Uaithne Java

Diseñado por: **Juan Luis Paz Rojas**

Prólogo

La arquitectura de software puede llegar a ser muy apasionante, pero también puede llegar a ser muy frustrante. En mi trabajo como arquitecto de software he visto pasar antes mis ojos muchos proyectos, algunos que habían iniciado otras personas y otros en los que yo decidía cómo habrían de hacerse, algunos con arquitecturas más acertadas, otros con arquitecturas que no era apropiada para los requisitos de desarrollo y algunos donde ésta no existía.

Me quiero enfocar aquí, sobre todo en la arquitectura del backend de los sistemas de información, cuya arquitectura suele ser engañosamente simple, y se suelen construir con una arquitectura n capas. Estos sistemas suelen tener una interfaz Web (típicamente Single Page Application) o para dispositivos móviles. En estos sistemas el desacople de la interfaz del backend suele ser grande (o por lo menos esto es deseado), y al final la interfaz acaba invocando al servidor para que realice alguna operación.

Un tema importante en la arquitectura de software, aunque muchas veces olvidado, es cómo conseguir una arquitectura que impulse la productividad del equipo de desarrollo. La idea es conseguir con menor esfuerzo humano mayores y mejores resultados tanto en la fase de desarrollo inicial como en la siguientes fases de modificación y ampliación del sistema. Si se observa lo que ocurre en otros sectores, la mejora de la productividad se consigue a través de la automatización de las tareas repetitivas que no requieren de un esfuerzo cognitivo significativo por parte de quién las realiza;. El desarrollo de una aplicación es un proceso de producción de un producto, muchas veces a medida, en la cual la productividad añadida a los usuarios finales suele ser valorada pero la productividad durante el desarrollo de esta suele ser olvidada, y en donde, con la elección de la arquitectura y de las herramientas apropiadas se puede conseguir mejorar mucho la productividad del equipo de desarrollo.

Con Uaithne planteo una arquitectura alternativa para el backend de las aplicaciones que permita conseguir mejorar significativamente la productividad del equipo al desarrollar esta parte de la aplicación sin sacrificar las virtudes de una arquitectura n capas.

La filosofía central de Uaithne es conseguir que el backend se asemeje a un juego de LEGO®, en donde se disponen de piezas altamente intercambiables que se componen hasta conseguir el resultado deseado, cuya función final es conseguir la ejecución de una operación (comando).

El desarrollo de la arquitectura Uaithne ha sido un reto para mejorar la productividad del equipo de desarrollo que inicié a mediados de agosto de 2011, y cuyo primer primer prototipo lo presenté el 28 de agosto de 2011; y que ha ido madurando a lo largo del tiempo, tras la exitosa implantación en muchos proyectos en Delonia Software de diversas índole, y

gracias a la colaboración de Delonia he conseguido llevar a Uaithne a tener un grado de madurez suficiente y respaldarlo con proyectos reales de gran tamaño que validan los fundamentos de la arquitectura y las herramientas propuestas.

Herramientas complementarias

Mejorar la productividad del equipo de desarrollo ha sido una de las grandes motivaciones detrás del diseño de de la arquitectura de Uaithne, y sí sola representa una gran mejoría respecto a la clásica arquitectura n capas. Tras implementar la nueva arquitectura aún quedaba margen para mejorar la productividad, generando código automáticamente que se encargue de manejar la verbosidad requerida por Java, y sobre todo, que genere el código de acceso a la base de datos con sus respectivas queries SQL que maneje la mayoría de operaciones.

Uaithne viene acompañado, opcionalmente, con un generador de código que permite generar (y regenerar) las operaciones y entidades a partir de una pequeña definición, así como la lógica de acceso a base de datos requeridas por estas, usando para ello MyBatis como framework de acceso a la base de datos. De esta forma, solo se deja al programador aquellas actividades en las que realmente se requiera trabajo humano debido a las decisiones que se deben tomar.

La arquitectura de Uaithne y el generador de código trabajando juntos han demostrado en multitud de proyectos de diversos tamaños ser una herramienta de gran utilidad para la programación del backend de aplicaciones consiguiendo durante ello una alta productividad del equipo de desarrollo.

En este documento se plantea únicamente la arquitectura, el generador de código se presenta por separado y su uso es opcional.

El arpa de Dagda

Dagda (“el buen dios”), líder, figura paterna y de inmenso poder, druida de los Tuatha Dé; símbolo de la vida y la muerte, conocido entre otros nombres por Ruad Rofhessa (“señor de gran conocimiento”). Dagda se convirtió en rey de los Tuatha Dé Danann después de que su predecesor, Nuada, fuera herido en batalla; se le atribuye un reinado de 70 u 80 años (dependiendo de la fuente).

Dagda poseía muchos objetos mágicos, pero tal vez su posesión más valiosa fue Uaithne, una hermosa y poderosa arpa que trajo el miedo y la destrucción de los enemigos de los Tuatha Dé Danann; esta poderosa arpa, también conocida como Dur Dá Blá ("Roble de dos florecimientos") o como Cóir Cethairchuir ("Armonía de cuatro ángulos") , tenía, entre otras, la capacidad de controlar y poner en orden las estaciones, infundir coraje a los corazones de los guerreros y curar sus heridas después de la batalla; la música de Uaithne podría lanzar hechizos a todos los que podían oír las mágicas notas que flotaban sobre las verdes colinas de Irlanda.

Durante la segunda batalla de Mag Tured, el arpa fue robada por los Fomoré, con ella en su poder los Fomoré pensaron que podían usarla para lanzar hechizos contra los Tuatha Dé Danann. Por ello, Dagda y sus compañeros Lug y Ogma fueron al lugar donde se refugiaban los Fomoré para recuperarla. Una vez dentro lograron abrirse paso hasta hasta la sala de banquetes en la que los Fomoré festejaban la captura del arpa, en cuya pared se encontraba colgado el instrumento. Entonces, Dagda llamó al arpa con el siguiente cántico:

¡Ven roble de los dos florecimientos!
¡Ven, armonía formada de cuatro ángulos!
¡Ven verano, ven invierno!
¡Voz de arpas, gaitas y flautas!

Al oírlo, el arpa se descolgó de la pared y voló mágicamente hasta las manos de su dueño, con tanta rapidez que mató a nueve personas a su paso. Dagda comenzó entonces a tocar a Uaithne e interpretó el Golltraigi ("el acorde del llanto"), lo que causó que todos los Fomoré presentes comenzaran a llorar y gemir; seguidamente, tocó el Genntraigi ("el acorde de la risa"), que originó que todos riesen sin control; y finalmente Dagda interpretó el Súantraigi ("el acorde del sueño"), que los dejó a todos profundamente dormidos. Aprovechando ello, Dagda, Lugh y Ogma salieron ilesos del campamento.

Fuente:
http://es.wikipedia.org/wiki/Uaithne
http://en.wikipedia.org/wiki/The_Dagda
http://www.transceltic.com/irish/dagda-s-harp
http://irelandnow.com/dagda.html
http://www.geocaching.com/geocache/GC2MXCW_el-arpa-de-dagda
http://arpafeerica.blogspot.com.es/2010/03/leyenda-celta-del-arpa-de-dagda.html
http://www.ucc.ie/celt/online/T300011/
http://www.ucc.ie/celt/online/G300010/
http://viajeroimaginario.wordpress.com/2012/11/30/mitos-celtas/
http://www.cyclopaedia.es/wiki/Uaithne
http://es.forvo.com/word/uaithne/

juanluispaz@gmail.com
https://github.com/juanluispaz/uaithne-java
Versión: 1.0.0 - 27 de marzo de 2018

Índice

Uaithne Java

Diseñado por: **Juan Luis Paz Rojas**

Introducción

Objetivos

- Fácil desarrollo de backend de aplicaciones
- Independiente de su complejidad o tamaño
- Que permita crecer sin que cada vez resulte más complejo de mantener
- Que permita modificar el comportamiento fácilmente
- Que permita mejorar la productividad del equipo que trabaja en el desarrollo de la aplicación

Filosofía

*Conseguir que el backend se asemeje a un **juego de LEGO®**, en donde se disponen de **piezas altamente intercambiables que se componen** hasta conseguir el resultado deseado.*

Principios

- Toda acción que realice la aplicación se realiza con comandos, siendo un comando un objeto que representa la acción a ser ejecutada y que contiene toda la información necesaria para poder ser llevada a cabo. A estos objetos se les denominarán “operaciones”.
- La ejecución de las operaciones se delega a otros objetos a los que se denominarán “ejecutores”.
- La ejecución de las operaciones se debe poder realizar en diferentes capas, siendo el resultado de la operación la suma de las acciones realizadas en las diferentes capas.
- Las capas deben tener un bajo acoplamiento entre ellas, la dependencia entre ellas debe ser mínima, por lo que se comunican a través de una única interfaz, la del ejecutor, que representa una capa.
- Las capas se deben poder ordenar fácilmente, permitiendo añadir una nueva capa en medio de las ya existentes, o reordenarlas, sin que ello involucre hacer grandes cambios.
- Los ejecutores tienen la capacidad de tratar aquellas operaciones que así indique o tratar todas aquellas operaciones que lo atraviesen.
- Para facilitar la programación, las operaciones se agrupan en unidades lógicas que se denominarán “módulos”.
- Debe permitir la separación de competencias, de forma tal que la lógica que no está interrelacionada, pero aun así, necesaria para completar la ejecución de una operación, se pueda separar en diferentes capas. Al estratificar la lógica se consigue

tener piezas más pequeñas que permiten mejorar la mantenibilidad de la aplicación y facilitan su reutilización.
- Debe permitir tener interceptores, un interceptor permite realizar acciones antes o después de la ejecución de una operación, o cuando ocurre un error; un interceptor puede llegar a modificar la información contenida dentro de la operación o su resultado; un interceptor puede incluso evitar que se ejecute una operación. Un interceptor se consigue al tener una capa con la lógica del interceptor que delega la ejecución en otra capa.

Patrones de diseño

Uaithne es el resultado de una ingeniosa combinación de los patrones de diseños:

- Command (Orden)
- Chain of Responsibility (Cadena de responsabilidad)
- Strategy (Estrategia)

Esta combinación de patrones dan como resultado una estructura sobre la cual construir el backend de una aplicación, permitiendo incorporar muchas de las ventajas de la Programación Orientada a Aspectos sin sus complicaciones.

Notas

- Se requiere como mínimo Java 8 ya que en lo aquí propuesto se hace uso de funciones Lambdas.

- En los ejemplos aquí mostrado se hace uso de la API de fechas y horas de Java 8, específicamente de la clase `LocalDateTime` que permite almacenar la fecha y la hora sin zona horaria.

- Lo aquí presentado está diseñado para ser ejecutado sin tener dependencias complejas, por lo que se puede desplegar en un contenedor web sencillo, como puede ser Tomcat, Jetty, Jetty Embebido, o incluso en un contenedor OSGI como puede ser Apache Felix o su derivado Apache Kafka.

- En los ejemplos aquí mostrados se hace uso de la anotación `@Data` provista por el proyecto Lombok (https://projectlombok.org/) con el fin de reducir la cantidad de código verboso requerido por Java, siendo su uso completamente opcional.

  La anotación `@Data` provoca que en la clase sobre la cual se aplica se genere automáticamente para cada uno de los campos sus respectivos getter y setter, así como también genera la implementación de los métodos `equals`, `hashCode` y `toString`.

Aplicación de ejemplo

Con el fin de mostrar cómo utilizar Uaithne se va a utilizar como ejemplo una pequeña aplicación de manejo de calendarios, en esta aplicación existen diversos calendarios que a su vez contienen eventos. La base de datos para esta aplicación está compuesta por las siguientes tablas:

```
CREATE TABLE calendar (
    id          int          NOT NULL PRIMARY KEY,
    title       varchar(30)  NOT NULL,
    description varchar(200)
);

CREATE TABLE event (
    id          int          NOT NULL PRIMARY KEY,
    title       varchar(30)  NOT NULL,
    start       datetime     NOT NULL,
    end         datetime     NOT NULL,
    description varchar(200),
    calendar_id int          NOT NULL REFERENCES calendar(id)
);
```

Entendiendo los componentes del framework

Entidades

En Uaithne la información y la lógica debe estar separada, la data se representa utilizando POJOs (Plain Old Java Objects), estas clases encapsulan la información sin añadir lógica de ningún tipo, por lo que una clase de entidad solo contiene los campos donde se almacena la información; las propiedades que permiten acceder o establecer el valor de estos campos, y constructores de instancia si así se desea. Usualmente suelen tener su propia implementación de los métodos `equals`, `hashCode` y `toString`, adicionalmente suelen implementar la interfaz `Serializable`.

Para la aplicación de calendario se va a crear una entidad por cada tabla de base de datos. Estas entidades son las que luego se utilizarán para transportar la información contenida en sus correspondientes tablas. Las clases que representan las entidades serían similares a:

```
@Data
public class Calendar implements Serializable {
    private int id;
    private String title;
    private String description;

    // La anotación @Data genera automáticamente:
    // getters, setters, equals, hashCode, toString

    public Calendar() { }
    // Otros constructores...
}
```

```
@Data
public class Event implements Serializable {
    private int id;
    private String title;
    private LocalDateTime start;
    private LocalDateTime end;
    private String description;
    private int calendarId;

    // La anotación @Data genera automáticamente:
    // getters, setters, equals, hashCode, toString

    public Event() { }
    // Otros constructores...
}
```

Operaciones

Las operaciones representan una acción a ser ejecutada por el sistema, representada como un objeto, el cual contiene la información necesaria para su ejecución, pero en ningún caso contiene la lógica para ser ejecutada.

Todas operaciones deben extender de la clase `Operation` cuya definición es la siguiente:

```
public class Operation<RESULT> implements Serializable {

}
```

Esta clase recibe como argumento genérico el tipo del resultado de la operación, en el caso de que no retorne ningún valor, el tipo del resultado debe ser la clase `Void` que solo admite nulo.

Las operaciones se modelan igual que las entidades, solo contienen los campos donde se almacena la información, las propiedades que permiten acceder o establecer el valor de estos campos, y constructores de instancia si así se desea, usualmente suelen tener su propia implementación de los métodos `equals`, `hashCode` y `toString`, adicionalmente se suelen implementar la interfaz `Serializable` (aquí ya incluida por defecto en la clase base `Operation`).

Todas las operaciones deben extender la clase `Operation`, así que si por ejemplo, si se desea obtener un listado de eventos para un calendario en particular y un momento en particular se puede crear una clase que representa la acción de obtener ese listado que sería similar a:

```
@Data
public class SelectMomentEvents extends Operation<List<Event>>
{
    private int calendarId;
    private LocalDateTime moment;

    // La anotación @Data genera automáticamente:
    // getters, setters, equals, hashCode, toString

    public SelectMomentEvents() { }
    // Otros constructores...
}
```

En la operación `SelectMomentEvents` se tiene que:

- Para poder ser ejecutada esta operación requiere del id del calendario y el momento que se desea consultar.
- El resultado de la operación es de tipo `List<Event>`.

Contexto

El contexto es una clase que contiene información complementaria a la ejecución de una operación, su contenido no depende de la operación que se trate de ejecutar.

En el contexto se suele colocar información referente a la identidad del usuario que realiza la operación, sus permisos, etc. En aplicaciones web el contexto se suele almacenar en la sesión del usuario ya que su contenido no se puede asumir como correcto si se envía desde el navegador sin comprometer la seguridad del sistema.

La clase `Context` representa la clase del contexto, a la que se le deben añadir las propiedades necesarias, cuya definición inicial es la siguiente:

```
public class Context implements Serializable {
    // Añadir aquí propiedades del contexto
}
```

Se puede tratar la clase de contexto como una entidad más por lo que, al igual que una clase de entidad, solo contiene los campos donde se almacena la información, las propiedades que permiten acceder o establecer el valor de estos campos, constructores si así se desea, usualmente suelen tener su propia implementación de los métodos `equals`, `hashCode` y `toString`, adicionalmente suele implementar la interfaz `Serializable`.

Elementos típicos a incluir en el contexto:

- **Identificador del usuario que se autenticó en la aplicación**: esta propiedad contiene el identificador del usuario que hizo login, es decir, que introdujo el usuario y la contraseña para ser validado en el sistema.
- **Identificador del usuario que ejecuta la aplicación**: esta propiedad contiene el identificador del usuario que está utilizando la aplicación, en muchos casos es el mismo usuario que hizo login, pero su manejo por separado permite manejar situaciones más complejas, como son usuarios que necesitan operar como si fuesen otro usuario (típico en soporte técnico), incluirlo desde un principio en la aplicación es recomendado aunque a priori no se vea su utilización, ya que modificar a posteriori la lógica de la aplicación resulta muy complejo, respecto al escaso coste de manejarlo desde un principio.
- **Roles del usuario que ejecuta la aplicación**: esta propiedad contiene la información necesaria para conocer qué operaciones pueden ser ejecutadas por el usuario que está utilizando la operación. Los roles se pueden definir en un array de enteros de 32 bits como un mapa de bits, donde cada bit dentro del array corresponde a un rol. Al codificar los roles en un array de enteros de 32 bits permite ocupe poco espacio, y se pueda usar también como mapa de bits en JavaScript sin ningún problema (JavaScript solo soporta mapa de bits sobre enteros de 32 bits). Para facilitar la lectura del mapa de bits se suele crear un método para cada rol que retorna un booleano que indica si el usuario posee ese rol en el mapa de bits.

