Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Compoñente para importar servizos web REST a partir dunha especificación OpenAPI 3.0 nun sistema de integración de datos

Estudante: María Pombo García
Dirección: Manuel Álvarez Díaz
Dirección: Carlos Santos Canelles

A Coruña, febreiro de 2022.

Dedicado a todos os que me acompañaron ao longo destes anos, e aos que non poideron estar.

Agradecementos

Aos meus familiares por darme esta oportunidade e por todo o apoio e as ensinanzas recibidas.

Aos todos os compañeiros e compañeiras que me acompañaron durante estes anos de estudo.

Aos meus directores Manuel Álvarez e Carlos Santos por guiarme ao longo de todo o proxecto e pola paciencia ata o último día.

Resumo

A empresa Denodo comercializa un produto de integración de datos coñecido como Denodo Platform. O obxectivo deste produto é facilitar tarefas de integración de datos heteroxéneos e distribuídos en diferentes repositorios de información, como poden ser bases de datos relacionais, multidimensionais e servizos REST e SOAP, entre outras. Este proxecto centraráse en mellorar o soporte que ofrece a aplicación para os servizos web REST.

Na actualidade, existen especificacións amplamente apoiadas pola industria para expoñer os contratos implementados por servizos web REST. Unha delas é a *OpenAPI specification*. A partir da especificación OpenAPI dun servizo REST, é posible xerar automaticamente peticións para consultar o servizo, así como coñecer o esquema das respostas. Soportar este tipo de especificación permitiría simplificar a importación de novos servizos REST na Denodo Platform.

O obxectivo deste proxecto é desarrollar un compoñente no lado servidor da plataforma para acceso, obtención e procesado da especificación OpenAPI asociada a un servizo web concreto, así como unha extensión dos compoñentes de acceso a servizos web da plataforma para soportar OpenAPI. Para mellor a usabilidade deste compoñenete, desenvolverase un compoñente gráfico para a aplicación de escritorio, permitindo dar de alta un servizo REST seleccionando unha das operacións definidas na especificación OpenAPI indicada.

Seguindo a liña de ampliar as funcionalidades e o soporte da *Denodo Platform*, tamén se desenvolverá un compoñente que permita integrar consultas contra un motor GraphQL na plataforma Denodo.

Abstract

The company Denodo commercializes a data integration product known as Denodo Platform. The objective of this product is to facilitate the integration of heterogeneous and distributed data in different information repositories, such as relational databases, multidimensional databases and REST and SOAP services, among others. This project will focus on improving the application's support for REST web services.

Currently, there are widely supported industry specifications for exposing contracts implemented by REST web services. One of them is the OpenAPI specification. From the OpenAPI specification of a REST service, it is possible to automatically generate requests to query the service, as well as to know the response scheme. Supporting this type of specification would simplify the import of new REST services into the Denodo Platform.

The objective of this project is to develop a server-side component of the Platform for accessing, obtaining and processing the OpenAPI specification associated with a specific web service, as well as an extension of the application's web service access components to support OpenAPI. In order to improve the usability of this component, a graphical component will be developed for the desktop interface, allowing to register a REST service by selecting one of the operations defined in the OpenAPI specification indicated in a visual and simple way.

In line with extending the functionalities and support of the Platform, a component that allows the integration of queries against a GraphQL engine in the Denodo platform will be included.

Palabras chave:

- Integración de datos
- API RESTful
- Endpoint
- Especificación OpenAPI
- Swagger
- GraphQL

Keywords:

- Data integration
- API RESTful
- Endpoint
- OpenAPI Specification
- Swagger
- GraphQL

Índice Xeral

Índice de Figuras

Índice de Táboas

Capítulo 1

Introdución

A información converteuse nun dos activos máis valiosos nas últimas décadas, multiplicándose exponencial-mente a cantidade de datos que deben almacenar e manexar os sistemas informáticos. Xunto co aumento dos datos xurdiron novas necesidades, provocando que ao longo dos anos se foran creando novos tipos de repositorios de datos con diferentes estruturas e en diferentes localizacións: bases de datos relacionais, non relacionais, distribuídas, orientadas a obxectos, na nube, etc. Como consecuencia, o acceso á información volveuse cada vez máis complexo dado que cada fonte de datos ten os seus propios protocolos de acceso e un formato ou unha representación dos datos diferente.

Denodo ofrece un software chamado Plataforma Denodo co obxectivo de unificar e simplificar o acceso a un amplo catálogo de fontes de datos, diferentes entre elas: sistemas multidimensionais, bases de datos relacionais, documentos XML ou JSON, ficheiros planos, servizos REST ou SOAP, etc. Con este proxecto preténdese ampliar o soporte que ofrece a Plataforma, concretamente para dous tipos de repositorios de datos: APIs REST e GraphQL. Nos seguintes puntos describirase o soporte que teñen actualmente estas fontes de datos dentro da aplicación, así como o alcance que terán as novas melloras.

1.1 Situación Actual

Comezaremos detallando as características que ofrece a aplicación da empresa, a Denodo Platform, para as APIs REST. Este tipo de APIs baséanse principalmente en executar peticións HTTP contra un determinados URLs, permitindo deste xeito consultar, crear, actualizar ou eliminar datos do repositorio. A Plataforma permite crear uns elementos denominados *data sources*, os cales simbolizan unha fonte da que se poden extraer datos, e que neste caso representarían unha petición do servizo REST. Para representar e reproducir correctamente o comportamento da petición, no momento de crear o *data source* é necesario que o usuario lle proporcione certa información á aplicación, como por exemplo o seu método HTTP, o URL

sobre o que se executa, os parámetros ou o corpo que deben enviarse, cabeceiras, etc.

Unha vez creado o *data source* con todos os valores necesarios da petición, pode crearse sobre el outro elemento denominado vista base. Esta vista encárgase de executar as consultas contra a API facendo uso da información do *data source* sobre o que se creou, e está formada por unha serie de atributos (como os dunha táboa dunha base de datos relacional) que representarán cada un dos campos devoltos na resposta da operación, así como o seu tipo de dato. Polo tanto, para crear unha vista base dunha petición HTTP é preciso coñecer tamén o esquema da súa resposta para introducilo na Denodo Platform.

Co *data source* e a vista base configurados correctamente, o usuario xa poderá utilizar a vista para executar a petición contra a API dende a Plataforma. Sen embargo, todo este proceso descrito anteriormente é necesario levalo a cabo para cada unha das peticións do servizo REST. Ademais, habitualmente os clientes necesitan integrar na Plataforma Denodo máis dunha API REST, e dentro de cada unha pode haber numerosos recursos aos que acceder, cada un co seu propio *endpoint*, métodos HTTP e diferentes esquemas de resposta. Isto implica que os desenvolvedores de modelos de datos das empresas clientes deben configurar manualmente o *data source* de todas estas operacións, introducindo os valores necesarios na aplicación e dando como resultado un proceso tedioso, repetitivo e pouco eficiente.

No caso dos servizos SOAP, a Denodo Platform ofrece a posibilidade de importalos facendo uso do seu documento WSDL. Neste arquivo recóllese unha descrición da interface pública do servizo, con toda a información necesaria para crear *data sources* e vistas base que representen as súas operacións. O usuario pode indicar a ubicación deste documento para ser procesado pola Plataforma, a cal mostrará ao cliente unha lista coas peticións do servizo para as que é posible crear unha vista base. Posteriormente, o usuario pode seleccionar unha destas peticións e a vista créase automaticamente. Contar con un soporte parecido para os servizos REST sería de moita utilidade xa que reduciría considerablemente o tempo e a complexidade de importar as súas peticións.

En segundo lugar, as fontes de información GraphQL basean o seu funcionamento na creación dun esquema onde se define a información de todos os datos que se poderán acceder, así como as peticións que se poderán executar. Ao executar estas peticións pode indicarse exactamente que campos queremos que devolva o servidor, omitindo todos os demáis.

Na actualidade a Plataforma Denodo non dispón formalmente de soporte para GraphQL, non pode accederse a este tipo de repositorios de datos do mesmo xeito que se accede ás bases de datos JDBC, ODBC, APIs REST, etc. Sen embargo, no caso de ser necesario o seu uso Denodo ofrece APIs e librerías a través das que os clientes poden implementar unha clase Java coa lóxica de negocio necesaria para poder usar fontes de datos non soportadas pola aplicación.

1.2 Alcance e Obxectivos

Como se mencionou na introdución deste capítulo, a finalidade deste proxecto é ampliar o soporte e as funcionalidades que ofrece a Plataforma Denodo para a creación de modelos de datos contra servizos REST e engadir o acceso a fontes de datos GraphQL. Ambas melloras reducirán a curva de aprendizaxe e mellorarán a efectividade dos desenvolvedores de modelos de datos que utilicen este tipo de fontes. Este proceso levarase a cabo mediante o desenvolvemento de dous compoñentes na parte do servidor: un deles permitirá importar un servizo REST a través da súa especificación OpenAPI e o compoñente restante integrará as consultas contra fontes GraphQL. A continuación describiranse brevemente ambos elementos.

Para a problemática de importar operacións de servizos REST unha por unha, considérase crear unha nova funcionalidade co obxectivo de automatizar este proceso e facelo o máis simple posible. Pero para iso, a Plataforma necesita coñecer certa información da API para ser capaz de representar ás súas operacións de forma automática. De forma similar ao caso dos servizos SOAP explicados anteriormente, a nova funcionalidade fará uso dunha especificación utilizada para expoñer os contratos implementados por servizos web REST: a OpenAPI specification [1]. A partir da especificación OpenAPI dun servizo REST, é posible saber todas as peticións que se poden executar contra o servizo, coñecer os seus URLs, corpos das peticións, esquema e formato da resposta, etc.

Polo tanto, o usuario poderá indicar un documento OpenAPI a través dun URL ou de arquivos JSON ou YAML para ser procesado. Unha vez obtidas as súas peticións móstranse ao usuario para que poida seleccionar as operacións que quere representar e a Plataforma encargaríase de crear automaticamente a súa vista base utilizando a información da especificación OpenAPI, sen que o usuario teña que introducila manualmente. Sen embargo, ofrecerase tamén a posibilidade ao cliente de configurar certos parámetros da vista: poderase escoller entre unha vista base JSON ou XML, copiar a configuración doutros *data sources*, onde almacenar a vista, engadir un prefixo ao seu nome, etc.

Ademais, desenvolverase tamén unha interface gráfica na parte cliente da Plataforma para acceder a esta nova funcionalidade OpenAPI de maneira sinxela e visual. Dende esta interface poderá introducirse a especificación OpenAPI desexada e seleccionar as peticións das que se quere crear vistas base, así como escoller os parámetros de configuración mencionados anteriormente.

En segundo lugar, farase uso das APIs de Denodo mencionadas no apartado anterior para implementar un novo compoñente que ofreza a posibilidade de executar consultas sobre GraphQL, aportando desta maneira certo soporte para este tipo de fonte de datos á Denodo Platform. Este compoñente será capaz de converter as consultas relacionais que lanza o usuario dende as vistas base á sintaxe propia de GraphQL, executar a consulta, obter a resposta e

adaptala ao formato que debe devolverse na vista base. No que respecta á súa parte gráfica, Denodo xa conta con unha interface para importar e utilizar este tipo de compoñentes.

Capítulo 2

Base Tecnolóxica

Neste capítulo describiranse brevemente as linguaxes, tecnoloxías e ferramentas utilizadas durante o desenvolvemento deste proxecto. Tendo en conta que se está a desenvolver unha mellora para a aplicación da empresa, a *Denodo Platform 8.0*, o conxunto de tecnoloxías empregadas están condicionadas ás políticas aplicadas na organización.

2.1 Linguaxes

A continuación detallaranse as linguaxes de programación utilizadas para a construción do compoñente.

2.1.1 Java

Linguaxe de programación orientada a obxectos e estándar global para desenvolver e distribuír aplicacións. Permite desenvolver, implementar e utilizar de forma eficaz aplicacións e servizos. [2]

2.1.2 JSON

JavaScript Object Notation (*JSON*) é un formato lixeiro de intercambio de datos independente das linguaxes de programación, sinxelo de ler, escribir e interpretar. Está constituído por dúas estruturas universais, soportadas por todas as linguaxes de programación: [3]

- Unha colección de pares nome/valor, implementada na maioría de linguaxes de programación como un rexistro, estrutura, obxecto, dicionario, etc.
- Unha lista ordenada de valores, que adoita ser implementada polas linguaxes como un vector ou unha lista.

2.1.3 XML

Extensible Markup Language (*XML*) é un subconxunto de SGML (estándar para definir linguaxes de marcado para documentos), estandarizado pola comunidade The World Wide Web Consortium (W3C). Trátase dunha linguaxe de *tags* deseñada para transportar e almacenar datos. Establece un conxunto de normas de uso dos *tags* que permiten expresar información de forma estruturada e facilmente *parseable*. [4]

2.1.4 YAML

YAML Ain't Markup Language é unha linguaxe de serialización de datos (proceso de converter unha estrutura de datos ou obxecto nun formato especial para ser transmitido a través dunha rede informática) deseñado para ser útil e amigable para as persoas que traballan con datos. O seu funcionamento baséase en caracteres *unicode* imprimibles, que poden proporcionar información estrutural ou conter os datos. [5]

2.1.5 GraphQL

Linguaxe de consultas e entorno de execución no lado do servidor que permite realizar consultas utilizando un sistema de tipos que define os datos aos que queremos acceder. O seu funcionamento basease na creación dun esquema, no que se definen todos os datos posibles que se poderán consultar, xunto co seu formato e o seu tipo. As peticións son aceptadas ou rexeitadas baseándose neste esquema.

Ao realizar unha consulta podemos indicar que campos específicos do esquema queremos recuperar, podendo controlar dende o cliente que datos obter do servidor. Por outra parte, a organización en tipos e campos dos datos permite acceder aos servizos de *GraphQL* executando peticións *POST* que levan no corpo a consulta a realizar contra un único *endpoint*, a diferencia de outras arquitecturas como *REST*. [6]

2.1.6 SQL

Structured Query Language ou *SQL*, é unha linguaxe de consultas estandarizada deseñada para acceder e manipular bases de datos. Está soportada pola maioría de xestores de bases de datos relacionais do mercado. A linguaxe *SQL* componse de comandos, cláusulas, operadores e funcións, que permiten crear consultas mediante as que podemos crear, recuperar, inserir, actualizar ou eliminar información dunha base de datos. [7]

A empresa desenvolveu unha extensión desta linguaxe, chamada ***Denodo Virtual Query Language* (Denodo VQL)**, que amplía *SQL* para adaptala ás necesidades dun entorno de integración de información distribuída, mantendo a súa sintaxe. [8]

2.1.7 OpenAPI

OpenAPI é unha linguaxe de definición de servizos REST coa que se pode crear unha **OpenAPI Specification**, a cal define unha interface estándar para as *API RESTful*. Esta interface permite coñecer as funcionalidades do servizo sen necesidade de inspeccionar o código fonte ou a documentación. Todas as operacións da *API* poden ser representadas indicando os seguintes elementos: o seu *endpoint*, tipo de método *HTTP*, parámetros de entrada e saída da operación, métodos de autenticación, licencias, etc. As especificacións pódense escribir en *YAML* ou en *JSON*. [9]

2.1.8 LaTeX

Sistema de composición de textos orientado á produción de documentación técnica e científica. Está dispoñible como software libre e é o principal estándar para o desenvolvemento de documentos científicos. *LaTeX* separa o contido da súa presentación, permitindo aos autores centrarse en crear un contido de calidade sen preocuparse do seu estilo e formato. Mediante unhas sinxelas instrucións podemos seleccionar unha parte do texto e definila, por exemplo, como un título, unha cita ou unha lista, e *LaTeX* encargarase de darlle o estilo adecuado. [10]

2.2 Frameworks e librerías

2.2.1 Swagger

- **Swagger Core:** implementación Java de *Swagger/OpenAPI* que permite crear, consumir e traballar con definicións *OpenAPI*. [11]
- **Swagger Parser:** librería para analizar e procesar definicións *OpenAPI* (en formato *JSON* ou *YAML*) en representación *swagger-core* como *Java POJO*. [12]

2.2.2 GraphQL-Java

Librería que ofrece obxectos Java para traballar con esquemas e outros elementos de GraphQL. [13]

2.2.3 Swing

Librería de *Java* que permite crear unha interface gráfica. Proporciona compoñentes "lixeiros" para crear as vistas que, dentro do posible, funcionan igual en todas as plataformas. [14]

2.2.4 Apache

- **Commons:** esta librería permite crear e manter compoñentes Java reutilizables. Xeralmente utilizouse nos casos onde os obxectos ofrecidos por Java non eran suficientes para o desenvolvemento óptimo do código, polo que se recurriu ás funcionalidades desta librería: novas estruturas de datos, métodos de manipulación de *Strings*, operacións IO, etc. [15]

- **Log4j:** librería utilizada pola empresa para rexistrar trazas de depuración e información sobre erros ou eventos que ocorren durante a execución da aplicación. [16]

2.2.5 Probas

- **TestNG:** *Framework* de probas deseñado para cubrir todas as categorías de *testing*: unitarias, funcionais, integración, etc. Inspirado en *JUnit* e *NUnit*, introduce novas funcionalidades que o melloran e facilitan o seu uso: soporte para probas baseadas en datos, configuración flexible das probas, soporte para parámetros e anotacións, entre outras. [17]

2.3 Protocolos

- **HTTP:** *Hypertext Transfer Protocol* é un protocolo a nivel de aplicación con estrutura cliente-servidor, onde ambas partes comunícanse intercambiando mensaxes individuais. A conexión é iniciada sempre polo cliente, que envía unha mensaxe denominada petición ao servidor, o cal devolve unha mensaxe de resposta cos datos solicitados.

  As peticións e as respostas *HTTP* teñen formatos distintos. As peticións están formadas polos seguintes campos: un método *HTTP* que define a operación a executar (*GET, POST, DELETE, PUT, HEAD* ou *OPTIONS*); o *path* do recurso solicitado, excluíndo o protocolo, o dominio e o porto da conexión; a versión do protocolo *HTTP*; cabeceiras *HTTP* (opcionais) e un corpo só en métodos como *POST*, cando se require enviar información ao servidor.

  Por outra parte, as respostas *HTTP* teñen unha estrutura distinta: a versión do protocolo *HTTP*, un código do estado resultante da operación e unha breve mensaxe que describe do código de estado, cabeceiras *HTTP* (opcionais) e o recurso solicitado, no caso de que sexa necesario. [18]

  O protocolo *HTTP* é a base de calquera intercambio de datos na Web. Sobre el fóronse construíndo diferentes estruturas e arquitecturas. Entre elas atópase ***REST (REpresentational State Transfer)***, un estilo arquitectónico proposto no ano 2000 por Roy

Fielding para a construción de aplicacións distribuídas. Aínda que a estrutura *REST* non esixe o uso de *HTTP*, xeralmente é o protocolo co que máis se utiliza. Os servizos que cumpren o conxunto de principios arquitectónicos definidos polo estilo *REST* denomínanse *RESTful*. Estes principios enuméranse a continuación: [19]

- Estrutura cliente-servidor.
- Sen estado: o servidor non garda información de estado.
- Interface uniforme: as operacións e códigos de resposta deben ser sempre os mesmos e funcionar igual en todos os servizos. Non se especifica cales deben ser, pero na práctica utilizase habitualmente *HTTP*.

A tecnoloxía sobre a que imos traballar, a **OpenAPI Specification**, define unha interface estándar para as *API RESTful* que cumpren con estes principios arquitectónicos. Esta interface permítenos coñecer as funcionalidades do servizo sen necesidade de inspeccionar o código fonte ou a documentación, xa que representa todos os recursos accesibles da API xunto con información dos métodos que se poden executar sobre estes. [9]

2.4 Ferramentas de desenvolvemento

- **Apache Maven:** ferramenta software utilizada para a compresión e xestión de proxectos. Basease no concepto dun Modelo de Obxectos do Proxecto (POM) e permite xestionar a construción, os informes e a documentación dun proxecto. Ofrece un sistema de xestión de dependencias, un conxunto de estándares e unha lóxica para a execución do código en diferentes fases do ciclo de vida de desenvolvemento de software. [20]
- **IntelliJ IDEA:** IDE IDE para desenvolver aplicacións en diferentes linguaxes. Inclúe numerosas funcionalidades que facilitan a escritura de código, así como un *debugger* que permite depurar o software para atopar e solucionar posibles erros. Ademais, ofrece a posibilidade de instalar *plugins* e accesorios para engadir novas funcionalidades. [21]

  Un dos *plugins* utilizados no proxecto foi o **SonarLint**, unha extensión para encontrar e corrixir erros, vulnerabilidades e *code smells* (certas características do código que poden indicar un problema máis profundo) durante o desenvolvemento do código. Ademais de resaltar os problemas, ofrece as posibles maneiras de solucionalos, axudando a conseguir un código eficiente e seguro. [22]
- **Postman:** aplicación que permite crear, *testear* e consumir API REST (aínda que tamén permite executar peticións sobre servizos SOAP), polo que foi utilizada para facer probas e comprobacións sobre os servizos REST e especificacións OpenAPI que se utilizaron para o desenvolvemento do proxecto. [23]

- **Jenkins:** ferramenta de automatización para crear e probar proxectos de software. É utilizada pola empresa para integrar os cambios que se realizan no desenvolvemento dos seus produtos, así como para obter unha nova compilación do proxecto, executar tests, etc. [24]
- **Git:** sistema de control de versións distribuído de código aberto, deseñado para grandes e pequenos proxectos. Facilítanos o control dos cambios que se van realizando sobre os arquivos do proxecto. [25]
- **Overleaf:** editor en liña de *LaTeX* no que se realizou a memoria do proxecto. Conta con un sistema de control de versións e modelos para utilizar de forma gratuíta.

2.5 Sistemas de Xestión de Datos

2.5.1 Plataforma Denodo

Software de virtualización de datos desenvolvido pola empresa. A súa finalidade é crear unha capa de entrega de datos unificada como unha estrutura lóxica de datos, ocultando a complexidade do acceso á información e creando unha capa semántica para expoñer os datos de forma amigable e sinxela para o negocio. Dende o centro de control da Plataforma podemos executar e acceder aos distintos compoñentes que ofrece a ferramenta, como por exemplo: [26]

- *Virtual DataPort Server*: servidor sobre o que corre a aplicación.
- *Data Catalog*: interface *web* dirixida aos analistas de datos, desenvolvedores de aplicacións e usuarios que facilita a navegación a través dos datos e metadatos para a súa exploración e análise. [27]
- *Virtual DataPort Administration Tool*: interface de escritorio dende a que se poden desenvolver e administrar proxectos de virtualización de datos. Permite conectarse a diversas fontes de datos, combinalas e transformalas para desenvolver vistas e servizos de datos. Estes servizos poderán ser publicados para o seu acceso en múltiples formatos. [28]
- *Web Desing Studio*: versión web da *Virtual DataPort Administration Tool*. A diferencia desta última interface, Desing Studio pode ser accedida utilizando un navegador e non require unha instalación separada. [29]

Capítulo 3

Estado do Arte

Unha *Application Programming Interface* ou *API* é un conxunto de definicións e protocolos que se utiliza no desenvolvemento de aplicacións, para permitir a comunicación entre servizos e aplicacións cliente. Existen no mercado distintas aplicacións que permiten consumir e probar *APIs REST*, cada unha con distintas capacidades e especificacións.

Unha das máis utilizadas é *SoapUI* [30], un software de código aberto que se utiliza no *testing* de *APIs* para validar servizos web de arquitectura *REST* ou *SOAP*. Conta cunha interface dende a que podemos crear proxectos de probas para o noso servizo web de forma sinxela. No caso dun servizo *REST*, poderíamos importar un ficheiro *WADL* (especificación que nos indica como compoñer unha petición e describe a interface que proporciona dito servizo) ou indicar o URL dunha petición. Se utilizamos un documento WADL *SoapUI* creará automaticamente un proxecto cunha estrutura xerárquica no seguinte orde: o servizo, os seus recursos, métodos para cada recurso e un esqueleto de petición para estes métodos. Cada unha destas peticións poden ser executadas modificando os seus parámetros e vendo a resposta devolta pola *API*.

En canto ao soporte de *OpenAPI*, *SoapUI* permite importar unha especificación *OpenAPI* 2.0 ou inferior, xa que aínda non está soportada a versión 3.0. Unha vez importada, analízase e créase unha árbore con todos os seus recursos, os métodos atopados para cada un e un exemplo de petición para probar cada método. Podemos facer cambios sobre este esquema, engadindo ou eliminando recursos, métodos e peticións, ou modificar *paths*, parámetros, etc. Ademais, xérase automaticamente un descritor WADL para o servizo importado, que se mostra na figura 3.1 xunto coa árbore de *endpoints*. Por outra parte, ademais de importar unha especificación tamén é capaz de crealas, xa que permite exportar un proxecto *REST* como *OpenAPI*, xerando automaticamente a súa correspondente especificación (de versión 2 ou inferior) en formato JSON ou YAML.

Outro software similar é *Postman* [23], utilizado normalmente para crear, *testear* e consumir *API REST* aínda que tamén permite executar peticións sobre servizos SOAP. Mediante unha sinxela e intuitiva interface podemos definir calquera tipo de petición *HTTP*, executala

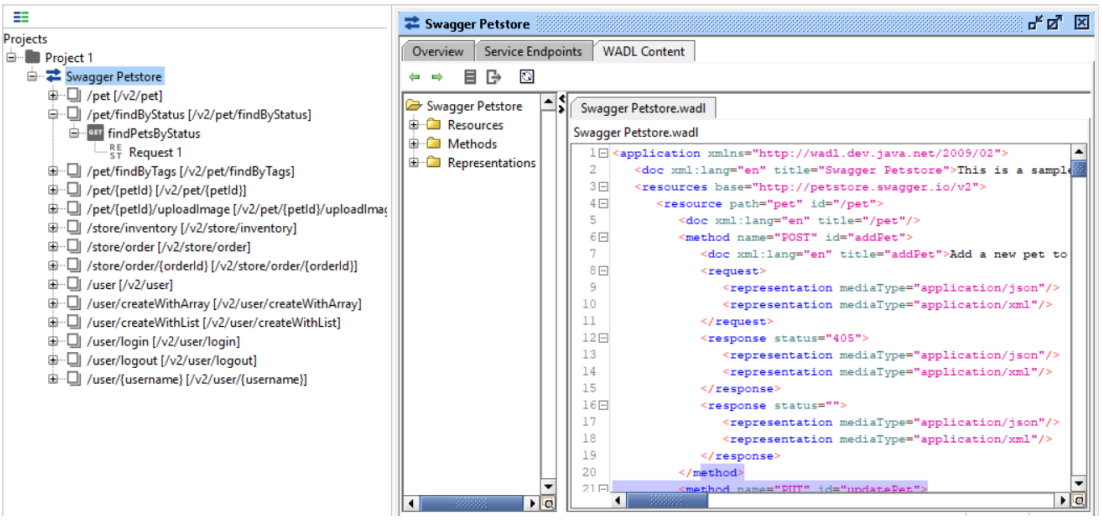

Figura 3.1: Especificación *OpenAPI* importada en *SoapUI*

contra a *API* e ver a súa resposta. *Postman* tamén conta con funcionalidades para o *testing* de *APIs*, tales como a automatización de fluxos de traballo, captura de peticións *HTTP* e a posibilidade de escribir *scripts* de proba, xunto con algúns exemplos destes *scripts*. Na figura 3.2 móstrase a interface principal de *Postman* cunha petición *GET* executada.

De maneira similar a *SoapUI*, *Postman* conta coa capacidade de importar unha OpenAPI Specification de versión 3 ou inferior para ser procesada e mostrar toda súa información de maneira amigable e funcional, tal como se mostra na figura 3.2, podendo ver todos os recursos dispoñibles e crear unha ventá de execución dunha petición con só seleccionar un dos métodos da especificación. Sen embargo, non permite xerar unha nova especificación *OpenAPI* a partir dun servizo *REST* existente. Para isto sería necesario utilizar unha ferramenta externa chamada *API Transformer*, capaz de converter unha colección de *Postman* en OpenAPI Specification. [31]

Existen outros métodos para acceder a servizos REST que non se basean en definir unha petición *HTTP* para os diferentes *endpoints* da *API*, executala e obter unha resposta en formato *HTTP*. Este é o caso de *GraphQL*, unha linguaxe de consultas que nos ofrece unha forma diferente de comunicarnos cunha *API*. Os recursos accesibles na API e os métodos que se poden executar represéntanse nun esquema de tipos, e os datos solicítanse mediante peticións *POST* a un único *endpoint*. Estas peticións levan no seu corpo a consulta a executar, na que se pode especificar exactamente que datos se queren visualizar, a diferencia das aplicacións anteriores nas que non se podía ”personalizar” a resposta, facendo máis eficientes as peticións ao evitar obter datos innecesarios. Na parte esquerda da imaxe 3.3 móstrase un exemplo básico dun esquema *GraphQL*: ”Book” e ”Author” son os recursos aos que se pode acceder, e ”book”,”bookById” e ”booksByAuthor” as operacións que se poden executar. Na parte dereita

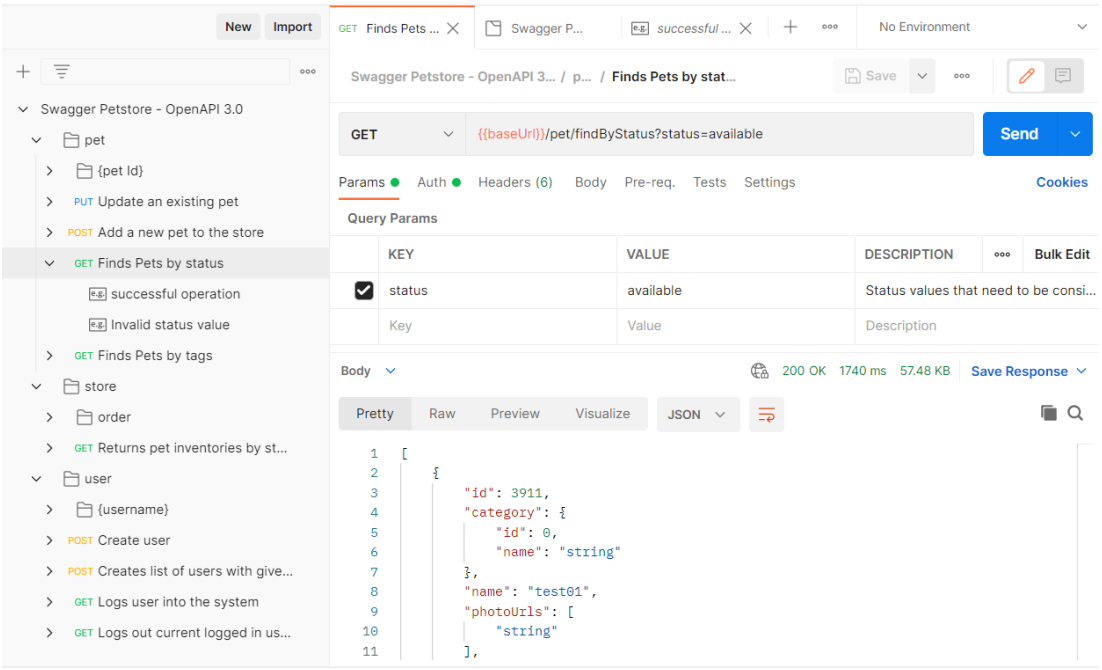

Figura 3.2: Especificación *OpenAPI* importada en *Postman*

da imaxe ilústrase a execución dunha consulta sobre este esquema utilizando Postman xunto co seu resultado.

