Uaithne Java

Designed by: **Juan Luis Paz Rojas**

Foreword

Software architecture can be very exciting, but it can also be very frustrating. In my work as a software architect, I have seen many projects happen before, some that other people had started and other projects which I designed, some with more successful architectures, others with architectures that were not appropriate for the requirements of development and somewhere the requirements did not exist.

I want to focus here especially on the architecture of the backend of information systems, whose architecture is often deceptively simple, and are usually built with an architecture n-tier layers. These systems usually have a Web interface (typically Single Page Application) or an interface for mobile devices. In these systems, the decoupling of the backend interface is usually large (or at least this is desired), and in the end the interface finishes invoking the server to perform some operation.

An important issue in software architecture, although often forgotten, is how to get an architecture that drives the productivity of the development team. The idea is to achieve greater and better results with less human effort in both, the initial development phase, and the next phases of modification and expansion of the system. If we observe what happens in other sectors, the improvement of productivity is achieved through the automation of repetitive tasks that do not require a significant cognitive effort on the part of who performs them. The development of an application is a process of producing a product, often tailored, in which the productivity added to the end users is usually valued. But, the productivity during the development of this is usually forgotten, in which, with the choice of the architecture and the appropriate tools, you can greatly improve the productivity of the development team.

With Uaithne I propose an alternative architecture for the backend of the applications that allows to boost the productivity of the team and to develop this part of the application without sacrificing the virtues of an n-tier architecture.

The central philosophy of Uaithne is to make the backend resemble a LEGO® game, where highly interchangeable pieces are made up to achieve the desired result, whose final function is to achieve the execution of an operation (command).

The development of the Uaithne architecture has been a challenge to improve the productivity of the development team that I started in mid-August 2011, and whose first prototype I presented on August 28, 2011; and that has been maturing over time. After the successful implementation in many projects in Delonia Software of various kinds, and thanks to the collaboration of Delonia, I have managed to bring Uaithne to a sufficient degree of

maturity and support it in real large size projects that validate the fundamentals of the architecture and the proposed tools.

Complementary tools

Improving the productivity of the development team has been one of the great motivations behind the design of the architecture of Uaithne, and it alone represents a great improvement over the classical n-tier architecture. After implementing the new architecture there was still room to improve productivity, automatically generating code that is responsible for handling the verbosity required by Java, and above all, that generates the access code to the database with its respective SQL queries that handles most operations.

Uaithne is accompanied, optionally, with a code generator that allows to generate (and regenerate) the operations and entities from a small definition, as well as the logic of access to database required by them, using MyBatis as database access framework. In this way, only those activities that really require human work are left to the programmer due to the decisions that must be made by him.

The Uaithne architecture and the code generator working together have proved in a multitude of projects of different sizes to be a very useful tool for the programming of the backend of applications, and during this time, achieving a high productivity of the development team.

In this document only the architecture is considered, the code generator is presented separately and its usage is optional.

The Dagda's harp

Dagda ("the good god"), is a leader, father-figure of immense power, druid of the Tuatha Dé; symbol of life and death, known among other names by Ruad Rofhessa ("lord of great knowledge"). Dagda became king of the Tuatha Dé Danann after his predecessor, Nuada, was wounded in battle; a reign of 70 or 80 years is attributed to him (depending on the source).

Dagda possessed many magical objects, but perhaps his most valuable possession was Uaithne, a beautiful and powerful harp that brought fear and destruction to the enemies of the Tuatha Dé Danann. This powerful harp, also known as Dur Dá Blá ("Oak of two blossoms") or as Cóir Cethairchuir ("Harmony of four angles"), had, among others, the ability to control and put in order the seasons, infusing courage to the hearts of the warriors and heal their wounds after the battle; Uaithne's music could cast spells on everyone who could hear the magical notes that floated over the green hills of Ireland.

During the second battle of Mag Tured, the harp was stolen by the Fomoré; with it in their power the Fomoré thought they could use it to cast spells against the Tuatha Dé Danann. Therefore, Dagda and his companions Lug and Ogma went to the place where the Fomoré took refuge to recover it. Once inside, they managed to make their way up to the banquet room in which the Fomoré celebrated the capture of the harp, on whose wall the instrument was hung. Then, Dagda called the harp with the following song:

Come oak of the two blossoms*!*
Come, harmony formed of four angles!
Come summer, come winter!
Voice of harps, bagpipes and flutes!

Upon hearing it, the harp fell off the wall and flew magically into the hands of its owner, so quickly that it killed nine people in its path. Dagda then began to play Uaithne and performed the Golltraigi ("the chirping chord"), which caused all the present Fomoré to begin to cry and moan; next, he played the Genntraigi ("the chord of laughter"), which caused everyone to laugh without control; and finally Dagda interpreted the Súantraigi ("the dream chord"), which left them all soundly asleep. Taking advantage of it, Dagda, Lugh and Ogma left unharmed of the camping.

Source:
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
Version: 1.0.0 - March 27, 2018

Table of contents

Uaithne Java

Designed by: **Juan Luis Paz Rojas**

Introduction

Goals

- Easy application for backend development
- Regardless of its complexity or size
- Allow to grow without increasing complexity or reduce maintainability
- Allow to modify the behaviour easily
- Allow to improve the productivity of the team involved in the application development

Philosophy

*Get the backend resemble a **set of LEGO®**, where there are **highly interchangeable pieces** that are **made up to achieve** the desired result.*

Principles

- Every action performed by the application is done with commands. A command is an object that represents the action to be executed and contains all the information required to be achieved. These objects will be called “operations”.
- The execution of any operation is delegated to other objects that will be called “executors”.
- The execution of an operation must be possible in different layers, being the result of the sum of the actions performed in the different layers.
- The layers must have a low coupling among each other, having a minimal dependency. All the communication between layers is made between the single Executor interface, which represents a layer.
- Layers should be easily arranged, allowing addition to a whole new layer in the middle of the existing ones or reorganising it, without necessary large changes in the system.
- An executor can handle all the operations that cross it or its specified ones.
- To make easy programming, operations are grouped into logical units called “modules”.
- Separation of concerns must be allowed. Thus, all the required logic that is not interrelated can be separated into different layers. Having smaller pieces of logic is possible to improve the maintainability and reusability of it.
- Build interceptors must be possible. An interceptor allows performing actions before or after the execution of an operation, or when an exception is thrown. An interceptor may modify the data contained inside the operation, or the result of it, or even prevent the execution of the operation. When using interceptors, it is possible to have

some logic as a layer around an implementation, delegating the implementation into another layer.

