Directrices Arquitectura SW

Oficina de Calidad

Versión: 1.1.1

Control de versiones

Versión	Fecha	Autor	Descripción / Comentarios
V01_v01	01-09-2017	Oficina de Calidad	Versión inicial
1.1.1	22-02-2021	Oficina de Calidad	Pie de página y documentos relacionados

Índice

1. INTRODUCCIÓN 5
1.1. OBJETO 5
1.2. ALCANCE 5
1.3. GLOSARIO DE TÉRMINOS 5
1.4. DOCUMENTOS RELACIONADOS 5
2. DIRECTRICES BÁSICAS DE DISEÑO. 7
2.1. PRINCIPIOS SOLID 7
2.1.1. Principio de responsabilidad única (Single Responsibility Principle) 7
2.1.2. Principio abierto/cerrado (Open/Close Principle) 8
2.1.3. Principio de sustitución de Liskov (Liskov Substitution Principle) 8
2.1.4. Principio de Segregación de la Interface (Interface Segregation Principle) 9
2.1.5. Principio de Inversión de Dependencia (Dependency Inversion Principle) 9
2.2. DOMAIN DRIVEN DESIGN (DDD) 9
2.3. TEST DRIVEN DEVELOPMENT (TDD) 10
3. ARQUITECTURA MIR 10
3.1. VISIÓN GENERAL 11
3.2. DESCRIPCIÓN DE CAPAS 12
3.2.1. Capa de Presentación 12
3.2.2. Capa de Adaptadores de Entrada (Servicios) 12
3.2.3. Capa de Negocio (DOMINIO) 16
3.2.4. Capa de Persistencia (DAO) 17
3.2.5. Capa de Salida (Integración Externa) 17
3.2.6. Capa Transversal. 18
3.3. COMUNICACIÓN ENTRE CAPAS 18
4. NOMENCLATURA DE OBJETOS 18
4.1. MÓDULOS 18
4.2. PAQUETES 19
4.3. RESUMEN DE NOMENCLATURA DE MÓDULOS Y PAQUETES 20
5. PARAMETRIZACIÓN Y CONFIGURACIÓN DE LAS APLICACIONES 21

Índice de ilustraciones

Ilustración 1. Visión general de la arquitectura 11
Ilustración 2 Comunicación entre capas 18
Ilustración 3 Esquema de nomenclatura de módulos y paquetes 20

Índice de tablas

Tabla 1 Clasificación de las operaciones por idempotencia 14
Tabla 2 Códigos HTTP de respuesta 15

1. INTRODUCCIÓN

1.1. OBJETO

El objeto de este documento es establecer las directrices de diseño y la definición a alto nivel de la Arquitectura lógica que deben seguir las aplicaciones desarrolladas para el Ministerio del Interior, así como la línea de actuación que se va a seguir desde el departamento de Calidad para velar por su cumplimiento. Queda fuera del alcance de este documento el establecimiento de tecnologías permitidas, cuya definición se realizará en fases posteriores y que servirán de nuevas entradas para retroalimentar el contenido de este documento, estableciendo nuevas directrices que deberán ser tenidas en cuenta por los equipos de desarrollo.

1.2. ALCANCE

Este documento va dirigido a todas aquellas personas involucradas en el ciclo de vida de desarrollo de aplicaciones para la Subdirección General de Tecnologías de la Información y las Comunicaciones

1.3. GLOSARIO DE TÉRMINOS

Término	Etiquetas	Descripción
MIR		*Ministerio del Interior Gobierno de España*
SOLID		*Conjunto de buenas prácticas o principios básicos del diseño y de la programación orientada a objetos*
DDD		*Acrónimo de "Domain Driven Design": Diseño guiado por el dominio*
TDD		*Acrónimo de "Test Driven Development": Desarrollo guiado por pruebas de software*
AGE		*Administración General del Estado*
CRUD		*Acrónimo de "Create, Read, Update and Delete", que se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.*
MOCK		*Objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos que esperan mensajes de una clase en particular para sus métodos.*
POJO		*Acrónimo de Plain Old Java Object: clases simples que no dependen de un framework en especial.*

1.4. DOCUMENTOS RELACIONADOS

/Ubicación/Documento	Descripción
PublicacionIntranet/plantillas/doc/02.DSI/MIR-APLIC-DSI-WS-XXXXX.docx	Entregable de Integración de Servicios WEB
PublicacionIntranet/Doc_Especifica_Subsecretaria/MIR-INT-DSI-	Normativa de parametrización y configuración de aplicaciones WEB

PARAMETRIZACION APLICACIONES.docx	
PublicacionIntranet/Doc_Especifica_Subsecretaria/MIR-ANEXO-NORMA-USO-CERTIFICADO-DIGITAL-APLIC-JAVA.docx	Normativa de uso de certificado en las aplicaciones java
PublicacionIntranet/plantillas/doc/02.DSI/MIR-APLIC-DSI-ARQUITECTURA.docx	Entregable de Arquitectura
PublicacionIntranet/02.DSI/MIR-INT-DSI-NORMA-MODELO_DATOS.docx	Normativa y documentación del modelo de datos
PublicacionIntranet/plantillas/doc/02.DSI/MIR-APLIC-DSI-INTEGRACION-CC-XXXXX.docx	Entregable de integración de componentes comunes
PublicacionIntranet/plantillas/doc/02.DSI/MIR-APLIC-DSI-DESCRIPCION_SCRIPTS.docx	Descripción de scripts de base de datos y de ejecución de procesos y

2. DIRECTRICES BÁSICAS DE DISEÑO.

Con el objeto de estandarizar los desarrollos realizados para el Ministerio del Interior y garantizar que el diseño de las aplicaciones cumple con unos requisitos mínimos de calidad, se han establecido unas directrices básicas de diseño enfocadas a obtener un producto software robusto, altamente cohesivo y débilmente acoplado, capaz de adaptarse al cambio con el menor coste de desarrollo posible, facilitando las tareas de mantenimiento y abstrayéndolo de los cambios tecnológicos que rodeen al núcleo del negocio.

