMapping("/teams/{teamId}/players")` per indicar que tots els mètodes del controlador s'han d'accedir amb aquest format. Amb això, tots els mètodes del controlador podran extreure del *path* l'identificador de l'equip i no us caldrà repetir-ho a cada un d'ells.

Un cop creada la classe, cal que hi afegiu una referència al repositori dels jugadors dels equips fent que Spring el carregui amb el seu mòdul d'injecció de dependències:

```
@Autowired
private PlayerRepository playerRepository;

public PlayerController() {
}

public PlayerController(PlayerRepository playerRepository) {
    this.playerRepository = playerRepository;
}
```

Creeu primer la funcionalitat que permeti la consulta de tots els jugadors d'un equip. Per fer-ho, creeu un mètode anomenat `getTeamPlayers` amb el següent codi:

```
@RequestMapping(method = RequestMethod.GET)
public @ResponseBody List<Player> getTeamPlayers(@PathVariable int teamId) {
    return this.playerRepository.getByTeam(teamId);
}
```

Heu anotat el mètode amb l'anotació `@RequestMapping(method = RequestMethod.GET)` per indicar que aquest mètode respondrà a peticions HTTP de tipus `GET` i ho farà a l'URI per defecte que heu configurat al controlador (*/teams/{teamId}/players*).

Heu anotat el mètode amb `@ResponseBody` per indicar a Spring que torni el resultat en un format que sigui comprensible per al client a partir de les capçaleres HTTP que el client envia amb la petició. Per defecte, es torna en format JSON.

Torneu la llista d'equips com una llista d'objectes Java de tipus `Player` i Spring s'encarregarà de transformar-los a format JSON.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams/1/players; si tot ha anat bé, veureu la representació JSON dels jugadors del FC Barcelona al navegador:

```
[
{"id":1,"name":"LionelMessi","goals":472,"age":29,"teamId":1},
{"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1}}
]
```

Passareu a crear ara el servei de consulta de les dades d'un jugador d'un equip (l'operació `Read` de CRUD); per fer-ho, creeu un mètode anomenat `getById` amb el següent codi:

```
@RequestMapping(value = "{playerId}", method = RequestMethod.GET)
public @ResponseBody Player getById(@PathVariable int teamId, @PathVariable int
     playerId) {
    return this.playerRepository.get(teamId, playerId);
}
```

El codi és molt similar a la consulta dels jugadors d'un equip; com a particularitat, cal comentar que ara tornem un objecte Java de tipus `Player` que Spring convertirà a JSON abans de tornar-lo al client i que hem anotat el mètode amb l'anotació `@RequestMapping(value = ''{playerId}'', method`

= RequestMethod.GET) per indicar que la informació del jugador la tornarem quan ens arribin peticions GET a l'URI */teams/{teamId}/players/{playerId}*; és a dir, el que posem darrere de */teams* serà l'identificador de l'equip al qual pertany el jugador, i el que posem darrere de *players* serà l'identificador del jugador que volem consultar.

Els paràmetres que rep el mètode s'han anotat amb les anotacions `@PathVariable int teamId` i `@PathVariable int playerId`, respectivament. Aquesta és la forma que proporciona Spring per extreure els paràmetres d'una petició. En aquest cas, estem extraient dos paràmetres del *path* i els passem com a *int* al mètode `getById`.

Per provar si funciona, desplegueu el projecte fent *Run* a NetBeans i accediu amb un navegador a l'URL localhost:8080/springrestteamsioc/teams/1/players/1; si tot ha anat bé, veureu la representació JSON del jugador "Lionel Messi" del "FC Barcelona" al navegador:

```
[{
  {"id":1,
   "name":"Lionel Messi",
   "goals":472,
   "age":29,
   "teamId":1}
 }
]
```

Un cop fetes les dues operacions de consulta sobre els jugadors d'un equip passareu ara a crear el servei que permeti donar d'alta un jugador d'un equip (l'operació `Create` de CRUD); per fer-ho, creeu un mètode anomenat `create` amb el següent codi:

```
@RequestMapping(method = RequestMethod.POST)
public @ResponseBody ResponseEntity<Player> create(@PathVariable int teamId,
    @RequestBody Player player) {
    player.setTeamId(teamId);
    this.playerRepository.add(player);
    return new ResponseEntity<>(player, HttpStatus.CREATED);
}
```

Heu anotat el mètode amb l'anotació `@RequestMapping(method = RequestMethod.POST)` per indicar que aquest mètode respondrà a peticions HTTP de tipus `POST` i ho farà a l'URI per defecte que heu configurat al controlador (*/teams/{teamId}/players*).

