Generació automàtica de clients JAX-RS

Estudiant: Enric Ruiz Martínez
Consultor: Oscar Escudero Sánchez

Índex

Capítol 1. Introducció

1.1 Descripció del projecte

El present projecte final de carrera (PFC, en endavant) consisteix en el l'anàlisi, disseny i implementació d'un *framework* que simplifiqui el manteniment de llibreries client per a serveis distribuïts que exposen una interfície *Representational State Transfer* (REST, en endavant) i estiguin implementats en Java seguint l'especificació JAX-RS.

L'especificació JAX-RS afegeix suport al llenguatge de programació Java per crear serveis web d'acord amb una arquitectura REST en base a l'ús d'anotacions per indicar com un cert mètode és exposat a l'API REST. Com veurem més endavant, existeixen múltiples implementacions i algunes difereixen de l'especificació en alguns punts.

La idea darrera del PFC és aprofitar les anotacions JAX-RS que descriuen el servei web per a generar de manera automàtica un client complert i compatible amb el servei descrit per les anotacions. De fet l'especificació JAX-RS ja ofereix un mecanisme per a usar la implementació del servei com a client. La diferencia amb el mètode proposat en aquest PFC és que el codi generat no depèn del codi del servei per a ser executat i que el llenguatge de programació del client no ha de ser necessàriament Java.

El *framework* es recolza en dues entitats fonamentals: el cercador i el generador. El cercador usa els mecanismes de reflexió de Java per a detectar, en el codi del servei, els mètodes que exposen recursos REST i el generador usa la sortida del cercador per a generar un client per al servidor descrit.

El *framework* permet el desenvolupament de cercadors per a les diverses implementacions de JAX-RS i la combinació d'aquests amb generadors de diferents llenguatges de programació.

1.2 Objectius generals i específics

L'objectiu principal del PFC és oferir als desenvolupadors un conjunt d'eines que permetin simplificar el manteniment de clients REST mitjançant l'automatització completa del seu proces de desenvolupament.

Per aquest motiu és necessària la creació d'un *framework* al voltant del qual construir aquestes eines. En concret ha de permetre:

- Personalitzar la cerca de mètodes en funció d'una implementació determinada JAX-RS.
- Crear generadors de clients REST per a diferents llenguatges de programació.
- Coordinar les combinacions de cercadors i generadors amb l'objectiu d'obtindre la llibreria client.

D'altra banda, el conjunt d'eines desenvolupades s'ha d'integrar en el proces de construcció dels serveis web. En aquest PFC ens preocuparem de desenvolupar un *plug-in*

per l'eina Apache MAVEN que, recolzant-se amb el *framework*, permeti:

- Analitzar si el codi del servei és correcte per a ser processat per la lògica del *framework*.
- Generar el codi del client i desar-lo en una localització concreta del sistema de fitxers.

Pel que fa al suport d'implementacions de JAX-RS suportades, aquest PFC pretén oferir, inicialment, suport per a Jersey i Apache Wink. El primer d'ells respecta l'estàndard i el segon afegeix un conjunt d'anotacions de collita pròpia a les que cal donar suport.

Respecte als clients generats, es desenvoluparà únicament el generador de codi Java ja que serà el més senzill de justificar i analitzar en el context d'aquest PFC.

A més durant el desenvolupament del PFC s'implementarà un servei web de proba que ens permetrà il·lustrar l'ús de les dues implementacions de JAX-RS suportades inicialment així com ser la base sobre la qual realitzar els tests del *framework*.

1.3 Pla de treball

La planificació del PFC ve donada pel calendari d'avaluació continuada de l'assignatura. Per a cadascuna de les dates d'entrega es descriu el contingut treballat en cada fase i així com l'entrega a realitzar.

1.3.1 Pla de treball (22 Febrer 2015 – 9 Març 2015)

- Descripció del PFC
- Objectius generals i específics
- Descripció de les diferents entregues

1.3.2 Anàlisi i disseny (10 Març 2015 – 13 Abril 2015)

- Introducció a l'especificació JAX-RS
    - Anàlisi del *framework* Jersey
    - Anàlisi del *framework* Apache Wink
- Anàlisi i disseny del *framework*
- Introducció de l'eina Apache MAVEN
- Anàlisi i disseny del *plug-in* d'Apache Maven per a la generació automàtica de clients REST

1.3.3 Implementació (14 Abril 2015 – 1 Juny 2015)

- Definició i implementació del servei per probes
- Implementació del *framework*

- Implementació de l'adaptador per Jersey
- Implementació de l'adaptador per Apache Wink
- Implementació del generador Java
- Implementació del *plug-in* d'Apache MAVEN
- Implementació de tests per verificar el correcte funcionament de *framework*

1.3.4 Memòria i presentació (2 Juny 2015 – 15 Juny 2015)

- Realització de la memòria del projecte
- Realització de la presentació del projecte

Capítol 2. Introducció a JAX-RS

JAX-RS és una API del llenguatge de programació Java que afegeix suport per a la creació de serveis web d'acord amb el patró arquitectònic REST. En aquest capítol es descriuran, a grans trets, l'especificació de l'API i algunes de les implementacions disponibles.

2.1 Especificació

JAX-RS ofereix un conjunt d'anotacions Java que permet descriure la signatura d'un servei REST mitjançant l'anotació de classes i mètodes que seran exposats com a recursos web.

Per aquest motiu l'especificació usa HTTP com a protocol de xarxa i ofereix un mapatge clar i concís entre les URIs, verbs HTTP i els corresponents mètodes i classes que exposen el servei web.

A més, ofereix la possibilitat d'emprar diferents tipus de continguts, com per exemple XML o JSON, d'una manera estàndard.

2.1.1 Anotacions principals

A continuació es fa un recull de les anotacions més representatives i didàctiques de totes les disponibles a l'especificació.

De les anotacions que poden ser aplicades tan a classes com a mètodes, destaquen:

- @Path: Especifica la ruta relativa (URI) a un recurs. Aquesta anotació identifica un recurs arrel si es troba en una classe i identifica un sub-recurs si es troba en un mètode. La ruta que identifica a un sub-recurs és el resultat de concatenar la ruta que identifica el recurs amb la que identifica el sub-recurs.
- @Consumes: Especifica els *media types* que accepta el recurs anotat. L'anotació en el sub-recurs reemplaça la possible anotació del recurs.
- @Produces: Especifica els *media types* de resposta del recurs anotat. L'anotació en el sub-recurs reemplaça la possible anotació del recurs.

De les anotacions exclusives pels mètodes cal destacar:

- @GET: Especifica que el mètode anotat es accessible mitjançant una petició GET.
- @POST: Especifica que el mètode anotat es accessible mitjançant una petició POST.
- @PUT: Especifica que el mètode anotat es accessible mitjançant una petició PUT.
- @DELETE: Especifica que el mètode anotat es accessible mitjançant una petició DELETE.

A més, els paràmetres dels mètodes es poden anotar (entre d'altres) amb:

- @PathParam: Especifica que el valor del paràmetre anotat ha de ser extret de la URI de la petició. El valor de l'anotació identifica el nom del paràmetre usat com a

plantilla en l'anotació @Path de la classe o del mètode en qüestió.

- **@QueryParam:** Especifica que el valor del paràmetre anotat ha de ser extret dels *query parameters* de la URI de la petició. El valor de l'anotació identifica el nom d'un *query parameter*.
- **@DefaultValue:** Especifica un valor per defecte per als paràmetres anotats amb @QueryParam, @MatrixParam, @CookieParam, @FormParam o @HeaderParam. El valor especificat serà usat si el paràmetre especificat no es troba en la URI de la petició.

2.2 Exemple

A continuació es mostra el codi d'una classe Java amb algunes de les anotacions descrites a l'apartat anterior:

```
@Path("articles")
public class ArticlesResource
{
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public ArticleDto create(final ArticleDto article) { }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ArticleDto> getAll(@DefaultValue("false") @QueryParam("stock") final
boolean inStock) { }

    @Path("{id}")
    @DELETE
    public void delete(@PathParam("id") final Long id) { }

    @Path("{id}")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public ArticleDto get(@PathParam("id") final Long id) { }
}
```

Aquesta classe exposa un servei web que admet:

- Peticions POST a la URI /articles amb un document JSON serializable al POJO ArticleDto com a contingut i un POJO del mateix tipus com a resposta.
- Peticions GET a la URI /articles amb un *query parameter* opcional anomenat «stock» i que respon amb una col·lecció de POJOs de tipus ArticleDto serializable a JSON.
- Peticions DELETE a la URI /articles/{id} on el *path parameter* «id» serà mapat a un Long i genera una resposta sense contigut.
- Peticions GET a la URI /articles/{id} on el *path parameter* «id» serà mapat a un Long i respon un POJO ArticleDto serializable a JSON.

2.3 Implementacions

Existeixen moltes implementacions, *providers* segons l'especificació, de JAX-RS. En els següents apartats es descriuran les implementacions que s'han usat com a referent per a la realització d'aquest projecte i les raons per ser escollides.

2.3.1 Jersey

Jersey (https://jersey.java.net/) és un *framework* de codi obert que permet la implementació de serveis web RESTful en Java, incorpora suport per JAX-RS en la versió 1.1 (JSR311) i 2.0 (JSR399). Es considerat per molts com la implementació *de facto* i és per aquest motiu que ha estat escollit com un dels *frameworks* als que donar-hi suport.

2.3.2 Apache Wink

Apache Wink (https://wink.apache.org/) és un *framework* de codi obert que permet la implementació de serveis web RESTful en Java i que incorpora suport per JAX-RS 1.1 (JSR311). Ha estat escollit ja que afegeix un conjunt d'anotacions complementaries molt útils que ajudaran a il·lustrar la flexibilitat del present projecte.

Una de les anotacions més interessants que afegeix és l'anotació @Parent, que permet afegir una referència al recurs pare i facilita d'aquesta manera la resolució de la URI on un cert recurs ha estat mapat. A més, simplifica la jerarquització dels recursos i complementa la funció de l'anotació @Path.

Capítol 3. Anàlisi i disseny del *framework*

3.1 Introducció

La problemàtica que tracta de simplificar el projecte és el manteniment de clients HTTP necessaris per a comunicar-se amb serveis web implementats segons l'API JAX-RS. En concret, el *framework* ha de permetre la generació automàtica d'un client HTTP a partir de l'anàlisi del codi del servidor.

En aquest capítol es descriu en profunditat els requeriments, funcionament i disseny del *framework*.