Design patterns

Uaithne is the result of an ingenious combination of design patterns:

- Command
- Chain of Responsibility
- Strategy

This combination of patterns results in structure on which the backend of an application is build, allowing to take advantages of the Aspect Oriented Programming features without their complications.

Remarks

- At least Java 8 is required because Lambdas functions are used in this design.
- Examples here uses Java 8 date and time API, especially the class `LocalDateTime` that allows storing a date and time without time zone.
- The design has no complex external dependencies, in consequence, you can use it in a simple web container, such as Tomcat, Jetty, Embedded Jetty, or even, in an OSGi container like Apache Felix or its derivative Apache Kafka.
- Examples here use the `@Data` annotation provided by the Lombok project (https://projectlombok.org/) with the purpose to reduce the amount of verbose code required by Java. Its usage is optional.

  The `@Data` annotation automatically generates the getters and the setters for each field in the annotated class, as well, generate the methods `equals`, `hashCode` and `toString`.

Example application

To show how to use Uaithne, a small calendar application is going to be used as an example. This application has calendars with events. The calendar application database has the following tables:

```
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)
);
```

Understanding the framework components

Entities

In Uaithne the information and the logic must be separated,and the data is represented using POJOs (Plain Old Java Objects); these classes encapsulate the information without adding logic of any kind, so an entity class will only contain the fields to store the information, the properties that allow access or establish the value of these fields, and the instance constructors if desired. Usually, they have their own implementation of the `equals`, `hashCode` and `toString` methods. Additionally, they usually implement the `Serializable` interface.

For the calendar application, an entity will be created for each database table. These entities are those that will then be used to transport the information contained in their corresponding tables. The classes that represent the entities would be similar to:

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

    // The @Data annotation automatically generates:
    // getters, setters, equals, hashCode, toString

    public Calendar() { }
    // Other constructors...
}
```

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

    // The @Data annotation automatically generates:
    // getters, setters, equals, hashCode, toString

    public Event() { }
    // Other constructors...
}
```

Operations

Operations represent an action to be executed by the system, represented as an object, which contains the necessary information for its execution, but it never contains the logic to be executed.

All operations must extend from the `Operation` class which definition is as follows:

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

}
```

This class receives as generic argument the type of the result of the operation, in case it doesn't return any value, the type of the result must be the `Void` class that only supports null.

The operations are modeled just like the entities, they only contain the fields where the information is stored, the properties that allow accessing or establishing the value of these fields, and instance constructors if desired. Usually, they have their own implementation of the methods `equals`, `hashCode` and `toString`; additionally, the `Serializable` interface is implemented (here already included by default in the base `Operation` class).

All operations must extend the `Operation` class, for example, if you want to get a list of events for a particular calendar and a particular moment, you can create a class that represents the action of getting that list that would be similar to:

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

    // The @Data annotation automatically generates:
    // getters, setters, equals, hashCode, toString

    public SelectMomentEvents() { }
    // Other constructors...
}
```

In the `SelectMomentEvents` operation:

- In order to be able to execute this operation, the id of the calendar and the moment that you want to consult are required.
- The type of the operation's result is `List<Event>`.

Context

Context is a class that contains complementary information to the execution of an operation, and its content is independ on the operation to be executed.

The information about the identity of the user who performs the operation, its permissions, etc. is usually placed in the context. In web applications the context is usually stored in the user's session because its content can not be assumed to be correct if it is sent from the browser without compromising the security of the system.

The `Context` class represents the execution context. In this class you must add the necessary properties. The initial definition of the `Context` class is:

```
public class Context implements Serializable {
    // Add here the context properties
}
```

You can treat the context class as another entity, so that, just like an entity class, it only contains the fields where the information is stored, the properties that allow access or establish the value of these fields, constructors if desired. Usually, it has its own implementation of the `equals`, `hashCode` and `toString` methods. Additionally, it usually implements the `Serializable` interface.

Typical elements to include in the context:

- **User's identifier authenticated in the application:** this property contains the identifier of the user who is logged in, i.e., the identifier of the user who entered the username and password validated in the system.
- **User's identifier who runs the application**: this property contains the identifier of the user who is using the application; in many cases it is the same user who logged in. But keeping both separated allows handling more complex situations, such as users who need to operate as if they were another user (typical in technical support). Doing this separation from the beginning in the application is recommended even if its use is not seen a priori, because modifying the application's logic a posteriori is very complex, regarding the low cost of handling it from the beginning.
- **Roles of the user who runs the application**: this property contains the necessary information to know which operations can be executed by the user who is using the operation. The roles can be defined in an array of 32-bit integers as a bitmap, where each bit within the array corresponds to a role. When coding the roles in an array of 32-bit integers, it allows to store it in less space, and can also be used as a bitmap in JavaScript without any problem (JavaScript only supports bitmap over 32-bit integers). To facilitate the reading of the bitmap, a method is usually created for each role that returns a boolean that indicates whether the user has that role in the bitmap.
- **Language**: if necessary, you can include a property that indicates the user's language, allowing some operations to take internationalization into account.