La resposta que tornem és de tipus `ResponseEntity<Player>` per tal que hi puguem afegir la representació del jugador creat i el codi HTTP (en aquest cas un codi 201, `CREATED`, per indicar que el recurs s'ha creat satisfactòriament).

Fixeu-vos que tan sols amb aquesta informació Spring és capaç, quan rep una petició `POST` a l'URI */teams/{teamId}/players* amb una representació JSON d'un jugador, de crear un objecte de tipus `Player` i passar-lo al mètode `create`.

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 documentar i provar API RESTful teniu moltes alternatives, entre les quals cal destacar Swagger swagger.io i Postman www.getpostman.com, que és una extensió per al navegador Google Chrome.

per fer peticions POST 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.

Afegiu el paràmetre `-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 equips feu un *command prompt*:

```
curl localhost:8080/springrestteamsioc/teams
```

Per provar la funcionalitat que permet afegir un jugador al FC Barcelona desplegueu el projecte fent *Run* a NetBeans i feu una petició POST a l'URL localhost:8080/springrestteamsioc /teams/1/players especificant la informació del jugador amb JSON; això ho podeu fer amb cURL i la següent comanda:

```
curl –H "Content–Type: application/json" –X POST –d "{\"id\":\"3\",\"name\":\"
    Neymar da Silva Santos\",\"age\":\"24\",\"goals\":\"100\"}" http://
    localhost:8080/springrestteamsioc/teams/1/players
```

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

Executeu la comanda anterior i després consulteu els jugadors del FC Barcelona amb:

```
curl localhost:8080/springrestteamsioc/teams/1/players
```

Si tot ha anat bé, veureu que Neymar s'ha afegit com a jugador del FC Barcelona:

```
[
 {"id":1,"name":"Lionel Messi","goals":472,"age":29,"teamId":1},
 {"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1},
 {"id":3,"name":"Neymar da Silva Santos","goals":100,"age":24,"teamId":1}
]
```

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

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody ResponseEntity<Player> update(@PathVariable int teamId,
    @RequestBody Player player) {
    player.setTeamId(teamId);
    this.playerRepository.update(player);
    return new ResponseEntity<>(player, HttpStatus.OK);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la creació d'un jugador, 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 jugador desplegueu el projecte fent *Run* a NetBeans i feu una petició PUT a l'URL lo-

calhost:8080/springrestteamsioc /teams/1/players especificant la nova informació del jugador; això ho podeu fer amb cURL i la següent comanda:

```
curl –H "Content–Type: application/json" –X PUT –d "{\"id\":\"3\",\"name\":\"
    Neymar da Silva Santos\",\"age\":\"24\",\"goals\":\"120\"}" http://
    localhost:8080/springrestteamsioc/teams/1/players
```

Pèrdua de dades

Us pot passar que en afegir el mètode `update` per modificar la informació d'un jugador es torni a desplegar l'aplicació, i, com que feu servir un repositori *in memory*, es perdin les dades dels jugadors que heu donat d'alta. Si us trobeu en aquest cas simplement cal que torneu a fer l'alta del jugador *Neymar* i després en feu la modificació.

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

```
curl http://localhost:8080/springrestteamsioc/teams/1/players/3
```

Si tot ha anat bé, veureu que el nombre de gols que ha marcat Neymar ha passat de 100 a 120:

```
[
 {"id":3,"name":"Neymar da Silva Santos","goals":120,"age":24,"teamId":1}
]
```

Creem ara el servei que permeti esborrar un jugador d'un equip (l'operació `Delete` de CRUD); per fer-ho, creeu un mètode anomenat `delete` amb el següent codi:

```
@RequestMapping(value = "{playerId}", method = RequestMethod.DELETE)
public @ResponseBody ResponseEntity<Player> delete(@PathVariable int teamId,
    @PathVariable int playerId) {
    this.playerRepository.delete(teamId, playerId);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Les consideracions en aquest cas són les mateixes que hem vist per al cas de la consulta d'un jugador d'un equip, i l'únic que canvia és que fareu servir el verb `DELETE` en lloc del verb `GET`.

Per provar la funcionalitat que permet esborrar un jugador d'un equip desplegueu el projecte fent *Run* a NetBeans i feu una petició `DELETE` a l'URL localhost:8080/springrestteamsioc/{teamId}/players/{playerId} especificant l'equip al qual pertany el jugador i l'identificador del jugador que volem esborrar; això ho podeu fer amb cURL i la següent comanda (per exemple, per esborrar el jugador Neymar del FC Barcelona):

