DGTE-001: Desarrollo de servicios negocio y de acceso a datos aplicaciones Java EE

Versión 1.4

Área de Arquitectura

GERENCIA INFORMÁTICA
JOSEFA VALCÁRCEL, 44
28027-MADRID

Contenido

1 Introducción ... 11
1.1 Objetivo ... 11
1.2 Destinatarios del documento ... 12
1.3 Convenios de la especificación. ... 12
2 Visión general ... 13
2.1 Objetivos ... 13
2.2 Servicios de negocio ... 14
2.2.1 Naturaleza atómica de los servicios de negocio ... 14
2.2.2 Invocaciones asíncronas ... 15
2.2.2.1 Invocación en paralelo ... 16
2.2.2.2 Procesos de larga duración ... 16
2.3 Estructura de la capa de negocio (Servicios de negocio de aplicación y de dominio): Puerto de servicio ... 17
2.3.1 Reutilización del puerto de servicio. ... 17
2.3.2 Seguridad del servicio. ... 18
2.4 Exposición de los servicios de negocio a otras aplicaciones (adaptadores de servicios) ... 18
2.5 Entidades manejadas por el negocio ("dominio interno"). ... 18
2.6 Servicios de acceso a datos ... 19
2.6.1 Soporte a la migración de los backends de almacenamiento de datos ... 21
2.6.2 Soporte al paralelo de backends distintos durante la migración ... 22
2.7 Lógica transaccional ... 23
2.7.1 Recursos no transaccionales ... 24
2.8 Representación esquemática ... 25
2.8.1 Lógica de negocio vista por el cliente ... 25
2.8.2 Lógica de negocio dentro de las aplicaciones ... 26
2.8.3 Visión de conjunto ... 26
2.8.4 Entidades de negocio ... 27
2.8.4.1 Un ejemplo: la Java™ Persistence API ... 29
3 Servicios de Negocio ... 33
3.1 Introducción ... 33
3.2 Estado conversacional de los servicios de negocio ... 35
3.3 Interfaz de negocio ... 36
3.3.1 Requerimientos de los métodos de negocio ... 36
3.3.2 Requerimientos del interfaz de negocio ... 36
3.3.3 Interfaces locales y remotas ... 37
3.4 Inyección de dependencias ... 37
3.4.1 Ejemplo: inyección de un servicio de negocio ... 38
4 Patrones para el diseño y la implementación de servicios de negocio ... 41
4.1 Acoplamiento entre aplicaciones e inestabilidad de los contratos ... 41
4.2 Resolución de las interfaces de negocio. Granularidad de las interfaces (servicios de negocio de aplicación y de dominio). ... 41
4.3 Cantidad de datos devuelta por método ... 42
4.3.1 Partición de la información ... 43
4.3.1.1 Ejemplo ... 44
4.3.2 Devolución de jerarquía de entidades ... 44

4.3.3 Paginación 45
4.4 Manejo de múltiples backends 46
4.5 Invocación de servicios externos a través de adaptadores de servicio de salida 47
5 Control transaccional de los servicios de negocio 49
5.1 Recursos no transaccionales 49
5.1.1 Ordenación y posicionamiento 49
5.2 Modelo declarativo 50
5.2.1 Atributos de control de transacciones 50
5.2.2 Utilización del atributo de control de transacción 51
5.3 Configuración del contenedor EJB v. 3.0 51
5.4 Marcar una transacción para el *rollback* 52
5.5 Gestión de las excepciones 53
5.6 Servicios de negocio asíncronos 54
5.6.1 Servicios intrínsecamente asíncronos 54
6 Servicios de acceso a datos 57
6.1 Principios fundamentales 57
6.2 Características de un servicio de acceso a datos 59
6.3 Interfaz de acceso a datos 60
6.3.1 Requerimientos de los métodos de acceso a datos 61
6.3.2 Requerimientos del interfaz de acceso a datos 61
6.3.3 Ejemplo: estructura básica de un servicio de acceso a datos 62
7 Patrones para el diseño y la implementación de servicios de acceso a datos 65
7.1 Soporte a múltiples backends 65
7.1.1 Soporte a la migración de los backends de almacenamiento de datos 67
7.1.2 Configuración del backend en tiempo de despliegue 69
7.1.3 Soporte al paralelo de backends distintos durante la migración 70
7.1.4 Directrices de diseño para la implementación de paralelo de backends 70
7.1.4.1 Ejemplo de acceso concurrente a múltiples backend en escritura 71
8 Control transaccional en los servicios de acceso a datos 75
8.1 Control transaccional declarativo 75
8.2 Manejo de recursos transaccionales 75
8.2.1 Contextos de persistencia 76
8.2.2 Referencias a contextos de persistencia 77
8.3 Manejo de recursos no transaccionales 78
8.3.1 Estado conversacional 79
8.3.2 Stateful beans 79
8.3.3 Recursos no transaccionales utilizados por el bean 80
8.3.4 EJBCommandSupport 80
8.3.5 Invocación de comandos 81
8.3.6 Métodos de negocio 81
8.3.7 Terminación de una transacción 82
8.4 Paginación 82
9 Escenarios de integración de servicios 84
9.1 Integración entre subsistemas de Internet e Intranet 84
9.2 Publicación de servicios en Intranet 84
9.3 Publicación de servicios Web en Internet 85
9.4 Librerías reutilizables 85

10 Estructura de proyecto 86
10.1 Visión general 86
10.1.1 Conceptos básicos de integración de aplicaciones 86
10.1.2 Elementos arquitectónicos de integración (adaptadores y puertos de servicio). 87
10.2 Estructura de proyecto 87
10.2.1 Adaptadores 87
10.2.1.1 Adaptadores de servicios web (SOAP) 87
10.2.1.2 Adaptadores REST 90
10.2.1.3 Adaptadores de presentación. 93
10.2.2 Puertos de servicio 93
10.2.3 Implementación de lógica de negocio 93
10.2.4 Capa de persistencia y Servicios de Acceso a Datos 94
10.2.5 Capa de adaptadores de Salida. 94
10.2.5.1 Cliente de servicio web 94
10.2.5.2 Declaración de puntos de servicio a través de JNDI 94
10.3 Criterio de seguridad para el acceso a Servicios de Negocio 95
11 Apéndice A: Implementación de ejemplo 97
11.1 Requisitos 97
11.2 Identificación de los servicios de negocio 97
11.3 Definición de los servicios de negocio 99
11.3.1 Alta de cita 100
11.3.2 Consulta de personas 100
11.3.3 Alta de personas 100
11.3.4 Pago de tasas 101
11.3.5 Creación y grabación de PDF 101
11.4 Implementación de los servicios de negocio 101
11.5 Estructura del componente EJB Alta de cita 101
11.6 Inyección de otros componentes 102
11.7 Lógica de negocio 103
11.8 Consulta de tramitación 104
11.8.1 Comando para la consulta host 105
11.9 Remoción de componentes 106
11.10 Atributos de control de transacción 106
11.11 Descriptor de despliegue 107
11.12 Tabla de decisiones 108
12 Bibliografía 109

Índice de ilustraciones

Ilustración 1 – Servicio de negocio dependiente de otros servicios de negocio (diferentes niveles de abstracción). 15
Ilustración 2 Servicios de negocio de aplicación (con puerto de servicio) y servicios de negocio de dominio. 17
Ilustración 3 – Particionado del modelo de dominio de datos utilizados por las aplicaciones 19
Ilustración 4 – Viejo modelo: visibilidad de las bases de datos entre aplicaciones 20
Ilustración 5 – Servicio de acceso a datos con múltiples implementaciones de su interfaz de acceso a datos 21
Ilustración 6 – Implementación de partida: backend alojado en el entorno host 21
Ilustración 7 – Implementación final: backend alojado en la plataforma Oracle 22
Ilustración 8 –Paralelos entre bases de datos distintas 23
Ilustración 9 – Racional del patrón comando 24
Ilustración 10 – Separación en capas proporcionada por las abstracciones de servicios y entidades de negocio (visto por un cliente de servicio) 25
Ilustración 11 – Capas que participan en la ejecución de un servicio 26
Ilustración 12 – Propagación de una entidad a través de las capas de la aplicación 28
Ilustración 13 – Adaptación de estructuras de datos en capas distintas 29
Ilustración 14 – Separación por capas vista desde la capa de servicios (adaptadores de servicio). 31
Ilustración 15 – Ejemplo de flujos de invocaciones entre los servicios de negocio y de datos a través del puerto de servicio y las distintas capas 34
Ilustración 16 – Visibilidad entre los componentes 35
Ilustración 17 – Entidades relacionadas 44
Ilustración 18 – Interfaz para paginación de datos 46
Ilustración 19 – Capa de servicio de acceso a datos siendo usada por la capa de negocio. 57
Ilustración 20 – Interacción prohibida entre aplicaciones y sistemas de almacenamiento de datos. Violación del principio de encapsulamiento. 58
Ilustración 21 – Aislamiento de los sistemas de almacenamiento de datos entre aplicaciones. 58
Ilustración 22 – Interfaces de negocio: reutilización de las implementaciones 60
Ilustración 23 – Sustitución de servicios de acceso a datos que utilicen la misma interfaz de acceso a datos 60
Ilustración 24 – Los servicios de acceso a datos son visibles solo en el interior de la aplicación que los define (sus interfaces son locales). 62
Ilustración 25 – Particionado del modelo de dominio de datos utilizados por las aplicaciones 66
Ilustración 26 – Servicio de acceso a datos con múltiples implementaciones de su interfaz de acceso a datos 67
Ilustración 27 – Implementación de partida: backend alojado en el entorno host 68
Ilustración 28 – Implementación final: backend alojado en la plataforma Oracle 69
Ilustración 29 – Configuración durante el despliegue: selección del backend 70
Ilustración 30 – Proxy de servicios de acceso a datos para la implementación de paralelos 72
Ilustración 31 – Servicio de acceso a datos proxy 73
Ilustración 32. Composición de entidades del contrato en un adaptador de servicio web y relación con las entidades de dominio. 88
Ilustración 33 – Algoritmo del proceso de alta de cita 98
Ilustración 34 – Servicios de negocios remotos identificados 99

Índice de ejemplos de código

Código 1 – Inyección de dependencia con los metadatos de la plataforma Java™ EE v. 5 38
Código 2 – Inyección de un servicio de negocio 38
Código 3 – Indicar el modelo de control de transacción utilizado por un Enterprise JavaBean™ 52
Código 4 – Marcar una transacción para el rollback a través del contexto de sesión 52
Código 5 – Marcar una excepción como excepción de aplicación que marque la transacción para el rollback 53
Código 6 – Estructura básica de un servicio de acceso a datos 63
Código 7 – Interfaz de acceso a datos 63
Código 8 – Pseudo-código de un proxy de un servicio que delega las operaciones a un vector de implementaciones delegadas 73
Código 9 – Pseudo-código de un servicio que utiliza dos servicios que utilizan dos backends distintos 74
Código 10 – Recuperar el contexto de persistencia en un Enterprise JavaBean™ 76
Código 11 – Inyección de dependencia en un Enterprise JavaBean™ a través de metadatos 77
Código 12 – Inyección de dependencia en un Enterprise JavaBean™ utilizando el descriptor de despliegue 78
Código 13 – Interfaz de un comando 78
Código 14 – Enterprise JavaBean™ de tipo stateful 80
Código 15 – Utilizar recursos no transaccionales en un Enterprise JavaBean™ 80
Código 16 – Método de invocación de comandos del interfaz EJBCommandSupport 81
Código 17 – Ejemplo: Servicio de acceso a datos que invoca un comando 81
Código 18 – Utilizar el atributo MANDATORY para requerir una transacción existente durante la ejecución de un método 81
Código 19 – Recibir el evento asociado al finalizar una transacción en un Enterprise JavaBean™ 82
Código 20 – Extracto de los métodos del interfaz javax.persistence.Query 83
Código 21.Lookup de la URL del punto de servicio. 95
Código 22 – Interfaz de negocio del componente EJB del servicio de negocio de alta de cita 102
Código 23 – Estructura básica del componente EJB del servicio de negocio de alta de cita 102
Código 24 – Inyección de dependencia de los componentes EJB utilizados 103
Código 25 – Recuperación de componentes del contexto de sesión del componente EJB 103
Código 26 – Pseudo-código de la lógica de negocio del servicio de alta de cita 104
Código 27 – Interfaz del servicio de acceso a datos de consulta de tramitación 104
Código 28 – Estructura del servicio de acceso a datos de consulta de tramitación 105
Código 29 – Estructura del comando para efectuar la consulta de tramitación lado HOST 105
Código 30 – Comando @Remove de un componente EJB stateful 106
Código 31 – Atributos de control de transacción en el EJB que implementa el servicio de alta de cita 106
Código 32 – Atributos de control de transacción en el método de negocio del EJB que implementa el servicio de alta de cita 106
Código 33 – Ejemplo de descriptor de despliegue para el componente EJB de alta de cita 107

Índice de tablas

Tabla 1. Tecnología a aplicar en cada entorno de ejecución....90
Tabla 2 – Servicios de negocios proporcionados por las aplicaciones....98
Tabla 3 – Servicios de negocios proporcionados por las aplicaciones....99
Tabla 4 – Interfaz del servicio de negocio de alta de cita....100
Tabla 5 – Interfaz del servicio de negocio de consulta de personas....100
Tabla 6 – Interfaz del servicio de negocio de alta de cita....101
Tabla 7 – Interfaz del servicio de negocio de pago de tasas....101
Tabla 8 – Interfaz del servicio de negocio de creación y grabación de PDF....101
Tabla 9 – Tabla de decisiones....108

1 Introducción

1.1 Objetivo

En el contexto del *Programa de Transformación Tecnológica*, la Dirección General de Tráfico ha decidido tomar medidas para asegurar coherencia y homogeneidad en todo desarrollo en la plataforma Java™ Enterprise Edition (1). Con esta especificación la Dirección General de Tráfico pretende proporcionar el conjunto de tecnologías y patrones a utilizarse durante la construcción de *servicios de negocio* y *servicios de acceso a datos,* así como en la manera de exponer estos servicios entre aplicaciones de forma controlada; categorizando el negocio de la organización. Los objetivos de estos patrones son:

- **Encapsular** la funcionalidad específica de las implementaciones que dependen de un concreto backend[1] en componentes de negocio que se utilicen solo a través de un conjunto de interfaces, iguales para todos los backends involucrados, que aíslen las aplicaciones de las peculiaridades de la tecnología que se está utilizando.

- **Facilitar la migración** de los sistemas existentes en la nueva plataforma tecnológica aislando la lógica de negocio de las aplicaciones de los cambios que se produzcan en la capa de acceso a datos.

- **Facilitar** los procesos de **auditoría** del código proporcionando *observables* que se puedan *medir* de forma ágil y de manera automática.

Esta especificación define y describe las tecnologías elegidas por la Dirección General de Tráfico y el modelo de programación a utilizarse durante el desarrollo de *servicios de negocio y acceso a datos* y en la exposición de dichos servicios entre distinta aplicaciones (modelo SOA[2]de arquitectura). La especificación de los patrones impuestos por Arquitectura intenta facilitar y armonizar el desarrollo de servicios entre aplicaciones de distintas áreas de negocio describiendo las herramientas disponibles en la Dirección General de Tráfico y las soluciones propuestas por la Arquitectura a problemas comunes.

Desde el punto de vista de desarrollo de servicios de negocios esta especificación ha sido diseñada para facilitar el desarrollo de aplicaciones data-centric (que son las predominantes en la DGT) y proporcionar los mecanismos necesarios para aislar los desarrollos de los backends a los que se conectan y permitir mezclar en *pseudo-transacciones* recursos que participan en transacciones JTA (2) y recursos no transaccionales. El modelo de programación utilizado para el manejo de la lógica transaccional es el modelo *declarativo*: se contemplarán exclusivamente los casos en que sea posible la propagación del contexto transaccional JTA (2) de forma automática, haciendo explícita prohibición de la utilización de API para el manejo programático de transacciones y datos tales como JTA (2), JPA (3) y JDBC.

[1] Tales como, por ejemplo, Oracle, un DB alojado en host, un sistema de gestión de contenido, etc.
[2] Service Oriented Architecture

Para llegar a publicar el negocio desarrollado en el ámbito de una aplicación a otros proyectos se sigue la filosofía y características detalladas en la "*Arquitectura Hexagonal*" (4) (actualmente renombrada a arquitectura de puertos y adaptadores). Con este modelo se consigue desacoplar las tecnologías de publicación (SOAP, REST, etc.) y la interfaz de usuario o capa de presentación, de la lógica de negocio de la aplicación. Esta arquitectura está orientada principalmente al desarrollo de adaptadores (5) y definición de puertos de servicio para la integración entre módulos y/o aplicaciones, así como en la segregación de interfaces e implementaciones a todos los niveles (6).

El objetivo principal que persigue la arquitectura se basa en estructurar las aplicaciones como componentes cohesivos funcional y/o tecnológicamente, con separación clara de responsabilidades entre los mismos y tratando de minimizar las dependencias entre ellos, que en todo caso se acoplaran a través del mínimo número de puntos posibles (y siempre bajo abstracciones que independizan de sus implementaciones). De esta manera se trata de garantizar la evolución y el mantenimiento de las aplicaciones hacia los nuevos paradigmas que están surgiendo en el desarrollo de aplicaciones de empresa.

1.2 Destinatarios del documento

Este documento está dirigido a los arquitectos, analistas y desarrolladores de aplicaciones en la plataforma Java™ EE (1) para aplicaciones de la Dirección General de Tráfico.

El documento presupone que el lector tiene un conocimiento básico de las API que componen la especificación Java™ EE (1), con particular énfasis a las API Java™ Transaction API (JTA) (2) y Java™ Persistence API (JPA) (3). No obstante, donde se revise la necesidad, se hará explícita referencia a las secciones pertinentes de la documentación oficial de dichas API.

1.3 Convenios de la especificación.

En esta especificación usaremos como convenio NOMBRE_OPERACION como denominación de la operación servicio en nomenclatura capitalizada sin acentos. Ejemplos de su aplicación pueden ser "*AltaNuevoConductor*" o "*TransferenciaVehiculoSinMatricular*". Esta clase de interfaz tendrá como métodos las operaciones a invocar como, por ejemplo, "*alta*" o "*transferencia*".

En adelante NOMBRE_RESUMEN será la denominación de toda una operativa que la aplicación va a compartir. Se aplicará principalmente para el nombrado de artefactos y librerías. Se utilizará como una composición capitalizada y sin acentos: "*NuevosConductores*" o "*PreMatriculacion*".

TOPICO significa también un grupo funcional o agrupación de operaciones relacionadas dentro de una temática común. Se aplicará principalmente para los nombres de paquetes. Se utilizará en minúsculas y sin acentuación. En los ejemplos anteriores bien pudieran ser "*gestión.personas*" y "*vehículos.transferencias*" en los casos anteriores. Si es necesario, pueden tener más de un nivel.

ACRONIMO es el nombre de cuatro letras que identifica a la aplicación para los Departamentos de Calidad, Pruebas y Sistemas. Este acrónimo es asignado al proyecto al inicio del mismo.

AREA representa al área de negocio desde un punto de vista funcional donde está enmarcada la aplicación que provee el servicio. No se debe confundir el área de negocio, con el área organizacional, aunque en algunos casos coincidan. Se denominará en minúsculas y sin acentos. Las áreas identificadas en la DGT son *"conductores"*, *"vehiculos"*, *"examenes"*, *"sanciones"*, *"arquitectura"*, ***"observatorio"*** y *"horizontales"* (sistemas de información común).

2 Visión general

2.1 Objetivos

Los objetivos que Arquitectura persigue definiendo esta especificación son los siguientes:

- Establecer un modelo de componente estándar para construir servicios de negocio en la plataforma Java™ EE v. 5 (1) para las aplicaciones de la Dirección General de Tráfico.

- El modelo de construcción de servicios de negocio facilitará el análisis y el diseño de la lógica de negocio de las aplicaciones. La separación de la lógica de negocio en componentes dedicados facilitará la reutilización de servicios existentes entre distintas aplicaciones y dentro de una aplicación (especialmente si es compleja) y reducirá la duplicación innecesaria de código y los costes de mantenimiento de las aplicaciones.

- La definición de las entidades de negocio internas *("dominio interno")*[3] manejadas por el negocio permitirá encomendar las responsabilidades y la gestión de estas estructuras de datos al equipo al que le corresponda siguiendo los criterios y los protocolos de actuación establecidos por la Dirección General Tráfico.

- El uso de la arquitectura hexagonal (o de puertos y adaptadores) en los servicios de negocio externos permite la exposición de la lógica de negocio (normalmente a nivel alto de abstracción) a través de los denominados "adaptadores de servicio".

- La especificación para la construcción de servicios de acceso a datos será el modelo de componente estándar para construir servicios de datos en la plataforma Java™ EE v. 5 (1) para las aplicaciones de la Dirección General de Tráfico.

- La arquitectura de los servicios de acceso a datos agilizará la realización de aplicaciones que exploten mecanismo de persistencia de datos[4]: los desarrolladores no deberán preocuparse del

[3] Este es el dominio esencial de las aplicaciones que define los conceptos con los que se trabaja. En la DGT se ha definido tradicionalmente en la capa de negocio o en las capas trasnversales.
[4] Bases de datos, sistemas de ficheros, gestores documentales, sistemas externos, etc.

manejo de API y conceptos de nivel más bajo tales como transacciones (Java™ Transaction API –JTA- (2) y Java™ Transaction Service –JTS- (7)).

- La arquitectura de los servicios de acceso a datos soporta únicamente el modelo transaccional declarativo proporcionado por Java™ EE v. 5.

2.2 Servicios de negocio

Los servicios de negocio son los componentes en los cuales se abstrae la lógica de negocio (distribuida en una capa de negocio), formando bloques de funcionalidad de negocio *atómica* con los cuales se pueden construir flujos de ejecución de lógica de negocio cada vez más complejos.

Un servicio de negocio es la *unidad* de lógica de negocio y su interfaz representa las operaciones *atómicas*[5] que el negocio define.

El servicio de negocio tiene las siguientes características:

- Define y expone una *interfaz de negocio* (patrón *published interface* (6)).
- Encapsula la lógica de negocio que implementa su interfaz de negocio.
- Actúa como punto de entrada de las invocaciones a lógica de negocio (patrón *abstract modules* (6)).
- Demarca la naturaleza atómica de las operaciones de negocio.
- Desacopla la lógica de negocio de las tecnologías que subyacen a las implementaciones de los servicios.
- Debe ser funcionalmente cohesivo y procurar no mezclar elementos funcionales poco relacionados entre ellos. Cada servicio de negocio debería servir para un solo propósito (patrón *cohesive modules* (6)).

2.2.1 Naturaleza atómica de los servicios de negocio

Los servicios de negocio exponen a través de sus interfaces la funcionalidad definida por el negocio. Las operaciones definidas por el negocio se hacen visibles a los clientes a través de su API, es decir, a través de las interfaces que cada servicio expone. Los métodos de interfaz de los servicios de negocio actúan como punto de entrada a la capa de negocio (según el servicio de negocio del que estemos hablando) y estos métodos deben considerarse atómicos desde el punto de vista del negocio.

[5] En este contexto, la palabra atómica adquiere un sentido más genérico: una *operación de negocio* es parte del conjunto de operaciones definidas por el negocio. Siendo la lógica de negocio encapsulada totalmente en servicios la unidad mínima de invocación coincide con una operación de negocio definida por la interfaz de un servicio. Este concepto se aclarará más adelante.

Las operaciones de negocio pueden componerse en flujos más complejos y utilizarse durante la construcción de otros servicios de negocio. Uno de los objetivos de los servicios de negocio, de hecho, es la completa encapsulación de la lógica que, junto a una apropiada actividad de gobierno, permitirá catalogar los servicios y las operaciones existentes para evitar la proliferación innecesaria de código y reducir los impactos en el mantenimiento asegurando que la lógica de negocio esté centralizada en lugares apropiados.

La funcionalidad de servicios de negocio más complejos podrá depender de otros servicios de negocio y la lógica resultante podrá contener una orquestación de invocaciones a operaciones de negocio de otros servicios, como queda ejemplificado en la figura siguiente.

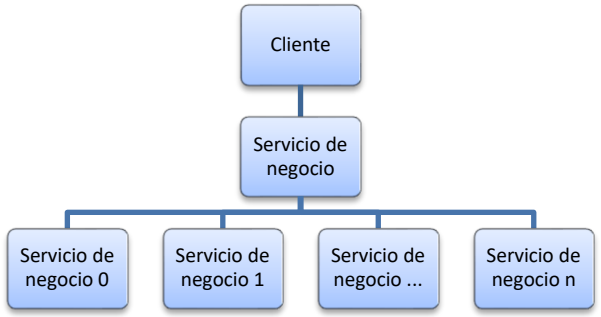


Ilustración 1 – Servicio de negocio dependiente de otros servicios de negocio (diferentes niveles de abstracción).

La lógica que invoca servicios de negocio, indicada como el *cliente* en Ilustración 1, solo es consciente de estar invocando una operación de negocio definida en la interfaz de un servicio de negocio, desacoplándose de las dependencias y de las tecnologías que se estén utilizando para implementar dicho servicio. Si un servicio de negocio invoca otros servicios de negocio, desde el punto de vista del llamante, no deberá perderse la naturaleza atómica de las operaciones, y quien implemente el servicio de negocio se deberá responsabilizar de garantizar esta semántica; el servicio de negocio es el punto de entrada de las invocaciones por los clientes y el punto de demarcación de la naturaleza atómica[6] de las operaciones invocadas: aunque la implementación se haya realizado a través de un flujo de invocaciones a varios servicios de negocio desde el punto de vista del llamante la operación es atómica.