2.1. PRINCIPIOS SOLID

Los principios SOLID, son un conjunto de buenas prácticas o principios básicos del diseño y de la programación orientada a objetos, impulsados por *Robert C. Martin*:

- Produce código más fácil de mantener y ampliar
- Evita tener que refactorizar continuamente código
- Forma parte de los desarrollos ágiles y la programación adaptativa
- Se debe utilizar con el desarrollo guiado por pruebas

Nos ayuda a escribir código de calidad:

- Bajo acoplamiento
- Alta cohesión
- Reutilizable
- Fácil de cambiar/evolucionar
- Fácil de probar

2.1.1. Principio de responsabilidad única (Single Responsibility Principle)

"*Las clases deben tener sólo una responsabilidad":* nunca debe haber más de una razón para cambiar una clase. Una clase debe concentrarse en hacer sólo una cosa.

La arquitectura propuesta, promociona esta idea en base al ejercicio de descomposición modular que sugiere, al distribuir las responsabilidades en torno a un hexágono de cualquier número de aristas.

Indicadores de un posible incumplimiento de este principio:

- En una misma clase están involucradas dos capas de la arquitectura. Si mezclamos responsabilidades de dos capas en una misma clase, será un buen indicio de incumplimiento de este principio.
- El número de métodos públicos: Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos. Se debe de analizar la estrategia adecuada para agruparlos y separarlos en distintas clases.
- Los métodos que usan cada uno de los campos de esa clase: si tenemos dos campos, y uno de ellos se usa en unos cuantos métodos y otro en otros cuantos, esto puede estar indicando que cada campo con sus correspondientes métodos podrían formar una clase independiente. Normalmente habrá métodos en común, así que esas dos nuevas clases tendrán que interactuar entre ellas.
- Número de imports: Si es necesario importar un número elevado de clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más.

- Dificultad para probar la clase: si encontramos dificultad en escribir tests unitarios sobre la clase, o no conseguimos el grado de granularidad que nos gustaría, se debe valorar la opción de dividir la clase en dos.
- Modificaciones constantes: Cada vez que escribes una nueva funcionalidad, esa clase se ve afectada. Si una clase se modifica a menudo, es porque está involucrada en demasiadas cosas y por lo tanto se debe de dividir en otras clases más especialistas.
- Número de líneas: Si una clase es demasiado grande, se debe dividirla en clases más manejables.

2.1.2. Principio abierto/cerrado (Open/Close Principle)

"*Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación".* Tenemos que ser capaces de extender el comportamiento de nuestras clases sin necesidad de modificar su código. Esto nos ayuda a seguir añadiendo funcionalidad con la seguridad de que no afectará al código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.

Este principio se suele resolver utilizando polimorfismo. En vez de obligar a la clase principal a saber cómo realizar una operación, la delega a los objetos que utiliza, de tal forma que no necesita saber explícitamente cómo llevarla a cabo. Estos objetos tendrán una interfaz común que implementarán de forma específica según sus requerimientos.

Tener código cerrado a modificación y abierto a extensión nos da la máxima flexibilidad con el mínimo impacto.

En la arquitectura propuesta este principio se cumple por medio del uso de los puertos y adaptadores. Si en un momento determinado deseamos cambiar o extender una dependencia sólo tenemos que crear un nuevo adaptador para conectarlo al puerto correspondiente sin que eso implique cambio alguno en el núcleo funcional de la arquitectura.

Indicadores de un posible incumplimiento de este principio:

Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio.

2.1.3. Principio de sustitución de Liskov (Liskov Substitution Principle)

"Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa".

Si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la clase padre.

El principio de Liskov nos ayuda a utilizar la herencia de forma correcta, y a tener mucho más cuidado a la hora de extender clases.

La arquitectura propuesta sigue esta idea en la definición de puertos y adaptadores. Cada puerto es exactamente un punto de extensión y cada adaptador una variante de sustitución.

Indicadores de un posible incumplimiento de este principio:

Si un método sobrescrito no hace nada o lanza una excepción, es muy probable se esté violando el principio de sustitución de Liskov. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

Para cumplir con este principio, lo más habitual es ampliar esa jerarquía de clases. Podemos extraer a otra clase padre las características comunes y hacer que la antigua clase padre y su hija hereden de ella.

2.1.4. Principio de Segregación de la Interface (Interface Segregation Principle)

"Muchas interfaces cliente específicas son mejores que una interfaz de propósito general". Cuando se crean interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas.

El principio de segregación de interfaces nos ayuda a no obligar a ninguna clase a implementar métodos que no utiliza. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas.

La definición de los puertos de la arquitectura propuesta, se realiza de acuerdo a estas ideas de segregación.

Indicadores de un posible incumplimiento de este principio:

Si al implementar una interfaz, uno o varios de los métodos no tienen sentido y es necesario dejarlos vacíos, es muy probable que se esté violando este principio. Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos, sin importar que una clase necesite implementar varias interfaces, pues lo más importante es que use todos los métodos definidos por esas interfaces.

2.1.5. Principio de Inversión de Dependencia (Dependency Inversion Principle)

"*Depender de abstracciones y no depender de concreciones*". El código que es el núcleo de nuestra aplicación no debe depender de los detalles de implementación, como pueden ser el framework utilizado, modelo de persistencia, … Todos estos aspectos se deben de especificar mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

El proceso de desacoplamiento masivo que promueve la arquitectura propuesta va exactamente en esta dirección, ya que cada puerto es un punto de extensión definido en términos abstractos a partir del cual se expresan las colaboraciones algorítmicas que orquesta el cuerpo funcional aislado de nuestro aplicativo.

Indicadores de un posible incumplimiento de este principio:

Cualquier instancia directa de clases complejas o módulos es una violación de este principio.

2.2. DOMAIN DRIVEN DESIGN (DDD)

"Domain driven design" no es una tecnología ni una metodología, es básicamente un enfoque para el desarrollo de software que fue introducido por *Eric Evans* en su libro *"Domain-Driven Design: Tackling Complexity in the Heart of Software.* Su objetivo principal es facilitar la creación de aplicaciones complejas centrándose en la definición de un modelo orientado a objetos que sea

rico, expresivo y en constante evolución que refleje el conocimiento del dominio y que sea independiente de cualquier tipo de comunicación con elementos de infraestructura. DDD no requiere la utilización específica de ningún tipo de patrón, framework, tecnología o arquitectura: la verdadera importancia reside en su modelo de dominio, independientemente de cómo se comunique con el resto de componentes de la aplicación, siempre y cuando se cumplan las buenas prácticas y los principios de calidad apropiados.