- **Idioma**: de ser necesario, se puede incluir una propiedad que indique cual es el idioma del usuario, permitiendo así que algunas operaciones tengan en cuenta la internacionalización.

- **Datos del usuario**: de ser necesario, y por motivo de rendimiento, se puede incluir algunos pocos datos del usuario que debido a la frecuencia que se usa es conveniente tenerlo en la sesión en lugar de leerlo constantemente en la base de datos. Es muy importante incluir la menor cantidad de información posible, ya que su almacenamiento en la sesión resulta costoso.

- **Tokens de seguridad**: si la aplicación utiliza algún token de seguridad, como puede ser por ejemplo el necesario para evitar ataques XSRF (Cross Site Request Forgery).

- **Conexión a la base de datos**: que debe ser marcada como transient para excluirla así de la serialización (evitando así ser almacenada en la sesión, por ejemplo). Este campo debe ser excluido de los métodos `equals`, `hashCode` y `toString`. El incluir la conexión a la base de datos en el contexto de ejecución permite controlar fácilmente la transaccionalidad del sistema y el ámbito de esta.

Al ejecutar una operación siempre está disponible el contexto de ejecución, y se puede usar para completar la información necesaria para realizar la ejecución de esta.

Es muy conveniente crear operaciones contextualizadas en lugar de otras más genéricas, por lo que es preferible crear una operación “Dame mis pólizas” en lugar de la más genérica “Dame las pólizas del usuario indicado”, la creación de operaciones contextualizadas reduce enormemente los problemas de seguridad relacionados a la identidad del usuario, ya que la identidad de este se lee del contexto que siempre se almacena en un lugar seguro (como en la sesión del usuario en el servidor) en lugar de confiar en el solicitante.

Se recomienda **usar operaciones contextualizadas en lugar de las más genéricas**, aunque eso implique tener dos operaciones que hagan aparentemente lo mismo, con la salvedad de que una lee la información del contexto y otra de la operación.

Adicionalmente, para facilitar la gestión de contexto y subcontextos se suele incluir un constructor que recibe por argumento otro contexto, permitiendo así sacar una copia de este, y un método, que aquí se le llama `updateFrom`, que copia los valores del contexto que recibe por argumento al contexto sobre el cual se invoca.

El contexto de ejecución se debe almacenar en un lugar seguro, típicamente en la sesión del usuario del servidor, bajo ningún concepto la información del contexto debe provenir del navegador, dispositivo móvil, o cualquier otra fuente que no sea de confianza. Para poder detectar si el contexto contiene información que se debe almacenar se incluye el método `hasData` que retorna un booleano que indica si el contexto contiene información que se debe almacenar en la sesión, si retorna false es porque la sesión no es requerida, y de existir, puede ser destruida.

Para facilitar el cierre de sesión se incluye el método `clearUserData`, que borra todos los datos del usuario del contexto, forzando así a destruir la sesión al final de la ejecución de

todo el proceso. El método `isUserLoggedIn` retorna un booleano que indica si hay un usuario activo.

Ejemplo de contexto para la aplicación de manejo de calendario:

```
// Context.java
import java.io.Serializable;
import java.sql.Connection;
import lombok.Data;
import lombok.ToString;

@Data
@ToString(exclude = "databaseConnection")
public class Context implements Serializable {
    private int userId;
    private int realUserId;
    private int[] roles;
    private String userName;
    private String language;
    private String xsrfToken;
    private transient Connection databaseConnection;

    public Context() { }
    public Context(Context context) {
        this.updateFrom(context);
    }

    public void updateFrom(Context context) {
        this.userId = context.userId;
        this.realUserId = context.realUserId;
        this.roles = context.roles;
        this.userName = context.userName;
        this.language = context.language;
        this.xsrfToken = context.xsrfToken;
        this.databaseConnection = context.databaseConnection;
    }

    public boolean hasData() {
        return userId > 0 || realUserId > 0 || roles != null || roles.length > 0
                || userName != null || language != null || xsrfToken != null;
    }

    public boolean isUserLoggedIn() {
        return userId > 0;
    }
```

```
    public void clearUserData() {
        userId = 0;
        realUserId = 0;
        roles = null;
        userName = null;
        language = null;
        xsrfToken = null;
    }

    /**
     * Permite saber si el usuario tiene un rol en particular dentro del array
     * de mapa de bits de roles, donde cada bit es un rol
     *
     * @param index índice dentro del array donde está el rol
     * @param bit posición del bit dentro del bitmap que indica si tiene el rol
     * @return true si el usuario tiene el rol, false en caso contrario
     */
    public boolean hasRole(int index, int bit) {
        return (roles[index] & (1 << bit)) != 0;
    }

    public boolean canViewCalendar() {
        return hasRole(0, 0);
    }
    public boolean canCreateCalendar() {
        return hasRole(0, 1);
    }
    public boolean canUpdateCalendar() {
        return hasRole(0, 2);
    }
    public boolean canDeleteCalendar() {
        return hasRole(0, 3);
    }

    public boolean canViewEvent() {
        return hasRole(0, 4);
    }
    public boolean canCreateEvent() {
        return hasRole(0, 5);
    }
    public boolean canUpdateEvent() {
        return hasRole(0, 6);
    }
    public boolean canDeleteEvent() {
        return hasRole(0, 7);
    }
}
```

Ejecutores

Un ejecutor es el encargado de ejecutar la lógica asociada a una operación, pudiendo haber dos casos:

- Ejecutar una lógica para **todas las operaciones** que atraviesen el ejecutor
- Ejecutar una lógica para las **operaciones de un tipo** concreto

Todos los ejecutores extienden de la clase `Executor`, cuya definición es la siguiente:

```
public class Executor {

    protected final Executor next;
    final HashMap<Class, BiFunction> handlers = new HashMap<>();

    public final Executor getNext() { [...] }

    public final <RESULT, OPERATION extends Operation<RESULT>>
        RESULT execute(OPERATION operation, Context context) { [...] }

    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    { [...] }

    protected final <RESULT, OPERATION extends Operation<RESULT>>
        void handle(Class<OPERATION> operationType,
                    BiFunction<OPERATION, Context, RESULT> implementation)
    { [...] }

    public Executor() { [...] }
    public Executor(Executor next) { [...] }
}
```

La clase `Executor` contiene los siguiente elementos:

- **`HashMap<Class, BiFunction> handlers`**: HashMap de uso interno que contiene como valor la función a ser ejecutada cuando se reciba una operación del tipo indicado en la clave. De no estar registrada ninguna entrada en este diccionario para el tipo de operación que se trate de ejecutar, se delega la ejecución al ejecutor indicado en la propiedad `next`.

- **`Executor next`**: Propiedad de solo lectura, con su respectivo getter, que contiene la referencia al siguiente ejecutor a ser ejecutado si el ejecutor actual no es capaz de manejar la operación recibida. De ser nulo, si la operación no es manejada por el ejecutor actual, se inicia una excepción que indica que no se ha podido manejar la operación.

- **`RESULT execute(OPERATION operation, Context context)`**: Este método se encarga de recibir una operación y provocar su ejecución, adicionalmente recibe el contexto de ejecución.

- **`RESULT executeAnyOperation(OPERATION operation, Context context)`**: Este método se encarga de ejecutar una operación independientemente de su tipo, adicionalmente recibe el contexto de ejecución. Este método puede ser reemplazado por otra implementación que típicamente acaba invocando el método `execute` de la propiedad `next` o a la implementación base para continuar la ejecución de la operación.

- **`void handle(Class<OPERATION> operationType, BiFunction<OPERATION, Context, RESULT> handler`**: Este método se encarga de registrar la función pasada como segundo argumento encargada de ejecutar la lógica asociada a la operación de tipo pasado como primer argumento.

  Nota: El tipo `BiFunction<OPERATION, Context, RESULT>` forma parte de los tipos para manejar los Lambdas en Java 8 y quiere decir que recibe una función cuyo primer argumento es de tipo `OPERATION`, el segundo argumento es de tipo `Context` y el tipo retornado es `RESULT`.

Ejecutores para todos los tipos de operaciones

Para crear un ejecutor que realice una tarea sin importar el tipo de la operación basta con extender la clase `Executor` y reimplementar el método `executeAnyOperation` de esta clase. Ejemplo:

```
public class ExecutorForAllOperations extends Executor {

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        // Lógica a ser ejecutada antes de la ejecución de la operación
        RESULT result = next.execute(operation, context);
        // Lógica a ser ejecutada después de la ejecución de la operación
        return result;
    }

    // Constructores ...
}
```

En este tipo de ejecutores típicamente se suele acabar llamando el método `execute` de la propiedad `next` (de la clase base) o a la implementación base del método sobreescrito para continuar la ejecución de la operación, y se suele usar en tareas como:

- Escritura en el log
- Apertura y cierre de la transacción

- En combinación con el uso de interfaces o clases bases se puede verificar si la operación implementa un tipo y en función de ello realizar una lógica

Ejecutores para un tipo concreto de operación

Para crear un ejecutor que realice una tarea para un tipo de operación en particular basta con extender la clase `Executor` e implementar un método para cada operación que se desea ejecutar, donde el primer argumento es la operación, el segundo es el contexto y retorna el resultado de la operación.

Una vez implementadas la operaciones se debe seleccionar el constructor de la clase base que se desea sobreescribir, en función de si la implementación actual necesita recibir un siguiente ejecutor o no. Dentro del constructor implementado hay que hacer indicar qué método es el encargado de ejecutar una operación, para ello se llama al método `handle` (de la clase base) que recibe como argumentos la clase de la operación y el método encargado de ejecutar la lógica asociada a esta.

Por ejemplo, si se desea crear un ejecutor que implemente la operación `SelectMomenEvents` la clase sería similar a:

```
public class EventsDatabaseExecutor extends Executor {
    private List<Event> selectMomentEvents(SelectMomentEvents operation, Context context)
    { [...] }

    public EventsDatabaseExecutor(Executor next)
    {
        super(next);

        handle(SelectMomentEvents.class, this::selectMomentEvents);
    }
}
```

Y si adicionalmente se quisiera implementar la operación `SelectEventById` la clase sería similar a:

```
public class EventsDatabaseExecutor extends Executor {
    private List<Event> selectMomentEvents(SelectMomentEvents operation, Context context)
    { [...] }

    private Event selectEventById(SelectEventById operation, Context context)
    { [...] }

    public EventsDatabaseExecutor(Executor next)
    {
        super(next);

        handle(SelectMomentEvents.class, this::selectMomentEvents);
        handle(SelectEventById.class, this::selectEventById);
    }
}
```

Ejecutando una operación

Para ejecutar una operación hay que invocar el método `execute` del ejecutor donde se va a ejecutar la operación, este método recibe dos argumentos genéricos con el tipo de la operación y el tipo del resultado de la operación, y dos argumentos con la operación y el contexto. Ejemplo:

```
SelectMomentEvents op = new SelectMomentEvents ( [...] );
// Alguna otra inicialización requerida por la operación
List<Event> result = executor.execute(op, context);
```

Manejo de excepciones

De ocurrir una excepción durante la ejecución de una operación el framework colecta automáticamente la entrada de la ejecución para que esta pueda ser registrada posteriormente por algún log, para ello se crea una nueva excepción que incluye internamente la excepción no controlada, la operación y el contexto, de forma tal que se pueda disponer de toda la información necesaria para reproducir el problema.

Excepciones públicas

Si se desea tener excepciones propias que no sean alteradas por el framework, es necesario que estas excepciones extiendan de `PublicException`, cuya definición es la siguiente:

```
public class PublicException extends RuntimeException {
    public PublicException() {
    }

    public PublicException(String message) {
        super(message);
    }

    public PublicException(String message, Throwable cause) {
        super(message, cause);
    }

    public PublicException(Throwable cause) {
        super(cause);
    }

}
```

Excepciones internas

Cualquier excepción que no extienda de `PublicException` será tratada como una excepción interna, que representa un funcionamiento anormal del sistema, para el cual se va a recolectar la información necesaria para poder reproducir el problema con facilidad.

Información recolectada al ocurrir una excepción:

- **Excepción** ocurrida durante la ejecución de la operación
- **Operación** que provocó la excepción
- **Contexto** usado durante la ejecución de la operación

Para colectar la información de ejecución lo que se hace es iniciar una excepción de tipo `OperationExecutionException`, cuya definición es:

```
public final class OperationExecutionException extends RuntimeException {
    private Operation operation;
    private Context context;
    private String simpleMessage;

    public Operation getOperation() { [...] }
    public Context getContext() { [...] }
    public String getSimpleMessage() { [...] }

    public boolean sameContent(Operation operation, Context context) {
        return this.operation == operation && this.context == context;
    }

    public OperationExecutionException(Operation operation, Context context)
    {[...]}
    public OperationExecutionException(Operation operation, Context context,
        String message) { [...] }
    public OperationExecutionException(Operation operation, Context context,
        String message, Throwable cause) { [...] }
    public OperationExecutionException(Operation operation, Context context,
        Throwable cause) { [...] }

    private static String createMessage(Operation operation, Context context,
                                        String message, Throwable cause) { [...] }

}
```

En la medida en que la excepción interna va atravesando los diferentes ejecutores que conforman la secuencia de ejecución se comprueba si la operación y el contexto contenidos en la excepción son los mismos que los usados para la ejecución (usando el método `sameContent`), de diferir alguno de ellos se crea una nueva excepción interna que contendrá la excepción interna anterior y los objetos de operación y contexto que correspondan a la ejecución en ese punto, de forma tal, que si por ejemplo, se tiene que en la implementación de una operación se lanzó otra operación, y esta última falla, la

excepción resultante al inicio de la ejecución de la primera excepción, contendrá la información de ambas operaciones.

Nota: puede ser de utilidad modificar la implementación del método `createMessage en` esta excepción, para que, durante la generación del mensaje de la excepción, en lugar de escribir el `toString` de la operación y el contexto, escriba el nombre de la clase y el JSON de cada uno de ellos, evitando así la necesidad de tener una implementación (y confiar en que haya sido implementada apropiadamente) del método `toString` en cada entidad, operación y en el contexto.

El método `createMessage` genera el mensaje por defecto a ser incluido como mensaje de la excepción, la propiedad `simpleMessage` contiene el mensaje simple de la excepción, sin ninguna parte creada automáticamente por esta, y de no tener un mensaje propio usa el mensaje de la excepción interna. El mensaje creado incluye el mensaje especificado para la operación (o en su defecto el de la excepción interna), y luego se añade el nombre completo del tipo de la operación, la representación en string de la operación y la del contexto, de forma tal que al momento de ocurrir un error se disponga de toda la información de entrada del sistema que se usó para ejecutar la operación, disponiendo así de toda la información necesaria para reproducir el error.

La representación en string de la operación y el contexto en el mensaje de la excepción se obtiene al crear la instancia de esta, asegurando así que la información allí contenida corresponde al momento en el cual se inició la excepción, si por algún motivo el contenido del contexto o de la operación cambia durante el tratamiento de la excepción, el mensaje de esta contendrá el valor que tenían al momento de ocurrir error, permitiendo así recrear con exactitud la situación que lo provocó.

Bus de ejecución

La base de Uaithne son las operaciones, y su implementación se realiza en diferentes ejecutores que se van componiendo, de forma tal que el resultado de la ejecución de la operación es la suma de las implementaciones realizadas en los diferentes ejecutores para la operación que fueron atravesados durante su ejecución. Toda esta composición de ejecutores conforman el bus de ejecución de la aplicación.

La idea central en la construcción de los diferentes ejecutores es poder realizar la separación de competencias, de forma tal que cada implementación se dedique, en la medida de lo posible, a una única tarea.

Por ejemplo, para registrar un usuario en el sistema, se requiere:

- Cifrar la contraseña
- Insertar el usuario en la base de datos
- Enviar el email de bienvenida
- Insertar el registro de auditoría
- Escribir en el log de la aplicación

Cada ítem de esta lista corresponde a una competencia, que debería ser implementada por separado, cada una de ella en su propio ejecutor, y luego ser compuestas para dar como resultado el fragmento del bus encargado de su ejecución.

Al descomponer la lógica de esta manera, se encuentra con que hay un elemento central que corresponde a la acción principal, en este ejemplo, insertar el usuario en la base de datos, y otra serie de elementos que complementan o condicionan la acción principal, envolviendola dentro de su propia lógica, a estos se le denomina interceptores, y tienen un comportamiento similar a los triggers en base de datos.