2.2.2 Invocaciones asíncronas

El equipo de Arquitectura de la Dirección General de Tráfico quiere proporcionar semántica asíncrona a las operaciones de negocio de forma que sea posible para un cliente ejecutar invocaciones de servicios de negocio sin quedarse a la espera de que termine la ejecución de dicha operación.

Desde el punto de vista de las operaciones de negocio, las invocaciones asíncronas se dividirán en dos categorías:

[6] En otras palabras, de la *transaccionalidad*.

- Operaciones asíncronas de las cuales se pretende obtener la respuesta durante la invocación de la operación de negocio en curso.

- Operaciones asíncronas de las cuales no se pretende obtener la respuesta.

A la primera categoría pertenecen las operaciones de negocio que se benefician del poder paralelizar operaciones no interrelacionadas, las respuestas de las cuales se esperan antes de que se termine la invocación de la operación en curso. A la segunda categoría pertenecen las operaciones de negocio cuyo objetivo es únicamente el envío de una solicitud sin necesidad de esperar una respuesta (o al menos, no de forma inmediata): funcionalmente se proporciona entonces un mecanismo para la invocación de procesos de larga duración.

Las invocaciones asíncronas permiten en ambos casos agilizar la ejecución de las operaciones de negocio.

2.2.2.1 **Invocación en paralelo**

Al día de hoy, las operaciones de negocio de la primera categoría se pueden implementar a través de la JMS API; pero los servidores de aplicaciones han desarrollado una especificación al respecto similar a las herramientas más sofisticadas para el threading de la plataforma Java™ Standard Edition 5 (Java™ SE 5), las cuales facilitan notablemente la implementación de este tipo de funcionalidad. Las técnicas estándar de Java™ SE 5 no deberían utilizarse dentro de un contenedor porque generalmente suponen la creación de hilos de ejecución *ad hoc (que pueden llegar a interferir con el servidor de aplicaciones)*. Por tanto esta especificación, exige el uso de la API *Commonj* (8) que permite ejecutar tareas en paralelo dentro de un servidor de aplicaciones de forma controlada por el contenedor.

2.2.2.2 **Procesos de larga duración**

La segunda categoría de operaciones de negocio también se implementarán utilizando la JMS API aunque no se excluye la posibilidad de invocar otra tipología de componentes, tales como los servicios web, implementando la lógica de correlación de mensajes necesaria para obtener la respuesta del mensaje de forma asíncrona.

En este caso no se asume como requisito la necesidad de establecer una correlación entre solicitud y respuesta: la responsabilidad de proporcionar la respuesta de una operación invocada de esta manera, cuando se necesite, es entonces responsabilidad de quien publica la operación en cuestión. Este tipo de soluciones involucra un cambio de semántica en la operación que se ofrece, ya que en vez de retornar en una sola operación el resultado de una invocación, se suele desdoblar la operación para contemplar la posibilidad bien de preguntar más adelante por el estado de finalización de una operación de la que se ha obtenido en la primera llamada un identificador único (*polling*), bien de ofrecer por parte del llamante un método para ser informado del resultado de la operación tras ser finalizado por el servicio invocado (*callback*).

2.3 Estructura de la capa de negocio (Servicios de negocio de aplicación y de dominio): Puerto de servicio

Para conseguir una organización interna de la lógica de negocio ordenada (especialmente en aplicaciones complejas), se recomienda desde el Departamento de Arquitectura de la DGT (aunque no se obliga a ello[7]), que la capa de negocio se estructure en 2 niveles de servicios de negocio. Los más externos, corresponden a un nivel de abstracción más alto y se puede considerar la **subcapa de servicios de aplicación**[8] que es de presencia obligatoria; y los más internos son lo que se puede considerar como la **subcapa de servicios de dominio**[9]**;** que en los diseños basados en dominio (*Domain Driven Design* (9)) se implementan en base al "dominio interno" de la aplicación. La presencia de esta última capa es opcional aunque desde el departamento de arquitectura se recomienda si el negocio es suficientemente complejo.

Las interfaces de los servicios de negocio de aplicación tienen una importancia considerable porque están destinadas a ser expuestas a otras aplicaciones a través de los adaptadores de servicio. Estas interfaces se denominarán, atendiendo a la arquitectura hexagonal, **el puerto de servicio.**

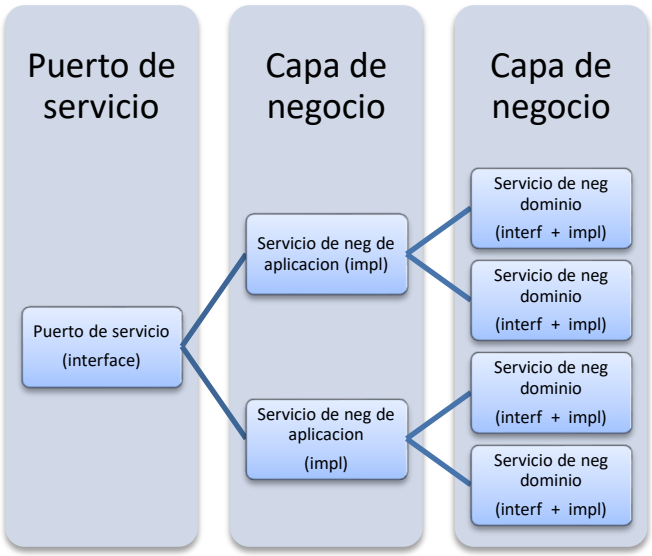


Ilustración 2 Servicios de negocio de aplicación (con puerto de servicio) y servicios de negocio de dominio.

2.3.1 Reutilización del puerto de servicio.

La publicación de un servicio en dos o más tecnologías distintas, es un requisito que puede aparecer a menudo en una aplicación. Dentro de la arquitectura propuesta es posible reutilizar el mismo puerto de servicio para ambas publicaciones.

[7] No es opcional el establecimiento del puerto de servicio y la implementación por la capa de negocio de la aplicación. Lo que es opcional es la presencia de una capa de servicios de negocio de dominio dentro de la capa de negocio para estructurar las capas de negocio complejas.
[8] Service layer o Application services en http://dddsample.sourceforge.net/characterization.html#Services
[9] Domain services en http://dddsample.sourceforge.net/characterization.html#Services

Básicamente se trata de realizar un adaptador de servicio (2.4) por tecnología de publicación para exponer el servicio de forma agnóstica por medio de distintas tecnologías. Cada uno de ellos instancia la interface del puerto de servicio (cuya implementación se encuentra en la lógica de negocio de la aplicación) e invoca las operaciones necesarias, que no deben variar por realizarse la invocación desde tecnologías distintas.

2.3.2 Seguridad del servicio.

Para establecer la seguridad del servicio se deben utilizar los mecanismos expuestos en la especificación DGTE-002 (10).

2.4 Exposición de los servicios de negocio a otras aplicaciones (adaptadores de servicios)

A través de los adaptadores de integración (SOAP, REST) se puede exponer el negocio a otras aplicaciones. No obstante, se debe tener en cuenta que este negocio de alto nivel de abstracción puede llegar a ser usado desde la presentación de la propia aplicación (esto último será dependiente de cómo haya organizado el diseño de la capa de negocio la aplicación), llamándose entonces adaptadores de presentación.

Se debe desarrollar un módulo por cada adaptador de entrada existente en función de su tecnología SOAP, REST o Presentación. Queda prohibido el desarrollo de adaptadores de integración tipo EJB: El uso de EJB remoto sólo está permitida para código legado que tenga clientes activos haciendo uso de dicha interfaz. Para estos casos es obligatorio que la aplicación provea a su vez de un adaptador de integración basados en protocolos web (SOAP/REST)

Los adaptadores de integración siempre accederán al negocio por medio de los puertos de servicio mientras que en el caso de los adaptadores de presentación, será opcional, ya que podrán acceder a los servicios de negocio internos (o servicios de dominio) directamente.

Esta especificación establece la obligatoriedad, en caso de ser necesario ofrecer servicios, de ofrecer al menos un adaptador de servicio basado en protocolos web (REST o SOAP). La recomendación del Departamento de Arquitectura es utilizar preferentemente interfaces REST, siendo además preceptivo que existan interfaces REST cuando el cliente de un servicio sea una aplicación funcionando en un dispositivo móvil.

2.5 Entidades manejadas por el negocio (“dominio interno”).

En las interfaces que los servicios de negocio proporcionarán como contratos, podrán manejarse estructuras de datos definidas por la aplicación a la que pertenece el servicio o **dominio interno.**

Las responsabilidades derivadas de la existencia de las entidades de **dominio interno** estarán a cargo de la aplicación que las define. Comúnmente estas responsabilidades serán la definición de las entidades y la definición de sus interfaces de negocio en las interfaces de los servicios que las manejan.

Los adaptadores de servicio, los puertos de servicio, los servicios de negocio (de aplicación y en caso de ser necesarios, de dominio), las entidades de negocio y los servicios de acceso a datos (2.6), son las herramientas conceptuales a través de las cuales la Dirección General de Tráfico quiere conseguir el absoluto desacoplamiento entre las aplicaciones y los objetos por ellas manejados.

2.6 Servicios de acceso a datos

Debido a la importancia y a la gran utilización que hacen las aplicaciones de los sistemas de almacenamiento de datos se estima oportuno añadir un conjunto de abstracciones para desacoplar las aplicaciones de las peculiaridades de los backends que estén utilizando.

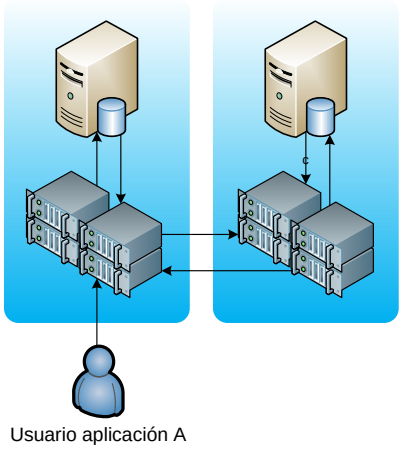


Ilustración 3 – Particionado del modelo de dominio de datos utilizados por las aplicaciones

Como queda ilustrado en la figura precedente, en el modelo propuesto por esta especificación cada aplicación adquirirá la responsabilidad de definir y manejar su modelo de datos. La misma aplicación, así como las demás aplicaciones que utilicen servicios de negocio proporcionados por ésta, no tendrán visibilidad de los sistemas de almacenamiento utilizados para organizar los datos. La situación en la cual una aplicación tiene visibilidad del backend de almacenamiento de datos utilizado por otra aplicación[10], ilustrada en la siguiente figura, queda prohibida por esta especificación de construcción de servicios de negocio y servicios de acceso a datos.

La responsabilidad adquirida por las aplicaciones sobre su propio modelo de datos tiene como consecuencia la imposibilidad por otra aplicación de acceder directamente a los backends de almacenamiento que no sean de su competencia.

[10] A través de cualquier medio: conexión directa, creación de vistas ad hoc, etc.

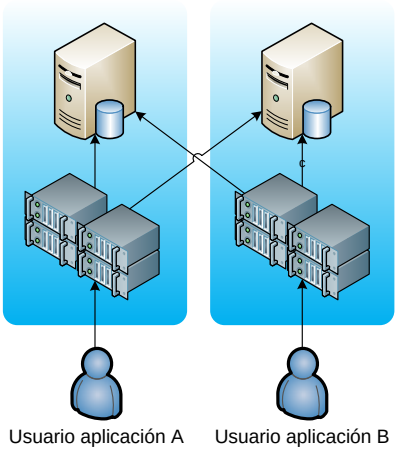


Ilustración 4 – Viejo modelo: visibilidad de las bases de datos entre aplicaciones

Una aplicación interactuará con los sistemas de almacenamiento de datos de su competencia a través de una categoría de componentes llamados *servicios de acceso a datos*. Los servicios de acceso a datos son componentes diseñados para encapsular exclusivamente lógica de acceso a datos y separar las aplicaciones de los sistemas de almacenamiento de datos que estén utilizando. Servicios de este tipo no podrán por lo tanto contener ningún tipo de lógica de negocio (entre otras cosas sería una violación del principio de capas o layers).

Los servicios de acceso a datos implementan una interfaz que será utilizada para desacoplar las implementaciones de los servicios de acceso a datos y facilitar la migración de las aplicaciones cuando cambie el backend de almacenamiento de datos.

Las implementaciones de los servicios de acceso a datos serán ofrecidas por la plataforma de servicios de acceso a datos, de forma que las aplicaciones sólo tendrán que implementar un cliente hacia los servicios de acceso a datos.

Como puede verse en la imagen siguiente, un cliente puede estar utilizando, a través de su interfaz de negocio, una o más implementaciones de servicios de acceso a datos. Si cambiase un backend podría por ejemplo sustituirse una implementación o incluso añadir una nueva implementación para suplir las necesidades que se hayan producido.

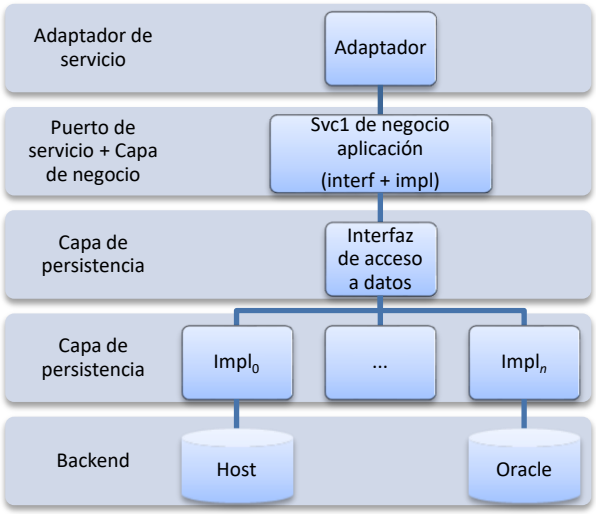


Ilustración 5 – Servicio de acceso a datos con múltiples implementaciones de su interfaz de acceso a datos

2.6.1 Soporte a la migración de los backends de almacenamiento de datos

Así mismo, los servicios de acceso a datos se utilizarán para construir soluciones que permitan dar soporte a múltiples sistemas de almacenamiento de datos durante la vida de una aplicación.

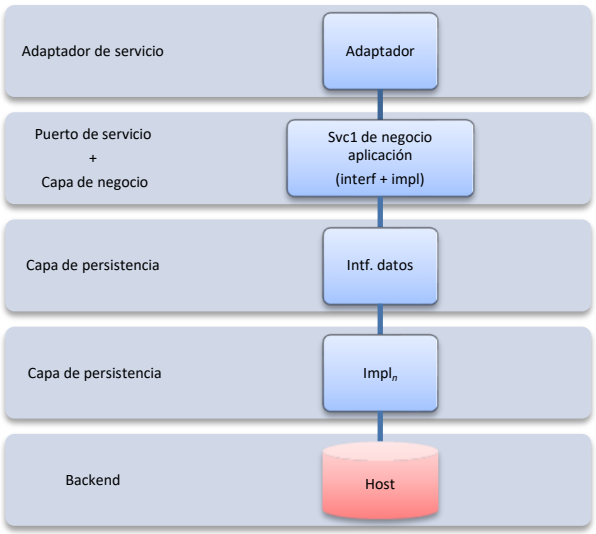


Ilustración 6 – Implementación de partida: backend alojado en el entorno host

Una aplicación podría por ejemplo estar invocando un servicio de negocio que a su vez utiliza un servicio de acceso a datos cuyo backend subyacente fuera una base de datos alojada en el entorno host, como puede verse en la Ilustración 6.

Cuando cambie un backend de almacenamiento de datos será suficiente proporcionar una implementación alternativa de un conjunto de servicios de acceso a datos sin que la lógica del llamante se vea afectada. Será entonces suficiente con realizar una implementación alternativa de las interfaces de acceso a datos de los servicios de acceso a datos afectados por la migración y sustituir la implementación de los servicios a migrar.

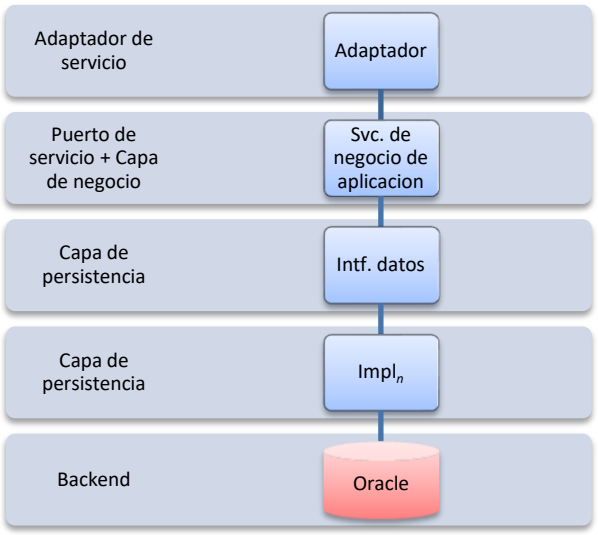


Ilustración 7 – Implementación final: backend alojado en la plataforma Oracle

Los servicios de acceso a datos, junto con los servicios de negocio, separan ortogonalmente la lógica de negocio de la lógica de acceso a datos: este patrón proporcionará ayuda eficaz a la hora de migrar aplicaciones existentes y construir nuevas aplicaciones en cuanto que la lógica que maneja y explota datos está completamente desacoplada de la lógica que proporciona dichos datos.

2.6.2 Soporte al paralelo de backends distintos durante la migración

Durante la migración de bases de datos se podría tener que pasar a través de una temporada en que habrá que dar soporte a un conjunto de bases de datos distintas en paralelo.

La tipología de acceso a cada una de estas bases de datos dependerá de cómo cada aplicación esté utilizando las bases de datos que se están migrando. En general podría requerirse acceso en lectura, escritura o una combinación de ellos.

El Departamento de Arquitectura de la Dirección General de Tráfico quiere dar soporte a este caso de uso.

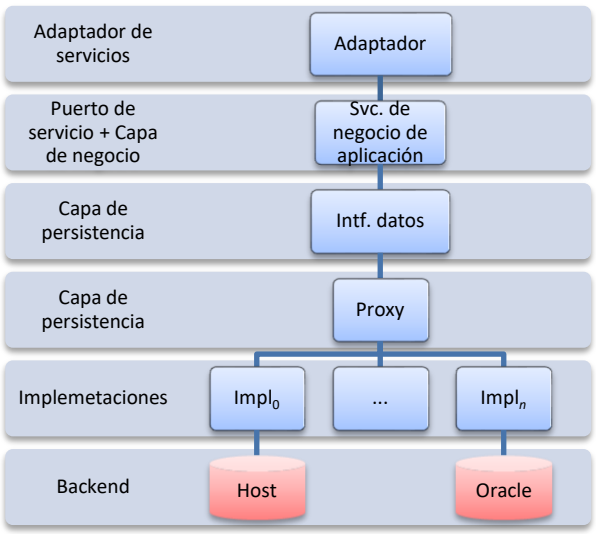


Ilustración 8 –Paralelos entre bases de datos distintas

En los capítulos siguientes se detallarán patrones y recomendaciones para solucionar problemas de este tipo.

2.7 Lógica transaccional

Los servicios de negocio de aplicación son los componentes que proporcionan la definición de las operaciones publicadas por las aplicaciones a través de adaptadores para ser consumidas por las aplicaciones externas. Los servicios de acceso a datos forman parte de la categoría de objetos que manejarán *recursos transaccionales*. En ambos casos, con requerimientos distintos, es necesario proporcionar características básicas de naturaleza atómica y coherencia en los flujos de operaciones que se pueden construir.

A nivel de los servicios de acceso a datos, que son *manejadores* de recursos, es necesario explotar los mecanismos necesarios para poder manejar los recursos de forma transaccional. A nivel de los servicios de negocio es necesario garantizar que los servicios de acceso a datos que se invocan dentro de un servicio se ejecuten dentro de la misma transacción y, a nivel aún más alto, que múltiples invocaciones de servicios de negocio también se ejecuten dentro de la misma unidad de trabajo transaccional (especialmente cuando los servicios de negocio de aplicación se implementan en base a servicios de negocio de dominio).

Java™ Enterprise Edition (1) proporciona una API para el manejo de recursos transaccionales que soportan el modelo transaccional declarativo: según este patrón, la lógica que maneja esta categoría de recursos no necesita invocar explícitamente la API para el manejo de la lógica transaccional y en su lugar se proporcionan mecanismos adecuados para que la configuración sea completamente declarativa. La especificación Java™ EE (1) define el uso de la API JTA (2) para el manejo de recursos transaccionales.

Este modelo permite la escritura de código que, sin la necesidad de invocar la API de forma explícita, se ejecute dentro de una transacción. El manejo de este tipo de recursos asegura que, al final de una transacción, el conjunto de los recursos se encuentre en una situación coherente y, si se produjera algún error, todos ellos anularían las operaciones efectuadas y volverían al estado que tenían al comenzar la transacción.

2.7.1 Recursos no transaccionales

En el marco del *Proyecto de Transformación Tecnológica*, el equipo de Arquitectura de la Dirección General de Tráfico ha detectado la necesidad de manejar también recursos no transaccionales[11]. Para integrar de forma sencilla el manejo de esta categoría de recursos, esta especificación proporciona patrones y modelos de programación ad hoc como el patrón *Command* (5): durante el desarrollo de las aplicaciones, cuando se necesite manejar recursos no transaccionales, habrá que seguir las indicaciones de esta especificación para garantizar la *homogeneidad*, la *calidad* y la *posibilidad de auditar* el código producido por los equipos de desarrollo de las aplicaciones.

El patrón Command ayuda a colmar la falta de control transaccional de ciertos recursos. Especificaciones como JTA (2) y JTS (7) permiten la construcción de sistemas que soporten el control transaccional de las operaciones que involucren sus recursos. Los recursos sin esta propiedad necesitan de un mecanismo para grabar las operaciones ejecutadas dentro de una transacción y de un mecanismo para poderlas deshacer. Esta especificación, como queda detallado en los siguientes capítulos, aborda este problema obligando a que cada operación que involucre un recurso no transaccional se produzca a través de un comando. El comando es una estructura que contiene también el flujo de operaciones, llamado *flujo de compensación*, necesarias para deshacer una operación efectuada anteriormente. La responsabilidad de proporcionar el flujo de operación correspondiente a cada operación es responsabilidad del equipo de desarrollo que está realizando el servicio de acceso a datos.

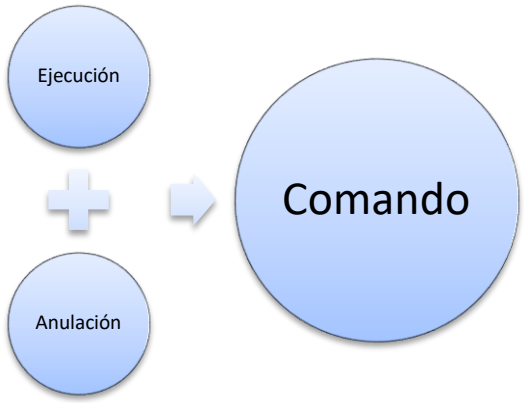


Ilustración 9 – Racional del patrón comando

Este patrón permite encapsular un comando con la operación necesaria para efectuar su anulación. Teniendo traza de los comandos ejecutados dentro de una transacción, en caso de error se puede soli-

[11] No transaccionales en el sentido JTA (2), es decir, recursos que no pueden ser manejados a través del API JTA.

citar la anulación de estos comandos ejecutando todas las operaciones de anulación correspondientes. Las anulaciones se ejecutarán en el orden inverso en el que se han ejecutado los comandos.

Podrán darse casos en los cuales los comandos no puedan anularse y por lo tanto el flujo de anulación, o compensación, no existirá. El Departamento de Arquitectura de la Dirección General de Tráfico sugiere, siempre que sea posible, diseñar procesos que manejen recursos no transaccionales que sean compensables.

2.8 Representación esquemática

Esta sección resume la filosofía que ha justificado la solución propuesta por el Departamento de Arquitectura desde el punto de vista de los actores que participan en las invocaciones.

2.8.1 Lógica de negocio vista por el cliente

Desde el punto de vista del negocio, así como se publica a los clientes de los servicios, el resultado de las abstracciones propuestas se ilustra en la siguiente imagen:

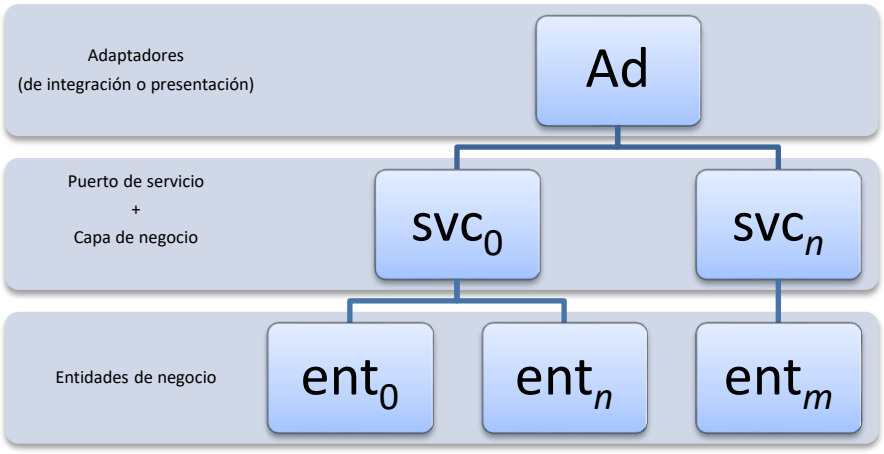


Ilustración 10 – Separación en capas proporcionada por las abstracciones de servicios y entidades de negocio (visto por un cliente de servicio)

La lógica de negocio, encapsulada en servicios, maneja *entidades* a través de las API que cada aplicación proporcionará. En la estabilidad de estas API está la base del aislamiento que este diseño proporcionará a las aplicaciones durante la migración de las bases de datos. Modelando las entidades de negocio como *objetos opacos*[12] las aplicaciones:

[12] Manejables exclusivamente a través de un API proporcionada por el proveedor de la entidad en cuestión.