- **User information**: if necessary, and for performance reasons, you can include some user data that due to the frequency used is convenient to have it in the session instead of reading it constantly from the database. It is very important to include as little information as possible, since its storage in the session is expensive.
- **Security tokens**: if the application uses a security token, by instance, the token need to avoid XSRF (Cross Site Request Forgery) attacks.
- **Database connection**: it must be marked as transient, i.e. to exclude it from serialization (thus avoiding being stored in the session, for example). This field must be excluded from the `equals`, `hashCode` and `toString` methods. The inclusion of the database connection in the execution context allows to easily control the transactionality of the system and the scope of it.

When executing an operation, the execution context is always available, and it can be used to complete the information needed to execute it.

It is very convenient to create contextualized operations instead of more generic ones, so it is preferable to create an operation "Give me my policies" instead of the more generic "Give me the policies of the indicated user", the creation of contextualized operations greatly reduces the security problems related to the identity of the user, since the user's identity is read from the context that is always stored in a secure place (as in the user's session on the server) instead of relying on the requestor.

It is recommended **to use contextualized operations instead of the more generic ones**, even if that implies having two operations that apparently do the same, except that one reads the information from the context and the other from the operation.

Additionally, to facilitate the management of context and subcontexts, a constructor is usually included that receives another context by argument, allowing to create a copy of the context, and the a `updateFrom` method, which copies the values from the context it receives by argument to the context over which it is invoked.

The execution context must be stored in a secure place, typically in the session of the user in the server. Under no circumstances should the context information come from the browser, mobile device, or any other untrusted source. To be able to detect if the context contains information that must be stored, the `hasData` method that returns a boolean is included, which indicates whether the context contains information that must be stored in the session. If the `hasData` method returns false it is because the session is not required, and if the session exists, it can be destroyed.

To facilitate close a session, the `clearUserData` method is included, which deletes all user data from the context, thus forcing the session to be destroyed at the end of the execution of the entire process. The `isUserLoggedIn` method returns a boolean that indicates if there is an active user.

Example of context for the calendar management application:

```java
// 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;
    }

    /**
     * Allows to know if the user has a particular role within the role bitmap
     * array, where each bit is a role
     *
     * @param index index within the array where the role is
     * @param bit position of the bit in the bitmap that represents the role
     * @return true if the user has the role, false otherwise
     */
    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);
    }
}
```

Executors

An executor is in charge of executing the logic associated with an operation, and there may be two cases:

- Execute a logic for **all the operations** that the executor goes through the executor
- Execute a logic for **operations of a specific type**

All executors extend the `Executor` class, for which the definition is:

```
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) { [...] }
}
```

The `Executor` class contains the following elements:

- **`HashMap<Class, BiFunction> handlers`**: HashMap for internal use that contains as a value the function to be executed when an operation of the type indicated in the key is received. If no entry is registered in this dictionary for the type of operation to be executed, the execution is delegated to the executor indicated in the `next` property.

- **`Executor next`**: Read-only property, with its respective getter, which contains the reference to the next executor to be executed if the current executor is not able to handle the received operation. In case of null, if the operation is not handled by the current executor, an exception is raised stating that the operation could not be handled.

- **`RESULT execute(OPERATION operation, Context context)`**: This method is responsible for receiving an operation and executing it. Additionally, it receives the execution context.

- **`RESULT executeAnyOperation(OPERATION operation, Context context)`**: This method is responsible for executing an operation independently of its type. Additionally, it receives the execution context. This method can be replaced by another implementation that typically ends up invoking the `execute` method of the `next` property or the base implementation to continue the execution of the operation.

- **`void handle(Class<OPERATION> operationType, BiFunction<OPERATION, Context, RESULT> handler`**: This method is responsible for registering the function passed as the second argument responsible for executing the logic associated with the operation of type passed as the first argument.

  Note: The type `BiFunction<OPERATION, Context, RESULT>` is part of the types to handle the Lambdas in Java 8 and means that it receives a function whose first argument is of type `OPERATION`, the second argument is of type `Context` and the type returned is `RESULT`.

Executors for all types of operations

To create an executor that performs a task regardless of the type of operation, simply extend the `Executor` class and reimplement the `executeAnyOperation` method of this class. Example:

```
public class ExecutorForAllOperations extends Executor {

    @Override
    protected <RESULT, OPERATION extends Operation<RESULT>>
        RESULT executeAnyOperation(OPERATION operation, Context context)
    {
        // Logic to be executed before the execution of the operation
        RESULT result = next.execute(operation, context);
        // Logic to be executed after the execution of the operation
        return result;
    }

    // Constructors ...
}
```

In this type of executors usually end up by calling the `execute` method of the `next` property (of the base class) or the base implementation of the overwritten method to continue the execution of the operation. It is usually used in tasks such as:

- Writing in the log
- Opening and closing of the transaction
- In combination with the use of interfaces or base classes; it is possible to verify if the operation implements a type and, based on this, perform a logic

Executors for a specific type of operation

To create an executor that performs a task for a particular type of operation, simply extend the `Executor` class and implement a method for each operation that you want to execute, where the first argument is the operation, the second is the context, and the return is the result of the operation.

Once the operations are implemented, you must choose what constructor of the base class you want to overwrite, depending on whether the current implementation needs to receive a next executor or not. Within the implemented constructor, it is necessary to indicate which method is responsible for executing an operation, for this we call the `handle` method (of the base class) that receives as arguments the class of the operation and the method in charge of executing the logic associated with the operation.

For example, if you want to create an executor that implements the `SelectMomenEvents` operation, the class would be similar to:

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

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

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

And if you would additionally like to implement the `SelectEventById` operation, the class would be similar to:

```
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);
    }
}
```

Executing an operation

To execute an operation, you must invoke the `execute` method of the executor where the operation is going to be executed. This method receives two generic arguments with the type of the operation and the type of the result of the operation, and two arguments with the operation and the context. Example:

```
SelectMomentEvents op = new SelectMomentEvents ( [...] );
// Some other initialization required by the operation
List<Event> result = executor.execute(op, context);
```

Exception handling

If an exception occurs during the execution of an operation, the framework automatically collects the execution entry so that it can be subsequently registered by some log. For this, a new exception is created that internally includes the unhandled exception, the operation and the context, so that, you have all the necessary information to reproduce the problem.