Los interceptores en algún punto suelen acabar invocando al método `execute` de otro ejecutor, típicamente el que se encuentra en la propiedad `next` (de la clase base). Ejemplo:

```
public class InterceptEventsExecutor extends Executor {
    public List<Event> selectMomentEvents(SelectMomentEvents operation, Context
context)
    {
        // Lógica a ser ejecutada antes de la ejecución de la operación
        List<Event> result = next.execute(operation, context);
        // Lógica a ser ejecutada después de la ejecución de la operación
        return result;
    }

    public InterceptEventsExecutor(Executor next) {
        super(next);

        handle(SelectMomentEvents.class, this::selectMomentEvents);
    }
}
```

Al armar el bus de ejecución se enlazan las instancias de los diferentes ejecutores hasta construir una línea de ejecución de un grupo de operaciones, típicamente agrupadas por su módulo. Ejemplo:

```
new InterceptEventsExecutor(new EventsDatabaseExecutor())
```

Al construir esa sección del bus de esta manera se obtiene que para poder ejecutar, por ejemplo, la operación `SelectMomentEvents`, se pasa primero por el interceptor `InterceptEventsExecutor` antes de llegar a la implementación de `EventsDatabaseExecutor`, de forma tal que el resultado de la ejecución de la acción es la suma de la lógica contenida en cada uno de los ejecutores que la operación atraviesa durante su ejecución.

Bifurcando el bus de ejecución

En ocasiones puede ser útil bifurcar el flujo de ejecución en función de algún criterio y delegar la ejecución de la operación a algún segmento de bus de ejecución en particular, para ello:

- hay que crear un ejecutor
- hacer que contenga cada uno de los segmentos en los que puede delegar la ejecución, para ello podría tener, por ejemplo, una propiedad de tipo `Executor`, a la cual se deba asignar la referencia a la cabeza del segmento, y así una propiedad por segmento posible
- implementar la ejecución de la operación, donde en función de un criterio se decide en cuál ejecutor delegar la ejecución de la operación

La más común de las bifurcaciones es la que permite unir los diferentes fragmentos de bus de ejecución, surgidos tras la separación de la aplicación en diferentes módulos, en un único bus de ejecución que actúa como puerta de entrada de todas las operaciones que ejecuta el sistema. Para manejar este tipo de situaciones se ofrece la clase `MappedExecutor`.

La clase `MappedExecutor` implementa un ejecutor que permite delegar la ejecución de una operación en otro ejecutor, en función de su tipo, y si no está registrado ningún ejecutor para ese tipo de operación, la ejecución se delega al ejecutor que se haya especificado en la propiedad `next`.

La definición de la clase `MappedExecutor` es la siguiente:

```
public final class MappedExecutor extends Executor {

    public final void handle(Executor executor) { [...] }

    public final <RESULT, OPERATION extends Operation<RESULT>>
        void handle(Class<OPERATION> operationType, Executor executor) { [...] }

    public MappedExecutor() { [...] }
```

```
    public MappedExecutor(Executor next) { [...] }
}
```

La clase `MappedExecutor` contiene, adicionalmente a los elementos ya provistos por la clase `Executor`, los siguientes elementos:

- **`void handle(Class<OPERATION> operationType, Executor executor)`**: Permite registrar el ejecutor pasado como segundo argumento como el que se va a usar al ejecutar la operación de tipo pasado como primer argumento.
- **`void handle(Executor executor)`**: Este método recorre un segmento de bus de ejecución, moviéndose a través de la propiedad `next` de los ejecutores, y va añadiendo para cada una de las operaciones detectas, su correspondiente entrada que indica que la ejecución de la operación detectada se delega en el ejecutor pasado por argumento.

  En el proceso de búsqueda de las operaciones implementadas en el segmento de bus de ejecución, para cada ejecutor que lo compone se obtiene todas las operaciones manejadas por este y registra, para cada una de las operaciones detectadas, el ejecutor pasado por argumento como el encargado de ejecutar la operación.

  Para recorrer el segmento de bus de ejecución se recorre, para cada ejecutor que lo compone, y detecta las operaciones implementadas por este, y una vez que ha terminado con ese ejecutor avanza al siguiente, que se encuentra en la propiedad `next` y repite el proceso, se detiene cuando `next` sea nulo, `this` o `this.next`. En caso de encontrarse con un `MappedExecutor`, la secuencia de avance es la misma, es decir, solamente avanza por el ejecutor que se encuentre en la propiedad `next`.

De todos los métodos ofrecidos por la clase `MappedExecutor`, probablemente el más útil es `handle(Executor executor)`, que es el que permite añadir de manera cómoda todas las operaciones implementadas en un segmento de bus de ejecución, permitiendo así agrupar varios segmentos de bus de ejecución en un único ejecutor.

Ejemplo, al dividir la capa de acceso a la base de datos en diferentes módulos, se generan diferentes segmentos de bus de ejecución, que hay que agrupar en un único ejecutor que represente esta capa, para ello se podría hacer algo como lo siguiente:

```
MappedExecutor databaseExecutor = new MappedExecutor();
databaseExecutor.handle(new CalendarsDatabaseExecutor());
databaseExecutor.handle(new InterceptEventsExecutor(new
EventsDatabaseExecutor()));
```

Al hacer esto se consigue que el ejecutor `databaseExecutor` sea capaz de ejecutar todas las operaciones implementadas en `CalendarsDatabaseExecutor`, InterceptEventsExecutor y en `EventsDatabaseExecutor`.

Capas de la aplicación

En una configuración típica para un sistema de información, el sistema contiene por lo menos las siguientes capas claramente definidas:

- **Acceso a los datos**: En esta capa se colocan todas las implementaciones necesarias para acceder a los datos (por ejemplo desde la base de datos), algunas de ellas pueden tener algunos interceptores para complementar la acción principal.

- **Backend**: En esta capa se colocan todas las implementaciones que representan acciones en el sistema que no están centradas en la base de datos (aunque podrían hacer uso de ella), por ejemplo, manejo de la autenticación del usuario, algunas de estas implementaciones pueden tener algunos interceptores para complementar la acción principal.

- **Frontend**: Opcional. En muchas circunstancias se requiere agrupar varias operaciones en una, evitando así tener que encadenar peticiones al servidor, cosa que destruye el rendimiento de la aplicación, o porque se desea asegurar que todo se ejecuta dentro de la misma transacción. En esta capa se colocan todas las implementaciones de estas operaciones cuya lógica se descompone en otras acciones en el sistema, pero que no representan una acción real en el sistema por sí misma, sino que es es una mera agrupación de operaciones por motivos de conveniencia.

  Es muy importante tener en cuenta, que si se realizan peticiones al servidor cuya respuesta va a ser utilizada para solicitar otra petición al servidor, este tipo de encadenamiento secuencial de acciones suele ser nefasto para el rendimiento de la aplicación, y en la medida de lo posible debe ser evitado. Por lo que en caso de necesitar este tipo de encadenamiento de peticiones, lo mejor es agruparlas en una, para ello se crea una operación de conveniencia cuya implementación no hace más que ejecutar cada una de las operaciones requeridas, y retornar una entidad que suele contener una propiedad con el resultado de cada una de las operaciones ejecutadas.

  Otro caso fácil de olvidar, es el ámbito de la transaccionalidad del sistema, si se invoca una serie de peticiones desde un sistema externo (por ejemplo, el navegador), cada petición posee su propia transacción. Si se desea que varias peticiones compartan una única transacción, en necesario para ello convertirlas en una única petición siguiendo el mismo esquema antes expuesto.

Adicionalmente se suelen añadir algunos interceptores que no distinguen el tipo de operación:

- **Manejo del ámbito de la transacción de base de datos**: este interceptor asigna la conexión a la base de datos para ser ejecutada en las siguientes capas y controla la transacción, de forma tal que todas las acciones que se ejecuten tras él en la base de datos se encuentren en una misma transacción, a la cual se hará `rollback` si el

resultado de la ejecución de la operación acaba en una excepción, de lo contrario se hace un `commit`.

- **Escritura en el log de la aplicación**: este interceptor escribe en el log de la aplicación cualquier error que ocurra durante la ejecución de una operación, para ello detecta si ha ocurrido una excepción durante la ejecución de esta, y de ser así lo escribe en el log de la aplicación. Este interceptor, típicamente, cuando se encuentra en el ambiente de desarrollo, suele escribir adicionalmente la operación y contexto que recibe de entrada y el resultado de la operación.

- **Validación de los datos**: Opcional. Si se usa un framework de validación, este interceptor permite validar que los datos provenientes de sistemas externos cumple con todas las validaciones puestas sobre operaciones y entidades, y de no cumplirlas inicia una excepción que indica que los datos suministrado no cumplen con los requisitos exigidos.

Acceso al bus

La construcción del bus se hace de manera estática para toda la aplicación, en un singleton, de forma tal que se construya una única vez para toda la aplicación durante la inicialización de esta. La clase que permite acceder al bus de ejecución de las operaciones se le suela llamar "`Bus`" y suele contener dos propiedades estáticas que sirven de puntos de entrada:

- **`Executor getServiceBus()`**: Retorna el bus de ejecución a ser utilizado por servicios expuestos por la aplicación para su consumo por aplicaciones externas, por ejemplo, el navegador.

- **`Executor getBus()`**: Retorna el bus de ejecución a ser utilizado de manera interna por la aplicación.

Esta separación permite tener ciertas diferencia en función de cuál punto de entrada se invoque, por ejemplo, se puede hacer que para el caso de servicios externos se tenga que pasar por el interceptor de validación de los datos, pero para el bus interno esto sea omitido.

Ejemplo de Bus

La clase `Bus` contiene la lógica de acceso e inicialización del bus, y es el método `createBus` el encargado de crear el bus de ejecución, que se ejecuta una única vez durante la inicialización de la aplicación. Esta clase podría lucir de la siguiente manera:

```
// Bus.java

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class Bus {
```

```
private static final Logger LOGGER = Logger.getLogger(Bus.class.getName());

private static Executor bus;
private static Executor serviceBus;

public static Executor getBus() {
    if (bus == null) {
        createBus();
    }
    return bus;
}

public static Executor getServiceBus() {
    if (serviceBus == null) {
        createBus();
    }
    return serviceBus;
}


private static void createBus() {
    DataSource dataSource;
    try {
        InitialContext initialContext = new InitialContext();
        dataSource = (DataSource)
                     initialContext.lookup("java:comp/env/jdbc/myDataBase");
    } catch (NamingException ex) {
        LOGGER.log(Level.SEVERE, "Unable to get the database connection", ex);
        return;
    }

    MappedExecutor dataAccessExecutor = new MappedExecutor();
    dataAccessExecutor.handle(new CalendarImpl());
    dataAccessExecutor.handle(new EventImpl());
    dataAccessExecutor.handle(new EncryptPasswordInterceptor(new UserImpl()));
    dataAccessExecutor.handle( [...] );
    [...]

    MappedExecutor backendExecutor = new MappedExecutor(dataAccessExecutor);
    backendExecutor.handle(new AuthenticationImpl(dataAccessExecutor));
    backendExecutor.handle( [...] );
    [...]

    MappedExecutor frontendExecutor = new MappedExecutor(backendExecutor);
    frontendExecutor.handle(new GroupedOperationsImpl(frontendExecutor));
    frontendExecutor.handle( [...] );
    [...]

    Executor sqlSessionInterceptor =
        new SqlSessionInterceptor(dataSource, frontendExecutor);
```

```
        bus =  new LogDebugInterceptor(LOGGER, "MyApplication",
                                       sqlSessionInterceptor);

        Executor validatorInterceptor = new
ValidatorInterceptor(sqlSessionInterceptor);
        serviceBus = new LogDebugInterceptor(LOGGER, "MyApplication-Services",
                                             validatorInterceptor);
    }
}
```

Hay que destacar que las instancias de cada ejecutor se crean una única vez y se reutilizan para ejecutar diferentes operaciones en diferentes hilos de ejecución, por lo que es importante que internamente no almacenen el estado de la aplicación, en consecuencia, cualquier campo o propiedad que exista dentro de los ejecutores debe ser para almacenar la configuración del bus de ejecución, y no para almacenar información relativa a la ejecución de una operación en particular.

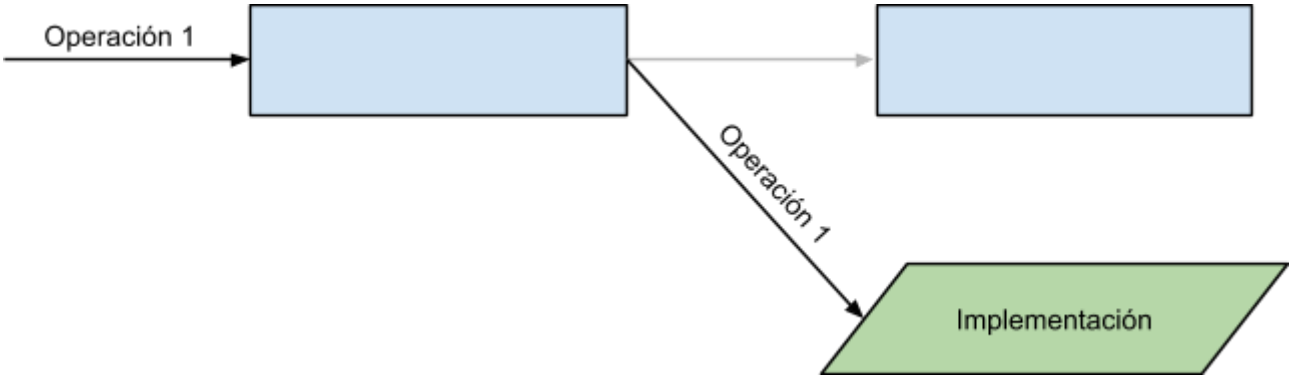
Si por alguna razón, que debe ser evitada en la medida de lo posible, y que debe ser reservada para usos muy avanzados, se necesita crear una variable que contenga información específica a la ejecución de una operación y que deba ser compartida por varios ejecutores se recomienda usar un campo marcado como local al hilo de ejecución (ver la clase `ThreadLocal` de Java), teniendo en cuenta que hay que controlar su asignación y desasignación en todos los casos (puede ser de ayuda utilizar para tal fin un interceptor que controle la salida normal como también la salida en caso de excepción).

Patrones para tejer el bus de ejecución

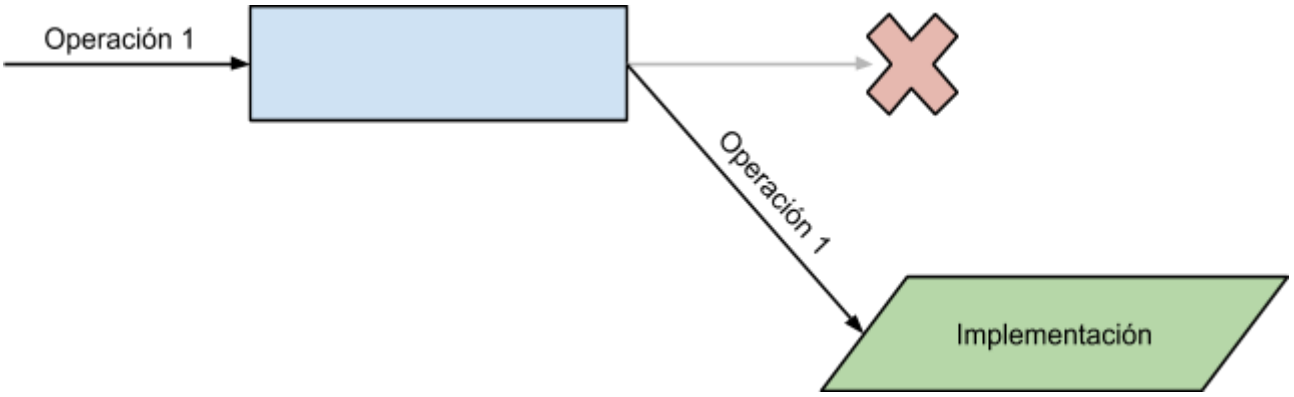
Al realizar la configuración del bus de ejecución surgen una serie de patrones típicos en función del comportamiento que se desea obtener, pudiendo ser combinados para conseguir comportamientos muchos más complejos.

Implementa

Este es el caso más sencillo, en el cual se añade al bus de ejecución la implementación de un módulo sin que este dependa de ningún otro. Este patrón se utiliza para añadir la lógica del módulo, pudiendo ser complementada en las capas más externas.



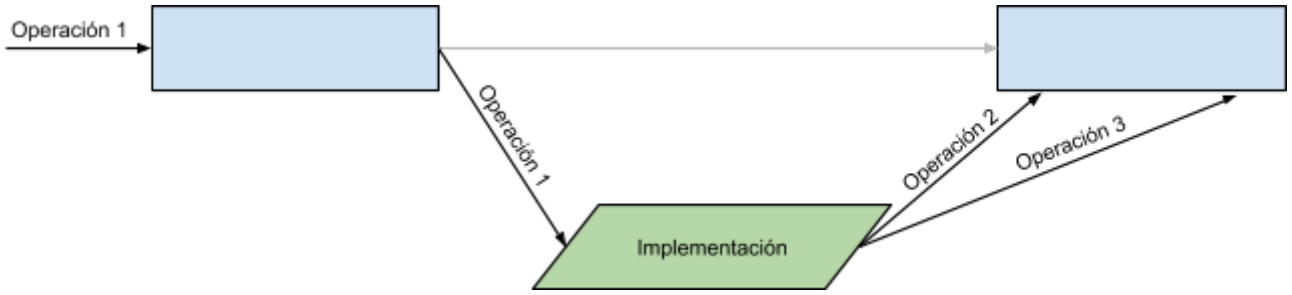
Sección de bus de ejecución que utiliza `MappedExecutor` con next distinto de nulo



Sección de bus de ejecución que utiliza `MappedExecutor` con next nulo

Implementa y utiliza

En este caso la implementación de un módulo inicia la ejecución de operaciones de otros módulos que se encuentran implementadas en un nivel más interno del bus, al final del proceso el resultado de la primera operación ejecutada depende del resultado de las siguientes operaciones sin que el invocante de esta tenga que tener conocimiento de esto. Este patrón se utiliza para añadir la lógica del módulo en el que existe al menos una operación en la que para poder realizar su ejecución se deben realizar otras operaciones para conseguir su resultado; la lógica del módulo puede ser complementada en las capas más externas.