- Están cumpliendo los *contratos* que proporcionan a las demás manteniendo estables estas interfaces.

- Se mantienen estables durante la migración porque dependen sólo y exclusivamente de los contratos de negocio expuesto por las demás aplicaciones que estén utilizando.

Gracias a los adaptadores de servicio esta lógica de negocio es expuesta a los clientes de los servicios a través de distintas tecnologías.

2.8.2 Lógica de negocio dentro de las aplicaciones

Las aplicaciones, frente a una invocación de uno de sus servicios, ejecutarán la lógica de negocio que implementa este servicio. En la ilustración siguiente proponemos una representación esquemática de las capas que participan en la ejecución de un servicio.

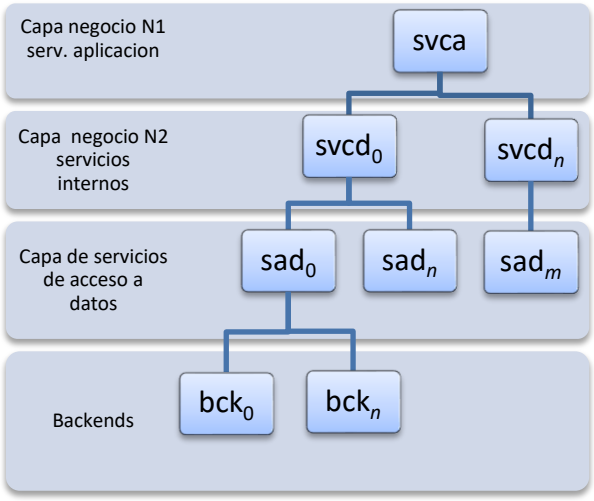


Ilustración 11 – Capas que participan en la ejecución de un servicio

2.8.3 Visión de conjunto

En conjunto las aplicaciones:

- Interactúan entre ellas exclusivamente a través de los adaptadores de los servicios de negocio.

- En el caso de los servicios de negocio de alto nivel de abstracción (o de aplicación) se denomina puerto de servicio a la interfaz que expone sus operaciones al exterior a través del adaptador.

- Los servicios de negocio (internos normalmente) interactúan con los backend de almacenamiento de datos a través de servicios de acceso a datos.

- Todo servicio de negocio (ya sea servicio de aplicación o de dominio) expone una interfaz de negocio para soportar la sustitución de las implementaciones a lo largo de la vida de las aplicaciones.

2.8.4 Entidades de negocio

El concepto de *entidad de negocio* no se realizará en una implementación *canónica*[13] en la plataforma Java™ así como ocurre con los servicios de negocio y los servicios de acceso a datos. Este concepto se aplicará a los objetos manejados por el negocio de una aplicación y esta matización procederá del análisis funcional de la aplicación.

Como se ha explicado someramente en las secciones precedentes, los servicios de negocio podrán invocar en la misma aplicación:

- API publicadas por servicios de negocio.
- API publicadas por servicios de acceso a datos.

Mientras que en otras aplicaciones se podrá invocar a las API de los clientes de los servicios de negocio (expuestos normalmente a través de los adaptadores de servicio).

Durante el flujo de llamadas entre servicios de negocio y entre servicios de negocio y servicios de acceso a datos se producirá un flujo de información entre componentes. Esta información se compondrá también de:

- Datos procedentes de la misma aplicación desde donde se está efectuando la invocación.
- Datos procedentes de servicios proporcionados por otras aplicaciones.

Subrayar la existencia del concepto de entidad de negocio en fase de análisis ayudará a las aplicaciones a diseñar API de servicios que aíslen apropiadamente a las aplicaciones que dependan de estos servicios para que no dependan de las implementaciones de los objetos de negocio publicados por otras aplicaciones (mediante transformación de entidad en los adaptadores de salida). Aunque una entidad de negocio aparezca como una sencilla estructura de datos hay que tener en cuenta que esta estructura, en sí, es también parte del *contrato* expuesto por un servicio. Por lo tanto, así como las aplicaciones se encargarán de establecer interfaces estables para sus servicios, también tendrán que prestar atención a mantener estables las API necesarias para manejar los objetos que aparecen en estos contratos.

Aunque parezca conveniente exponer estructuras de datos tan sencillas como un JavaBean™ habrá que tener en cuenta:

[13] Esto significa que esta especificación no define como este concepto debe realizarse en la plataforma Java™ EE.

- Utilizar internamente en una aplicación el mismo tipo de dato exportado en una API que la aplicación publica genera un acoplamiento que puede ser complicado remover si en el futuro la aplicación quisiera cambiar la implementación de algún tipo de dato utilizado internamente.

Las API que manejan POJO[14], si por un lado simplifican el desarrollo de las aplicaciones, por otro lado pueden subestimar este problema en fase de análisis.

Se puede pensar que resulta conveniente utilizar la misma estructura de datos en todas las capas. Pero habrá que tener en cuenta que el contrato de los servicios podría estar acoplado a la implementación de los componentes realizados en las demás capas (hasta la capa de acceso a datos), como queda ejemplificado en la siguiente ilustración.

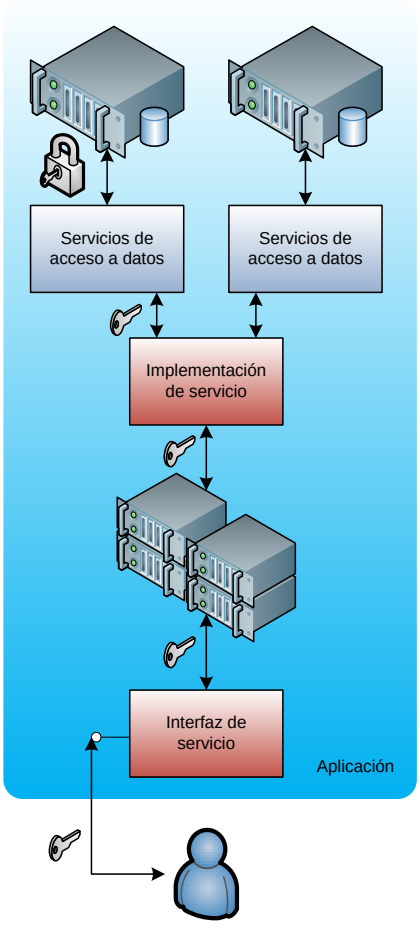


Ilustración 12 – Propagación de una entidad a través de las capas de la aplicación

[14] Eventualmente enriquecidos con metadatos.

Si con el paso del tiempo fuere necesario aportar modificaciones en la capa de acceso a datos, por ejemplo porque se ha cambiado de gestor de almacenamiento de datos y el modelo de dominio ha cambiado, inevitablemente será necesario adaptar las nuevas estructuras de datos a los viejos contratos. La adaptación de las estructuras de datos podría efectuarse tanto a nivel de servicio de acceso a datos como a nivel de servicio de negocio. Evidentemente se aconseja efectuar esta adaptación lo antes posible para limitar este efecto en las capas de nivel superior. Ambas alternativas se ilustran en la figura siguiente.

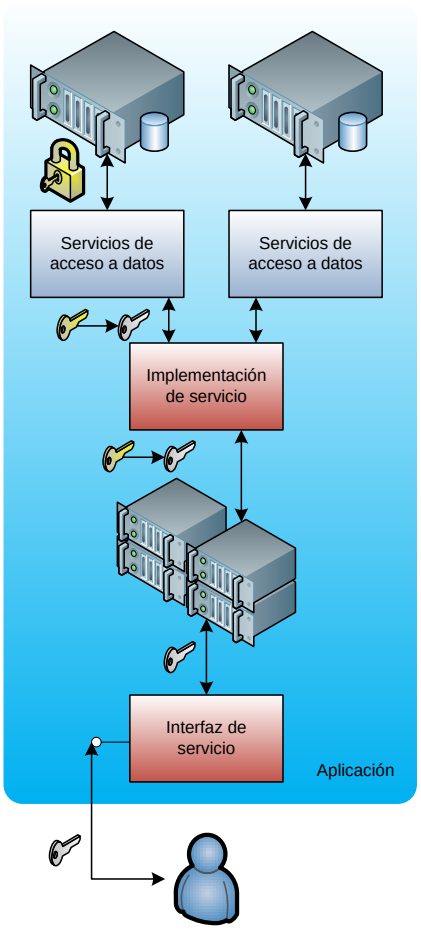


Ilustración 13 – Adaptación de estructuras de datos en capas distintas

2.8.4.1 Un ejemplo: la Java™ Persistence API

La Java™ Persistence API (JPA) (3), por ejemplo, permite implementar como JavaBeans™ los tipos de datos manejados por una aplicación y utilizarlos en toda las capas de una aplicación hasta la capa de persistencia de datos. En la capa de persistencia, a través de los metadatos de configuración disponi-

bles en esta especificación, se establece una correspondencia entre los campos de un JavaBean™ y el backend de persistencia utilizado para grabarlos: en el caso de una base de datos relacional, básicamente, hay que establecer una correspondencia entre el JavaBean™ y sus campos y la tabla y sus columnas. Esta especificación, evidentemente, hace que la capa de persistencia sea más sencilla de modelar y que entre la aplicación y la capa de persistencia no haya nada más que *configuración*.

La Java™ Persistence proporciona un modelo de programación sencillo a través del cual se obtiene un desacoplamiento entre la forma en la cual se representan y se manejan los datos (puros JavaBeans™) y la capa de persistencia. De hecho, quién maneje una entidad no es siquiera consciente de que está utilizando un elemento de la capa de persistencia. Si por un lado este aligeramiento del modelo de programación es una ventaja en fase de desarrollo, en fase de diseño no hay que olvidar que la utilización de las *entidades* en la API de una aplicación puede ocasionar acoplamientos no deseados entre aplicaciones. Aunque la flexibilidad en la configuración de las entidades JPA podrá ayudar a subsanar este tipo de problema cuando se ocasionen hay que detectar estas situaciones para anular estos acoplamientos por estar prohibidos por el Departamento de Arquitectura.

Algunos de los problemas que se pueden encontrar en este escenario son las siguientes:

- La aplicación que utiliza los servicios publicados depende de los tipos de datos que aparecen en la interfaz del servicio y por lo tanto tendrá que disponer de la definición de todas las clases involucradas.

- Durante la migración de base de datos es razonable asumir que el modelo lógico y físico de las bases de datos utilizados por una aplicación sea candidatos a revisiones. Asumir que una entidad JPA sea una representación razonablemente estable en el tiempo[15] de los tipos de datos manejados internamente parece una afirmación que no garantiza el adecuado aislamiento requerido por Arquitectura.

La capa de servicios de negocio y la capa de acceso a datos, por lo tanto, tienen la responsabilidad de proporcionar el nivel adecuado de aislamiento dentro y fuera de cada aplicación:

- La capa de servicios de negocio (a través de los adaptadores de servicio) publica los contratos a través de los cuales las aplicaciones pueden interactuar.

- La capa de negocio publica los contratos en el puerto de servicio a través de los cuales los adaptadores se comunican con el negocio de las aplicaciones.

- La capa de servicios de acceso a datos proporciona un nivel de desacoplamiento entre una aplicación y los backends que esta utiliza. La capa de servicios de acceso a datos proporciona además el mecanismo que desacopla una aplicación de los tipos de datos a persistir manejados internamente en los casos en que este desacoplamiento adicional se estime necesario.

- La capa de adaptadores de salida, proporciona un nivel de desacoplamiento entre el negocio de la aplicación y los contratos de los servicios externos que consume.

[15] Se entiende durante la migración.

La imagen siguiente ilustra, desde el punto de los servicios de negocio, el resultado de esta abstracción:

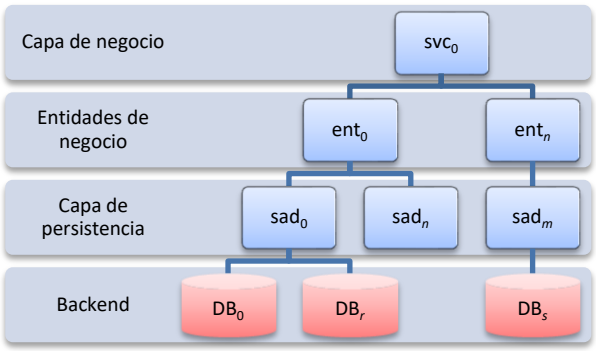


Ilustración 14 – Separación por capas vista desde la capa de servicios (adaptadores de servicio).

- Un servicio negocio (ya sea de aplicación o de dominio) maneja entidades de negocio a través de las API de otros servicios e, internamente, a través de las interfaces de los servicios de acceso a datos.
- Las implementaciones de los servicios de acceso a datos proporcionan las herramientas para el manejo programático de las entidades de negocio.
- Las implementaciones de los servicios de acceso a datos proporcionan la lógica de persistencia contra las bases de datos sobre las cuales esa entidad tendrá que persistirse.

3 Servicios de Negocio

3.1 Introducción

Los *servicios de negocio* son los componentes que encapsulan la lógica de negocio de las aplicaciones (conforman la implementación de la capa de negocio) y deben cumplir lo indicado en 2.2. Los servicios de negocio contendrán exclusivamente lógica de negocio y encapsularán convenientemente la funcionalidad de negocio en componentes reutilizables. Esta funcionalidad se puede exponer hacia otras aplicaciones gracias a los adaptadores de servicio de integración (conformando la capa de servicios). Los adaptadores de servicio llegan a ofrecer negocio a las demás aplicaciones consumiendo el negocio de la aplicación a través de un contrato expuesto por un adaptador. El servicio de Negocio consume servicios externos a través de un adaptador de salida que desacopla la complejidad tecnológica de la implementación de los clientes de servicio.

Por el rol que desempeñan, los servicios de negocio pueden invocar:

- Servicios de negocio de la misma aplicación (servicios de negocio de dominio) o de otra aplicación, a través del adaptador de salida (4.5).

- Servicios de acceso a datos de la aplicación a la que pertenecen.

Desde el departamento de arquitectura se recomienda estructurar la capa de negocio en 2 niveles de abstracción de servicios. Aquellos de más alto nivel (obligatorios de implementar) se llaman servicios de negocio de aplicación y la interfaz de estos servicios se denomina puerto de servicio. Los servicios más internos (caso de existir) se llaman servicios de negocio de dominio y se suelen implementar en base a las entidades de negocio internas de la aplicación (dominio interno).

Un servicio de negocio podrá entonces tener las dependencias ilustradas en la figura siguiente:

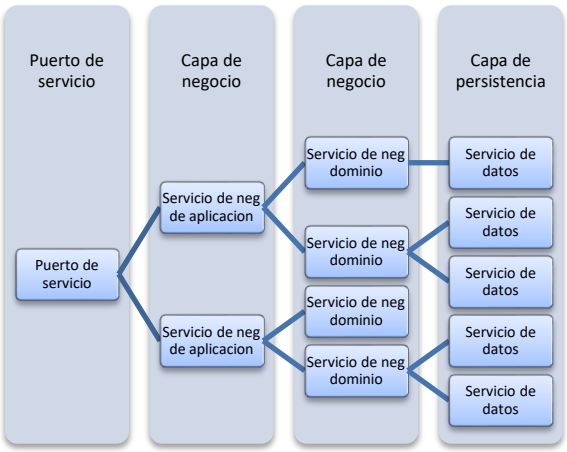


Ilustración 15 – Ejemplo de flujos de invocaciones entre los servicios de negocio y de datos a través del puerto de servicio y las distintas capas

Los servicios de negocio de aplicación (cuya interfaz se denomina puerto de servicio) pueden entonces invocar a otros servicios de negocio cuando se necesite utilizar una funcionalidad del negocio que ya ha sido encapsulada en un servicio para componer otro servicio cuyo flujo de ejecución se constituye de la orquestación de invocaciones hacia otros servicios de negocio. Los servicios de negocio pueden invocar indistintamente servicios de negocio de la misma aplicación o servicios de negocio publicados por otras aplicaciones (a través de los adaptadores de salida).

Los servicios de negocio utilizarán los servicios de acceso a datos para recuperar y escribir los datos que necesiten manejar y utilizarán exclusivamente las *interfaces de acceso a datos* cuando tengan que acceder a un backend de almacenamiento de datos. Un servicio de acceso a datos, como se verá más adelante en este documento, desacopla la lógica de negocio de la lógica necesaria para acceder a un backend de almacenamiento de datos y es un componente definido y utilizado internamente en las aplicaciones. Los servicios de negocio invocarán exclusivamente los servicios de acceso a datos publicados dentro de la misma aplicación y no tendrán siquiera visibilidad de los servicios de acceso a datos de aplicaciones externas.

Será responsabilidad del desarrollador el preocuparse de separar adecuadamente la lógica de negocio de la lógica de acceso a datos: la lógica de negocio se encapsulará en servicios de negocio mientras la lógica de acceso a datos se encapsulará en servicios de acceso a datos.

En el modelo de programación definido por la Dirección General de Tráfico; el puerto de servicio (o interfaz) de los servicios de negocio de aplicación es el punto de entrada de las invocaciones a lógica de negocio (capa de negocio). En calidad de punto de entrada para la ejecución de lógica de negocio, los servicios de negocio son los responsables de la gestión de las transacciones[16], en la situación donde esté previsto.

En la figura siguiente queda sintetizada la visibilidad de estos componentes.

[16] Los servicios de negocio serán responsables del ciclo de vida de la transacción y por lo tanto la podrán arrancar, terminar y asegurar que se ejecuten dentro de un contexto transaccional existente.

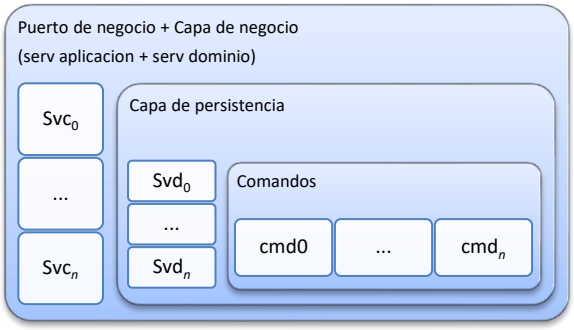


Ilustración 16 – Visibilidad entre los componentes

3.2 Estado conversacional de los servicios de negocio

Los servicios de negocio no deberán mantener estado conversacional: esto quiere decir que la clase correspondiente no tiene que mantener ningún tipo de estado entre invocaciones a un método. Esto significa que la clase que implementa un servicio de negocio deberá:

- No utilizar campos de instancia que no sean de sólo lectura y en ese caso deberán ser **`static`** y **`final`**.

- No almacenar datos en el *thread local storage*.

En consecuencia un servicio de negocio deberá implementarse como:

- Enterprise JavaBean como *stateless*, a través de la anotación **`@Stateless`** o a través del descriptor de despliegue.

Como se verá en los capítulos siguientes puede haber una excepción a este caso: un servicio de negocio que utilice servicios de acceso a datos implementados como componentes EJB de tipo *stateful* u otro servicio de negocio de otra aplicación que, por la misma razón, decida implementarse como componente EJB de tipo *stateful*.

El Departamento de Arquitectura de la Dirección General de Tráfico permite implementar servicios de negocios como componentes EJB de tipo *stateful* exclusivamente para poder liberar los recursos *stateful* utilizados durante su invocación. Este artificio técnico no autoriza al implementador a mantener estado dentro del componente que, en todo caso, deberá implementarse siguiendo las normas dictadas en esta especificación.

3.3 Interfaz de negocio

Los servicios de negocio deben definir y utilizar las *interfaces de negocio*, es decir, las interfaces a través de las cuales la lógica de negocio es presentada al exterior, sea el llamante un servicio de la misma aplicación (caso del puerto de servicio en los servicios de negocio de aplicación) o sea el servicio un componente a exportar fuera de la frontera de la aplicación que se está desarrollando (caso de los adaptadores de servicio): de esta manera se garantiza el encapsulamiento de las implementaciones de las interfaces de negocio en componentes bien determinados y se facilita la sustitución de implementaciones en el caso en que se presentase esta necesidad.

3.3.1 Requerimientos de los métodos de negocio

Un método de negocio tiene que cumplir las restricciones impuestas en la Enterprise JavaBeans™ Specification v. 3.0 (3), es decir[17]:

- El nombre del método no puede empezar con la cadena **`ejb`**.
- El método tiene que ser declarado **`public`**.
- El método no puede ser declarado **`abstract`**.
- El método no puede ser declarado **`final`**.
- Las clases utilizadas en la firma de los métodos de negocio[18] tienen que cumplir las restricciones del canal[19] sobre el cual se hará la invocación al EJB.

3.3.2 Requerimientos del interfaz de negocio

Las interfaces de negocio de un servicio de negocio están sujetas a los requerimientos de Arquitectura y a los requerimientos de la Enterprise JavaBeans™ v. 3.0 Specification (3) de las cuales recogemos las más importantes:

- No extiende **`javax.ejb.EJBObject`**.
- No extiende **`javax.ejb.EJBLocalObject`**.
- La interfaz NO puede declararse como remota. El departamento de Arquitectura no permite el uso de EJB remotos (excepto para aquellas aplicaciones legadas que mantengan dependencias de otras aplicaciones)

[17] Véase la JSR 220.
[18] Como parámetros y como valor de retorno.
[19] Este punto se detallará en las secciones siguientes.

- Puede lanzar excepciones aplicativas.

- Puede tener super interfaces.

Por imposición de la Enterprise JavaBeans™ v. 3.0 Specification (3) y restricciones del departamento de Arquitectura, las interfaces de negocio tienen que ser explícitamente declaradas interfaces de negocio del EJB y esto se puede cumplir de las siguientes maneras:

- A través de las anotaciones **`Local`** (usadas en el puerto de servicio de los servicios de aplicación y en la interface de los servicios de dominio, caso de existir**)**

- A través del descriptor de despliegue del EJB que se debe usar de igual forma que en las anotaciones.

Una representación esquemática de la estructura básica de un servicio de negocio está representada en la siguiente ilustración.

3.3.3 Interfaces locales y remotas

Los Enterprise JavaBeans™ soportan invocaciones *locales* y *remotas*. El departamento de Arquitectura DGT, NO permite la definición de interfaces remotas (excepto para aquellas aplicaciones legadas que mantengan dependencias de otras aplicaciones)

3.4 Inyección de dependencias

Como se ha introducido, un servicio de negocio puede invocar servicios de negocio y servicios de acceso a datos. Estas invocaciones crean una relación de dependencia entre el objeto llamante y los objetos invocados que debe ser resuelta por quien implementa el servicio de negocio.

El modelo de programación de la Dirección General de Tráfico impone que estas relaciones de dependencia se resuelvan a través del mecanismo de inyección de dependencia definido por la especificación EJB v. 3.0 (3). En general será entonces suficiente anotar los campos del EJB en los cuales se desee inyectar los beans dependientes con la oportuna anotación[20]:

[20] La especificación Java EE v. 5 define más de una anotación para inyectar como dependencias recursos gestionados por el servidor de aplicaciones como Data Sources, colas y tópicos JMS, Enterprise JavaBeans™, etc.

```
package es.trafico.XXXX.neg.matriculacion.beans;

import javax.annotation.Resource;
import javax.ejb.EJB;
import javax.ejb.EJBContext;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class MatriculacionBean
  implements MatriculacionLocal {

  @EJB
  private MatriculacionPlacaLocal matriculacionPlacaBean;
  @EJB
  private VehiculoInfoLocal vehiculoInfoBean;
  @Resource
  EJBContext ctx;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void matricular() {
    // lógica de negocio
  }
```

Código 1 – Inyección de dependencia con los metadatos de la plataforma Java™ EE v. 5

La especificación Java EE v. 5 (11), la especificación Enterprise JavaBeans™ v. 3.0 (3) y la especificación Common Annotations for the Java™ Platform (12) describen en detalle las anotaciones en cuestión.

3.4.1 Ejemplo: inyección de un servicio de negocio

El siguiente fragmento de código ejemplifica como inyectar una referencia a un EJB dentro de otro componente EJB:

```
package es.trafico.servicios.arquitectura.matriculacion.beans;

import javax.ejb.EJB;
import javax.ejb.Stateless;

@Stateless
public class MatriculacionBean
  implements MatriculacionRemote {

  @EJB
  private MatriculacionPlacaLocal matriculacionPlacaBean;
```

Código 2 – Inyección de un servicio de negocio

La Enterprise JavaBeans™ 3.0 Specification (3) define la anotación **`@EJB`** para inyectar referencias a interfaces de componentes EJB. En el caso más sencillo se puede utilizar la anotación EJB para inyectar una interfaz: el servidor de aplicaciones inyectará el único componente conocido que implemente esta

interfaz. En el caso en que haya más componentes que implementen la interfaz requerida será necesario proporcionar informaciones adicionales a través de los atributos de la anotación **`@EJB`**[21].

[21] O, alternativamente, de la oportuna sección de configuración en el descriptor de despliegue.