Unha librería denominada *OpenAPI-to-GraphQL* ofrece a posibilidade de converter unha OpenAPI Specification nunha interface *GraphQL* que resolvería todas as consultas traducíndoas a peticións de *API REST*. Desta forma, tendo a especificación *OpenAPI* non sería necesario implementala en GraphQL de forma manual. [32] Para o proceso contrario tamén existe outra librería: *graphql-to-openapi*, que obtén unha especificación dun servizo a partir dun esquema *GraphQL*.

Se nos centramos no propio sector da empresa, podemos atopar produtos doutros fabricantes que ofrecen ferramentas de integración de datos que permiten tamén o acceso a *APIs REST*. A continuación detállanse brevemente algunhas destas aplicacións:

- Oracle desenvolveu o software de integración *Data Service Integrator*, que proporciona ás empresas unha ampla gama de servizos de datos deseñados para mellorar o acceso a distintas fontes de información. [33] Esta aplicación permite tamén a invocación de servizos *REST*, indicando o *URL* do *endpoint* que da acceso aos recursos. Con respecto ao soporte de *OpenAPI*, en ningún documento da aplicación se recollen funcionalidades relacionadas con esta tecnoloxía. [34]

- IBM comercializa *Cloud Pak for Data*, unha plataforma que ofrece un entramado de datos para conectar e acceder a datos en silos locais ou na nube, simplificando o seu

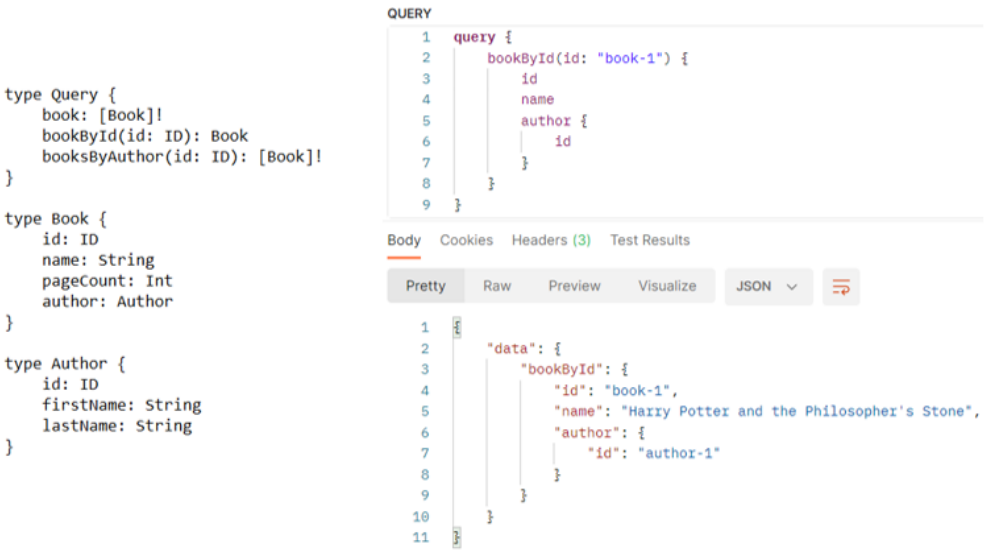

Figura 3.3: Exemplo de esquema e consulta GraphQL

acceso ao descubrilos e organizalos automaticamente para mostralos de forma amigable e útil aos usuarios. [35] En canto ao soporte de *APIs REST*, este software permite crear unha conexión coa nosa *API*, ademais de proporcionar unha colección de *APIs REST* ás que conectarse e que se poden utilizar para recompilar, organizar e analizar os nosos datos. De igual maneira que a ferramenta de Oracle, non se recollen funcionalidades relacionadas con *OpenAPI* na súa documentación. [36]

- TIBCO ofrece unha ferramenta chamada *TIBCO Data Virtualization* coa que podemos xestionar vistas virtualizadas e servizos de datos que acceden, transforman e entregan datos útiles para a toma de decisións do negocio. Da mesma maneira, que as anteriores ferramentas, tamén ofrece soporte para importar e acceder a *APIs REST*. [37] A diferenza das dúas aplicacións anteriores, *TIBCO Data Virtualization* conta cunha característica relacionada con *OpenAPI*: xera automaticamente unha especificación *OpenAPI* dun servizo *REST* cando este se publica dende dita aplicación. Sen embargo non se permite importar unha especificación para ser procesada e crear un novo servizo. [38]

No referente á propia ferramenta da empresa, a Plataforma Denodo permite invocar, despregar e exportar servizos REST, ademais ten soporte para xerar automaticamente a especificación OpenAPI deste servizo despois de publicalo, pero non conta con soporte para importar unha especificación e acceder ao seu servizo REST. Con este proxecto búscase engadir esa nova funcionalidade á Plataforma para que ofreza a posibilidade de importar unha especificación OpenAPI.

	Acceso a API REST	Importar OpenAPI Specification	Xerar OpenAPI Specification
SoapUI	SI	SI	SI
Postman	SI	SI	SI*
Oracle Data Service Integrator	SI	NON	NON
IBM Cloud Pak for Data	SI	NON	NON
TIBCO Data Virtualization	SI	NON	SI
Denodo Platform	SI	NON	SI

*Necesario o uso dunha tecnoloxía externa á aplicación

Táboa 3.1: Táboa comparativa das tecnoloxías mencionadas

A continuación móstrase unha táboa resumo comparando as funcionalidades de todas as aplicacións e tecnoloxías citadas:

Como se mencionou no capítulo 1 Introdución, para importar un servizo REST na Plataforma Denodo é necesario ir introducindo e configurando manualmente operación por operación. Polo tanto, permitir na Plataforma o acceso a un servizo REST mediante unha especificación OpenAPI simplifica considerablemente a importación desta fonte de datos, xa que nun só documento recollese toda a información necesaria sobre o servizo a cal doutra maneira debería ser introducida manualmente. Coa información da especificación e o URL dunha petición xa sería posible crear unha vista base de forma automática. Ademais, un usuario sen demasiados coñecementos técnicos podería levar a cabo este proceso.

Capítulo 4

Análise de viabilidade

Antes de comezar co desenvolvemento dun proxecto é recomendable analizar a súa viabilidade en certos puntos clave, como poden ser o aspecto temporal, económico e tecnolóxico. Desta maneira, podemos averiguar posibles riscos ou restricións que poden xurdir ao longo do proxecto e planificar como solventar ou evitar estes problemas. A continuación, analizaranse brevemente estes tres aspectos do proxecto:

- No referente ao **aspecto temporal**, a duración do desenvolvemento completo dos compoñentes que se expoñen está adaptada para un traballo de fin de grao, polo que estas novas funcionalidades a implementar ideáronse para unha duración equivalente a 12 créditos. Nun principio no supón un risco relevante, sen embargo, este parámetro podería chegar a afectar ao proxecto no caso de que xurdisen complicacións que provocaran retrasos, xa que a máis tempo de desenvolvemento maior é o custo do proxecto.

- O **aspecto económico** toma algo máis de relevancia dado que o proxecto se está a realizar cunha empresa, polo que é necesario estimar o custo dos recursos humanos e materiais que deben destinarse ao desenvolvemento das melloras. No que respecta ao custo de recursos físicos, inclúese o ordenador portátil que aportou a empresa no que se levou a cabo a implementación do proxecto. Dado que todo o proceso se realizou de forma telemática, non se contemplan gastos en instalacións, e o custo de luz e Internet non correría a cargo da empresa. Por outra banda, nos gastos de recursos humanos debe incluírse a remuneración das prácticas e a asignación dun titor dentro da empresa para a formación sobre entorno no que se vai traballar e a dirección na implementación das novas funcionalidade.

- Por último, tras analizar o **aspecto tecnolóxico** concluíuse que non ten demasiada relevancia para o proxecto no que respecta a riscos. As tecnoloxías a utilizar ao longo do proxecto, ademais de ser *opensource*, son moi coñecidas e utilizadas dentro da industria

para o desenvolvemento de software: IntelliJ Idea, Apache, Swagger, Swing, etc. Polo tanto, os riscos derivados do uso destas tecnoloxías é mínimo. A única tecnoloxía privada do proxecto é a propia aplicación da empresa, a Plataforma Denodo, máis a organización aporta unha licencia de desenvolvemento para que poida ser instalada e realizar probas sobre ela.

Tras o análise destes puntos clave podemos concluír que os aspectos que poden afectar de forma máis ou menos relevante ao proxecto son o temporal e o aspecto económico, xa que este traballo de fin de grao estase a realizar conxuntamente cunha empresa. Sen embargo, ningún dos tres parámetros indica que o proxecto sexa inviable ou teña demasiados riscos, polo que debería ser finalizado exitosamente.

Capítulo 5

Introdución ao desenvolvemento realizado

5.1 Introdución

Neste proxecto proponse desenvolver un novo compoñente para o servidor da Plataforma Denodo que permita crear unha vista base a partir das operacións dunha especificación OpenAPI dun servizo web, así como unha interface gráfica que permita invocar este compoñente de forma sinxela e visual. De forma complementaria e para continuar mellorando o soporte de fontes de datos da Plataforma, desenvolverase outro compoñente que ofreza a posibilidade de integrar consultas contra un motor GraphQL.

Nas seguintes seccións deste capítulo explicarase máis detalladamente a estrutura do compoñente e que tecnoloxías se utilizan en cada capa, así como a metodoloxía seguida durante o proceso e as iteracións nas que se dividiu.

5.2 Tecnoloxías

Este apartado expón as tecnoloxías utilizadas en cada unha das etapas do proxecto, ademais dunha breve contextualización da aplicación *Denodo Platform* sobre que a se vai realizar a mellora.

En primeiro lugar, o IDE utilizado durante todo o proxecto foi IntelliJ IDEA, xunto cun *plugin* de revisión de código chamado SonarLint. Para a construción do proxecto utilizouse a ferramenta de xestión de proxectos Maven, deseñada para empaquetar, construír e xestionar proxectos Java. Utilizáronse estas tecnoloxías xa que son as empregadas actualmente pola empresa.

En segundo lugar, como se mencionou anteriormente, a *Plataforma Denodo* proporciona ás empresas unha solución global para o acceder en tempo real a vistas de fontes de datos que

utilizan tecnoloxías e modelos de datos moi diferentes. A aplicación consta dunha arquitectura cliente-servidor e divídese en tres módulos:

- **Virtual DataPort:** módulo ao que pertence o compoñente creado neste proxecto. Permite a integración en tempo real da información recollida nas fontes de datos.
- **ITPilot:** ofrece unha maneira sinxela de acceder e estruturar datos que residen na Web.
- **Scheduler:** ferramenta que permite programar tarefas nos módulos anteriores para extraer, filtrar ou exportar datos.

Ademais, a aplicación ofrece unha interface de escritorio chamada *Virtual DataPort Administration Tool* dende a que se pode acceder e facer uso dos módulos anteriormente mencionados de forma gráfica e sinxela. Nesta interface levarase a cabo o *Frontend* da funcionalidade que se está a desenvolver mediante a creación dun novo *wizard*, mentres que o *Backend* formará parte do módulo *Virtual DataPort* (VDP). A figura 5.1 amosa un diagrama que ilustra a súa arquitectura xeral. Resáltanse en amarelo os compoñentes que se crearon neste proxecto.

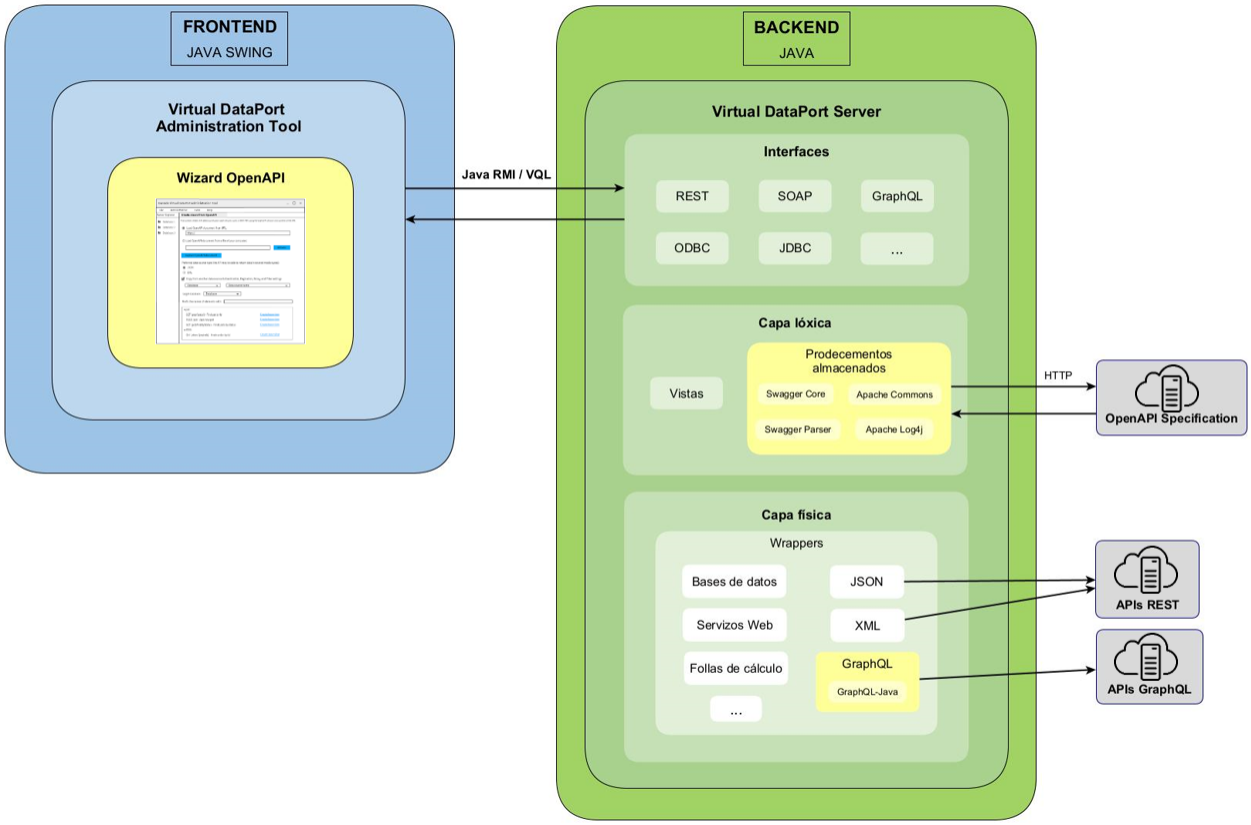

Figura 5.1: Arquitectura do proxecto

O módulo VDP actúa como un mediador que consegue abstraer todas as fontes de datos

para que parezan táboas dunha única base de datos ”virtual”, a cal permite crear vistas para combinar e integrar eses datos.*Virtual DataPort* é capaz de soportar fontes de datos estruturadas, semiestruturadas e non estruturadas, como poden ser bases de datos, servizos web SOAP ou REST, arquivos planos, follas de cálculo, servidores LDAP, etc.

Unha vez importados os datos con VDP, pódense combinar en vistas utilizado operacións de selección, proxección, unión, etc. Todas estas operacións execútanse nunha linguaxe propia da empresa chamada *Denodo VQL (Virtual Query Language)*, a cal extende SQL engadindo funcionalidades adaptadas ao entorno de traballo da aplicación: a integración de datos distribuídos. A *Administration Tool* permite crealas de maneira gráfica, sen necesidade de escribir manualmente os comandos.

Como se pode observar na imaxe 5.1, a arquitectura de *Virtual DataPort* divídese en tres capas: capa física, lóxica e interfaces. A continuación detállanse cada un destes niveles.

A **capa física** ou capa de acceso a datos, abstrae as capas superiores da comunicación coas fontes de datos e pretende homoxeneizar o acceso ás fontes. O seu funcionamento baséase nos ***wrappers***: cada tipo de fonte conta cun *wrapper* que recibe consultas en sintaxe VQL, convérteas á sintaxe da fonte para executar a consulta, extrae os datos, interprétaos e envíaos á seguinte capa no formato solicitado dende VDP. *Virtual DataPort* inclúe os *wrappers* das fontes de datos para as que da soporte a aplicación, pero ofrece unha API para crear os nosos propios *wrappers* en caso de que a fonte coa que traballamos non estea soportada. É o caso do *wrapper* GraphQL que tamén se desenvolveu neste proxecto, o cal se implementou seguindo as directrices desta API que proporciona a aplicación para posteriormente importalo na Plataforma e integrar as consultas contra GraphQL. Para a súa implementación utilizouse a librería *graphql-java*.

Por enriba desta capa atópase a **capa lóxica**, a cal contén **vistas base** que executan consultas con sintaxe VQL sobre as fontes de datos e reciben a información obtida polos *wrappers* da capa física. Cada vista ten unha serie de atributos (extraídos da fonte de datos) de xeito similar a unha táboa nunha base de datos relacional, e cada un deles ten un tipo de dato (*integer, string, array*, etc). O obxectivo principal desta capa é combinar os datos devoltos por diferentes fontes de datos en **vistas derivadas**, creando así novas relacións que aporten unha información máis ampla.

Esta capa pódese ampliar mediante procedementos almacenados (ou *stored procedures*), xa que Denodo ofrece unha API para desenvolvelos e poder importalos ao servidor, engadindo así novas funcionalidades á Plataforma. Un procedemento almacenado é un programa que contén unha lóxica de negocio que pode ser utilizada por varios clientes. Como consecuencia, nesta capa lóxica de *Virtual DataPort*, constituíndo o núcleo do *backend* da mellora, desenvolvéronse novos procedementos almacenados en Java que se encargan de implementar a lóxica de negocio da mellora proposta neste proxecto: crear unha vista base a partir dunha

especificación OpenAPI. Nestes procedementos utilizouse a librería Swagger, que nos permite traballar de forma cómoda con documentos OpenAPI.

Na Denodo Platform para crear unha vista base que recupere datos dun repositorio é necesario crear antes outros dous elementos: un *data source* e un *wrapper*. En primeiro lugar, o *data source* (ou fonte de datos) representará o repositorio sobre o que queremos executar consulta. Por exemplo, se a nosa fonte é unha base de datos MySQL, deberemos crear un novo *data source* de tipo JDBC indicando os parámetros que permitan á Plataforma Denodo conectarse á base de datos e facer consultas sobre ela: dirección da base de datos, credenciais, *driver*, etc. No caso de que o repositorio de datos sobre o que se vai traballar fora un servizo web coma neste proxecto, cada operación do servizo sería un *data source*, xa que cada unha delas pode devolver datos. O tipo de *data source* sería JSON ou XML e debe indicarse que petición HTTP executa a operación, o seu URL, parámetros, etc.

Os *data sources* poden crearse dende a interface Administration Tool ou mediante o comando VQL *CREATE DATASOURCE <tipo_datasource> <nome_datasource> <parametros>*. Para o caso do servizo web mencionado no exemplo anterior no que se quere acceder a un sitio web a través dunha petición HTTP para obter un recurso *book* que se corresponda co ISBN pasado por parámetro o comando de creación sería o seguinte:

```
1 CREATE DATASOURCE JSON "nome_datasource"
2     ROUTE HTTP 'http.CommonsHttpClientConnection,120000' GET
      'https://www.books.com/book/@{in_isbn}'
3     HEADERS (
4         'Accept' = 'application/json'
5     )
6     AUTHENTICATION OFF
7     PROXY DEFAULT;
```

Como se pode observar no comando VQL, a operación do servizo WEB que se representa no *data source* é unha petición HTTP GET en formato JSON sobre o URL *https://www.books.com/book/isbn*, que non require autenticación. Para executar esta petición é necesario proporcionar o identificador do obxecto *isbn* que se quere recuperar. Este parámetro representase na Denodo Platform co formato *@{in_nomeparametro}* e denomínase como **variable de interpolación**. Desta forma distínguese que é unha variable que debe recibir un valor concreto (non outro elemento da URL) e cando se executen vistas base sobre este *data source* solicitaráse ao usuario que introduza un valor para este parámetro.

En segundo lugar, é necesario crear un novo *wrapper* asociado ao *data source* anterior, que como se explicou na capa física, encargarase de recibir as consultas executadas dende a vista base que se creará posteriormente, convertelas en consultas do *data source*, obter os resultados devoltos e envialos de novo á vista. Para que o *wrapper* poida executar as consultas sobre os *data sources*, debe saber previamente o esquema da resposta que vai obter, é dicir, o

nome dos campos que vai recibir, o seu tipo de dato (*string, integer, array, register,* etc), se poden ser nulos e se se poden ordenar ou actualizar. No exemplo da operación dun servizo web, corresponderíase co corpo da resposta devolta ao executar a petición.

A Plataforma permite crear *wrappers* para as fonte de datos soportadas mediante o comando *CREATE WRAPPER <tipo_wrapper> <nome_wrapper> <parametros>*. Seguindo con exemplo da petición GET anterior, o seu *wrapper* sería o seguinte:

```
CREATE WRAPPER JSON "nome_wrapper"
    DATASOURCENAME="nome_datasource"
    OUTPUTSCHEMA (jsonfile = 'JSONFile' : REGISTER OF (
        isbn = 'isbn' : 'java.lang.String' (OBL) SORTABLE,
        title = 'title' : 'java.lang.String' (OPT) SORTABLE,
        author = 'author' : 'java.lang.String' (OPT) SORTABLE,
        price = 'price' : 'java.lang.Integer' (OPT) SORTABLE
    )
    );
```

Neste caso, o esquema da resposta da fonte de datos está formado polo ISBN indicado na URL (ao tratarse da variable de interpolación márcase como *EXTERN* e obrigatoria) e a información do libro atopado: identificador, nome do libro, título, autor e prezo. Estes tres últimos campos poden ordenarse (están marcados como *SORTABLE*) e o ISBN non pode ser nulo (leva a etiqueta *OBL*).

Por último, unha vez creado o *data source* que representa a fonte de datos e o *wrapper* que actuará de intermediario, poderemos crear a vista base que permitirá executar as consultas. Á hora de crear unha vista base deben definirse os seus atributos, que representan a que campos da fonte de datos pode acceder a vista. Ademais, tamén deben describirse os métodos de búsqueda que terá a vista, os cales definen que consultas se poderán executar. Isto é necesario xa que algunhas fontes de datos só permiten executar uns determinados tipos de consultas sobre os seus datos. Desta forma, nos métodos de búsqueda dunha vista deben especificarse para cada atributo os seguintes valores:

- Conxunto de **operadores** que se poden aplicar sobre o atributo nunha consulta. Se estea valor é "ANY" todos os operadores están permitidos.
- A **obrigatoriedade** do atributo, que pode tomar tres valores: *OBL* para un atributo que debe aparecer en todas as consultas, *OPT* se é opcional que apareza na consulta e *NOS* se non se permiten consultas con ese atributo.
- A súa **multiplicidade**, é dicir se o atributo pode tomar varios valores nunha mesma consulta. Pode ter un valor numérico coa cantidade de valores que pode tomar, *ANY* para un número ilimitado de valores e *ZERO* se non se poden executar consultas con ese atributo.

- Unha lista cos **posibles valores** que se poden usar para consultar o atributo. Por defecto utilizase unha lista baleira.

No método de búsqueda pódense indicar tamén que campos se poderán mostrar no resultado das consultas coa etiqueta *OUTPUTLIST*. De forma similar aos anteriores compoñentes, as vistas créanse co comando *CREATE TABLE <nome_wrapper> <parametros>*. A continuación ilústrase un exemplo:

```
CREATE TABLE "nome_vista" I18N es_euro (
        in_isbn : text (extern),
        isbn : text,
        title : text,
        author : text (DESCRIPTION = 'Author of the book'),
        price : int
    )
    CACHE OFF
    TIMETOLIVEINCACHE DEFAULT
    ADD SEARCHMETHOD "nome_vista" (
        I18N us_est
        CONSTRAINTS (
            ADD isbn (=) OBL ONE
            ADD title (any) OPT ANY
            ADD author (any) OPT ANY
            ADD price NOS ZERO ()
        )
        OUTPUTLIST (isbn, title, author, price)
        WRAPPER (JSON "nome_wrapper")
    );
```

Observando o comando anterior pódese ver que a vista consta de catro atributos: isbn, título, autor e price. Estes atributos son os campos do *data source* aos que pode acceder, rexíndose polas normas descritas no *search method* que se engade ao final do comando. Segundo o método de búsqueda, no resultado da consulta mostraranse os catro atributos xa que aparecen todos en *OUTPUTLIST*. Por outra parte, o campo isbn é obrigatorio para executar unha consulta e só pode utilizarse co operador ”igual que” tomando un único valor por consulta. Para os atributos ”title” e ”author” é opcional incluílos na consulta, ademais pódese usar calquera operador sobre eles e poden tomar varios valores nunha mesma consulta. Sen embargo, non se poden executar consultas directas contra ”price”, xa que ten obrigatoriedade *NOS*, multiplicidade *ZERO* e posibles valores unha lista baleira.

Por último, por enriba da capa física e lóxica está capa máis alta do servidor: a **capa de interfaces** ou capa de usuario, a cal contén as interfaces que expoñen *Virtual DataPort* ás aplicacións clientes e encárgase de manter a comunicación entre ambos. Non se fixeron cambios nesta capa durante o proxecto.

5.3 Metodoloxías e iteracións

A metodoloxía utilizada ao longo do proxecto foi o Proceso Unificado de Desenvolvemento Software (PUDS), un marco de desenvolvemento iterativo e incremental que se guía mediante casos de uso e que se centra na arquitectura do software. A continuación, lístanse as seis iteracións nas que se dividiu o proxecto:

- **Iteración 0:** formación sobre a plataforma Denodo e o seu entorno.
- **Iteración 1:** implementación do primeiro procedemento almacenado que procesará a especificación OpenAPI para mostrar todas as súas operacións.
- **Iteración 2:** desenvolvemento dun novo *wizard* na interface da *Administration Tool* que invoque o procedemento anterior e permita ver as operacións dunha OpenAPI Specification seleccionada.
- **Iteración 3:** implementación do segundo procedemento almacenado que cree una vista base a partir dunha operación HTTP da especificación OpenAPI.
- **Iteración 4:** modificación do *wizard* desenvolvido na iteración 2 para que permita acceder ao segundo procedemento almacenado.
- **Iteración 5:** implementación do *wrapper* para GraphQL.

No capítulo 6 Planificación e análise de costos descríbense máis en detalle todas estas iteracións.

Capítulo 6

Planificación e análise de costos

No capítulo previo indicouse que o proxecto foi desenvolvido mediante unha metodoloxía chamada Proceso Unificado de Desenvolvemento Software (PUDS). Seguindo as directrices desta metodoloxía, establecéronse 6 iteracións principais nas que desenvolver o proxecto. Ao comezo de cada iteración realízase unha breve fase de análise do traballo que se vai levar a cabo e posteriormente faráse a implementación do código e as probas de integración. Ao finalizar unha iteración organízase unha reunión co xefe de proxecto para a revisión das melloras implementadas.

Ademais, neste capítulo tamén se expoñerán os gastos que se estimaron para o completo desenvolvemento do proxecto, incluíndo os gastos de recursos humanos, materiais e outros tipos de gastos.

A continuación descríbense as seis iteracións mencionadas anteriormente, e nas imaxes 6.1 e 6.2 amósase o seu diagrama de Gantt:

- **Iteración 0:** determinación dos requisitos do cliente e descrición dos compoñentes que deben desenvolverse para solventalos. Nesta fase inclúese tamén o período de formación na aplicación da empresa, o entorno de traballo e as tecnoloxías que se van utilizar.

- **Iteración 1:** nesta fase comezase o desenvolvemento do primeiro procedemento almacenado, no lado do servidor. Ao final da iteración, o procedemento debe ser capaz de obter unha especificación OpenAPI dun URL ou dun String e procesala para devolver a listaxe das súas operacións.

- **Iteración 2:** creación da interface no lado cliente la Plataforma. Esta interface inicial debe permitir introducir un URL ou seleccionar un arquivo co documento OpenAPI. Unha vez construídos os elementos desta interface, debe implementarse a súa comunicación co servidor (mediante comandos VQL) para que sexa capaz de invocar o procedemento desenvolvido na Iteración 1. A través desta comunicación, a interface enviará o

documento OpenAPI introducido para ser procesado, obter de volta a lista de peticións e mostralas na interface en forma de árbore.