Public exceptions

If you want to have your own exceptions that are not altered by the framework, it is necessary that these exceptions extend from `PublicException`, whose definition is the following:

```
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);
    }

}
```

Internal exceptions

Any exception that doesn't extend from `PublicException` will be treated as an internal exception, which represents an abnormal functioning of the system, for which the necessary information will be collected in order to reproduce the problem easily.

Information collected when an exception occurs:

- **Exception** occurred during the execution of the operation
- **Operation** that caused the exception
- **Context** used during the execution of the operation

In order to collect the execution information, what is done is to raise an exception of type `OperationExecutionException`, for which the definition is:

```
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) { [...] }

}
```

As the internal exception goes through the different executors that make up the execution sequence, it is checked if the operation and context contained in the exception are the same as those used for the execution (using the `sameContent` method). If the operation or context are different, a new internal exception is created with the previous internal exception and the operation and context objects that correspond to the execution at that point. In that way if, for example, the implementation of an operation has been launched by another operation, and

the latter fails, the resulting exception at the beginning of the execution of the first exception, will contain the information of both operations.

Note: It may be useful to modify the implementation of the `createMessage` method in this exception, so that, during generation of the exception message, instead of writing the operation's `toString` and context, write the name of the class and JSON of each of them, thus avoiding the need to have an implementation (and trust that it has been implemented properly) of the `toString` method in each entity, operation and context.

The `createMessage` method generates the default message to be included as the exception message. The `simpleMessage` property contains the simple message of the exception, without any part created automatically by it, and if it does not have its own message, it uses the message of the internal exception. The created message includes the message specified for the operation (or in its absence the one of the internal exception), and then the full name of the operation type is added, the string representation of the operation and the context. In this way, when an error occurs, all the input information of the system that was used to execute the operation is available, thus having all the necessary information to reproduce the error.

The string representation of the operation and the context in the exception message is obtained when creating the instance of the exception, thus ensuring that the information contained there corresponds to the moment in which the exception was initiated. If for some reason the content of the context or the operation changes during the treatment of the exception, the message of this will contain the value that they had at the time of error, thus allowing to recreate with accuracy the situation that caused it.

Execution bus

The base of Uaithne are the operations, and their implementation is made in different executors that are being composed; in such a way the result of the execution of the operation is the sum of the implementations made in the different executors for the operation that were traversed during its execution. All this composition of executors make up the execution bus of the application.

The central idea in the construction of the different executors is to perform the separation of concerns, so that each implementation is dedicated, as far as possible, to a single task.

For example, to register a user in the system, it is required:

- Encrypt the password
- Insert the user in the database
- Send the welcome email
- Insert the audit log
- Write in the application log

Each item of this list corresponds to one concern, which should be implemented separately, each of them in its own executor, and then be composed to give as a result the bus fragment responsible for its execution.

By decomposing the logic in this way, there is a central element that corresponds to the main action, which in this case is the insertion of the user in the database, and another series of elements that complement or condition the main action, wrapping it within their own logic; these are called interceptors, and have a behavior similar to triggers in the database.

Interceptors at some point, usually end up invoking the execute method of another executor, typically the one found in the `next` property (of the base class). Example:

```
public class InterceptEventsExecutor extends Executor {
    public List<Event> selectMomentEvents(SelectMomentEvents operation, Context
context)
    {
        // Logic to be executed before the execution of the operation
        List<Event> result = next.execute(operation, context);
        // Logic to be executed after the execution of the operation
        return result;
    }

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

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

When the execution bus is assembled, the instances of the different executors are linked to build a line of execution of a group of operations, typically grouped by its module. Example:

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

By constructing this section of the bus in this way, it is obtained that in order to execute, for example, the `SelectMomentEvents` operation, the interceptor `InterceptEventsExecutor` is first passed before reaching the implementation of `EventsDatabaseExecutor`, in such a way that the result of the execution of the action is the sum of the logic contained in each of the executors that the operation goes through during its execution.

Forking the execution bus

Sometimes it may be useful to fork the execution flow according to some criteria and delegate the execution of the operation to a particular bus segment. For that purpose:

- you have to create an executor
- make it contain each of the segments in which you can delegate the execution, for that you could have, for example, a property of type `Executor`, to which the reference to the head of the segment must be assigned, and thus one property per possible segment
- implement the execution of the operation, where according to some criteria, it is decided in which executor to delegate the execution of the operation

The most common forks is the one that allows to join different execution bus fragments, have risen after the separation of the application in different modules, in a single execution bus that acts as a gateway of all the operations that the system executes. The `MappedExecutor` class is offered to handle this type of situation.

The `MappedExecutor` class implements an executor that allows to delegate the execution of an operation to another executor, depending on its type, and if no executor is registered for that type of operation, the execution is delegated to the executor that has been specified in the property `next`.

The definition of the `MappedExecutor` class is the following:

```
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) { [...] }
}
```

The `MappedExecutor` class contains, in addition to the elements already provided by the `Executor` class, the following elements:

- **`void handle(Class<OPERATION> operationType, Executor executor)`**: It allows to register the executor passed as the second argument as the one to be executed for the operation of type passed as the first argument.

- **`void handle(Executor executor)`**: This method traverses an execution bus segment, moving through the `next` property of the executors, and adds for each one of the detected operations its corresponding entry that indicates that the execution of the detected operation is delegated to the last executor by argument.

  In the process of searching for the operations implemented in the execution bus segment, for each executor that composes it, it obtains all the operations handled by it and registers for each of the operations detected, the executor passed by argument as the manager to execute the operation.

  To travel through the execution bus segment, the `handle` method detects each executor that composes it, and detects the operations implemented by it, and once it has finished with that executor it advances to the next one, which is in the next property and repeats the process, stops when next is null, `this` or `this.next`. In case of encountering a `MappedExecutor`, the sequence of advance is the same, that is, only advances by the executor that is in the `next` property.

Of all the methods offered by the `MappedExecutor` class, probably the most useful is `handle (Executor executor)`, which is the one that allows to add in a comfortable way all the operations implemented in an execution bus segment, thus allowing grouping several execution bus segments in a single executor.

By example, when you divide the database access layer in different modules, different execution bus segments appear, which must be grouped into a single executor that represents this layer, for which you could do something like the following:

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

By doing this, the executor `databaseExecutor` is able to execute all the operations implemented in `CalendarsDatabaseExecutor`, `InterceptEventsExecutor` and in `EventsDatabaseExecutor`.

Layers of an application

In a typical configuration for an information system, the system contains at least the following clearly defined layers:

- **Data access**: In this layer all the necessary implementations are placed to access the data (for example from the database), some of them may have some interceptors to complement the main action.

- **Backend**: In this layer, all the implementations that represent actions in the system not centered in the database are placed (although they could make use of it), by example, user authentication management; some of these implementations may have some interceptors to complement the main action.

- **Frontend**: Optional. In many circumstances it is required to group several operations in one, thus avoiding having to chain requests to the server, which destroys the performance of the application, or because you want to ensure that everything is executed within the same transaction. This is the layer in which all the implementations of these operations are placed. In the implementation, the logic of this operations are broken down into other actions in the system; but these operations never represent a real action in the system by itself, but rather it is a mere grouping of operations for reasons of convenience.

  It is very important to take into account that if requests are made to the server, whose response will be used to request another request from the server, this type of sequential chaining of actions is usually harmful to the performance of the application, and as far as possible it should be avoided. So in case you need this type of chain of requests, it is best to group them into one, and for this, a convenience operation is created whose implementation does not do more than execute each of the required operations, and returns an entity that usually contains one property with the result per each of the operations executed.

  Another easy case to forget, is the scope of the transactionality of the system; if a series of requests is invoked from an external system (for example, the browser), each request has its own transaction. If it is desired that several requests share a single transaction, it is necessary to convert them into a single request following the same scheme described above.

Additionally, some interceptors that do not distinguish the type of operation are usually added:

- **Database transaction scope management**: this interceptor assigns the connection to the database to be executed in the following layers and controls the transaction, so that all the actions executed after it in the database they are in the same transaction, which will `rollback` if the result of the execution of the operation ends in an exception, otherwise a `commit` is made.

- **Writing in the application log**: this interceptor writes in the log of the application any error that occurs during the execution of an operation, for it detects if an exception has occurred during the execution of this, and if so, writes it in the application log. This interceptor, typically, when it is in the development environment, writes additionally the input operation and context and the result of the operation.

- **Data validation**: Optional. If a validation framework is used, this interceptor allows to validate that the data coming from external systems complies with all the validations placed on operations and entities, and if it doesn't comply with them, it throws an exception indicating that the supplied data do not meet the requirements.

Access to the bus

The construction of the bus is done in a static way for the entire application, in a singleton, in such a way that it is built only once for the whole application during the initialization of it. The class that allows access to the bus to execute operations is usually called "`Bus`" and usually contains two static properties that serve as entry points:

- **`Executor getServiceBus()`**: Returns the execution bus to be used by services exposed by the application for consumption by external applications, for example, the browser.

- **`Executor getBus()`**: Returns the execution bus to be used internally by the application.

This separation allows to have certain differences depending on which entry point is invoked, for example, it can be done that for the case of external services which have to go through the validation interceptor of the data, but not for the internal bus.

Bus example

The `Bus` class contains the bus initialization and access logic, and the `createBus` method is responsible for creating the execution bus, which is executed only once during the initialization of the application. This class could look like this:

```
// 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);
    }
}
```

It should be noted that the instances of each executor are created only once and are reused to execute different operations in different execution threads, so it is important that internally they don't store the state of the application. Consequently, any field or property that exists within the executors should be to store the configuration of the execution bus, and not to store information relating to the execution of a particular operation.

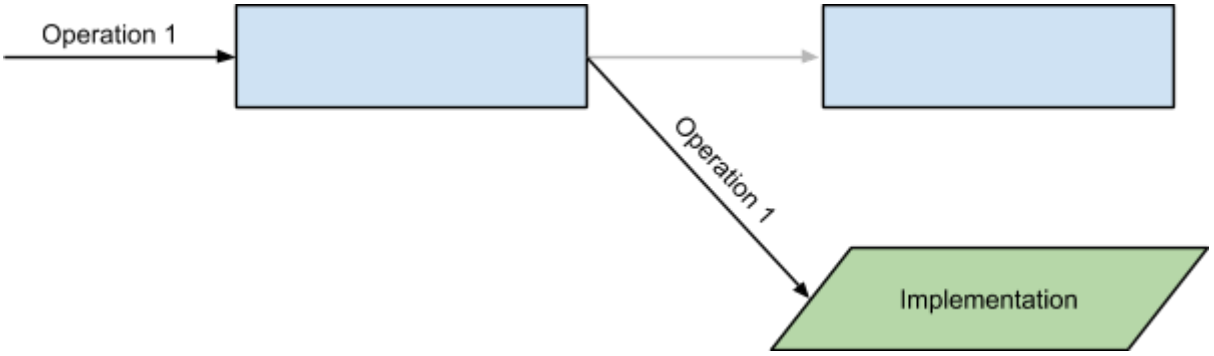
If for some reason, which should be avoided as far as possible, and that must be reserved for very advanced uses, it is necessary to create a variable that contains information specific to the execution of an operation and that must be shared by several executors. It is recommended to use a field marked as local to the execution thread (see the Java `ThreadLocal` class), taking into account that its assignment and de-assignment must be controlled in all cases (it may be helpful to use an interceptor to control the normal output as well as the exit in case of exception).

Patterns to weave the execution bus

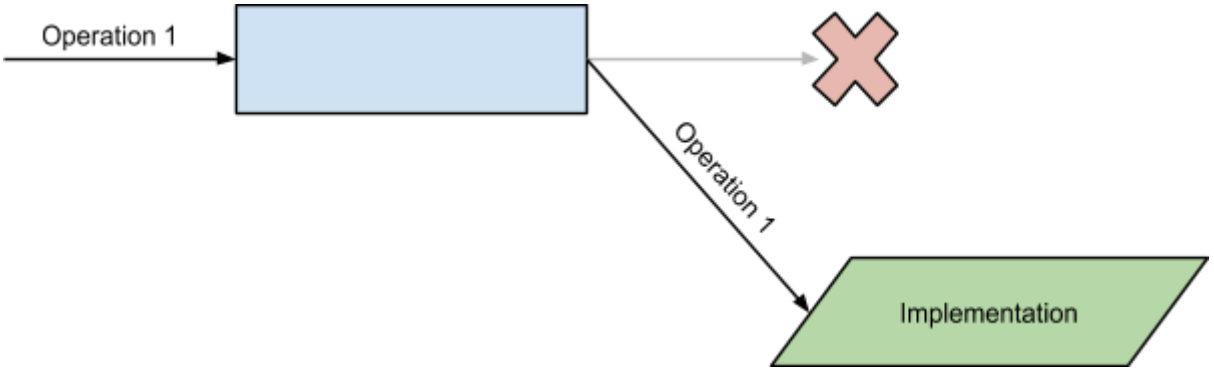
When performing the configuration of the execution bus, a series of typical patterns arise depending on the desired behavior, and can be combined to achieve many more complex behaviors.

Implement

This is the simplest case, in which the implementation module is added to the execution bus without it being dependent on any other module. This pattern is used to add the logic of the module, which can be complemented in the outer layers.



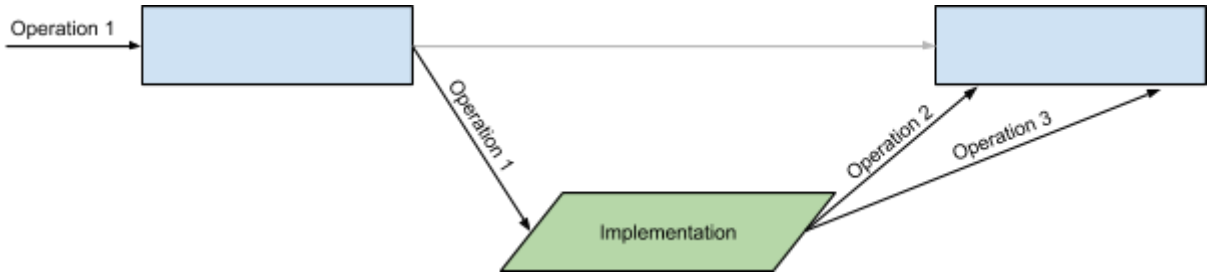
Execution bus section using MappedExecutor with next non-null



Execution bus section using MappedExecutor with next null

Implement and use

In this case, the implementation of a module initiates the execution of operations of other modules that are implemented in a more internal level of the bus. At the end of the process, the result of the first operation executed depends on the result of the following operations without the invoker having knowledge of this. This pattern is used to add the logic of the module in which there is at least one operation in which, in order to carry out its execution, other operations must be performed to achieve its result; the logic of the module can be complemented in the outermost layers.



Execution bus section using MappedExecutor with next non-null

Intercept

An interceptor is an element that gets in the flow of execution of the operation allowing to modify its behavior; for example, it can be said that a database trigger acts as an interceptor of the operations performed on a table.

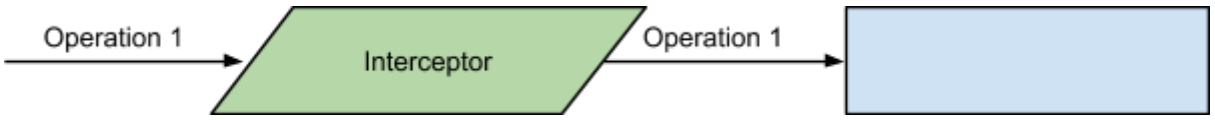
The objective of an interceptor is to complement the logic of execution of an operation, being able to separate the logic of the operation in different competences; for example, the logic to write in the application log inside an interceptor, the logic to check if the user has permission to execute the requested operation inside another interceptor, and finally the logic to access to the database inside the implementation. In this form, the result of the execution of the operation is the sum of the three parts.