4 Patrones para el diseño y la implementación de servicios de negocio

El Departamento de Arquitectura de la Dirección General de Tráfico incluye en esta especificación unos patrones y un modelo de programación para que sirvan de norma y referencia durante el desarrollo de las aplicaciones Java™ para la DGT.

4.1 Acoplamiento entre aplicaciones e inestabilidad de los contratos

Las interfaces son herramientas conceptuales y tecnológicas que, a lo largo de la vida de una aplicación, favorecen el desacoplamiento entre implementaciones distintas de un mismo *contrato*. El único acoplamiento que queda es el acoplamiento con la interfaz misma: por esta razón el desacoplamiento de las implementaciones y la posibilidad de intercambiarlas se funda en la *inmutabilidad* de los contratos.

Durante el diseño habrá que tomar las medidas que sean necesarias para que la definición de una interfaz de negocio sea efectivamente inmutable. Debido a que las interfaces de este tipo representan características funcionales del sistema no pueden darse soluciones tecnológicas que sean válidas en todo caso. Si la lógica de negocio de una aplicación cambia, pueden darse los siguientes escenarios:

- Los contratos de los servicios afectados cambian de manera acorde y habrá que notificar, actualizar y adaptar todos los servicios que dependan de las interfaces modificadas.

- Es posible mantener inalteradas las interfaces de negocios y trasladar la complejidad añadida en las implementaciones subyacentes.

Esta especificación impone a las aplicaciones hacer las implementaciones que sean necesarias para mantener inalteradas las interfaces de los servicios que exponen al exterior la lógica de negocio.

4.2 Resolución de las interfaces de negocio. Granularidad de las interfaces (servicios de negocio de aplicación y de dominio).

Un problema recurrente cuando se tiene que diseñar interfaces de negocio[22] es decidir la *resolución* de los métodos que se van a crear. Por un lado exportar públicamente métodos que desempeñan una gran cantidad de lógica de negocio aporta como ventaja:

[22] Este problema se amplifica cuando los servicios en cuestión proporcionan datos a través de su API.

- La reducción de invocaciones al servicio.
- La simplificación de escenarios como el control de transacciones entre múltiples invocaciones.
- La reducción del *overhead* debido a múltiples invocaciones.
- Los problemas consecuencia de un grano demasiado fino de los métodos de negocio.

Las dos soluciones tienen que ser complementarias: por un lado hay que exportar cada método de negocio que el análisis funcional detecte porque *por definición* será parte del contrato de un componente. Por el otro lado la fase de análisis ayudará a detectar los escenarios en los cuales la publicación de un método de grano más grueso sea conveniente para quien utilice el contrato de negocio.

En esos casos es aconsejable recubrir las invocaciones recurrentes con un único método del mismo servicio para que quien utilice ese contrato pueda tomar la decisión de reducir el número de invocaciones, teniendo en cuenta que este recubrimiento tiene que ser una solución no solo técnicamente apropiada sino también *funcionalmente* apropiada. Como queda ilustrado sintéticamente en la figura siguiente, los beneficios derivados de la consolidación de múltiples servicios en uno con resolución más gruesa serán los siguientes:

- El contrato de negocio será claro y exhaustivo e incluirá los parámetros requeridos por los servicios que consolida.
- Se reducirá el número de invocaciones remotas y el overhead asociado al paso por parámetro de los argumentos a los métodos de negocio.
- La transacción eventualmente involucrada en estas invocaciones estará abierta un tiempo más reducido.
- Las transacciones potencialmente distribuidas podrán iniciarse y localizarse en un único método de negocio.
- Cuando sea posible se podrá deducir la topología de despliegue de componentes más apropiada, teniendo en cuenta las relaciones entre *beans*, permitiendo concentrar localmente *beans* con una elevada afinidad entre ellos.

4.3 Cantidad de datos devuelta por método

Por razones similares a las que se detallan en la sección precedente, durante el análisis y el diseño de las interfaces de negocio, especialmente de la interfaces de tipo remoto, es fundamental tener en cuenta la cantidad de datos máxima que se puede transportar durante una invocación.

Si por un lado el diseño de las interfaces de negocio debe ser una tarea ortogonal al diseño de la solución técnica más apropiada para solucionar las necesidades de un componente, por el otro, en ciertos

casos, habrá que tener en cuenta las idiosincrasias de la solución funcional propuesta para encontrar el balance entre requerimientos funcionales y técnicos del componente a desarrollar. Un caso conocido a priori por el Departamento de Arquitectura al que hay que prestar la apropiada atención es la devolución de datos por parte de los servicios de negocio[23].

Se puede dar a priori el caso en que un método de negocio pueda devolver una cantidad de información muy grande o incluso *desconocida*. Los casos en los que más comúnmente pueda presentarse este problema son los servicios de búsqueda de información. En este caso podrían darse las siguientes situaciones:

- Si el modelo de datos es muy articulado una *entidad* puede arrastrar consigo todas sus relaciones. Si un método de negocio devuelve una entidad **`X`** sería razonable pensar que toda entidad en relación con ella fuera accesible a través de la instancia de **`X`**.

- Si el método de negocio devuelve un conjunto de entidades de tamaño desconocido[24].

Estos problemas corroboran que la propagación de estructuras pertenecientes a la capa de datos a través de otras capas está generalmente desaconsejada. Si por un lado se genera acoplamiento entre capas e incluso aplicaciones distintas, por el otro lado se tendría que solucionar el "problema" de la resolución de las relaciones entre entidades.

4.3.1 Partición de la información

En el caso en que un servicio maneje estructuras de datos muy grandes, el desarrollador deberá tomar medidas oportunas para que la ejecución del servicio sea ágil y no se devuelva información innecesariamente grande.

Si por un lado la práctica de encapsulación para aumentar la reutilización de código impondría reutilizar las mismas estructuras de datos donde sea posible, por el otro lado se darán casos en que una estructura de datos referencie mucha más información de la que un método de negocio debería devolver.

Las opciones sugeridas en estos casos son:

- Definición de un tipo de datos ad hoc: esta solución se puede enmarcar dentro de un patrón comúnmente llamado *Value Object* o *Data Transfer Object*.

- Recuperación tardía de los datos requeridos. En lugar de poblar una estructura completa de datos[25] un servicio:

    - Devolverá una estructura parcialmente poblada[26].

[23] Esta consideración afecta igualmente a los servicios de acceso a datos.
[24] Es el caso, por ejemplo, de las búsquedas.
[25] Y todas las estructuras por ella referenciadas.

- Devolverá un resultado, un *token*, a través del cual sea posible rellenar la estructura devuelta poblándola con la información que se requiere.

Nótese que se ha excluido voluntariamente el caso en que las entidades se carguen de manera "perezosa". La razón de esta exclusión es que un requerimiento para que ese caso de uso fuera viable sería que la entidad estuviera todavía conectada a la base de datos. Esta asunción está prohibida en la Arquitectura de persistencia de la Dirección General de Tráfico.

La opción de carga ad hoc de datos se aconseja sobre todo en los casos en que la estructura de datos a devolver es muy grande y no se puede prever razonablemente en qué información un cliente podría estar interesado. En ese caso, entonces, se sugiere proporcionar un objeto que actúe de *token*, es decir, que contenga la información necesaria para poder recuperar datos adicionales *a posteriori*. En el caso más genérico este objeto contendrá las claves necesarias para recuperar la entidad en cuestión de la base de datos.

4.3.1.1 Ejemplo

Si por ejemplo un objeto de negocio de tipo X contiene n propiedades, un método de negocio podría querer poblar sólo un subconjunto m de ellas. El método de negocio podrá entonces devolver una instancia de X y, si el cliente de la invocación necesita recuperar el valor de la propiedad p_i deberá invocar un método ad hoc pasando por parámetro el objeto de negocio e indicando que necesita recuperar la propiedad p_i.

4.3.2 Devolución de jerarquía de entidades

El concepto de *relación* entre entidades es un concepto que debería estar restringido a la capa de persistencia y no propagarse hasta la capa de los servicios de negocio: las estructuras de datos manejadas por los servicios deberán surgir de las necesidades funcionales del servicio y no de las características técnicas de las soluciones utilizadas en esa o en otras capas.

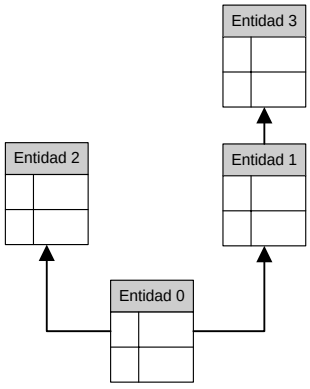


Ilustración 17 – Entidades relacionadas

[26] Cuando los requerimientos del negocio lo permitan.

Si, por ejemplo, un servicio quisiera devolver la entidad 0 ilustrada en la figura precedente (Ilustración 17), habría que decidir qué hacer con cada una de las relaciones que esta entidad establece con otras. La interfaz de un servicio, así como establece sus parámetros de entrada, también define el *parámetro de salida*. En casos parecidos a estos no sería funcionalmente correcto arrastrar las relaciones y devolver al cliente estructuras de datos no previstas por el negocio. Será funcionalmente más correcto y homogéneo recoger en otra estructura de datos (una entidad de *negocio tipo DTO*) los datos significativos para el negocio. La procedencia de estos datos no será significativa a la hora de diseñar la interfaz y las entidades de negocio: aunque todos procedan de un backend de almacenamiento de datos.

En otros casos un servicio de negocio podrá proporcionar funcionalidades de *búsqueda*. En algunos tipos de búsquedas puede desconocerse la cantidad de información que satisface la condición de filtro proporcionada al buscador: en estos casos no es viable devolver *todos* los resultados encontrados sino que será necesario *paginar* adecuadamente el servicio afectado por este problema.

Para gestionar la criticidad de casos como estos, que deben emerger en fase de análisis funcional, es necesario seguir las siguientes indicaciones:

- Encapsular los datos devueltos por el negocio en oportunas entidades de negocio, desvinculándolas de las entidades JPA (3) que eventualmente se estén utilizando para recuperar dichos datos.

- Evaluar y controlar la cantidad máxima de información que un método pueda devolver.

- Proporcionar un mecanismo para que sea posible obtener la información en *páginas* en el caso en que el control de la cantidad de registros devueltos no sea una solución funcionalmente viable.

- Proporcionar exclusivamente la información prevista por el contrato de negocio y evitar llenar estructuras de datos considerablemente más grandes. En el caso en que el grafo de entidades relacionadas entre sí sea muy grande, considerar patrones alternativos como DTO.

- Generalizando las consideraciones del punto anterior, en el caso en que se manejen estructuras de datos muy grandes, proporcionar un mecanismo a través del cual sea posible recuperar exclusivamente la información que se desea.

4.3.3 Paginación

Esta especificación no define un patrón para implementar la *paginación* y se limita a sugerir buenas prácticas para que las implementaciones que se produzcan respondan a los parámetros de calidad definidos por el Departamento de Arquitectura.

La paginación es una de las técnicas con la cual se pueden resolver los problemas expuestos en la sección § 4.3. Muchas aplicaciones tienen como requerimiento la necesidad de buscar y sucesivamente analizar la lista de resultados de la búsqueda. Buscar aplicando un filtro a una lista de datos es una ac-

tividad a diseñarse teniendo en cuenta las características del conjunto de datos sobre el cual se va a efectuar la búsqueda. Las características a tener en cuenta mayormente son las siguientes:

- Tamaño del conjunto de datos.
- Gradiente de crecimiento del conjunto de datos.
- Requerimientos del cliente durante el acceso a los datos paginados: dirección de lectura (unidireccional, bidireccional), soporte para el acceso directo a páginas o registros concretos, etc.

En este capítulo se darán las indicaciones que apliquen a los servicios de negocio que, al no manejar directamente los sistemas de almacenamiento de datos, solo tienen que asumirse la responsabilidad de proporcionar un contrato de negocio válido a sus clientes.

En los casos en los cuales pueda presentarse este tipo de problema, el Departamento de Arquitectura de la Dirección General de Tráfico sugiere implementar la funcionalidad de paginación de datos y diseñar el contrato de negocio afectado para que, semánticamente, proporcione esta funcionalidad. Un ejemplo del interfaz que podría implementar el objeto que efectúa la paginación de una lista de datos podría ser el siguiente:

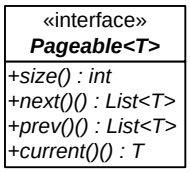


Ilustración 18 – Interfaz para paginación de datos

En este ejemplo, desde el punto de vista del cliente, una lista paginada será un *iterador* que otorga acceso a una lista de registros en sólo lectura. El iterador hará las veces de *proxy* hacía los componentes que permitirán recuperar los datos en páginas, como los DAOs y los servicios de acceso a datos que garantizan la encapsulación y la reutilización de la lógica de acceso a datos.

El iterador podrá ser local o remoto, según las necesidades de la aplicación:

- Localmente se podrán mantener las informaciones necesarias para recuperar una página de datos cualquiera.
- Remotamente en el caso en que el estado de la iteración se mantenga en otra capa y el cliente disponga de una funcionalidad limitada tal como *avanza* o *retrocede* de página.

4.4 Manejo de múltiples backends

Para facilitar la migración es probable que una aplicación utilice en paralelo más de un backend por un periodo transitorio de tiempo. Los patrones descritos en esta especificación facilitan la implemen-

tación de lógica de este tipo y pueden aislar completamente la aplicación durante la migración de backends.

Una interfaz de negocio, por definición, solo expone el contrato funcional establecido por el servicio en cuestión y utilizará a su vez otras interfaces de negocio para desempeñar su trabajo. Desde el punto de vista de la implementación esto se realiza en una implementación de interfaces de negocio que a su vez utiliza otras instancias de interfaces de negocio[27] que se resolverán a través de un mecanismo de inyección de dependencia oportunamente configurable.

Las soluciones que se adopten para solucionar este tipo de problemas no deberán afectar a la implementación del servicio de negocio, el cual siempre se limitará a invocar una instancia de *servicio de acceso a datos*. La complejidad añadida para dar soporte a escenarios tales como el soporte de múltiples backends en paralelo se delegará a la capa de acceso a datos que, en cada caso, escogerá la solución más apropiada para solucionar el problema manteniendo desacopladas la capa de los servicios de negocio de la capa de acceso a datos.

4.5 Invocación de servicios externos a través de adaptadores de servicio de salida

La invocación desde una aplicación A que intenta consumir un servicio a otra aplicación B que lo ofrece, se deberá realizar a través de un servicio de negocio que deberá poseer una interfaz que contenga únicamente aquellas operaciones involucradas en la llamada a dicho servicio externo. La implementación de dicho servicio será la encargada de realizar las invocaciones al cliente del servicio remoto, aislando por tanto su invocación del resto del negocio de la aplicación. A este patrón se le conoce conceptualmente como un adaptador de servicio de **salida** (distinguiéndose del resto de adaptadores comentados en esta especificación, como los adaptadores de integración o de presentación, cuya misión es ser la **entrada** de invocaciones externas).

Debe quedar claro que bajo este patrón se invocará cualquier servicio externo exceptuando aquellos que deben implementarse bajo servicios de acceso a datos, como es el caso del acceso a la persistencia relacional (Oracle) o al Host, tanto de forma directa como a través de plataformas de servicios de acceso a datos. Por tanto se deberá construir un adaptador de servicio externo para la invocación de cualquier servicio ofrecido por un componente común a través de su cliente, la invocación a servicios externos de otras aplicaciones, el acceso a Ldap, etc.

Gracias al aislamiento que ofrecen estos adaptadores de servicio de salida se podrá reemplazar fácilmente la implementación de la llamada a un servicio externo por una implementación mock sin que se altere la programación del resto de la aplicación.

[27] O de servicios de acceso a datos

5 Control transaccional de los servicios de negocio

Los servicios de negocio son los puntos de entrada de las invocaciones a operaciones de negocio y, como se ha explicado, pueden requerir la consideración de *operaciones atómicas*. Por esta razón los servicios de negocio son puntos clave en la cadena de invocación en los cuales puede ser necesario garantizar que las operaciones ejecutadas de ahí en adelante se ejecuten de manera atómica: es decir, o todas las operaciones se ejecutan con éxito o ninguna operación tiene que ejecutarse. Eventuales *efectos* generados por operaciones que ya se hayan ejecutado al momento del fallo tendrán que deshacerse para volver a la situación que se presentaba antes de la invocación del servicio de negocio.

La plataforma Java™ EE utiliza un API, la API JTA (2), para solventar el problema del control de las transacciones para un subconjunto de recursos que puedan ser manejados por dicha API. Recursos que puedan manejarse de forma transaccional a través del API JTA son por ejemplo:

- Conexiones a bases de datos.
- Colas de mensajes JMS.

5.1 Recursos no transaccionales

Durante el desarrollo de aplicaciones Java™ para la Dirección General de Tráfico puede darse la situación en que un componente tenga que manejar un recurso que no pueda ser gestionado a través de un gestor de transacciones JTA[28]. Para organizar la gestión de estos recursos esta especificación propone un patrón llamado *Command*. Los servicios de negocio serán responsables de poder propagar adecuadamente la notificación de *rollback* a todo servicio que está participando en el flujo de invocaciones. La responsabilidad de actuar de manera adecuada a fin de garantizar el *rollback* de las operaciones efectuadas está a cargo de cada componente que implemente el patrón *Command*[29]. El acceso a los recursos se hará a nivel de servicio de acceso a datos y por lo tanto en los servicios de negocio no hay complejidad añadida por este factor.

5.1.1 Ordenación y posicionamiento

Al manejar recursos transaccionales y no transaccionales es aconsejable tomar medidas adicionales a la hora de implementar un servicio. Cuando sea posible, Arquitectura aconseja:

[28] Un ejemplo es una conexión a host establecida por el conector JCA actualmente en uso en la Dirección General de Tráfico.
[29] El patrón Command se detallará en el capítulo Servicios de acceso a datos.

- Intentar agrupar el manejo de recursos transaccionales y no transaccionales: si se manejan *n* recursos transaccionales y *m* recursos no transaccionales, averiguar si es posible agrupar las *n* operaciones y las *m* operaciones en dos grupos que no se intersecten e invocar las *n* operaciones transaccionales primero. De esta manera, si falla una cualquiera de las *n* operaciones transaccionales será el coordinador de recursos quien haga rollback de la transacción sin involucrar ningún flujo de compensación.

5.2 Modelo declarativo

Con el modelo declarativo la gestión de las transacciones es una responsabilidad del contenedor y esto se traduce en que el desarrollador no tiene que escribir código para hacer el *commit* o el *rollback* de una transacción que utilice recursos transaccionales. Las responsabilidades del desarrollador en este modelo son:

- Informar al contenedor sobre cómo se tiene que manejar una transacción para un componente dado.

- Marcar la transacción para el rollback cuando se producen errores que no se pueden gestionar.

De acuerdo con lo que establece el estándar EJB v. 3.0 (3) se permite al desarrollador, controlando excepciones aplicativas, decidir si quiere seguir ejecutando la transacción o si prefiere marcarla para el *rollback*. De esta manera se pueden diseñar transacciones flexibles donde pueda preverse el caso en que una operación pueda generar errores aplicativos como validaciones de parámetros o control de reglas de negocio y seguir por rutas alternativas en la ejecución de la transacción.

La configuración de las transacciones para los componentes se efectuará de acuerdo al *contenedor* que se esté utilizando. La Dirección General de Tráfico ha establecido que se utilizarán los Enterprise JavaBeans™ v. 3.0.

5.2.1 Atributos de control de transacciones

El modelo declarativo permite declarar el nivel de control de transacción requerido hasta el nivel de método de negocio. Los atributos de control de transacción que se pueden utilizar con el modelo declarativo son los siguientes:

- **`REQUIRED`**: Una transacción es requerida: si existe una transacción en curso se utilizará, si no existe el contenedor creará una nueva.

- **`MANDATORY`**: Una transacción existente es requerida: si no existe el contenedor lanzará una excepción.

- **`REQUIRES_NEW`**: Una transacción nueva es requerida: el contenedor iniciará siempre una nueva transacción suspendiendo una transacción existente, si ella existe.

- **SUPPORTS**: Una transacción no es requerida: si existe una transacción en curso se utilizará, si no existe el método se ejecutará fuera de un contexto transaccional.

- **NOT_SUPPORTED**: Una transacción no es requerida: si existe una transacción en curso se suspenderá durante la ejecución del método.

- **NEVER**: La transacción no es soportada y el método no se puede ejecutar dentro de una transacción.

5.2.2 Utilización del atributo de control de transacción

Arquitectura ha definido el patrón para la utilización del atributo de control de transacción en los servicios de negocios y en los servicios de acceso a datos.

El punto de entrada de las invocaciones, que normalmente será un método de negocio publicado en un componente (un servicio de negocio) a través de una interfaz de negocio, será el punto en que se declare la necesidad de ejecutar la lógica dentro de una transacción. Normalmente esto se hará a través del atributo **REQUIRED** que, en el caso en que una transacción no esté arrancada, instruirá al contenedor para arrancar una transacción nueva.

Todos los métodos que por diseño no son métodos de negocio y por lo tanto no actúan como punto de entrada de las invocaciones a los componentes de negocio tendrán que estar marcados con:

- **SUPPORTS**: en el caso en que dicho método ejecute lógica que no necesariamente tenga que ejecutarse dentro de una transacción, como actividades de lectura de una base de datos.

- **MANDATORY**: en el caso en que dicho método necesite ejecutarse dentro de una transacción. Si una transacción no estuviera en curso el contenedor lanzará un error que se deberá considerar un bug.

5.3 Configuración del contenedor EJB v. 3.0

La configuración del contenedor dependerá del contenedor que se está utilizando. En el caso de los Enterprise JavaBeans™ v. 3.0, el modelo declarativo adquiere el nombre de *Container-Managed Transactions* y la configuración se efectúa en el descriptor de despliegue de los Enterprise JavaBeans™.

El contenedor EJB v. 3.0 puede ser configurado para manejar las transacciones de un EJB a través del nodo **<transaction-type>** del descriptor de despliegue del módulo EJB asignándole el valor **Container**. Este valor es el valor utilizado por defecto en la mayoría de los contenedores EJB.

Para indicar que un EJB utiliza este modelo de control de transacción a través de metadatos, se puede anotar la clase del bean con la anotación **TransactionManagement** asignándole el valor **TransactionManagementType.CONTAINER** como ilustrado en el siguiente ejemplo:

```
@Stateless
@TransactionManagement{TransactionManagementType.CONTAINER}
public class MiServicioBean
    implements IMiServicio {
  [...]
}
```

Código 3 – Indicar el modelo de control de transacción utilizado por un Enterprise JavaBean™

5.4 Marcar una transacción para el *rollback*

Por uniformidad con la especificación EJB v. 3.0, el código puede ejecutarse dentro de una transacción y controlar cuándo puede continuar ejecutándose y decidir si el error es irrecuperable, y entonces marcar la transacción para el *rollback*, o cuándo el error es un error aplicativo previsto en el algoritmo de ejecución, y entonces seguir ejecutando la transacción.

Para marcar una transacción para el *rollback* en el modelo de control de transacción declarativa se utiliza el método **`setRollbackOnly()`** del interfaz **`EJBContext`** que puede adquirirse con inyección de dependencia como en el ejemplo siguiente:

```
@Stateless
public class MiServicioBean
  implements IMiServicio {
  @Resource
  SessionContext context;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void MiMetodo() {
    try {
      [...]
    } catch (Exception ex) {
      context.setRollbackOnly();
      throw ex;
    }
  }
}
```

Código 4 – Marcar una transacción para el rollback a través del contexto de sesión

La responsabilidad de hacer el *rollback* de una transacción, en el modelo de programación definido por la Dirección General de Tráfico, generalmente será del método que ha arrancado la transacción, es decir, del método de negocio publicado en la interfaz de negocio del servicio de negocio.

En lugar de utilizar directamente la instancia de **`SessionContext`** inyectada en el componente EJB se puede marcar apropiadamente una *excepción aplicativa* para que la transacción se marque para el *rollback* cuando dicha excepción sea lanzada por un componente. La aplicación de esta meta-información se puede hacer a través de anotaciones o de las oportunas secciones del descriptor de despliegue.

```
import javax.ejb.ApplicationException;

@ApplicationException(rollback=true)
public class MyServiceException extends Exception {
  // implementación
}
```

Código 5 – Marcar una excepción como excepción de aplicación que marque la transacción para el rollback

La excepción del ejemplo precedente es marcada con la anotación **`@ApplicationException`**. Esta excepción será parte del método de negocio de un EJB y será propagada al cliente cuando se produzca. Además, ya que ha sido marcada con el atributo **`rollback`** de la anotación **`@ApplicationException`** con valor **`true`**, esta excepción marcará automáticamente la transacción para el rollback.

Como ha quedado matizado anteriormente el Departamento de Arquitectura de la Dirección General de Tráfico aconseja diseñar los componentes de manera que sea quien inicia el flujo quien tenga la responsabilidad de marcar una transacción para el *rollback*. Un componente que lance una excepción del tipo de la excepción ilustrada en Código 5, invocado dentro de un flujo de negocio, no dejará la posibilidad a quien inicia el flujo de ignorar la excepción y seguir adelante con el flujo de negocio y por lo tanto podría dar lugar en un componente EJB menos flexible a la hora de utilizarse, sobre todo cuando el componente sea de tipo remoto y por lo tanto reutilizable por las demás aplicaciones en un flujo no previsible por quien ha desarrollado el componente.