```
curl –X DELETE localhost:8080/springrestteamsioc/teams/1/players/3
```

Executeu la comanda anterior i després consulteu els jugadors del FC Barcelona amb:

```
curl localhost:8080/springrestteamsioc/teams/1/players/
```

Si tot ha anat bé, veureu que Neymar ja no és jugador del FC Barcelona:

```
[
 {"id":1,"name":"Lionel Messi","goals":472,"age":29,"teamId":1},
 {"id":2,"name":"Luis Suarez","goals":100,"age":29,"teamId":1}
]
```

I amb això ja heu codificat i provat el servei web RESTful que us permet treballar amb els equips i els seus jugadors. El codi final de la classe `TeamController` és el següent:

```
package cat.xtec.ioc.controller;

import cat.xtec.ioc.domain.Team;
import cat.xtec.ioc.domain.repository.TeamRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TeamsController {

    @Autowired
    private TeamRepository teamRepository;

    public TeamsController() {
    }

    public TeamsController(TeamRepository teamRepository) {
        this.teamRepository = teamRepository;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/teams")
    public @ResponseBody
    List<Team> getAll() {
        return this.teamRepository.getAll();
    }

    @RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Team getById(@PathVariable int id) {
        return this.teamRepository.get(id);
    }
}
```

I el de la classe `PlayerController` és el següent:

```
package cat.xtec.ioc.controller;

import cat.xtec.ioc.domain.Player;
import cat.xtec.ioc.domain.repository.PlayerRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/teams/{teamId}/players")
public class PlayerController {

    @Autowired
    private PlayerRepository playerRepository;

    public PlayerController() {
    }
```

```
25
26      public PlayerController(PlayerRepository playerRepository) {
27          this.playerRepository = playerRepository;
28      }
29
30      @RequestMapping(method = RequestMethod.GET)
31      public @ResponseBody
32      List<Player> getTeamPlayers(@PathVariable int teamId) {
33          return this.playerRepository.getByTeam(teamId);
34      }
35
36      @RequestMapping(value = "{playerId}", method = RequestMethod.GET)
37      public @ResponseBody
38      Player getById(@PathVariable int teamId, @PathVariable int playerId) {
39          return this.playerRepository.get(teamId, playerId);
40      }
41
42      @RequestMapping(method = RequestMethod.POST)
43      public @ResponseBody
44      ResponseEntity<Player> create(@PathVariable int teamId, @RequestBody Player
            player) {
45          player.setTeamId(teamId);
46          this.playerRepository.add(player);
47          return new ResponseEntity<>(player, HttpStatus.CREATED);
48      }
49
50      @RequestMapping(method = RequestMethod.PUT)
51      public @ResponseBody
52      ResponseEntity<Player> update(@PathVariable int teamId, @RequestBody Player
            player) {
53          player.setTeamId(teamId);
54          this.playerRepository.update(player);
55          return new ResponseEntity<>(player, HttpStatus.OK);
56      }
57
58      @RequestMapping(value = "{playerId}", method = RequestMethod.DELETE)
59      public @ResponseBody
60      ResponseEntity<Player> delete(@PathVariable int teamId, @PathVariable int
            playerId) {
61          this.playerRepository.delete(teamId, playerId);
62          return new ResponseEntity<>(HttpStatus.OK);
63      }
64  }
```

2.4 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament dels serveis web RESTful amb Spring i les heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après:

- Les nocions bàsiques dels serveis web RESTful amb Spring.
- Desenvolupar, desplegar i provar un servei web RESTful senzill amb Spring.
- Fer tests unitaris dels serveis web RESTful amb Spring.
- Desenvolupar, desplegar i provar un servei web RESTful complex amb Spring que inclou operacions CRUD sobre un recurs.

Per aprofundir en aquests conceptes i veure quins són els conceptes que hi ha darrere de HATEOAS (de l'anglès *Hypermedia as the Engine of Application State*) us recomanem la realització de l' activitat associada a aquest apartat.

3. Serveis web RESTful amb Spring. Consumint serveis web

Explicarem, mitjançant exemples, com podem consumir serveis web RESTful remots amb diferents tipus de clients.

> 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`) 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 emprar 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 ho podem fer amb l'API de baix nivell `java.net.HttpURLConnection` o alguna llibreria propietària que us fes la vida una mica més fàcil. Una opció és utilitzar la classe `RestTemplate`, que proporciona Spring, o bé l'API client de JAX-RS; les dues opcions permeten fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

> Spring proporciona la classe `RestTemplate` per **simplificar la comunicació** HTTP entre el client i el servidor. `RestTemplate` permet fer tota mena de peticions HTTP als serveis web RESTful remots de forma fàcil.

`RestTemplate` es pot utilitzar per fer tests d'integració per provar els serveis RESTful, **estiguin o no** desenvolupats amb Spring.

AJAX és un conjunt de tècniques de desenvolupament d'aplicacions web que permeten crear aplicacions web client asíncrones que cada vegada s'utilitzen més per accedir a serveis web RESTful, tant des d'aplicacions web com des d'aplicacions mòbils.

3.1 Un client Java per al servei web RESTful "'Hello, World!!!'"

Veurem com consumir serveis web RESTful des d'una aplicació Java *stand-alone* utilitzant la classe `RestTemplate` que proporciona Spring per fer-ho.

3.1.1 Creació i configuració inicial del projecte

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

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