- **Iteración 3:** engloba o desenvolvemento do segundo procedemento almacenado no servidor, o cal se encarga de crear un *data source*, *wrapper* e vista a partir dunha operación concreta da especificación.

- **Iteración 4:** modificación da interface da Iteración 2 para incluír un menú que permita configurar as vistas que se desexan crear. Ademais, establecerase a comunicación co segundo procedemento almacenado. Deste xeito, a interface xa será capaz de crear vistas base despois de procesar unha especificación OpenAPI, polo que a primeira funcionalidade quedaría completada.

- **Iteración 5:** desenvolvemento do *wrapper* GraphQL no servidor da Plataforma. Será necesario comezar configurando os parámetros de entrada que terá o *wrapper*, e posteriormente implementar a lóxica de negocio mediante a cal este compoñente será capaz de executar consultas contra un servizo GraphQL. Tras executar unha consulta, o *wrapper* debe obter o seu contido e adaptalo á sintaxe da Denodo Platform, polo que tamén será necesario implementar código que realice este procesamento.

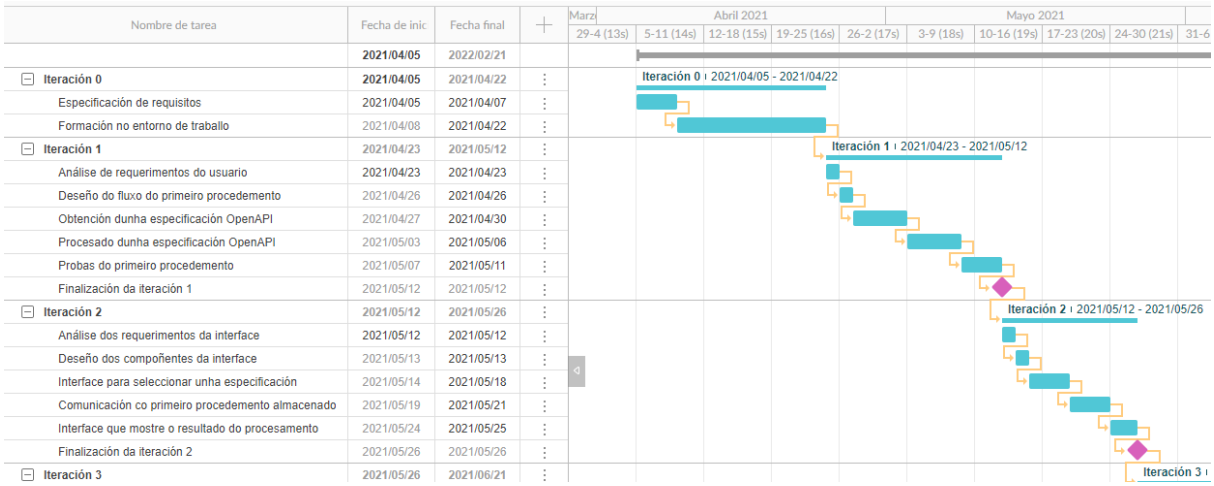

Figura 6.1: Diagrama de Gantt das iteracións 0, 1 e 2

Ao longo do desenvolvemento do proxecto mantivéronse estas iteracións diferenciadas, sen embargo, non se mantiveron os tempos estimados para algunhas tarefas. Na iteración 2, a construción da árbore de operacións tiña unha complexidade maior á estimada e obrigou a engadir dúas novas clases Java que a implementasen. Por outra banda, durante a iteración 3 xurdiu a necesidade de compaxinar o proxecto cos exames da universidade, polo que non se

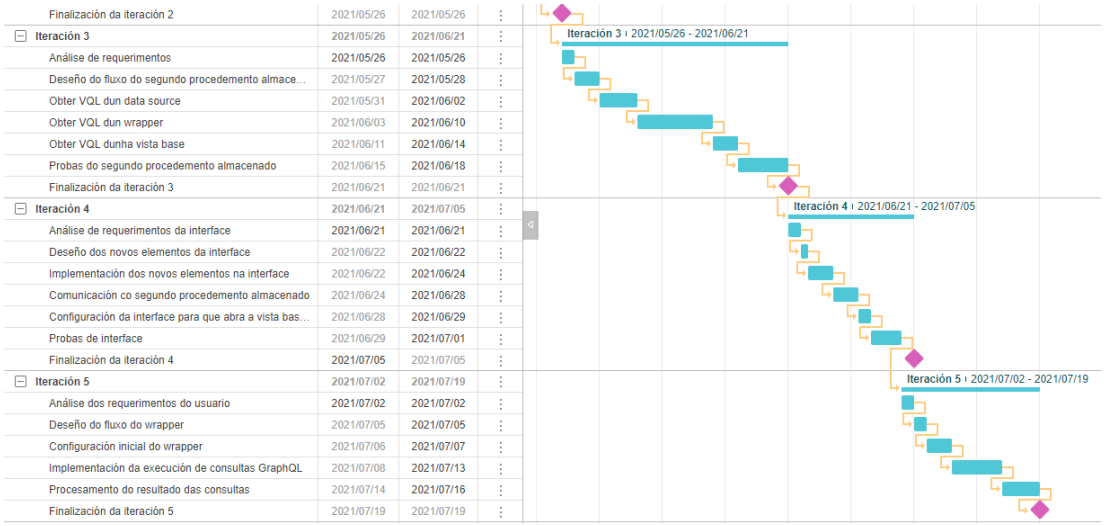

Figura 6.2: Diagrama de Gantt das iteracións 3, 4 e 5

avanzou coa mesma dedicación que anteriormente. Como consecuencia, o proxecto alargouse un total de 9 días con respecto á previsión inicial.

No referente aos custos que supuxo o proxecto, desglosaránse en tres tipos: recursos humanos, custos de material e custos indirectos. En primeiro lugar, o custo de recursos humanos estimouse utilizando as horas de traballo semanais previstas e a duración inicial do proxecto. Neste caso a xornada laboral do programador son 20 horas/semana, e o tempo estimado dedicado polo analista ao proxecto serían 6 horas/semana. Tendo en conta que a previsión incial de duración para o proxecto son 16 semanas, os cálculos serían da seguinte forma

- Custo programador: 20 horas/semana * 16 semanas * 10 €/hora = 3200 €
- Custo analista: 6 horas/semana * 16 semanas * 15 €/hora = 1440 €

En segundo lugar, nos gastos de material debe sumarse o valor que perdeu durante este período o ordenador portátil que aportou a empresa para desenvolver o proxecto. Este valor está estimado en torno a 65 euros.

Por último, ao montante total de gastos temos que engadir custos indirectos como a electricidade e o uso de internet. No que respecta á luz, estímanse uns 170 euros, e para internet uns 80 euros.

Polo tanto, a suma de todas estas cantidades deixa **un total de 4.475 €** que se necesitarían para levar a cabo este proxecto.

Capítulo 7

Requisitos do sistema

7.1 Introdución

Os requisitos deste proxecto foron identificados tendo en conta as necesidades que un usuario da Plataforma Denodo podería ter á hora de querer importar un servizo REST utilizando a súa especificación OpenAPI de forma sinxela e eficiente.

En primeiro lugar, a interface gráfica que se desenvolveu na *Administration Tool* dá soporte a dúas maneiras diferentes de almacenamento dunha especificación OpenAPI: nun arquivo de texto ou nunha localización na rede. Deste xeito, o usuario poderá utilizar a forma que mellor se axuste á súa situación e ás súas necesidades. Unha vez introducido un URL ou seleccionado un archivo en local, poderá procesarse a especificación para obter os seus recursos. Este procesado execútase invocando un procedemento almacenado que se desenvolveu na capa lóxica de *Virtual DataPort*. Este *stored procedure* recorre toda a especificación extraendo para cada un dos seus recursos unha serie de campos que permitan coñecer de maneira resumida cómo se realiza a invocación a cada un dos recursos contidos na especificación: método HTTP, ruta relativa da petición, descrición e o nome do recurso accedido. Por exemplo: *GET, /book/{isbn}, Find book by id.* Campos como o formato do corpo dunha petición ou cabeceiras só se mostrarán ao usuario despois de crear a vista.

Unha vez finalizado o procesamento da especificación, o procedemento almacenado devolve a información á interface, onde se engadirán dous novos elementos: unha árbore coa información das operacións atopadas e un panel de configuración. A árbore organizarase polo nome do recurso accedido (na imaxe 7.1 móstrase un exemplo), é dicir, un recurso ”book” será un nodo da árbore e despregará unha lista coas operacións que accedan a libros, un recurso ”movie” será outro nodo coas súas correspondentes peticións, e así sucesivamente.

Por outra banda, o panel de configuración permitirá seleccionar certos parámetros que terá a vista que queremos crear, como poden ser:

- O tipo da vista, que poderá ser JSON ou XML dependendo de que formato que prefira o

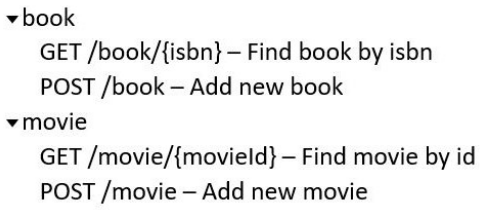

Figura 7.1: Listaxe das operacións contidas nunha especificación OpenAPI agrupadas por recurso

usuario. O seu *data source* e o seu *wrapper* tamén serán deste tipo. Para que esta opción funcione correctamente, a API debe ser capaz de devolver a información no *media type* seleccionado.

- A base de datos na que se vai crear a vista. Na Plataforma Denodo podemos ter múltiples bases de datos para organizar os nosos *data sources* e as súas vistas, polo que o usuario poderá escoller en cal delas almacenar a nova vista base creada coa interface.

- Un prefixo que se engadirá ao nome da vista.

- Copiar a configuración doutro *data source* e aplicalo ao *data source* da vista base que se vai crear. Desta maneira evítase configurar as vistas creadas unha por unha, e a nova vista adoptará os valores da configuración da fonte de datos seleccionada (autenticación, paxinación, configuración do proxy, etc), os cales poderanse modificar posteriormente se é necesario.

Na árbore, cada operación levará ao lado un botón *Create base view* e ao premer este botón, comezará o fluxo de creación dunha nova vista base para a operación seleccionada. Como se mencionou no capítulo de 5.1 Introdución, debe crearse un *data source*, un *wrapper* e por último a vista base. A creación destes tres elementos levarase a cabo por outro procedemento almacenado desenvolvido para este proxecto, o cal se encargará de obter para a operación escollida todos os parámetros necesarios para construír os comandos VQL que crearán estes tres elementos. Unha vez formados os comandos precisos serán executados e, se se crearon os compoñentes correctamente, abriranse na *Administration Tool* automaticamente a ventá de configuración do *data source* e da vista base.

Se o usuario non seleccionou a opción de copiar a configuración doutro *data source*, a interface situará o foco a ventá de configuración do *data source* que se acaba de crear para que o usuario poida modificar as súas propiedades. Sen embargo, se o usuario copiou a configuración dunha fonte de datos xa existente, a interface manterá o foco no *wizard* da árbore, xa que se presupón que non se van facer cambios na configuración.

En resumo, o *wizard* de creación de vistas a partir de OpenAPI funcionará en dúas fases, cada unha executada por un procedemento almacenado: o listado das operacións da especificación e a creación do *data source*, *wrapper* e vista base dunha operación. Cabe mencionar, que estes dous procedementos almacenados poden funcionar independentemente sen necesidade da interface, xa que poden invocarse dende a VQL Shell a través de comandos. Sen embargo, para facer este proceso máis sinxelo e visual para o usuario, decidiuse crear un novo *wizard* na interface de escritorio da *Administration Tool* dende o que se poidan acceder estes comandos.

En canto ao *custom wrapper* de GraphQL, este debe ser capaz de recibir unha consulta en VQL dunha vista base e convertela á sintaxe dunha consulta GraphQL para executala sobre o repositorio. Por exemplo, se a consulta que solicita o usuario dende a interface é *SELECT isbn, title, price FROM book WHERE isbn=123*, o resultado da conversión sería: *bookById(isbn: "123") { isbn title price }*. Despois de executar esta consulta, ten que obter e interpretar a súa resposta e devolver os datos solicitados no formato correcto.

Para este compoñente non será necesario desenvolver unha interface gráfica debido a que a Plataforma xa ofrece un *wizard* que simula un *data source* xenérico que utiliza un *custom wrapper* e crea vistas base sobre el para poder executar consultas. É dicir, en lugar de crear un *data source* tipo JSON que utilizará o *wrapper* JSON predefinido pola aplicación, créase un *data source* de tipo *Custom* no que se debe indicar manualmente a localización do *custom wrapper* que se quere utilizar.

7.2 Actores

A continuación indícanse os actores que interactúan con este novo compoñente:

- **Aplicación consumidora de servizos VDP:** este actor representa as aplicacións que poden executar peticións contra o *wrapper* de GraphQL ou as vistas JSON e XML que foron configuradas dende o *wizard* OpenAPI que fai uso da nova funcionalidade desenvolvida.

- **Desenvolvedor SQL:** actor con acceso ás funcionalidades de *Aplicación consumidora de servizos VDP* que tamén pode crear novos recursos. Representaría un usuario dunha empresa que utiliza a Plataforma Denodo para crear as fontes de datos e as vistas que representan os datos almacenados noutros sistemas.

Na figura 7.2 móstrase un diagrama coas relacións entre os actores mencionados:

Figura 7.2: Xerarquía de actores que utilizan o compoñente

7.3 Casos de Uso

No presente apartado enumeraranse os casos de uso do proxecto, indicando para cada un as precondicións nas que se deben executar, o fluxo de traballo normal, as post-condicións que deben ocorrer unha vez rematado e as posibles excepcións que pode devolver.

CU-01	**Acceder ao wizard OpenAPI**
Precondicións	O desenvolvedor SQL accede dende a interface Administration Tool.
Fluxo normal	O actor abre o wizard para crear unha vista base a partir dunha especificación OpenAPI na interface Administration Tool da Plataforma Denodo.
Post-condicións	Ningunha.
Excepcións	Ningunha.

Figura 7.3: Caso de uso 1

CU-02	**Introducir especificación OpenAPI mediante un arquivo local**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI*.
Fluxo normal	O actor selecciona a opción *Load OpenAPI document from a file of your computer*, preme o botón *Browse* e explora o seu dispositivo para seleccionar un arquivo do que obter unha especificación OpenAPI para poder procesala (CU-3).
Post-condicións	Ningunha.
Excepcións	Ningunha.

Figura 7.4: Caso de uso 2

CU-03	**Procesar especificación OpenAPI**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI* e indicou unha dirección URL (CU-01) ou arquivo local (CU-02) para obter a especificación.
Fluxo normal	O actor pulsa o botón de *Explore OpenAPI document* e a especificación introducida é procesada polo servidor para listar as súas operacións. Para cada unha móstrase: método HTTP, ruta relativa da petición, descrición e *tag* (nome do recurso accedido).
Post-condicións	Móstrase a lista de operacións en forma de árbore, xunto coas opcións para configurar a creación dunha vista.
Excepcións	URL inválida. Especificación OpenAPI inválida.

Figura 7.5: Caso de uso 3

CU-04	**Seleccionar o tipo da vista base a crear**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI* e xa procesou unha especificación (CU-03).
Fluxo normal	O actor selecciona un dos tipos que se ofrecen para crear a vista base: JSON ou XML.
Post-condicións	A vista base resultante, así como o seu *data source* e o seu *wrapper* será do tipo escollido se este é soportado pola.
Excepcións	Ningunha.

Figura 7.6: Caso de uso 4

CU-05	**Copiar configuración doutro *data source***
Precondicións	O desenvolvedor SQL accede dende o wizard OpenAPI e xa procesou unha especificación (CU-03).
Fluxo normal	O actor selecciona a opción *Copy from another data source Authentication, Pagination, Proxy and Filter settings* e escolle unha base de datos e un dos seus *data sources* para copiar a súa configuración
Post-condicións	O *data source* da vista base resultante será configurado de igual maneira que o *data source* escollido.
Excepcións	O usuario non ten os privilexios para acceder ao *data source* seleccionado.

Figura 7.7: Caso de uso 5

CU-06	**Seleccionar a base da datos na que crear a vista base**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI* e xa procesou unha especificación (CU-03).
Fluxo normal	O actor selecciona unha das bases de datos dispoñibles para crear nela a nova vista base.
Post-condicións	A vista base resultante pertencerá a base de datos seleccionada.
Excepcións	Ningunha.

Figura 7.8: Caso de uso 6

CU-07	**Indicar un prefixo para a vista base**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI* e xa procesou unha especificación (CU-03).
Fluxo normal	O actor introduce un prefixo para engadir ao nome da vista base e os seus compoñentes asociados (*data source* e *wrapper*).
Post-condicións	O nome da vista base resultante, o do seu *data source* e o do seu *wrapper* comezaran co prefixo introducido polo actor.
Excepcións	Ningunha.

Figura 7.9: Caso de uso 7

CU-08	**Crear vista base**
Precondicións	O desenvolvedor SQL atópase no wizard *Create views from OpenAPI* e xa procesou unha especificación (CU-03).
Fluxo normal	O actor pulsa o botón *Create base view* dunha operación mostrada na árbore tras o procesamento da especificación (CU-03).
Post-condicións	Créase a vista base, o seu *data source* e o seu *wrapper* a partir da operación seleccionada. Ábrense en cadansúa pestana a vista e o *data source*, este último mostrarase en caso de que non se copiara ningunha configuración móstrase, polo contrario seguirá a mostrarse o *wizard* OpenAPI.
Excepcións	Ningunha.

Figura 7.10: Caso de uso 8

CU-09	**Executar petición contra unha vista base creada co compoñente**
Precondicións	Ningunha.
Fluxo normal	O actor lanza unha petición para obter datos dunha vista base creada a partir dunha especificación OpenAPI, o seu *wrapper* é o que se encarga de solicitar o datos e recollelos da fonte de datos correspondente.
Post-condicións	Os datos devólvense ao actor que lanzou a petición.
Excepcións	Excepcións HTTP por errores na petición á API.

Figura 7.11: Caso de uso 9

CU-10	**Crear unha nova vista base utilizando o *wrapper* GraphQL**
Precondicións	Ningunha.
Fluxo normal	O actor crea unha nova vista base na Plataforma Denodo que utilizará o *wrapper* GraphQL implementado no proxecto.
Post-condicións	O acceso aos datos para esta vista base levarase a cabo polo mencionado *wrapper*.
Excepcións	Ningunha.

Figura 7.12: Caso de uso 10

CU-11	**Executar petición contra unha vista base que utiliza o *wrapper* GraphQL**
Precondicións	O actor executou previamente o anterior Caso de Uso 10.
Fluxo normal	O actor lanza unha petición para obter datos dun servizo creado sobre o *wrapper* GraphQL desenvolvido no proxecto. Este *wrapper* encargarase de acceder ao esquema GraphQL e obter os datos solicitados.
Post-condicións	Os datos devólvense ao actor que lanzou a petición.
Excepcións	Excepcións por erros na petición ao motor GraphQL.

Figura 7.13: Caso de uso 11

7.4 Modelo de casos de uso

Para comprender mellor ás relacións entre as funcionalidades listadas anteriormente deseñouse un modelo de casos de uso para cada un dos actores definidos na figura 7.2. En primeiro lugar, ilústrase o diagrama correspondente ao actor *Desenvolvedor SQL* na figura 7.14. Como se pode ver, debe introducir unha dirección URL ou seleccionar un arquivo do seu dispositivo para poder procesar unha especificación OpenAPI. Despois de introducir un documento e procesalo, poderá acceder ás funcionalidades para crear a vista base (escoller o seu tipo, copiar a configuración, etc). Independentemente deste proceso, poderá crear novos servizos que utilicen como *wrapper* para acceder a datos o que se implementou para GraphQL neste proxecto.

En segundo lugar, móstranse na figura 7.15 as funcionalidades do actor *Aplicación consumidora de servizos VDP*. Como se mencionou no apartado de Actores, só pode interactuar co compoñente en tempo de execución, cando se lanzan consultas contra unha vista base creada a partir dunha especificación OpenAPI ou sobre o *wrapper* GraphQL. Estas funcionalidades tamén poden ser levadas a cabo polo *Desenvolvedor SQL*, xunto coas mencionadas anteriormente.

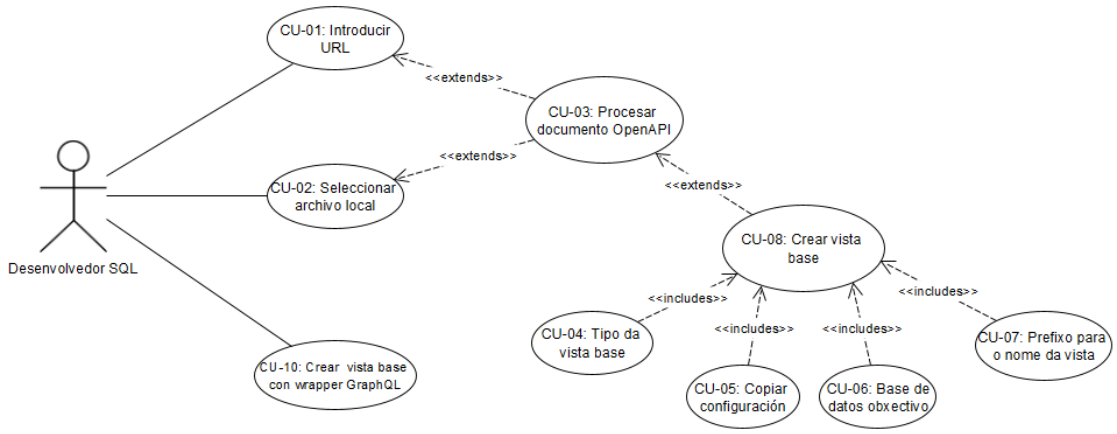

Figura 7.14: Casos de uso para o actor Desenvolvedor SQL

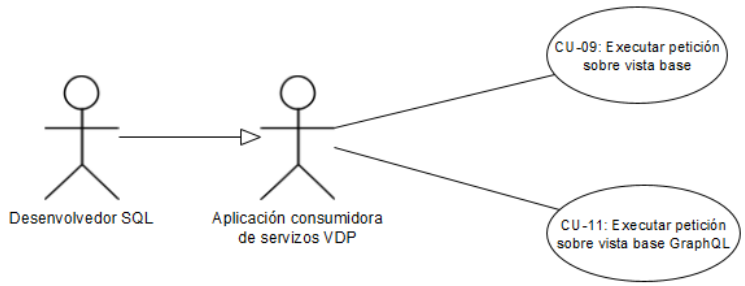

Figura 7.15: Casos de uso para o actor Aplicación consumidora de servizos VDP

7.5 Prototipado de interface

Na figura 7.16 mostrase un prototipado do *wizard Create views from OpenAPI* na súa pantalla inicial, onde se debe introducir a especificación para procesar posteriormente. Pódese ver que da soporte ás dúas maneiras mencionadas para introducir o documento OpenAPI.

Por outra parte, na figura 7.17 pódese ver que unha vez procesado o documento OpenAPI o *wizard* cambia para mostrar a árbore de operacións coa información de cada unha, así como todos os compoñentes necesarios para configurar a creación da vista base.

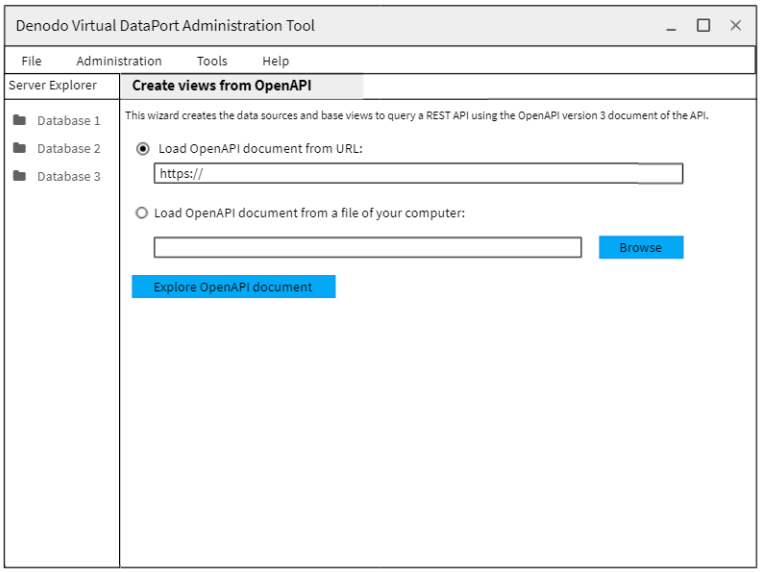

Figura 7.16: Prototipado da interface para introducir a especificación

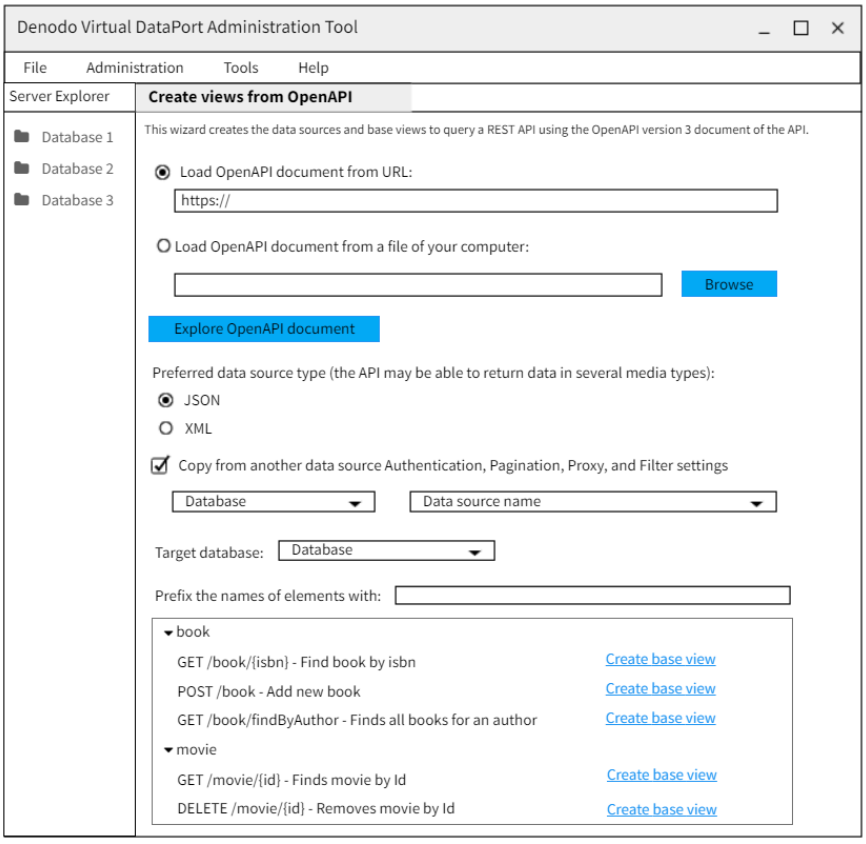

Figura 7.17: Prototipado da interface despois de procesar a especificación

Capítulo 8

Deseño

8.1 Introdución e obxectivos

Na fase de deseño do software levouse a cabo unha descrición das características que deben ter os novos compoñentes que se queren desenvolver para cumprir cos obxectivos do proxecto. Ademais, esta tarefa realizouse en base a un obxectivo principal: obter un produto modular que facilite futuras ampliacións e un mantemento sinxelo e eficiente. Desta maneira, intentou manterse a maior independencia posible entre os compoñentes que conforman o software.

8.2 Resumen de patróns usados

Os principais patróns de deseño que se utilizaron ao longo do proxecto foron os seguintes:

- **Deseño por capas:** utilizado pola empresa para desenvolver o seu software. Divide o sistema en diferentes capas independentes entre sí pero que se comunican entre elas, polo que un cambio nunha das capas non afectaría ás demais. Simplifica tamén a ampliación destas capas para engadir novas funcionalidades.

- **Singleton:** patrón no que unha clase só pode ter unha única instancia. Utilízase na empresa para as diferentes vistas da interface *AdminTool*, para que só poida existir unha instancia de cada unha delas.

- **Modelo-Vista-Controlador:** este patrón establece unha estrutura en tres capas na que cada unha leva a cabo un papel moi concreto. A capa Modelo encárgase de manexar os datos, a capa Vista encárgase de construír os compoñentes da interface que verá o usuario, e o Controlador enlaza as anteriores capas e implementa a lóxica de negocio da aplicación.

- **Factoría:** unha clase principal (a clase factoría) encárgase de delegar a creación de determinados obxectos a unha subclase, en lugar de invocar o constructor do propio obxecto.
- **Iterador:** patrón no que se define unha interface que implementa todos os métodos necesarios para percorrer os elementos dunha estrutura de datos sen coñecer a súa estrutura interna.
- **KISS (Keep It Simple, Stupid!):** defende que a simplicidade do sistema favorece un mellor funcionamento en comparación cun sistema máis complexo.

8.3 Arquitectura xeral

A nova funcionalidade desenvolvida neste proxecto consta dunha arquitectura divida en dous sistemas: cliente (*frontend*) e servidor (*backend*), como se pode ver no diagrama da figura 8.1 destacando en amarelo os elementos creados neste proxecto.

O subsistema servidor implementa a lóxica de negocio da creación dunha vista a partir duna especificación OpenAPI e da integración das consultas sobre un motor GraphQL. A parte de OpenAPI desenvolveuse na capa lóxica facendo uso de dous novos procedementos almacenados. Por outra banda, a integración das consultas GraphQL implementouse na capa física mediante a creación dun novo *custom wrapper*.

En canto ao subsistema cliente, implementouse mediante unha nova vista ou *wizard* na interface de escritorio da Plataforma Denodo. Dende esta vista poderase acceder de forma visual á funcionalidade OpenAPI mencionada anteriormente.