5.5 Gestión de las excepciones

Como se ha introducido en las secciones anteriores, las excepciones aplicativas lanzadas durante la ejecución de los métodos de negocio pueden, en algunos casos, no marcar la transacción para el *rollback*.

Las excepciones aplicativas son las excepciones de tipo *checked*, es decir, las que no heredan de **`RuntimeException`** y en principio pertenecen al API del servicio porque tienen que ser declaradas. El lanzamiento de una excepción de este tipo no marca la transacción para el *rollback* y esta situación debe considerarse parte de la lógica de negocio que se está desarrollando.

Debido al modelo de programación y al modelo transaccional que se ha elegido, hay que seguir este patrón:

El control de la transacción debería ejecutarse dentro del método que ha arrancado la transacción. Por lo tanto debería ser el método que ha arrancado la transacción, quién marque la transacción para el rollback.

Por esta razón, cuando se organiza la aplicación en componentes tales como los servicios de negocio y los servicios de acceso a datos, se debe tener cuidado en diseñar el API de los servicios para que contengan las excepciones que pueden lanzarse en tiempo de ejecución y que las mismas tengan la información suficiente para que la lógica llamante pueda decidir si marcar la transacción para el *rollback* o si se puede seguir.

Esta especificación prohíbe explícitamente la práctica de envolver una excepción aplicativa y lanzar una excepción de tipo *unchecked* como **`EJBException`** para evitar marcar una transacción para el rollback.

5.6 Servicios de negocio asíncronos

Las interfaces de negocio podrán definir métodos cuya invocación sea asíncrona. En esta revisión de la especificación sólo se permite declarar si se utiliza un servicio de negocio de forma síncrona o asíncrona exclusivamente con la resolución de interfaz.

La configuración de una llamada asíncrona se efectuará durante la configuración de las dependencias entre servicios: cuando un servicio *a* declare una dependencia con el servicio *b* a través de la interfaz de negocio i_n, se podrá declarar que se utiliza la interfaz i_n de forma asíncrona y las llamadas se efectuarán de acuerdo a como se ha configurado el mecanismo de invocaciones asíncronas, que es un mecanismo ortogonal a los servicios de negocio.

Arquitectura prevé la utilización de llamadas asíncronas entre servicios de negocio.

Los métodos asíncronos, no difieren de los demás métodos y están sujetos a los mismos requisitos detallados anteriormente: el conjunto de tipos de datos que pueden utilizar está restringido a los tipos de datos aprobados por Arquitectura (y las agregaciones permitidas). La invocación asíncrona podrá ser una característica de la invocación o del servicio en sí:

- Será una característica de la invocación, si es una invocación en particular la que se configura para ser una invocación asíncrona.

- Será una característica del servicio en sí, si es el servicio quién es intrínsecamente asíncrono.

5.6.1 Servicios intrínsecamente asíncronos

Un servicio puede ser asíncrono por naturaleza. En este caso será decisión de quien lo desarrolla, y no de quien lo invoca, que la invocación se ejecute como invocación asíncrona.

Evidentemente quién proporcione una interfaz de negocio asíncrona deberá responsabilizarse de proporcionar la implementación para ejecutar la invocación asíncrona. La Dirección General de Tráfico dará soporte a invocaciones asíncronas utilizando la API de mensajería JMS.

Quién proporcione un servicio de negocio asíncrono deberá proporcionar entonces un EJB 3.0 para consumir el mensaje recibido y procesar el mensaje que el cliente envíe al servicio. El modelo de programación elegido por la Dirección General de Tráfico establece que:

- Los mensajes a recibirse por un servicio asíncrono sean mensajes sobre un dominio Point to Point.

- La configuración de la cola para consumir los mensajes es responsabilidad de quien implementa el servicio.

- La oportuna documentación de los parámetros de configuración para enviar mensajes a dicha cola es responsabilidad de quien implementa el servicio.

- El tipo de mensaje a enviar y consumir deberá preferiblemente ser un **`ObjectMessage`**.

6 Servicios de acceso a datos

6.1 Principios fundamentales

Los *servicios de acceso a datos* son la tipología de servicios cuyo objetivo es la obtención y el manejo de los datos. Estos servicios son utilizados por los servicios de negocio y dentro de la misma aplicación, y podrían ser usados también por otros servicios de datos de la misma aplicación; no obstante, esto último dependerá del tipo de patrón de diseño que se use para la construcción de la capa de persistencia.

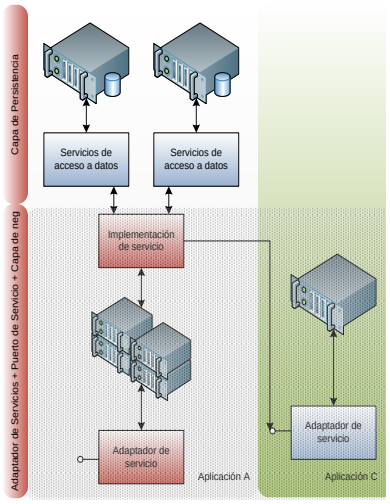


Ilustración 19 – Capa de servicio de acceso a datos siendo usada por la capa de negocio.

Los servicios de acceso a datos son *realizaciones* de una interfaz de negocio que se caracteriza por manipular datos: ninguna lógica de otro tipo se permite incluir en un servicio de acceso a datos.

El objetivo fundamental que ha inspirado la definición del concepto de servicio de acceso a datos es la necesidad de separar la lógica de negocio de la lógica necesaria al manejo del conjunto de backends de almacenamiento de datos utilizados por una aplicación. Así como los servicios de negocio son los componentes en los cuales se encapsula la lógica de negocio (conformando la capa de negocio de las aplicaciones), los componentes de acceso a datos son los componentes a los que se delega la responsabilidad de interaccionar con el sistema de almacenamiento de datos que le corresponde. Por tanto, se siguen los principios del patrón de capas (*layers*).

Escenarios tales como el ilustrado en la figura siguiente no podrán producirse con el adecuado encapsulamiento de la lógica de acceso a datos en oportunos componentes.

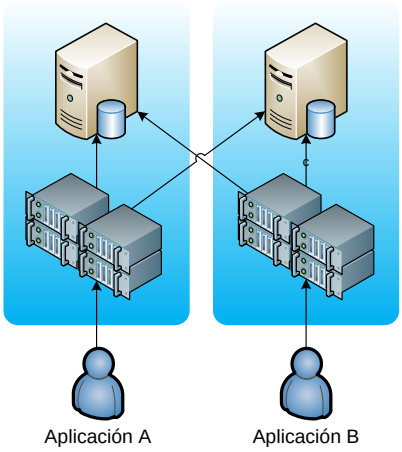


Ilustración 20 – Interacción prohibida entre aplicaciones y sistemas de almacenamiento de datos. Violación del principio de encapsulamiento.

Cada aplicación utilizará un conjunto de servicios de acceso a datos que serán los actores que interactuarán con cada uno de los sistemas de almacenamientos de datos utilizados por la aplicación, como queda ilustrado en la Ilustración 19.

El escenario planteado en Ilustración 20 será entonces sustituido por el escenario siguiente:

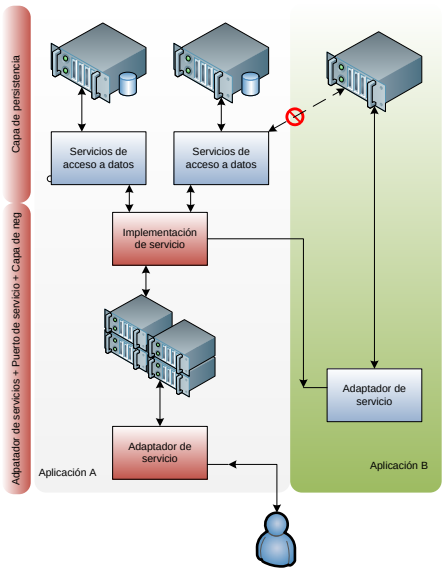


Ilustración 21 – Aislamiento de los sistemas de almacenamiento de datos entre aplicaciones.

En este escenario, cada uno de los sistemas de almacenamiento de datos se podrá alcanzar solo a través de un oportuno servicio de acceso a datos. Las aplicaciones, además, no podrán invocar directamente servicios de acceso a datos de otras aplicaciones y podrán acceder exclusivamente a la funcio-

nalidad publicada a través de los puertos de servicio de los servicios de negocio (vía un adaptador de servicio de integración).

Las aplicaciones, así como reutilizan los flujos de lógica de negocio implementados en los servicios de negocio, podrán reutilizar las implementaciones de los servicios de acceso a datos que, en este escenario, actúan como DAO[30] o repositorios[31]. Se debe tener en cuenta que la DGT define la capa de persistencia bajo los paquetes *es.trafico.ACRONIMO.dao* pero esto no condiciona que el patrón de diseño usado para implementar la capa de persistencia tenga que ser un DAO.

Los servicios de acceso a datos utilizarán las *interfaces de acceso a datos* para desacoplarse de los servicios de negocio.

Los servicios de acceso a datos tendrán la responsabilidad de manejar los recursos no transaccionales a través del manejo de instancias que utilicen el patrón Command (4).

Un servicio de acceso a datos solo puede invocarse desde servicios de negocio. La invocación de servicios de acceso a datos desde otros componentes está prohibida porque no se garantizaría la ejecución de los mismos bajo los contextos adecuados[32].

6.2 Características de un servicio de acceso a datos

Un servicio de acceso a datos, en el modelo de programación para aplicaciones Java™ de la Dirección General de Tráfico, será un Enterprise JavaBean™ de sesión y por lo tanto tendrá las mismas características descritas en los capítulos precedentes (§ **¡Error! No se encuentra el origen de la referencia.**) con las siguientes diferencias:

- Es un bean de tipo *stateless* si maneja exclusivamente recursos transaccionales.
- Es un bean de tipo *stateful* si maneja también recursos no transaccionales.
- Extiende la clase abstracta **`EJBCommandSupport`** si maneja recursos no transaccionales y necesita flujos de compensación.

La clase abstracta `EJBCommandSupport` es una clase de utilidad proporcionada por el Departamento de Arquitectura como ayuda a los equipos de desarrollo.

[30] Data Access Objects
[31] Repository pattern: http://martinfowler.com/eaaCatalog/repository.html.
[32] Se puede garantizar la existencia del contexto transaccional, por ejemplo, porque este ha sido creado por un servicio de negocio y propagado hacia el servicio de acceso a datos, que lo utiliza para manejar sus recursos transaccionales dentro de una transacción.

6.3 Interfaz de acceso a datos

Cada servicio de acceso a datos implementa la *interfaz de acceso a datos* que corresponda. La interfaz de acceso a datos es la interfaz utilizada por un servicio de negocio para interaccionar con los servicios de acceso a datos.

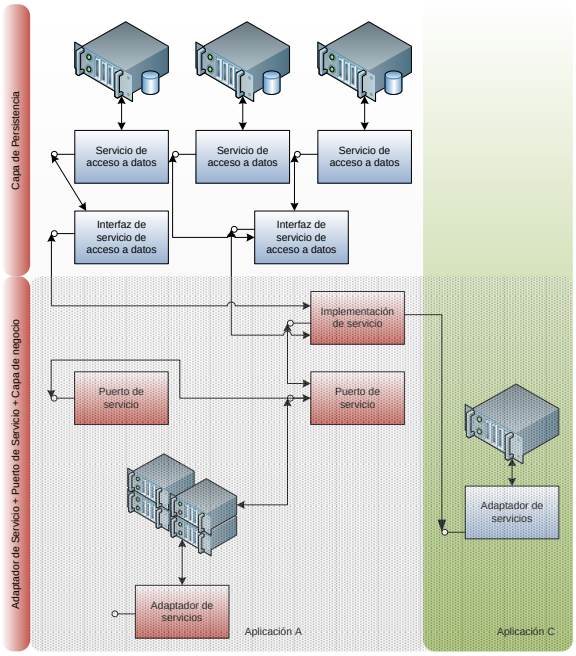


Ilustración 22 – Interfaces de negocio: reutilización de las implementaciones

El nivel de desacoplamiento introducido por la interfaz de acceso a datos permitirá implementar fácilmente los escenarios planteados en capítulos precedentes (2.6 y sub-secciones).

Las interfaces de acceso a datos facilitan la sustitución, la composición o la orquestación de múltiples componentes ya que distintos servicios que implementen la misma interfaz de acceso a datos serán indistinguibles por el cliente que los invoca.

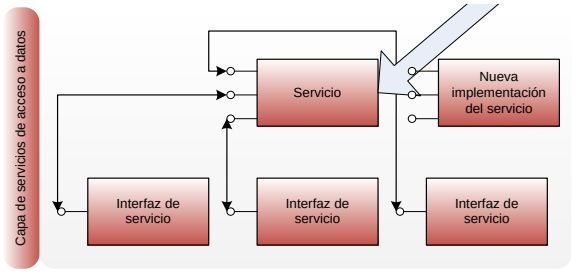


Ilustración 23 – Sustitución de servicios de acceso a datos que utilicen la misma interfaz de acceso a datos

6.3.1 Requerimientos de los métodos de acceso a datos

Con analogía a lo que se ha detallado para las interfaces de los servicios de negocio, las interfaces de acceso a datos están sujetas a las normas a las que están sujetas las interfaces de los Enterprise JavaBean™ v. 3.0 y a las normas a las que están sujetos los métodos de interfaz. Los métodos de las interfaces de acceso a datos están sujetos a las mismas normas detalladas en el apartado 3.3.1.

6.3.2 Requerimientos del interfaz de acceso a datos

Las interfaces de acceso a datos de un servicio de acceso a datos están sujetas a los requerimientos del Departamento de Arquitectura de la Dirección General de Tráfico. Estos requerimientos se suman a los requerimientos establecidos por la Enterprise JavaBean™ 3.0 Specification (3) y que han sido detallados en el apartado 3.3.2 para las interfaces de los servicios de acceso a negocio:

- No extiende **`javax.ejb.EJBObject`**.
- No extiende **`javax.ejb.EJBLocalObject`**.
- Puede lanzar excepciones aplicativas.
- Puede tener súper interfaces.

Un requerimiento adicional que se impone a los servicios de acceso a datos es el de utilizar exclusivamente interfaces locales:

- El componente EJB que implementa una interfaz de acceso a datos declarará e implementará exclusivamente interfaces *locales*.

Este requerimiento está dictado por la visibilidad de un servicio de acceso a datos: ya que este tipo de servicio será disponible exclusivamente a la aplicación que lo define, por razones de simplicidad y de rendimiento se impone que utilicen solo interfaces locales.

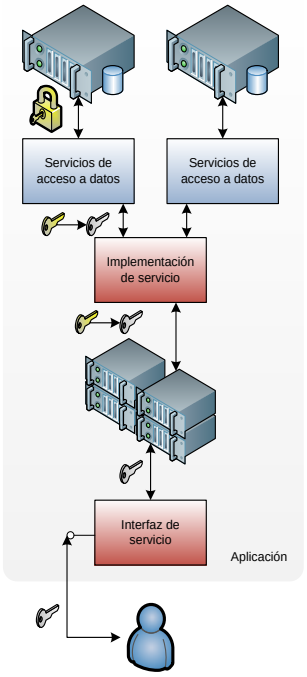


Ilustración 24 – Los servicios de acceso a datos son visibles solo en el interior de la aplicación que los define (sus interfaces son locales).

Las principales consecuencias para el equipo de desarrollo de esta tipología de componentes serán las siguientes:

- Mejor rendimiento.
- Posibilidad de utilizar paso de parámetros por referencia.
- Afinidad con la aplicación que los define.

6.3.3 Ejemplo: estructura básica de un servicio de acceso a datos

La estructura básica de un servicio de acceso a datos es análoga a la estructura básica de un servicio de negocio, ya que ambos son componentes EJB.

El siguiente ejemplo ilustra la estructura básica de un servicio de acceso a datos.

```
package es.trafico.ACRONIMO.dao.<paquetes>;

import es.trafico.framework.servicios.datos.ServicioDeDatos;
import javax.ejb.Stateless;

@Stateless
public class DatosPersonaBean
  implements IDatosPersona {

  public String getDatosPersona(string id) {
        // implementación
  }

 // metodos de acceso a datos
}
```

Código 6 – Estructura básica de un servicio de acceso a datos

Este servicio de acceso a datos es un componente EJB de tipo *stateless* que implementa una interfaz de acceso a datos, **`IDatosPersona`**. La interfaz de acceso a datos tiene que ser una interfaz local, como la interfaz del ejemplo siguiente:

```
package es.trafico.ACRONIMO.dao.local.<paquetes>;

import javax.ejb.Local;

@Local
public interface IDatosPersona {
  String getDatosPersona(string id);
}
```

Código 7 – Interfaz de acceso a datos

7 Patrones para el diseño y la implementación de servicios de acceso a datos

Las interfaces de acceso a datos son básicamente la herramienta con la que se consigue el desacoplamiento entre los servicios de negocio y los servicios de acceso a datos, constituyendo los elementos que conforman la capa de persistencia de la aplicación.

Por su propia naturaleza, las interfaces de acceso a datos son visibles solo en el interior de las aplicaciones que las definen y por lo tanto no están sujetas a los requisitos funcionales de los contratos publicados por una aplicación.

Aún así, las interfaces de datos son responsables de proporcionar datos a los servicios de negocio y por lo tanto podrían verse impactadas por algunas decisiones tomadas en fase de diseño de los servicios de negocio como por ejemplo el soporte a la paginación de datos (4.3.3).

Otra preocupación del Departamento de Arquitectura de la Dirección General de Tráfico es cómo soportar el desarrollo y el mantenimiento de aplicaciones en el escenario de una migración de base de datos. La implementación de un servicio de acceso a datos depende en cierta medida del sistema de almacenamiento de datos concreto que se esté utilizando. Las interfaces de acceso a datos también permiten crear soluciones a problemas comunes durante la migración de las bases de datos con el objetivo de limitar el impacto en el desarrollo de las aplicaciones.

7.1 Soporte a múltiples backends

Como ha quedado expuesto en capítulos anteriores, una aplicación podría tener como requisito dar soporte a múltiples sistemas de almacenamiento al mismo tiempo.

En esta sección el Departamento de Arquitectura de la Dirección General de Tráfico introducirá los casos más comunes y sugerirá pautas y patrones que sirvan como norma y referencia durante el desarrollo de las aplicaciones Java™ para la DGT.

Debido a la importancia y a la gran utilización que hacen las aplicaciones de los sistemas de almacenamiento de datos se estima oportuno añadir un conjunto de abstracciones para desacoplar las aplicaciones de las peculiaridades de los backends que estén utilizando.

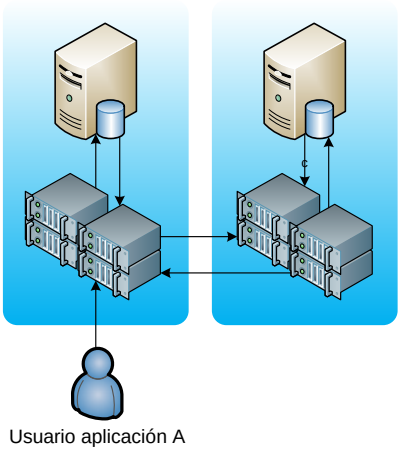


Ilustración 25 – Particionado del modelo de dominio de datos utilizados por las aplicaciones

Como queda ilustrado en la figura precedente, en el modelo propuesto por esta especificación cada aplicación adquirirá la responsabilidad de definir y manejar su modelo de datos. La misma aplicación, así como las demás aplicaciones que utilicen servicios de negocio proporcionados por ésta, no tendrán visibilidad de los sistemas de almacenamiento utilizados para organizar los datos. La situación en la que una aplicación tiene visibilidad del backend de almacenamiento de datos utilizado por otra aplicación[33], ilustrada en la siguiente figura, queda prohibida por esta especificación de construcción de servicios de negocio y servicios de acceso a datos.

La responsabilidad adquirida por las aplicaciones sobre su propio modelo de datos tiene como consecuencia la imposibilidad por otra aplicación de acceder directamente a los backends de almacenamiento que no sean de su competencia.

Una aplicación interactuará con los sistemas de almacenamiento de datos de su competencia a través de los *servicios de acceso a datos*. Los servicios de acceso a datos implementan una interfaz que será utilizada para desacoplar las implementaciones de los servicios de acceso a datos y facilitar la migración de las aplicaciones cuando cambie el backend de almacenamiento de datos.

Como puede verse en la imagen siguiente, un cliente puede estar utilizando, a través de su interfaz de negocio, una o más implementaciones de servicios de acceso a datos. Si cambiase un backend podría por ejemplo sustituirse una implementación o incluso añadir una nueva implementación para suplir las necesidades que se hayan producido.

[33] A través de cualquier medio: conexión directa, creación de vistas ad hoc, etc.

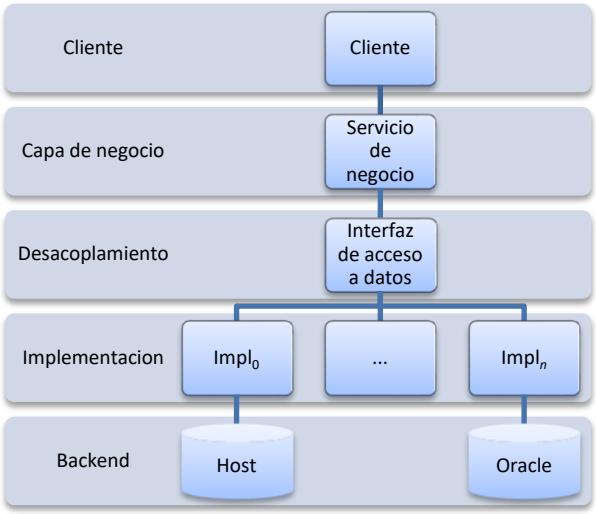


Ilustración 26 – Servicio de acceso a datos con múltiples implementaciones de su interfaz de acceso a datos

7.1.1 Soporte a la migración de los backends de almacenamiento de datos

Los servicios de acceso a datos encapsulan toda la lógica de acceso a un determinado backend de almacenamiento de datos y proporcionan operaciones definidas en la interfaz del servicio de acceso a datos.

Cuando cambie un backend de almacenamiento de datos será suficiente proporcionar una implementación alternativa de un conjunto de servicios de acceso a datos sin que la lógica del llamante se vea afectada, en virtud de la separación de responsabilidades que se concreta en la ocultación de la implementación concreta del servicio de acceso a datos al exterior del mismo.

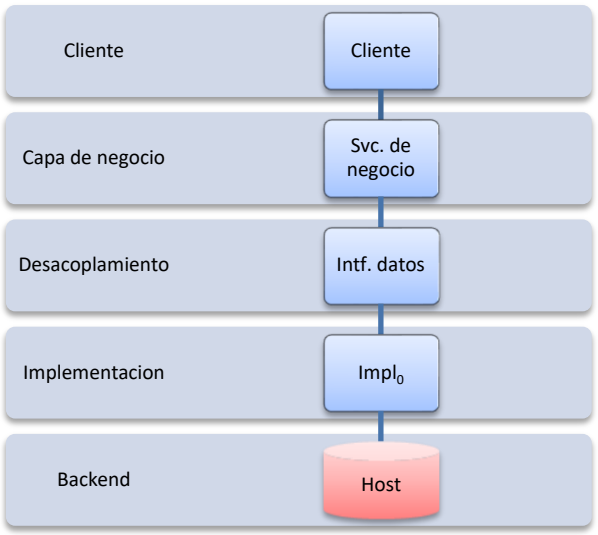


Ilustración 27 – Implementación de partida: backend alojado en el entorno host

Una aplicación podría por ejemplo estar invocando un servicio de negocio que a su vez utiliza un servicio de acceso a datos cuyo backend subyacente fuera una base de datos alojada en el entorno HOST, como puede verse en la Ilustración 27.

Debido a la necesidad de migrar el backend HOST a otro backend de almacenamiento de datos, por ejemplo una base de datos relacional Oracle, se realizará una implementación alternativa de las interfaces de acceso a datos de los servicios de acceso a datos afectados por la migración. Los servicios de negocio que utilicen estos servicios de acceso a datos no se verán afectados en cuanto la interfaz de acceso a datos de las cuales dependen no ha sufrido modificaciones.

El resultado de la actividad de migración será el mismo conjunto de módulos presentes en la situación de partida, con la única excepción de la implementación del servicio de acceso a datos. En la situación final la implementación del servicio de acceso a datos se verá sustituida por una implementación que utiliza exclusivamente el backend Oracle, como se aprecia en la Ilustración 28.

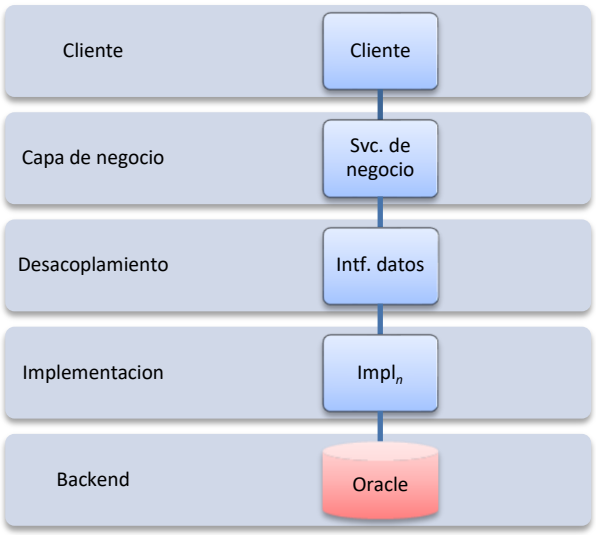


Ilustración 28 – Implementación final: backend alojado en la plataforma Oracle

Los servicios de acceso a datos, conjuntamente a los servicios de negocio, separan ortogonalmente la lógica de negocio de la lógica de acceso a datos: este patrón proporcionará ayuda eficaz a la hora de migrar aplicaciones existentes y construir nuevas aplicaciones en cuanto la lógica que maneja y explota datos está completamente desacoplada de la lógica que proporciona dichos datos.