```
1  package cat.xtec.ioc.springresthelloioc.controller;
2
3  import cat.xtec.ioc.springresthelloioc.domain.Greeting;
4  import java.util.concurrent.atomic.AtomicLong;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RequestMethod;
7  import org.springframework.web.bind.annotation.RequestParam;
8  import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 public class GreetingController {
12
13     private static final String template = "Hello, %s";
14     private final AtomicLong counter = new AtomicLong();
15
16     @RequestMapping(method =  RequestMethod.GET, value = "/hello")
17     public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!
           ") String name) {
18         return new Greeting(counter.incrementAndGet(),
19                             String.format(template, name));
20     }
21 }
```

Aquest és un servei web RESTful desenvolupat amb Spring que respon a peticions `GET` a l'URI */hello* amb "*Hello, World!!!*" si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

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.

El servei web torna la salutació en format JSON i respon a les peticions `GET` a */hello*:

```
1 {"id":1,"content":" Hello, World!!!"}
```

I a les peticions `GET` a */hello?name=User* amb:

```
1 {"id":1,"content":"Hello, User!"}
```

Un cop importat el projecte cal que modifiqueu el pom.xml per afegir les dependències cap a Jackson per tal que Spring pugui fer la transformació de dades de JSON a objecte Java. Per fer-ho, afegiu les següents línies al fitxer pom.xml:

```
1 <dependency>
2     <groupId>com.fasterxml.jackson.core</groupId>
3     <artifactId>jackson-databind</artifactId>
```

```
    <version>2.5.3</version>
</dependency>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i prement l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el client Spring.

3.1.2 Creació del client Java 'stand-alone'

El primer que necessitem és una classe Java que representarà el model del domini a la part client i que Spring construirà a partir del missatge JSON retornat pel servei web.

Fixeu-vos que en el nostre cas es podria haver utilitzat per fer això la classe `Greeting` que teniu al paquet `cat.xtec.ioc.springresthelloioc.domain`, i ens estalviaríem de crear-ne una de nova. En crearem una, ja que l'habitual és que el client i el servei web no estiguin al mateix projecte; nosaltres els hem posat al mateix projecte per simplicitat.

Creeu, doncs, la classe Java que modelarà la resposta del servei web al paquet `cat.xtec.ioc.springresthelloioc.client` i l'anomeneu `GreetingClient`:

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

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class GreetingClient {

    private long id;
    private String content;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public String toString() {
        return "Greeting{" +
                "id='" + id + '\'' +
                ", content=" + content +
                '}';
    }
}
```

Es tracta d'una classe Java normal amb una propietat per a cada un dels camps que conté la salutació que torna el servei web. L'única particularitat és l'anotació `@JsonIgnoreProperties(ignoreUnknown = true)` per indicar que en la transformació de JSON a objecte Java ignori les propietats que no tinguin un mapatge definit.

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