DDD se centra en tres principios fundamentales:

- Se debe de poner el foco en el **dominio** (área funcional a la que un usuario aplica un software "[Negocio]") y en la **lógica del dominio** (terminología y conceptos clave del dominio. Identifica los atributos y las relaciones entre las entidades pertenecientes al ámbito del dominio).
- Complejidad del diseño en el modelo de dominio.
- Colaborar constantemente con expertos en dominio, con el fin de mejorar el modelo de aplicación y resolver cualquier problema.

Las características que debe de tener el modelo de dominio que diseñemos son las siguientes:

- Alineado con el Negocio y sus procesos.
- Aislado de otras capas de la aplicación.
- Reusable.
- Fácil de mantener, versionar y probar.
- Sin (o con mínimas) dependencias con frameworks de infraestructura.
- Diseñado con un modelo de programación basado en POJO: independientes de cualquier framework de desarrollo.

2.3. TEST DRIVEN DEVELOPMENT (TDD)

TDD es un proceso de desarrollo que consiste en codificar pruebas, desarrollar y refactorizar de forma continua el código construido que se basa en todos los principios SOLID y facilita su cumplimiento. Es muy recomendable seguir el proceso de desarrollo TDD para realizar un buen diseño de las aplicaciones, garantizando la existencia de una buena documentación orientada al dominio y la robustez de las aplicaciones, gracias a la facilidad para alcanzar un elevado porcentaje de cobertura de pruebas unitarias.

3. ARQUITECTURA MIR

Con el fin de facilitar el cumplimiento de las directrices básicas de diseño expuestas en el punto anterior, se ha optado por un modelo de N-Capas basado principalmente en el modelo de *Arquitectura Hexagonal* propuesto por *Alistair Cockburn*, apostando por el cumplimiento de los principios S.O.L.I.D. El modelo de capas se basará en el patrón MVC (Modelo-Vista-Controlador) y en el patrón de arquitectura hexagonal (también llamado de puertos y adaptadores).

El modelo propuesto, encaja con las ideas que establece Domain-Driven Design (DDD) pues fomenta que el Dominio (Negocio) de la aplicación sea el centro de todas las capas y que no se acople a ningún agente externo ni a ninguna tecnología.

El objetivo principal que persigue este enfoque, se basa en estructurar las aplicaciones como componentes cohesivos, tanto funcional como tecnológicamente, con separación clara de responsabilidades, tratando de minimizar las dependencias entre ellos. Dichas dependencias se resolverán mediante abstracciones a través de interfaces, independizando a los clientes de sus implementaciones. De esta manera, se facilitará el mantenimiento y la evolución de las

aplicaciones hacia los nuevos paradigmas y/o frameworks que puedan surgir en el desarrollo de aplicaciones empresariales.

Se debe de tener en cuenta que el enfoque de la arquitectura hexagonal puede que añada excesiva complejidad a aplicaciones de tamaño reducido y escaso valor de negocio (como pudiera ser el caso de aplicaciones de administración tipo CRUD, frontales web que sólo orquestan llamadas a servicios sin aportar valor de negocio, ...) para este tipo de aplicaciones se podría dejar libertad (ordenada) de diseño de su arquitectura interna, favoreciendo el uso de frameworks que agilicen el desarrollo, velando por el cumplimiento de las directrices de construcción, mediante el análisis de código realizado por Sonarqube.