7.1.2 Configuración del backend en tiempo de despliegue

El mecanismo de desacoplamiento entre los servicios de negocio y los servicios de acceso a datos será efectivo en la medida en que el cambio de una implementación a otra pueda efectuarse de manera ágil y sencilla.

Existen dos posibilidades para efectuar el cambio de implementación (consultar la Guía de Desarrollo para saber cuándo hay que usar cada mecanismo):

1. Utilizar los mecanismos de inyección de dependencia proporcionados por los estándares que componen la plataforma Java™ EE (1). Esta opción es la adecuada cuando la aplicación implementa directamente el acceso al backend. Dichos mecanismos están basados en los *metadatos* proporcionados por el equipo de desarrollo y de despliegue a través de anotaciones o las equivalentes informaciones en los descriptores de despliegue. Los equipos de desarrollo podrán utilizar mecanismos de inyección de dependencia distintos o más refinados en la aplicación que están desarrollando mientras estos no interfieran y respeten la compatibilidad con los mecanismos estándar de la plataforma.

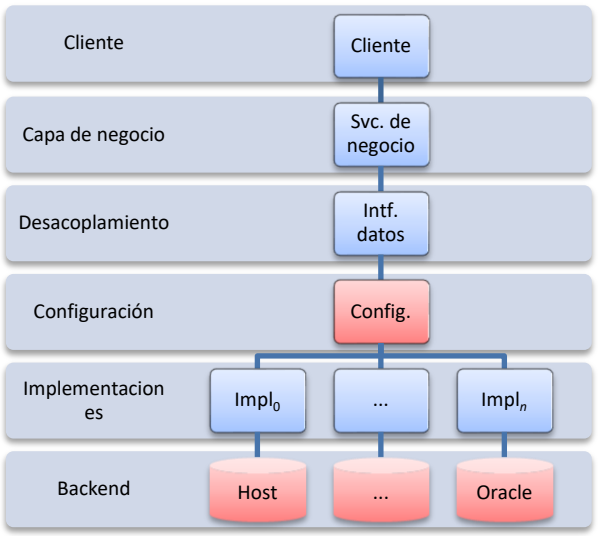


Ilustración 29 – Configuración durante el despliegue: selección del backend

2. Usar la plataforma de servicios de acceso a datos (DAAS). Esta es la opción preferente salvo acuerdo con el Departamento de Arquitectura en función de la situación del proyecto. Cuando se usan servicios de acceso a datos proporcionados por esta plataforma, la aplicación no implementa directamente el acceso al backend sino que invoca un servicio externo que le proporciona sus datos; este servicio externo es quien realiza el cambio, accediendo a un backend distinto y proporcionando los mismos datos según se establece en la interfaz del servicio.

7.1.3 Soporte al paralelo de backends distintos durante la migración

Un servicio de acceso a datos proporcionará una implementación concreta de una interfaz de acceso a datos por un determinado backend de almacenamiento de datos (a tal efecto puede considerarse la plataforma de servicios de acceso a datos como un backend). Un escenario al que el equipo de Arquitectura de la Dirección General de Tráfico quiere dar soporte es el *paralelo* de backends durante la migración.

7.1.4 Directrices de diseño para la implementación de paralelo de backends

Las múltiples implementaciones de la misma interfaz de acceso a datos, una para cada backend que sea necesario utilizar durante el proceso de migración, podrán ser necesarias en las situaciones en las cuales no sea viable completar la migración de un backend en una sola iteración. En este escenario, durante la migración desde un backend a otro, se supone que existirá un periodo de tiempo durante el cual las aplicaciones deberán acceder contemporáneamente a múltiples backends. Las tipologías de accesos a los backends podrán ser de varios tipos:

- Accesos en paralelo para lectura y/o escritura: en este escenario una aplicación necesitará leer y/o escribir datos en múltiples backends de forma redundante. Una escritura, por ejemplo, podrá concretarse en múltiples escrituras en múltiples backends distintos.

- Accesos concurrentes para lectura y/o escritura: en este escenario una aplicación leerá y escribirá datos de forma concurrente en múltiples backends. Una escritura, por ejemplo, podrá ocurrir parcialmente en un backend y parcialmente en otro backend.

- Una combinación de las precedentes.

El concepto de servicio de acceso a datos pretende solventar el problema de escribir aplicaciones que soporten estos escenarios sin incurrir en costes o complejidades añadidas a la hora de desarrollar las aplicaciones. Cada servicio de acceso a datos implementará una interfaz de acceso a datos para un determinado backend. El soporte a los escenarios descritos anteriormente se dará a través de la *orquestación*, posiblemente debajo de la misma interfaz, de un conjunto de servicios de acceso a datos que soporten todos los backend que en cada momento sean necesarios. El desacoplamiento fundamental de la lógica de negocio se concreta en el uso de las interfaces de acceso a datos: la lógica de negocio invocará exclusivamente métodos de la interfaz de un servicio de acceso a datos que a su vez, según las necesidades, orquestará invocaciones al conjunto oportuno de servicios de acceso a datos que se necesiten.

La lógica de configuración de las aplicaciones, evidentemente, deberá permitir la modificación de las dependencias entre servicios de forma que una aplicación no se vea afectada según procedan las actividades de migración. Para lograr este objetivo se utilizarán mecanismos de inyección de dependencia configurados externamente a las aplicaciones.

Básicamente, esta solución es una combinación de los dos casos de uso presentados en las secciones precedentes: por un lado este patrón permite inyectar una o más implementaciones de una determinada interfaz de acceso a datos en el servicio que la requiere. Por el otro lado el mecanismo de inyección es lo suficientemente flexible para permitir que los cambios en la configuración de las dependencias puedan efectuarse sin necesidad de modificar el código de la aplicación desplegada.

7.1.4.1 Ejemplo de acceso concurrente a múltiples backend en escritura

Para aclarar los conceptos expresados en las secciones precedentes se presenta como ejemplo un servicio de acceso a datos que tiene como caso de uso el acceso concurrente a un conjunto de backends.

En este ejemplo, durante la primera fase de migración podría ser necesario implementar un paralelo entre la vieja y la nueva base de datos. Los servicios de negocio utilizan los servicios de acceso a datos a través de sus interfaces de negocio y por lo tanto no se verán afectados en las modificaciones que afecten las implementaciones de dichos componentes siempre que sus interfaces no se modifiquen.

Las situaciones que se presentan en este ejemplo son las siguientes:

- Algunos servicios escribirán la *misma* información en ambos backend.

- Algunos servicios tendrán que escribir parte de la información en un backend y parte en el otro.
- Algunos servicios tendrán que leer la información de ambos backend.

El primer caso de uso se da por ejemplo en las situaciones en las cuales el nuevo backend ya soporta la escritura de la información que se está persistiendo pero, por alguna razón[34], es necesario seguir insertando esta información en el backend viejo. Este caso de uso es fácilmente solucionable a través de un *proxy* que delegue la operación a *sendos* backend, como queda ilustrado en la siguiente figura.

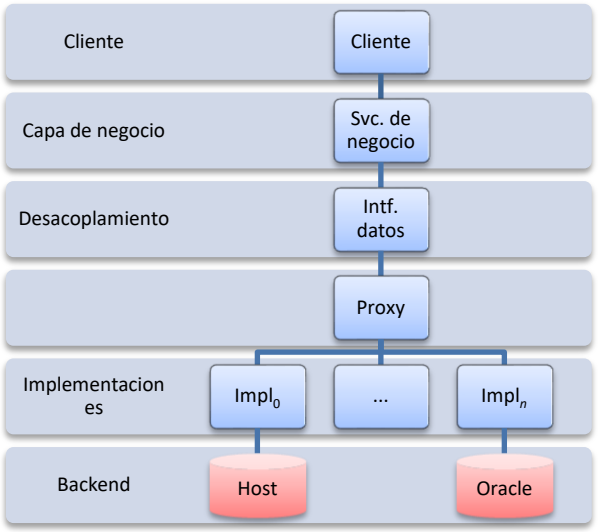


Ilustración 30 – Proxy de servicios de acceso a datos para la implementación de paralelos

El proxy, así como las dos versiones del servicio de acceso a datos, implementarán la misma interfaz de acceso a datos. El servicio que necesite esta funcionalidad dependerá de una implementación que, en la primera fase de la migración, será un proxy que efectúa una simple delegación de responsabilidad a las implementaciones que respalda. En este caso, la invocación de un método del interfaz de acceso a datos del proxy se podrá implementar tan sencillamente como en el pseudo-código ilustrado a continuación.

[34] Tales como para mantener la integridad referencial de esta base de datos ya que la migración de las tablas afectadas afectarían las relaciones mantenidas con ellas por otras tablas todavía en fase de migración.

```
public void ServiceProxy implements IServicio {
  IServicio delegates[];

  public void doSomething(Param param) {
    for (IServicio d : delegates) {
      param.doSomething(param);
    }
  }
}
```

Código 8 – Pseudo-código de un proxy de un servicio que delega las operaciones a un vector de implementaciones delegadas

Por lo tanto una solución posible es, después de implementar el servicio contra el nuevo backend, orquestar las invocaciones a través de un proxy que implemente la interfaz de negocio del servicio de acceso a datos, como ilustrado someramente en la ilustración siguiente.

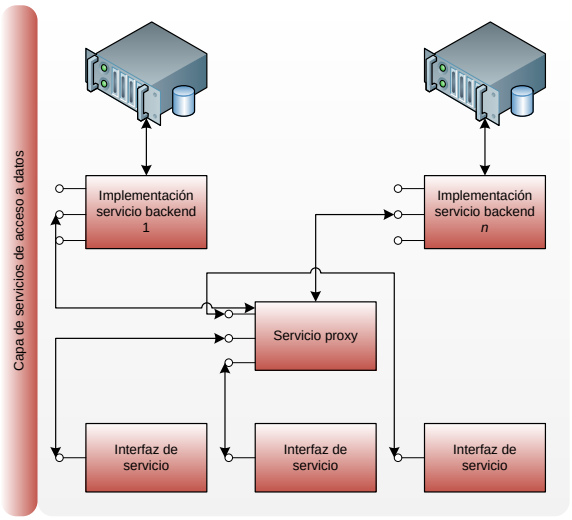


Ilustración 31 – Servicio de acceso a datos proxy

Con técnicas parecidas se pueden atacar los casos de uso siguientes. En lugar de delegar cada operación secuencialmente a cada uno de los servicios de acceso a datos delegados a través de un proxy, será posible implementar un objeto que contenga la mínima lógica necesaria para que las operaciones se vayan efectuando con el backend correcto.

De hecho, si el servicio de acceso a datos ya está configurado para utilizar el backend correcto, el código necesario para cumplir esta función se reduciría otra vez al pseudo-código ilustrado en precedencia.

```
public void Service implements IServicio {
  IOldService old;
  INewService new;

  public void doSomething(Param param) {
    // some logic here
    old.doThis();
    // some logic here
    new.doThat();
  }
}
```

Código 9 – Pseudo-código de un servicio que utiliza dos servicios que utilizan dos backends distintos

8 Control transaccional en los servicios de acceso a datos

Como ha quedado explicado anteriormente, la demarcación transaccional se efectúa declarativamente en los servicios de negocio[35]. Por lo tanto, mientras se utilicen recursos manejables por un gestor transaccional que utilice JTA (2), en los servicios de datos no aparecerá lógica destinada al manejo de las transacciones. Los servicios de acceso a datos participan entonces en las transacciones iniciadas por los servicios de negocio. En este caso de uso, por lo tanto, un componente de acceso a datos podrá implementarse como un Enterprise JavaBean™ (3) de tipo *stateless*.

Sin embargo, los servicios de acceso a datos también manejan recursos no transaccionales en pseudo-transacciones utilizando el patrón C*ommand* y durante el desarrollo de este tipo de servicios de acceso a datos hay que seguir las directrices impuestas por esta especificación. En particular cuando se manejan recursos de este tipo el contenedor no puede gestionar el commit y el *rollback* de las operaciones efectuadas sobre estos recursos y por lo tanto es necesario disponer de un mecanismo que permita ejecutar y deshacer dichos comandos. Por esta razón, la Dirección General de Tráfico impone en su modelo de programación el uso de Enterprise JavaBeans™ (3) de tipo *stateful* para poder propagar el evento asociado al terminarse una transacción.

8.1 Control transaccional declarativo

Los servicios de acceso a datos están sujetos a las recomendaciones detalladas por los servicios de negocio (§ 5).

En lo que concierne la declaración y el manejo de las transacciones JTA (2) por parte de los servicios de negocio, aplica integralmente la información proporcionada en § 5.

8.2 Manejo de recursos transaccionales

Un servicio de acceso a datos que maneje recursos transaccionales tales como DataSources, Persistence Contexts JPA (3), colas JMS, etc., se programará como un POJO que utiliza directamente dichos recursos. El control transaccional estará garantizado por la utilización del modelo declarativo y por la propagación del contexto transaccional por parte del contenedor.

[35] No puede efectuarse una invocación a servicios de datos sino desde un servicio de negocio.

8.2.1 Contextos de persistencia

Una categoría de servicios de acceso a datos utilizarán recursos proporcionados por una implementación de la Java™ Persistence API como los *contextos de persistencia*[36] y las *unidades de persistencia*[37] manejados por el contenedor.

Los EJB 3.0 (3) pueden obtener una referencia a dichos recursos a través de los mecanismos previstos por la especificación Java™ EE v. 5 (11):

- Lookup en el contexto del bean.
- Inyección de dependencia efectuada por el contenedor.

Las dos opciones son equivalentes y esta especificación sugiere optar por la inyección de dependencias por parte del contenedor como en el ejemplo siguiente:

```
@Stateful
public class MatriculacionPlacaBean
  implements MatriculacionPlacaLocal,
    SessionSynchronization{

  private MatriculacionPlacaCommand mpc = null;
  /**
   * inject dependent resources
   */
  @PersistenceContext
  EntityManager em;
  @Resource
  SessionContext ctx;
```

Código 10 – Recuperar el contexto de persistencia en un Enterprise JavaBean™

En este ejemplo se solicita al contenedor la inyección de referencias a los recursos necesarios para el funcionamiento del bean:

- Una referencia a un **`EntityManager`** a través de la anotación **`@PersistenceContext`**.
- Una referencia al contexto de sesión del bean a través de la anotación **`@Resource`**.

Los recursos inyectables en una instancia de un EJB 3.0 (3) están desglosados en la *Enterprise JavaBeans™ 3.0 Specification* (3) que a su vez utiliza la especificación *Common Annotations for the Java™ Platform* (12).

[36] Persistence context en la literatura oficial.
[37] Persistence units en la literatura oficial.

8.2.2 Referencias a contextos de persistencia

Una funcionalidad prevista por la Enterprise JavaBeans™ 3.0 Specification (3) es la posibilidad de declarar *referencias a contextos de persistencia* configurables en el descriptor de despliegue del módulo Java™ Enterprise y utilizarlos en el código de lookup para obtener las referencias a los contextos de persistencia.

La necesidad de utilizar este mecanismo se puede presentar si se detecta la necesidad de desacoplar la configuración de las unidades de persistencia del código que las utilizan. Concretamente podría ser necesario para un componente obtener una referencia a un contexto de persistencia de una determinada unidad de persistencia entre las unidades configuradas y visibles desde un determinado componente.

Para utilizar este mecanismo es necesario:

- Declara la dependencia de una clase de una referencia a un contexto de persistencia.
- Establecer el mapeo de la referencia en cuestión.
- Efectuar el lookup del contexto de persistencia en el contexto del bean.

Los pasos necesarios se ilustran en el siguiente ejemplo:

```
@PersistenceContext(name = "persistence/micontexto")
@Stateful
public class MatriculacionPlacaBean
  implements MatriculacionPlacaLocal,
    SessionSynchronization{

  private MatriculacionPlacaCommand mpc = null;
  @Resource
  SessionContext ctx;

@TransactionAttribute(
  TransactionAttributeType.MANDATORY)
public String getNewPlate() {
  EntityManager em = (EntityManager)
    ctx.lookup("persistence/micontexto");
```

Código 11 – Inyección de dependencia en un Enterprise JavaBean™ a través de metadatos

El código del ejemplo declara la dependencia de la clase **`MatriculacionPlacaBean`** de una referencia a un contexto de persistencia llamado **`persistence/micontexto`**. La Enterprise JavaBeans™ 3.0 specification (3) sugiere de utilizar el subárbol JNDI **`java:comp/env/persistence`** para almacenar las referencias a contextos de persistencia y la presente especificación fortalece esta sugerencia imponiendo esta norma. Por esta razón la propiedad **`name`** de la anotación **`@PersistenceContext`** del ejemplo precedente asume el valor **`persistence/micontexto`**[38].

[38] Este valor, de hecho, se entiende relativo a **`java:comp/env`**.

La declaración de una referencia a un contexto de persistencia en el descriptor de despliegue se hace a través del elemento **`persistence-context-ref`**:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns=http://java.sun.com/xml/ns/javaee
  version = "3.0"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<enterprise-beans>
  <session>
    <ejb-name>MatriculacionPlacaBean</ejb-name>
    <persistence-context-ref>
      <description>
         Descripcion
      </description>
      <persistence-context-ref-name>
        persistence/micontexto
      </persistence-context-ref-name>
      <persistence-unit-name>
        EJBServicePU
      </persistence-unit-name>
    </persistence-context-ref>
  </session>
</enterprise-beans>
</ejb-jar>
```

Código 12 – Inyección de dependencia en un Enterprise JavaBean™ utilizando el descriptor de despliegue

8.3 Manejo de recursos no transaccionales

Un servicio de acceso a datos que maneje un recurso no transaccional deberá hacerlo implementando el patrón *Command*. Este patrón prevé la implementación de la interfaz *Command* que declara los siguientes métodos:

```
public interface Command {
  void execute();
  void rollback();
}
```

Código 13 – Interfaz de un comando

El método **`execute`** recoge y ejecuta la lógica implementada por el servicio de acceso a datos. Los parámetros que eventualmente habrá que pasar al objeto en cuestión se proporcionarán a través de las demás interfaces que éste implemente y no a través del interfaz del comando. Asimismo, la devolución de eventuales parámetros de salida se obtendrá a través de un método de las demás interfaces del objeto.

Dado que se está asumiendo que los recursos manejados dentro de un comando sean recursos no transaccionales, se está asumiendo que la operación ejecutada en este método no necesite una operación de *commit* para terminarse.

El método **`rollback()`** debe implementarse para construir la lógica de un flujo de compensación, es decir la lógica necesaria para que en caso de *rollback* el recurso no transaccional pueda volver al estado de partida.

Básicamente este patrón define los métodos con los cuales un servicio se ejecuta (**`execute`**) y el método a invocarse para compensar las operaciones ejecutadas por el mismo en el caso en que se produjera algún error (**`rollback`**). El manejo de recursos no transaccionales prevé entonces la realización de flujos de compensación cuando el recurso no sea compatible con JTA (2). El flujo de compensación, en estos casos, será parte integrante de la actividad de análisis y desarrollo por parte de las aplicaciones.

El flujo de compensación es fundamental para garantizar la coherencia de los datos en caso de error, siendo el mismo el responsable del buen estado de los datos al terminarse con error una operación. Sin embargo un flujo de compensación no puede ser inmune a errores y por lo tanto hay que prever también un mecanismo para la notificación de los errores que se produzcan durante esta fase de ejecución de la aplicación[39].

8.3.1 Estado conversacional

Como ha quedado explicado, es necesario mantener un estado conversacional durante la duración de la transacción para los beans que manejen recursos no transaccionales. De esta manera se podrá recibir el evento asociado al final de una transacción JTA y actuar en consecuencia:

- Si la transacción termina con éxito, en general no será necesario ejecutar operaciones adicionales.

- Si la transacción termina con un rollback, será necesario ejecutar los flujos de compensación asociados a los comandos que se ejecutaron dentro de la transacción.

8.3.2 Stateful beans

Los servicios de acceso a datos que manejen recursos no transaccionales se implementarán entonces como stateful Enterprise JavaBeans™. Un bean stateful se puede declarar utilizando la anotación @Stateful o la sintaxis análoga en el descriptor de despliegue del Enterprise JavaBean™:

[39] La gestión de este tipo de errores no se cubre en la presente especificación.

```
package es.trafico.ACRONIMO.dao.<paquetes>;

import es.trafico.framework.servicios.datos.ServicioDeDatos;
import es.trafico.command.MatriculacionPlacaCommand;
import java.math.BigDecimal;
import java.rmi.LocalException;
import javax.ejb.EJBException;
import javax.ejb.SessionSynchronization;
import javax.ejb.Stateful;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateful
public class MatriculacionPlacaBean
  implements MatriculacionPlacaLocal,
    SessionSynchronization{
    // lógica
}
```

Código 14 – Enterprise JavaBean™ de tipo stateful

8.3.3 Recursos no transaccionales utilizados por el bean

Los recursos no transaccionales estarán encapsulados en instancias de clases que implementen el patrón *Command*, como ha quedado explicado anteriormente. El bean tiene que tener traza de estas instancias porque, en el caso en que la transacción termine con un *rollback*, los flujos de compensación de cada comando tienen que ejecutarse: por lo tanto los comandos se almacenarán como variables de instancia del bean:

```
@Stateful
public class MatriculacionPlacaBean
  implements MatriculacionPlacaLocal,
    SessionSynchronization{

    private MatriculacionPlacaCommand mpc = null;

    @PersistenceContext
    EntityManager em;
```

Código 15 – Utilizar recursos no transaccionales en un Enterprise JavaBean™

8.3.4 EJBCommandSupport

La clase abstracta EJBCommandSupport proporcionada por el Departamento de Arquitectura proporciona la funcionalidad básica necesaria para establecer una asociación entre comandos ejecutados y el estado conversacional del componente EJB que lo ha ejecutado.

El único método de esta interfaz que sea de interés para los equipos de desarrollos de aplicaciones Java™ para la DGT es el método:

```
void invokeCommand(Command cmd);
```

Código 16 – Método de invocación de comandos del interfaz EJBCommandSupport

8.3.5 Invocación de comandos

Los comandos a ejecutarse se invocarán a través del método proporcionado por la clase **`EJBCommandSupport`**. Un servicio de acceso a datos que necesite invocar comandos, entonces, utilizará un fragmento de código como el siguiente:

```
@Stateful
public class MatriculacionPlacaBean
  extends EJBCommandSupport
  implements MatriculacionPlacaLocal, ServicioDeDatos {

  private MatriculacionPlacaCommand mpc = null;

  @TransactionAttribute(
    TransactionAttributeType.MANDATORY)
  public String getNewPlate() {
    // logic here
    mpc = new MatriculacionPlacaCommand(new BigDecimal(3));
    this.invokeCommand(mpc);
    BigDecimal result = mpc.result();
    // logic here
    }
}
```

Código 17 – Ejemplo: Servicio de acceso a datos que invoca un comando

8.3.6 Métodos de negocio

Los métodos de negocio de un servicio de acceso a datos, tal y como ha sido explicado en el punto precedente, son métodos que esperan que una transacción ya esté arrancada por el contenedor y por lo tanto se tienen que declarar con el atributo MANDATORY o con la sintaxis equivalente en el descriptor de despliegue:

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public String getNuevaPlaca() {
  mpc = new MatriculacionPlacaCommand();
  mpc.execute(new BigDecimal(0));
  // lógica de negocio
}
```

Código 18 – Utilizar el atributo MANDATORY para requerir una transacción existente durante la ejecución de un método

8.3.7 Terminación de una transacción

Para recibir el evento asociado al finalizar una transacción, un bean de tipo stateful tiene que implementar la interfaz **`javax.ejb.SessionSynchronization`** e implementar sus métodos:

```
public void afterBegin()
  throws EJBException, RemoteException {
  return;
}

public void beforeCompletion()
  throws EJBException, RemoteException {
  return;
}

public void afterCompletion(boolean committed)
  throws EJBException, RemoteException {
  if (!committed)
  // invocar flujo de compensación
}
```

Código 19 – Recibir el evento asociado al finalizar una transacción en un Enterprise JavaBean™

El método **`afterCompletion`** se invocará al terminarse la transacción y el parámetro de entrada del método será:

- **`True`**, si la transacción ha terminado con éxito.
- **`False`**, si la transacción ha terminado con un rollback.

La lógica necesaria para cumplir este requerimiento está encapsulada dentro de la clase abstracta **`EJBCommandSupport`**.

8.4 Paginación

Podrán darse casos en los cuales sea necesario implementar una solución de paginación de datos para reducir el tamaño de la información devuelta por un servicio de negocio, implementando por ejemplo un *iterador* que permita solicitar *páginas* de datos.

En lo que concierne a los servicios de acceso a datos, será necesario implementar la funcionalidad requerida por el servicio de negocio de manera eficiente.

Esta especificación no entra en el detalle de cómo implementar este tipo de solución, ya que sistemas de almacenamiento de datos distintos tienen características distintas y tales que una solución óptima en uno no lo sea en otro.