Sección de bus de ejecución que utiliza `MappedExecutor` con next distinto de nulo

Intercepta

Un interceptor es un elemento que se interpone en el flujo de ejecución de la operación permitiendo modificar su comportamiento; por ejemplo, se puede decir que un trigger de base de datos actúa como un interceptor de las operaciones que se realizan sobre una tabla.

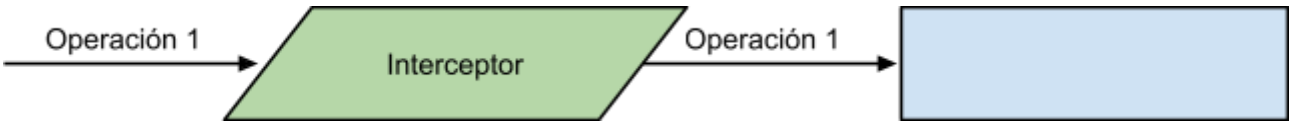
El objetivo de un interceptor es complementar la lógica de ejecución de una operación, pudiendo separar la lógica de la operación en diferentes competencias; por ejemplo: la lógica de escritura en el log de la aplicación en un interceptor, la lógica para comprobar si el usuario tiene permisos de ejecutar la operación solicitada en otro interceptor y finalmente la lógica de acceso a la base de datos en la implementación, de esta forma el resultado de la ejecución de la operación es la suma de las tres partes.

Separar la lógica en diferentes competencias permite mejorar la mantenibilidad del código y su legibilidad, así como facilitar su reutilización. Al separar cada competencia en una pieza distinta se consiguen piezas de código con una alta cohesión y un bajo acoplamiento, facilitando enormemente su modificación, y al encontrarse separadas también es posible reordenarlas, añadir más interceptores entre dos ya existentes, etc. permitiendo así reestructurar por completo el bus de ejecución sin que esto suponga un gran esfuerzo.

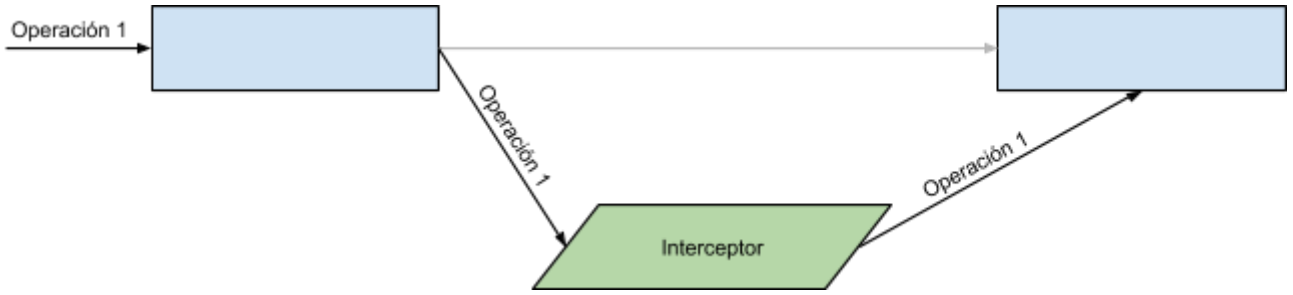
En función del comportamiento que tenga el interceptor se puede clasificar en: observador, modificador, controlador, suplantador y mixto.

Observador

En este caso el interceptor se comporta como un mero observador de la operación que lo cruza sin realizar ningún cambio sobre ella o en su flujo de ejecución, por ejemplo, un interceptor que escribe en el log de la aplicación la operación y su resultado.



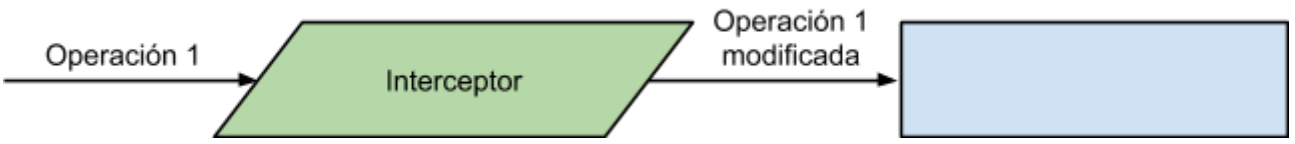
Interceptor interpuesto directamente antes de la implementación



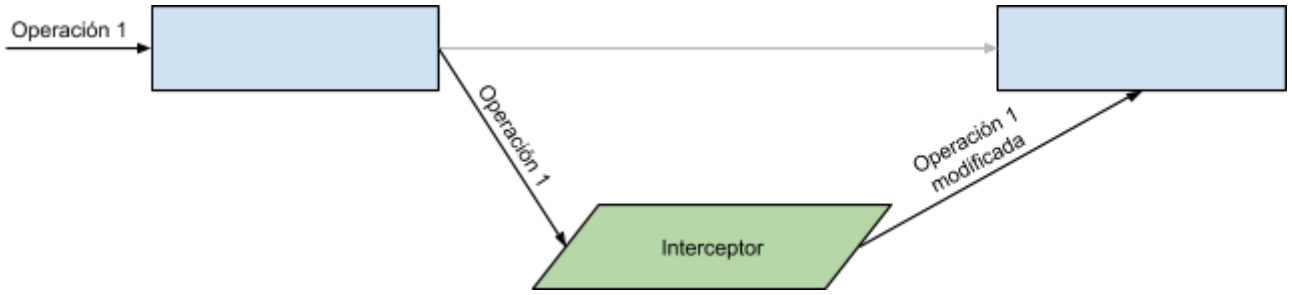
Interceptor interpuesto mediante el uso de una bifurcación del bus de ejecución utilizando `MappedExecutor` con next distinto de nulo

Modificador

En este caso el interceptor modifica la operación o su resultado, este tipo de interceptores se puede utilizar para completar la información que se necesita para ejecutar la operación pero que no es suministrada por quien solicita su ejecución, por ejemplo, añadir la información del usuario que solicita la ejecución de la operación, tomándola de la sesión. También se puede usar este tipo de interceptores para preparar la información de la operación que necesita ser tratada antes de su ejecución, por ejemplo, cifrar la clave del usuario antes de comprobar si puede iniciar sesión.



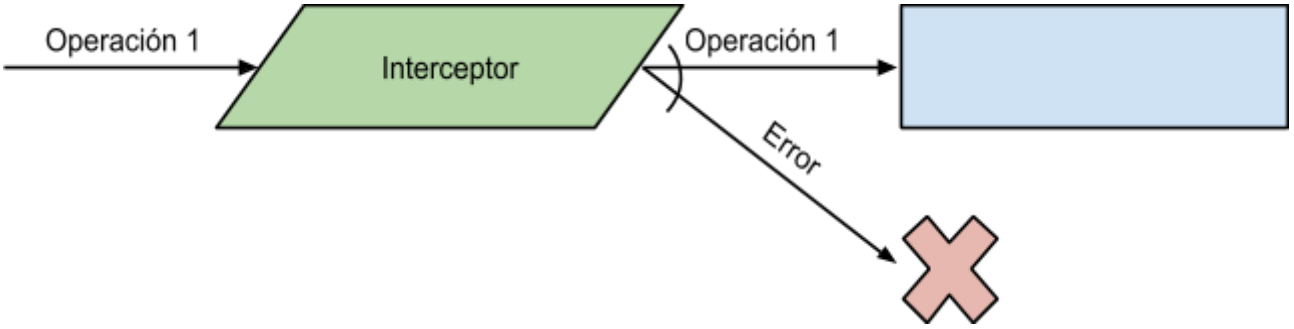
Interceptor interpuesto directamente antes de la implementación



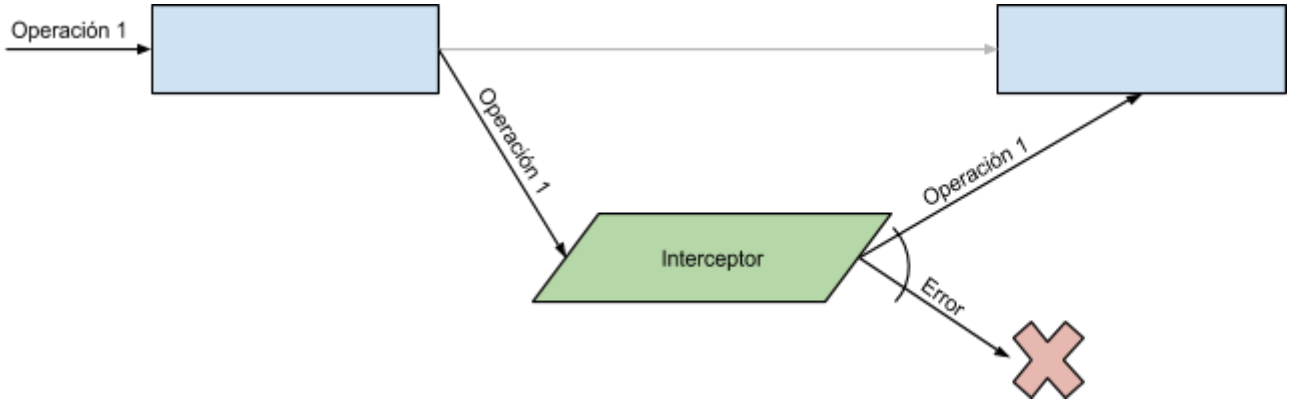
Interceptor interpuesto mediante el uso de una bifurcación del bus de ejecución utilizando `MappedExecutor` con Next distinto de nulo

Controlador

En este caso el interceptor decide si la operación debe continuar su ejecución, o en caso contrario detenerla, pudiendo para ello lanzar una excepción (o retornando un valor definido, según sea el comportamiento deseado). Por ejemplo, se puede utilizar este tipo de interceptores para asegurar que solo aquellos usuarios que tienen permiso puedan ejecutar una operación, y de no tenerlo lanzar una excepción.



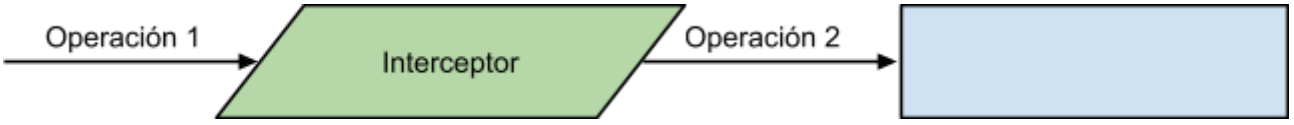
Interceptor interpuesto directamente antes de la implementación



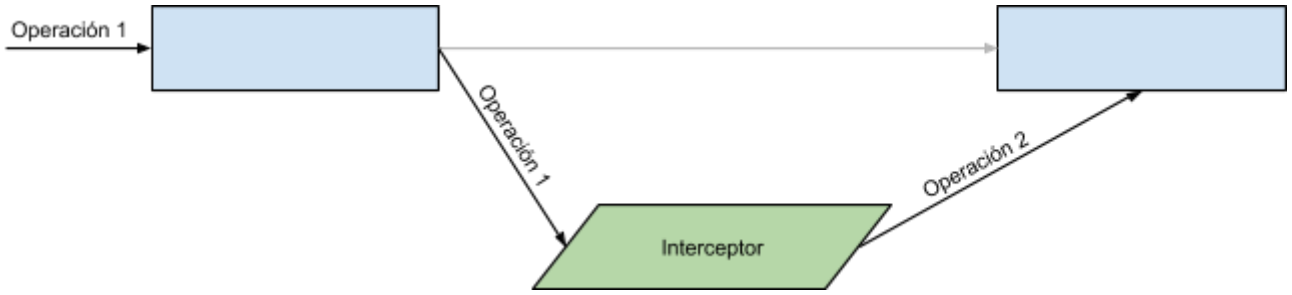
Interceptor interpuesto mediante el uso de una bifurcación del bus de ejecución utilizando `MappedExecutor` con next distinto de nulo

Suplantador

En este caso el interceptor ejecutado transforma la operación en una segunda operación que hace la tarea que ha debido ser ejecutada por la primera operación, dando la impresión de que ha sido la primera operación la que se ejecuta. Este tipo de interceptor es útil si, por ejemplo, se desea mantener la compatibilidad con una operación que ya no forma parte del sistema, por lo que hay que traducirla a una nueva operación que el sistema sea capaz de entender.



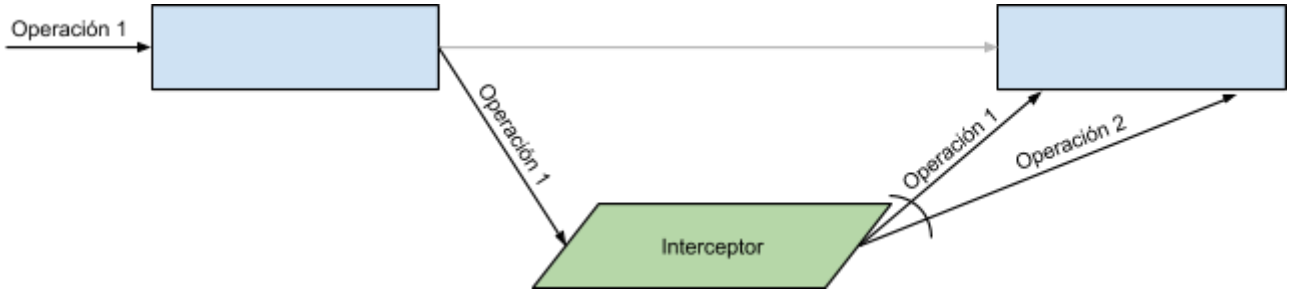
Interceptor interpuesto directamente antes de la implementación



Interceptor interpuesto mediante el uso de una bifurcación del bus de ejecución utilizando `MappedExecutor` con next distinto de nulo

Mixto

Un interceptor puede combinar cualquiera de los comportamientos antes indicados, por lo que es posible, por ejemplo, tener un interceptor que actúe como un controlador-suplantador, que bajo ciertas circunstancias permita ejecutar la operación y en otras la suplante por otra; un interceptor controlador-suplantador podría ser útil, por ejemplo, en circunstancias en las que normalmente los datos se extraen de una fuente (la base de datos de la aplicación, por ejemplo) pero en ciertos supuestos se deba extraer de una fuente alterna (si no se encontró en la base de datos, por ejemplo).



Interceptor interpuesto mediante el uso de una bifurcación del bus de ejecución utilizando `MappedExecutor` con next distinto de nulo

Tipos de operaciones típicas de acceso a la base de datos

Cualquier acción que se pueda realizar sobre un sistema se puede expresar como una operación, pero al interactuar con la base de datos surgen una serie de operaciones comunes, que aunque cambiando sus parámetros se repiten como un patrón:

Operaciones de consulta:

- **`Select One`**: Esta operación retorna la entidad cuyos campos coincidan con los criterios indicados en la operación. Una variante de este tipo de operación es aquella que retorna solamente la entidad que representa el primer registro encontrado.
- **`Select Many`**: Esta operación retorna un listado de entidades cuyos campos coincidan con los criterios indicados en la operación. Una variante de este tipo de operación es aquella en la que solo se trae los registros diferentes, usando la clausula `distinct` en la query SQL.
- **`Select Page`**: Esta operación retorna un listado de entidades a modo de vista paginada cuyos campos coincidan con los criterios indicados en la operación. Una variante de este tipo de operación es aquella en la que solo se trae los registros diferentes, usando la clausula `distinct` en la query SQL.

  Las operaciones de Select Page son las operaciones de consulta más complejas, y a la vez bastante útiles al construir la interfaz, por lo que su tratamiento desde el principio es altamente recomendable.

  Las operaciones de Select Page deben recibir por lo menos los siguientes valores:

  - **`limit`**: indica la cantidad de registros a ser obtenidos en una página de datos, es decir, es el tamaño máximo de la página. Si este campo es nulo indica que no tiene límite, por lo que se debe traer la máxima cantidad de registros posibles.
  - **`offset`**: indica la cantidad de registros a ser ignorados previos a la página de datos que se desea consultar. **Ejemplo**: Si se tiene que el tamaño de la página es de 20 registros, y se desea obtener la tercera página, el valor de este campo debería ser (3 - 1) * 20, que son 40 registros previos a ser ignorados, ya que se va a obtener los registros 41 al 60 (si se considera que los índices empiezan en uno). Si este campo es nulo indica que no se va a ignorar ningún registro.