```
1  package cat.xtec.ioc.springresthelloioc.client;
2
3  import org.springframework.web.client.RestTemplate;
4
5  public class HelloWorldClient {
6
7      public static void main(String[] args) {
8          RestTemplate restTemplate = new RestTemplate();
9          GreetingClient greeting = restTemplate.getForObject("http://localhost
               :8080/springresthellojavaclientioc/hello", GreetingClient.class);
10         System.out.println(greeting.toString());
11         greeting = restTemplate.getForObject("http://localhost:8080/
               springresthellojavaclientioc/hello?name=User", GreetingClient.
               class);
12         System.out.println(greeting.toString());
13     }
14 }
```

Com veieu, el client és molt simple: creem un objecte de tipus `RestTemplate` i cridem el mètode `getForObject` amb l'URL del servei web i la classe en la qual ha de convertir el missatge JSON rebut.

Aquesta és només una de les maneres d'utilitzar la classe `RestTemplate`; n'hi ha moltes més que us permetran fer crides amb tot el conjunt de verbs HTTP, passar paràmetres JSON, recuperar tota la informació de la resposta rebuda, etc. Per veure totes les opcions que us permet la classe `RestTemplate` podeu consultar la seva API en la següent adreça: bit.ly/2nywYZy.

Ara ja només 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/springresthellojavaclientioc/hello amb un navegador. El servei web us ha de tornar la salutació "*Hello, World!!!*" en format JSON.

Si tot és correcte ja podem executar el client; per fer-ho, poseu-vos damunt de la classe `HelloWorldClient` i feu *Run File* a NetBeans i veureu la salutació "*Hello, World!!!*" i "*Hello, User*" a la consola de sortida:

```
Greeting{id='1', content=Hello, World!!!}
Greeting{id='2', content=Hello, User}
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

3.2 Tests d'integració per al servei web RESTful '"Hello, World!!!"'

Vegem com fer tests d'integració d'un servei web RESTful amb JUnit i la classe `RestTemplate`.

El primer que farem serà desplegar a Glassfish el servei web REST que volem provar i després crearem el conjunt de tests d'integració que faran peticions HTTP al servei web.

> Els **tests d'integració** difereixen dels tests unitaris, ja 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

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

Descarregueu el codi del projecte "Springtestintresthelloioc" de l'enllaç que trobareu als annexos de la unitat i importeu-lo a NetBeans.

```
package cat.xtec.ioc.springresthelloioc.controller;

import cat.xtec.ioc.springresthelloioc.domain.Greeting;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping(method =  RequestMethod.GET, value = "/hello")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!
        ") String name) {
        return new Greeting(counter.incrementAndGet(),
                            String.format(template, name));
    }
}
```

Aquest és un servei web RESTful desenvolupat amb Spring que respon a peticions GET a l'URI */hello* amb "*Hello, World!!!*" si no li passem cap paràmetre, i si li

passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

El servei web torna la salutació en format JSON i respon a les peticions `GET` a */hello* amb:

```
{"id":1,"content":" Hello, World!!!"}
```

I a les peticions `GET` a */hello?name=User* amb:

```
{"id":1,"content":"Hello, User"}
```

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.

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 els tests.

JUnit

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 una 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.

Un cop importat el projecte cal que modifiqueu el pom.xml per afegir les dependències cap a Jackson per tal que Spring pugui fer la transformació de dades de JSON a objecte Java i a JUnit 4. Per fer-ho, afegiu les següent línies al fitxer pom.xml:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.3</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
    <type>jar</type>
</dependency>
```

Si recarregueu el pom.xml fent clic amb el botó dret damunt el nom del projecte i premeu l'opció *Reload POM* del menú contextual ja tindreu el projecte configurat i llest per començar a crear el client amb Spring.

Primer desplegueu el servei web com a part de l'aplicació Java EE que el conté fent *Clean and Build* i després *Run* a NetBeans, i comproveu que s'ha desplegat correctament accedint a l'URL localhost:8080/springtestintresthelloioc/hello amb un navegador. El servei web us ha de tornar la salutació "*Hello, World!!!*" en format JSON:

```
{"id":1,"content":" Hello, World!!!"}
```

Noteu que l'identificador retornat anirà canviant en funció del número de petició.

3.2.2 Creació i execució dels tests d'integració

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

El primer que necessitem és una classe Java que representarà el model del domini i que Spring construirà a partir del missatge JSON retornat pel servei web.

Fixeu-vos que per fer això es podria haver utilitzat la classe `Greeting` que teniu al paquet `cat.xtec.ioc.springresthelloioc.domain` i ens estalviaríem de crear-ne una de nova (tan sols li hauríeu d'haver afegit un constructor sense paràmetres).

Creeu un nou paquet dins de *Test Packages* anomenat, per exemple, `cat.xtec.ioc.springresthelloioc.controller`, i creeu-hi una classe Java anomenada `GreetingTest`:

```
package cat.xtec.ioc.springresthelloioc.controller;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class GreetingTest {

    private long id;
    private String content;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public String toString() {
        return "Greeting{" +
                "id='" + id + '\'' +
                ", content=" + content +
                '}';
    }
}
```

Es tracta d'una classe Java normal amb una propietat per a cada un dels camps que conté la salutació que torna el servei web. L'única particularitat és l'anotació `@JsonIgnoreProperties(ignoreUnknown = true)` per indicar que en la transformació de JSON a objecte Java ignori les propietats que no tinguin un mapatge definit.

Creeu ara al paquet `cat.xtec.ioc.springresthelloioc.controller` una classe Java anomenada `GreetingControllerTest`:

```
package cat.xtec.ioc.springresthelloioc.controller;

import javax.ws.rs.core.MediaType;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
```

```
9
10
11 public class GreetingControllerTest {
12     private static final String uri = "http://localhost:8080/
           springtestintresthelloioc/hello";
13
14
15 }
```

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 *Arrange-Act-Assert*), amb el qual els tests sempre tindran una fase de **preparació** (*Arrange*), una d'**execució** (*Act*) i una de **verificació** de resultats (*Assert*).

El primer test que farem serà un test que **comprovi que la petició */hello* sense cap paràmetre torna la representació JSON de la salutació "*Hello, World!!!*"** ; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloWorldWithoutName` dins la classe `GreetingControllerTest` amb el següent codi:

```
1  @Test
2  public void greetingShouldReturnHelloWorldWithoutName() {
3    // Arrange
4    RestTemplate restTemplate = new RestTemplate();
5
6    // Act
7    GreetingTest greeting = restTemplate.getForObject(uri, GreetingTest.class);
8
9    // Assert
10   assertEquals("Hello, World!!!", greeting.getContent());
11 }
```

Hem anotat el mètode `greetingShouldReturnHelloWorldWithoutName` amb l'anotació `@Test` per indicar que es tracta d'un mètode de test; en la fase de **preparació** simplement creem l'objecte `RestTemplate` per enviar les peticions HTTP al servei web RESTful, després **executem** la petició cridant el mètode `getForObject` i, finalment, **verifiquem** que la salutació retornada coincideix amb la que esperem.

Ja podem executar el test que hem creat fent *Test File* al menú contextual de la classe de *Test*, i si tot ha anat bé veureu el resultat de l'execució correcta dels tests a la finestra de *Test*.

Per mostrar alguna altra capacitat de la classe `RestTemplate` farem un altre test que comprovarà que el codi HTTP de retorn és un 200 (OK) i que el *content type* sigui JSON.

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

```
1  @Test
2  public void greetingShouldReturnOKAndJSON () {
3    // Arrange
4    RestTemplate restTemplate = new RestTemplate();
5
6    // Act
7    ResponseEntity<GreetingTest> response = restTemplate.getForEntity(uri,
         GreetingTest.class);
8
9    // Assert
10   assertEquals(HttpStatus.OK, response.getStatusCode());
11   assertEquals(MediaType.APPLICATION_JSON, response.getHeaders().getContentType
         ().getType() + "/" +              response.getHeaders().getContentType
         ().getSubtype());
12 }
```

Executeu-lo de la mateixa manera i comproveu que el test s'executa correctament.

El darrer test que farem serà un test que **comprovi que la petició *`/hello?name=User`* torna la representació JSON de la salutació "*Hello, User*"**; per fer-ho, creeu un mètode anomenat `greetingShouldReturnHelloNameWithName` dins la classe `GreetingControllerTest` amb el següent codi:

```
@Test
public void greetingShouldReturnHelloNameWithName() {
  // Arrange
  RestTemplate restTemplate = new RestTemplate();

  // Act
  GreetingTest greeting = restTemplate.getForObject(uri + "?name=User",
      GreetingTest.class);