3.1. VISIÓN GENERAL

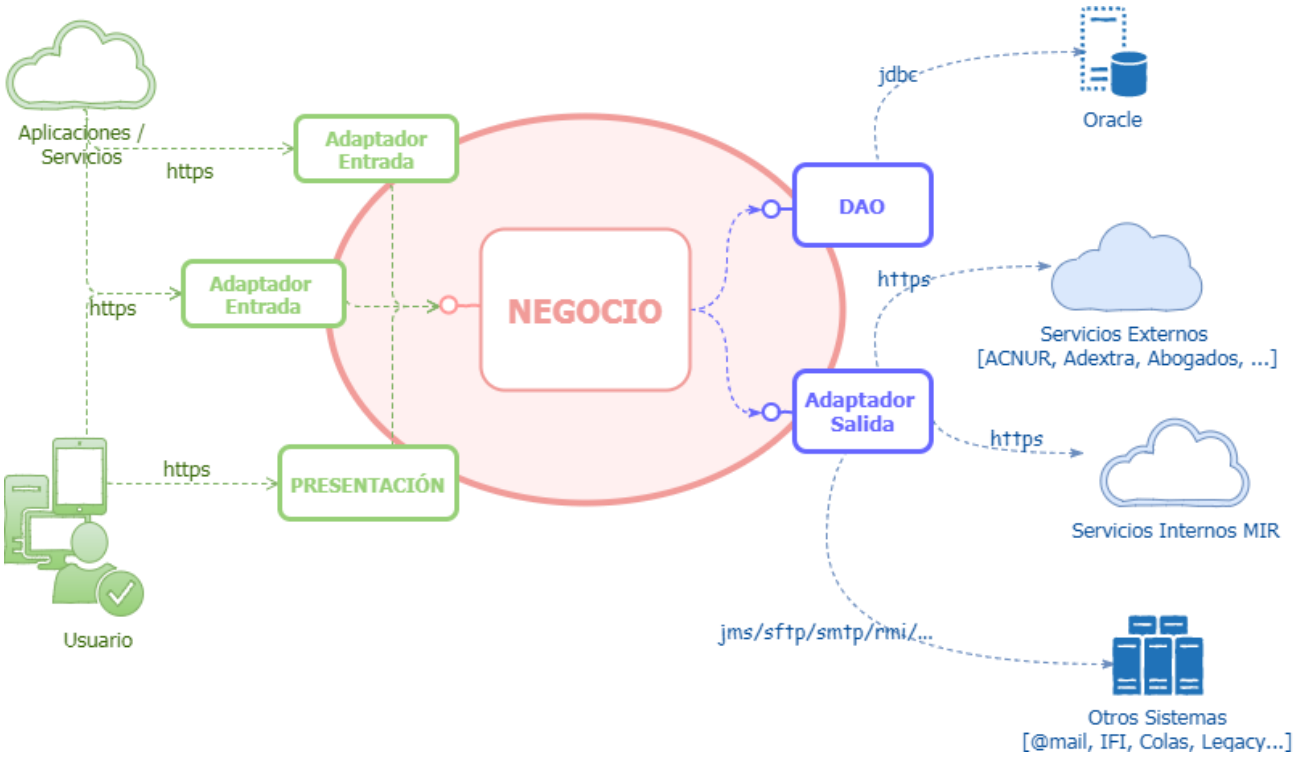

Ilustración 1. Visión general de la arquitectura

Beneficios del enfoque

- **Comprensión:** Facilita la comprensión del dominio (negocio).
- **Atemporalidad:** Abstrae al dominio de las dependencias tecnológicas y de frameworks.
- **Testabilidad:** Facilidad de testar la aplicación sustituyendo la implementación real de acceso a datos y/o servicios por mocks (gracias a la aplicación del Principio de Inversión de Dependencias de SOLID)
- **Facilitador del desarrollo dirigido por test (TDD)**, al estar basada en los contratos establecidos por los puertos que definamos, lo que permite posponer decisiones a nivel de implementación, sin parar el desarrollo actual.
- **Adaptabilidad**: Facilita los cambios de la aplicación para adaptarse a cambios de las reglas de negocio. Alta tolerancia al cambio gracias a la aplicación del Principio de Abierto/Cerrado de SOLID

- **Alta reutilización de código** debido a la división estricta de responsabilidades a nivel de servicios de aplicación, infraestructura y de dominio (Principio de Responsabilidad Única de SOLID)

3.2. DESCRIPCIÓN DE CAPAS

3.2.1. Capa de Presentación

Su responsabilidad es generar la interfaz de usuario y permitir a ésta, interactuar con el Sistema. Esta capa sólo puede invocar a la capa de Negocio a través de la interfaz del servicio de Negocio o desde un adaptador de entrada adecuado si la capa de presentación se encuentra desplegada en una unidad de despliegue diferente a la capa de Negocio.

3.2.1.1 Reglas generales de la capa

1. Utilizar patrón MVC (Model-View-Cotroller).
2. Se deben respetar los requisitos de Accesibilidad de la aplicación. Se debe valorar si la tecnología escogida para implementar esta capa tiene alguna incompatibilidad con los requisitos de accesibilidad que tiene que cumplir la aplicación.
3. El diseño de la interfaz debe ser adaptativo (*Responsive-design*) para permitir a la aplicación ser accesible desde múltiples dispositivos. Es recomendable para alcanzar este objetivo, seguir una estrategia de diseño "*Mobile-First*".
4. Las vistas deben ser simples y no deben contener lógica. Su responsabilidad será únicamente formateo y presentación de datos, delegando cualquier tipo de lógica al controlador.
5. Se debe evitar el envío duplicado de formularios y evitar ataques CSRF. Se recomienda el uso de soluciones basadas en el *token* sincronizador que implementan la mayoría de los frameworks de presentación e inhabilitar la opción de envío y/o utilizar otras técnicas de diseño gráfico, para dar feedback al usuario que la petición se está procesando, mejorando con ello la experiencia de usuario.
6. Los campos del formulario se deben de validar tanto en cliente (mejora la experiencia de usuario) como en servidor (seguridad de backend)
7. Debe limitarse el uso de sesiones http a guardar sólo la información imprescindible cuando no se pueda resolver de otro modo.
8. No sé debe utilizar la caché del cliente para almacenar información relevante. Si por las necesidades de la aplicación fuera necesario, dicha información debería ser cifrada antes de ser almacenada.
9. Esta capa debe implementarse para dar soporte a la internacionalización y dar opción de añadir nuevos idiomas y mejorar traducciones sin necesidad de modificar el código de esta capa.
10. Experiencia de usuario. Para el diseño de esta capa, se recomienda enfoques SPA (Single-Page-Application) para hacer más fluida la experiencia de usuario, siempre y cuando sea compatible con los requisitos Funcionales y No Funcionales de la aplicación.

3.2.2. Capa de Adaptadores de Entrada (Servicios)

Esta capa está basada en Servicios Web (SOAP/REST): figurarán los servicios que se exponen al resto de aplicaciones. Todo servicio que ofrezca una interfaz remota debe formar parte de esta capa.

Desde esta capa sólo se podrá invocar a la capa de Negocio a través de la interfaz del servicio de negocio.

3.2.2.1 Reglas generales de la capa

3.2.2.1.1 Interfaz de Servicios orientadas a Negocio.

a. La información del contrato debe tener únicamente información de Negocio. Los datos técnicos (datos de autenticación, ip cliente, definición de formatos del mensaje…) se deben de obtener de la petición http (ws-security y/o cabeceras http apropiadas, …)
b. Agrupar la información en agrupaciones lógicas (por ejemplo domicilio) para mejorar la legibilidad del contrato.

3.2.2.1.2 SOAP. Reglas definición wsdl

a. Separar el dominio del contrato (XML Schema) en un fichero xsd independiente:
    i. Reduce el tamaño y la complejidad del fichero wsdl.
    ii. Permite reutilizar schemas y namespaces.
b. Se debe definir los mensajes con las restricciones de los schemas XSD, para que el dato pueda ser validado contra el schema definido y no trasladar esta responsabilidad al código.
c. Ficheros adjuntos: se debe hacer a nivel de http attachment y no como un elemento del mensaje XML.
d. El wsdl debe cumplir con el estándar WS-I Basic Profile.
e. Se recomienda escribir primero el wsdl (*contract-first*) con el objetivo de obtener una interoperabilidad real e independiente de la tecnología: los generadores de WSDL pueden introducir dependencias con una tecnología concreta.