  El resultado de la operación debe ser un objeto que contenga los siguientes valores:

  - **`limit`**: contiene el `limit` usado en la operación que retornó este objeto y representa el tamaño máximo de la página.
  - **`offset`**: contiene el `offset` usado en la operación que retornó este objeto y representa la cantidad de registros previos que han sido ignorados.

    - **`dataCount`**: contiene la cantidad total de registros, si la operación especificó uno el valor de este campo es el indicado en el operación, sino el valor es el conteo de la cantidad total de registros existente sin paginar.
    - **`data`**: contiene el listado de los registros que pertenecen a la página solicitada.

  Al implementar una operación de Select Page es normal que se tenga que realizar dos consultas a la base de datos, una para traer los registros de la página de datos consultada y la otra para consultar la cantidad de registros existentes.

  Suele ser de utilidad, dar la posibilidad de consultar la cantidad de registros existente sin consultar una página de datos o permitir una manera de no tener que volver a contar la cantidad de registros existentes, para ello se puede añadir las siguientes propiedades a la operación:

    - **`dataCount`**: indica la cantidad total de registros, si se especifica el valor de este campo en la operación provoca que la operación no tenga que consultar la cantidad total de registros, en su lugar se asume que el valor es el aquí indicado, esto es útil para evitar la consulta adicional que cuenta los registros.
    - **`onlyDataCount`**: si se especifica `true` en este campo provoca que la operación únicamente consulte la cantidad total de registro, sin traerse los datos de ninguna de las páginas.

- **`Select Count`**: Esta operación retorna el número de entidades cuyo campos coincidan con los criterios indicados en la operación. Una variante de este tipo de operación es aquella en la que solo se trae los registros diferentes, usando la clausula `distinct` en la query SQL.

- **`Select Entity By Id`**: Esta operación retorna la entidad cuyos id sea el indicado en la operación. Una variante de este tipo de operación es aquella que retorna solamente la entidad que representa el primer registro encontrado.

Operaciones de modificación de entidades:

- **`Insert Entity`**: Esta operación inserta un registro con los valores indicados en la entidad contenida por la operación. Retorna, si así se desea, y de ser posible, el id del registro insertado, esto es recomendado, aunque luego no se use, ya que permite reutilizar la operación en más lugares o hacer seguimiento en los logs.

- **`Update Entity`**: Esta operación actualiza un registro con los valores indicados en la entidad contenida por la operación, el registro a actualizar es el que coincida con el id indicado en la entidad.

- **`Delete Entity By Id`**: Esta operación elimina un registro cuyo id sea el indicado en la operación.

- **`Save Entity`**: Esta operación inserta o actualiza un registro con los valores indicados en la entidad contenida por la operación, el registro a actualizar es el que coincida con el id indicado en la entidad, de no tener se inserta. Retorna, si así se

desea, y de ser posible, el id del registro insertado, esto es recomendado, aunque luego no se use ya que permite reutilizar la operación en más lugares o hacer seguimiento en los logs.

- **`Merge Entity`**: Esta operación actualiza un registro con los valores indicados en la entidad contenida por la operación cuyos valores sean distinto de nulo, el registro a actualizar es el que coincida con el id indicado en la entidad.

Operaciones de modificación de datos no restringidas a entidades:

- **`Insert`**: Esta operación inserta un registro con los valores indicados en la operación. Retorna, si así se desea, y de ser posible, el id del registro insertado, esto es recomendado, aunque luego no se use ya que permite reutilizar la operación en más lugares o hacer seguimiento en los logs.

- **`Update`**: Esta operación actualiza un registro con los valores indicados en la operación. Hay que tener en cuenta que en este tipo de operaciones, los campos de la operación pueden ser usados como parte del `where` de la query, o como valores a ser actualizados en la base de datos, pudiendo en algunos casos, si así se desea, ser omitido si el valor suministrado es nulo, dejando así el valor actual en la base de datos sin modificar.

- **`Delete`**: Esta operación elimina los registro con los valores indicados por la operación, cuyo criterio está definido en la implementación de esta.

Cabe destacar que las operaciones aquí indicadas suelen ser las típicas de base de datos, pero siempre es posible crear más operaciones que escapen de esta tipología, esto suele ocurrir muchas veces en el backend de la aplicación, donde una operación podría ser "Enviar aviso de cobro", **lo importante es que la operación mantenga siempre su significado**, por lo que sería incorrecto usar una operación de insertar factura en la tabla factura de la base de datos (del módulo que representa la tabla factura) como operación de emisión de factura, esta debería ser una operación diferente que debería estar en el módulo de facturación ubicado en la capa de backend.

Llamadas a procedimientos almacenados

Las llamadas a procedimientos almacenados en la base de datos se pueden expresar como una operación, preferiblemente acorde a uno de los tipos de operaciones típicas antes expuestas, teniendo en cuenta que cualquier salida que tenga el procedimiento debe ser retornada como parte del resultado de la operación y nunca modificando el objeto de la operación en sí mismo, esto es de especial atención en los procedimientos con parámetros de entrada y salida, donde la entrada se debe leer de la operación pero la salida se debe escribir en el resultado (nunca en la operación o en el contexto).

En la invocación de procedimientos almacenados, como en cualquier otra acción el resultado de la ejecución siempre se debe leer del objeto resultante de la operación. En el objeto de la operación o en el contexto, bajo ninguna circunstancia se debe almacenar el resultado de la ejecución, o cualquier estado intermedio de su ejecución.

Modelado de los datos

Vista de entidades

En muchas circunstancias, al consultar la base de datos, no se desea extraer un registro entero, sino que en su lugar se desea extraer parte de esa información, o incluso, esa información puede estar combinada con información de otra tabla; en esos casos se puede crear una entidad que solo contenga la información deseada, a este tipo de entidades se les denominará “vista de entidad” y su construcción es idéntica a la de una entidad con la diferencia que no representa de manera íntegra a un registro almacenado en una tabla.

Modelado orientado a objetos de los datos

Los datos de la base de datos se deben modelar en objetos tal como son en su origen, en ningún caso se deben aplicar transformaciones para tratar de construir una adaptación orientado a objetos.

Este punto afecta especialmente al manejo de las relaciones. Las relaciones en la base de datos deben ser representadas de la misma forma que están allí, por ejemplo, si un registro tiene el identificador de otro registro, al crear la entidad que lo representa esto debe mantenerse, es decir, en la entidad se incluirá el identificador del otro registro en lugar de tener una referencia a otro objeto (o en otros casos tener una referencia a una colección de objetos).

El objetivo es que las entidades representen los datos tal como son en la base de datos, sin realizar ningún artificio que altere el modelado relacional existente.

Agrupando resultados

Si se desea realizar una operación que retorne varios resultados, por ejemplo, un registro de la base de datos y sus relaciones, se debe crear una entidad que contendrá cada uno de los registros consultados a la base de datos, de tal forma que estos no sufran alteración respecto a la consulta realizada a la base de datos.

Clave primaria

Se recomienda el uso de claves primarias compuestas por un solo campo, siempre que sea posible (no vale con crear artificialmente una clave primaria de un solo campo), ya que esto facilita enormemente su uso en las capas superiores, especialmente en la construcción de la interfaz.

El uso de claves compuestas suele ser de gran utilidad en la base de datos, y estas pueden ser de dos naturaleza:

- **Todos los miembros de la clave primaria son a su vez claves foráneas**, aquí no queda otra opción que manejar en las operaciones todos los miembros de la clave primaria. Esta situación es típica en las tablas que controlan una relación muchos a

muchos, donde la tabla no representa ningún concepto en sí mismo. Tratar de crear aquí un identificador único no tiene sentido, además, solo serviría para añadir complejidad en el proyecto, sobre todo en la interfaz, por lo que el uso de estas claves primarias artificiales se desaconseja.

- **Algún miembro de la clave primaria es único de la tabla**, este es el caso en el cual se tiene como parte de la clave primaria una o varias claves foráneas, y adicionalmente un campo que es específico de la tabla (no es foráneo), en este caso se recomienda tratar de convertir ese campo en una clave alterna (marcándolo como único en la base de datos) y usarlo fuera de la base de datos como si fuese la clave primaria de la tabla. De no ser posible convertir ese campo en una clave alterna no queda otra opción que manejar en las operaciones todos los miembros de la clave primaria, tal como se expone en el punto anterior.

  Un ejemplo de esta situación **es la tabla de línea de factura**, **la clave primaria** de esta tabla suele ser **el identificador de la factura**, que es la clave foránea de la tabla factura, **y un identificador de la línea de la factura**. Para generar el identificador de la línea de la factura suele haber dos opciones:

  - **El identificador de la línea de la factura es único en la tabla**, pudiendo ser generado mediante un campo autoincremental o una secuencia, esta es la situación deseada ya que este campo se puede convertir en una clave alterna (marcándolo como único en la base de datos) y ser usado como si de la clave primaria se tratase en la aplicación.

  - **El identificador de la línea de la factura es único dentro de la factura**, pudiendo ser este valor, por ejemplo, el número de la línea en la factura, este tipo de valores suele ser complejos de manejar, especialmente en la interfaz, y requiere de una lógica compleja para poder ser generado o mantenido, por ejemplo, si se elimina una línea intermedia, por lo que su uso se desaconseja, lo deseable aquí sería poder transformar esta situación a la expuesta en el punto anterior, donde el identificador de la línea de la factura es único en la tabla, de no ser posible, no queda otra opción que manejar en las operaciones todos los miembros de la clave primaria y añadir la lógica para la creación y mantenimiento de este identificador.

Al crear operaciones se recomienda tener preferencia por las claves compuestas por un único campo sobre las que estén compuestas por más de un campo, sin que esto se traduzca en tratar de evitar el uso de claves compuestas en la base de datos, que suelen ser de gran utilidad, o la creación de identificadores artificiales en algunas tablas. La clave primaria usada en las operaciones no necesariamente tiene que ser la clave primaria de la base de datos, puede perfectamente una clave alterna que permita identificar inequívocamente el registro.

Permisos de ejecución

El manejo de la seguridad de la aplicación se hace mediante el control de las operaciones que el usuario puede ejecutar, sin valorar en ningún caso el contenido de la operación, para conseguir esto, puede ser necesario, en algunos caso, tener la misma operación duplicada, una contextualizada o otra más general, o una que admita una serie de parámetros y otras que adicionalmente admitan algunos más.

La definición de la operación debe ir acorde a los permisos que requiera, que deben ser invariables según el contenido de la operación, si los permisos requeridos varían en función del contenido se debe dividir esta operación hasta conseguir que cada operación tenga sus propios permisos sin depender del contenido, aunque eso signifique tener operaciones similares.

Campos opcionales en las operaciones

El uso de campos opcionales en la operaciones, es decir, campos que admiten nulos, es de gran ayuda, y evitan tener que crear muchas operaciones que hagan lo mismo, superconjuntos de la anterior, donde la siguiente versión permite añadir un campo más respecto a la anterior. Los campos opcionales se usan, por ejemplo, para tener una operación de consulta a la base de datos, que permite tener criterios de filtrados más exigente en la medida que se especifica el valor a esos campos, es decir, si el campo tiene valor se incluye esa condición en la cláusula `where` de la query.

El uso de campos opcionales en las operaciones debe quedar claramente indicado en su definición, mediante un atributo (de tener validaciones automáticas) o un comentario, de forma tal que la persona que vea la definición de la operación pueda saber con facilidad cuales son los campos opcionales sin tener que rebuscar en la implementación.

El uso de campos opcionales debe respetar el significado de la acción a ser realizada por la operación, por lo que es incorrecto tener una misma operación cuya naturaleza cambie en función de la presencia o ausencia de un valor.

Si se usan campos opcionales cuyo impacto va más allá de las condiciones de la cláusula `where` de la query, hay que evaluar su impacto en términos de seguridad, la permisología de la aplicación se basa en las operaciones que se pueden ejecutar por un usuario, no se valora ni cuestiona el contenido de la operación, por lo que si la presencia de un campo opcional provoca que la operación tenga dos niveles de permisología diferentes (en función de la presencia o ausencia del valor) es altamente recomendable dividir la operación en dos, de forma tal que se respete el principio de que los permisos de ejecución de la operación no dependan de su contenido.

Validación de los datos

Se recomienda el uso de un framework de validación, que permita, mediante el uso de anotaciones (preferiblemente) que se añadan a los campos o propiedades de las entidades y operaciones, validar los datos, de forma tal que las reglas allí expresadas contengan todas las restricciones, en lo que a los datos refieren, que existen en la base de datos.

La idea con este tipo de framework es complementar la definición de las clases, mediante el uso de atributos, que añaden restricciones a los campos, por ejemplo, en una propiedad de tipo `int`, se puede especificar el rango de valores válidos, o la longitud máxima de un string.

Por ejemplo, la definición de la entidad `Calendar`, al añadir los atributos de validación podría quedar como:

```
@Data
public class Calendar implements Serializable {
    private int id;

    @NotNull
    @Length(50)
    private String title;

    @Length(200)
    private String description;

    // La anotación @Data genera automáticamente:
    // getters, setters, equals, hashCode, toString

    public Calendar() { }
    // Otros constructores...
}
```

Este tipo de framework suele permitir indicar validaciones de tipo:

- Campo mandatorio u opcional, permitiendo o no valores nulos
- Rango de valores válidos en un número
- Longitud mínima y máxima de un string
- Validar string mediante expresiones regulares, por ejemplo, para validar un email o una URL
- Que una fecha sea del pasado
- Que una fecha sea del futuro
- Entre otras...

Los frameworks de este tipo en Java suelen estar basado en Java Beans Validations, la API estándar de Java para realizar validaciones, y estándar de facto para este fin, soportada por prácticamente todos los frameworks de Java. Hibernate Validations es una implementación de Java Beans Validations (existen otras más, para poder usar la API es necesario hacerlo mediante una implementación). Página web: http://hibernate.org/validator/

El uso de un framework de validación es altamente recomendado, sobre todo si se combina su capacidad con la interfaz, de forma tal que se pueda aprovechar esta información sin tener que programar manualmente la validación.

Al usar un framework de validación es posible escribir un interceptor que valide todas las operaciones que provengan de la interfaz en el servidor, de forma tal que se asegure que todas las operaciones que recibe este, cumplen los criterios antes de realizar cualquier otra acción.

Ordenado

En ocasiones es útil hacer que las operaciones de consulta a la base de datos reciban el `order by` a ser usado en la query, para ello se puede puede usar un campo de tipo `string` que reciba el criterio de ordenado.

El criterio de ordenado no es más que un string que contiene:

- Nombre del campo por el cual se va a ordenar
- Opcionalmente puede tener la dirección de ordenado, a ser: `asc` o `desc`
- Esto se puede repetir tantas veces como se quiera, separándolo con coma

Ejemplo:

- `title`
- `title asc`
- `title asc, description`
- `title asc, description desc`

No se recomienda que el contenido de este campo sea pasado directamente a la base de datos, sino que debe ser previamente transformado y traducido en el código.

Reglas que hay que tener en cuenta en el criterio de ordenado:

- El nombre del campo tiene que coincidir con el nombre de la propiedad en la entidad resultante, de forma tal que el valor aquí indicado no dependa de la construcción de la query. La idea aquí es que el invocante de la operación no tiene que conocer la estructura de la base de datos o como se arma la query.

- Hay una lista conocida con campos válidos para ordenar, si se incluye cualquier cosa que no cumpla con el formato o sea un campo no conocido, se debe provocar un error, impidiendo así la ejecución de la query. La idea aquí es prevenir cualquier riesgo de inyección SQL en este fragmento SQL.

Para traducir la regla de ordenado hay que:

- Normalizar las mayúsculas y minúsculas, pudiendo, por ejemplo, transformando todo a minúsculas.
- Reemplazar cualquier aparición de varios espacios en blanco por uno solo.
- Separar por comas, teniendo en cuenta que, opcionalmente, la coma puede contener espacios en blanco antes y después.
- Eliminar cualquier espacio en blanco que se encuentre al inicio o al final del fragmento.
- Para cada elemento resultante buscar su traducción a su equivalente en la base de datos, teniendo en cuenta que tiene tres variantes:
    - *`nombre_del_campo`* que traduce a *`nombre_de_la_columna`*
    - *`nombre_del_campo`* `asc` que traduce a *`nombre_de_la_columna`* `asc`
    - *`nombre_del_campo`* `desc` que traduce a *`nombre_de_la_columna`* `desc`

    - Si no está contenido en ninguna de estas tres variantes se lanza una excepción, negando así la ejecución de la operación.

- Una vez traducidos todos los términos a su equivalente en base de datos se vuelven a juntar en un único `string` separado por coma.
- El `string` resultante es el que se puede usar en la query.

El que la operación reciba la cláusula de ordenado de esta forma permite:

- hacer cambios en la query sin tener que modificar el código de los invocantes de la operación
- protegerse de inyecciones sql, ya que típicamente esta parte de la query no se pasa como parámetro de la base de datos, sino que se concatena a la query misma
- facilita el uso de la operación, ya que el invocante no tiene que conocer cómo se construye la query, solo conoce el nombre de los campos resultantes de la operación, y con eso puede construir la cláusula `order by`
- mantiene la lógica de base de datos dentro de la capa de acceso a la base de datos, ya que de esta forma no se ve la query dispersada a lo largo de múltiples capas

El que la operación no reciba el `string` a ser utilizado directamente en la query es altamente recomendable, se desaconseja seriamente recibir en este `string` la cláusula `order by` tal como lo necesita la base de datos.