3.2 Requeriments funcionals

1. El *framework* escanejarà les classes Java que composen el codi d'un servei web amb l'objectiu de detectar els mètodes que exposen recursos REST segons l'API de JAX-RS.
2. El *framework* transformarà els mètodes detectats en la fase d'escaneig en un objecte de domini que descriurà el comportament del mètode escanejat de manera genèrica.
3. El *framework* generarà el codi d'un client HTTP compatible amb el servei web a partir dels objectes generats en la fase de transformació.
4. Les fases d'escaneig i transformació poden dependre de la implementació de l'API JAX-RS emprada pel servei web. Per aquest motiu, el *framework* haurà de ser configurat per a fer servir una implementació concreta d'escàner i de transformador.
5. La fase de generació del client depèn del llenguatge de programació destí. Per aquest motiu, el *framework* haurà de ser configurat per a fer servir una implementació concreta de generador.
6. El *framework* coordinarà les fases de escaneig, transformació i generació del client.
7. El *framework* garantirà que els processos de generació dels client siguin reproduïbles. És a dir, donada una mateixa configuració es generaran els mètodes amb el mateix ordre.
8. El *framework* permetrà accelerar l'escaneig de classes especificant a la configuració el tipus de classe a cercar i el prefix dels *packages* on cercar.
9. El *framework* acceptarà la configuració del directori on el codi del client serà generat.
10. La configuració del *framework* es realitzarà de manera programàtica per tal de facilitar-ne la integració.
11. El *framework* reportarà els possibles errors d'escaneig, transformació i validació

que impedirien la correcta generació del client.

12. El *framework* disposarà d'un *log* on es deixarà constància de les diverses operacions realitzades durant la generació del client.

3.3 Requeriments no funcionals

El *framework* ha de realitzar l'anàlisi de codi Java per tal de cercar les classes JAX-RS que empra el servei web. Per aquest motiu, el desenvolupament es realitzarà en el llenguatge de programació Java usant les capacitats de reflexió que incorpora. En concret s'usarà Java 8 i caldrà una màquina virtual 1.8 o superior.

S'usarà Maven per a la gestió de dependències, per la construcció i per a l'execució de tests del projecte.

El *framework* es distribuirà en forma de llibreria *JAR* amb el nom "jaxrs-client-generator-{versió}.jar".

3.4 Arquitectura

L'arquitectura del projecte es pot descriure com un *pipeline* de tres fases: escaneig, anàlisi i generació on el treball que realitza el *framework* consisteix en coordinar les entrades i sortides de les diferents fases.

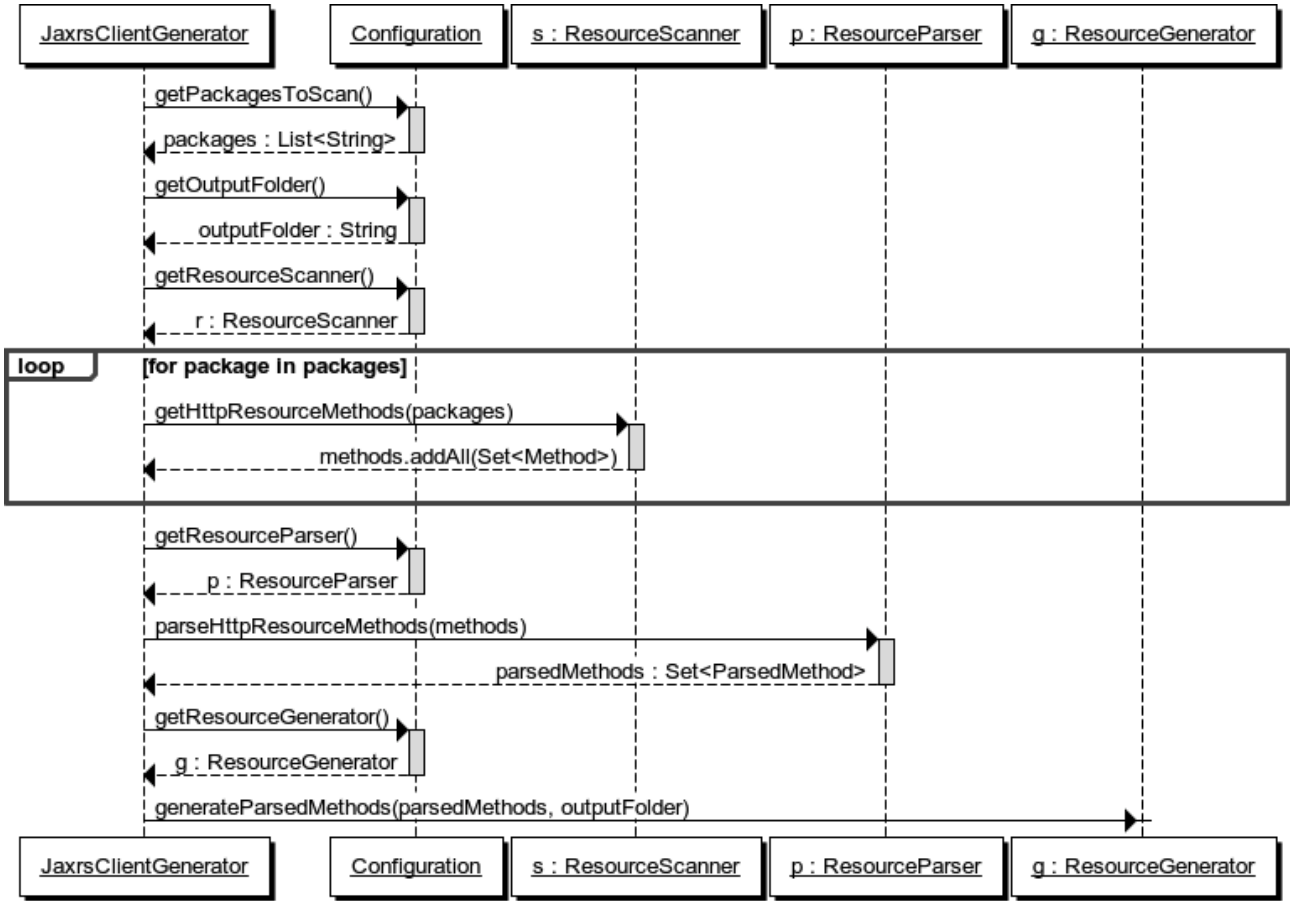
Primerament, la fase d'escaneig rep com a entrada un conjunt de paquets Java a escanejar i retorna com a sortida els mètodes Java (java.lang.reflect.Method) que exposen l'API REST.

A continuació, la fase d'anàlisi rep com a entrada el conjunt de mètodes Java que exposen l'API REST i els transforma en objecte de domini *ParsedMethod.*

Finalment, la fase de generació rep com a entrada la col·lecció de *ParsedMethods* a generar i per cadascun d'ells:

1. Valida que la generació del mètode es podrà dur a terme sense errors
2. Genera els recursos necessaris per a poder construir el client REST
3. Escriu en el sistema de fitxers el conjunt de recursos necessaris per a fer ús del client REST

Per un altra banda, el diagrama de seqüència següent il·lustra el procés, simplificat, de coordinació i *pipelining* durant la generació d'un client REST.



Cal destacar que les classes *ResourceScanner* i *ResourceParser* implementen l'escaneig i la transformació segons l'estàndard JAX-RS. En tot cas ambdues poden ser sobrecarregades per a suportar una implementació de JAX-RS concreta.

Per una altra banda, la classe *ResourceGenerator* és abstracta i la seva implementació depèn del llenguatge de programació destí.

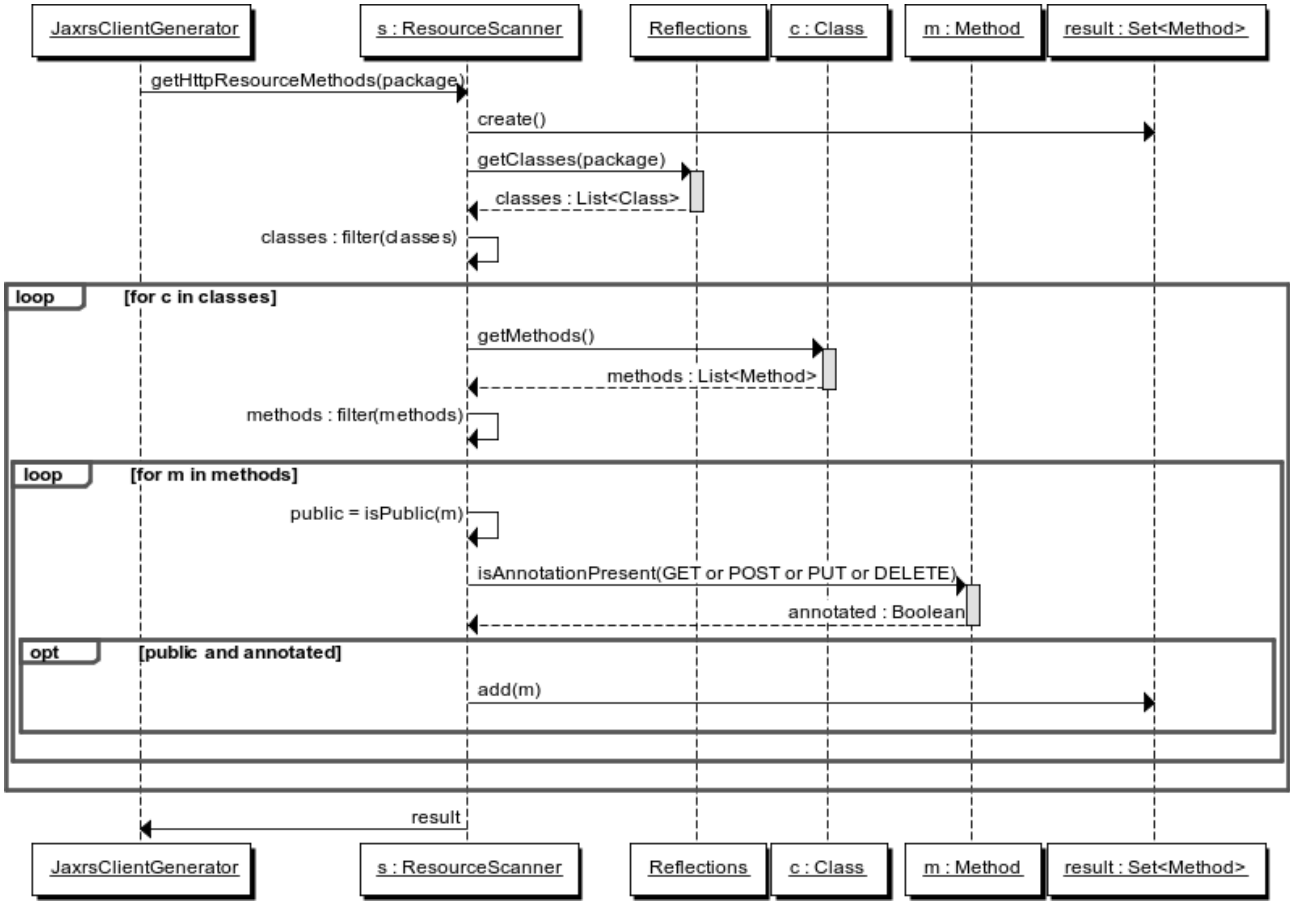
Aquesta divisió de responsabilitats facilitarà la reutilització i combinació dels components, és a dir, diferents implementacions de escàners, *parsers* i generadors podran ser combinats.