8.4 Subsistema backend

8.4.1 Obxectivos

Este subsistema encárgase de implementar a lóxica de negocio do noso compoñente. A comunicación coa base de datos da plataforma e coa interface de usuario non se implementaron neste proxecto xa que ditas tarefas deléganse na estrutura de clases xa desenvolvida pola empresa para o funcionamento da plataforma. O *backend* foi desenvolvido enteiramente en Java e nel faise uso la librería Swagger para o procesado de especificacións OpenAPI e das librerías de utilidade de Apache Commons.

8.4.2 Arquitectura

Como xa se mencionou anteriormente, o subsistema *backend* está constituído por dous procedementos almacenados e o *custom wrapper* GraphQL. Na figura 8.1 poden verse estes

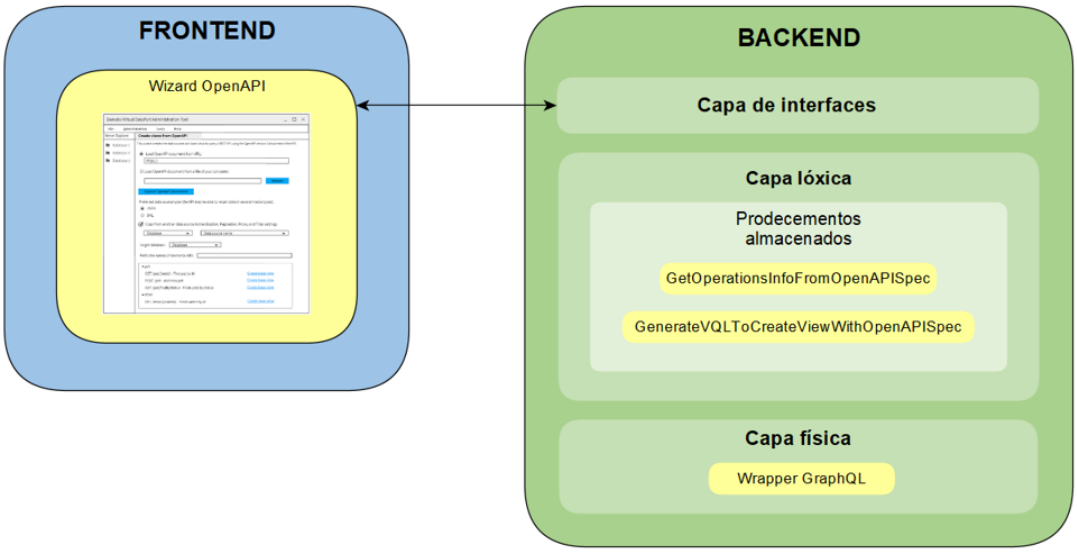

Figura 8.1: Arquitectura xeral do compoñente desenvolvido

tres compoñentes nas súas respectivas capas.

Os procedementos almacenados foron desenvolvidos na capa lóxica do módulo *Virtual DataPort* da Plataforma Denodo, seguindo as directrices da API que ofrece a empresa para os *stored procedures*. Unha vez creados poden ser invocados dende o servidor para darlle a funcionalidade de procesar unha especificación OpenAPI. Un destes procedementos encárgase de listar as operacións dunha especificación OpenAPI, e denomínase *GetOperationsFromOpenAPISpec*. O segundo leva a cabo a creación do *data source*, *wrapper* e vista base dunha operación, e chámase *GenerateVQLToCreateViewWithOpenAPISpec*. Nos seguintes apartados deste capítulo explicaranse en detalle ambos procedementos.

Cada procedemento almacenado ten uns parámetros de entrada, necesarios para executar a súa lóxica de nogocio, e uns de saída, onde se mostra o resultado final da execución. Todos os *stored procedures* do servidor de Denodo deben extender a clase *AbstractStoredProcedure* onde se definen unha serie de operacións que teñen que implementar, ilustradas na figura 8.2 xunto cos dous procedementos desenvolvidos para o proxecto.

A continuación descríbese cada unha destas funcións:

- String getName(): devolve o nome do comando co que se invocará o procedemento.
- String getDescription(): devolve unha descrición do procedemento.
- StoredProcedureParameter[] getParameters(): invocase cada vez que se chama o procedemento e devolve un *array* con cada parámetro de entrada e de saída. Un parámetro defínese como un obxecto *StoredProcedureParameter* e débense especificar os seguintes

valores: nome do parámetro, tipo Java (*VARCHAR*, *INTEGER*, etc), dirección (se é de entrada ou de saída) e se pode ser nulo ou non, que se indica con un *boolean*.

- void doCall(Object[] inputValues): este é o método principal e invocase polo motor de execución cando se chama ao procedemento. Sería o equivalente a un *main()* xa dende el execútase toda a lóxica do *procedure*: recibe un *array* de cos parámetros de entrada, realiza as operacións necesarias para cumprir a finalidade do procedemento e constrúe a saída que se vai devolver.

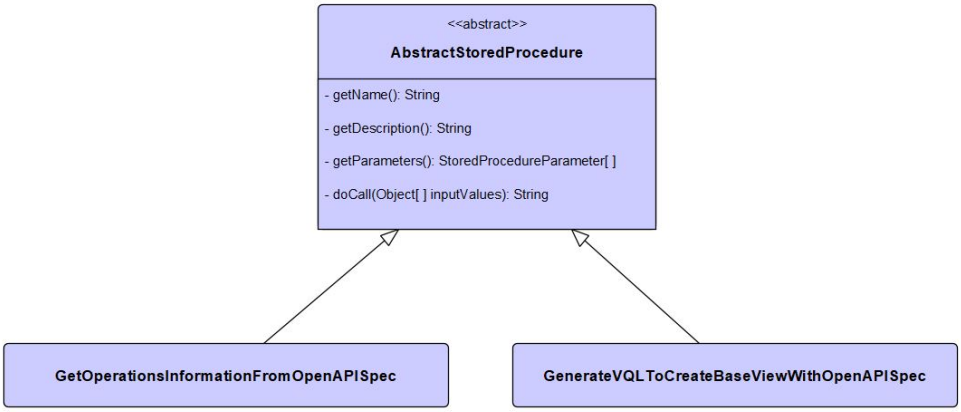

Figura 8.2: Diagrama dos procedementos almacenados

Ademais destas operacións que obriga a implementar a clase abstracta, en cada un dos procedementos implementáronse os métodos necesarios para levar a cabo a lóxica de negocio desexada. Xeralmente, a operación doCall mencionada anteriormente é a que dirixe o fluxo da clase, xa que vai invocando cada un destes métodos para ir formando o resultado final.

Por outra banda, o *custom wrapper* para integrar consultas GraphQL forma parte da capa física de VDP, e do mesmo xeito que para os procedementos Denodo tamén ofrece unha API para desenvolver *wrappers* capaces de recuperar información de fontes de datos non soportadas pola Plataforma. O novo *wrapper* debe extender da clase *AbstractCustomWrapper* e implementar os métodos establecidos nela:

- **CustomWrapperInputParameter[] getDataSourceInputParameters():** contén os parámetros que o usuario debe proporcionar para ser capaz de conectarse á fonte de datos do *wrapper*. Por exemplo, pode ser necesario proporcionar un usuario e unha contrasinal para autenticarse, ou como no caso de GraphQL o URL onde se atopa definido o esquema de datos do repositorio.
- **CustomWrapperInputParameter[] getInputParameters():** neste método defínense os parámetros de entrada que vai necesitar o *wrapper* para crear unha vista base sobre

a fonte, indicando para cada parámetro un nome, descrición, obrigatoriedade, tipo de dato e se depende do entorno. No caso do *wrapper* GraphQL terá un único parámetro de entrada que será a *query* a executar en cada vista.

- **CustomWrapperSchemaParameter[] getSchemaParameters(Map<String, String> inputValues):** aquí debe construírse o esquema de saída que van ter os datos despois de executar a consulta contra o motor. Neste caso o esquema construíuse adaptado ao formato de saída de GraphQL, que se explicará máis adiante.

- **void run(CustomWrapperConditionHolder condition, List<CustomWrapperFieldExpression> projectedFields, CustomWrapperResult result, Map<String, String> inputValues):** método principal do *wrapper*, similar ao *doCall()* dos procedementos. Execútase cada vez que se utiliza o *wrapper* e encárgase de lanzar a consulta escollida polo usuario contra o repositorio GraphQL, recibe a resposta e engade os campos ao esquema de saída para ser devoltos ao usuario.

En canto á estrutura de paquetes do *backend*, na figura 8.3 pódese observar que consta de dous paquetes: *storedprocedure* e *customwrapper*, ambos contidos no módulo do servidor de *Virtual DataPort* (denodo-vdp-server). Dentro do paquete *storedprocedure* creáronse dous novos procedementos almacenados que conteñen practicamente toda a lóxica de negocio da nova funcionalidade para OpenAPI, e no paquete *customwrapper* desenvolveuse o novo *wrapper* que integrará as consultas GraphQL na Plataforma Denodo. Os procedementos e o *wrapper* importáronse no servidor *Virtual DataPort* para integralos na aplicación. Unha vez engadidos ao servidor, este encárgase de comunicar o novos compoñentes coa base de datos e coa interface de usuario.

Na seguintes seccións explicarase en detalle a implementación do *wrapper* GraphQL e dos procedementos almacenados que definen o compoñente así como a lóxica de negocio que implementan.

8.4.3 GetOperationsInfoFromOpenAPISpec

Este procedemento encárgase de realizar a parte inicial do fluxo do novo compoñente desenvolvido: invócase para mostrar ao usuario un listado das operacións do documento OpenAPI sobre as que pode crear unha vista base. Os seus parámetros de entrada e de saída descríbense a continuación.

Parámetros de entrada

- *specification_url*: dirección URL onde se atopa a especificación OpenAPI. Pode ser nulo.

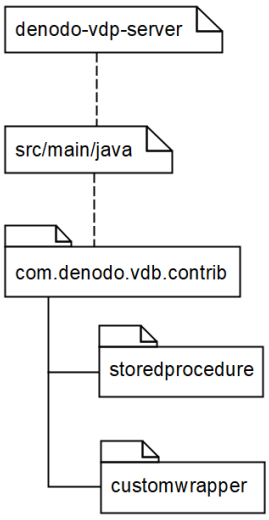

Figura 8.3: Diagrama de paquetes do subsitema *backend*

- *specification_as_string*: contido completo da especificación como unha cadea de caracteres. Pode ser nulo.

É obrigatorio que un dos parámetros teña valor e o outro sexa nulo, no caso de introducir un valor para os dous ou non introducir ningún mostrarase un erro. O comando para invocar este procedemento tería a seguinte sintaxe: *GET_OPERATIONS_FROM_OPEN_API_SPEC(<url>, <specification_string>)*. Por exemplo: *GET_OPERATIONS_FROM_OPEN_API_SPEC('https://www.books.com/openapi.json', null)*.

Parámetros de saída

O procedemento devolverá unha tupla de catro campos por cada operación atopada, indicando os seguintes valores:

- *http_method*: método HTTP da petición (exemplo: GET).
- *path*: ruta relativa da petición, sen parámetros (exemplo: */pet/findByStatus*).
- *summary*: breve descrición da operación (exemplo: *Find pet by status*).
- *tag*: nome do recurso ao que accede á petición (exemplo: *pet)*.

Na seguinte imaxe 8.4 amósase un exemplo das tuplas que devolvería este procedemento:

http_method	path	summary	tag
GET	/book/{isbn}	Finds a book by isbn	book
POST	/book	Creates a new book	book
GET	/movie/{id}	Finds a movie by id	movie
DELETE	/movie/{id}	Removes a movie by id	movie

Figura 8.4: Exemplo da saída do procedemento GetOperationsFromOpenAPISpec

Fluxo de traballo do procedemento

Na figura 8.5 amósase o diagrama de fluxo correspondente a este procedemento. Como se indicou anteriormente, é posible invocalo indicando unha URL ou o *string* da especificación. No caso de utilizar unha URL, a clase deberá conectarse á dirección para descargar a especificación OpenAPI que contén e gardala como unha cadea de caracteres. Unha vez teñamos a especificación como un *string* debemos convertela a un obxecto OpenAPI da librería Swagger para poder acceder á súa lista de operacións e comezar a procesala.

A lista de operacións dun obxecto OpenAPI pode obterse co método *getPaths()*, que nos devolverá un obxecto Paths no que se recollen as peticións do servizo web ordenadas por recursos. Por exemplo, para o recurso "book" haberá asociada unha lista con todas operacións do servizo que conteñan na URL /book. Polo tanto, a función principal deste *stored procedure* é recorrer todos os recursos de Paths accedendo ás súas operacións e gardando para cada unha os parámetros de saída listados anteriormente: método HTTP, ruta relativa, descrición e nome do recurso.

Cando non queden máis recursos que procesar, o procedemento devolverá unha tupla por cada operación atopada coa súa correspondente información, como na anterior figura 8.4.

Operacións do procedemento

A continuación lístanse as operacións que implementa este procedemento almacenado e unha breve descrición de cada unha:

- **String getOpenApiSpecification(String specificationUrl, String specificationAsString, DenodoSession session):** devolve a especificación en formato string. Realiza o proceso de conectarse á unha dirección URL e obter o string da especificación en caso de que sexa necesario.
- **OpenAPI parseSpecification(String openAPIScpecification):** converte un string a un obxecto OpenAPI.
- **Object[] getOperationInfo(String key, PathItem pathItem, HttpMethod httpMethod):** obtén a información correspondente aos parámetros de saída do procede-

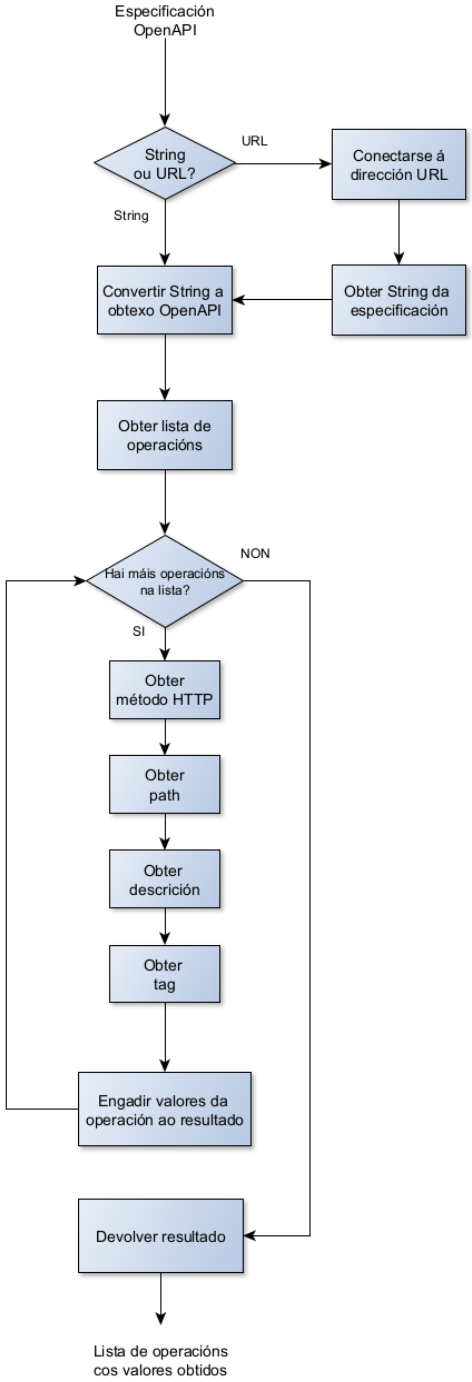

Figura 8.5: Diagrama de fluxo do procedemento GetOperationsFromOpenAPISpec

mento a partir unha operación OpenAPI que se lle pasa por parámetro (pathItem), e devolvea como un *array* de obxectos ao método doCall do procedemento para engadir

os campos ao resultado final.

8.4.4 GenerateVQLToCreateViewWithOpenAPISpec

Este procedemento recibe á entrada unha operación concreta dunha especificación OpenAPI e leva a cabo todos os pasos necesarios para xerar as instruccións VQL que creen o seu *data source*, *wrapper* e a vista base de dita operación. Ademais, permite incluír certos parámetros para a configuración da vista. Nos seguintes puntos describiranse os seus parámetros de entrada e a súa saída.

Parámetros de entrada

- *specification_url*: URL da localización da OpenAPI Specification. Pode ser nulo.
- *specification_as_string*: OpenAPI Specification completa en texto, en formato JSON ou YAML. De igual maneira que no primeiro procedemento, só se poderá indicar a especificación con un destes dous parámetros. Pode ser nulo.
- *name_prefix*: prefixo que levarán os nomes do *data source*, o *wrapper* e a vista base. Parámetro opcional.
- *path*: ruta relativa da operación, sen protocolo e sen *hostname*. Non pode ser nulo.
- *http_method*: Método *HTTP* da operación. Non pode ser nulo. Debe ser algún dos seguintes valores: *GET, POST, PUT, DELETE, HEAD, OPTIONS*.
- *preferred_media_type*: Tipo da vista base que se vai crear: *JSON* ou *XML*. Non pode ser nulo.
- *copy_settings_from_data_source_database*: nome da base datos á que pertence o *data source* a copiar. Pode ser nulo.
- *copy_settings_from_data_source*: nome do *data source* a copiar. Pode ser nulo.
- *copy_settings_from_data_source_type*: tipo do *data source* a copiar, pode ser *JSON* ou *XML*. Pode ser nulo.
- *target_database*: Nome da base de datos onde se quere crear a vista base. Non pode ser nulo.

Respecto a estes parámetros, débense ter en conta as seguintes consideracións: só se pode introducir unha dirección URL ou unha cadea de carácteres (lanzaríase un erro cando se indiquen as dúas), e os tres parámetros de *copy_settings_from_data_source* deben ou ser todos

nulos ou ter valor os tres. Por exemplo, se queremos procesar unha especificación OpenAPI que se atopa no URL https://www.books.com/openapi.json coa finalidade de crear unha vista base JSON para a súa operación GET /book/isbn e almacenala na nosa base de datos *books_bd*, o comando para invocar este procedemento almacenado tería a seguinte forma:

GENERATE_VQL_TO_CREATE_VIEW_WITH_OPENAPI_SPEC(https://www.books.com/openapi.json, null, 'prefixo_', '/book/{isbn}', 'GET', 'JSON', null, null, null, 'books_bd')

Se desexamos copiar a configuración dun *data source* de tipo JSON chamado *ds_admin* xa existente na base de datos *admin_bd*, o comando sería:

GENERATE_VQL_TO_CREATE_VIEW_WITH_OPENAPI_SPEC(https://www.books.com/openapi.json, null, 'prefixo_', '/book/{isbn}', 'GET', 'JSON', 'admin_bd', 'ds_admin', 'JSON', 'books_bd')

Parámetros de saída

Este procedemento devolverá unha tupla de dous campos por cada instrución VQL indicando os seguintes valores:

- *datasource_name*: nome do *data source* e da vista base que se vai crear.
- *creation_vql*: sentencia VQL.

Na imaxe 8.6 amósase un exemplo das tuplas que devolvería este procedemento para unha petición POST /book, e nos seguintes apartados explicaráse o proceso que se levou a cabo para obter esta saída.

datasource_name	creation_vql
JSON_POST_book	CONNECT DATABASE target_database;
JSON_POST_book	CREATE DATASOURCE JSON "JSON_POST_book" ROUTE HTTP 'http.CommonsHttp...
JSON_POST_book	CREATE WRAPPER JSON "JSON_POST_book" DATASOURCENAME = "JSON_POST_b...
JSON_POST_book	CREATE TABLE "JSON_POST_book" I18N es_euro (in_isbn : text (extern), in_title : t...
JSON_POST_book	CLOSE;

Figura 8.6: Resultado de executar GenerateVQLToCreateViewWithOpenAPISpec sobre o exemplo da petición POST /book

Fluxo de traballo

O fluxo que executa este procedemento pódese dividir en catro fases principais: obter todos os parámetros da operación necesarios para os seguintes pasos, configuración do *data source* da petición e obtención do seu VQL, configuración do *wrapper* da operación e obtención do seu VQL e onfiguración da vista base da operación e obtención do seu VQL.

Na figura 8.7 amósase un diagrama que resume este fluxo, e nos seguintes apartados iránse desglosando cada unha das fases. O resultado final deste procedemento é obter a vista base

dunha operación, pero para conseguilo é necesario crear antes un *data source* e un *wrapper*, como se explicou no capítulo 5.1 Introdución. Para crear estes tres elementos que representen á operación serán necesarios diversos parámetros da mesma (URL, corpo, resposta, etc), os cales están almacenados na súa especificación OpenAPI. Dado que no primeiro procedemento *GetOperationsInfoFromOpenAPISpec* só se obtiveron tres características que resumían a operación, é necesario volver obter a especificación completa para acceder ao demais campos. Xa que o tempo de execución e os recursos que pode consumir o *parseado* dun documento OpenAPI son mínimos, optouse por levar a cabo este proceso en ambos procedementos, aínda que existiría a posibilidade de utilizar a caché da plataforma para almacenar a especificación. Para evitar repetir código, este segundo procedemento invocará as dúas funcións que se implementaron no primeiro *stored procedure* para obter a especificación: *getOpenApiSpecification()* e *parseSpecification()*.

Unha vez teñamos a especificación como un obxecto OpenAPI, comezará a primeira fase do fluxo na que se conseguirán todos os demais atributos necesarios para describir a operación.

Obter parámetros da operación

No obxecto da especificación OpenAPI obtido ao inicio do procedemento accédese a súa lista de operacións e utilizando o método HTTP e a ruta relativa da entrada búscase o obxecto da petición correspondente, do cal se extraerá a información necesaria para a creación da súa vista base. Xunto co método HTTP que xa temos, os demais campos a obter da definición OpenAPI son os seguintes:

- **Ruta completa da petición:** nas especificacións OpenAPI os obxectos que representan operacións indican cal é a súa ruta relativa, sen embargo para poder crear unha vista base que execute peticións contra a API que define o documento precisase dunha ruta completa (con protocolo, *hostname*, etc). As definicións OpenAPI contan con campo denominado *servers* onde se almacena un *array* de obxectos Server coas súas URLs correspondentes, as cales poden ser relativas para indicar que a ubicación do *host* é a dirección onde se serve o documento OpenAPI. Este campo *servers* é global (aplícase a toda definición), pero cada obxecto PathItem pode conter tamén un *array* de Servers específico para esa petición, polo que para obter a ruta completa dunha operación procederase da seguinte maneira:

  - Se o campo *servers* da operación non está baleiro: seleccionarase a primeiro URL que teña protocolo https. No caso de non haber ningunha con ese protocolo escollerase a primeira dirección do array.

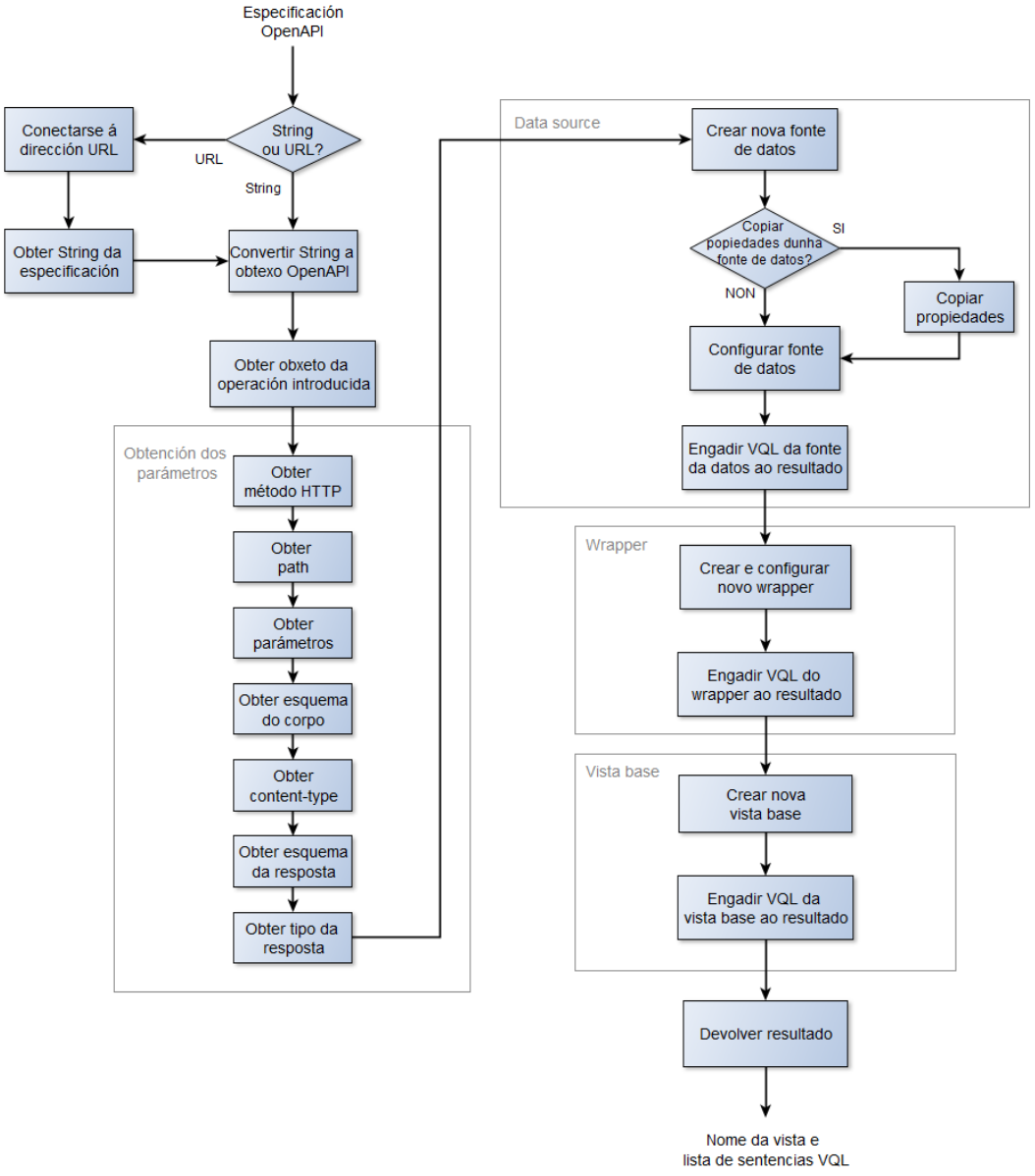

Figura 8.7: Diagrama de fluxo do procedemento GenerateVQLToCreateViewWithOpenAPISpec.

  - Se o campo servers da operación está baleiro: o URL obterase do campo servers global do mesmo xeito que no punto anterior, dando prioridade tamén ao protocolo https.
  - Se todos os URLs dos servidores son relativas: no caso de que a especificación se lle indicara ao procedemento como un URL, obterase o *host* da mesma e concatenarase á primeira URL do *array* e á ruta da operación. No caso de que a definición se introducise como un String e todas as URLs fosen relativas, non habería ningunha forma de construír unha dirección completa e polo tanto a vista base resultante non podería executar consultas contra á API.

- **Corpo da petición como obxecto Schema:** o corpo das peticións represéntase na especificación mediante un obxecto Schema (subconxunto extendido de JSON Schema Specification Wright Draft 00 [39]), que pode ser un obxecto, un *array* ou un tipo primitivo. O *body* da operación será necesario neste formato para crear os obxectos wrapper e base view.

- **Corpo da petición como String:** para configurar a fonte de datos asociada ao wrapper da vista debe converterse o Schema anterior a un content-type válido para HTTP (neste caso permítense JSON e XML). Para realizar a conversión implementáronse funcións recursivas, as cales se invocan a elas mesmas no caso de que haxa subesquemas dentro do obxecto Schema, para transformar todos os campos ao formato escollido.

- **Tipo do corpo da petición:** o parámetro de entrada *preferred_media_type* define que tipo de vista base se quere crear e, por tanto, o content-type da operación asociada a esa vista. O procedemento permite escoller entre JSON ou XML, sen embargo, pode ocorrer que a petición que estamos procesando non soporte o tipo escollido. Para evitar que se produzan erros ao executar a vista base contra a API, compróbase no obxecto OpenAPI que o *array* de tipos soportados pola operación contén o *preferred_media_type* introducido, e no caso que non se atope no *array* seleccionaríase o seu primeiro elemento como content-type da visa base.

- **Resposta da petición como obxecto Schema:** do mesmo xeito que o corpo da petición, os campos da súa resposta tamén se representan utilizando un obxecto Schema. Os códigos de resposta que pode devolver unha petición almacénanse no campo , onde para cada código hai un valor (esquema JSON, mensaxe de erro, etc) asociado. Para configurar os atributos de saída que vai haber na vista base só nos interesa consultar o código de resposta 200 OK, para obter o Schema que devolverá a API ao lanzar unha petición satisfactoria contra ela. De non existir o código de resposta 200, o procedemento utilizará a resposta *default* proporcionada por Swagger para cubrir os casos de

respostas non declaradas.

- **Tipo da resposta:** *content-type* do contido que devolve a resposta do punto anterior.
- **Parámetros da petición:** cada operación do documento OpenAPI conta con un campo *parameters* no que se define un mapa con todos os parámetros de entrada da petición e información relativa a eles: breve descrición, tipo (*query, header, path,* etc), obrigatoriedade, etc.

Despois de gardar todos estes parámetros podemos comezar o proceso de obtención dos comandos para crear o *data source*, o *wrapper* e a vista base correspondentes á petición que estamos a procesar. Este proceso divídese en dúas partes:

1. Crear tres obxectos Java definidos por Denodo: *DataSourceVO*, *WrapperVO* e *BaseViewVO*. Os atributos destes obxectos serán os parámetros da operación HTTP que obtivemos anteriormente (tipo de petición, URL, corpo da petición, estrutura da resposta etc).
2. Obter a sentencia VQL de cada un destes obxectos utilizando métodos internos da empresa.