Implementación del acceso a la base de datos

Para implementar el acceso a la base de datos no existe ningún requerimiento especial, por lo que se puede utilizar cualquier framework que se desee. Sin embargo, se recomienda modelar los datos tal como son en su origen, y evitar cualquier intento de transformar los datos de la base de datos en un modelo orientado a objetos.

Los beneficios de modelar los datos de la base de datos en un modelo orientado a objetos suelen ser pocos cuando hay que tratar con servicios que serializan los datos de por medio, por ejemplo, el servicio que envía los datos a la aplicación web, pero las complicaciones añadidas son muchas, tales como manejar la discrepancia entre los dos mundos, controlar las cargas diferidas de los datos y su manejo durante la serialización, operaciones más complejas, redundancia de datos, etc. Por estos motivos y más, se recomienda no tratar de modelar la base de datos con un modelo orientado a objetos al menos que se tenga una buena razón para ello.

Se puede usar cualquier framework de acceso a base de datos que se desee, pero para realizar su apropiada selección es necesario entender qué tipo de vida tiene la base de datos respecto al proyecto:

- **Diseño centrado en la base de datos**: Si se tiene una aplicación cuyo diseño de los datos está centrado en la base de datos, en el cual es factible modificar la base de datos cada vez que la aplicación lo requiera, lo más apropiado es usar un framework que permita acceder a la base de datos tal como es, y permita validar las queries

hechas en la aplicación durante la compilación. Algunos frameworks en esta categoría son:

  - **jOOQ**: Framework de muy larga trayectoria que permite, partiendo de la base de datos, generar todo el código de acceso a esta, de forma tal que se puedan escribir las queries en Java al estilo de Linq de .Net, donde las queries son validadas por el compilador de Java, en lugar de ser un mero string. Este framework es gratuito para bases de datos libres y de pago para bases de datos de pago. Página web: http://www.jooq.org

  - **Querydsl**: Framework similar a jOOQ pero completamente software libre. En varios aspectos más sencillo que jOOQ y un poco menos potente, pero con menor documentación. Puede resultar una muy buena opción si no se desea usar jOOQ. Este framework ofrece la posibilidad de escribir la query para generar el SQL directamente (recomendado) o escribir queries que generen la consulta JPA o JDO (no recomendado ya que añade más complejidad sin mayor beneficio). Página web: http://www.querydsl.com

- **Discrepancias entre la base de datos y el sistema**: Si es necesario tolerar discrepancias entre la base de datos y la aplicación, por ejemplo, cuando el diseño de la base de datos no está bien hecho, la base de datos es difícil de cambiar, etc.; lo mejor es usar un framework que separe ambos mundos, que permita representar las queries en string (hay que tener cuidado con los ataques de inyección SQL). Algunos frameworks que se pueden utilizar en esta situación son:

  - **Apache DbUtils**: Framework muy sencillo y muy potente para acceder a la base de datos, especialmente si se aprovecha las capacidades de la clase `BeanHandler`. No hay que confundirse, su sencillez no limita su potencia. Página web: http://commons.apache.org/proper/commons-dbutils/

  - **MyBatis**: Framework más complejo de acceso a base de datos, con algunas características añadidas, y algunas complejidades también, cuyo rasgo distintivo es que las queries se escriben en un fichero XML separado de la lógica de la aplicación y que posee facilidades añadidas para escribir sin mayores complicaciones queries dinámicas. Mediante el uso de los `Result Maps` ofrece potentes capacidades para adaptar el resultado de las queries en objetos Java, aunque lo más sencillo, y menos código requiere, es no utilizarlos, y en lugar de ello hacer que las queries retornen columnas cuyo nombre corresponda a la propiedad en el objeto Java donde se deben asignar. Página web: http://www.mybatis.org/mybatis-3/

De cara a la mantenibilidad del sistema a largo plazo, lo mejor es utilizar frameworks al estilo jOOQ o Querydsl, que permiten detectar con una simple compilación, cualquier cambio en la base de datos que sea incompatible con una parte de la aplicación.

Es perfectamente posible el uso de frameworks tipo JPA o JDO, como Hibernate, aunque hay que valorar el beneficio de su uso, ya que el usar un lenguaje de query propio añade una complejidad extra, una serie de limitaciones, y una cantidad de conocimientos

adicionales cuyo coste no compensa su beneficio. Si lo que preocupa es el poder usar diferentes bases de datos sin tener que reescribir enteramente las queries sql frameworks como jOOQ o Querydsl permiten cambiar fácilmente de base de datos, y detectar con facilidad aquellas queries que usen construcciones muy específicas, e imposibles de emular en otra base de datos, que requieran ser reescritas.

Implementación de servicios REST o WS

Para implementar servicios web o servicios REST se pueden utilizar los frameworks estándares de la JDK y en su implementación crear una instancia de la operación y mandarla a ejecutar en el Bus.

Los frameworks estándares de la JDK, y que son ampliamente utilizados en otros frameworks son:

- **JAX-WS** para servicios web
- **JAX-RS** para servicios REST

También puede ser de utilidad, especialmente para atender peticiones AJAX provenientes del navegador, un servicio que recibe el JSON de las operaciones y responda el JSON del resultado de esta, evitando así todo el código adicional de tener que exponer otro tipo de servicios (como unos servicios REST) para su uso desde el navegador. El código requerido para implementar este servicio, y toda la política de seguridad a su alrededor está incluido más adelante.

Implementación de interfaz

Estructurar el backend de la aplicación en operaciones es compatible con la mayoría de frameworks de interfaz, por lo que se puede usar el framework que más guste.

Actualmente lo más común a la hora de hacer la interfaz de un sistema es una página web, que siga la filosofía Single Page Application, en donde toda la lógica necesaria para construir la interfaz se ejecuta en el navegador, en este tipo de interfaces el servidor expone uno o varios servicios para ser consumidos por la interfaz mediante AJAX, en este caso puede ser conveniente, si no existe una gran separación entre el backend y la interfaz, tener un servicio que reciba el JSON de la operación a ejecutar y retorne el resultado en formato JSON. El código requerido para implementar este servicio, y toda la política de seguridad a su alrededor está incluido más adelante.

Otros elementos

Transacciones distribuidas

El uso de transacciones distribuidas está seriamente desaconsejado, su uso debe estar completamente justificado y no debe existir alguna otra alternativa de conseguir el mismo resultado.

El uso de transacciones distribuidas provocan serios problemas de rendimiento en la base de datos, y en muchos casos es conveniente tolerar el riesgo de cierto grado de inconsistencia y manejar manualmente las situaciones especiales.

Colas

Para implementar el manejo de colas se puede utilizar JMS, que es la API estándar de la JDK y utilizada por prácticamente todos los frameworks de Java. Una de la implementación más común de esta API, y utilizada en muchos frameworks, es Apache ActiveMQ. Página web: http://activemq.apache.org/

Tareas programadas

En la mayoría de casos, para hacer tareas programadas o demonios, es suficiente con usar la clase `Timer` de Java. Si las capacidades ofrecidas por esa API no son suficientes, se puede utilizar Quartz, este framework es el más utilizado más allá de la clase `Timer` de Java. Página web: http://www.quartz-scheduler.org/

Programación orientada a aspectos

Mediante el uso de interceptores se pueden obtener, de una manera sencilla, todas las capacidades de la programación orientada a aspectos. Todos los patrones para tejer el bus de ejecución corresponden a patrones de la programación orientada a objetos.

Inyección de dependencias

El uso de patrones de inversión de control e inyección de dependencias suele ser muy atractivo y conveniente. Teniendo en cuenta el diseño aquí propuesto, el uso de este tipo de patrones no es requerido en la implementación de los ejecutores u operaciones, ya que los elementos más críticos, típicamente identidad del usuario, permisos y conexión de la base de datos, ya se gestionan mediante el uso del contexto de ejecución, adicionalmente la separación de capas está garantizada en el propio diseño.

En el código aquí incluido, por facilidad, se usa el patrón singleton para obtener la instancia del bus, esto puede ser reemplazado y utilizar en su lugar la inyección de dependencias si así se considera conveniente.

El contexto de ejecución suele contener otros elementos que se suele delegar en la inyección de dependencias, por lo que al construir el objeto de contexto se debe inicializar apropiadamente.

Pruebas unitarias

El modelar la aplicación de la forma aquí propuesta conlleva que el código programado sea muy amigable a las pruebas unitarias, sin requerir añadir mayores complicaciones.

Algunos frameworks útiles para hacer pruebas unitarias son:

- **JUnit**: Framework de pruebas unitarias. Página web: http://junit.org/junit4/
- **Mokito**: Framework para creación de objetos falsos para usar en las pruebas unitarias donde sea realmente necesario. Página web: http://site.mockito.org/

Cómo hacer pruebas unitarias de la conexión a la base de datos ya dependerá del framework de base de datos que se elija.

Código del framework

```java
// OperationExecutionException.java
public final class OperationExecutionException extends RuntimeException {

    private Operation operation;
    private Context context;
    private String simpleMessage;

    public Operation getOperation() { return operation; }
    public Context getContext() { return context; }

    public String getSimpleMessage() {
        if (simpleMessage == null) {
            Throwable cause = getCause();
            if (cause != null) {
                if (cause instanceof OperationExecutionException) {
                    return ((OperationExecutionException) cause).getSimpleMessage();
                } else {
                    return cause.getMessage();
                }
            }
        }
        return simpleMessage;
    }

    public boolean sameContent(Operation operation, Context context) {
        return this.operation == operation && this.context == context;
    }

    public OperationExecutionException(Operation operation, Context context) {
        super(createMessage(operation, context, null, null));
    }

    public OperationExecutionException(Operation operation, Context context, String message) {
        super(createMessage(operation, context, message, null));
        this.operation = operation;
        this.context = context;
        simpleMessage = message;
    }

    public OperationExecutionException(Operation operation, Context context, String message,
        Throwable cause)
    {
        super(createMessage(operation, context, message, cause), cause);
        this.operation = operation;
        this.context = context;
        simpleMessage = message;
    }

    public OperationExecutionException(Operation operation, Context context, Throwable cause) {
        super(createMessage(operation, context, null, cause), cause);
        this.operation = operation;
        this.context = context;
    }
```

```
    private static String createMessage(Operation operation, Context context, String message,
        Throwable cause)
    {
        StringBuilder result = new StringBuilder();

        if (message == null) {
            if (operation == null) {
                result.append("An error happens executing an operation");
            } else {
                result.append("An error happens executing the operation ");
                result.append(operation.getClass().getSimpleName());
            }
            if (cause != null) {
                result.append(": ");
                if (cause instanceof OperationExecutionException) {
                    result.append(((OperationExecutionException) cause).getSimpleMessage());
                } else {
                    result.append(cause.getMessage());
                }
            }
        } else {
            result.append(message);
        }

        result.append("\n\nOperation type: ");
        if (operation != null) {
            result.append(operation.getClass().getTypeName());
        } else {
            result.append("null");
        }

        result.append("\n\nOperation: ").append(operation);
        result.append("\n\nContext: ").append(context);
        return result.toString();
    }
}

// PublicException.java
public class PublicException extends RuntimeException {
    public PublicException() { }
    public PublicException(String message) { super(message); }
    public PublicException(String message, Throwable cause) { super(message, cause); }
    public PublicException(Throwable cause) { super(cause); }
}

// Context.java
public class Context {
    // Add here the context properties
}

// Operation.java
import java.io.Serializable;

public class Operation<RESULT> implements Serializable {
}
```

```java
// Executor.java
import java.util.HashMap;
import java.util.function.BiFunction;

public class Executor {
    protected final Executor next;
    HashMap<Class, BiFunction> handlers;

    public final Executor getNext() { return next; }

    public final <RESULT, OPERATION extends Operation<RESULT>>
        RESULT execute(OPERATION operation, Context context)
    {
        try {
            return executeAnyOperation(operation, context);
        } catch (OperationExecutionException ex) {
            if (ex.sameContent(operation, context)) {
                throw ex;
            }
            throw new OperationExecutionException(operation, context, ex);
        } catch (PublicException ex) {
            throw ex;
        } catch (Exception ex) {
            throw new OperationExecutionException(operation, context, ex);
        }
    }

    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        if (handlers != null) {
            BiFunction<OPERATION, Context, RESULT> handler = handlers.get(operation.getClass());
            if (handler != null) {
                return handler.apply(operation, context);
            }
        }
        if (next != null) {
            return next.execute(operation, context);
        }
        throw new OperationExecutionException(operation, context,
                "No handler found for the operation: "
                + operation.getClass().getTypeName()
        );
    }

    protected final <RESULT, OPERATION extends Operation<RESULT>>
        void handle(Class<OPERATION> operationType,
                    BiFunction<OPERATION, Context, RESULT> handler)
    {
        if (handlers == null) {
            handlers = new HashMap<>();
        }
        handlers.put(operationType, handler);
    }

    public Executor() { this(null); }
    public Executor(Executor next) { this.next = next; }
}
```

```java
// MappedExecutor.java
public final class MappedExecutor extends Executor {

    public final void handle(Executor executor) {
        Executor current = executor;
        while (current != null && current != next && current != this) {
            if (current.handlers != null) {
                for (Class operationType : current.handlers.keySet()) {
                    handle(operationType, executor::execute);
                }
            }

            current = current.next;
        }
    }

    public final <RESULT, OPERATION extends Operation<RESULT>>
        void handle(Class<OPERATION> operationType, Executor executor)
    {
        handle(operationType, executor::execute);
    }

    public MappedExecutor() { }
    public MappedExecutor(Executor next) { super(next); }
}
```

Excepciones utilitarias

Validaciones no satisfechas

Esta excepción sirve para indicar que las validaciones de una operación o entidad no fueron satisfechas.

```
// ConstraintViolationException.java

import java.util.Set;
import javax.validation.ConstraintViolation;

public class ConstraintViolationException extends PublicException {

    private final Set<ConstraintViolation> constraintViolations;

    public Set<ConstraintViolation> getConstraintViolations() {
        return constraintViolations;
    }

    public ConstraintViolationException(Set<ConstraintViolation> constraintViolations) {
        super("" +  constraintViolations);
        this.constraintViolations = constraintViolations;
    }

    public ConstraintViolationException(Set<ConstraintViolation> constraintViolations,
        String message)
    {
        super(message);
        this.constraintViolations = constraintViolations;
    }

    public ConstraintViolationException(Set<ConstraintViolation> constraintViolations,
        String message, Throwable cause)
    {
        super(message, cause);
        this.constraintViolations = constraintViolations;
    }

    public ConstraintViolationException(Set<ConstraintViolation> constraintViolations,
        Throwable cause)
    {
        super("" +  constraintViolations, cause);
        this.constraintViolations = constraintViolations;
    }

}
```

Privilegios insuficientes

Esta excepción sirve para indicar que el usuario está tratando de ejecutar una acción para la que no tiene permisos.

```
// InsufficientPrivilegesException.java

public class InsufficientPrivilegesException extends PublicException {

    public InsufficientPrivilegesException() {
    }

    public InsufficientPrivilegesException(String message) {
        super(message);
    }

    public InsufficientPrivilegesException(String message, Throwable cause) {
        super(message, cause);
    }

    public InsufficientPrivilegesException(Throwable cause) {
        super(cause);
    }

}
```

Interceptores utilitarios

Interceptor de manejo de transacciones SQL

Este interceptor permite controlar el ámbito de la conexión y de la transacción SQL, siendo este el encargado de colocar en el contexto la conexión a la base de datos.

Si la ejecución de la operación provoca una excepción se hace un rollback de la base de datos, si retorna un resultado hace commit.

De existir una conexión de base de datos en el contexto este interceptor no hace nada.

Este interceptor se suele colocar en el bus antes de que se ejecute cualquier lógica en la aplicación, provocando así que todas las operaciones que se ejecuten tras él, estén en la misma transacción.

```
// SqlSessionInterceptor.java

import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;

public class SqlSessionInterceptor extends Executor {

    private final DataSource dataSource;

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        if (context.getDatabaseConnection() != null) {
            /*
             * Ya está usando una conexión a base de datos, no es necesario abrir otra
             */
            return next.execute(operation, context);
        }

        boolean rollback = true;
        Connection connection = null;
        /*
         * Se crea un nuevo contexto, ya que los cambios que se hagan sobre este
         * solo se pueden publicar sobre el contexto recibido por argumento
         * cuando se haga commit, en caso de rollback los cambios deben ser
         * descartados
         */
        Context internalContext = new Context(context);
        try {
            connection = dataSource.getConnection();
            connection.setAutoCommit(false);
            internalContext.setDatabaseConnection(connection);
```

```
            RESULT result = next.execute(operation, internalContext);
            rollback = false;
            return result;
        } catch (SQLException ex) {
            throw new OperationExecutionException(operation, internalContext, ex);
        } finally {
            if (connection != null) {
                try {
                    if (rollback) {
                        connection.rollback();
                    } else {
                        connection.commit();
                        /*
                         * Se publican los cambios hechos en el contexto interno
                         * en el contexto recibido por argumento
                         */
                        internalContext.setDatabaseConnection(context.getDatabaseConnection());
                        context.updateFrom(internalContext);
                    }
                    connection.setAutoCommit(true);
                    connection.close();
                } catch (SQLException ex) {
                    throw new OperationExecutionException(operation, internalContext, ex);
                }
            }
        }

    }

    public SqlSessionInterceptor(DataSource dataSource, Executor next) {
        super(next);
        this.dataSource = dataSource;
    }

}
```

Interceptor de manejo de transacciones MyBatis

Si en el proyecto se utiliza MyBatis para acceder a la base de datos, en lugar de tener en el contexto la conexión SQL, lo que se tiene el objeto `SqlSession` de MyBatis, por lo que en lugar de utilizar la clase `SqlSessionInterceptor` se utiliza esta, cuya lógica y comportamiento es análoga a la de la primera.