Finalment la classe *JaxrsClientGenerator* és l'encarregada de coordinar l'execució del *pipeline* definit a la configuració descrita per una instància de la classe *Configuration.*

3.4.1 Escàner de recursos JAX-RS

La classe *ResourceScanner* és responsable de l'escaneig de paquets de classes Java amb l'objectiu de identificar els mètodes Java (java.lang.reflect.Method) que implementen l'API JAX-RS.

En el següent diagrama de seqüència mostra en que consisteix el procés d'identificació de mètodes JAX-RS i quines la relació amb les entitats estan implicades.



La classe *ResourceScanner* ofereix suport per a la detecció de mètodes JAX-RS segons indica l'especificació. En concret considera els mètodes:

- Públics
- Anotats, necessàriament, amb una de les anotacions: GET, POST, PUT o DELETE

Per a la cerca de mètodes s'empren els mecanismes de reflexió disponibles al llenguatge de programació Java. En concret s'escanegen el conjunt de classes disponibles en el *classpath* del context d'execució actual i s'ofereixen una serie d'opcions per accelerar el procés d'escaneig.

Tal com s'ha indicat en apartats anteriors, la implementació d'un escàner pot dependre de la implementació de JAX-RS que es vulgui suportar. Es per aquest motiu que la classe disposa d'un constructor per defecte i d'un constructor on es poden especificar els predicats per a realitzar el filtratge de classes i mètodes amb l'objectiu de considerar les particularitats de la implementació suportada.

D'altra banda la classe disposa del mètode públic *getHttpResourceMethods* que accepta com a paràmetres:

1. El paquet Java a escanejar
2. Un booleà per indicar si s'ha de realitzar un escaneig recursiu *(true)* o no final *(false)*
3. De manera opcional, el sub-tipus de les classes a considerar durant la cerca

4. De manera opcional, el *ClassLoader* a emprar durant la cerca

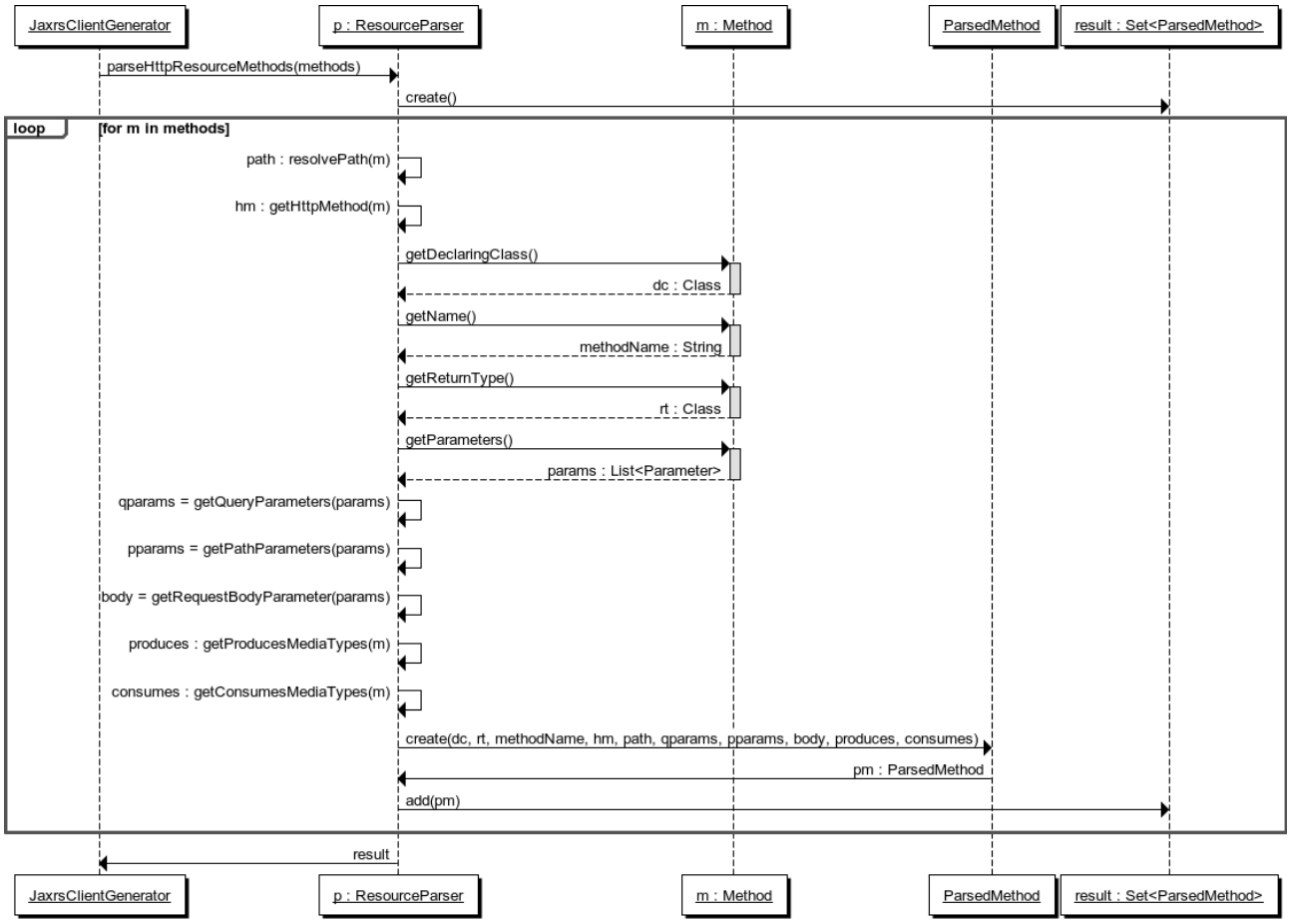
L'especificació del subtipus de les classes a cercar i la possibilitat de fitar la recursivitat de l'escaneig, permet reduir el conjunt de classes a escanejar i millorar, d'aquesta manera, el temps de cerca.

3.4.2 Analitzador de mètodes JAX-RS

La classe *ResourceParser* és responsable de convertir un conjunt de mètodes Java (java.lang.reflect.Method) en un conjunt d'objectes de domini *ParsedMethod.*

L'objecte *ParsedMethod* permet descriure un mètode HTTP independentment de la implementació JAX-RS emprada i és el tipus d'objecte que usarà la classe *ResourceGenerator* per tal de generar, finalment, el client HTTP.

El diagrama següent mostra, de forma simplificada, en que consisteix el procés de transformació.



Per a poder concretar la informació extreta per la classe ResourceParser, es realitzarà l'anàlisi dels atributs que formen part de l'objecte de domini *ParsedMethod*:

- El mètode Java escanejat en la fase anterior
- La classe Java on es troba declarat el mètode

- La classe Java de l'objecte que retorna el mètode en cas que no sigui *void*
- La classe genèrica Java de l'objecte que retorna el mètode en cas que no sigui *void*. Aquest camp es necessari per identificar correctament el tipus d'objectes parametritzats.
- La classe Java de l'objecte que s'espera com a *body en* la crida HTTP en cas que el paràmetre es trobi declarat. De tots els paràmetres del mètode aquest és l'únic que es presenta sense anotacions JAX-RS.
- El *path* HTTP complet on es troba mapat el mètode en qüestió.
- El verb HTTP de la demanda: *GET*, *POST*, *PUT* o *DELETE*
- El nom del mètode
- La llista de *query parameters* que accepta la demanda. Aquests paràmetres es troben anotats amb *@QueryParam* a la signatura del mètode.
- La llista de path *parameters* que accepta la demanda. Aquests paràmetres es troben anotats amb *@PathParam* a la signatura del mètode.
- La llista de *MediaTypes* que consumeix el recurs REST. Els elements d'aquesta llista corresponen als valors de l'anotació *@Consumes.*
- La llista de MediaTypes que produeix el recurs REST. Els elements d'aquesta llista corresponen als valors de l'anotació *@Produces.*

La transformació que es realitza en aquesta fase és útil ja que la majoria d'operacions que es realitzen són comunes per a les diverses implementacions de la tercera, i última, fase que representa la classe *ResourceGenerator*. D'aquesta manera s'afavoreix la reutilització i no duplicació de codi.

3.4.3 Resolució de "Paths"

Un dels atributs de l'objecte de domini *ParsedMethod* és el *path*, el qual consisteix en la plantilla completa de la *URL* on es troba mapat el mètode JAX-RS. Considerem per exemple el següent codi Java:

```
@Path("articles")
public class ArticlesResource
{
  @Path("{id}")
  @DELETE
  public void delete(@PathParam("id") final Long id) {...}
}
```

En aquest cas i seguint l'especificació, el *path* complet on es troba mapat el mètode *delete* consisteix en el resultat de concatenar el valor de l'anotació *@Path* de la classe amb el valor de la mateixa anotació al mètode: */articles/{id}*

Existeixen, però, implementacions de JAX-RS que afegeixen anotacions per tal de simplificar la gestió del *paths*. Per donar suport a les peculiaritats d'aquestes implementacions existeix la interfície *PathResolver* que donat un mètode Java i la classe