Nos seguintes parágrafos detallarase este proceso para cada un dos tres tipos de obxectos.

Configuración do *data source* e obtención do seu VQL

O primeiro elemento a crear é o *data source* que represente a petición HTTP. Segundo o valor do parámetro de entrada *preferred_media_type* poderá ser de tipo JSON (obxecto JSONDataSourceVO) ou XML (obxecto XMLDataSourceVO). Unha vez creado o obxecto Java correspondente ao tipo escollido polo usuario comezará a súa configuración, na que se deben establecer os seguintes valores:

- **Nome do datasource:** por defecto o procedemento asigna o nome *<tipoDatasource>_<tipoPeticion>_<rutaRelativa>* (tamén ao *wrapper* e á vista), e no caso de que o usuario proporcionase un prefixo engadiríase ao comezo deste nome. Por exemplo: 'prefixo_JSON_POST_book'.
- **Petición HTTP:** obtida no primeiro procedemento *GetOperationsFromOpenAPISpec*.
- **Ruta completa:** obtida na primeira fase deste procedemento.
- **Corpo da petición como String:** para as peticións que necesiten un corpo será necesario incluílo na configuración do *data source*, indicando os valores de cada campo do

corpo como variables de interpolación. Por exemplo, o corpo dunha petición POST /book en formato JSON sería o seguinte: *{isbn: @{in_isbn}, {title: @{in_title}*. Desta forma, cando se execute unha vista base creada sobre este *data source*, solicitarase ao usuario que introduza valores para as variables marcadas co formato *@{in_nomeVariable}*. Este parámetro foi obtido na primeira fase do procedemento.

- **Tipo do corpo:** obtido tamén na primeira fase.
- **Proxy da conexión:** estableceuse o proxy por defecto de Denodo.
- **Autenticación:** establecese a OFF por defecto, supoñendo que non é necesaria.

Esta sería a configuración base para todos os *data sources*, sen embargo, poderíase ampliar se o usuario seleccionou a opción de copiar a configuración doutro *data source*, é dicir, se as tres variables de entrada *copy_settings_from_data_source_data base*, *copy_settings_from_data_source* e *copy_settings_from_data_source_type* teñen valor distinto a nulo. Neste caso as propiedades que se copiarán son: autenticación, paxinación, configuración de proxy e de filtros. Os pasos levados a cabo para copiar estas propiedades son os seguintes:

1. Comprobar se existe a fonte de datos chamada *copy_settings_from_data_source* de tipo *copy_settings_from_data_source_type* na base de datos *copy_settings_from_data_source_database*.
2. Comprobar que o usuario que está executando este procedemento ten permisos suficientes para acceder a ese data source.
3. Se os dous pasos anteriores foron satisfactorios, copiar as súas propiedades ao noso obxecto JSONDataSourceVO ou XMLDataSourceVO.

Unha vez configurado o obxecto *data source* para que represente todas as características da nosa operación, xa podemos obter o seu VQL. Este paso realizarase co método interno de Denodo *toVQL()*. Na seguinte figura (8.8) ilústrase un exemplo do comando VQL que se devolvería no caso dun obxecto JSONDataSourceVO configurado para a petición POST /book.

Configuración do *wrapper* e obtención do seu VQL

Despois de crear o *data source*, comezará a creación e configuración do obxecto que represente o seu *wrapper*. O obxecto do *wrapper* dependerá tamén do tipo da fonte de datos, podendo ser JSONWrapperVO ou XMLWrapperVO. Para configuralo serán necesarios os seguintes parámetros:

- **Nome do *wrapper***: o mesmo que o do *data source*, *<tipoWrapper>_<tipoPeticion>_<ruta Relativa>* (por exemplo: 'prefixo_JSON_POST_book').

```
CREATE DATASOURCE JSON "JSON_POST_book"
    ROUTE HTTP 'http.CommonsHttpClientConnection,120000' POST
    'https://www.books.com/book}'
    POSTBODY '\{"isbn":"@{in_isbn}", "title":"@{in_title}",
    "author":"@{in_author}", "price":"@{in_price}"\}'
    HEADERS (
        'Accept' = 'application/json'
    )
    AUTHENTICATION OFF
    PROXY DEFAULT;
```

Figura 8.8: Exemplo do comando VQL para crear un data source

- **Nome do data source** creado previamente, para indicar que o *wrapper* vai asociado a el.
- **Parámetros da URL:** en caso de habelos son necesarios para o esquema do *wrapper*. Posteriormente explicaráse como debe ser este esquema. Obtivéronse no primeiro procedemento GetOperationsFromOpenAPISpec.
- **Corpo da petición como obxecto Schema:** no caso de que a petición leve corpo tamén é necesario para crear o esquema do wrapper. Foi obtido neste procedemento.
- **Resposta da petición como obxecto Schema:** necesario para crear o esquema do wrapper. Tamén se obtivo neste procedemento.

A dificultade da configuración do *wrapper* radica no seu esquema ou *metadata*. A *metadata* do *wrapper* debe conter os valores de entrada da operación (parámetros do URL ou corpo da petición) e de saída (resposta), indicando para cada campo o seu tipo de dato (*string, integer, array,* etc). Ademais, como se explicou na 5.1 Introdución para cada un dos campos é necesario especificar unha serie de etiquetas: operadores aplicables, obligatoriedade, multiplicidade e posibles valores. Toda esta información é obtida como resultado de procesar mediante funcións recursivas (xa que pode haber rexistros ou arrays anidados dentro dos esquemas) os obxectos Schema do corpo e a resposta que ofrece a especificación OpenAPI, obtendo toda a información necesaria para cada campo e engadíndoo ao esquema do *wrapper* un por un. A diferencia da creación do *data source* onde se utilizaba o corpo da petición en formato *string*, neste caso non aportaría todos os datos necesarios para reproducir de forma precisa a petición, por iso se utiliza o obxecto Schema.

Cando o *wrapper* conteña o nome do *data source* asociado e o esquema da resposta que vai devolver, xa é posible obter o seu comando VQL mediante o método interno de Denodo *generateWrapperVQL()*. Seguindo co exemplo da figura 8.8 dunha petición POST /book onde no corpo se envía un recurso ”book” cos campos isbn, título, autor e prezo, e a resposta unha

vez executada a petición correctamente devolvese ese mesmo obxecto ”book”, o comando VQL devolto para este obxecto JSONWrapperVO sería o seguinte:

```
CREATE WRAPPER JSON "JSON_POST_book"
    DATASOURCENAME = "JSON_POST_book"
    TUPLEROOT '/JSONFile'
    OUTPUTSCHEMA (jsonfile = 'JSONFile' : REGISTER OF (
        in_isbn = 'in_isbn' : 'java.lang.String' (OBL) EXTERN,
        in_title = 'in_title' : 'java.lang.String' (OBL) EXTERN,
        in_author = 'in_author' : 'java.lang.String' (OBL) EXTERN,
        in_price = 'in_price' : 'java.lang.Integer' (OBL) EXTERN
        isbn = 'isbn' : 'java.lang.String' (OBL) SORTABLE,
        title = 'title' : 'java.lang.String' (OPT) SORTABLE,
        author = 'author' : 'java.lang.String' (OPT) SORTABLE,
        price = 'price' : 'java.lang.Integer' (OPT) SORTABLE
    )
    );
```

Figura 8.9: Exemplo do comando VQL para crear un *wrapper*

Os campos extraídos do corpo da petición levan o prefixo ”in_” das variables de interpolación e están marcados como obrigatorios (OBL) e como valores non devoltos pola fonte de datos (EXTERN). A resposta da fonte de datos son os catro seguintes campos, dos que só é obrigatorio o ISBN.

Configuración da vista base e obtención do seu VQL

Por último, despois de crear o *data source* da operación e o seu *wrapper* asociado, xa podemos crear a vista base que executará as peticións sobre a fonte de datos utilizando o *wrapper* de intermediario. O obxecto Java que representa unha vista base é BaseViewVO (non é necesario diferenciar entre JSON e XML). Adicionalmente, Denodo conta con un método de utilidade que configura automaticamente unha vista base para un *data source* e un *wrapper* dados. Polo tanto, só será necesario executar o método *getBaseViewVO(DataSourceVO dsVO, WrapperVO wpVO, String targetDatabase, i18n)*, gardar o obxecto BaseViewVO que devolve e xa teremos a vista base da nosa operación.

Do mesmo xeito que os demais compoñentes, co método de Denodo *generateViewVQL()* obteremos o comando VQL que crearía este obxecto BaseViewVO. Na figura 8.10 móstrase a sentencia para unha vista creada sobre o *data source* e o *wrapper* dos pasos anteriores:

Todos estes comandos VQL que se foron obtendo son almacenados no resultado do procedemento almacenado, e unha vez finalizado o *procedure* son enviados xunto co nome dos elementos á interface para a súa execución. A continuación móstrase un exemplo das tuplas que se devolverían para o exemplo da petición POST /book:

```
CREATE TABLE "JSON_POST_book" I18N es_euro (
        in_isbn : text (extern),
        in_title : text (extern),
        in_author : text (extern),
        in_price : int (extern),
        isbn : text,
        title : text,
        author : text,
        price : int
    )
    CACHE OFF
    TIMETOLIVEINCACHE DEFAULT
    ADD SEARCHMETHOD "JSON_POST_book" (
        I18N us_est
        CONSTRAINTS (
            ADD in_isbn (=) OBL ONE
            ADD in_title (=) OPT ANY
            ADD in_author (any) OPT ANY
            ADD in_price (any) OPT ANY
            ADD isbn NOS ZERO ()
            ADD title NOS ZERO ()
            ADD author NOS ZERO ()
            ADD price NOS ZERO ()
        )
        OUTPUTLIST (isbn, title, author, price)
        WRAPPER (JSON "JSON_POST_book")
    );
```

Figura 8.10: Exemplo do comando VQL para crear unha vista base

Operacións do procedemento

Os métodos que implementa este procedemento almacenado para levar a cabo o fluxo descrito son os seguintes:

- **Object[] getMethodInformation(OpenAPI openAPI, String httpMethod, String inputPath, String preferredMediaType, String hostname):** obtén todos os parámetros necesarios da operación para crear os obxectos Java DataSourceVO, WrapperVO e BaseViewVO.
- **String createName(String inputNamePrefix, String inputPath, String inputHttpMethod, String preferredMediaType, String targetDatabase, DenodoSession session):** crea o nome que van levar os compoñentes (*<tipoDataSource>_<tipoPeticion>_<rutaRelativa>*).
- **String getValidMediaType(LinkedHashMap<String, MediaType> contentTypes, String preferredMediaType):** comproba que o tipo de *data source* (JSON ou XML) introducido polo usuario estea soportado pola operación obtendo súa lista de *media-types* admitidos. No caso de que non estea soportado comprobarase o segundo tipo, por

exemplo, se o usuario escolleu JSON pero non é valido comprobarase que XML apareza na lista de *media-types* e será utilizado no lugar de JSON. Se ningún dos dous é valido mostrarase un erro e o procedemento finalizará.

- **String parametersToInterpolatingVariables(List<Parameter> parameters, String inputPath):** converte os parámetros dun URL en variables de interpolación, é dicir, reescribeos co formato *@{in_nomeVariable}*. Por exemplo, para unha ruta */book/isbn*, devolverá */book/@{in_isbn}*.

- **List<String> getVQL(DenodoSession session, HashMap<String, Object> configurationParameters):** crea e configura os obxectos DataSourceVO, WrapperVO e BaseViewVO coa información almacenada no mapa *configurationParameters* e obtén os seus comandos VQL.

- **JSONWrapperMetaRegisterRawVO createMetadataJSON(List<Parameter> parameters, Schema postBodySchema, Schema responseSchema):** método auxiliar de *getVQL()* que crea o esquema dun *JSONWrapperVO* a partir dos parámetros do URL e o Schema do corpo e da resposta.

- **void bodyToMetadaJSON(String key, Schema bodySchema, WrapperMetaRegisterRawVO metaRegister, String inTag, List<String> requiredVariables):** función recursiva auxiliar de *createMetadataJSON()* que procesa o obxecto Schema do corpo da petición para convertelo ao formato do esquema do *wrapper*. No exemplo da figura 8.9 esta función devolvería os catro primeiros campos do *OUTPUTSCHEMA*: in_isbn, in_title, in_author e in_price.

- **void responseToMetadaJSON(String key, Schema value, WrapperMetaRegisterRawVO metaRegister):** función recursiva auxiliar de *createMetadataJSON()* que procesa o obxecto Schema da resposta da petición para convertilo ao formato do esquema do *wrapper*. No exemplo da figura 8.9 esta función devolvería os catro últimos campos do *OUTPUTSCHEMA* isbn, title, author e price.

- **XMLWrapperMetaRegisterRawVO createMetadataXML(List<Parameter> parameters, Schema postBodySchema, Schema responseSchema):** método auxiliar de *getVQL* que crea o esquema dun *XMLWrapperVO()* a partir dos parámetros do URL e o Schema do corpo e da resposta.

- **void bodyToMetadaXML(String key, Schema body, WrapperMetaRegisterRawVO metaRegister, String inTag, List<String> requiredVariables):** función recursiva auxiliar de *createMetadataXML()* que procesa o obxecto Schema do corpo da petición para convertilo ao formato do esquema do *wrapper*.

- **void responseToMetadaXML(String key, Schema value, WrapperMetaRegisterRawVO metaRegister, List<String> requiredVariables):** función recursiva auxiliar de *createMetadataXML()* que procesa o obxecto Schema da resposta da petición para convertilo ao formato do esquema do *wrapper*.
- **Class getClassType(Schema schema):** este método utilízase nas funcións recursivas que procesan os Schemas para saber se se está a procesar un campo simple (*integer, string*, etc) ou complexo (*array*, estrutura, rexistro, etc).
- **void schemaToJson(String key, Schema schema, JSONObject json, List<String> requiredVariables, List<String> variables, String currentLevel):** función recursiva que converte un obxecto Schema a un obxecto JSON.
- **String schemaToXml(Schema schema, int indent, String currentLevel):** función recursiva que converte un obxecto Schema a un String con formato XML, construíndo as etiquetas necesarias para cada atributo.
- **String parseElements(String name, Schema property, int indent, String currentLevel):** método auxiliar de *schemaToXML()*.
- **String indent(int indent):** método auxiliar de *schemaToXML()*.
- **String quote(String string):** método auxiliar de *schemaToXML()*.

8.4.5 Wrapper GraphQL

O funcionamento deste *custom wrapper* céntrase en ser capaz de converter as peticións recibidas dende a interface á sintaxe adecuada para executalas sobre o esquema GraphQL e devolver a información obtida ao usuario. Para levar a cabo este fluxo, o *wrapper* necesita unha serie de parámetros para poder conectarse primeiro ao servizo GraphQL que se vai consumir. Todos estes parámetros especifícanse no método *getDataSourceInputParameters()* que proporciona a API de Denodo e son os seguintes:

- **URL do esquema de datos:** como se explicou no capítulo 2 Base Tecnolóxica, o funcionamento de GraphQL baséase nun esquema no que se definen os posibles datos a consultar do servizo. Para poder executar peticións é imprescindible coñecer este esquema, polo que o usuario deberá proporcionar a súa dirección para poder consultalo. Por exemplo: https://www.books.com/graphql/schema.json
- **URL do servizo GraphQL:** do mesmo xeito que cando se consulta unha API, é necesario saber a que URL se deben enviar as peticións. Por exemplo: https://www.books.com/graphql

- **Credenciais de autenticación:** deberán aportarse usuario e contrasinal de que se utilice a autenticación HTTP Basic para conectarse ás URL

Unha vez definida a información necesaria para conectarse ao servizo, comezan a especificarse os parámetros de entrada que van necesitar as peticións executadas dende a interface. Neste caso o único parámetro que se necesita para executar unha consulta é a *query* GraphQL escollida polo usuario. No método *getInputParameters()* indicarase un só parámetro obrigatorio de tipo *string*.

O seguinte paso despois de especificar todos estes valores, é definir o esquema de saída para a *query* que se vai executar (na imaxe 8.13 móstrase un diagrama do fluxo que se vai explicar no resto deste apartado). Para este paso é necesario o esquema do parámetro "URL do esquema de datos" que se mencionou anteriormente (empregarase de exemplo o da figura 8.11). O *wrapper* executa un GET sobre esa dirección para obter o JSON do esquema, o cal será procesado facendo uso da librería *graphql-java* para convertilo nun obxecto e poder acceder facilmente aos seus campos. O primeiro paso será comprobar que a *query* introducida polo usuario está permitida. Todos os esquemas GraphQL teñen un tipo de dato chamado "Query" onde se recolle o conxunto de operacións que se poden executar, polo tanto, no obxecto do esquema accedese a este rexistro para comprobar que contén a consulta que aportou o usuario.

No caso de ser unha operación válida, comezará o paso de obter o esquema que devolvería a súa execución. Como se pode ver na figura 8.11, no tipo "Query" para cada petición móstrase que tipo de dato devolven, neste caso concreto un obxecto *Book* ou unha lista de obxectos *Book* (o símbolo ! indica que non pode ser nulo). Polo tanto, para o caso dunha petición *bookById* o esquema de saída tería os campos do tipo de dato *Book*, cos valores identificador, nome, páxinas e autor. Sen embargo, o campo *author* é de tipo obxecto, para o cal hai definido outro esquema (identificador, nome e apelidos) o cal habería que engadir ao esquema principal. Como consecuencia, o procesado do esquema implementouse de forma recursiva (mediante unha función que se chama a sí mesma), xa que podería haber máis niveis de anidamento.

Neste punto do fluxo teríamos todos os parámetros de entrada e o esquema da resposta para consulta que estamos a procesar, polo que xa poderíamos comezar coa execución da consulta. Nos servizos GraphQL as *queries* execútanse mediante peticións POST que levan no corpo a sintaxe da consulta. Desta maneira, o seguinte paso do fluxo é crear e configurar esta petición POST JSON e lanzala contra o servizo que se indicou no parámetro do inicio "URL do servizo GraphQL". Cando todos os parámetros da petición estén configurados, engadirase ao corpo a consulta a executar convertida ao formato da figura 8.12. Como se pode apreciar na imaxe, non se devolven todos os campos dos obxectos *Book* e *Author*, só unha parte deles. Unha das principais características de GraphQL e a posibilidade de escoller os campos exactos a obter do repositorio de datos, por tanto na construción do corpo do JSON tivéronse en conta tamén os campos elixidos polo usuario, non só os parámetros de entrada da consulta.

```
type Query {
    book: [Book]!
    bookById(id: ID): Book
    booksByAuthor(id: ID): [Book]!
}

type Book {
    id: ID
    name: String
    pageCount: Int
    author: Author
}

type Author {
    id: ID
    firstName: String
    lastName: String
}
```

Figura 8.11: Exemplo dun esquema GraphQL

Unha vez conformada completamente a petición POST, esta execútase contra a URL do servizo e gárdase a resposta recibida (como a que se amosa na figura 8.12). Dado que na librería *graphql-java* non existe un obxecto específico para a resposta dunha petición, a información devolta converteuse a un JSONObject. Compróbase que non se devolverán erros e se a petición se executou correctamente, comezará o último paso do fluxo: procesar a resposta para convertela ao formato adecuado para enviala de volta á interface. Da mesma maneira que ocorría cos esquemas pode haber campos anidados no JSON, polo que o procesado da resposta tamén debe ser recursivo para construír posición por posición un *array* de obxectos con todos os campos e os seus correspondentes valores. Este *array* engadiráse ao obxecto *CustomWrapperResult* e este será enviado a interface unha vez rematado o fluxo.

Métodos do *wrapper*

Na seguinte lista descríbense todos os métodos implementados para o *wrapper*, sen incluír os da API de Denodo que se explicaron na Arquitectura do *backend*.

- **void getSchemaComplexType(TypeDefinitionRegistry typeDefinitionRegistry, TypeDefinition typeSchema, List<CustomWrapperSchemaParameter> subSchemaParameters):** función recursiva que obtén o esquema dunha petición.
- **String buildGraphQLQuery(List<CustomWrapperSchemaParameter> baseViewSchema, List<CustomWrapperFieldExpression> projectedFields, String method, List<String> parameters):** método que constrúe a consulta GraphQL para o formato

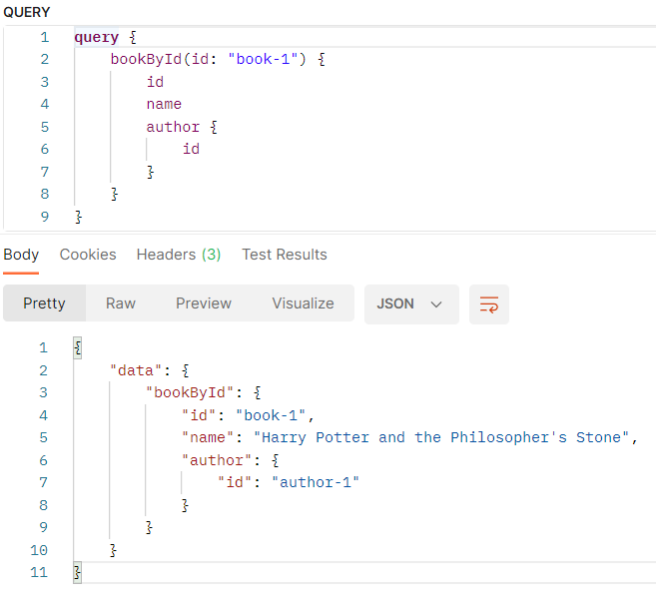

Figura 8.12: Exemplo do corpo dunha petición POST GraphQL e a súa resposta

do corpo dunha petición POST.

- **boolean isComplexType(String fieldName, List<CustomWrapperSchemaParameter> schema):** método auxiliar de *buildGraphQLQuery* que comproba se o campo dun esquema é de tipo simple ou é un obxecto.

- **String subFieldsToQueryString(String fieldName, List<CustomWrapperSchemaParameter> baseViewSchema):** método recursivo auxiliar de *buildGraphQLQuery* que procesa campos complexos e obtén os seus subesquemas para engadilos ao esquema principal dunha petición.

- **void processRequestResponse (JSONObject subObject, List<CustomWrapperSchemaParameter> baseViewSchema, List<CustomWrapperFieldExpression> projectedFields, Object[] resultData, int resultPosition):** función recursiva que converte a resposta JSON dunha petición POST ao un *array* de obxectos.

- **List <CustomWrapperSchemaParameter> getSubFieldsFromBaseViewSchema (String subField, List<CustomWrapperSchemaParameter> baseViewSchema):** método auxiliar de *processRequestResponse()* para procesar campos complexos.

- **int toJavaSqlType(String type):** recibe o tipo dun campo do esquema GraphQL e devolve o seu tipo equivalente para SQL.

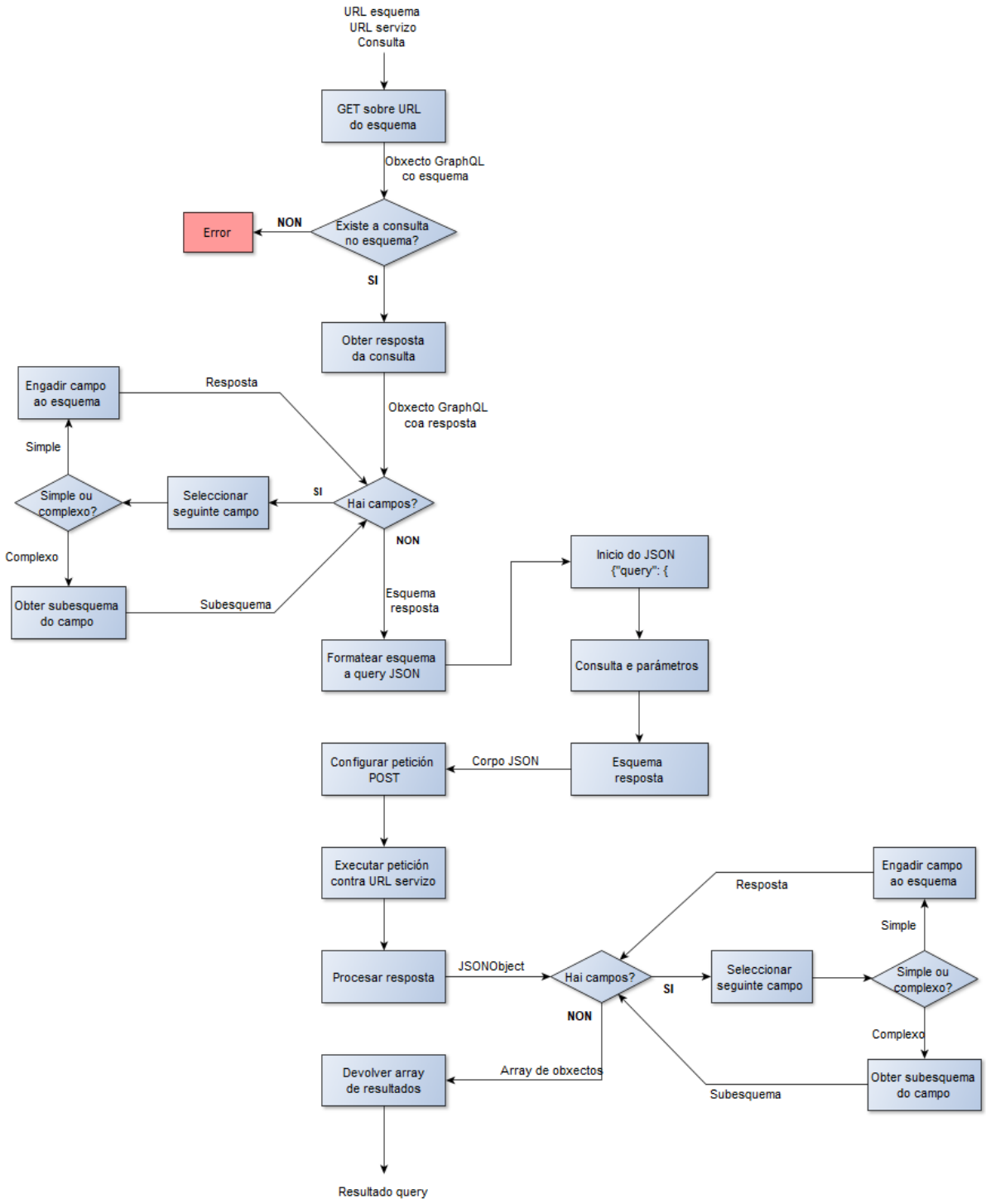

Figura 8.13: Fluxo de traballo do *wrapper* GraphQL

8.5 Subsistema *frontend*

8.5.1 Obxectivos

O obxectivo do subsistema *frontend* é ofrecer ao usuario unha forma visual e sinxela de acceder á funcionalidade desenvolvida no *backend*. Como se mencionou no 8.4 Subsistema backend, a aplicación da empresa sobre a que se traballou xa implementa a comunicación entre a interface e o servidor, polo que non foi necesario desenvolvela neste proxecto. A tecnoloxía principal utilizada para crear o *frontend* foi a librería Java Swing, a cal ofrece un amplo catálogo de compoñentes para desenvolver interfaces (menús, *checkboxes*, botóns, áreas de texto, etc) que funcionan de forma independente á plataforma.

8.5.2 Arquitectura

A estrutura do *frontend* para a creación dunha vista a partir dunha especificación OpenAPI está constituída por unha clase principal que crea o *wizard* que ve o usuario e as clases que lle implementan a lóxica de negocio sobre a que traballa a interface. Na figura 8.14 amósase a estrutura de clases deste subsistema, e ao longo deste apartado desglosaranse as características de cada unha. O novo *wizard* pódese acceder dende a *Adminstration Tool* entrando no menú despregable *Tools* e seleccionando *Create views from OpenAPI*.

Para engadir novos *wizards* á *Adminstration Tool*, debe crearse unha clase que siga o patrón *singleton*, xa que só pode haber unha instancia de cada *wizard*. Neste caso, trátase da clase *CreateViewsFromOpenAPIDockable* da figura 8.14. Esta clase só se encarga de crear unha instancia da ventá na que se vai incluír o *wizard*, non de definir os elementos que o compoñerán. Cando se crea a única instancia desta clase, créase tamén unha instancia da clase principal na que está definido o contido da interface. Esta clase explícase a continuación.

A clase que constrúe o *wizard* da funcionalidade é *CreateViewsFromOpenAPI*, e nela defínense todos os elementos visuais da interface: botóns, etiquetas, *checkboxes*, caixas de selección, árbore, etc. O *wizard* dividise en dúas partes: o panel superior onde se introducirá a especificación e o panel inferior onde se amosará unha árbore coas operacións atopadas. Ambos paneis créanse cando se invoca a interface, pero o panel inferior permanece oculto ata que se procese unha especificación.