Esta implementación asume que en el contexto, en lugar de estar la propiedad `databaseConnection` con la conexión JDBC a la base de datos, está la propiedad `myBatisSession` de tipo `SqlSession` con la sesión de MyBatis.

```
// MyBatisSessionInterceptor.java

import org.apache.ibatis.exceptions.PersistenceException;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
```

```java
public class MyBatisSessionInterceptor extends Executor {

    private final SqlSessionFactory myBatisSessionFactory;

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        if (context.getMyBatisSession() != null) {
            /*
             * Ya está usando una conexión a base de datos, no es necesario abrir otra
             */
            return next.execute(operation, context);
        }

        boolean rollback = true;
        SqlSession myBatisSession = null;
        /*
         * Se crea un nuevo contexto, ya que los cambios que se hagan sobre este
         * solo se pueden publicar sobre el contexto recibido por argumento
         * cuando se haga commit, en caso de rollback los cambios deben ser
         * descartados
         */
        Context internalContext = new Context(context);
        try {
            myBatisSession = myBatisSessionFactory.openSession();
            internalContext.setMyBatisSession(myBatisSession);

            RESULT result = next.execute(operation, internalContext);
            rollback = false;
            return result;
        } catch (PersistenceException ex) {
            throw new OperationExecutionException(operation, internalContext, ex);
        } finally {
            if (myBatisSession != null) {
                try {
                    if (rollback) {
                        myBatisSession.rollback();
                    } else {
                        myBatisSession.commit();
                        /*
                         * Se publican los cambios hechos en el contexto interno
                         * en el contexto recibido por argumento
                         */
                        internalContext.setMyBatisSession(context.getMyBatisSession());
                        context.updateFrom(internalContext);
                    }
                    myBatisSession.close();
                } catch (PersistenceException ex) {
                    throw new OperationExecutionException(operation, internalContext, ex);
                }
            }
        }
    }
```

```
    public MyBatisSessionInterceptor(SqlSessionFactory myBatisSessionFactory, Executor next) {
        super(next);
        this.myBatisSessionFactory = myBatisSessionFactory;
    }
}
```

Interceptor de escritura detallada en el log

Este interceptor permite escribir en el log las operaciones solicitadas y su resultado, no escribe los errores ya que estos se suelen escribir en el log en la parte más externa de la aplicación. Este interceptor recibe por argumentos el logger en el cual se va a escribir y un nombre corto para la aplicación.

```
// LogDebugInterceptor.java

import java.util.logging.Level;
import java.util.logging.Logger;

public class LogDebugInterceptor extends Executor {

    private final Logger logger;
    private final String shortName;

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        boolean fineLevelEnabled = logger.isLoggable(Level.FINE);
        if (fineLevelEnabled) {
            logger.log(Level.FINE,
              "{0}: Execute operation requested.\n\nOperation: {1}\n\nContext: {2}",
              new Object[]{shortName, operation, context});
            RESULT result = next.execute(operation, context);
            logger.log(Level.FINE,
              "{0}: Execute operation executed.\n\nResult: {1}\n\nOperation: {2}\n\nContext: {3}",
              new Object[]{shortName, result, operation, context});
            return result;
        } else {
            return next.execute(operation, context);
        }
    }

    public LogDebugInterceptor(Logger logger, String shortName, Executor next) {
        super(next);
        this.logger = logger;
        this.shortName = shortName;
    }

}
```

Interceptor de validación de los datos de una operación

Este interceptor verifica que la operación cumple con todos los requisitos expuestos mediante anotaciones de Java Bean Validations, de fallar alguno lanza la excepción pública `ConstraintViolationException`. Este interceptor se suele utilizar típicamente para validar la entrada de operaciones provenientes de fuentes externas como un servicio JSON usado por JavaScript para peticiones AJAX y se suele colocar incluso antes del manejo de la conexión a la base de datos.

```
// ValidatorInterceptor.java

import java.util.Set;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class ValidatorInterceptor extends Executor {

    private final Validator validator;

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        Set violations = validator.validate(operation);
        if (!violations.isEmpty()) {
            throw new ConstraintViolationException(violations);
        }
        return next.execute(operation, context);
    }

    public ValidatorInterceptor(Executor next) {
        super(next);
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }
}
```

Servlets utilitarios

Servlet base para el manejo del contexto

La clase `ManagedHttpServlet` provee una implementación abstracta de un `Servlet` que provee:

- Manejo y recuperación del contexto desde la sesión del usuario.
- Implementación de mecanismo de seguridad para prevenir ataques XSFR.
- Escritura en el log de información completa cuando falla la aplicación.
- Se descartan errores relacionados a la ruptura de la comunicación con el navegador.
- Manejo de códigos de error HTTP: 419 (sesión expirada), 401 (login requerido), 403 (privilegios insuficientes), 412 (precondiciones incumplidas) y 500 (error interno)
- Permite enviar desde una petición AJAX a URL precisa en la cual se encuentra el navegador, para que, en caso de ocurrir un error en el servidor se pueda ver con exactitud en qué página se generó la petición AJAX. Esto es muy útil en Single Web Applications.

Métodos abstractos:

- **`void process(HttpServletRequest request, HttpServletResponse response, Context context)`**: en este método se debe implementar la acción a ser realizada por el `Servlet`.
- **`void logError(String requestInfo, String referrer, Context context, Exception ex)`**: este método debe escribir en el log de la aplicación la excepción ocurrida.

Métodos con implementación por defecto:

- **`boolean validateXsrfToken(HttpServletRequest request)`**: método que retorna si se debe validar el token XSFR, por defecto se validan todas las peticiones excepto las GET.

Métodos para ser usados en la implementación:

- **`void processRequest(HttpServletRequest request, HttpServletResponse response)`**: Por defecto no se tienden ninguna petición, por lo que es necesario que se sobreescriba el método `doGet`, `doPost`, etc. del Servlet para los métodos HTTP que se desean atender, en su implementación se debe invocar a este método.
- **`void saveContext(HttpServletRequest request, HttpServletResponse response, Context context)`**: por defecto los cambios en el contexto se guardan

una vez se culmina el procesamiento del método `processRequest`, pero esto tiene un problema, si se invocan operaciones que puedan crear o destruir la sesión, o alterar el token de seguridad XSFR, se debe invocar este método antes de que se envíe el cuerpo de la página.

```
// ManagedHttpServlet.java

import java.io.IOException;
import java.net.SocketException;
import java.net.URLDecoder;
import java.util.Enumeration;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.xml.bind.DatatypeConverter;

public abstract class ManagedHttpServlet extends HttpServlet {

    protected abstract void process(HttpServletRequest request, HttpServletResponse response,
        Context context) throws Exception;

    protected abstract void logError(String requestInfo, String referrer, Context context,
        Exception ex);

    protected boolean validateXsrfToken(HttpServletRequest request) {
        return !"GET".equals(request.getMethod());
    }

    protected final void processRequest(HttpServletRequest request, HttpServletResponse response) {
        Context context = null;
        try {
            HttpSession session = request.getSession(false);
            if (session != null) {
                context = (Context) session.getAttribute("uaithne-context");
                if (context != null) {
                    context = new Context(context);
                } else {
                    context = new Context();
                }
            } else {
                context = new Context();
            }

            if (session != null) {
                String xsrfTokenSession = context.getXsrfToken();
                String xsrfTokenCookie = getCookieValue(request, "XSRF-TOKEN");
                String xsrfTokenHeader = request.getHeader("X-XSRF-TOKEN");

                if (xsrfTokenSession != null) {
                    if (!xsrfTokenSession.equals(xsrfTokenCookie)) {
                        session.invalidate();
                        unsetAllCookies(request, response);
                        context = new Context();
                    } else if (validateXsrfToken(request) &&
                                !xsrfTokenSession.equals(xsrfTokenHeader)) {
                        session.invalidate();
                        unsetAllCookies(request, response);
                        context = new Context();
```

```
                }
            }
        }

        try {
            process(request, response, context);
            saveContext(request, response, context);
        } catch (InsufficientPrivilegesException ex) {
            if (session == null) {
                unsetAllCookies(request, response);
                if (hasCookie(request, "JSESSIONID")) {
                    response.sendError(419); // session expired
                } else {
                    response.sendError(401); // LogIn needed
                }
            } else {
                response.sendError(403); // Insufficient privileges
            }
        } catch (ConstraintViolationException ex) {
            logError(getRequestInfo(request), getReferrer(request), context, ex);
            response.sendError(412); // Precondition Failed
        }
    } catch (SocketException ignore) {
        // ignore it
    } catch (IOException ignore) {
        // ignore it
    } catch (Exception ex) {
        try {
            logError(getRequestInfo(request), getReferrer(request), context, ex);
            response.sendError(500);
        } catch (IOException ignore) {
            // ignore it
        }
    }
}

protected final void saveContext(HttpServletRequest request, HttpServletResponse response,
    Context context)
{

    HttpSession session = request.getSession(false);

    if (!context.hasData()) {
        if (session == null) {
            return;
        }

        session.invalidate();
        unsetAllCookies(request, response);
        return;
    }

    if (session == null) {
        session = request.getSession();
    }

    Context sessionContext = (Context) session.getAttribute("uaithne-context");
    if (sessionContext == null) {
        sessionContext = new Context();
        session.setAttribute("uaithne-context", sessionContext);
    }
```

```
    if (sessionContext.equals(context)) {
        return;
    }

    sessionContext.updateFrom(context);

    String xsrfToken = sessionContext.getXsrfToken();
    if (xsrfToken != null) {
        setCookieValue(request, response, "XSRF-TOKEN", xsrfToken);
    }
}

private String getReferrer(HttpServletRequest request) {
    try {
        /*
         * La cabecera X-CURRENT permite enviar la página actual en el navegador,
         * especialmente útil en peticiones AJAX desde el navegador en
         * aplicaciones Single Page Application dónde fragment, que es lo que
         * está después del símbolo #, no es incluido, y allí es donde se dice
         * realmente en qué página se está.
         *
         * Al construir la petición AJAX se debe establecer manualmente el
         * header X-CURRENT, si se desea que sea aprovechado aquí para escribir
         * en el log cuál es la url que provocó el error. El valor de este
         * header se puede construir de la siguiente manera:
         *     '@' + btoa(encodeURI(window.location.href)) + '!'
         */
        String result = request.getHeader("X-CURRENT");
        if (result == null) {
            return request.getHeader("Referer");
        }
        result = result.trim();
        if (result.length() < 2) {
            return request.getHeader("Referer");
        }
        try {
            result = result.substring(1, result.length() - 1);
            result = new String(DatatypeConverter.parseBase64Binary(result));
            result = URLDecoder.decode(result, "UTF-8");
            return result;
        } catch (Exception ignore) {
            return request.getHeader("Referer");
        }
    } catch(Exception e) {
        return null;
    }
}

private String getRequestInfo(HttpServletRequest request) {
    try {
        StringBuilder out = new StringBuilder();

        out.append("Request: {");
        out.append("\"RemoteAddr\":\"").append(request.getRemoteAddr());
        out.append("\",\"RemoteHost\":\"").append(request.getRemoteHost());
        out.append("\",\"RemotePort\":\"").append(request.getRemotePort());
        out.append("\",\"Protocol\":\"").append(request.getProtocol());
        out.append("\",\"Method\":\"").append(request.getMethod());
        out.append("\",\"RequestURI\":\"").append(request.getRequestURI());
        out.append("\",\"QueryString\":\"").append(request.getQueryString());
        out.append("\"}. Headers: {");
```

```
        Enumeration<String> headerNames = request.getHeaderNames();
        boolean comma = false;
        while (headerNames.hasMoreElements()) {

            String headerName = headerNames.nextElement();
            if (comma) {
                out.append(",");
            } else {
                comma = true;
            }

            out.append("\"");
            out.append(headerName);
            out.append("\":[");

            Enumeration<String> headers = request.getHeaders(headerName);
            boolean requireComma = false;
            while (headers.hasMoreElements()) {
                if (requireComma) {
                    out.append(",");
                } else {
                    requireComma = true;
                }
                String headerValue = headers.nextElement();
                out.append("\"");
                out.append(headerValue);
                out.append("\"");
            }
            out.append("]");
        }
        out.append("}");

        return out.toString();
    } catch (Exception e) {
        return null;
    }
}

private String getCookieValue(HttpServletRequest request, String name) {
    Cookie[] cookies = request.getCookies();
    if (cookies == null) {
        return null;
    }
    for (Cookie cookie : cookies) {
        if (cookie.getName().equals(name)) {
            return cookie.getValue();
        }
    }
    return null;
}

public static void setCookieValue(HttpServletRequest request, HttpServletResponse response,
    String name, String value)
{

    String path = request.getContextPath() + "/";
    Cookie cookie = new Cookie(name, value);
    cookie.setHttpOnly(false);
    cookie.setPath(path);
    response.addCookie(cookie);
}
```

```
    private boolean hasCookie(HttpServletRequest request, String name) {
        Cookie[] cookies = request.getCookies();
        if (cookies == null) {
            return false;
        }
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }

    private void unsetAllCookies(HttpServletRequest request, HttpServletResponse response) {
        unsetCookie(request, response, "JSESSIONID");
        unsetCookie(request, response, "XSRF-TOKEN");
    }

    private void unsetCookie(HttpServletRequest request, HttpServletResponse response, String name) {
        String path = request.getContextPath() + "/";
        Cookie cookie = new Cookie(name, null);
        cookie.setHttpOnly(false);
        cookie.setMaxAge(0);
        cookie.setPath(path);
        response.addCookie(cookie);
    }
}
```

Servlet para peticiones AJAX

Esta clase permite crear un servicio RPC para peticiones AJAX desde el navegador que permite ejecutar operaciones pasadas en formato JSON, donde la propiedad de nombre "`type`" contiene el nombre de la operación que se desea ejecutar, y el resto de las propiedades corresponden a las propiedades de la operación, retorna el resultado de la operación en formato JSON.

Ejemplo de JSON a ser enviado para ejecutar la operación:

```
{
    "type": "SelectCalendarById",
    "id": 1234
}
```

El resultado de esta operación podría ser:

```
)]}',
{
    "id": 1234,
    "title": "Mi calendario",
    "description": "Descripción de mi calendario"
}
```

Las respuestas empiezan por el prefijo de seguridad `")]}',\n"` para evitar que se puedan usar en ataques XSRF combinados con ataques al JSON.

Manejo de la permisología

En la clase `JsonServiceExposeOperations` (que se debe implementar) se exponen las operaciones que se pueden ejecutar para una petición en particular, para ello se consulta al contexto si hay un usuario autenticado o si tiene algún rol en particular. La idea detrás de este control es que se publiquen solamente aquellas operaciones para las cuales el usuario tiene permisos para ejecutar, evitando así que un atacante pueda ejecutar operaciones para las cuales el usuario desde la interfaz no usaría nunca.