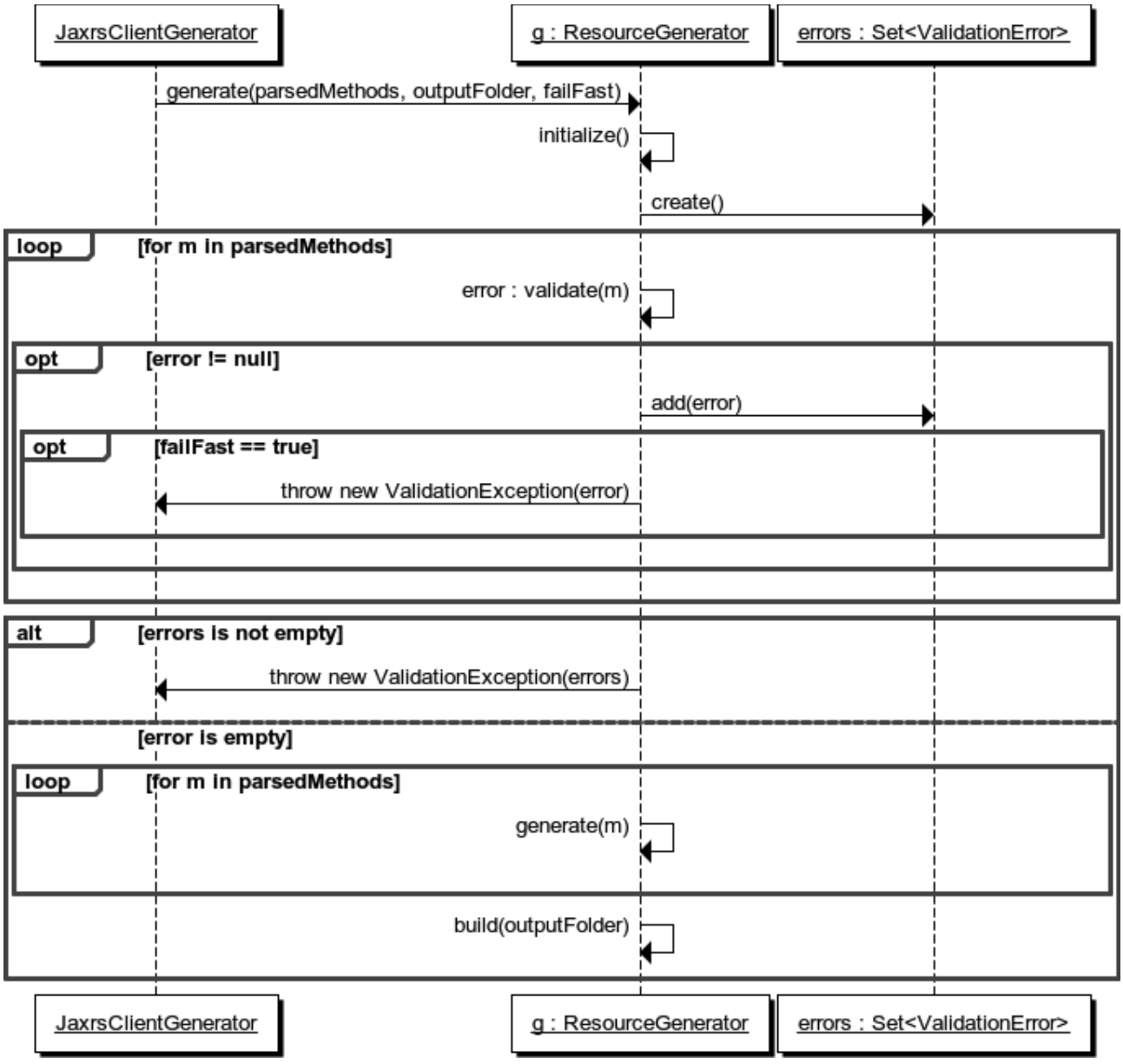
on es troba declarat retorna la plantilla del *path* corresponent.

Cal fer notar que la classe *ResourceParser* empra per defecte la implementació *JaxrsPathResolver* de la interfície *PathResolver* , la qual segueix l'estàndard definit a JAX-RS. Però es pot especificar, fent ús del constructor adequat, la implementació de *PathResolver* que s'ha d'emprar durant la fase de transformació.

3.4.4 Generador del client HTTP

La classe *ResourceGenerator* és responsable de generar el codi del client HTTP en base als objectes generats per l'analitzador.

Disposa del mètode públic *generate* que rep el conjunt de *ParsedMethods* a generar, el directori on s'ha de generar el codi del client i el mode de validació. El següent diagrama de seqüència il·lustra el procés de generació del client.



La classe disposa de quatre mètodes abstractes:

- EL mètode *initialize* ha de contindre tot el codi d'inicialització necessari.
- El mètode *validate* és l'encarregat d'indicar si un *ParsedMethod* pot ser generat correctament, en cas negatiu retornarà el conjunt d'errors de validació trobats i en cas positiu retornarà un conjunt buit d'errors.
- El mètode *generate* és l'encarregat de generar el codi corresponent al *ParsedMethod* indicat per paràmetre.
- El mètode *build* és l'encarregat de desar al directori de destí indicat els fitxers corresponents al client HTTP generat. De fet, es delega a criteri de l'implantador la manera d'acumular els mètodes generats abans de ser desats en el directori de destí.

En aquest cas no s'ofereix cap implementació per defecte, es tracta d'una classe abstracta i les implementacions depenen completament del llenguatge destí i del tipus d'ús que es vulgui donar al codi generat. De fet, el resultat final no té per que ser un client HTTP es podria, per exemple, implementar un generador que produís una documentació de l'API REST exposada de manera automàtica.

3.4.5 Configuració del framework

Per a configurar el *framework* cal inicialitzar l'objecte *JaxrsClientGenerator* amb la instància d'un objecte que implementi la interfície *Configuration*. En cap cas es força que la configuració hagi d'estar definida en un cert tipus fitxer o en certes propietats.

Aquesta estratègia de configuració permet al desenvolupador que vulgui integrar el *framework* d'escollir la manera de gestionar la configuració que millor s'adapti al seu projecte.

La interfície de configuració consta dels següents mètodes:

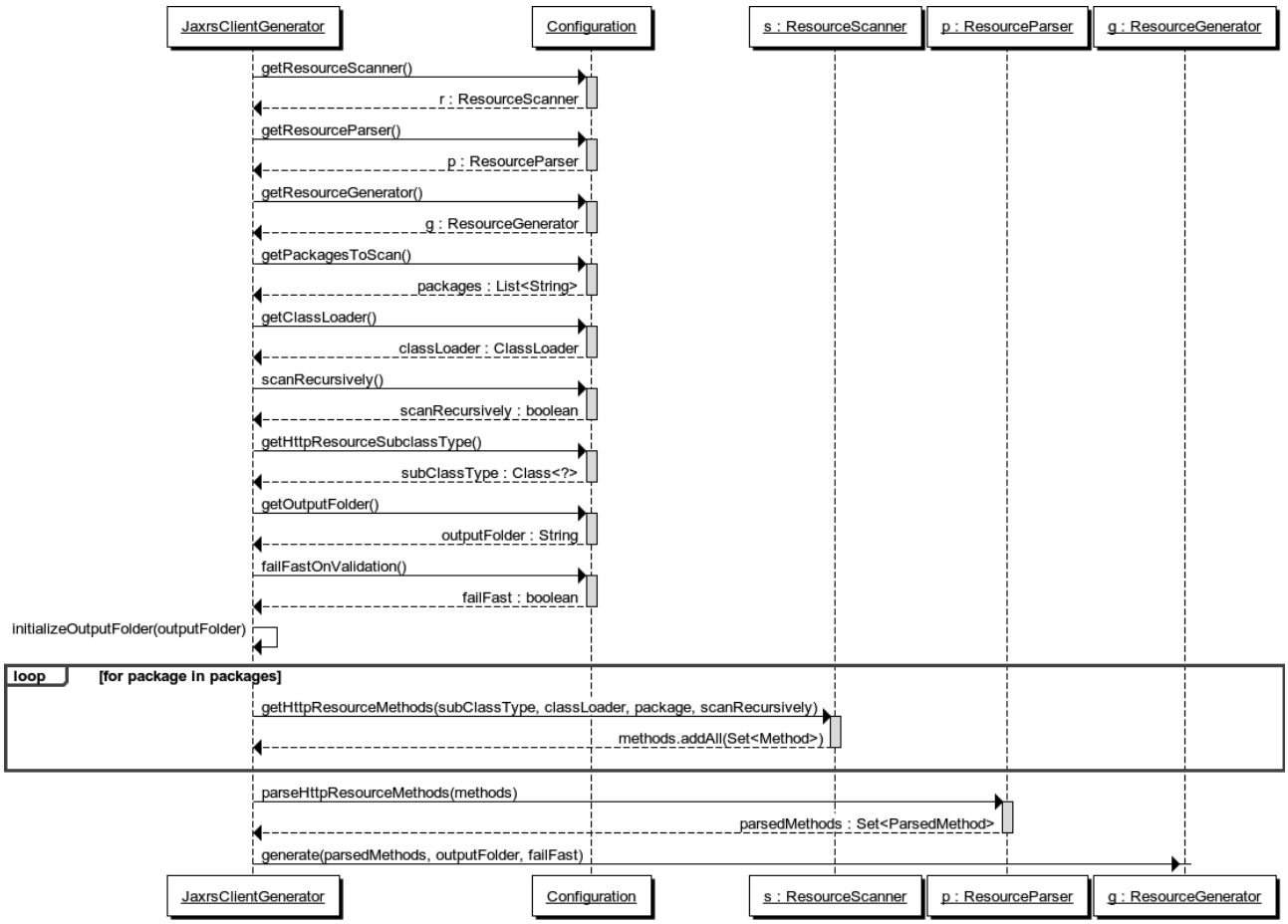
- *getResourceSanner()*, que retorna la instancia de l'escàner de recursos JAX-RS a emprar.
- *getPackagesToScan()*, que indica el conjunt de paquets Java en els quals cercar mètodes JAX-RS.
- *getClassLoader()*, que indica quin *ClassLoader* emprar per a realitzar l'escaneig. Es tracta d'un paràmetre opcional i per defecte s'empra el *ClassLoader* del context actual.
- *getHttpResourceSubclassType(),* indica el subtipus que han de tindre les classes a cercar. Es tracta d'un paràmetre opcional i per defecte es cerquen objectes de tipus *Object.*
- *scanRecursively(),* si retorna *true* el framework escanejarà cadascun dels paquets i subpaquets indicats de manera recursiva. Si retorna *false* únicament s'escanejarà el paquet indicat. Es tracta d'un paràmetre opcional i per defecte retorna *true.*
- *getResourceParser()*, que retorna la instància de l'analitzador de mètodes JAX-RS a emprar.

- *getResourceGenerator()*, que retorna la instància del generador de client HTTP a emprar.
- *getOuputFolder()*, que indica directori destí on generar el codi del client HTTP.
- *failFastOnValidation()*, si retorna *true* el procés de validació es pararà en el moment que el primer error es detecti. En cas contrari es realitzarà la validació de tots els mètodes abans de parar l'execució del procés. Es tracta d'un paràmetre opcional i per defecte retorna *false*.

Per a facilitar la creació i la validació del atributs requerits en la configuració, el *framework* disposa de la classe *ConfigurationBuilder* la qual implementa la interfície *Configuration* alhora que implementa una interfície fluida que valida els atributs requerits per la configuració. En altres paraules, no cal crear implementacions concretes de la classe *Configuration* si es fa ús de la classe *ConfigurationBuilder*.