3.2.2.1.3 REST. Reglas de definición

a. El api REST debe estar orientado a recursos y se debe seguir el enfoque RESTful en su definición e implementación.
b. Una vez publicada la API, no puede perder su compatibilidad con versiones anteriores, para evitar esta situación la API se debe versionar.
c. La versión de la API debe de aparecer en la URI.
d. El protocolo a usar será siempre HTTPS.
e. Peticiones: El cuerpo de las peticiones POST, PUT y PATCH deberán realizarse mediante el uso de JSON y por lo tanto debe validar que la cabecera http *Content-Type* tenga el valor application/json, en caso contrario debe responderse con un código de status HTTP 415 (Unsupported Media Type). Las peticiones de los clientes deben incluir la cabecera *Accept* indicando el tipo de respuesta (JSON, XML o ambos) que soportan.
f. Respuestas: El cuerpo del mensaje tendrá el formato (JSON/XML) requerido en la cabecera *Accept* incluida en la petición. Se recomienda usar pretty-print para mejorar la legibilidad.
g. Documentación de la API REST: se deberá documentar mediante Swagger (OpenAPI Specification)
    iii. Deben tener ejemplos completos de ciclos petición/respuesta.
    iv. Debe avisar del versionado de la API, así como la planificación de versiones a ser deprecadas.

h. Debe cumplir con la nomenclatura de recursos definida en este documento. Si la nomenclatura definida no cubriera la necesidad del proyecto, se debe realizar una consulta la departamento de Calidad, que se encargará de estudiar el escenario, aportar una solución y retroalimentar la normativa de nombrado existente.
i. Se debe utilizar el método HTTP adecuado para la funcionalidad requerida (cumplir con la semántica del verbo HTTP), cumplir con su naturaleza de idempotencia y devolver la información esperada.

Método de petición	Funcionalidad asociada	Operación Idempotente	Información devuelta
GET	Obtener información de los recursos	SI	Listado del recurso o colección de recursos solicitados.
POST	Creación de Recursos	NO	El nuevo recurso creado (incluyendo identificador del recurso creado)
PUT	Actualización Completa del recurso	SI	El recurso actualizado (completo)
PATCH	Actualización Parcial del recursos	NO	El recurso actualizado (completo)
DELETE	Borrado del recurso	SI	Documento vacío
HEAD	Conocer si un recurso está disponible	SI	Metadatos del recurso (la misma respuesta que GET pero sin el cuerpo)
OPTIONS	Obtener la lista de métodos HTTP que soporta un recurso	SI	Lista de métodos HTTP disponibles para el recurso.

Tabla 1 Clasificación de las operaciones por idempotencia

j. La respuesta del servicio debe incluir un códigos HTTP acorde con el resultado de la ejecución aportando la semántica adecuada:

Código	Semántica
200	Respuestas correctas cuando los métodos de petición sean GET, PUT, PATCH, HEAD u OPTIONS

201	*Created*. Respuesta correcta cuando el método de petición sea POST
204	*No Content*. Ejecución correcta pero no devuelve contenido (DELETE).
400	Petición incorrecta
401	Credenciales incorrectas o ausencia de las mismas
403	Credenciales correctas pero usuario no autorizado a obtener el recurso o actuar sobre él con la petición especificada
404	Recurso no encontrado
405	Método no permitido
415	Cuando se envíe información en un formato no soportado
422	Errores de validación como respuesta a peticiones POST, PUT y PATCH
429	Cuando se supere el límite de peticiones admitidas por usuario
500	Error de servidor

Tabla 2 Códigos HTTP de respuesta

3.2.2.1.4 REST. Nomenclatura de Recursos.

a. **https://***{nombre_host}[:puerto]***/api-***{acronimo}***-***[grupo_funciona]***/V***{versionAPI}***/***{ruta del recurso}***[?***{parametros}***]***
   i. Raíz de contexto compuesta por prefijo “api-“, el acrónimo de la aplicación y opcionalmente se puede establecer un descriptor que haga referencia a la funcionalidad aportada por la API.
   ii. A continuación se debe de indicar la versión de la API, compuesta por el prefijo “V” seguido de un número de 2 posiciones
   iii. La ruta y nombre del recurso. Sólo se permite el uso de nombres (no verbos), en minúsculas y en plural. Si es necesario usar más de una palabra, por legibilidad se separará cada palabra con un guión (“-“)
   iv. Opcionalmente se puede añadir una lista de parámetros para refinar la respuesta del servicio.
   v. Ejemplo de URI:
      https://*servicios.mir.es***/api-***regelec-registro***/V***02***/***peticiones/registros***/***7344
   vi. Excepcionalmente, es posible que no podamos exponer la API como recurso. En este caso excepcional, podemos usar un verbo a continuación del recurso para indicar la funcionalidad que queremos realizar sobre el mismo:
      …/peticiones/registros/7344/**desmarcar**
b. El identificador del recurso que se solicita debe figurar en la URI:
   …/peticiones/registros/**7344**
c. Los filtros de búsqueda deben indicarse en la URI y no en el cuerpo.
   …/peticiones/registros?**fechainicio=20170901&fechafin=20170931**
d. Para establecer filtros, restricción de obtención de campos, ordenación, paginación u otras necesidades en las búsquedas de recursos, se utilizarán parámetros en la URI y

no en el cuerpo y dichos parámetros deberán estar suficientemente documentados desde el punto de visto del consumidor de la API, para evitar confusiones.

e. La información de paginación se devolverá en la cabecera de respuesta “Link”, en la cual se especificarán los enlaces a las páginas siguiente, anterior, primera y última. La estructura de los link, debe cumplir con la RFC 5988 (web-linking) https://tools.ietf.org/html/rfc5988#page-6

3.2.2.1.5 REST. Principio de Idempotencia.

El principio de Idempotencia dice que la ejecución repetida de una petición con los mismos parámetros sobre un mismo recurso tendrá el mismo efecto en el estado del sistema, tanto si se ejecuta 1 o N veces.

Se debe tener en cuenta que el protocolo http no da la garantía que en caso de recibir un timeout en nuestra petición, ésta haya llegado y se haya ejecutado correctamente. Si la ejecución se realiza de manera idempotente, podemos realizar la misma petición tantas veces como consideremos necesario, hasta recibir una respuesta satisfactoria, teniendo la certeza que el estado del sistema será el que se deseaba cuando se realizó la primera petición.