Ejemplo de la clase `JsonServiceExposeOperations`

```
// JsonServiceExposeOperations.java
import com.google.gson.typeadapters.RuntimeTypeAdapterFactory;

public class JsonServiceExposeOperations {

    public static void registerOperations(Context context,
                                          RuntimeTypeAdapterFactory<Operation> factory)
    {

        /*
         * Operaciones disponibles incluso sin estar autenticado
```

```
 */

factory.registerSubtype(LogIn.class);
factory.registerSubtype(LogOut.class);
factory.registerSubtype(Register.class);
[...]

if (context.isUserLoggedIn()) {
    // Si no está autenticado no expone más operaciones
    return;
}

/*
 * Operaciones que solo requieren estar autenticado
 */

factory.registerSubtype(ChangePassword.class);
[...]

/*
 * Se añaden operaciones en función de los roles del usuario
 */

// Operaciones para la tabla Calendar

if (context.canViewCalendar()) {
    factory.registerSubtype(SelectCalendarById.class);
    [...]
}

if (context.canCreateCalendar()) {
    factory.registerSubtype(InsertCalendar.class);
    [...]
}

if (context.canUpdateCalendar()) {
    factory.registerSubtype(UpdateCalendar.class);
    [...]
}

if (context.canDeleteCalendar()) {
    factory.registerSubtype(DeleteCalendarById.class);
    [...]
}

// Operaciones para la tabla Event

if (context.canViewEvent()) {
    factory.registerSubtype(SelectEventById.class);
    [...]
}

if (context.canCreateEvent()) {
    factory.registerSubtype(InsertEvent.class);
    [...]
}

if (context.canUpdateEvent()) {
    factory.registerSubtype(UpdateEvent.class);
    [...]
}
```

```
        if (context.canDeleteEvent()) {
            factory.registerSubtype(DeleteEventById.class);
            [...]
        }
    }

}
```

Implementación del servicio

Para implementar la transformación de objetos Java a JSON y viceversa se utiliza en esta implementación Gson (https://github.com/google/gson) al cual se le indican el listado de operaciones permitidas para el usuario. Nota: este código usa la clase `RuntimeTypeAdapterFactory` cuyo código se puede descargar de https://github.com/google/gson/tree/master/extras/src/main/java/com/google/gson/typeadapters

El manejo de la fecha y hora se hace ignorando las diferencias de zonas horarias entre el navegador y el servidor, de forma tal que la fecha y hora que se indique en el javascript corresponda con la del servidor, como si estuviesen en la misma zona horaria. Este comportamiento se implementa en la clase `DateTypeAdapter` y puede no ser deseado en algunas aplicaciones. La clase `TimeTypeAdapter` hace lo mismo, pero para las nuevas clases de fechas y horas de la JDK 8, específicamente: `LocalDateTime`, `LocalDate`, `LocalTime` e `Instant`, la representación de fechas o horas con zona horaria no están incluidas en la implementación actual.

Requisitos:

- **`Executor getServiceBus()`** como método estático en la clase **`Bus`**: este método retorna el ejecutor de entrada a la aplicación desde el servicio JSON.
- **`void registerOperations(Context context, RuntimeTypeAdapterFactory<Operation> factory)`** como método estático en la clase **`JsonServiceExposeOperations`**: este método inicializa las operaciones que están permitidas ser ejecutadas en la solicitud actual. Para publicar una operación hay que llamar al método `registerSubtype` de `factory` y pasar por argumento la clase de la operación que se desea publicar.

```
// JsonService.java

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonParseException;
import com.google.gson.typeadapters.RuntimeTypeAdapterFactory;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Timestamp;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class JsonService extends ManagedHttpServlet {

    private static final Logger LOGGER = Logger.getLogger(JsonService.class.getName());

    @Override
    protected void process(HttpServletRequest request, HttpServletResponse response, Context context)
        throws Exception
    {
        Gson gson = initGson(context);
        String jsonOperation = getPostData(request);
        Operation operation;
        try {
            operation = fromJson(gson, jsonOperation);
        } catch(InsufficientPrivilegesException ex) {
            throw ex;
        } catch(JsonParseException ex) {
            throw new JsonParseException("Invalid JSON: " + jsonOperation, ex);
        } catch(Exception ex) {
            throw new JsonParseException("Invalid JSON: " + jsonOperation, ex);
        }
        if (operation != null) {
            Object result = Bus.getServiceBus().execute(operation, context);
            saveContext(request, response, context);
            String jsonResult = gson.toJson(result);
            writeOutput(response, jsonResult);
        }
    }

    private Gson initGson(Context context) {
        RuntimeTypeAdapterFactory<Operation> factory =
            RuntimeTypeAdapterFactory.of(Operation.class, "type");

        JsonServiceExposeOperations.registerOperations(context, factory);
        GsonBuilder builder = new GsonBuilder();

        DateTypeAdapter dateTypeAdapter = new DateTypeAdapter();
        builder.registerTypeAdapter(Date.class, dateTypeAdapter);
        builder.registerTypeAdapter(java.sql.Date.class, dateTypeAdapter);
        builder.registerTypeAdapter(java.sql.Time.class, dateTypeAdapter);
        builder.registerTypeAdapter(Timestamp.class, dateTypeAdapter);

        TimeTypeAdapter timeTypeAdapter = new TimeTypeAdapter();
        builder.registerTypeAdapter(LocalDateTime.class, timeTypeAdapter);
        builder.registerTypeAdapter(LocalDate.class, timeTypeAdapter);
        builder.registerTypeAdapter(LocalTime.class, timeTypeAdapter);
        builder.registerTypeAdapter(Instant.class, timeTypeAdapter);

        builder.registerTypeAdapterFactory(factory);
        Gson gson = builder.create();
        return gson;
    }

    private Operation fromJson(Gson gson, String jsonOperation)
        throws InsufficientPrivilegesException
    {
        try {
            return gson.fromJson(jsonOperation, Operation.class);
        } catch (JsonParseException e) {
```

```
            throw new InsufficientPrivilegesException(e);
        }
    }

    private void writeOutput(HttpServletResponse response, String output)
            throws ServletException, IOException
    {
        response.setContentType("application/json;charset=UTF-8");
        response.addHeader("Cache-Control", "no-cache");
        try (PrintWriter out = response.getWriter()) {
            out.print(")]}',\n");
            out.print(output);
        }
    }

    private static String getPostData(HttpServletRequest request) throws IOException {
        StringBuilder result = new StringBuilder();
        BufferedReader reader = request.getReader();
        reader.mark(10000);

        String line = reader.readLine();
        while (line != null) {
            result.append(line).append("\n");
            line = reader.readLine();
        }
        reader.reset();
        // do NOT close the reader here, or you won't be able to get the post data twice

        return result.toString();
    }

    @Override
    protected void logError(String requestInfo, String referrer, Context context, Exception ex) {
        LOGGER.log(Level.SEVERE,
            "Error processing json request.\n\nContext: " + context +
            "\n\nReferrer: " + referrer + "\n\nRequest info:" + requestInfo, ex);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

}

// DateTypeAdapter.java

import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonElement;
import com.google.gson.JsonParseException;
import com.google.gson.JsonPrimitive;
import com.google.gson.JsonSerializationContext;
import com.google.gson.JsonSerializer;
import com.google.gson.JsonSyntaxException;
import java.lang.reflect.Type;
import java.sql.Timestamp;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
import java.util.Locale;
import java.util.TimeZone;

public class DateTypeAdapter implements JsonSerializer<Date>, JsonDeserializer<Date> {
    private final DateFormat iso8601Format;

    public DateTypeAdapter() {
        this.iso8601Format = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", Locale.US);
        this.iso8601Format.setTimeZone(TimeZone.getTimeZone("UTC"));
    }

    @Override
    public JsonElement serialize(Date src, Type typeOfSrc, JsonSerializationContext context) {
        String dateFormatAsString = iso8601Format.format(src);
        return new JsonPrimitive(dateFormatAsString);
    }

    @Override
    public Date deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)
            throws JsonParseException
    {
        if (!(json instanceof JsonPrimitive)) {
            throw new JsonParseException("The date should be a string value");
        }
        Date date = deserializeToDate(json);
        if (typeOfT == Date.class) {
            return date;
        } else if (typeOfT == Timestamp.class) {
            return new Timestamp(date.getTime());
        } else if (typeOfT == java.sql.Date.class) {
            return new java.sql.Date(date.getTime());
        } else if (typeOfT == java.sql.Time.class) {
            return new java.sql.Time(date.getTime());
        } else {
            throw new IllegalArgumentException(getClass() + " cannot deserialize to " + typeOfT);
        }
    }

    private Date deserializeToDate(JsonElement json) {
        try {
            return iso8601Format.parse(json.getAsString());
        } catch (ParseException e) {
            throw new JsonSyntaxException(json.getAsString(), e);
        }
    }

}

// TimeTypeAdapter.java

import com.google.gson.JsonDeserializationContext;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonElement;
import com.google.gson.JsonParseException;
import com.google.gson.JsonPrimitive;
import com.google.gson.JsonSerializationContext;
import com.google.gson.JsonSerializer;
import java.lang.reflect.Type;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
```

```
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.time.temporal.Temporal;
import java.util.Locale;

public class TimeTypeAdapter implements JsonSerializer<Temporal>, JsonDeserializer<Temporal> {

    private final DateTimeFormatter iso8601Format;

    public TimeTypeAdapter() {
        this.iso8601Format = DateTimeFormatter
            .ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", Locale.US).withZone(ZoneOffset.UTC);
    }

    @Override
    public JsonElement serialize(Temporal src, Type typeOfSrc, JsonSerializationContext context) {
        String dateFormatAsString = iso8601Format.format(src);
        return new JsonPrimitive(dateFormatAsString);
    }

    @Override
    public Temporal deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)
            throws JsonParseException
    {
        if (!(json instanceof JsonPrimitive)) {
            throw new JsonParseException("The date should be a string value");
        }

        String text = json.getAsString();
        if (typeOfT == LocalDate.class) {
            return LocalDate.parse(text, iso8601Format);
        } else if (typeOfT == LocalTime.class) {
            return LocalTime.parse(text, iso8601Format);
        } else if (typeOfT == LocalDateTime.class) {
            return LocalDateTime.parse(text, iso8601Format);
        } else if (typeOfT == Instant.class) {
            return Instant.from(iso8601Format.parse(text));
        } else {
            throw new IllegalArgumentException(getClass() + " cannot deserialize to " + typeOfT);
        }
    }
}
```

Acceso del servicio desde el navegador

La función `rpc` permite enviar una petición al servidor con facilidad, el primer argumento es el objeto JavaScript con la información de la operación y la propiedad “`type`” a ser enviado al servidor, el segundo parámetro en una función que se ejecuta en caso de éxito y recibe por argumento el objeto JavaScript con la respuesta del servidor, y el tercer parámetro es una función que se ejecuta en caso de fracaso y recibe por argumento el error.

Esta implementación envía al servidor la URL actual del navegador completa en la cabecera `X-CURRENT` y también envía en la cabecera `X-XSRF-TOKEN` el valor del token de seguridad leído de la cookie `XSRF-TOKEN`. Adicionalmente transforma todas las fechas y horas enviadas por el servidor en objetos Date de JavaScript.

```
function rpc(operation, onSuccess, onError) {
    var req = new XMLHttpRequest();
    var url = 'http://www.example.com/JSONService';
    req.open('POST', url, true);

    req.setRequestHeader('Content-Type', 'application/json;charset=utf-8');
    req.setRequestHeader('Accept', 'application/json');
    req.setRequestHeader('Cache-Control', 'no-cache');

    var xsrfCookieValue = getCookie('XSRF-TOKEN');
    if (xsrfCookieValue) {
        req.setRequestHeader('X-XSRF-TOKEN', xsrfCookieValue);
    }
    req.setRequestHeader('X-CURRENT', '@' + btoa(encodeURI(window.location.href)) + '!');

    req.onreadystatechange = function () {
        if (req.readyState != 4) {
            return;
        }
        if (req.status >= 200 && req.status < 300) {
            try {
                var result = fromJson(req.responseText);
                if (onSuccess) {
                    onSuccess(result);
                }
            } catch (e) {
                e.request = req;
                e.operation = operation;
                if (onError) {
                    onError(e);
                }
            }
        } else {
            var err = Error('The execution of operation with type "'
                + (operation ? operation.type  : undefined)
                + '" failed. Result: ' + req.status + ' (' + req.statusText + ')');
            err.request = req;
            err.operation = operation;
            if (onError) {
                onError(err);
            }
        }
    };

    var json = JSON.stringify(operation);
    req.send(json);
}

/*
 * Funciones privadas
 */

function fromJson(str) {
    if (!str) {
        str = '';
    }
    var PROTECTION_PREFIX = /^\)\]\}',\n/;
    var result = str.replace(PROTECTION_PREFIX, '');
    return JSON.parse(result, dateParser);
}
```

```
var dateFormatISO =
    /^(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\d{2}):(\d{2}(?:\.\d*))(?:Z|(\+|-)([\d|:]*))?$/;

function dateParser(key, value) {
    if (typeof value === 'string') {
        if (dateFormatISO.test(value)) {
            return new Date(value);
        }
    }
    return value;
}

function getCookie(cname) {
    var name = cname + '=';
    var ca = document.cookie.split(';');
    for (var i = 0, length = ca.length; i < length; i++) {
        var c = ca[i];
        while (c.charAt(0) === ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) === 0) {
            return c.substring(name.length, c.length);
        }
    }
}
```

Servlet de subida de fichero

Para implementar un Servlet que permita subir un fichero solo hay que utilizar las capacidades que ofrecen los Servlets de Java, si se extiende de `ManagedHttpServlet`, se puede aprovechar las capacidades ya incluidas por este.

Nota: No hay que olvidar configurar la subida de fichero en la sección `multipart-config` del `web.xml` o usar la anotación `MultipartConfig`.

Ejemplo de Servlet de subida de fichero

```
// FileUpload.java

import java.io.InputStream;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Part;

public class FileUpload extends ManagedHttpServlet {
    private static final Logger LOGGER = Logger.getLogger(FileUpload.class.getName());

    @Override
    protected void process(HttpServletRequest request, HttpServletResponse response,
        Context context) throws Exception
    {
        /*
         * Se comprueba si el usuario tiene permisos para realizar la acción
         */
        if (!context.canUploadFile()) {
            throw new InsufficientPrivilegesException();
        }

        Part filePart = request.getPart("file");
        try(InputStream filecontent = filePart.getInputStream()) {

            /*
             * Se procesa el fichero subido
             */

            [...]
        }
    }

    @Override
    protected void logError(String requestInfo, String referrer, Context context,
        Exception ex)
    {
        LOGGER.log(Level.SEVERE, "Error while upload a file.\n\nContext: " + context +
            "\n\nReferrer: " + referrer + "\n\nRequest info:" + requestInfo, ex);
    }
```

```
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

Servlet descarga de fichero

Para implementar un Servlet que permita descargar un fichero solo hay que utilizar las capacidades que ofrecen los Servlets de Java, si se extiende de `ManagedHttpServlet`, se puede aprovechar las capacidades ya incluidas por este.

```
// FileDownload.java

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FileDownload extends ManagedHttpServlet {

    private static final Logger LOGGER = Logger.getLogger(FileDownload.class.getName());

    @Override
    protected void process(HttpServletRequest request, HttpServletResponse response,
        Context context) throws Exception
    {
        /*
         * Se comprueba si el usuario tiene permisos para realizar la acción
         */
        if (!context.canDownloadFile()) {
            throw new InsufficientPrivilegesException();
        }

        byte[] content = [...]
        if (content == null) {

            /*
             * Si el fichero no se encuentra se retorna el código 404
             */

            response.sendError(404); // Not found
            return;
        }

        String mimeType = null; //[...]
        String fileName = null; //[...]
        response.setContentType(mimeType);
        response.setContentLength(content.length);
        response.setHeader("Content-disposition", "attachment;filename=" + fileName);

        try (ServletOutputStream outStream = response.getOutputStream()) {
            outStream.write(content);
        }
    }
```

```
    @Override
    protected void logError(String requestInfo, String referrer, Context context,
        Exception ex)
    {
        LOGGER.log(Level.SEVERE, "Error while download a file.\n\nContext: " + context +
            "\n\nReferrer: " + referrer + "\n\nRequest info:" + requestInfo, ex);
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

}
```

Servlet de inicialización

Si la aplicación contiene demonios, tareas programadas, lógica de inicialización, etc. lo más conveniente es utilizar un Servlet de inicialización de Java.

Nota: No hay que olvidar incluir la clase en la sección `listener` del `web.xml` o usar la anotación `WebListener`.

Ejemplo de un Servlet de inicialización que crea una tarea programada

```
// ApplicationStartUpListener.java

import java.util.logging.Logger;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ApplicationStartUpListener implements ServletContextListener {

    private static final Logger LOGGER = Logger.getLogger(
        ApplicationStartUpListener.class.getName());

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        /*
         * Lógica a ser ejecutada cuando la aplicación se inicie
         */
        LOGGER.fine("Application starting...");
        SomeDaemon.startDaemon();
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        /*
         * Lógica a ser ejecutada cuando la aplicación se detenga
         */
        LOGGER.fine("Application shutting down...");
    }

}

// SomeDaemon.java

import java.util.Timer;
import java.util.TimerTask;
import java.util.logging.Logger;

public class SomeDaemon extends TimerTask {

    private static final Logger LOGGER = Logger.getLogger(SomeDaemon.class.getName());
    private static Timer timer;
    private static boolean isRunning = false;
```

```
    @Override
    public void run() {
        if (isRunning) {
            /*
             * Si la acción a ser ejecutada por la tarea programada es muy larga
             * esta se puede volver a invocar cuando la invocación previa aun
             * no ha terminado, por no que se descarta esa nueva ejecución entre
             * tanto la otra no haya terminado
             */
            return;
        }

        try {
            isRunning = true;
            /*
             * Lógica a ser ejecutada por la tarea programada
             */
            LOGGER.fine("The daemon is running...");
        } finally {
            isRunning = false;
        }
    }

    public static void startDaemon() {
        timer = new Timer("Some Daemon", true);
        long delay = 60000; // Esperar 60 segundos para la primera ejecución
        long period = 30000; // Ejecutar cada 30 segundos
        timer.scheduleAtFixedRate(new SomeDaemon(), delay, period);
    }

}
```

Ω fin Ω