3.4.6 Coordinador del "pipeline"

La classe *JaxrsClientGenerator* és el punt d'entrada del *framework* i la classe responsable de coordinar les diferents fases que que formen part del procés de generació amb l'objectiu d'obtindre una construcció reproduïble segons una configuració. En el diagrama de seqüencia següent és pot veure, en detall, les diferents operacions que coordina:



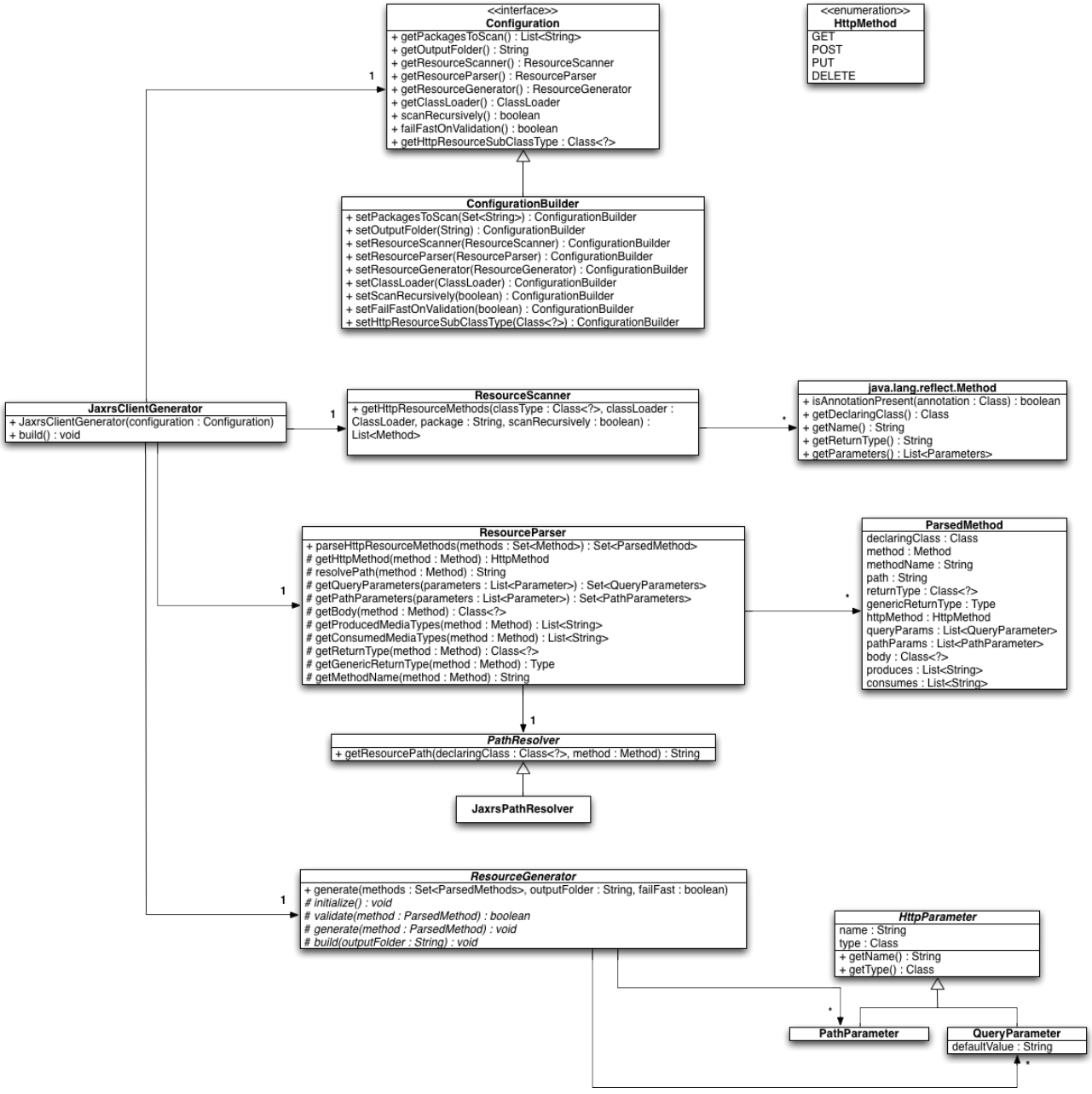
En conjunt, les operacions que coordina són:

1. Lectura de la configuració a emprar
2. Inicialització del directori destí, en concret elimina el contingut o crea el directori destí.
3. Ordenació alfabètica dels *packages* Java que han de ser escanejats segons la configuració.
4. Per a cada *package* a escanejar invoca al *ResourceScanner* indicat per la configuració amb l'objectiu d'obtindre el mètodes Java que exposen l'API REST.
5. Per a cada mètode obtingut en el punt anterior, invoca al *ResourceParser* indicat per la configuració amb l'objectiu d'obtindre la llista de *ParseMethods* a considerar en la següent fase.
6. Per a cada *ParsedMethod* obtingut en el punt anterior, invoca al *ResourceGenerator* indicat per a la configuració.

Tanmateix, qualsevol excepció que es produeixi durant el procés de generació es propaga de manera que el procés de generació s'interromp.

3.5 Diagrama de classes

A continuació es mostra el diagrama amb les classes i mètodes més rellevants del disseny segons el descrit en els apartats anteriors.



Capítol 4. Introducció a Apache Maven

Apache Maven és una eina de programari lliure que pretén millorar la gestió de la construcció d'un projecte Java i les seves dependències.

Es basa en l'ús d'un fitxer XML anomenat POM (sigles de Project Object Model) el qual descriu el projecte a construir, les seves dependències amb altres llibreries, l'ordre de construcció i els *plug-ins* requerits.

D'altra banda, les dependències i *plug-ins* son descarregats directament d'un o més repositoris en línia i es desa una copia a la màquina local.

A continuació es mostra l'estructura d'un fitxer POM on els elements *groupId*, *artifactId* i *version* constitueixen l'identificador únic (o coordenades) d'aquest projecte i la secció *dependencies* descriu, amb les respectives coordenades, les llibreries necessàries per a poder construir el projecte.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

La majoria de les funcionalitats de Maven són incloses mitjançant plug-ins. Existeixen, per exemple, plug-ins per a indicar la versió del compilador de Java a utilitzar o plug-ins per a configurar l'execució de tests unitaris i funcionals. Els plug-ins s'han de configurar en la secció plug-ins del POM.

```
<plug-ins>
    <plug-in>
        <groupId>org.apache.maven.plug-ins</groupId>
        <artifactId>maven-compiler-plug-in</artifactId>
        <version>3.1</version>
        <configuration>
            <source>1.8</source><target>1.8</target>
        </configuration>
    </plug-in>
</plug-ins>
```

Maven defineix els cicles de vida: *default*, *clean* i *site*. Cadascun d'ells està format per diferents fases que s'executen en ordre.

Així mateix, el cicle de vida *default* és el que conté, entre d'altres, les fases de construcció a les que els desenvolupadors estan més acostumats: *compile*, *test*, *package* i *install*.

Qualsevol de les fases dels cicles de vida pot ser iniciada des de la línia de comandes. Cal tindre en compte, però, que s'executaran totes les fases prèvies a la especificada. Per exemple, si executem "mvn clean install", maven executarà el cicle de vida *clean* i les fases del cicle de vida *default* previs a la fase *install*.

Capítol 5. Anàlisi i disseny del *plug-in* Maven

5.1 Introducció

El *plug-in* permet fer ús del *framework* des de fitxers POM i integrar d'aquesta manera la generació de clients REST en el cicle de construcció d'un projecte Java gestionat amb Maven. Requereix com a dependència el *framework* jaxrs-client-generator i s'executa en la fase *generate-sources* del cicle de vida *default.*

5.2 Requeriments funcionals

1. El *plug-in* s'ha de poder integrar en el cicle de construcció d'un projecte Maven en la fase *generate-sources*.
2. El *plug-in* emprarà el *framework* jaxrs-client-generator per a generar el codi del client HTTP corresponent a un servei web.
3. L'únic paràmetre de configuració obligatori ha de ser el *qualified name* de la classe de configuració a emprar per inicialitzar el *framework*.
4. El *plug-in* escriurà en un log les diverses operacions que està realitzant.

5.3 Requeriments no funcionals

S'usarà Java 8 per a la implementació del *plug-in* i Maven per a gestionar la construcció del *plug-in*.

El codi font es troba al projecte jaxrs-client-generator-maven-plugin i es distribueix com a llibreria JAR amb el nom jaxrs-client-generator-maven-plugin-{versió}.jar.

5.4 Arquitectura

L'arquitectura d'un *plug-in* de Maven és estandard i consisteix en:

- Una implementació de la classe *AbstractMojo* en la qual s'ha d'implementar la lògica del *plug-in* en el mètode sobrecarregat *execute()*
- Cadascun dels atributs del plug-in que s'han de poder configurar en el POM ha d'estar declarat com a atribut de la classe que implementa *AbstractMojo* i anotat amb l'anotació *@Parameter*. Aquesta anotació permet indicar el nom de la propietat en el POM, si es requerida i el seu valor per defecte entre d'altres atributs.