Separating the logic into different competences allows improving the maintainability of the code and its readability, as well as facilitating its reuse. By separating each competence in a different piece, you can achieve code pieces with a high cohesion and a low coupling, greatly facilitating its modification of the system, and when they are separated, it is also possible to reorder them, add more interceptors between two existing ones, etc. Thus, allowing to restructure the execution bus completely without this entailing a great effort.

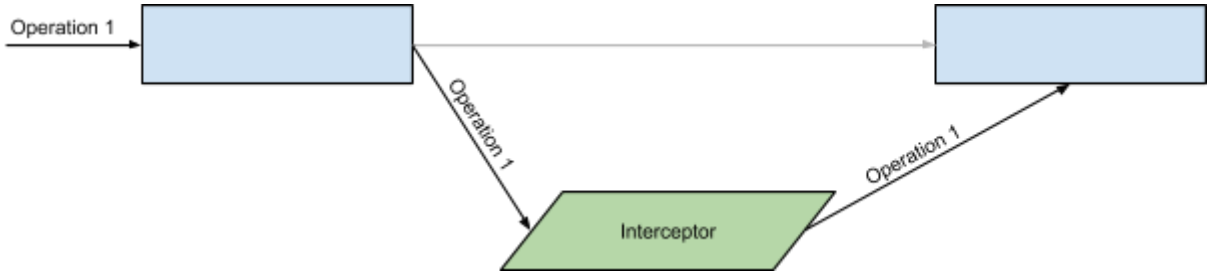
Depending on the behavior of the interceptor, it can be classified as: observer, modifier, controller, impersonator and a combination of the previous ones.

Observer

In this case, the interceptor behaves as a mere observer of the operation that crosses this interceptor, without making any change to the operation or in its execution flow; for example, an interceptor that writes in the application log the operation and its result.



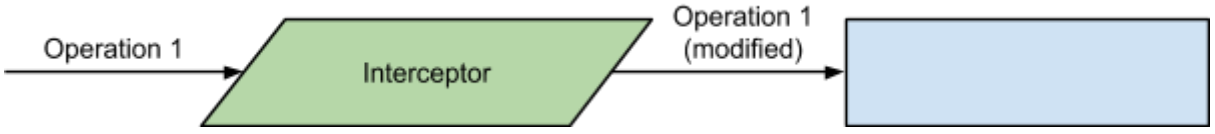
Interceptor interposed directly before implementation



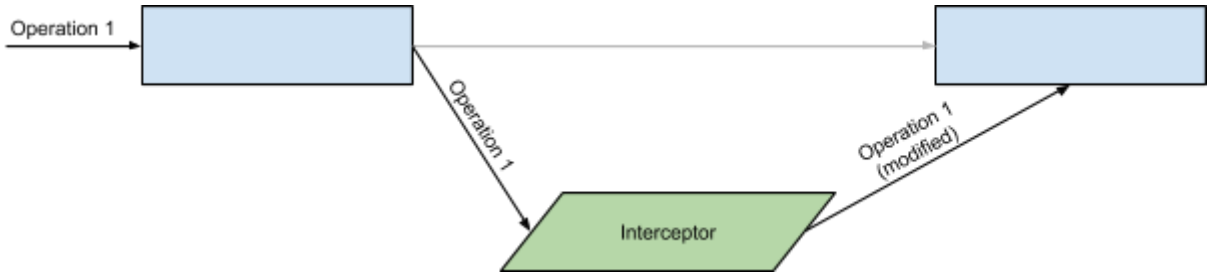
Interceptor interposed by using a branch of the execution bus using MappedExecutor with next non-null

Modifier

In this case, the interceptor modifies the operation or its result. This type of interceptors can be used to complete the information that is needed to execute the operation but that is not provided by the person requesting its execution; for example, adding the user's information who requests the execution of the operation, taking it from the session. You can also use this type of interceptors to prepare the information of the operation that needs to be treated before its execution; for example, encrypting the password of the user before checking if it can log in the application.



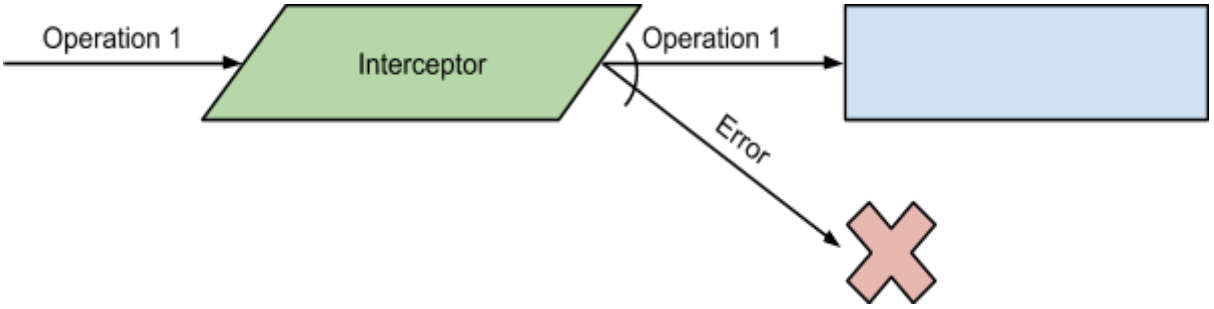
Interceptor interposed directly before implementation



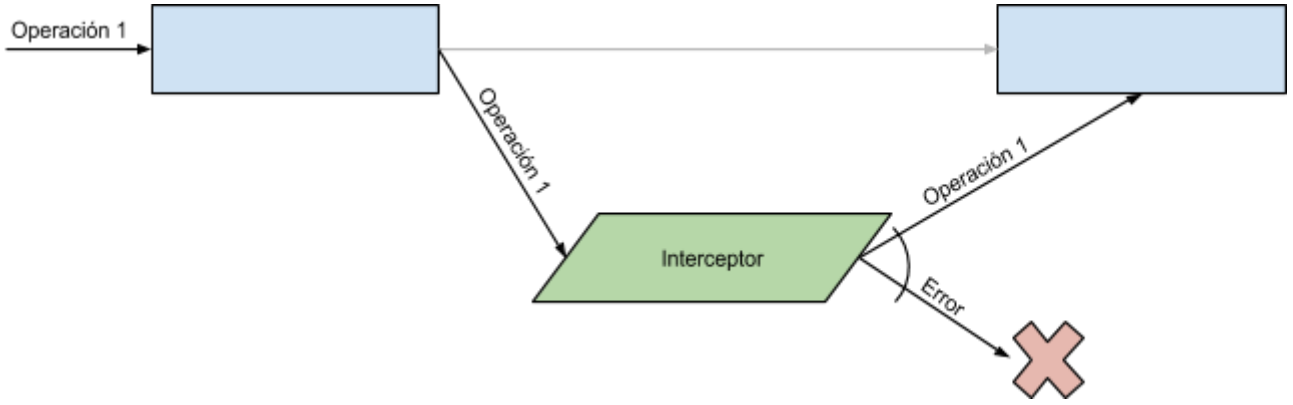
Interceptor interposed by using a branch of the execution bus using MappedExecutor with next non-null

Controller

In this case, the interceptor decides if the operation should continue its execution, or otherwise stop it, being able to throw an exception (or returning a defined value, depending on the desired behavior). For example, you can use this type of interceptors to ensure that only those users who have permission can execute an operation, and in case of not having it, throw an exception.



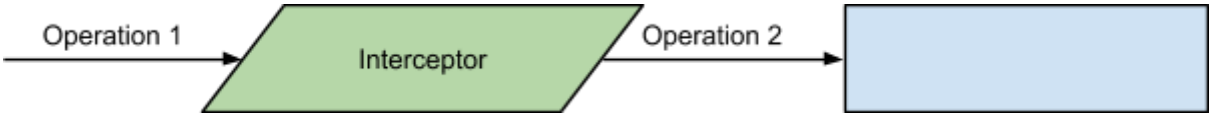
Interceptor interposed directly before implementation



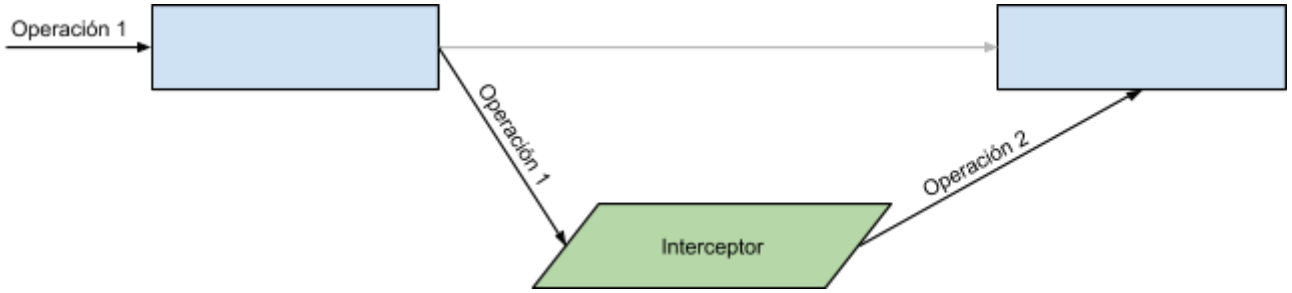
Interceptor interposed by using a branch of the execution bus using MappedExecutor with next non-null

Impersonator

In this case, the executed interceptor transforms the operation into a second operation that does the task that must have been executed by the first operation, giving the impression that the first operation has been executed. This type of interceptor is useful if, for example, you want to maintain compatibility with an operation that is no longer part of the system, so you have to translate it into a new operation that the system is able to understand.



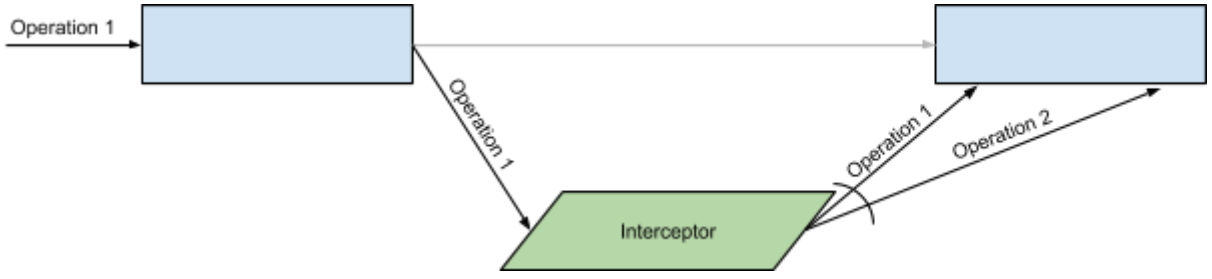
Interceptor interposed directly before implementation



Interceptor interposed by using a branch of the execution bus using MappedExecutor with next non-null