O panel superior do *wizard* (pode verse no capítulo 7 Requisitos do sistema, na sección 7.5 Prototipado de interface) estará formado por un menú onde se debe escoller entre introducir a dirección onde se atopa o documento OpenAPI ou seleccionar un ficheiro local. Unha vez que o usuario aporte unha especificación poderá comezar o procesado da mesma. No caso de que se introducira un URL a interface comprobará que a dirección sexa válida, e no caso de que se seleccionase un arquivo a clase encargarase de ler o seu contido e gardalo nun obxecto

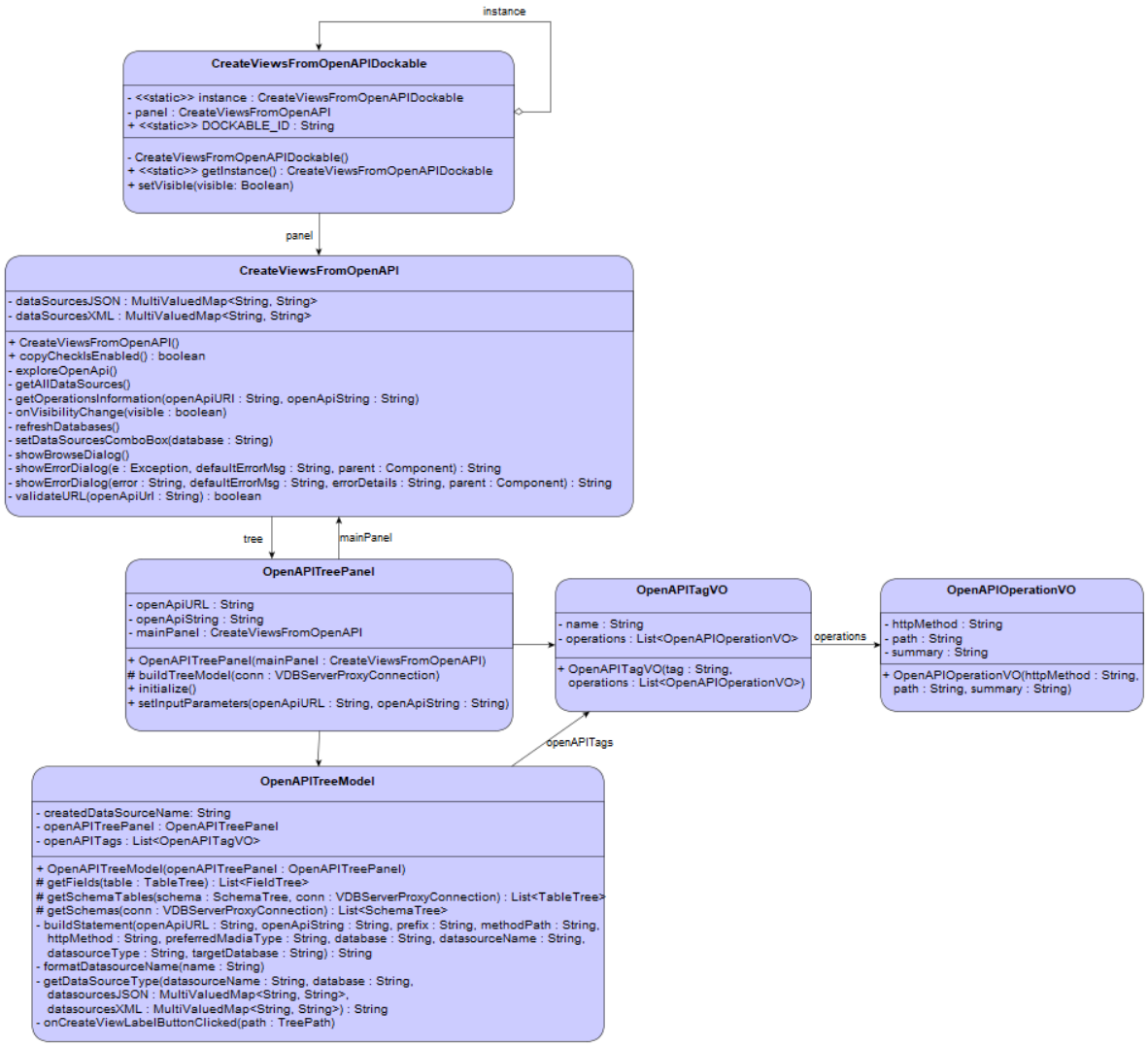

Figura 8.14: Diagrama de clases do subsistema *frontend*

string.

Despois de realizar todas as comprobacións, a clase pasaralle o URL ou o *string* á clase auxiliar *OpenAPITreePanel* a cal utilizaraos para construír o comando que invoque o procedemento almacenado GetOperationsFromOpenAPISpec. Como se mencionou na explicación do *backend*, a instrución que debe crear *OpenAPITreePanel* para chamar este procedemento ten a seguinte sintaxe: *GET_OPERATIONS_FROM_OPEN_API_SPEC(<url>, null)* ou *GET_OPERATIONS_FROM_OPEN_API_SPEC(null, <string_especifiacion>)*. Este comando será enviado ao *backend* para executarse a través dun método interno de Denodo: *VDBShellFacade.executeQuery()*. O resultado serán tantas tuplas como operacións se atoparon na especificación, indicando para cada unha o seu método HTTP, ruta relativa, descrición e nome dos

recursos. Este conxunto de tuplas utilizase para formar a árbore de operacións que verá o usuario no panel inferior, o cal listará as operacións agrupadas polo recurso ao que acceden (por exemplo un grupo coas operacións que utilicen o recurso */book*, outro grupo para */movies*, etc).

Para facilitar o procesamento das tuplas para crear a árbore, creáronse dúas clases internas dentro de *OpenAPITreePanel*: *OpenAPIOperationVO* e *OpenAPITagVO*. A clase *OpenAPIOperationVO* permite crear obxectos que representan unha operación OpenAPI cos seguintes atributos: método HTTP, ruta relativa e descrición. Por outra banda, a clase *OpenAPITagVO* crea un obxecto que representa o nome dun recurso xunto con todas as operación que acceden a el, polo que ten como atributos o nome do recurso e unha lista de obxectos *OpenAPIOperationVO*. Facendo uso destes dous obxectos, recorrese o conxunto de tuplas devoltas polo procedemento asignando cada operación ao recurso correspondente.

Unha vez ordenadas todas as operacións, a lista de obxectos *OpenAPITagVO* resultante é utilizada pola clase *OpenAPITreeModel* para compoñer a árbore. A clase define o modelo que vai ter a árbore, é dicir, que información se vai mostrar e que accións se poderán executar sobre el. Como se mencionou no capítulo 7 Requisitos do sistema e na sección 7.5 Prototipado de interface, na árbore mostrarase para cada operación o seu método HTTP, ruta relativa, descrición e un botón chamado *Create base view*, cuxo funcionamento se explicará nos seguintes parágrafos. Nesta clase *OpenAPITreeModel* establece a estrutura da árbore pero non se encarga do seu ensablamento coas operacións, este proceso relégase na clase de Denodo da que extende: a *AbstractDataSourceTreeModel*, a cal facendo uso desta estrutura e da lista de *OpenAPITagVO* constrúe o obxecto da árbore que se vai ver na interface. Dado que este proceso execútase nunha clase xa implementada pola empresa, non se detallará neste capítulo.

Chegados a este punto, coa especificación OpenAPI procesada e a árbore de operacións definida e construída, a clase principal *CreateViewsFromOpenAPI* mostrará ao usuario o panel inferior do *wizard*. Neste panel, ademais da lista de operacións inclúese un menú para configurar a creación das vistas base que ofrecerá as seguintes opcións:

- Elixir o tipo da vista (e por tanto do seu *data source* e do seu *wrapper*), que pode ser JSON ou XML.
- Escoller se se desexa copiar a configuración doutro *data source*, seleccionando, en caso afirmativo, nun menú despregable o seu nome, o seu tipo e a base da datos na que se atopa.
- Indicar un prefixo co que nomear á vista.
- Seleccionar en que base de datos se quere almacenar a nova vista.

Agora o usuario xa ten a posibilidade de crear vistas base dunha operación premendo o seu botón *Create base view* na árbore despois de escoller a súa configuración. Este botón levará a cabo a última fase da lóxica de negocio da interface, na que se invocará o procedemento almacenado GenerateVQLToCreateViewWithOpenAPISpec, utilizando como parámetros de entrada para o seu comando a especificación OpenAPI (URL ou string), a información da operación (método HTTP e ruta relativa) e os valores seleccionados no panel de configuración (tipo da vista, prefixo, base de datos na que se quere crear e en caso de copiar a configuración doutro *data source*, a seu nome, tipo e base de datos).

Dado que o botón *Create base view* definiuse na clase *OpenAPITreeModel*, esta encargarase de obter todos os parámetros, formar o comando que invoca o procedemento (*GENERATE_VQL_TO_CREATE_VIEW_WITH_OPENAPI_SPEC()*, executalo con *VDBShellFacade.executeQuery()* e almacenar o nome e os comandos VQL que devolve. Posteriormente, a clase executa unha por unha as instruccións VQL recibidas para crear o *data source*, o *wrapper* e a vista base. Unha vez estean todos os elementos creados, utilizando o nome devolto polo procedemento, o cal se corresponde co nome dos tres elementos, a clase *OpenAPITreeModel* pode abrir na interface en dúas novas pestanas o *wizard* de configuración do *data source* e da vista base. Se a opción de copiar a configuración doutro *data source* non estaba seleccionada, o foco da interface cambiarase á pestana do *data source* que se acaba de crear, no caso contrario o enfoque manterase no *wizard* OpenAPI.

O usuario poderá executar este proceso as tantas veces como operacións queira crear, podendo modificar os parámetros de configuración para cada unha delas.

Capítulo 9

Implementación

9.1 Software requerido

No presente subapartado listáranse o software e as librerías utilizadas para o desenvolvemento do proxecto. No que respecta ao sistema operativo, ao longo de todo o proxecto foi utilizado Windows xa que é o sistema co que traballa a empresa. Para a implementación do *backend* e do *frontend* foi necesario o seguinte software:

- Os módulos da **Plataforma Denodo** nos que se atopa o código fonte da aplicación.
- O kit de desenvolvemento **Java OpenJDK 8+**.
- A ferramenta de construción de proxectos **Maven 3.6.3+**.
- O software para revisión de código **SonarLint**.

9.2 Estrutura

A estrutura do proxecto divídese en dúas partes: o *backend*, no módulo do servidor da Plataforma Denodo, e o *frontend*, no módulo da interface. Dado que ambos son módulos Maven, os dous seguen a súa arquitectura xeral: un ficheiro pom.xml na raíz do módulo, código principal en *src/main/java* e tests en *src/test*.

Na figura A.15 pode verse a estrutura de paquetes correspondente ao *backend*. Como se pode observar, consta de dúas carpetas principais: *denodo-vdp-server*, que corresponde ao módulo onde se atopa o código fonte do Virtual DataPort Server, e *denodo-vdp-testng*, módulo de tests onde a empresa almacena as *suites* que executan probas sobre as clases do módulo servido. Estes tests explicaranse no seguinte capítulo 10 Probas.

En canto á estrutura do *frontend* (figura 9.2), está contida nun único paquete *denodo-vdp-admintool*, no que están contidas as clases que constrúen a interface de escritorio Admnistration Tool e os seus tests.

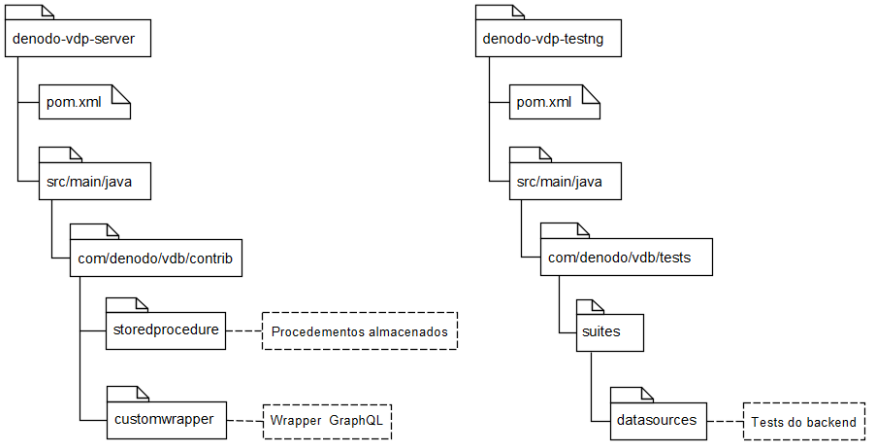

Figura 9.1: Diagrama de paquetes do *backend*

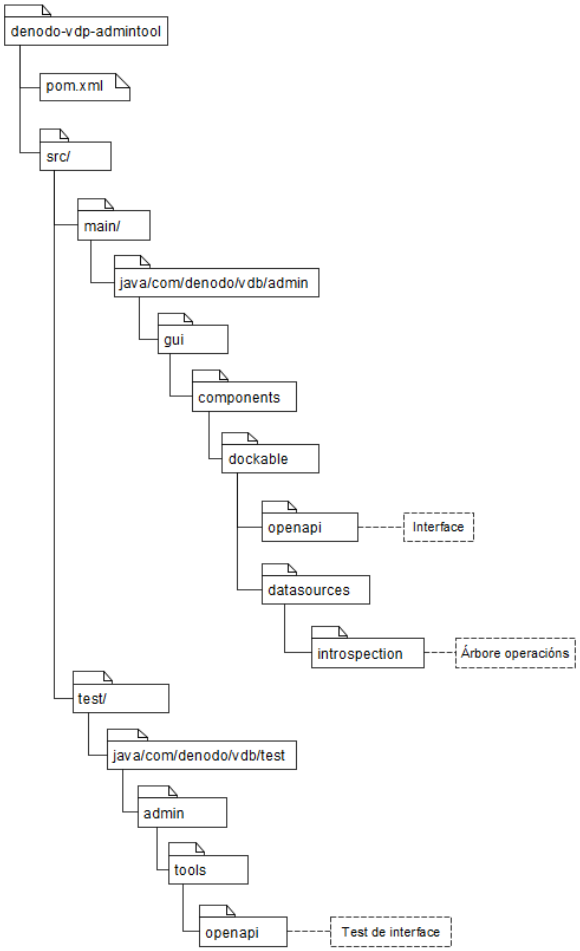

Figura 9.2: Diagrama de paquetes do *frontend*

9.3 Instrucións de compilación

Como se mencionou anteriormente, para a compilación do proxecto a empresa utiliza Apache Maven, polo que os módulos do servidor e da interface compílanse mediante o comando **mvn package** para posteriormente ser executados dende o IDE IntelliJ Idea.

Capítulo 10

Probas

10.1 Introdución

No ciclo de vida de calquera software é fundamental unha fase de probas a través da cal verificar o correcto funcionamento do sistema e atopar e reparar posibles erros. Ademais, nun entorno constantemente cambiante como o da empresa no que se modifica con relativa frecuencia o software para reparar problemas ou engadir melloras, as probas son esenciais para cercionarnos de que os novos cambios non afectan negativamente ao código desenvolvido anteriormente. Deste xeito, grazas ás probas ademais de depurar posibles *bugs* que haxa no software, se nun futuro se produce unha regresión no código por algunha modificación engadida poderemos detectalo rapidamente. Como consecuencia, a empresa pon especial énfase nas probas, tanto de *backend* como de *frontend*, xa que axudan a reducir o tempo e os recursos necesarios para solventar erros.

Adicionalmente, as probas tamén se utilizan para comprobar que o software implementado cumpre os requisitos iniciais do proxecto e as necesidades do cliente. Neste apartado expoñeranse as probas que se realizaron para *testear* o sistema desenvolvido.

10.2 Probas de integración do *backend*

Deseñáronse unha serie de probas para *testear* as sentencias VQL de execución dos dous procedementos almacenados que constitúen o *backend*, intentando abarcar o maior número de escenarios diferentes. Para cada sentencia aplícanse todas as combinacións de parámetros que se poden dar, utilizando como exemplo de especificación OpenAPI a que ofrece Swagger: a *Petstore - OpenAPI 3.0* [40]. Unha vez se executan as sentencias compárase o resultado obtido co contido dun ficheiro no que se almacena o resultado correcto que debería ser devolto pola correspondente sentencia. No caso que ambos resultados sexan iguais, a proba concluiría satisfactoriamente. Todas estas probas están recollidas nunha *suite* chamada *OpenAPITests*,

no módulo *denodo-vdp-testng*.

Para automatizar as probas mencionadas no parágrafo anterior, utilizouse o *framework* TestNG inspirado en JUnit, polo que ten un funcionamento similar, combinado cun *framework* desenvolvido pola empresa que inclúe funcións para simplificar tarefas moi repetidas nos *tests* (execución de consultas VQL, comparación de sentencias VQL, comprobación de errores devoltos por unha consulta, etc).

Cando se utiliza TestNG para a creación de probas, en primeiro lugar debe implementarse un método chamado *data provider*, no cal se establece unha configuración que se lle pasará por parámetro a cada proba, e leva a etiqueta *@DataProvider*. Neste caso crease un obxecto interno da empresa chamado *Configuration* onde se gardan os parámetros que se van necesitar nas probas: nome da *suite* e nome da base de datos que se vai utilizar na Denodo Platform.

En segundo lugar, despois de crear o *data provider* é necesario implementar un método denominado *createMetadata()* coa etiqueta *@AfterClass*. Este método execútase sempre ao inicio do fluxo e crea todos os elementos necesarios para os as probas, neste caso só é necesario crear unha base de datos pero pode utilizarse para crear *data sources*, vistas, etc.

Tendo implementados xa estes dous métodos, podemos comezar a crear as probas. Levan a etiqueta *@Test(dataProvider = "<nome_data_provider")*, a cal indica de que *data provider* recibe a configuración, xa que pode haber varios métodos deste tipo nunha mesma *suite*. En cada proba execútase o comando dun dos procedementos cos parámetros correspondentes e compróbase o resultado.

Por último, é necesario crear un método *dropMetadata()* coa etiqueta *@AfterClass* que se execute despois de todas as probas e elimine os elementos creados durante elas (bases de datos, *data sources*, *wrappers*, etc). Desta maneira evítase deixar elementos residuais que poidan interferir noutras probas.

10.3 Probas de interface

Para comprobar que a interface se executa correctamente e realiza o comportamento desexado, implementáronse probas de interface que se executan sobre unha aplicación Java Swing e levan a cabo o fluxo de traballo automaticamente e comproban en cada momento o estado da interface para verificar que realiza as transicións necesarias. É dicir, estes *tests* execútanse sobre a interface de escritorio da Plataforma Denodo, abrindo os *wizards* que se queren probar e simulando a execución dos posibles eventos que se poden levar a cabo sobre eles, verificando que se mostran os elementos correctos na posición que lles corresponde, os diálogos de erro necesarios, etc. No caso das probas implementadas para este proxecto, execútanse todos os pasos que se poden levar a cabo no *wizard* OpenAPI dende que se abre ata que se crean novas vistas.

Estas probas automatizáronse con TestNG de igual xeito que as de integración, pero non foi necesario crear un *data provier* nin un *createMetadata*. Só se implementou un método *@BeforeClass* que abre a interface *Administration Tool*, un método *@BeforeMethod* que abre o *wizard* OpenAPI no seu estado inicial ao comezo de cada proba e un último método *AfterMethod* que pecha o *wizard* despois de cada proba.

10.4 Probas de aceptación

As probas de aceptación realizáronse unha vez se finalizou a implementación e comproban que o produto final cumpre cos requisitos establecidos ao comezo do proxecto. Para probar a funcionalidade OpenAPI executáronse as probas sobre seu *wizard*, e para probar o *wrapper* GraphQL fíxose uso do *wizard* que ofrece a Plataforma Denodo para crear *data sources* a partir de *custom wrappers*. Cada Caso de Uso descrito no capítulo 7.3 probouse manualmente nas condicións pre-establecidas e verificouse que realizasen o fluxo correcto.

Capítulo 11

Conclusións e futuras liñas de traballo

11.1 Conclusións

Ao longo de todo o desenvolvemento mantivéronse presentes os obxectivos principais definidos ao comezo do proxecto. No que respecta aos servizos REST, a creación e configuración manual de cada unha das súas peticións xa pode realizarse de forma automatizada a través da especificación OpenAPI que define dita API REST. O tempo e a complexidade desta tarefa redúcese considerablemente para os desenvolvedores de modelos de datos grazas á nova funcionalidade implementada. A integración desta funcionalidade na aplicación da empresa está prevista para un das seguintes actualizacións que se publicarán ao longo deste ano, e xa foi presentada aos clientes en reunións e *meetings* oficiais sobre a Plataforma Denodo.

En canto a GraphQL, como resultado do proxecto engadiuse un novo tipo de repositorio de datos ao catálogo soportado pola Plataforma. Os clientes que utilicen esta fonte de información poderán comezar a integrala en Denodo xunto cos demais tipos. Por outra banda, evítase que os clientes invirtan tempo e recursos en utilizar as librerías de Denodo para desenvolver o seu propio *wrapper* GraphQL.

Polo tanto, podemos concluír que os requisitos iniciais do proxecto foron conseguidos satisfactoriamente. Ademáis, durante o desenvolvemento aplicáronse tamén os principios de deseño establecidos para o proxecto, conseguindo así un software modular e independente entre sí que simplificará a fase de mantemento do seu ciclo de vida e permitirá engadir funcionalidades de forma máis sinxela.

11.2 Futuras liñas de traballo

Seguindo o modelo da empresa, o software deseñouse pensando en conseguir a maior independencia posible entre os seus compoñentes para que nun futuro sexa máis sinxelo engadir novas funcionalidades que poidan solicitar os clientes. Algunhas das posibles melloras que se consideraron son as seguintes:

- Engadir soporte para diferentes tipos de autenticación cando se introduce o URL da especificación OpenAPI do servizo REST. Este caso é necesario no momento de aportar o URL dende a interface, non se necesita na execución dos procedementos almacenados, polo que para a implementación do proxecto asumiuse que non se necesitaba acceso con autenticación ou que se dispoñía do documento OpenAPI en local.
- Actualmente é obrigatorio crear un *data source*, *wrapper* e vista base para cada URL da especificación. Sería máis util e práctico permitir que un só *data source* poida soportar a creación de múltiples *wrappers* e vistas base, o cal ofrecería, entre outras vantaxes, definir toda a configuración de acceso a unha API (credenciais, paxinación, etc) nun único *data source* en lugar de varios.
- Convertir o prototipo *custom wrapper* que permite o acceso a GraphQL nun novo tipo fonte de datos da Plataforma, do mesmo xeito que as JDBC, JSON, arquivos planos, etc. Deste xeito, quedará máis integrado na Plataforma e poderánse desenvolver *wizards* e interfaces deseñadas exclusivamente para a súa configuración e creación de *data sources* e vistas base.

Apéndices

Apéndice A

Material adicional

Exemplo de capítulo con formato de apéndice, onde se pode incluír material adicional que non teña cabida no corpo principal do documento, suxeito á limitación de 80 páxinas establecida no regulamento de TFGs.

A.1 Instalación do software

É necesario ter instalada a Platforma Denodo 8.0 para probar o software implementado neste proxecto. Na páxina oficial de Denodo poden descargarse os executables da Plataforma para diferentes versións, neste caso debemos descargar o instalador de 8.0 e executalo no noso equipo. É obrigatorio dispoñer dunha licencia para poder utilizar as funcións da Plataforma.

Unha vez estea instalada a Denodo Platform 8.0, debemos descargar a actualización na que se atopen estas melloras, a cal pode descargarse dende mesma páxina que o instalador. Despois de actualizar a Plataforma correctamente, xa se poderán utilizar as novas funcionalidades.

A.2 Manual de usuario

Neste apartado explicarase paso a paso como pode o usuario facer uso as novas funcionalidades desenvolvidas dende a interface de escritorio Admnistration Tool. Ilustrarase con imaxes da propia interface como abrir os *wizards* correspondentes a cada funcionalidade, introducir os parámetros necesarios e executar consultas contra os repositorios de datos.

A.2.1 Crear vista base a partir dunha especificación OpenAPI

En primeiro lugar, detallarase como crear vistas base coa funcionalidade OpenAPI e como utilizalas para executar peticións contra o servizo REST. Para isto necesitaremos comezar abrindo o *wizard* que se implementou para o *frontend* desta mellora, o cal se atopa no menú

Tools da *Administration Tool* como se ve na imaxe A.1. Ao premer na opción *Create views from OpenAPI* mostraráse o mencionado *wizard* da imaxe A.2.

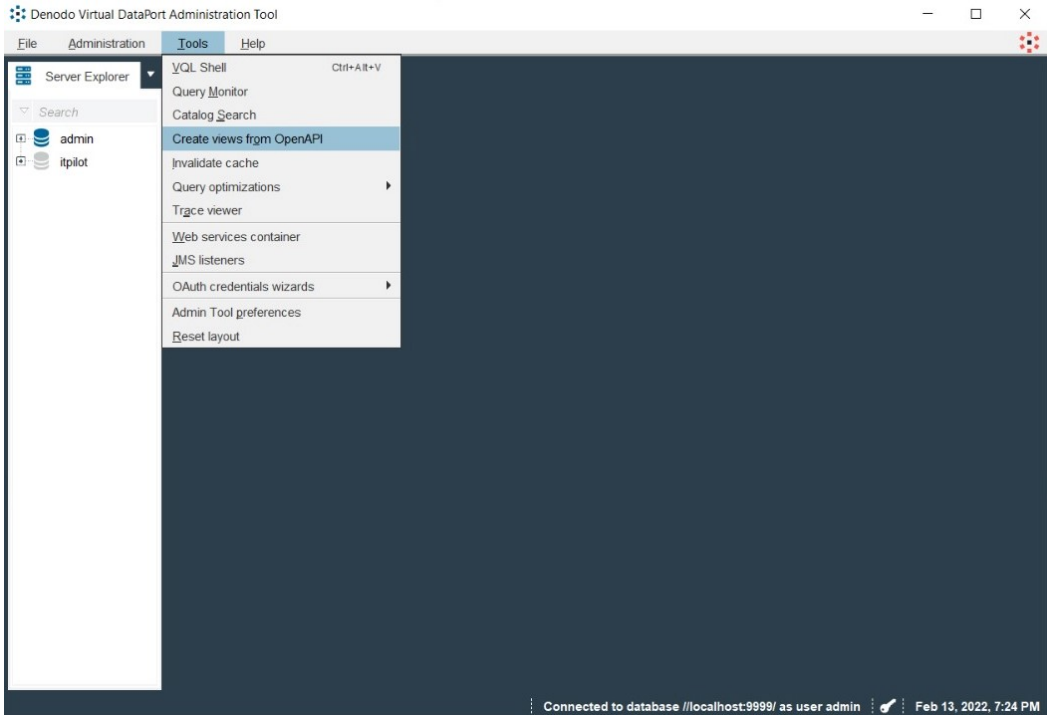

Figura A.1: Menu da interface dende o que se accede ao *wizard* OpenAPI

Co *wizard* aberto o usuario pode escoller entre proporcionar a especificación a través un URL (figura A.3) ou utilizando un ficheiro seleccionado do equipo local (figura A.4). Unha vez introducida a especificación premeremos no botón *Explore OpenAPI document* para que a interface procese o documento OpenAPI e nos mostre a lista de peticións atopadas e o panel de configuración para crear as súas vistas, tal como se ve na figura A.5.

Na especificación que acabamos de procesar había tres recursos: *pet, store* e *user*, os cales podemos desplegar na árbore para ver todas as súas operacións que se atoparon xunto co botón que nos permite crear as súas vistas base. Una vez escollida a configuración que queremos aplicar á nosa vista, seleccionaremos unha operación e premeremos o seu botón *Create base view* para que a interface cree a vista e abra o seu *wizard* e o do seu *data source*.

Dende a interface da vista base que acabamos de crear podemos acceder ao panel de execución e engadir certo número de parámetros que se incluiran na consulta que se lance contra o servizo REST. Estos parámetros engádense a través do botón *Conditions+* e a consulta pode executarse premendo o botón *Execute*. Nas seguintes figuras mostrase o proceso de crear unha vista para unha petición POST /pet, executala dende a *Administration Tool* para crear unha nova mascota e posteriormente consultar esa mascota cunha vista GET /pet/petId creada

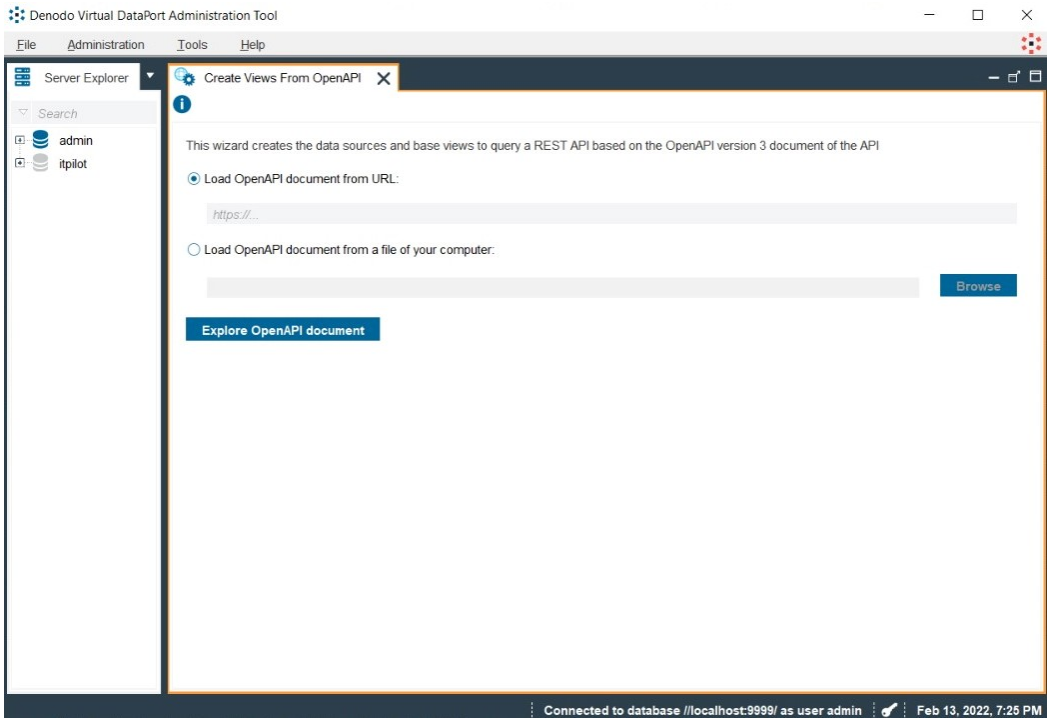

Figura A.2: *Wizard* OpenAPI

tamén a partir da funcionalidade OpenAPI.