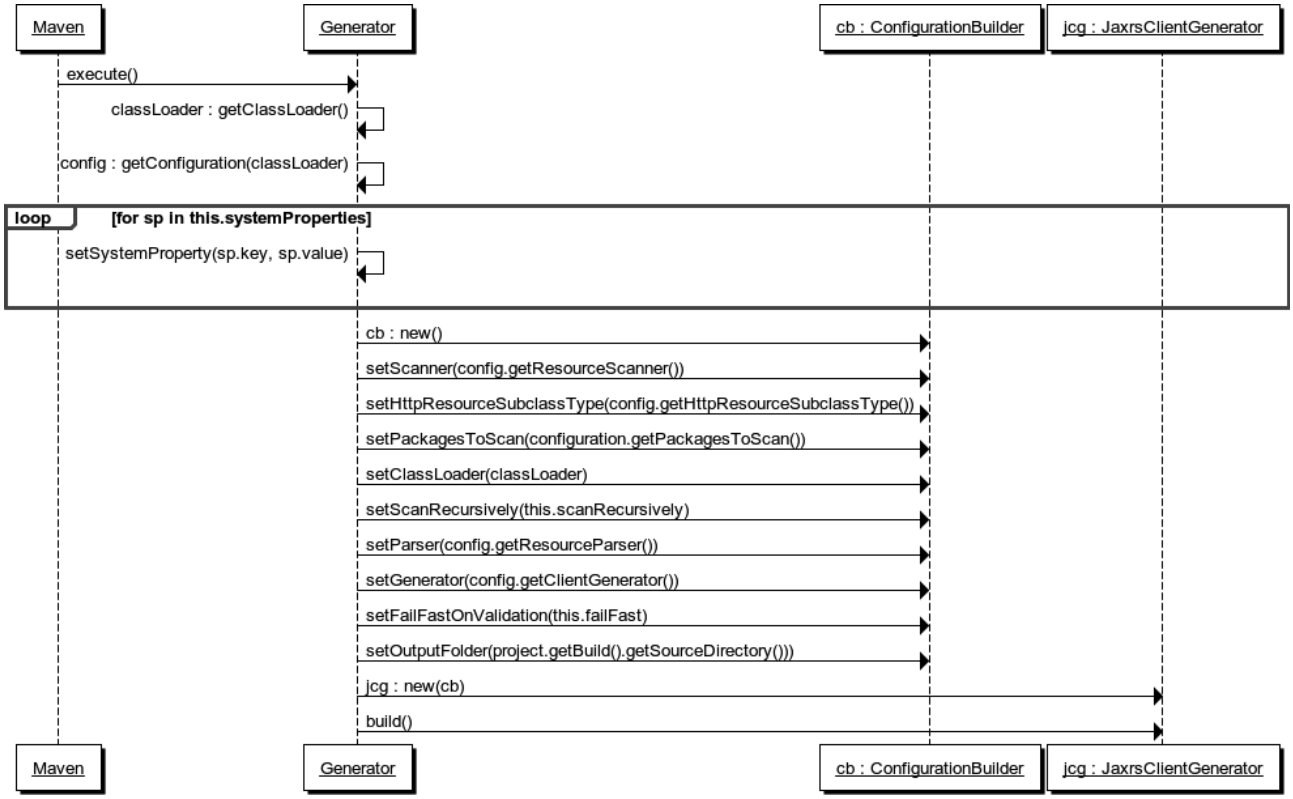
Pel cas que ens ocupa, el *plug-in* s'implementarà a la classe *Generator* i el mètode *execute()* serà l'encarregat de crear una nova instancia de la classe *JaxrsClientGenerator*

amb l'objectiu de iniciar el procés de generació del client HTTP.

Els atributs de configuració de que disposa el *plug-in* són:

- El *qualified name* de la classe de configuració que s'ha d'instanciar per tal d'inicialitzar el *framework.*
- Un booleà per indicar el valor del paràmetre de configuració *scanRecursively* del *framework* i que per defecte s'inicialitza a *true*.
- Un booleà per indicar el valor del paràmetre de configuració *failFastOnValidation* del *framework* i que per defecte s'inicialitza a *false.*
- Un conjunt de propietats del sistema que han de ser inicialitzades abans d'iniciar el procés de generació. Aquest mecanisme permet especificar atributs de configuració que queden fora de l'abast de la configuració estàndard del *framework*.

En el diagrama de seqüència següent es mostra el funcionament, en detall, del mètode *execute*.



En primer lloc, el mètode *getClassLoader* construeix un *ClassLoader* a mida basat en les dependències de l'actual projecte Maven, aquest *ClassLoader* és el que s'usarà per a instanciar la classe de configuració i per a realitzar el posterior escaneig de classes per part del *framework*.

A continuació, el mètode *getConfiguration* crea una nova instancia de la classe de configuració del *framework* especificada en el POM i s'inicialitzen les propietats del

sistema especificades.

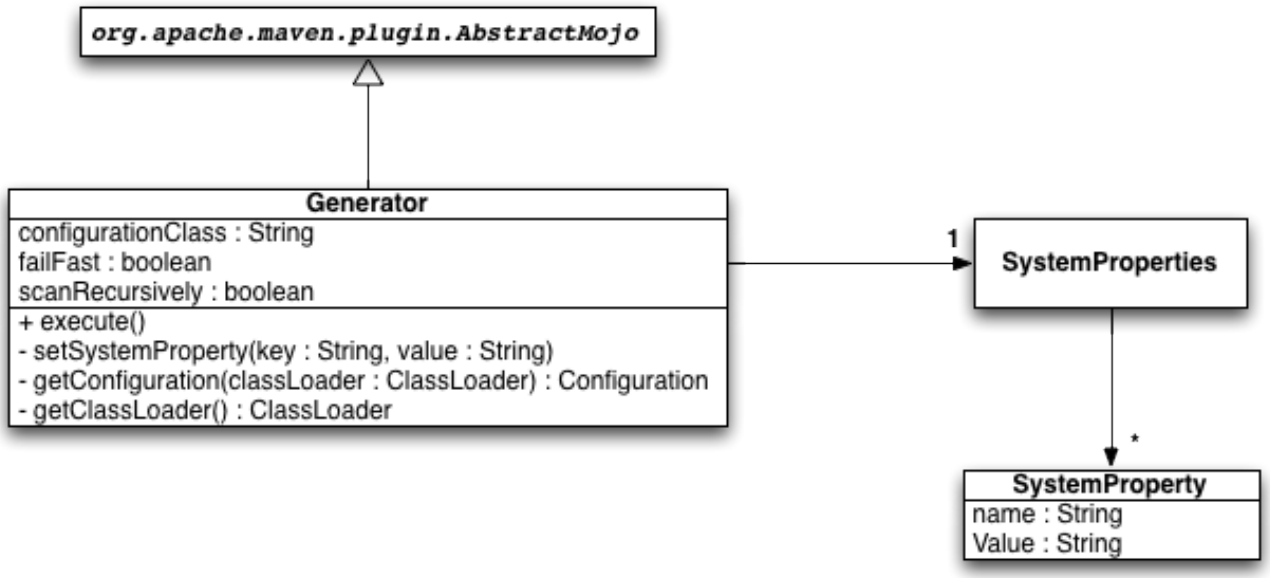
Tot seguit, s'usa la classe *ConfigurationBuilder* per a crear una nova configuració partint de la recentment instanciada i fixant els valors de:

- El *ClassLoader* a usar durant l'escaneig de mètodes REST
- El valor de l'atribut scan*Recursively*
- El valor de l'atribut *failFastOnvalidation*
- El directori on es desaran els fitxers resultants de la generació serà el directori *src* del projecte Maven

Finalment es crea una nova instancia de la classe *JaxrsClientGenerator* amb la configuració recent creada i s'invoca al mètode *build* per a començar el procés de generació.

5.5 Diagrama de classes

El diagrama de classes és molt simple. Cal fer notar que l'objecte *Configuration* retornat pel mètode *getConfiguration* es correspon al descrit en l'apartat 3.4.5 d'aquesta memòria.



Capítol 6. Generador de codi Java

Com a exemple d'implementació de la classe ResourceGenerator, s'ha desenvolupat un generador de clients REST per al llenguatge de programació Java. El codi es troba al projecte java-json-generator i es distribueix en forma de llibreria Jar amb el nom java-json-generator-{versió}.jar.

L'objectiu principal és disposar d'un generador més aviat didàctic que no pas complert, és per aquest motiu que la solució desenvolupada presenta les següents limitacions:

- Tots els mètodes a generar han de tindre una signatura única en el conjunt de mètodes que exposen l'API REST.
- L'únic MediaType suportat és a*pplication/json* tant des del punt de vista de la consumició com de la producció.
- Els mètodes anotats amb @POST i @PUT han d'indicar el cos (body) de la demanda com a paràmetre de la signatura.
- El valor de l'anotació @Path en las classes i mètodes que exposen l'API REST, no pot contindre expressions regulars. El valor de l'anotació només s'usarà com a plantilla dels paràmetres anotats amb l'anotació @PathParameter.

El generador consisteix bàsicament en:

1. La generació d'una única interfície Java que conté tots els mètodes que exposen l'API REST del servidor web amb les seves corresponents anotacions JAX-RS.
2. Un client HTTP que intercepta les crides als mètodes de la interfície generada i executa la demanda que descriuen les anotacions JAX-RS del mètode invocat.

6.1 Anàlisi del client HTTP

El client HTTP està format per les classes:

- ApiClient: La classe que conté el client HTTP pròpiament i implementa la interfície *InvocationHandler* amb l'objectiu de interceptar les crides REST.
- AuthorizationException: Excepció emesa per l'ApiClient en cas que el codi HTTP retornat per servidor sigui 401 (*Unauthorized*) o 403 *(Forbiden).*
- HttpException: Excepció emesa per l'ApiClient en cas que el codi HTTP retornat per servidor sigui més gran o igual que 400.
- Json: Classe d'utilitat per a la serialització i deserialització d'objectes amb JSON

Internament s'usa el client HTTP de codi lliure okhttp (https://github.com/square/okhttp) i s'incorpora suport per a Basic Authentication.

La implementació de l'*InvocationHandler* de la classe ApiClient consisteix en interceptar, únicament, les crides a mètodes que es troben anotats amb una de les anotacions de JAX-RS: GET, POST, PUT o DELETE i, en cas afirmatiu, construir una demanda HTTP segons les

anotacions JAX-RS presents en el mètode i executar-la contra el servidor.

Per tal de construir les demandes HTTP de manera correcta s'usen els mecanismes de reflexió de Java per a:

- Construir la URL de la demanda realitzant la substitució dels valors dels paràmetres anotats amb *@PathParam* a la plantilla indicada pel valor de l'anotació *@Path.*
- Concatenar a la URL els valors dels paràmetres anotats amb *@QueryParam*
- Afegir a la demanda la capçalera (*Header*) *Accept* si l'anotació *@Produces* està present.
- Afegir a la demanda la capçalera (*Header*) *Content-type* si l'anotació *@Consumes* està present.
- Cercar el cos de la demanda *(body)* d'entre els paràmetres del mètode en cas de que es tracti de demandes POST o PUT
- Cercar el tipus de la classe de retorn en cas que en retorni alguna

6.2 Anàlisi del generador

Tal com s'ha descrit anteriorment, l'objectiu d'aquest generador és la creació d'una única interfície Java que contingui el conjunt de mètodes que exposen l'API REST del servidor web amb les seves corresponents anotacions JAX-RS.

Durant el procés de generació el conjunt de mètodes de la interfície s'emmagatzema en memòria i és en l'etapa final quan s'escriu a disc el codi de la interfície.

6.2.1 Validació de mètodes

El procés de generació (apartat 3.4.4 d'aquesta memòria) realitza la validació de tots els mètodes abans d'iniciar-ne la generació. En aquesta implementació concreta el procés de validació consisteix en verificar que cap dels mètodes infringeix les limitacions descrites en la introducció d'aquest capítol.