Por ello, se debe implementar el servicio respetando la naturaleza de idempotencia de los métodos HTTP, pues forma parte de la semántica del método y el desarrollador que consuma nuestro servicio tomará decisiones de diseño de tolerancia a fallos y política de reintentos de su sistema, basándose en la semántica del método HTTP que está utilizando.

3.2.3. Capa de Negocio (DOMINIO)

Para el desarrollo de la capa de Negocio, se ha optado por una solución basada en exponer a través de interfaces (puerto de servicio) las funcionalidades que ofrece sin dar detalles de su implementación. Deberán usarse los mecanismos de inyección de dependencias para resolver las dependencias. El diseño de los servicios de negocio se basará en principios de diseño de arquitectura orientada a servicios, con especial énfasis en aquellos casos en que haya de ofrecerse un alto rendimiento y/o abordar transacciones de larga duración, en cuyo caso deberá diseñarse basándose en patrones de procesamiento asíncrono, de cara a asegurar la escalabilidad manteniendo los tiempos de respuesta definidos en los RNF del proyecto.

En la implementación de esta capa, debe plantearse el uso de motores de reglas de negocio que permitan modificar su comportamiento de forma ágil sin necesidad de modificar el código de la implementación del servicio de negocio.

Desde esta capa sólo se podrá invocar a la capa de Persistencia (DAO) y Adaptadores de Salida a través de sus puertos de servicio.

3.2.3.1 Reglas generales de la capa

1. Se debe implementar con objetos planos (POJO).
2. Se debe garantizar la calidad del dato antes de disparar la ejecución de la regla de negocio.
3. El modelo transaccional del acceso a datos se debe gestionar en esta capa y no delegarla a la capa de persistencia.
4. Se debe modelar una jerarquía de excepciones de Negocio para trasladar correctamente los errores a la capa de Presentación y/o de adaptadores de integración.

5. Debe trabajar únicamente con entidades de dominio interno: no debe tener dependencias con entidades de índole tecnológico (por ejemplo entidades JPA).
6. NO se debe de introducir lógica de negocio en los “Aspectos”.
7. Parametrizar lo máximo posible la implementación de la capa de Negocio, mediante motores de reglas de negocio y sacando del código fuente, los valores de referencia de Negocio (valores hardcoded) trasladándolos a entidades externas, accesibles desde esta capa, que puedan ser modificados fácilmente sin provocar cambios en el código fuente.

3.2.4. Capa de Persistencia (DAO)

La abstracción que representa la capa de servicios de acceso a datos y su bajo acoplamiento con la capa de negocio permite la evolución física de los datos sin afectar a ninguna aplicación, ya que lo cambios se centralizan en la propia capa de servicios de acceso a datos, independiente de todas las aplicaciones.

La implementación de esta capa se especializará en el sistema de persistencia utilizado, descargando a la capa de Negocio de la responsabilidad de tener este conocimiento, que invocará a los servicios de acceso a datos a través de su interfaz sin necesidad de saber si el dato se encuentra en una base de datos relacional, NoSQL,….

3.2.4.1 Reglas generales de la capa

1. Se debe utilizar un mapeador Objeto-Relacional (ORM) para implementar la capa de persistencia.
2. Se debe emplear el patrón DAO para obtener más de flexibilidad y capacidad de adaptación.
3. Debe delegar la gestión de la transacción a la capa de Negocio.
4. Aislamiento tecnológico: debe transformar las entidades JPA en entidades de dominio interno y aislar a la capa de Negocio de la tecnología de persistencia utilizada.
5. No incluir lógica de negocio.
6. Se recomienda el uso de caché para almacenar consultas recurrentes y con ello optimizar el uso de la infraestructura y mejora del rendimiento general de la aplicación.
7. Bloqueo de Base de Datos: por defecto se utilizará el bloqueo optimista. Si las necesidades del proyecto, requirieran el bloqueo pesimista, esta decisión debería ser consensuada con el departamento de Calidad y Sistemas.

3.2.5. Capa de Salida (Integración Externa)

La responsabilidad de esta capa es la construcción de adaptadores para las invocaciones de servicios externos a la unidad de despliegue, de forma que el negocio quede desacoplado de los artefactos tecnológicos concretos que sean necesarios utilizar para realizar la invocación a dichos servicios. Esto es extensible a las utilidades que la aplicación necesite utilizar (como por ejemplo mecanismos de caché) de forma que la aplicación no quede acoplada a los servicios de infraestructura o a implementaciones concretas.

Desde esta capa se invocará cualquier servicio externo exceptuando aquellos que deben implementarse bajo servicios de acceso a datos. Se deberá construir un adaptador de servicio externo para la invocación de cualquier servicio ofrecido por un servicio común a través de su cliente, la invocación a servicios externos (AGE), el acceso a Ldap,....

El aislamiento que ofrecen estos adaptadores a la capa negocio facilitará las pruebas, al poder sustituir la implementación de una llamada al servicio real por un *mock* y mejorará la gestión de las dependencias de otros servicios que necesite y se estén desarrollando en paralelo.

3.2.5.1 Reglas generales de la capa

1. Aislamiento tecnológico: debe transformar las entidades de dominio externo aportadas por los clientes de servicios, en entidades de dominio interno y aislar a la capa de Negocio de la tecnología utilizada.
2. No incluir lógica de negocio.

3.2.6. Capa Transversal.

Contendrá las utilidades de propósito general, los java beans que utilicemos para encapsular la información que viaja entre las capas de la arquitectura y los componentes de infraestructura (Logging, Auditoría, Seguridad,…).

3.2.6.1.1 Reglas generales de la capa

1. Se recomienda AOP para implementar el uso de componentes de infraestructura y así disminuir la intrusión de código y mejorar la legibilidad del mismo.
2. No incluir lógica de negocio.

3.3. COMUNICACIÓN ENTRE CAPAS

La comunicación entre capas se debe establecer mediante objetos del modelo (value object) que almacenarán información y viajarán a través de todas las capas de la aplicación.

El sentido de comunicación entre capas es unidireccional. No se permite la invocación desde las capas internas a las capas externas de la arquitectura.