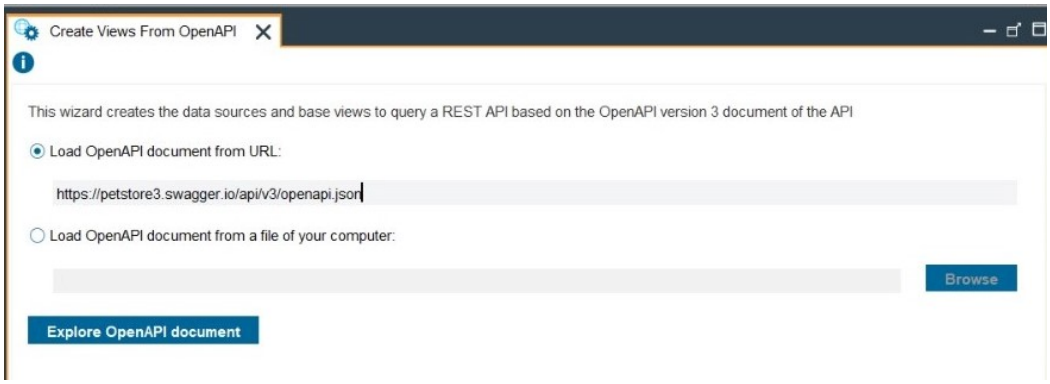

Figura A.3: Exemplo de proporcionar a especificación mediante o URL

Figura A.4: Exemplo de proporcionar a especificación mediante un ficheiro local

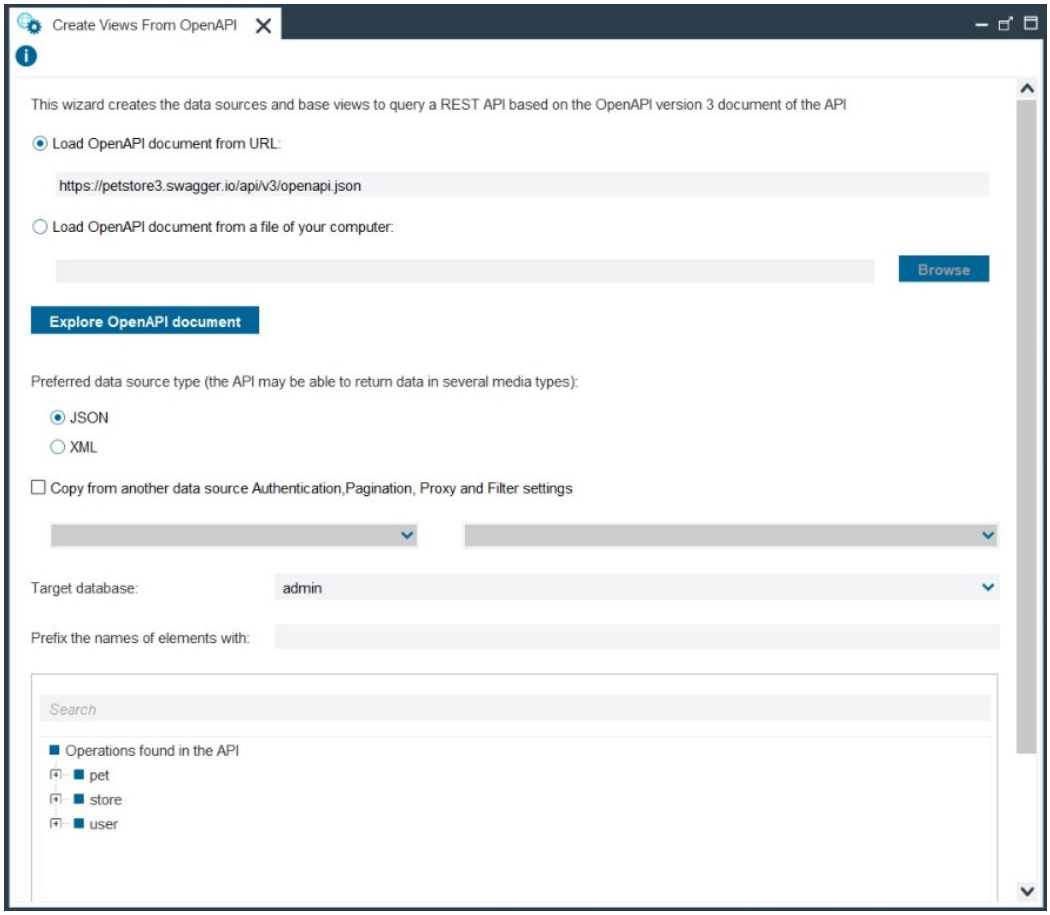

Figura A.5: Resultado de procesar unha especificación

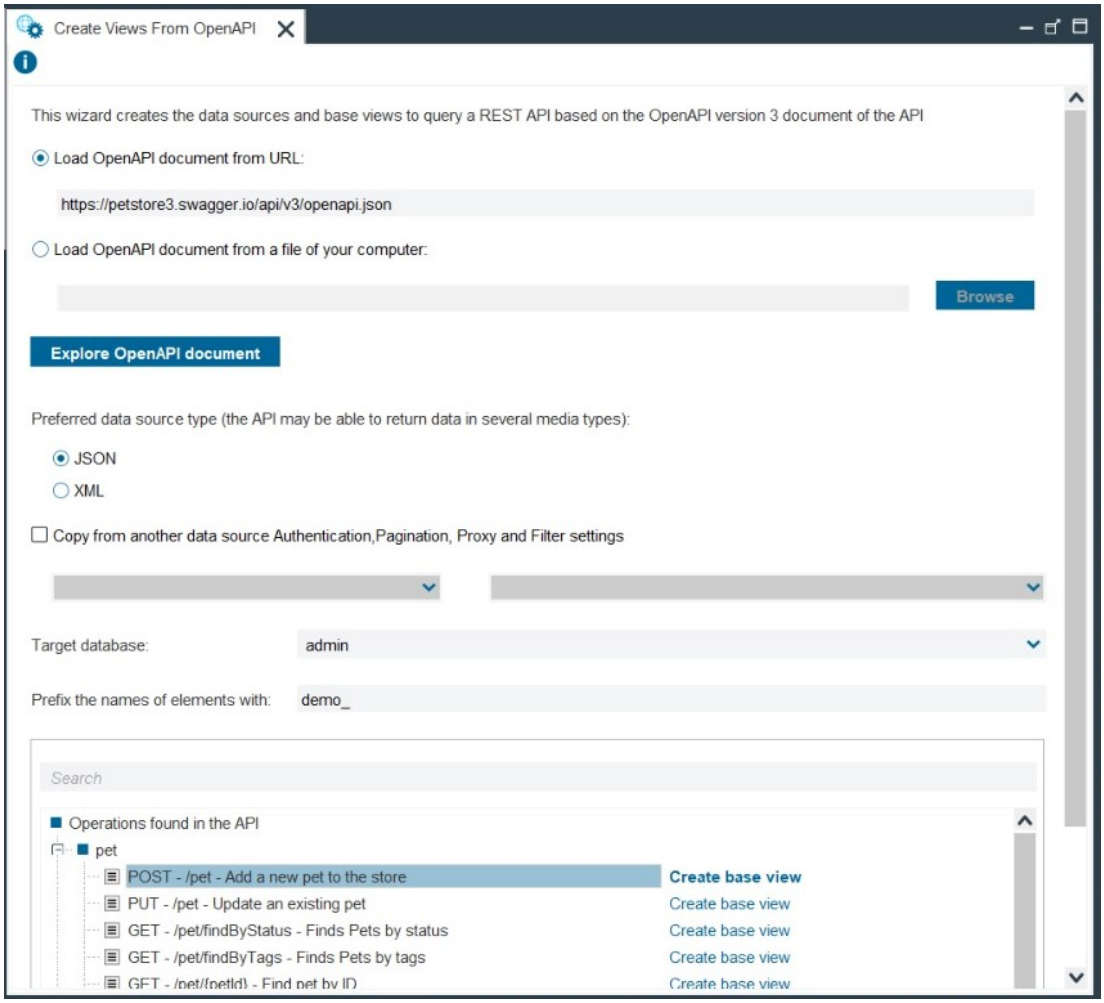

Figura A.6: Creación da vista base para unha petición POST /pet utilizando o prefixo *demo_*

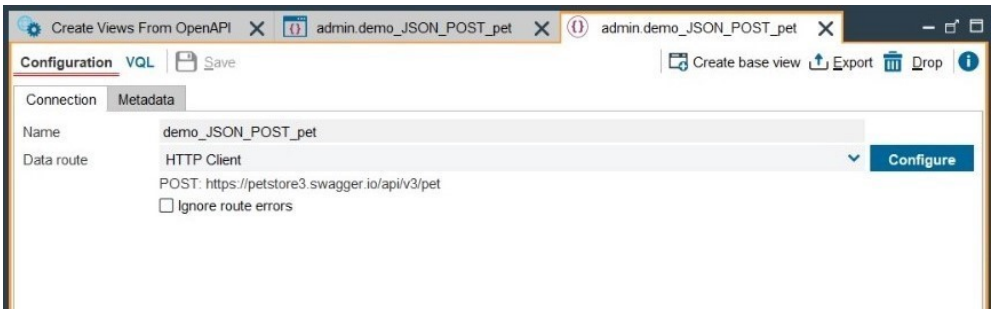

Figura A.7: *Data source* creado pola funcionalidade que representa a petición POST /pet

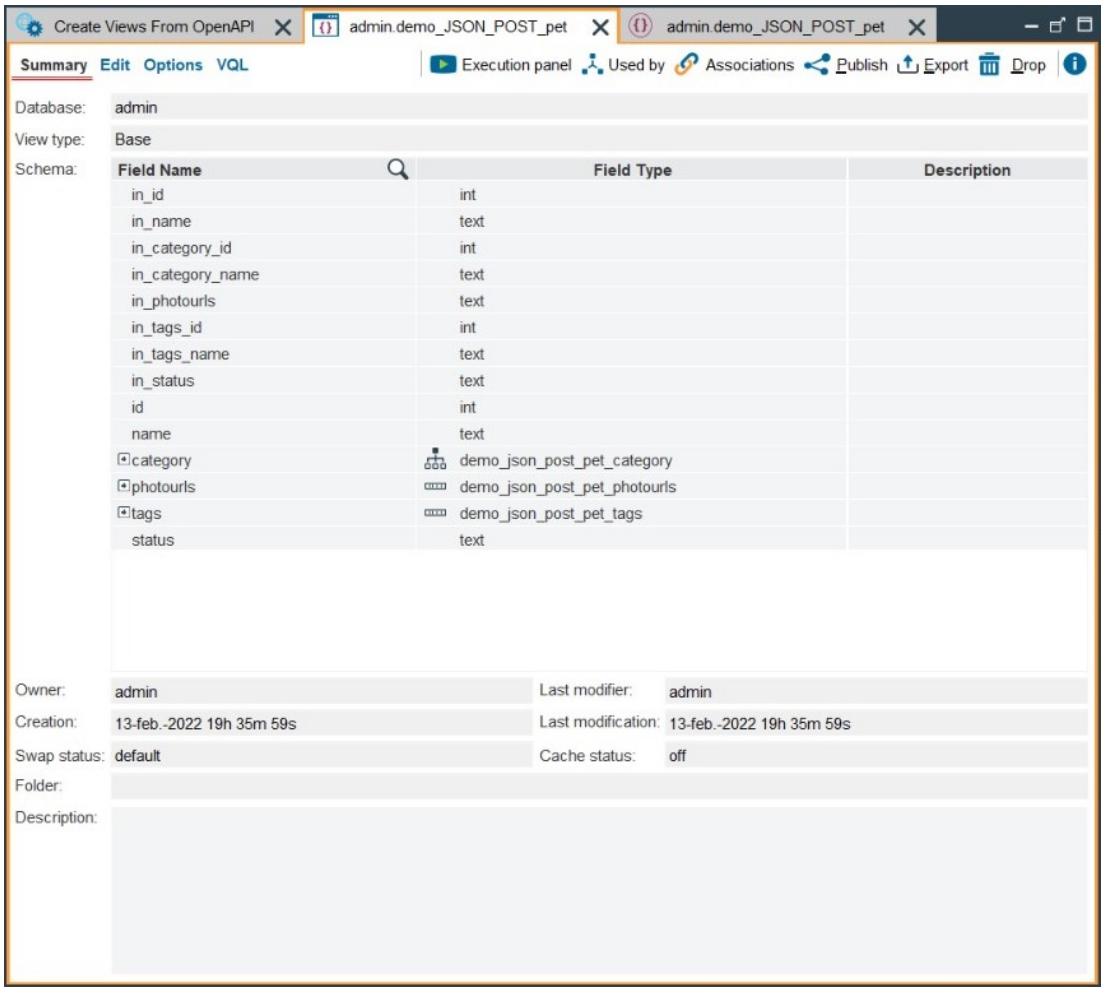

Figura A.8: Vista base creada pola funcionalidade que representa a petición POST /pet

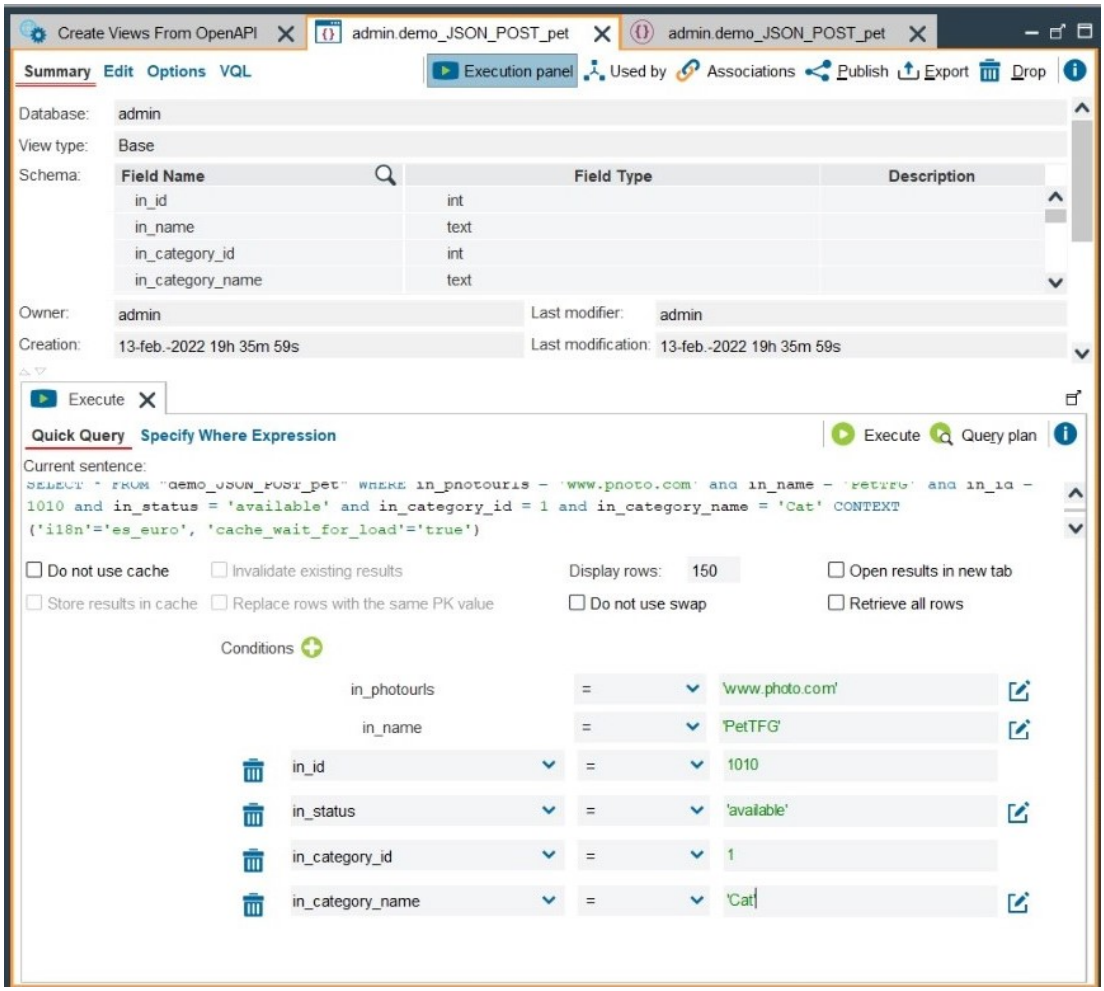

Figura A.9: Panel de execución para executar unha constulta contra o servizo REST utilizando a vista base da petición POST /pet

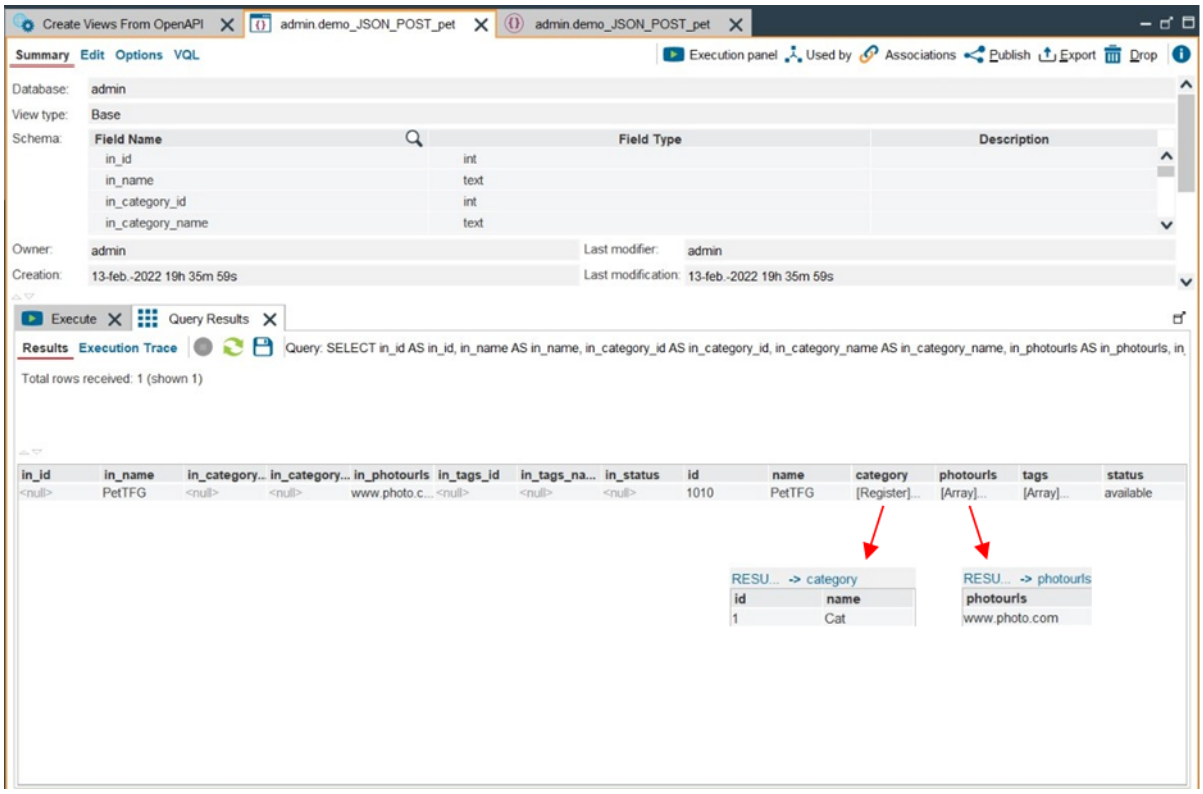

Figura A.10: Resultado da execución da consulta anterior

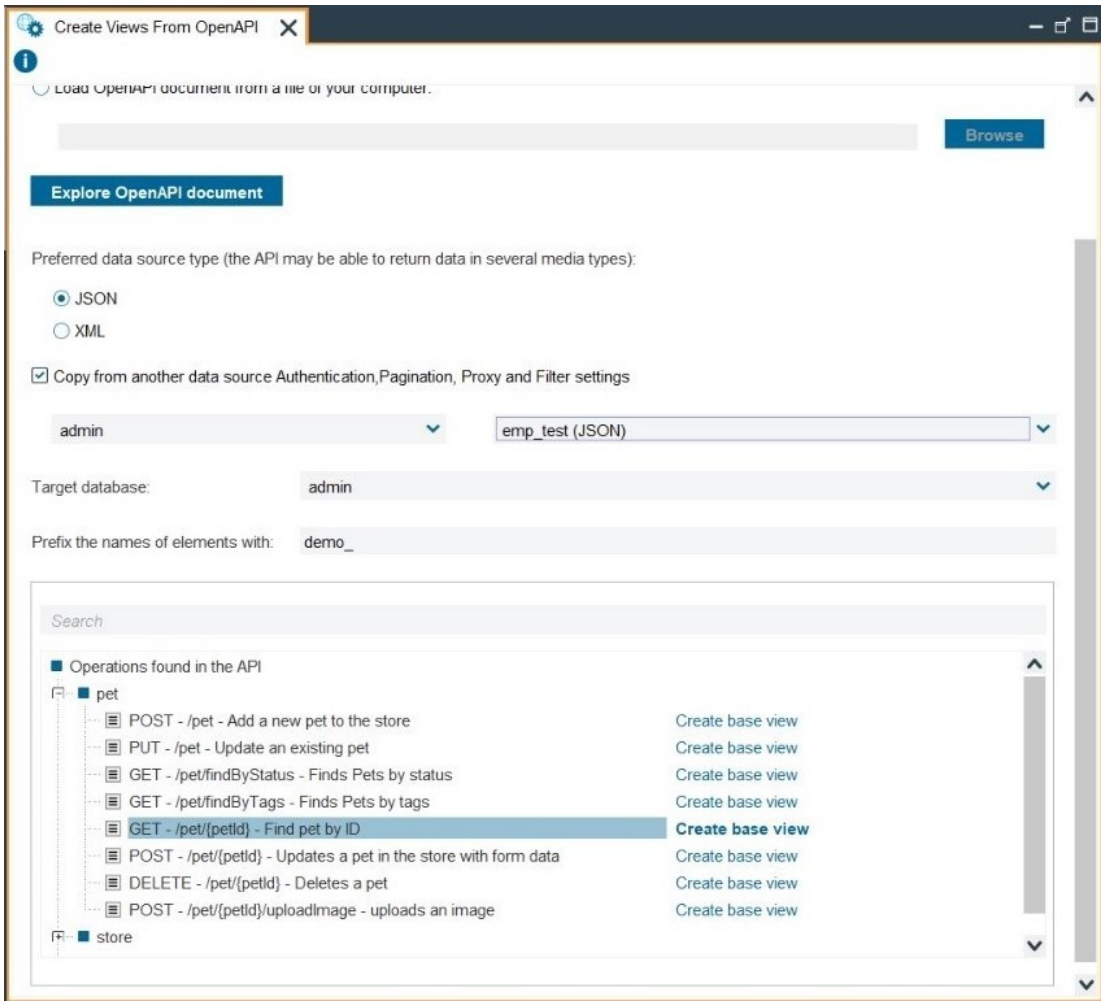

Figura A.11: Creación da vista base para unha petición GET /pet/petId utilizando o prefixo *demo_* e copiando a configuración do *data source emp_test*

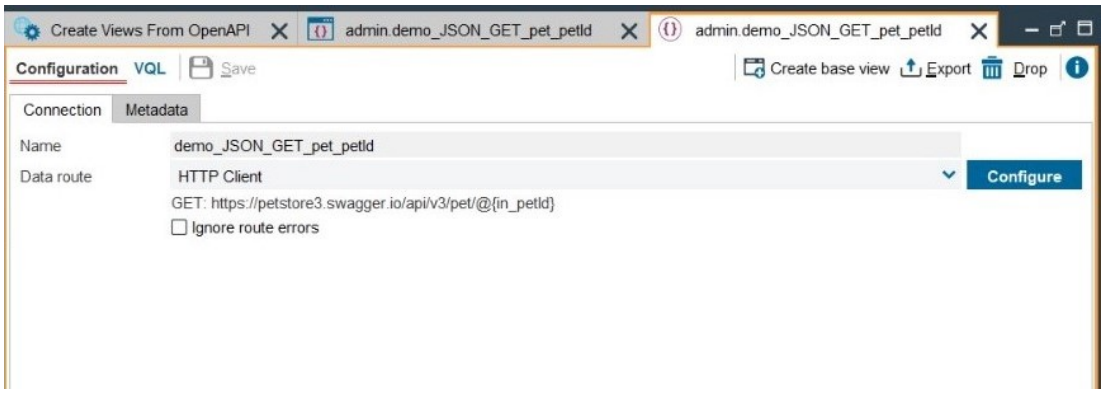

Figura A.12: *Data source* creado pola funcionalidade que representa a petición GET /pet/petId

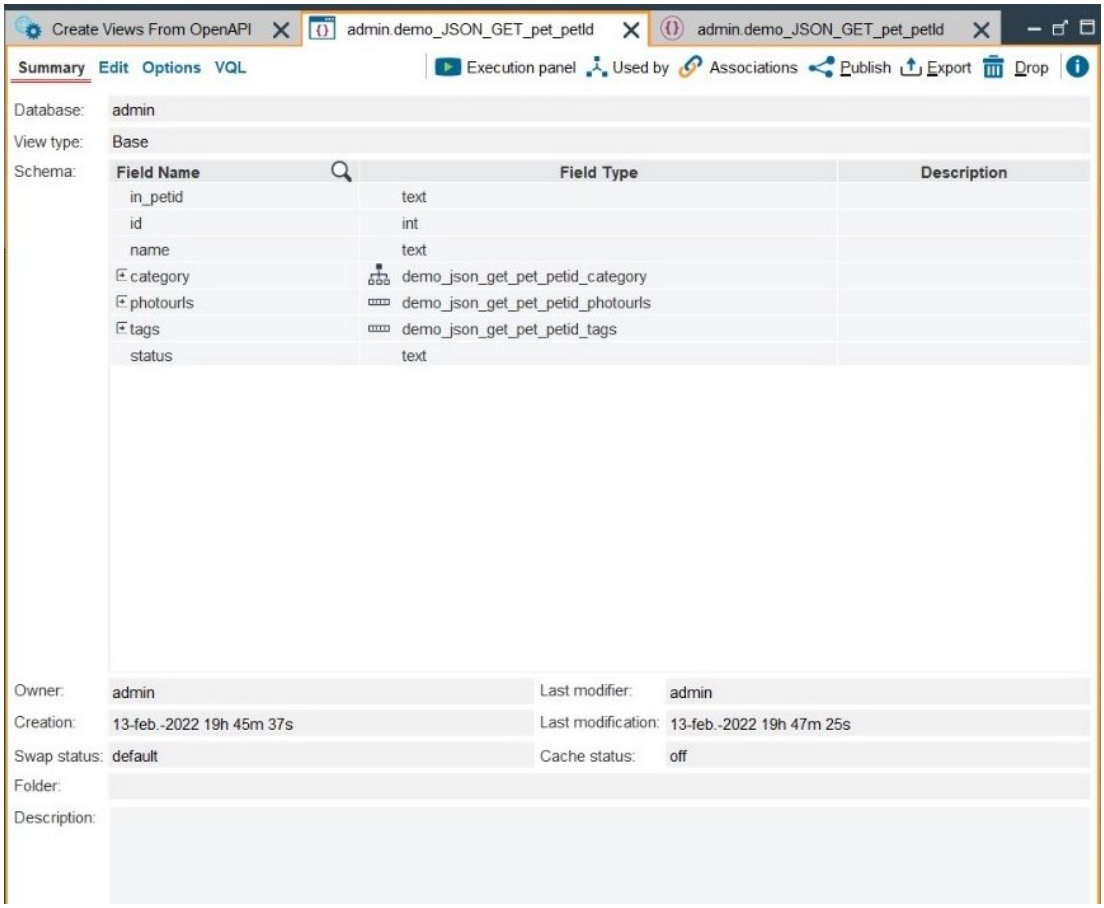

Figura A.13: Vista base creada pola funcionalidade que representa a petición GET /pet/petId

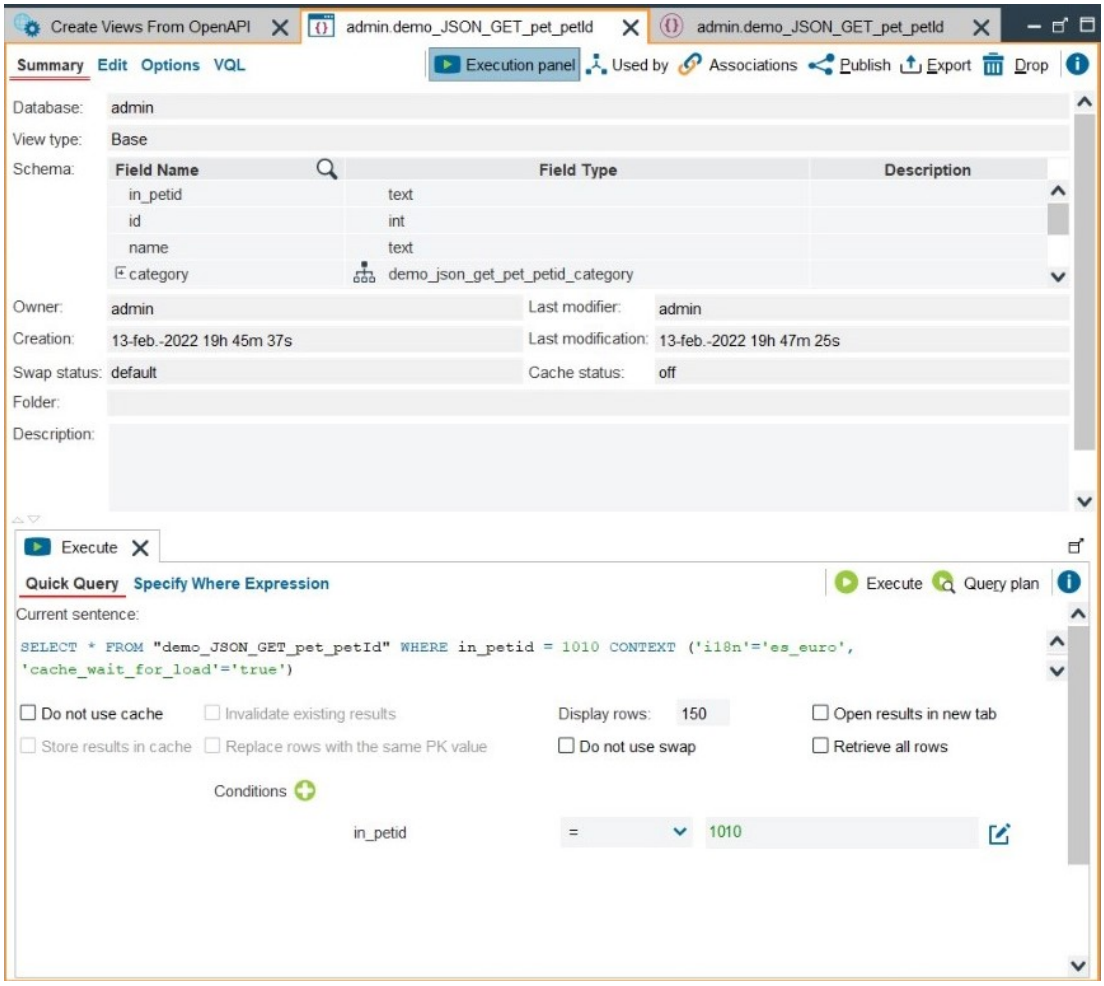

Figura A.14: Panel de execución para executar unha constulta contra o servizo REST utilizando a vista base da petición GET /pet/petId