Una solución genérica a este problema podría implementarse utilizando la API JPA (3). El interfaz javax.persistence.Query proporciona los siguientes métodos:

```
Query setFirstResult(int pos);
Query setMaxResult(int num);
```

Código 20 – Extracto de los métodos del interfaz javax.persistence.Query

Estos métodos pueden utilizarse para construir un servicio de acceso a datos que cree páginas de resultados de tamaño prefijado.

Las contraindicaciones de esta solución es principalmente que la paginación se delega al driver JDBC. Drivers distintos y bases de datos distintas pueden tener comportamientos distintos en lo que concierne el tamaño de los datos transmitidos que, al final, afectará al rendimiento de este método.

9 Escenarios de integración de servicios

Este apartado describe las tecnologías identificadas como típicas dentro de la DGT.

A continuación se detallan posibles escenarios y se incorporan criterios donde esta especificación sólo se aplicará parcialmente.

9.1 Integración entre subsistemas de Internet e Intranet

Las aplicaciones que tengan un subsistema desplegado en Internet y otro en Intranet, y que utilicen como plataforma WebSphere™ 7.0 deberán cumplir para la integración entre los subsistemas esta especificación y desarrollar un adaptador de servicio.

Se establece para estas aplicaciones que la publicación de servicio en Intranet y la invocación desde el entorno Internet se efectúe con tecnología REST o Web Services SOAP. En el subsistema de Intranet se crearán los "*Servicios de Negocio*" y "*Servicios de Acceso a Datos*" adecuados para realizar las operaciones y se crearán las adaptadores de integración para permitir al subsistema de Internet consumir el negocio expuesto.

Sólo se desplegarán "*Servicios de Acceso a Datos*" en el entorno de Intranet.

No se permitirá que exista lógica de negocio y/o de acceso a datos residente en el entorno de Internet. Los modelos de datos y las llamadas a servicios corporativos se realizarán preferiblemente en el entorno Intranet del aplicativo.

La comunicación entre subsistemas de Internet e Intranet debe ser segura y autenticada. Para ello se debe utilizar la guía de la especificación de seguridad DGTE-002 (13).

9.2 Publicación de servicios en Intranet

La publicación de Servicios de Negocio en Intranet deberá cumplir con esta especificación. Este canal siempre debe ser seguro con acceso autenticado, si la operativa está bajo los criterios indicados en el apartado 10.3.

9.3 Publicación de servicios Web en Internet

La publicación de servicios en Internet precisa de la construcción del adaptador de Servicios Web (SOAP o REST) en Intranet. Este adaptador se conectara con la capa de servicios de la aplicación en Intranet.

Una vez que se disponga de un Servicio Web operativo en Intranet éste se podrá publicar en Internet mediante la funcionalidad de Gateway que ofrece Datapower. Para ello conviene contactar con el Departamento de Sistemas.

El acceso y circulación de datos del Servicio Web deben ser seguros, con acceso autenticado, si el servicio está bajo el criterio indicado en el apartado 10.3.

9.4 Librerías reutilizables

A menudo se precisa promocionar un determinado conjunto de funcionalidades auxiliares como componentes separados. Es una buena práctica que fomenta la modularidad y el mantenimiento entre aplicaciones (o entre subsistemas de una misma aplicación). Estas funcionalidades típicamente producen librerías que son, en sí mismas, potencialmente reutilizables.

En esta especificación se prescribe que las librerías potencialmente reutilizables por otras aplicaciones se deben publicar en el repositorio de artefactos de la DGT como activos reutilizables.

Serán los equipos de desarrollo los responsables de separar, documentar y cumplir con las normativas de publicación de estos activos.

Dentro de esta especificación se establece una clara línea de responsabilidad en la creación, publicación y mantenimiento de estos activos: Su soberanía y propiedad debe corresponder al dominio específico del área de negocio. Este es un compromiso ineludible con el dominio y conocimiento que se tiene dentro del Área correspondiente.

Para la publicación efectiva de este activo en el repositorio de la DGT se ha de seguir en todas sus implicaciones lo señalado en la Guía de Desarrollo de la DGT para este procedimiento.

El ámbito de reutilización de funcionalidad queda restringido a los subsistemas de una misma aplicación. Si existe la necesidad de reutilización en otras aplicaciones, se deberá exponer dicha funcionalidad mediante servicio web, que las aplicaciones consumirán a través de los adaptadores de servicio.

10 Estructura de proyecto

10.1 Visión general

La estructuración de un proyecto en la DGT que provea de Servicios de Negocio y pueda llegar a publicar los mismos a otros proyectos sigue la filosofía y características detalladas en la "*Arquitectura Hexagonal*" (4) (actualmente renombrada a arquitectura de puertos y adaptadores). Con este modelo se consigue desacoplar las tecnologías de publicación e interfaz de usuario de la lógica de negocio (interna) de la aplicación. Esta arquitectura está orientada principalmente al desarrollo de adaptadores (5) y puertos para la integración entre módulos y/o aplicaciones, así como en la separación por capas (layers) y segregación de interfaces e implementaciones a todos los niveles.

Los objetivos principales que persigue la misma se basan en estructurar las aplicaciones como componentes cohesivos funcional y/o tecnológicamente, con separación clara de responsabilidades entre los mismos y tratando de minimizar las dependencias entre ellos, que en todo caso se acoplaran a través del mínimo número de puntos posibles (y siempre bajo abstracciones por medio de interfaces que independizan de sus implementaciones).

10.1.1 Conceptos básicos de integración de aplicaciones

Un escenario básico de integración se establece entre dos aplicaciones en la que una de ellas ofrece un servicio (*Service Provider*) y la otra lo utiliza (*Service Consumer*). La necesidad de esta comunicación se establece en términos de responsabilidad y propiedad.

El servicio ofrecido pertenece a un determinado dominio en el que opera la aplicación que ofrece el mismo (por ejemplo Conductores). La otra aplicación opera en un dominio distinto (por ejemplo Vehículos) pero ha de comunicarse o consultar a la primera determinada información. La motivación de esta interacción puede obedecer a una notificación relevante para el negocio de la aplicación proveedora o la consulta de información relevante para completar una operación en la aplicación consumidora. En este ejemplo la necesidad de comunicación se efectúa en términos de propiedad.

En otros casos la necesidad se establece en términos de responsabilidad. Un ejemplo es el de un módulo responsable de la presentación de datos de una aplicación desplegada en un entorno de Internet, que se ha de comunicar con otro módulo responsable de la consecución de operaciones desplegado en el entorno de Intranet. Ambos módulos pueden pertenecer al mismo ámbito funcional. En este caso la comunicación se establece en términos de responsabilidad.

10.1.2 Elementos arquitectónicos de integración (adaptadores y puertos de servicio).

En cualquier de los casos expresados anteriormente la aplicación proveedora de servicio tiene la infraestructura necesaria para realizar la operación requerida y a esa infraestructura la denominaremos **lógica de negocio**. En la DGT, esta lógica de negocio es la denominada capa de negocio (ver **¡Error! No se encuentra el origen de la referencia.**), en la que se da a las aplicaciones libertad en su diseño respetando los principios generales indicados en la Guía de Desarrollo.

Esta aplicación proveedora del servicio debe además desarrollar un **puerto de servicio** (un "*API de servicio*" operativa como POJO[i] y que es definido en base a *interfaces, en Java,* que son implementadas por el negocio) que abstrae dicho negocio de cara a los diversos adaptadores que pueden requerir el uso de dicho negocio. Dado que este API debe agrupar los conceptos básicos de negocio de alto nivel (o grano gordo) de la aplicación, la nomenclatura de paquetes seguirá una semántica de negocio particular de la DGT. Este **puerto de servicio** cumple la misión de ser un servicio de aplicación (ver **¡Error! No se encuentra el origen de la referencia.**).

Los **adaptadores** pueden cumplir una misión tecnológica para la integración vía servicios web, etc. con otras aplicaciones, o simplemente de representación de la información para interacción del usuario, como es el caso de los frontales web (JSF…) o los clientes pesados (Swing, AWT…).

10.2 Estructura de proyecto

A continuación se va a ir desgranando la estructura y relaciones que deben cumplir los proyectos desarrollados en la DGT bajo esta especificación. Obviamente, dependiendo de lo que se vaya a desarrollar; se usarán unas partes u otras. No obstante, la arquitectura general aplica a todos los proyectos.

10.2.1 Adaptadores

10.2.1.1 Adaptadores de servicios web (SOAP)

Los adaptadores de servicios web ofrecen una URL denominada "*endpoint*", a la cual se pueden dirigir las peticiones de operación y la comunicación se establece en un esquema de petición/respuesta de datos cuya estructuración es conocida por ambas partes. Como especificación de este conocimiento mutuo entre el proveedor y los potenciales consumidores, la aplicación proveedora publica un documento en un lenguaje convenido llamado WSDL[40]. Este documento se conoce usualmente como "*el contrato de servicio*". Es importante reconocer el hecho de que el WSDL no está atado a operaciones descritas bajo servicios web y que es una manera universal de describir operaciones de servicios en cualquier tecnología.

[40] http://www.w3.org/TR/wsdl

Suele ser una mala costumbre usar contratos cuyos parámetros o respuestas se constituyen en estructuras opacas, sin un significado que se pueda fácilmente denominar. Esto obedece a una tendencia defensiva frente a la naturaleza cambiante de los contratos de servicio durante el transcurso del desarrollo de software. Sin embargo esta práctica origina una falta de claridad de los términos del contrato de los servicios y propicia, finalmente, que éstos sean infrautilizados o sólo utilizados por los conocedores de ciertos convenios internos que están escondidos en esos términos abiertos.

El significado conceptual de los objetos estructurados que viajan en la capa de integración les otorga una enorme importancia en la reutilización y en la forma de concebir la organización como partes bien definidas y verbalizadas. El lenguaje natural se puede utilizar de esta manera como metáfora de la manera de operar de los servicios, ya que estos constituyen frases con significado funcional.

Llamaremos en esta especificación a los objetos estructurados de forma conceptual que se definen en el wsdl como "*Entidades del contrato*". Estas construcciones corresponden con el patrón de integración de aplicaciones empresariales llamado "*Document message*"[41]. Estas entidades, inventariadas y publicadas, conforman un modelo en la capa de integración con significado que pueden compartir diversas aplicaciones.

El establecimiento y conocimiento de este lenguaje común acelera la capacidad de la organización para catalogar eficientemente sus activos y operar con ellos con naturalidad.

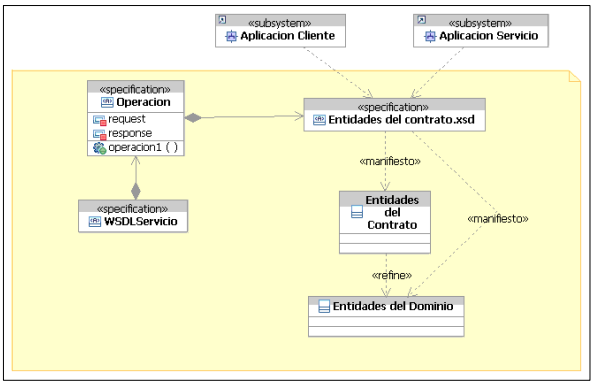
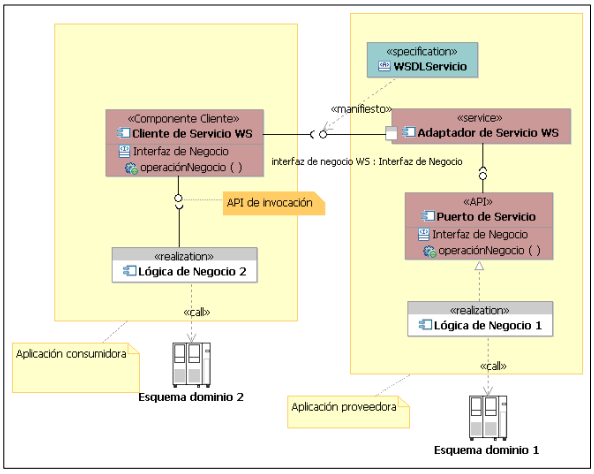


Ilustración 32. Composición de entidades del contrato en un adaptador de servicio web y relación con las entidades de dominio.

Un contrato WSDL consiste en un conjunto de operaciones. Cada operación está estructurada en un esquema de petición/respuesta. Tanto las peticiones como las respuestas de manera potencial pueden consistir en un conjunto de entidades definidas mediante plantillas XSD[42].

[41] Document message http://www.eaipatterns.com/DocumentMessage.html
[42] http://www.w3.org/XML/Schema



Cada contrato WSDL en la aplicación proveedora debería estar ubicado en la ruta relativa *"\src\main\wsdl"*. También es admitida la ubicación en *"\src\main\webapp\WEB-INF\wsdl"*.

En el ejemplo que vamos siguiendo este fichero se denominaría validafirma.wsdl.

El contrato se debe crear primero, en fase de diseño si es posible, y luego generar las clases de servicio y los ficheros de "*binding*". Esta técnica "*Top-Down*" es la adecuada en términos de contrato de servicio y está ampliamente recomendada[43] por la comunidad internacional.

10.2.1.1.1 Buenas prácticas en el diseño de los contratos WSDL

- Si es posible, se deberían utilizar XSD externos para las entidades que aparezcan en la petición o la respuesta.
- Se debe usar preferiblemente el estilo "*document/literal*"[44], se valorarán escenarios de excepción bien justificados.
- Es adecuado agrupar operaciones por contenido funcional.
- Usar restricciones de expresión regular (*pattern*), enumeraciones y los tipos básicos XSD[45] más próximos a los necesarios. Por ejemplo, es preferible usar *"dateTime"* en vez de *"string"*, o *"NMTOKEN"* para recibir series de cadenas que deben cumplir un patrón de expresión.
- Especificar la ocurrencia mínima y máxima de cada parámetro de petición/respuesta
- Comprobar la operatividad del "*endpoint*" en los entornos donde se realice el despliegue. Almacenar esas pruebas con herramientas como soapUI.

10.2.1.1.2 Tecnología de servicios Web

[43] http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/com.ibm.etools.webservice.doc/concepts/cwstopdown.html
[44] Estilo de WSDL http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/
[45] http://www.w3.org/TR/xmlschema-2/#built-in-datatypes

Asumiendo que para proyectos nuevos la plataforma J2EE es el servidor de aplicaciones WebSphere™ 7.0, la tecnología conveniente para la creación del adaptador de servicio Web deberá ser exclusivamente JAX-WS.

Sólo se permitirá el uso de tecnología JAX-RPC[46], para proyectos destinados a ser desplegados en WebSphere™ 6.1.

Los clientes de servicio utilizarán tecnología JAX-WS y/o JAX-RPC dependiendo de las aplicaciones consumidoras del servicio. Aunque en general se debería favorecer JAX-WS.

En el siguiente cuadro se relacionan los entornos y tecnologías que se deben utilizar.

Servidor de Aplicaciones	Adaptador	Cliente
WebSphere 7.0	JAX-WS	JAX-WS y JAX-RPC[47]
WebSphere 6.1	JAX-RPC	JAX-RPC

Tabla 1. Tecnología a aplicar en cada entorno de ejecución

10.2.1.2 Adaptadores REST

Los adaptadores REST ofrecen un conjunto de recursos identificados de forma unívoca por el URI en el que se ofrecen, accesibles a través del protocolo HTTP. Estos módulos están formados por los artefactos tecnológicos que permiten exponer un servicio de negocio mediante el protocolo HTTP, debiendo estar, al igual que los adaptadores para servicio web, protegidos contra invocaciones no autorizadas.

10.2.1.2.1 Transferencia de ficheros de gran tamaño con REST

En este punto se describe cual es la solución tecnológica a utilizar en caso de que se necesite transferir ficheros de gran tamaño utilizando la tecnología REST.

El objetivo principal que se quiere cubrir con esta solución consiste en permitir la transmisión de los datos de ficheros grandes en Streaming, evitando malas prácticas como por ejemplo introducir todo el fichero dentro del body de un único mensaje HTTP.

Dependiendo del servidor de aplicaciones donde se vaya a desplegar el servicio habrá que seleccionar una tecnología u otra:

[46] JAX-RPC
http://www.ibm.com/developerworks/library/ws-jaxrpc1/
[47] Bajo petición de aplicaciones consumidoras

- WAS7: Se utilizará la especificación JAX-RS 1.1.1 junto con su implementación Jersey (se recomienda Jersey 1.19.4)
- WAS9: Se utilizará la especificación JAX-RS 2.0 sin ninguna implementación concreta de forma que se delegue en Runtime en la implementación que incluye el servidor de aplicaciones.

A continuación se indican una serie de directrices que se deben seguir para implementar los distintos tipos de transacciones:

- Subida de fichero + metadatos (Cliente->Servidor):
  Este tipo de transacción nos permite el envío en Streaming de ficheros de cliente a servidor, teniendo también la posibilidad de enviar algunos metadatos asociados a estos ficheros, como por ejemplo el nombre de fichero.
  Se configurará como una operación POST que recibirá peticiones con el tipo MIME "multipart/form-data".
  Tanto en el lado Cliente como en el Servidor, se utilizará la librería 'commons-fileupload:commons-fileupload:' (ej. versión 1.2) para gestionar la transferencia de ficheros con Multipart. Seguidamente se ofrecen unas recomendaciones para el uso correcto de la librería "commons-fileupload".
  Cliente:
    - Se utilizará la clase "PostMethod", "FilePart" y "MultipartRequestEntity" para preparar la petición a enviar al servidor.
    - Con una referencia a un objeto de la clase "PostMethod" invocaremos el método "public void setRequestEntity(RequestEntity requestEntity)". A este método le pasaremos una referencia de un objeto de la clase "MultipartRequestEntity".
    - Para inicializar el objeto de la clase "MultipartRequestEntity" utilizaremos el constructror "public MultipartRequestEntity(Part[] parts, HttpMethodParams params)". A este constructor le pasaremos un array de FilePart y los parámetros del objeto de la clase "PostMethod" (usando el método "public HttpMethodParams getParams()").
    - Cada objeto de la clase FilePart se inicializará utilizando el constructor "public FilePart(String name, File file)" de forma que asociaremos al objeto File el metadato que representa su nombre de fichero (pudiendo recuperarse este dato en el lado del servidor).
    - Para enviar la petición POST al servidor utilizaremos una instancia de HttpClient (librería 'commons-httpclient: commons-httpclient: ') e invocaremos al método "public int executeMethod(HttpMethod method) throws IOException, HttpException" introduciendo como parámetro la referencia al objeto de la clase "PostMethod" que hemos utilizado en pasos anteriores.

  Servidor:
    - Se utilizará la clase "ServletFileUpload" para gestionar la recepción de la información. Esta clase se utilizará en conjunción con la clase "DiskFileItemFactory" que tiene como objetivo principal persistir temporalmente los ficheros recibidos sin pasar por la memoria (evitando desbordamiento del Heap de la JVM). El objeto de la clase "DiskFileItemFactory" se configurará apuntando a una ruta temporal donde se almacenarán los ficheros tal como vayan llegando. El umbral que se debe configurarar, a partir del cual un fichero se almacenará en memoria o se persistirá temporalmente será el representado por la propiedad "DiskFileItemFactory.DEFAULT_SIZE_THRESHOLD" (10240 bytes).

- Se parseará la petición utilizando el método "public List parseRequest(HttpServletRequest request) throws FileUploadException" de la clase "ServletFileUpload", obteniendo como resultado una lista de FileItem.
- Recorreremos la lista de FileItem y almacenaremos en su destino final cada uno de los FileItem. En este caso, un FileItem representa a un fichero que ha sido recibido y persistido temporalmente (si su tamaño supera el umbral). Para guardar de forma permanente un FileItem podremos utilizar el método "public void write(File)".
- De cada FileItem podremos recuperar datos adicionales (que han sido expresamente incrustados desde el cliente) tales como el nombre del fichero, utilizando el método "String getName()".

- Subida de fichero (Cliente->Servidor):
  A diferencia del anterior tipo de transacción, esta solo está pensada para transmitir en Streaming un fichero sin la posibilidad de enviar datos adicionales asociados a este.
  Se configurará como una operación POST que recibirá contenido (@Consumes) con el tipo MIME "application/octet-stream" (propiedad "APPLICATION_OCTET_STREAM" de la clase "MediaType").
  El servicio devolverá (@Produces) al cliente el estado final de la subida, esto se podrá hacer utilizando el tipo MIME "application/json" (propiedad "APPLICATION_JSON" de la clase "MediaType").
  En esta operación, una vez recuperado el InputStream de la petición, se persistirá el contenido recibido sin pasar por memoria para evitar que ante ficheros grandes se desborde el Heap de la JVM. Para persistir la información recibida sin pasar por memoria, se puede utilizar el método estático "public static void copyInputStreamToFile(InputStream source, File destination) throws IOException" de la clase "FileUtils" (librería 'commons-io:commons-io:').

- Descarga de fichero (Servidor->Cliente):
  Este tipo de transacción nos permite enviar desde el servidor hacia el cliente un fichero determinado que ha sido solicitado por el cliente.
  Se configurará como una operación POST que recibirá contenido (@Consumes) con el tipo MIME "application/json" (propiedad "APPLICATION_JSON" de la clase "MediaType"). De esta forma se pondrá indicar desde el cliente que fichero se quiere descargar.
  El servicio devolverá (@Produces) al cliente el fichero como un Stream de Bytes y para ello se utilizará el tipo MIME "application/octet-stream" (propiedad "APPLICATION_OCTET_STREAM" de la clase "MediaType").
  En esta operación, una vez que tengamos un objeto de la clase File que represente al fichero a descargar, devolveremos el Stream binario con los datos del fichero utilizando por ejemplo el método "public static Response.ResponseBuilder ok(Object entity)" de la clase Response de JAX-RS. En este caso también podemos transferir el nombre del fichero utilizando la cabecera HTTP "Content-Disposition" ya que se trata de una cabecera estandarizada para transportar un conjunto de metadatos sobre un fichero recibido en una respuesta HTTP. El nombre del fichero se podrá incrustar en el valor de dicha cabecera concatenando el nombre después de la cadena "attachment;filename=". Se puede encontrar más información sobre esta cabecera en la RFC6266.
  Desde el punto de vista del cliente, habrá que tener en cuenta que cuando se recibe la respuesta con el fichero (InputStream), se tendrá que utilizar la clase "FileUtils" de la librería 'commons-io:commons-io:' para que el fichero recibido no sea cargado en memoria (evitando des-

bordamiento del Heap de la JVM) sino que se cargue en el sistema de almacenamiento persistente que proceda.

10.2.1.3 **Adaptadores de presentación.**

Los adaptadores de presentación (capa de presentación de la aplicación) dirigen las interacciones de la interface de usuario hacia la lógica de negocio a través del puerto de servicio para las operaciones de alto nivel de abstracción como en el resto de los adaptadores comentados pero pudiéndose invocar, en caso de empaquetarse juntas ambas capas (si se desea diseñar el negocio en esta forma) a interfaces locales equivalentes tecnológicamente a las existentes en el puerto de servicio y que serán implementadas en la propia capa de negocio. ¿Cuál es la diferencia de este nivel de servicio con respecto al establecido en puerto de servicio? En principio este servicio interno (ver apartado posterior) debe ser el que maneje el dominio interno de la aplicación. Este dominio interno conforma la esencia de la aplicación y debería construirse en forma rica conteniendo todas las operaciones pertenecientes al mismo.

10.2.2 Puertos de servicio

El puerto de servicio será una interface local EJB.

La interfaz sólo puede contener operaciones anotadas como *"@Local"(*caso de implementarse en base a EJB*)* y las excepciones (caso de existir) de las operaciones definidas.

La interfaz estará segregada de tal manera que sólo contengan clases de interfaz con la signatura de las operaciones de negocio denominada NOMBRE_OPERACION. Se evitará, en la medida de lo posible, construir fachadas que engloben funcionalidades diversas bajo una única clase de interfaz. O dicho de otra forma; las interfaces deben ser cohesivas en la funcionalidad ofertada. Es preferible tener un mayor número de interfaces con pocas operaciones (pero relacionadas entre sí) que muchas operaciones no relacionadas en una única interface.

10.2.3 Implementación de lógica de negocio

Comprende la implementación de los puertos de servicio (como *beans* EJB o no, dependiendo de la tecnología usada), así como (opcionalmente) la capa de servicios internos de negocio que contienen en su firma el dominio interno (o propio) de la aplicación.

En estos módulos deberían encontrarse las implementaciones de los puertos de servicio (obligatoriamente), así como de los servicios de negocio internos (caso de haber decidido estructurar la aplicación de esta forma) que orquestan las llamadas a los servicios de datos necesarias para resolver las operaciones.

10.2.4 Capa de persistencia y Servicios de Acceso a Datos

En la capa de persistencia se hacen las llamadas a los Servicios de Acceso a Datos donde se consultan y modifican las bases de datos corporativas (ya sea relacionales o las de host, vía el componente común desarrollado para las mismas). Queda prohibido el uso de servicios de acceso a datos para acceder a otro tipo de sistemas externos (ldap, clientes de servicio externos, etc). Esto deberá realizarse a través de adaptadores de salida siguiendo el modelo de arquitectura hexagonal.

Se debe tener en cuenta que en esta capa no se puede invocar a servicios externos así como realizar operaciones de auditoría (son estas responsabilidades de la capa de negocio a través de los adaptadores de salida).

10.2.5 Capa de adaptadores de Salida.

En la capa de adaptadores de salida se hacen las llamadas a los servicios externos a la aplicación. La implementación consiste en el desarrollo de los clientes consumidores de dichos servicios.