No se permite la comunicación directa desde la capa de presentación y adaptadores de entrada a las capas DAO y adaptador de salida sin pasar por la capa de Negocio.

Desde la capa transversal no se puede realizar ninguna invocación directa al resto de capas.

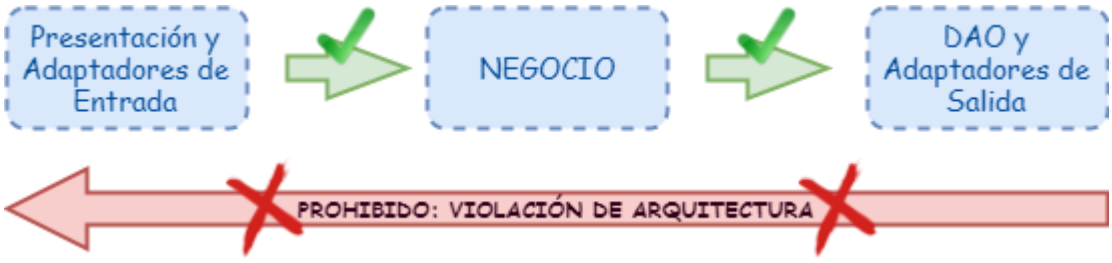

Ilustración 2 Comunicación entre capas

4. NOMENCLATURA DE OBJETOS

4.1. MÓDULOS

En el caso de optar por una configuración de proyecto multimódulo, se debe seguir la siguiente nomenclatura de módulo basándose en la capa lógica que pertenezca.

Patrón de nomenclatura de módulos:

id_capa-nombre_modulo-[tipo]

Desglose:

- **id_capa**: Nombre de la capa lógica a la que pertenece el módulo. Posibles valores:
  - **"ear" :** módulo responsable de la generación del fichero ear
  - Capa de Presentación
    - "web" : elementos de la capa de Presentación (módulo responsable de la generación del war)
    - "pres" : clases java de la capa de presentación. Opcional. El módulo web puede incluir también las clases java de la capa de presentación.
  - Capa de adaptadores de entrada (depende del tipo de adaptador)
    - "rest" : clases java que ofrecen la interfaz REST del servicio de negocio.
    - "soap" : clases java que ofrecen la interfaz SOAP del servicio de negocio.
  - "neg" : capa de Negocio
  - "dao" : capa de Persistencia
  - "adaptador-salida": capa de adaptadores de salida
  - "util": capa transversal de utilidades
- **nombre_modulo**: Nombre significativo de la agrupación funcional del módulo.
- **tipo**: Opcional, sólo se utilizará si se decide separa las clases de interfaz de las de implementación. Posibles valores:
  - "api" : agrupación de las interfaces que ofrece la capa correspondiente.
  - "impl" : agrupación de las clases de implementación.

4.2. PAQUETES

Patrón de nomenclatura de paquetes [Mínimo]:

En el caso de no seguir las directrices de Arquitectura MIR y separación por capas, la nomenclatura mínima que se debe cumplir es la siguiente:

es.mir.aplicacion.[subsistema].{nombre_paquete}*

Patrón de nomenclatura de paquetes [Arquitectura MIR]:

es.mir.aplicacion.[subsistema].id_capa.[subCapa].{nombre_paquete}*.[tipo]

Desglose:

- **aplicación**: Acrónimo de aplicación.
- **subsistema**: Nombre del subsistema. Opcional: sólo si se ha decidido dividir la aplicación por subsistemas
- **id_capa**: Nombre de la capa lógica a la que pertenece el paquete. Posibles valores:
  - "pres": capa de Presentación.
  - "servicios" : capa de adaptadores de entrada
  - "neg" : capa de negocio
  - "dao" : capa de Persistencia
  - "adaptador.salida": capa de adaptadores de salida
  - "util" : capa transversal de utilidades
- **subCapa:** Opcional. Agrupador especial para conceptos propios dentro del contexto de la capa principal.

- Capa de presentación: Agrupación interna de paquetes en esta capa, será libre y se adaptará al framework y/o patrón empleado.
- Capa de adaptadores de entrada: en este agrupador se identifica el tipo de adaptador tecnológico:
  - "rest" : adaptador REST
  - "soap" : adaptador SOAP
- Capa de negocio.
  - "vo" : entidades de dominio interno
- Capa de Persistencia
  - "entidades" : entidades JPA de la capa de persistencia (sólo utilizar en capa DAO)
- Todas las capas (Excepto capa transversal)
  - "util" : clases de utilidades exclusivas de la capa correspondiente.

- **nombre_paquete**: Nombre significativo de la agrupación funcional de clases.
- **tipo**: Tipo de clases que agrupa el paquete. Posibles valores
  - "": las interfaces que ofrece cada capa irá en su paquete base.
  - "impl": clases que implementan las interfaces que ofrece la capa correspondiente.