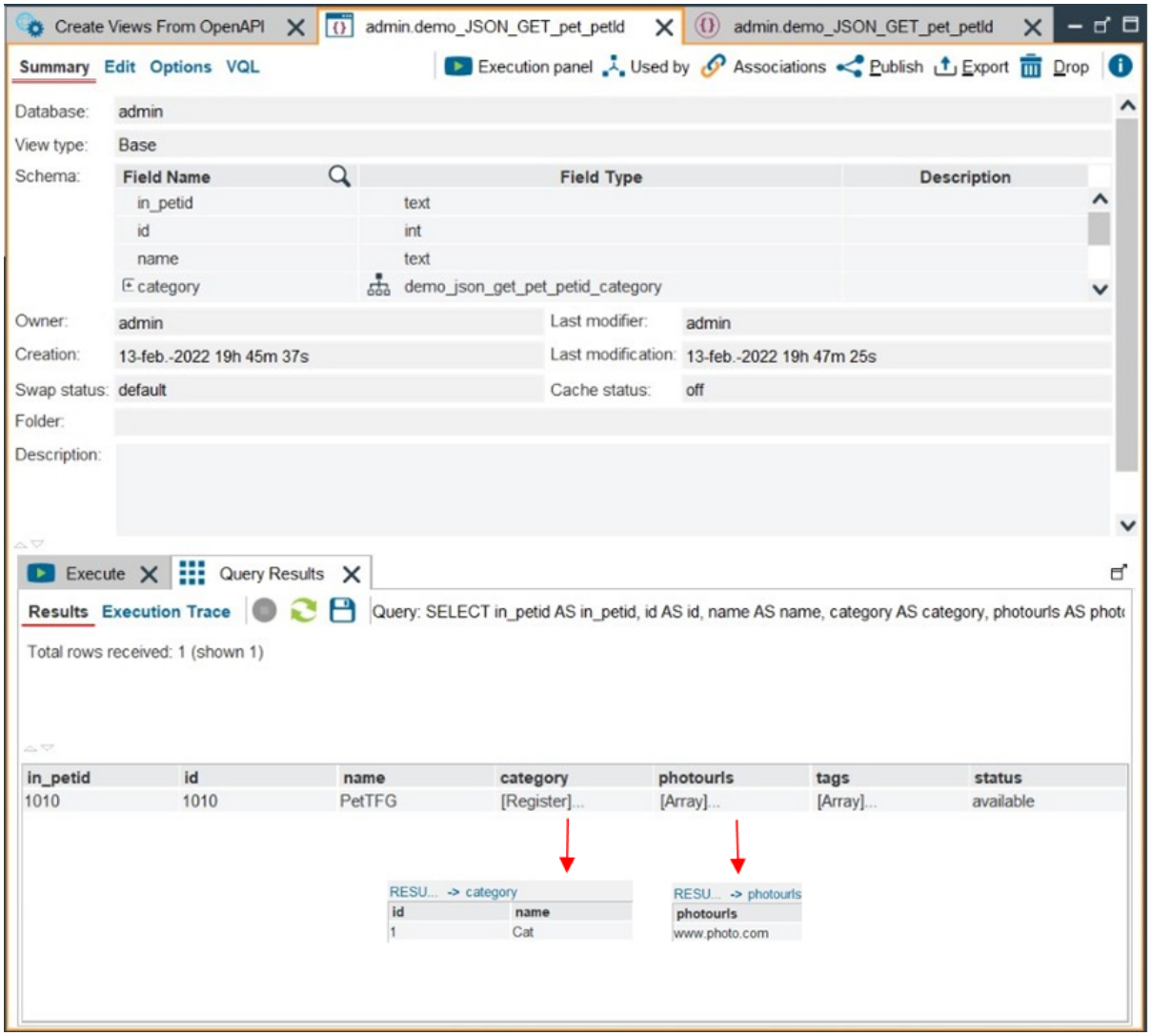

Figura A.15: Resultado da execución da consulta anterior

A.2.2 Executar consultas contra un servizo GraphQL

En segundo lugar, ilustrarase como executar consultas utilizando o *wrapper* GraphQL que se desenvolveu. Como se mencionou anteriormente, é necesario crear un *custom data source* dende o menú *File* > *New* > *Data source* > *Custom* (figura A.16). A continuación, debemos configurar o *data source* cos parámetros necesarios para que utilice o *custom wrapper* implementado. Primeiro, como se ve na imaxe A.17 debemos proporcionar un nome ao *data source* e, no campo *Class name*, a ruta onde se atopa a clase Java que implementa o *wrapper*, neste caso *com.denodo.vdb.contrib.customwrapper.GraphQLWrapper*.

Depois de introducir estes parámetros, debemos premer no botón verde que aparece despois da etiqueta *Click to refresh the input parameters of the data source*. Deste xeito, a interface

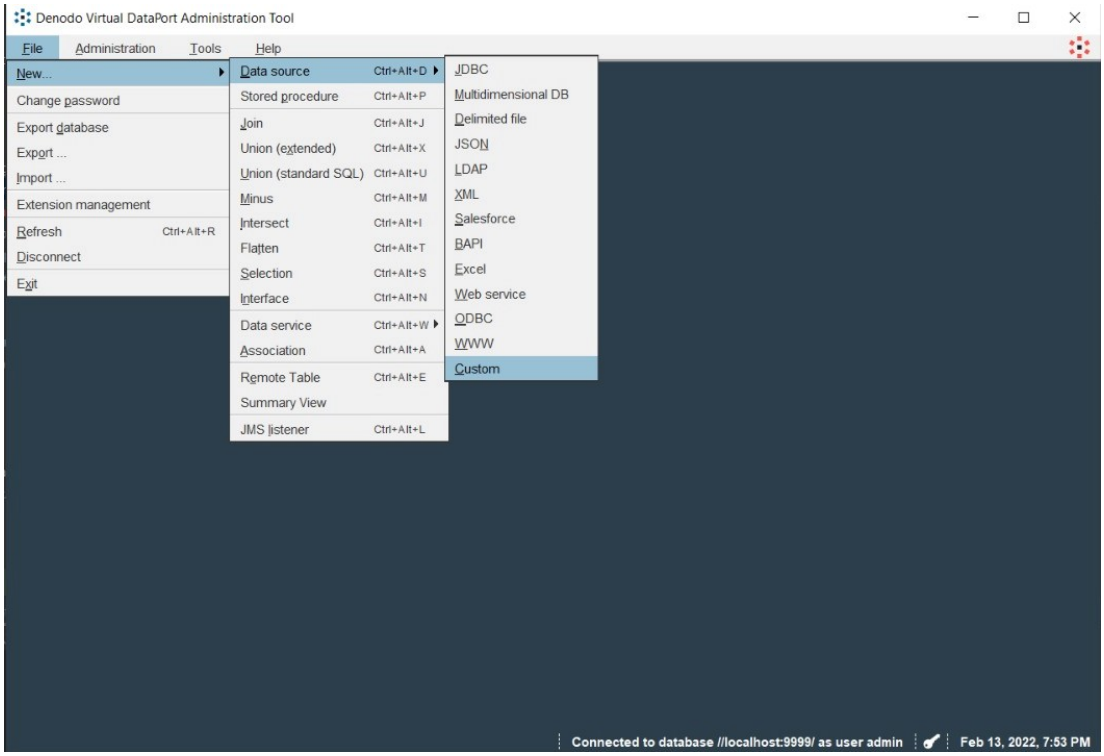

Figura A.16: Menú dende o que se abre o *wizard* para crear un *data source* que utilice o *wrapper* GraphQL

accederá ao *wrapper* (facendo uso do *Class name*) e obterá os seus parámetros de entrada. No subcapítulo 8.4 Subsistema backend, cando se detallou o deseño deste *wrapper* indicouse que contaba con dous parámetros de entrada para poder funcionar correctamente: a localización do esquema GraphQL e o URL do servizo contra o que se lanzarán as consultas (para este exemplo utilizouse un pequeno servizo GraphQL desplegado en local). Como consecuencia, ao premer o botón que refresca os parámetros de entrada aparecerán novos campos nos que se nos requerirán estes dous valores, como se ve na figura A.18. Despois de introducir as direccións do esquema e o servizo, xa poderemos crear vistas base sobre este *data source* para executar consultas GraphQL.

Para crear a vista base sobre o *data source* premeremos no botón da esquina superior dereita *Create base view*. A continuación un diálogo solicitarános introducir a operación de GraphQL para a cal queremos crear a vista (figura A.19), pero non se indicarán os seus parámetros de entrada, só o nome da operación. Para que a vista se cree correctamente, esta petición debe estar soportada polo servizo GraphQL, é dicir, no seu esquema de datos, dentro do obxecto *Query* debe existir unha petición con este nome.

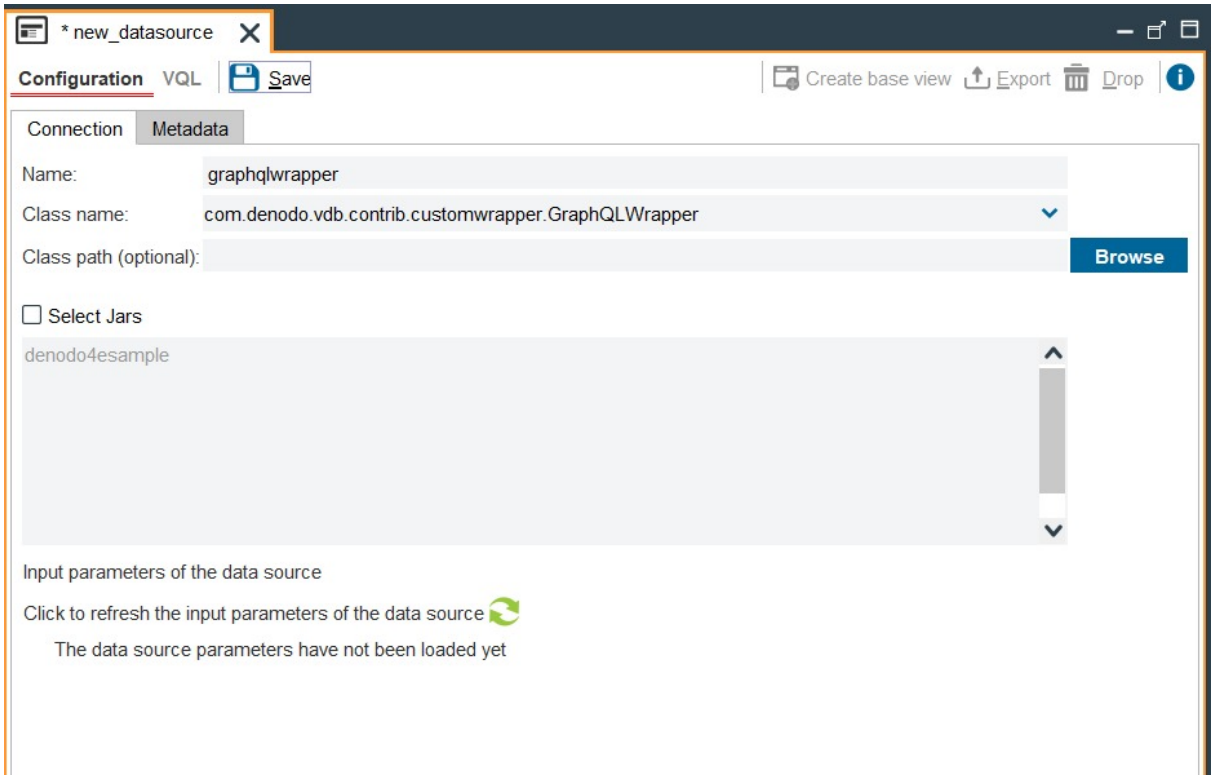

Figura A.17: Configuración do *custom data source* que utilizará o *wrapper* GraphQL

Despois de introducir o nome da operación e premer en *Ok*, abrirásenos o *wizard* da vista que acabamos de crear, tal como se ve na figura A.20. A diferencia das vistas JSON creadas na funcionalidade anterior, as peticións deben executarse dende a VQL Shell mediante comandos relacionais. Na figura A.21 móstrase un exemplo de como executar estas consultas. Na imaxe, queremos lanzar unha consulta que solicita á vista *bookById* obter o libro que teña como identificador *'book-1'*, mostrando na resposta os seguintes campos: identificador do libro, nome do libro, identificador do autor e apelido do autor.

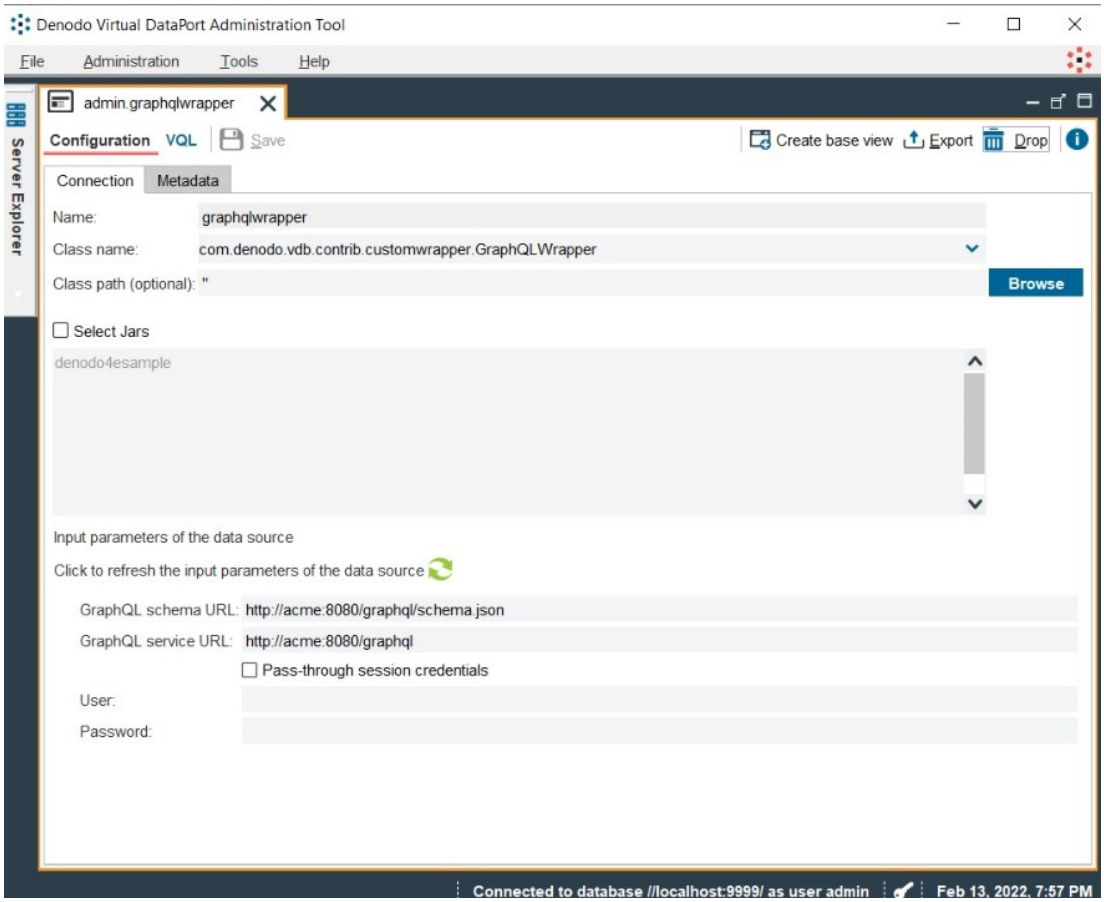

Figura A.18: Configuración do *custom data source* que utilizará o *wrapper* GraphQL

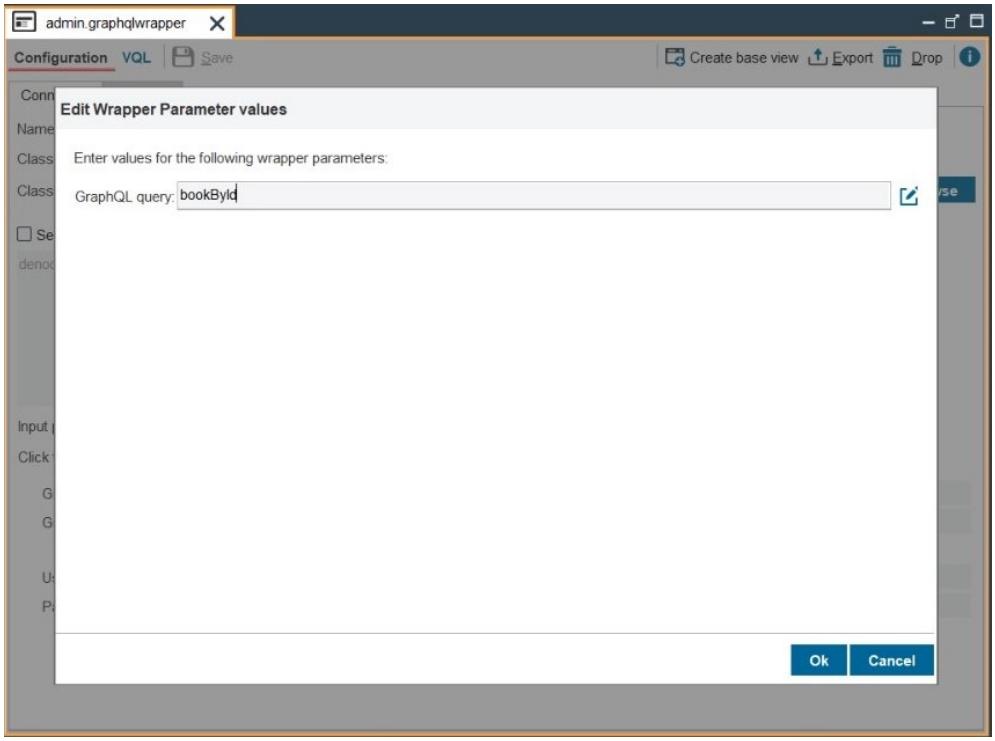

Figura A.19: Creación dunha vista base para a consulta GraphQL *bookById*

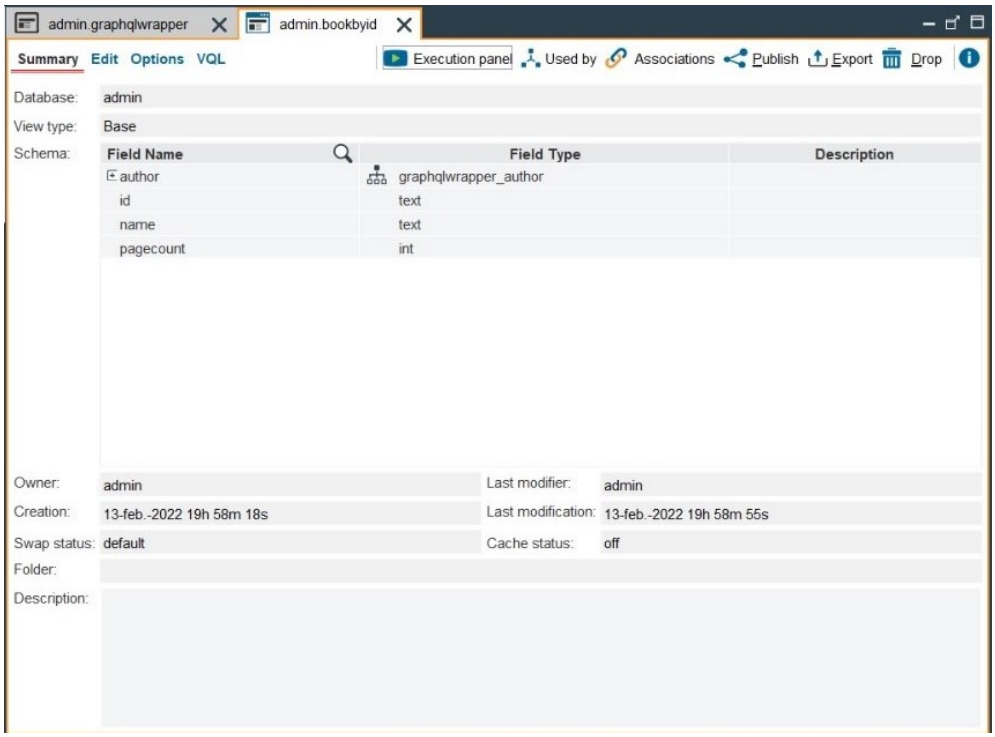

Figura A.20: Vista base da consulta GraphQL *bookById*

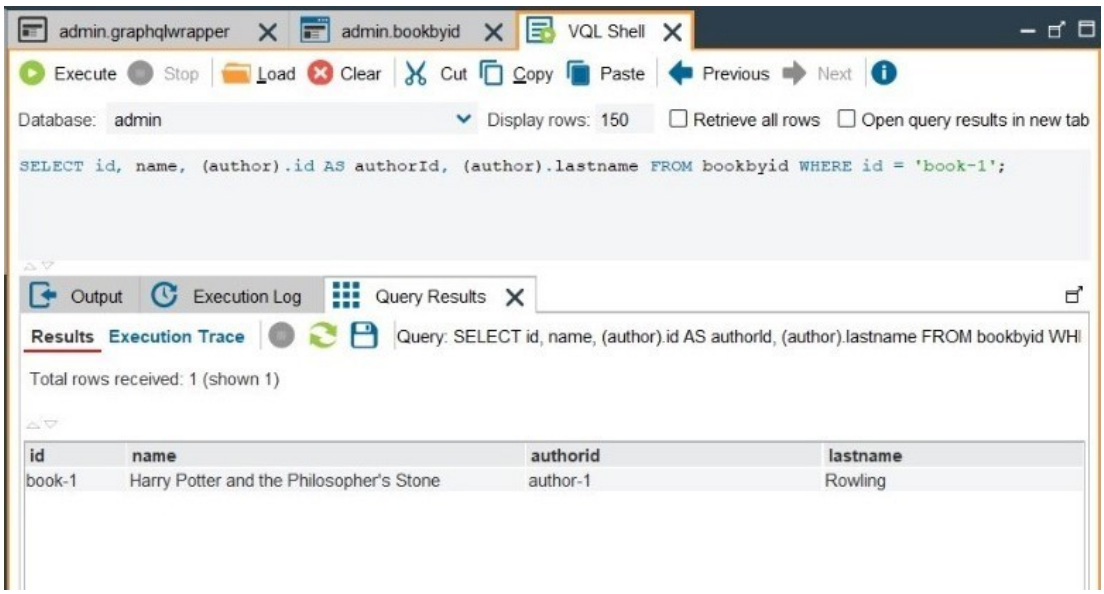

Figura A.21: Execución e resultado dunha consulta utilizando a vista base da petición *bookById*

Relación de Acrónimos

IDE Integrated Development Enviroment. 9

JDBC Java Database Connectivity. 2

JSON JavaScript Object Notation. 1

ODBC Open Database Connectivity. 2

REST Representational state transfer. 1

SOAP Simple Object Access Protocol. 1

XML Extensible Markup Language. 1

Glosario

Backend Subsistema dun software constituído pola parte servidor da aplicación. 20

Framework Estrutura de soporte tecnolóxico e conceptual definida mediante módulos de software concretos que se utiliza como base para organizar e desenvolver software. 8

Frontend Subsistema dun software constituído pola parte cliente da aplicación. 20

Bibliografía

[1] Swagger, "Openapi specification." [En liña]. Dispoñible en: https://swagger.io/specification/

[2] Oracle, "Conozca más sobre la tecnología java." [En liña]. Dispoñible en: https://www.java.com/es/about/

[3] "Introducing json." [En liña]. Dispoñible en: https://www.json.org/json-en.html

[4] "Extensible markup language (xml) 1.0 (fifth edition)." [En liña]. Dispoñible en: https://www.w3.org/TR/xml/

[5] "Yet another markup language (yaml) 1.0." [En liña]. Dispoñible en: https://yaml.org/spec/history/2001-12-10.html

[6] "Graphql." [En liña]. Dispoñible en: https://graphql.org/

[7] "Introduction to sql." [En liña]. Dispoñible en: https://www.w3schools.com/sql/sql_intro.asp

[8] "Language for defining and processing data: Vql." [En liña]. Dispoñible en: https://community.denodo.com/docs/html/browse/6.0/vdp/vql/language_for_defining_and_processing_data_vql/language_for_defining_and_processing_data_vql

[9] SmartBear. [En liña]. Dispoñible en: https://swagger.io/specification/

[10] "Latex – a document preparation system." [En liña]. Dispoñible en: https://www.latex-project.org/

[11] [En liña]. Dispoñible en: https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Getting-started

[12] [En liña]. Dispoñible en: https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

[13] "Graphql java." [En liña]. Dispoñible en: https://github.com/graphql-java/graphql-java/

[14] Oracle, "Package javax.swing." [En liña]. Dispoñible en: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html#package_description

[15] Apache, "Welcome to apache commons." [En liña]. Dispoñible en: https://commons.apache.org/

[16] ——, "Apache log4j 2." [En liña]. Dispoñible en: https://logging.apache.org/log4j/2.x/index.html

[17] TestNG, "Framework testng." [En liña]. Dispoñible en: https://testng.org/doc/index.html

[18] Mozilla, "Generalidades del protocolo http." [En liña]. Dispoñible en: https://developer.mozilla.org/es/docs/Web/HTTP/Overview#mensajes_http

[19] R. T. Fielding, "Architectural styles and the design of network-based software architectures," 2000.

[20] SonatypeCompany, "Maven: The definitive guide."

[21] JetBrains. [En liña]. Dispoñible en: https://www.jetbrains.com/idea/

[22] "Sonarlint." [En liña]. Dispoñible en: https://plugins.jetbrains.com/plugin/7973-sonarlint

[23] Postman, "About postman." [En liña]. Dispoñible en: https://www.postman.com/company/about-postman/

[24] Jenkins, "Jenkins user documentation." [En liña]. Dispoñible en: https://www.jenkins.io/doc/

[25] Git. [En liña]. Dispoñible en: https://git-scm.com/

[26] D. Technologies, "Launching the data catalog." [En liña]. Dispoñible en: https://community.denodo.com/tutorials/browse/datadiscovery/1datacatalogintro

[27] ——, "First steps - administration tool." [En liña]. Dispoñible en: https://community.denodo.com/tutorials/browse/basics/2fs2admintool

[28] ——, "First steps - administration tool." [En liña]. Dispoñible en: https://community.denodo.com/tutorials/browse/basics/2fs2admintool

[29] ——, "Desing studio." [En liña]. Dispoñible en: https://community.denodo.com/docs/html/browse/8.0/en/platform/new_features/new_features_virtual_dataport#design-studio

[30] SoapUI. [En liña]. Dispoñible en: https://www.soapui.org/

[31] APIMatic, "Api transformer." [En liña]. Dispoñible en: https://www.apimatic.io/transformer/

[32] "Openapi-to-graphql." [En liña]. Dispoñible en: https://developer.ibm.com/open/projects/openapi-to-graphql/

[33] Oracle, "Oracle data service integrator." [En liña]. Dispoñible en: https://www.oracle.com/middleware/technologies/data-service-integrator.html

[34] ——, "Fusion middleware developing integration projects with oracle data integrator - using web services." [En liña]. Dispoñible en: https://docs.oracle.com/middleware/122126/odi/develop/GUID-F193637E-388D-447C-9C8E-420BD135C093.htm#ODIDG-GUID-1F817640-A5EB-4FF4-A541-3A94834E30CA

[35] IBM, "Ibm cloud pak for data." [En liña]. Dispoñible en: https://www.ibm.com/es-es/products/cloud-pak-for-data

[36] ——, "Ibm cloud pak for data - available apis." [En liña]. Dispoñible en: https://www.ibm.com/support/producthub/icpdata/docs/content/SSQNUZ_latest/dev/avail-apis.html

[37] TIBCO, "Tibco data virtualization." [En liña]. Dispoñible en: https://www.tibco.com/products/data-virtualization

[38] ——, "Tibco® data virtualization and business directory - release notes." [En liña]. Dispoñible en: https://docs.tibco.com/pub/tdv/8.3.0/TIB_tdv_8.3.0_relnotes.pdf?id=1

[39] Swagger, "Swagger data models (schemas)." [En liña]. Dispoñible en: https://swagger.io/docs/specification/data-models/

[40] ——. [En liña]. Dispoñible en: https://petstore3.swagger.io/api/v3/openapi.json