  // Assert
  assertEquals("Hello, User", greeting.getContent());
}
```

El codi és molt similar al primer test; executeu-lo de la mateixa manera i comproveu que el test s'executa correctament.

Aquests tres exemples us donen la base per tal que pugueu fer tots els tests d'integració que se us acudeixin i així tenir els serveis web RESTful totalment provats!

> L'ús de `RestTemplate` és l'opció que proporciona Spring; els mateixos tests d'integració els podríeu haver escrit amb qualsevol altre bastiment que permeti fer peticions HTTP. Una opció perfectament vàlida hauria estat utilitzar **JAX-RS**, l'API client per consumir serveis web RESTful que proporciona Java EE.

Noteu que, a part dels tests d'integració, Spring també us permet fer tests unitaris per provar els serveis web RESTful.

3.3 Un client JavaScript per al servei web RESTful "'Hello, World!!!'"

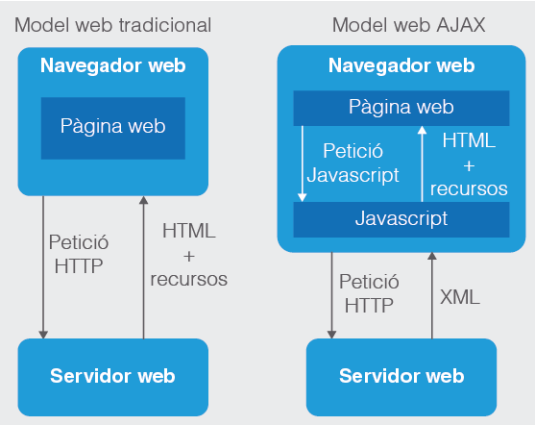
Veurem com consumir serveis web RESTful des d'una aplicació web amb peticions AJAX de JavaScript. Utilitzarem Angular JS com a bastiment JavaScript per fer les peticions AJAX.

> **AJAX** (de l'anglès *Asynchronous JavaScript And XML*) és una tècnica de desenvolupament que permet fer les pàgines web d'una aplicació web interactives amb JavaScript.

AJAX és un conjunt de tècniques (vegeu la figura 3.1) de desenvolupament d'aplicacions web que permeten crear aplicacions web client asíncrones. Amb

AJAX, la part client de les aplicacions web pot enviar i recuperar informació del servidor de forma asíncrona (en *background*) sense interferir en com es mostra i com es comporta la pàgina. Aquesta forma de desacoblar la capa d'intercanvi de dades de la capa de presentació permet a les pàgines web, i, per extensió, a les aplicacions web, intercanviar contingut dinàmicament amb el servidor sense haver de recarregar tota la pàgina. Tot i que inicialment estava centrat en XML, cada vegada més el format de les dades intercanviades tendeix més a ser JSON, ja que aquest és un format nadiu per a les dades en JavaScript.

Figura 3.1. AJAX



Objectiu de l'exemple

L'objectiu d'aquest exemple no és aprofundir en el coneixement d'Angular JS, sinó que només pretén mostrar com es pot fer servir AJAX com a client per consumir serveis web RESTful. Si voleu aprofundir en el coneixement d'Angular JS ho podeu fer a l'URL angularjs.org.

Les aplicacions que fan servir AJAX segueixen, en molta mesura, els principis de disseny inherents a l'arquitectura d'aplicacions REST. Podem veure cada una de les peticions AJAX com una petició a un servei REST que tornarà les dades en un format determinat, moltes vegades JSON, per tal que l'aplicació web les tracti i mostri a l'usuari sense necessitat de recarregar tota la pàgina.

Aquestes característiques fan d'AJAX una de les formes més utilitzades a l'hora de consumir serveis web RESTful des de pàgines o aplicacions web.

Actualment hi ha multitud de bastiments JavaScript que proporcionen, entre moltes altres coses, la possibilitat de fer peticions AJAX. Potser el més popular des dels seus inicis és jQuery.

Angular JS és un altre bastiment JavaScript que, entre moltes altres coses, proporciona capacitats i ajudes per fer peticions AJAX a serveis web RESTful.

A l'exemple veurem un client Angular JS que consumirà amb AJAX el servei web RESTful de salutacions.

3.3.1 Creació i configuració inicial del projecte

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

El projecte té un servei web RESTful configurat a la classe `GreetingController` amb el següent codi:

```
1 package cat.xtec.ioc.springresthelloioc.controller;
2
```

```
import cat.xtec.ioc.springresthelloioc.domain.Greeting;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping(method =  RequestMethod.GET, value = "/hello")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World!!!
        ") String name) {
        return new Greeting(counter.incrementAndGet(),
                            String.format(template, name));
    }
}
```

Aquest servei web és un servei web RESTful desenvolupat amb Spring que respon a peticions `GET` a l'URI */hello* amb "*Hello, World!!!*" si no li passem cap paràmetre, i si li passem un paràmetre anomenat `name` torna una salutació personalitzada canviant la paraula *World* pel nom especificat a `name`.

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.

El servei web torna la salutació en format JSON i respon a les peticions `GET` a */hello* amb:

```
{"id":1,"content":" Hello, World!!!"}
```

I a les peticions `GET` a */hello?name=User* amb:

```
{"id":1,"content":"Hello, User!"}
```

3.3.2 Creació del client JavaScrip amb Angular JS

El primer que necessitem és crear un mòdul controlador Angular JS que consumeixi el servei web. Per fer-ho, creeu un fitxer anomenat helloangular.js a *Web Pages/static* amb el següent contingut:

```
angular.module('demo', [])
  .controller('Hello', function ($scope, $http) {
      $http.get('http://localhost:8080/springresthelloangularclientioc/hello').
              then(function (response) {
                  $scope.greeting = response.data;
              });
  });
```

Aquest mòdul controlador, anomenat *Hello*, es representa com una funció JavaScript a la qual se li passa com a paràmetres els components `scope` i `http`. La funció fa servir el component `http` per fer una crida al servei RESTful a */hello*.

Si la crida té èxit, s'assignarà el JSON retornat des del servei web la variable `scope.greeting` amb la creació d'un objecte JavaScript que representarà el

model. L'establiment d'aquest objecte al model fa que Angular el pugui utilitzar al DOM de la pàgina que fa la sol·licitud i mostrar el seu contingut a l'usuari.

Un cop tenim el controlador ens cal crear la pàgina HTML que el carregarà al navegador de l'usuari. Per fer-ho, creeu un fitxer anomenat helloangular.html a *Web Pages/static* amb el següent contingut:

```
<!doctype html>
<html ng-app="demo">
    <head>
        <title>Hello AngularJS</title>
        <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/
            css/bootstrap.min.css">
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/
            angular.min.js"></script>
        <script src="helloangular.js"></script>
    </head>

    <body>
        <div ng-controller="Hello">
            <p>The ID is {{greeting.id}}</p>
            <p>The content is {{greeting.content}}</p>
        </div>
    </body>
</html>
```

A la secció *head* de la pàgina hi posem aquestes dues etiquetes:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.
    js"></script>
<script src="helloangular.js"></script>
```

La primera descarrega la llibreria `angular` del CDN (de l'anglès *Content Delivery Network*) per tal que no l'haguem d'incorporar al projecte, i el segon carrega el codi JavaScript del controlador que hem creat (helloangular.js).

Angular JS habilita un conjunt d'etiquetes pròpies que podrem utilitzar a les pàgines HTML. A l'exemple fem servir l'atribut `ng-app` per indicar que la pàgina és una aplicació Angular JS i l'etiqueta `ng-controller` per indicar que el controlador de la pàgina serà el controlador `Hello` definit al fitxer JavaScript helloangular.js.

Finalment, la pàgina accedeix a l'objecte del model `greeting` que ens ha deixat el controlador i mostra l'identificador i la salutació.

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

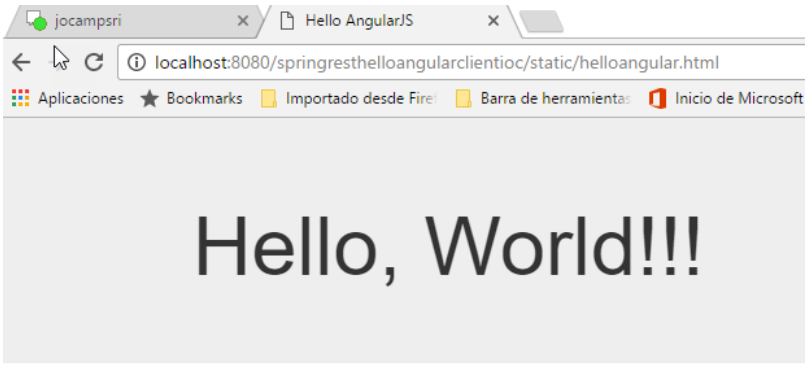
3.3.3 Desplegament del servei web i prova del client Angular

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/springresthelloangularclientioc/hello amb un navegador. El servei web us ha de tornar la salutació "*Hello, World!!!*" en format JSON.

Si tot és correcte ja podem provar el client Angular JS accedint a l'URL localhost:8080/springresthelloangularclientioc/static/helloangular.html amb un navegador, i veureu la salutació "*Hello, World!!!*" tornada pel servei web (vegeu la figura 3.2).

Figura 3.2. Execució del client Angular JS



3.4 Què s'ha après?

En aquest apartat heu vist les bases per al desenvolupament de clients Java i JavaScript que accedeixin a serveis web RESTful, i els heu treballat de forma pràctica mitjançant exemples.

Concretament, heu après a:

- Desenvolupar i provar un client Java *stand-alone* que us permeti consultar un servei web RESTful mitjançant la classe `RestTemplate`.
- Fer tests d'integració amb `RestTemplate` que us permetin provar els serveis web RESTful.
- Fer crides AJAX a serveis web RESTful.

Per veure com habilitar sol·licituds Cross-Origin per un servei web RESTful us recomanem la realització de l'activitat associada a aquest apartat.