4.3. RESUMEN DE NOMENCLATURA DE MÓDULOS Y PAQUETES

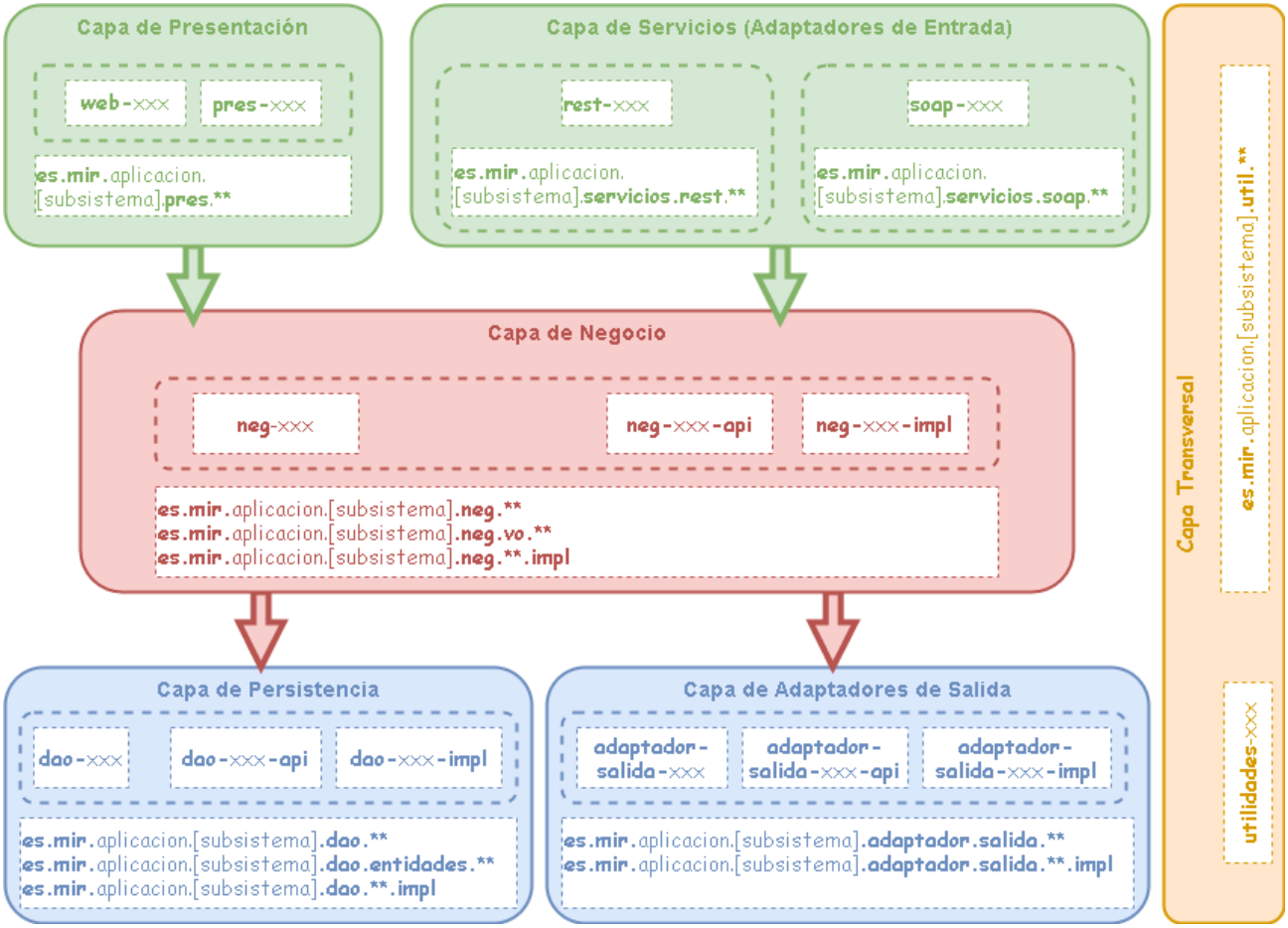

Ilustración 3 Esquema de nomenclatura de módulos y paquetes

- **Presentación [web-xxx, pres-xxx]:**
  - es.mir.aplicacion.[subsistema].pres.**: clases de la capa de presentación.
  - es.mir.aplicacion.[subsistema].pres.util.** : clases de utilidades de uso exclusivo en la capa de presentación
- **Adaptadores de entrada [rest-xxx, soap-xxx]:**
  - es.mir.aplicacion.[subsistema].servicios.rest.**: clases que ofrecen la interfaz rest del servicio de negocio
  - es.mir.aplicacion.[subsistema].servicios.soap.**: clases que ofrecen la interfaz soap del servicio de negocio
  - es.mir.aplicacion.[subsistema].servicios.util.** : clases de utilidades de uso exclusivo en la capa de adaptadores de entrada.
- **Negocio [neg-xxx {neg-xxx-api, neg-xxx-impl}]:**
  - es.mir.aplicacion.[subsistema].neg.** : Interfaces de Negocio (Puertos de Servicio)
  - es.mir.aplicacion.[subsistema].neg.**.impl : clases de implementación de negocio.
  - es.mir.aplicacion.[subsistema].neg.vo.** : clases de dominio interno (value object) encargadas del transporte de datos entre capas
  - es.mir.aplicacion.[subsistema].neg.util.** : Clases de utilidades de uso exclusivo en la capa de negocio
- **Persistencia [dao-xxx {dao-xxx-api, dao-xxx-impl}]:**
  - es.mir.aplicacion.[subsistema].dao.** : Interfaces de Acceso a Datos
  - es.mir.aplicacion.[subsistema].dao.**.impl : clases de implementación de acceso a datos
  - es.mir.aplicacion.[subsistema].dao.entidades.** : entidades JPA
  - es.mir.aplicacion.[subsistema].dao.util.** : Clases de utilidades de uso exclusivo en la capa de persistencia.
- **Adaptadores de salida [adaptador-salida-xxx {adaptador-salida-xxx-api, adaptador-salida-xxx-impl}]:**
  - es.mir.aplicacion.[subsistema].adaptador.salida.** : Interfaces de acceso a servicios externos.
  - es.mir.aplicacion.[subsistema].adaptador.salida.**.impl : clases de implementación de acceso a servicio externo (cliente de servicio web)
  - es.mir.aplicacion.[subsistema].adaptador.salida.util.** : Clases de utilidades de uso exclusivo en la capa de adaptadores de salida.
- **Transversal [utilidades-xxx]:**
  - es.mir.aplicacion.[subsistema].util.**: Clase de utilidad compartidas por más de una capa.

5. PARAMETRIZACIÓN Y CONFIGURACIÓN DE LAS APLICACIONES

Los ficheros de propiedades (independientes o dependientes de entorno) así como los certificados digitales que necesite la aplicación, no deben empaquetarse en la misma unidad de despliegue (jar/war/ear) donde se empaqueta el resto de código.

Dichos elementos de parametrización y configuración deben ser nombrados y ubicados atendiendo a la normativa vigente:

- Parametrización de las aplicaciones:

  PublicacionIntranet/Doc_Especifica_Subsecretaria/MIR-INT-DSI-PARAMETRIZACION APLICACIONES.docx

- Normativa de uso de certificados digitales:

PublicacionIntranet/Doc_Especifica_Subsecretaria/MIR-ANEXO-NORMA-USO-CERTIFICADO-DIGITAL-APLIC-JAVA.docx