De cara a simplificar la estructura de proyecto de esta capa se permite unificar en un único módulo tanto las interfaces como sus implementaciones de todos los clientes de servicios necesarios, delegando a la nomenclatura de paquetes la semántica para agrupar los servicios consumidos y la separación de interfaces e implementaciones.

10.2.5.1 **Cliente de servicio web**

Esta versión de la especificación elimina la obligatoriedad, por parte de las aplicaciones proveedoras, del desarrollo y publicación de los clientes de los servicios que ofrezcan en el repositorio de librerías reutilizables de la DGT.

Por lo tanto el Service Consumer tiene la responsabilidad de desarrollar la parte cliente del servicio que quiere consumir

10.2.5.2 **Declaración de puntos de servicio a través de JNDI**

El cliente de servicio y sus artefactos asociados no deben conocer la información de los "*endpoints*" de los servicios Web que se vayan a utilizar. Las URL que dan servicio no se deben explicitar ni en el código ni en ficheros de propiedades. Esto acarrearía serios problemas de mantenimiento en distintos entornos o entre distintas versiones de los servicios.

Para obtener la información del "*endpoint*" se debe usar un mecanismo de desacoplamiento que provee el servidor de aplicaciones con JNDI[48].

[48] JNDI

La aplicación proveedora tiene que declarar un recurso URL en el servidor de aplicaciones WebSphere™. Este recurso será la URL del "*endpoint*" para aquel entorno. En el documento de implantación de la aplicación proveedora se debe detallar cómo dar de alta todos los recursos URL necesarios para publicar sus servicios Web.Este parámetro se obtiene mediante un "*lookup*" del recurso creado en el servidor de aplicaciones. En el siguiente ejemplo muestra cómo se hace esta operación:

```
URL url = (URL) contexto.lookup (MI_END_POINT);
service = new MiServicioLocator().getServicio(url);
```

Código 21.Lookup de la URL del punto de servicio.

10.3 Criterio de seguridad para el acceso a Servicios de Negocio

Los servicios de negocio publicados en Internet e Intranet deben mantenerse con una configuración de seguridad apropiada, documentada y operativa. Para ello se deben usar los mecanismos y pautas de la especificación "*DGTE-002: Desarrollo de aplicaciones seguras en la plataforma Java™ EE*" (10). Se recomienda especialmente la lectura del capítulo 4 de esa especificación.

Se debe utilizar autenticación y transporte seguro en aquellos servicios expuestos para su invocación remota de cara a cumplir con la legislación vigente en materia de seguridad y normas complementarias (Esquema Nacional de Seguridad [49], Ley Orgánica de Protección de Datos (14), Política de firma de la DGT…).

El método de autenticación debe ser decidido en función de lo establecido en la política de firma de la DGT.

Este será, por tanto, el "**Criterio de Seguridad para Servicios de Negocio",** y servirá de referencia para los escenarios que se describen más adelante.

http://java.sun.com/products/jndi/

[49] http://www.csi.map.es/csi/pg5e42.htm

11 Apéndice A: Implementación de ejemplo

En este capítulo se describirá someramente un ejemplo de proceso de desarrollo de una aplicación Java™ EE que utilice esta especificación.

11.1 Requisitos

En este ejemplo se desarrollarán servicios para las siguientes aplicaciones:

- Gestión de Citas
- Generación de PDF
- Gestión de Personas
- Gestión de Tasas

En la imagen siguiente se describe el algoritmo a ejecutarse por el proceso de negocio de alta de cita.

11.2 Identificación de los servicios de negocio

El algoritmo del proceso de alta de una cita se detalla en la ilustración siguiente. El análisis del flujo de las operaciones de negocio, así como la identificación de los servicios a realizarse y los servicios existentes reutilizables, permitirá identificar los servicios de negocio que cada aplicación deberá proporcionar y sus principales características.

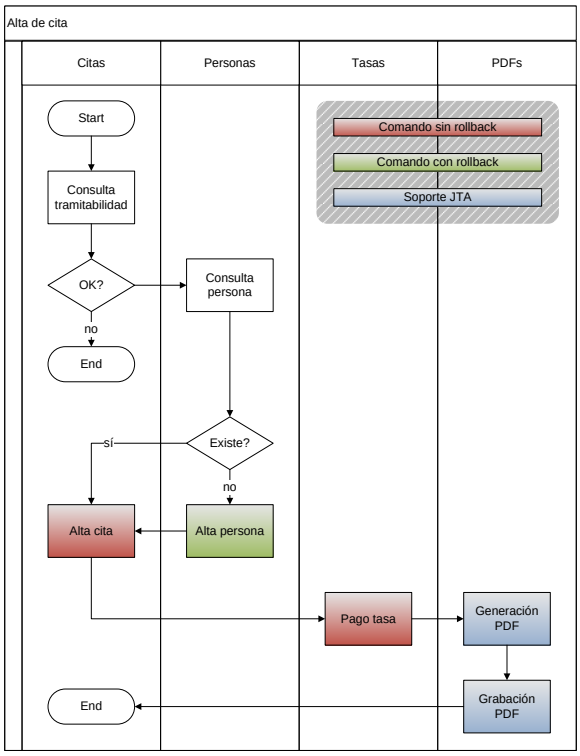


Ilustración 33 – Algoritmo del proceso de alta de cita

En el algoritmo ilustrado se han marcado con colores distintos los procesos que utilizan recursos transaccionales y los procesos cuyas transacciones deberán manejarse a través de implementaciones del patrón *comando*.

En este ejemplo se asume que no esté disponible ningún servicio y por lo tanto habrá que definir qué servicios de negocio y de acceso a datos tendrá que implementar cada aplicación.

En la siguiente tabla se detallan los servicios de negocio que se identifican en el algoritmo precedente.

Aplicación	Servicio de negocio
Exámenes	Alta examen Consulta de tramitabilidad
Conductores	Consulta conductor Alta conductor
Tasas	Pago de tasa
PDF	Generación PDF

Tabla 2 – Servicios de negocios proporcionados por las aplicaciones

Será responsabilidad de cada aplicación determinar qué servicios de negocio publicar a las demás aplicaciones. En este ejemplo, como mínimo, los servicios a publicarse son los siguientes:

Aplicación	Servicio de negocio remotos
Exámenes	Alta examen
Conductores	Consulta conductor Alta conductor
Tasas	Pago de tasa
PDF	Generación PDF

Tabla 3 – Servicios de negocios proporcionados por las aplicaciones

El único servicio que no es estrictamente necesario publicar al exterior es el servicio de consulta de tramitabilidad ya que ninguna de las aplicaciones está invocándolo.

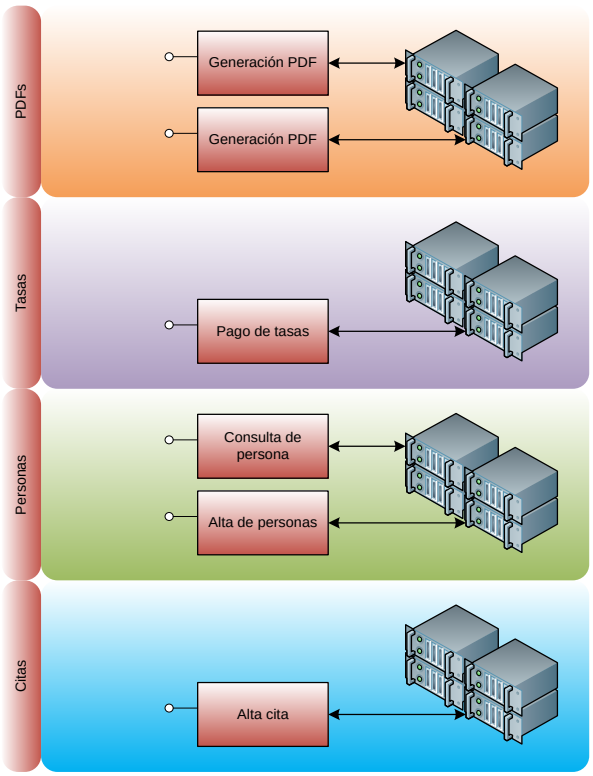


Ilustración 34 – Servicios de negocios remotos identificados

11.3 Definición de los servicios de negocio

Las operaciones de negocio remotas e imprescindibles detectadas en la sección precedente no necesariamente deberían implementarse en servicios de negocios distintos. En el caso de la aplicación de Personas, por ejemplo, podría implementarse un servicio de negocio cuya interfaz contuviera las dos operaciones necesarias:

- Consulta de persona.

- Alta de persona.

De hecho, el Departamento de Arquitectura de la Dirección General de Tráfico aconseja consolidar operaciones que funcionalmente puedan pertenecer a la misma interfaz de negocio. De esta manera se reducen el número de servicios de negocio y de interfaces publicadas. El caso de operaciones de creación, borrado y modificación sobre entidades de negocio, como es el caso de las operaciones necesarias en la aplicación de Personas, son candidatas ideales para pertenecer a una única interfaz de negocio que contenga las operaciones básicas sobre la entidad de negocio Persona.

Sin embargo, a solo a título de ejemplo, en este caso se implementarán interfaces y servicios de negocios distintos para cada operación.

11.3.1 Alta de cita

La interfaz de negocio de este servicio es la siguiente:

Método	Descripción
Cita altaCita(Cita cita)	Método de negocio para solicitar el alta de una cita. Se proporciona una instancia de la entidad de negocio Cita y el servicio devuelve una instancia de Cita poblada con la información proporcionada por el servicio.

Tabla 4 – Interfaz del servicio de negocio de alta de cita

11.3.2 Consulta de personas

La interfaz de negocio de este servicio es la siguiente:

Método	Descripción
Persona consultaPersona(Persona persona)	Método de negocio para hacer una consulta en la base de datos de personas. El método requiere una instancia de la entidad de negocio Persona en la cual se poblarán los campos correspondientes a los criterios de búsqueda. Los criterios de búsqueda serán campos que permitan identificar de forma univoca una persona. El método de negocio devolverá la instancia de persona encontrada en la base de datos o null si no se encuentra la persona solicitada.

Tabla 5 – Interfaz del servicio de negocio de consulta de personas

11.3.3 Alta de personas

La interfaz de negocio de este servicio es la siguiente:

Método	Descripción
void alta(Persona persona)	Método de negocio para solicitar el alta de una persona. Se proporciona una instancia de la entidad de negocio Persona poblada con los campos necesarios para solicitar el alta. El servicio no devuelve ningún objeto en el caso termine de ejecutarse con éxito. En caso contrario el método de negocio lanzará una excepción.

Tabla 6 – Interfaz del servicio de negocio de alta de cita

11.3.4 Pago de tasas

La interfaz de negocio de este servicio es la siguiente:

Método	Descripción
void pagar(Tasa tasa)	Método de negocio para solicitar el pago de una tasa. Se proporciona una instancia de la entidad de negocio Tasa poblada con los campos necesarios para solicitar el pago de una tasa. El servicio no devuelve ningún objeto en el caso en que termine de ejecutarse con éxito. En caso contrario el método de negocio lanzará una excepción.

Tabla 7 – Interfaz del servicio de negocio de pago de tasas

11.3.5 Creación y grabación de PDF

La interfaz de negocio de este servicio es la siguiente:

Método	Descripción
String createPDF(String info)	Método de negocio para solicitar la creación de un PDF. En este ejemplo se deja sin implementar y la firma del método no tiene ningún significado.
void savePDF(String pdf)	Método de negocio para solicitar la grabación de un PDF. En este ejemplo se deja sin implementar y la firma del método no tiene ningún significado.

Tabla 8 – Interfaz del servicio de negocio de creación y grabación de PDF

11.4 Implementación de los servicios de negocio

Tal y como se ve en la ilustración precedente, en el proceso de alta de cita participan servicios publicados por cuatro aplicaciones y tres de estos servicios utilizan *comandos*. Además, el mismo proceso de alta de cita en la aplicación de Citas se ejecuta a través de un comando sin flujo de compensación.

En este ejemplo se implementarán algunos servicios de negocio como EJB de tipo stateful, ya que a su vez deberán interaccionar con servicios de acceso a datos de esta tipología.

En este apéndice solo se ilustra la implementación de un servicio de negocio, el servicio de negocio de alta de cita

11.5 Estructura del componente EJB Alta de cita

El interfaz de negocio para el servicio de negocio de *alta de cita* es el siguiente:

```
@Local
public interface CitaLocal {
  public Cita altaCita(Cita cita);

  @Remove
  public void remove();
}
```

Código 22 – Interfaz de negocio del componente EJB del servicio de negocio de alta de cita

El prototipo para el componente EJB 3.0 que implemente este servicio podría ser el siguiente:

```
@Stateful
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class CitaBean implements CitaLocal{

  // inyección de dependencia de otros EJBs

  // inyección de dependencia del contexto de sesión
  @Resource
  SessionContext ctx;


  @Remove
  public void remove() {
    // método para remover el EJB
    // remover EJB stateful dependientes
  }

@TransactionAttribute(TransactionAttributeType.REQUIRED)
  public Cita altaCita(Cita cita) {
    // lógica de negocio
  }
}
```

Código 23 – Estructura básica del componente EJB del servicio de negocio de alta de cita

11.6 Inyección de otros componentes

La estructura de la implementación de la lógica de negocio de este servicio es la siguiente, asumiendo que los demás servicios de negocio estén disponibles, deberán comenzar con la inyección de dependencias de los componentes EJB utilizados por este componente:

```
@Stateful
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class CitaBean implements CitaLocal {

  @EJB
  private IPDFService pDFService;
  @EJB
  private IPersonas personas;
  @EJB
  private IConsulta consultaPersonas;
  @Resource
  SessionContext ctx;
  private ITramitacion tramitacion;
  private IGestionDeCitas gestionDeCitas;
  private IPagoTasas tasas;
```

Código 24 – Inyección de dependencia de los componentes EJB utilizados

Los componentes EJB utilizados por este componente se inyectan a través de campos anotados con la anotación **`@EJB`**. Siendo un componente *stateful*, es posible también inyectar referencias a componentes de tipo *stateful* a través de esta anotación. A título de ejemplo se utilizará también el mecanismo de lookup en el contexto de sesión del EJB de las referencias a componentes *EJB stateful*. Por esta razón, en el fragmento de código precedente, se inyecta el recurso de tipo **`SessionContext`** y se declaran tres referencias a las siguientes interfaces de negocio:

- ITramitacion
- IGestionCitas
- IPagoTasas

Obtener una referencia a un componente a través del contexto de sesión del componente EJB se efectúa a través de las siguientes invocaciones:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Cita altaCita(Cita cita) {
  tramitacion = (ITramitacion)
    ctx.lookup("ejb/ITramitacion");
  gestionDeCitas = (IGestionDeCitas)
    ctx.lookup("ejb/IGestionDeCitas");
  tasas = (IPagoTasas)
    ctx.lookup("ejb/IPagoTasas");
```

Código 25 – Recuperación de componentes del contexto de sesión del componente EJB

11.7 Lógica de negocio

La lógica de negocio de la aplicación implementará el algoritmo definido durante el análisis funcional del componente. El pseudo-código que implementa el algoritmo de la Ilustración 3351 es el siguiente:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Cita altaCita(Cita cita) {
  if (!tramitacion.consultaTramitacion(cita)) {
    return null;
  }

  Persona found =
    consultaPersonas.consultaPersona(
      cita.getPersona());
  if (found == null) {
    personas.alta(cita.getPersona());
  }

  Cita ret = gestionDeCitas.alta(cita);
  tasas.pagar(new Tasa());

  String pdfPlaceHolder = pDFService.createPDF("");
  pDFService.savePDF(pdfPlaceHolder);

  return ret;
}
```

Código 26 – Pseudo-código de la lógica de negocio del servicio de alta de cita

11.8 Consulta de tramitación

El servicio de acceso a datos *consulta de tramitación,* implementa la lógica necesaria para establecer las comunicaciones con el backend de almacenamiento de datos correspondiente y efectuar la búsqueda requerida.

Este componente será un componente *local*, como todo servicio de acceso a datos, e implementará la siguiente interfaz de acceso a datos:

```
@Local
public interface ITramitacion {
  public boolean consultaTramitacion(Cita cita);
}
```

Código 27 – Interfaz del servicio de acceso a datos de consulta de tramitación

El código de la implementación de este servicio de acceso a datos deberá utilizar un *comando*, ya que esta operación utiliza un recurso no transaccional para el cual se implementará el patrón *comando.*

Al ser un servicio de sola lectura, no será necesario implementar un flujo de *rollback* y el componente EJB correspondiente podrá ser s*tateless*.

El siguiente fragmento ilustra la estructura de este servicio:

```
@Stateless
public class TramitacionBean implements ITramitacion {

  public boolean consultaTramitacion(Cita cita) {
    ConsultaTramitacionCommand ctc =
      new ConsultaTramitacionCommand();
    ctc.setCita(cita);
    ctc.execute();
    return ctc.result();
  }
}
```

Código 28 – Estructura del servicio de acceso a datos de consulta de tramitación

11.8.1 Comando para la consulta host

La consulta contra el backend host se efectuará siguiendo las indicaciones del Departamento de Arquitectura de la Dirección General de Tráfico. La estructura del *comando* que implemente esta funcionalidad es el siguiente:

```
public class ConsultaTramitacionCommand
  implements Command {
  private Cita cita;
  private boolean result = false;

  public void execute() {
    CHostConsultaTramitacion consulta =
      new CHostConsultaTramitacion();
    result = consulta.isTramitable(cita);
    return;
  }

  public void rollback() {
  }

  public Cita getCita() {
    return cita;
  }

  public void setCita(Cita cita) {
    this.cita = cita;
  }

  public boolean result() {
    return result;
  }
}
```

Código 29 – Estructura del comando para efectuar la consulta de tramitación lado HOST

11.9 Remoción de componentes

Para efectuar correctamente la limpieza del entorno de ejecución, se puede implementar un método del interfaz de negocio de los componentes *EJB stateful* para que comuniquen al contenedor de componentes EJB que el EJB está listo para ser removido.

Un posible *comando* es el siguiente:

```
@Remove
public void remove() {
  if (gestionDeCitas != null) {
    gestionDeCitas.confirm();
  }
  if (tasas != null) {
    tasas.remove()
  }
}
```

Código 30 – Comando @Remove de un componente EJB stateful

El comando del ejemplo precedente está marcado con la anotación **`@Remove`** para que el contenedor libere de inmediato el correspondiente EJB en cuanto este método se ejecute con éxito. En esta implementación de ejemplo este método a su vez invoca un método **`@Remove`** de todo EJB stateful inyectado dentro de sí mismo.

11.10 Atributos de control de transacción

Los componentes utilizan el modelo de transacciones declarativas. Por lo tanto es suficiente utilizar los atributos de las transacciones en los métodos de negocio de las interfaces de los componentes EJB.

En el caso del componente que implementa el servicio de negocio de *alta de cita*, por ejemplo, se ha declarado un atributo de transacción por omisión **`NOT_SUPPORTED`**:

```
@Stateful
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class CitaBean implements CitaLocal {
```

Código 31 – Atributos de control de transacción en el EJB que implementa el servicio de alta de cita

El método de negocio de la interfaz de negocio del componente EJB, al contrario, requiere que una transacción esté en curso para poderse ejecutar:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Cita altaCita(Cita cita) {
```

Código 32 – Atributos de control de transacción en el método de negocio del EJB que implementa el servicio de alta de cita

11.11 Descriptor de despliegue

El descriptor de despliegue de un componente EJB 3.0 (3) es opcional. En este caso, ya que se están recogiendo referencias a componentes EJB stateful a través del contexto de sesión del componente EJB, es aconsejable desacoplar el nombre utilizado en el contexto del componente EJB y el nombre utilizado en fase de despliegue, o el nombre por omisión, en el descriptor de despliegue del componente EJB:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-
jar_3_0.xsd">
<enterprise-beans>
  <session>
    <ejb-name>CitaBean</ejb-name>
    <ejb-ref>
      <ejb-ref-name>ejb/IPagoTasas</ejb-ref-name>
      <remote>
        es.trafico.dgte_001.tasas.IPagoTasas
      </remote>
    </ejb-ref>
    <ejb-local-ref>
      <ejb-ref-name>
        ejb/IGestionDeCitas
      </ejb-ref-name>
      <local>
     es.trafico.dgte_001.citas.datos.IGestionDeCitas
      </local>
    </ejb-local-ref>
    <ejb-local-ref>
      <ejb-ref-name>ejb/ITramitacion</ejb-ref-name>
      <local>
        es.trafico.dgte_001.citas.datos.ITramitacion
      </local>
    </ejb-local-ref>
  </session>
</enterprise-beans>
</ejb-jar>
```

Código 33 – Ejemplo de descriptor de despliegue para el componente EJB de alta de cita

11.12 Tabla de decisiones

Problema	Solución
Crear un servicio de negocio	Implementar un componente EJB v. 3.0 stateless
Crear un servicio de acceso a datos	Implementar un componente EJB v. 3.0 stateless
Crear un servicio de acceso a datos que utilice recursos no transaccionales en escritura	Encapsular los flujos de compensación en implementaciones de la interfaz **`Command`**. Implementar un componente EJB v. 3.0 stateful que extienda la clase **`EJBCommandSupport`** y que invoque dichos comandos a través del método **`invokeCommand`**.
Crear un servicio de acceso a datos que utilice recursos no transaccionales en lectura	Encapsular el método de lectura en una implementación de la interfaz **`Command`**. Implementar un componente EJB v. 3.0 stateful que extienda opcionalmente la clase **`EJBCommandSupport`** y que invoque dichos comandos a través del método **`invokeCommand`**. También se podrá realizar un EJB stateless e utilizar el comando directamente en el caso dicho comando no requiera flujos de compensaciones ya que el acceso es en sólo lectura.
Crear un servicio de negocio que utilice servicios de acceso a datos stateful	Implementar un componente EJB v. 3.0 stateful con un método @Remove en el cual se invocan los métodos @Remove de los componentes EJB stateful que se utilizan.

Tabla 9 – Tabla de decisiones

12 Bibliografía

1. **Shannon, Bill (Sun Microsystems).** JSR 244: Java Platform, Enterprise Edition 5 (Java EE 5) Specification. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=244.

2. **Rao, Sankara (Sun Microsystems).** JSR 907: Java Transaction API. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=907.

3. **DeMichiel, Linda (Sun Microsystems) y Keith, Michael (Oracle).** JSR 220: Enterprise JavaBeans 3.0. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=220.

4. **Cockburn, Alistair.** Arquitectura hexagonal. *http://alistair.cockburn.us.* [En línea] 2005. http://alistair.cockburn.us/Hexagonal+architecture.

5. **Gamma, Erich, y otros, y otros.** *Design patterns: elements of reusable object-oriented software.* s.l. : Addison Wesley, 1995. 0201633612.

6. **Knoernschild, Kirk.** *Modularity patterns (PATTERN CATALOG).*

7. **Sun Microsystems.** Java Transaction Service (JTS). *Sun Developer Network (SDN).* [En línea] http://java.sun.com/javaee/technologies/jts/index.jsp.

8. **Oracle.** *http://www.jcp.org/en/jsr/detail?id=237.*

9. **Evans, Eric.** *Domain driven design.* s.l. : Addison-Wesley. ISBN: 0-321-12521-5 .

10. **Crisóstomo, Enrico.** *DGTE-002: Desarrollo de aplicaciones segu-ras en la plataforma Java2 EE.Modelo declarativo de la seguridad.* Madrid : DGT, 2010.

11. **Chinnici, Roberto (Sun Microsystems, Inc.) y Shannon, Bill (Sun Microsystems, Inc.).** JSR 316: Java Platform, Enterprise Edition (Java EE 6) Specification. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=316.

12. **Mordani, Rajiv (Sun Microsystems).** JSR 250: Common Annotation for the Java Platform. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=250.

13. **DGT.** *DGTE-002: Desarrollo de aplicaciones seguras en la plataforma Java™ EE.* Madrid : DGT, 2010.

14. **BOE.** *Ley Orgánica 15/1999 de 13 de diciembre de Protección de Datos de Carácter Personal, (LOPD).* Madrid : BOE, 1999.

15. **Object Management Group.** OMG's CORBA Website. *OMG's CORBA Website.* [En línea] http://www.corba.org/.

16. **Saks, Kenneth (Sun Microsystems, Inc.).** JSR 318: Enterprise JavaBeans 3.1. *Java Community Process.* [En línea] http://jcp.org/en/jsr/detail?id=318.

17. **Navarro, David.** *DGTE-004: Integración y construcción de servicios web.* Madrid : DGT, 2010.

18. **Oracle-Sun.** JSR 220: Enterprise JavaBeansTM 3.0. [En línea] 11 de 5 de 2006. http://jcp.org/en/jsr/detail?id=220.

19. —. JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0. [En línea] 11 de 5 de 2006. http://jcp.org/en/jsr/detail?id=224.

20. **Hohpe, Woolf.** *Enterprise Integration Patterns.* s.l. : Adisson Wesley, 2004.

21. **Group, Object management.** Reusable Asset Specification. *http://www.omg.org/.* [En línea] 27 de 4 de 2009. http://www.omg.org/technology/documents/formal/ras.htm.

22. **Calidad, Departamento de.** *Guía de Desarrollo.* Madrid : DGT, 2010.

23. **Cockburn, Alistair.** Arquitectura hexagonal. [En línea] 2009. http://alistair.cockburn.us/Hexagonal+architecture.

24. **Oracle-Sun.** java Community Process. *JSR 220: Enterprise JavaBeansTM 3.0.* [En línea] 11 de 5 de 2006. http://jcp.org/en/jsr/detail?id=220.

25.

[i] Plain Old Java Objects