Cal destacar que si algun mètode infringeix alguna de les limitacions, el generador retorna i *loga* un error descrivint clarament el problema. Per exemple, en cas que s'infringeixi la restricció de la signatura única, el generador retornarà un error de validació que contindrà les classes que declaren el mètode amb la mateixa signatura.

La precisió en la descripció dels errors de validació es clau per a poder fer un ús efectiu del *framework* i, en el cas dels generadors, és una responsabilitat que recau en l'implantador.

6.2.2 Generació de mètodes

El procés de generació consisteix en convertir cadascun dels objectes ParsedMethod en mètodes Java que formin part d'una mateixa interfície Java.

L'aspecte d'aquests mètodes serà molt similar al que presentaven en la implementació del servidor excepte que:

- L'anotació @Path tindrà com a valor el *path* complet on es troba mapat el mètode

- Els únics paràmetres presents seran: els anotats amb @PathParam o @QueryParam i el que no està anotat amb cap anotació JAX-RS.
- El tipus dels paràmetres anotats amb *@QueryParam* serà *java.util.Optional* parametritzat amb el tipus original del paràmetre.
- Els noms dels paràmetres es veuran alterats per tal de fer-los coincidir amb el valor, normalitzat, de les anotacions.

La generació del codi de la interfície es realitza mitjançant la llibreria de codi obert jcodemodel (https://github.com/phax/jcodemodel).

6.3 Ús en projectes Java

Les dependències que ha d'incloure un projecte Java que vulgui emprar el client generat són:

- La llibreria que conté el generador *java-json-generator* així com les seves dependències transitives
- La llibreria que conté la interfície generada o una copia de la interfície

Tal com es pot deduir cal usar la interfície generada conjuntament amb la classe *ApiClient* disponible a la llibreria del generador. En concret, la classe *ApiClient* disposa de dos mètodes estàtics que simplifiquen la creació d'una classe *Proxy* per a la interfície generada i les invocacions de la qual estan manegades per una instància de la classe *ApiClient.*

Per exemple, suposem que tenim aixecat un servidor web en la nostra màquina escoltant al port 8080 i que hem usat el generador per tal d'obtindre la interfície *Api.java*. El següent codi construirà un *Proxy* de la interfície generada gestionat per una instància de l'*ApiClient*:

```
Api api = ApiClient.newInstance("http://localhost:8080/demo", Api.class);
```

A partir d'aquest moment, invocaríem els mètodes disponibles a la interfície com si es tractes d'una interfície Java normal, només que l'*ApiClient* s'encarregarà de realitzar les crides HTTP corresponents de manera totalment transparent a l'usuari.

```
Enterprise enterprise = new Enterprise();
enterprise.setName("e1");
enterprise.setAddress("a1");
enterprise = api.createEnterprise(enterprise); // POST al servidor

api.listAllEnterprises(Optional.empty(), Optional.empty()); // GET al servidor
```

Capítol 7. Servei REST de prova

Per tal de demostrar i comprovar el bon funcionament del *framework* s'han desenvolupat dos serveis web de proba, un implementat amb Jersey i l'altre amb Apache Wink.

Encara que es tracta de dos serveis web diferents, exposen la mateixa API i implementen la mateixa lògica. En el següents apartats és mostrarà l'especificació del servei i les diferències entre les dues implementacions realitzades.

7.1 Especificació

La idea principal és exposar a l'API del servei web el CRUD (Create, Read, Update i Delete) del model de dades següent:

A continuació s'enumeren els *endpoints* que ha de presentar l'API per a realitzar el CRUD de l'entitat Enterprise:

Creació	Verb	POST
	URL	/enterprises
	Body	Enterprise
	PathParams	-
	QueryParams	-
	Produces	applicacion/json
	Consumes	applicacion/json
Llistat	Verb	GET
	URL	/enterprises
	Body	-
	PathParams	-
	QueryParams	- name: filtre d'empresa per nom - address: filtre d'empresa per adreça
	Produces	applicacion/json
	Consumes	-
Lectura	Verb	GET
	URL	/enterprises/{id}
	Body	-
	PathParams	- id: identificador de l'empresa

	QueryParams	-
	Produces	applicacion/json
	Consumes	-
Modificació	Verb	PUT
	URL	/enterprises/{id}
	Body	Enterprise
	PathParams	- id: identificador de l'empresa a modificar
	QueryParams	-
	Produces	applicacion/json
	Consumes	applicacion/json
Eliminació	Verb	DELETE
	URL	/enterprises/{id}
	Body	-
	PathParams	- id: identificador de l'empresa a eliminar
	QueryParams	-
	Produces	-
	Consumes	-

A continuació s'enumeren els *endpoints* que ha de presentar l'API per a realitzar el CRUD de l'entitat Employee:

Creació	Verb	POST
	URL	/enterprises/{enterprise_id}/employers
	Body	Employee
	PathParams	- enterprise_id: l'identificador de l'empresa on treball l'empleat
	QueryParams	-
	Produces	applicacion/json
	Consumes	applicacion/json
Llistat	Verb	GET
	URL	/enterprises/{enterprise_id}/employers
	Body	-
	PathParams	- enterprise_id: l'identificador de l'empresa on treball l'empleat
	QueryParams	-
	Produces	applicacion/json
	Consumes	-
Lectura	Verb	GET
	URL	/enterprises/{enterprise_id}/employers/{id}

	Body	-
	PathParams	- enterprise_id: l'identificador de l'empresa on treball l'empleat - id: l'identificador de l'empleat a cercar
	QueryParams	-
	Produces	applicacion/json
	Consumes	-
Modificació	**Verb**	PUT
	URL	/enterprises/{enterprise_id}/employers/{id}
	Body	Employee
	PathParams	- enterprise_id: l'identificador de l'empresa on treball l'empleat - id: l'identificador de l'empleat a modificar
	QueryParams	-
	Produces	applicacion/json
	Consumes	applicacion/json
Eliminació	**Verb**	DELETE
	URL	/enterprises/{enterprise_id}/employers/{id}
	Body	-
	PathParams	- enterprise_id: l'identificador de l'empresa on treball l'empleat - id: l'identificador de l'empleat a eliminar
	QueryParams	-
	Produces	-
	Consumes	-

Com es pot observar en les URLs, la gestió de l'entitat *Employee* s'ha mapat com un subrecurs de l'entitat *Enterprise*. En altres paraules, qualsevol de les URLs de la gestió d'empleats inclou de manera implícita la URL d'una empresa en concret.

D'aquesta manera s'ha establert un jerarquia on el recurs encarregat de la gestió d'empreses és el "pare" del recurs encarregat de la gestió d'empleats.

7.2 Implementació

Ja que es tracta d'un servei web amb propòsits demostratius i que la part rellevant és l'API exposada, l'estat del domini no es persisteix en base de dades sinó que es manté en memòria. Aquesta decisió d'implementació provoca que al reiniciar el servidor l'estat del servidor desapareix.

En els apartats següents s'exposen els detalls més rellevants de les dues implementacions

proposades.

7.2.1 Jersey

La implementació del servei web amb *Jersey* es troba en el projecte anomenat *jersey-demo-app* i es distribueix en format WAR (*Web Application Archive*).

Les classes que exposen l'API REST són:

- *com.abiquo.jersey.demo.app.service.EnterpriseService*
- *com.abiquo.jersey.demo.app.service.EmployeeService*

A més de contindre els mètodes JAX-RS aquestes classes s'encarreguen del manteniment del domini de dades.

En el paquet *com.abiquo.jersey.demo.app.model* es troben les classes de model i transport del servidor.

Jersey es considera la implementació *de facto* de JAX-RS i tal com es pot comprovar en les classes de servei, les classes i mètodes es troben anotats tal com indica l'especificació JAX-RS.

Per altra banda, en el paquet *com.abiquo.jersey.demo.app.client* es troba la classe *JerseyClientGeneratorConfig* que implementa la interfície de configuració del *framework*. Tal com s'ha explicat, la implementació d'aquest servei web segueix l'estàndard de JAX-RS es per això que si s'analitza la classe de configuració es pot veure que usa:

- L'escàner de recursos proporcionat pel *framework*
- El *parser* de recursos i el *PathResolver* proporcionat pel *framework*
- El generador *java-json-generator*

7.2.2 Apache Wink

La implementació del servei web amb *Apache Wink* es troba en el projecte anomenat *wink-demo-app* i es distribueix en format WAR (*Web Application Archive*). Com es pot comprovar, aquesta implementació és equivalen a l'anterior en quant a organització de paquets i a la gestió del domini.

Les classes que exposen l'API REST són:

- *com.abiquo.wink.demo.app.service.EnterpriseService*
- *com.abiquo.wink.demo.app.service.EmployeeService*

En el paquet *com.abiquo.wink.demo.app.model* es troben les classes de model i transport del servidor.

Apache Wink incorpora anotacions pròpies i complementaries a JAX-RS. En aquest cas concret s'ha usat l'anotació *@Parent* a nivell de classe per a simplificar la gestió de *paths* i per fer explícita la jerarquia entre les dues classes de servei.

Per altra banda, en el paquet *com.abiquo.wink.demo.app.client* es troben les classes:

- *WinkClientGeneratorConfig:* Implementació la interfície de configuració del *framework*
- *WinkPathResolver*: Implementació de la interfície *PathResolver* per a incorporar la resolució de *paths* on estigui present l'anotació *@Parent*

Si s'analitza la classe de configuració es pot veure que:

- Usa l'escàner de recursos proporcionat pel *framework*
- Usa el *parser* de recursos proporcionat pel *framework* però amb el *WinkPathResolver.*
- Usa el generador *java-json-generator*

Capítol 8. Integració amb Maven

En aquest capítol es mostrarà un exemple de generació d'un client REST fent ús del *plug-in* Maven, dels serveis web de probes i del *framework* desenvolupats. La integració amb *Jersey* es troba al projecte *jersey-demo-app-client* i la integració amb *Apache Wink* al projecte *wink-demo-app-client*.

En concret es mostrarà la generació del client REST per al servei web implementat amb *Jersey*, ja que el procediment és molt similar per ambdues implementacions.

Si s'examina el projecte *jersey-demo-app-client* es pot veure que només conté un fitxer *pom.xml*. Aquest fitxer conté les directrius per a la generació i compilació del client REST usant Maven:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
```

En primer lloc es declaren les coordenades del projecte i com s'ha d'empaquetar, en aquest cas el resultat de la compilació serà una llibreria JAR:

```
  <modelVersion>4.0.0</modelVersion>
   <groupId>com.abiquo</groupId>
   <artifactId>jersey-demo-app-client</artifactId>
   <version>1.0.0</version>
   <packaging>jar</packaging>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>
```

A continuació es declaren les dependències del projecte, en aquest cas necessitem les classes del servei web a generar i del *framework* jaxrs-client-generator. Cal remarcar que no cal incloure la dependència al generador java-json-plug-in ja que és inclosa de manera transitiva per la dependència a les classes del servidor web.

```
   <dependencies>
      <dependency>
         <groupId>com.abiquo</groupId>
         <artifactId>jersey-demo-app</artifactId>
         <version>1.0.0</version>
         <classifier>classes</classifier>
      </dependency>
      <dependency>
         <groupId>com.abiquo</groupId>
         <artifactId>jaxrs-client-generator</artifactId>
         <version>1.0.0</version>
       </dependency>
   </dependencies>
```

Per últim cal configurar els *plug-ins* que s'executaran durant el procés de construcció del projecte:

```
   <build>
     <plug-ins>
```

Per una part s'usa el maven-compiler-plug-in per indicar que el projecte requereix Java 8

per a ser construït:

```
  <plug-in>
    <artifactId>maven-compiler-plug-in</artifactId>
    <version>2.3.1</version>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
    </configuration>
  </plug-in>
```

Per una altra banda cal configurar el *plug-in* encarregat de la generació del client:

```
  <plug-in>
    <groupId>com.abiquo</groupId>
    <artifactId>jaxrs-client-generator-maven-plug-in</artifactId>
    <version>1.0.0</version>
```

El *plug-in* s'executarà en la fase *generate-sources*, es a dir, abans de la fase de compilació:

```
    <executions>
      <execution>
        <phase>generate-sources</phase>
        <goals>
          <goal>generate</goal>
        </goals>
      </execution>
    </executions>
```

La configuració del plug-in consisteix en:

- La definició de la propietat del sistema 'javagen.package.name' que serà usada pel generador java-json-generator per a fixar el nom del paquet de la interfície generada.
- El nom complet de ja classe de configuració a usar. En aquest cas es troba disponible a les classes del servidor web.
- El *flag* per indicar que no volem realitzar un escaneig recursiu dels paquets a escanejar.

```
    <configuration>
      <systemProperties>
        <systemProperty>
          <name>javagen.package.name</name>
          <value>com.abiquo.jersey.demo.app.client</value>
        </systemProperty>
      </systemProperties>
      <configurationClass>com.abiquo.jersey.demo.app.client.JerseyClientGeneratorConfig</configurati
onClass>
      <scanRecursively>false</scanRecursively>
    </configuration>
  </plug-in>

</plug-ins></build></project>
```

Finalment per a realitzar la construcció del client cal executar la comanda 'mvn clean install' de manera que, el codi de la interfície es generi al subdirectori src/ i la llibreria JAR resultant de la compilació al subdirectori target/

Conclusions

Aquest projecte neix a partir de la identificació d'un coll d'ampolla en el desenvolupament de clients HTTP per als serveis web d'Abiquo. Per una part, la companyia ofereix un client Java oficial, però el seu manteniment entre versions implica bona part del temps de desenvolupament de versió. Per una altra banda, sorgeix la necessitat d'oferir clients HTTP en llenguatges diferents a Java i no es disposen de recursos suficients per cobrir aquestes necessitats.

Donat que tots els serveis web de la companyia estan desenvolupats amb Java seguint l'especificació JAX-RS, resulta molt interessant i factible la possibilitat d'automatitzar la creació de clients HTTP a mida.

Podem dir que el *framework* desenvolupat compleix les expectatives inicials i que fins i tot les ha superat. Actualment no només s'usa per a la generació de clients HTTP sinó que, a més, s'han desenvolupat generadors de documentació automàtica d'API.

Finalment, la realització d'aquest projecte m'ha permès aplicar els coneixements adquirits en els estudis de l'enginyeria en la identificació i solució d'un problema en el procés de desenvolupament d'un producte real. A més, des d'un punt de vista tècnic, m'ha permès endinsar-me en les noves característiques de Java 8 i ampliar els coneixements sobre les característiques de reflexió del llenguatge de programació Java.

Bibliografia

Representational state transfer [en línia]
https://en.wikipedia.org/wiki/Representational_state_transfer

Java API for RESTful Services (JAX-RS) [en línia] https://jax-rs-spec.java.net/

JSR 311: JAX-RS: The Java API for RESTful Web Services [en línia]
https://jcp.org/en/jsr/detail?id=311

Jersey [en línia] https://jersey.java.net/

Apache Wink [en línia] https://wink.apache.org/

Apache Maven Project [en línia] https://maven.apache.org/

Pipelining (computing) [en línia] http://en.wikipedia.org/wiki/Pipeline_%28computing%29

Maven – Introduction to the Build Lifecycle [en línia]
https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Maven – Guide to Developing Java plug-ins [en línia]

https://maven.apache.org/guides/plug-in/guide-java-plug-in-development.html

Annex A: Manual de construcció

Requeriments

Els requeriments per a construir el *framework* són:

- Ubuntu 14.04
- Oracle Java JDK 8
- Apache Maven 3
- Connexió a Internet

La totalitat del desenvolupament ha estat realitzat en Ubuntu 14.04 i és l'únic sistema operatiu en el que s'ha realitzat la compilació i s'han executat les probes, és per aquest motiu que Ubuntu apareix com a requeriment.

El *framework* fa un ús intensiu de les noves característiques de Java 8, com per exemple: expresions *Lambda*, *streaming* de col·leccions i mètodes *default* a interficies.

S'empra Apache Maven per a gestionar les dependències i la construcció del *framework.* Cal, d'aquesta manera, disposar d'una connexió a Internet per a realitzar la descarrega de les dependències necessàries.

A continuació s'indiquen diversos enllaços web per a la instal·lació de Java 8 i Apache Maven en el sistema operatiu Ubuntu 14.04:

http://www.webupd8.org/2012/09/install-oracle-java-8-in-ubuntu-via-ppa.html

http://askubuntu.com/questions/420281/how-to-update-maven-3-0-4-3-1-1

http://linuxg.net/how-to-install-apache-maven-3-2-1-on-ubuntu-14-04-linux-mint-17-and-their-derivative-systems/

Descripció dels projectes

El codi font presentat s'estructura al voltant de 8 projectes Java ben diferenciats:

1. **jaxrs-client-generator:** Aquest projecte conté el codi font del *framework* i la resta de projectes depenen d'ell.
2. **Jaxrs-client-generator-maven-plug-in:** Un *plug-in* de Maven que permet fer ús del *framework* des de fitxers *pom* i integrar d'aquesta manera la generació de clients RESTFul en el cicle de construcció d'un projecte Java.
3. **java-json-generator:** Un exemple de generador de codi Java el qual només accepta JSON i no disposa de suport per *paths* que continguin expressions regulars. En concret genera una interfície Java amb els mètodes JaxRs detectats i per una altra banda s'usa un *proxy* per a interceptar les crides a aquesta interfície i convertir-les en crides HTTP.
4. **jersey-demo-app:** Un servei RESTFul implementat amb Jersey.
5. **jersey-demo-app-client:** Un exemple d'integració del *plug-in* desenvolupat, amb l'objectiu de generar un client Java per al servidor del projecte *jersey-demo-app*.

6. wink-demo-app: El mateix servei RESTFul del projecte *jersey-demo-app* però implementat amb Apache Wink.
7. wink-demo-app-client: Un altre exemple d'integració del *plug-in* desenvolupat, amb l'objectiu de generar un client Java per al servidor del projecte wink *-demo-app*.
8. client-use-demos: Aquest projecte inclou dos programes Java que exemplifiquen l'ús dels clients autogenerats *jersey-demo-app-client* i *wink-demo-app-client*.

Guia de navegació pel codi font

Es recomana construir tot el codi font i després importar tots els projectes a Eclipse per a facilitar la lectura del JavaDoc i la navegació per les diferents classes.

1. El punt d'entrada per analitzar el codi és el projecte *jaxrs-client-generator,* en concret es recomana començar per la classe *JaxrsClientGenerator.java* i el métode *build()*
2. En el projecte *java-json-generator* es troba un bon exemple d'una implementació de la classe *ResourceGenerato.java.* En concret val la pena analitzar en primer lloc la classe *JavaClientGenerator.java* i en segon lloc la classe *ApiClient.java*
3. En el projecte *jersey-demo-app* es pot veure l'aspecte que presenta un servei RESTFul implementat en JaxRs. En concret val la pena analitzar *EnterpriseService.java* i *EmployeeService.java*
4. En el projecte *jersey-demo-app-client* es pot veure una integració del *plug-in* desenvolupat i l'aspecte de la interficie Java generada pel *java-json-generator*. Cal analitzar el fitxer *pom.xml* i la classe *Api.java* respectivament
5. Per últim, en el projecte *client-use-demos*, es pot veure un exemple d'ús del client generat a la classe *JerseyMain.java*

Compilació

Per a realitzar la construcció dels projectes que composen el PFC, només cal descomprimir el fitxer *codi-font.zip* i executar des de la carpeta codi-fot i des d'un terminal:

```
$ mvn clean install
```

Aquesta comanda construirà tots els projectes i executarà els tests corresponents. Qualsevol dels subprojectes (separats per carpetes) es pot construir de manera independent amb la mateixa comanda, només cal tindre en compte l'ordre de compilació:

1. jaxrs-client-generator
2. jaxrs-client-generator-maven-plug-in
3. java-json-generator
4. jersey-demo-app
5. jersey-demo-app-client
6. wink-demo-app
7. wink-demo-app-client

8. client-use-demos

Aplicacions de proba

Els següents projectes són aplicacions i integracions de proba del *framework* desenvolupat:

- jersey-demo-app i jersey-demo-app-client
- wink-demo-app i wink-demo-app-client
- client-use-demos

En aquest apartat ens centrarem només amb les aplicacions de proba relacionades amb *Jersey* ja que el procés descrit es equivalen per a les relacionades amb *Apache Wink*.

Configuració del servidor RESTFul

Per a poder executar probes cal aixecar el servidor corresponent. Per aquest propòsit s'usa el plug-in de Maven *jetty-plug-in* que permet aixecar un servidor web en qüestió de segons.

Des del directori *jersey-demo-app* cal executar la comanda:

```
$ mvn clean install jetty:run
```

Aquesta comanda aixecarà un servidor web accessible des del port 8080, per parar-lo només cal fer Ctrl + C i interrompre, així, la seva execució.

Ús del client auto generat

El projecte *client-use-demos* conté la classe *JerseyMain.java* que mostra com s'usa el client generat, cal tindre en compte que el codi considera que el servidor es troba aixecat a *localhost* i escoltant al port 8080.