Combination

An interceptor can combine any of the behaviors indicated above, so it is possible, for example, to have an interceptor that acts as an impersonator-controller, that under certain circumstances allows the operation to be executed and in others to supplant it by another one; an interceptor controller-impersonator could be useful, for example, in circumstances in which data is normally extracted from a source (the database of the application, for example) but in certain cases must be extracted from an alternate source (if it was not found in the database, for example).



Interceptor interposed by using a branch of the execution bus using MappedExecutor with next non-null

Typical operations of access to the database

Any action that can be performed on a system can be expressed as an operation, but when interacting with the database, a series of common operations arise, which although changing its parameters, are repeated as a pattern:

Query operations:

- **`Select One`**: This operation returns the entity whose fields match the criteria indicated in the operation. A variant of this type of operation is one that returns only the entity that represents the first record found.
- **`Select Many`**: This operation returns a list of entities whose fields match the criteria indicated in the operation. A variant of this type of operation is one in which only the different registers are brought back, using the `distinct` clause in the SQL query.
- **`Select Page`**: This operation returns a list of entities as a pages view whose fields match the criteria indicated in the operation. A variant of this type of operation is one in which only the different registers are brought back, using the `distinct` clause in the SQL query.

  Select Page operations are the most complex query operations, and at the same time quite useful when building the interface, so its consideration from the beginning is highly recommended.

  Select Page operations must receive at least the following values:

  - **`limit`**: indicates the number of records to be obtained in a data page, that is, it is the maximum size of the page. If this field is null, it indicates that it has no limit, so the maximum number of possible records must be brought back.
  - **`offset`**: indicates the number of records to be ignored prior to the data page that you wish to consult. The offset can be considered as the index (including if you consider that the indexes begins in zero) from which you will bring records. **Example**: If you have the size of the page as 20 records, and you want to obtain the third page, the value of this field should be (3 - 1) * 20, which is 40 records before being ignored, since you will get the records 41 to 60 (if you consider that the indexes start at one). If this field is null, it indicates that no record will be ignored.

  The result of the operation must be an object that contains the following values:

  - **`limit`**: contains the `limit` used in the operation that returned this object and represents the maximum size of the page.
  - **`offset`**: contains the `offset` used in the operation that returned this object and represents the number of previous records that have been ignored.
  - **`dataCount`**: contains the total number of records; if the operation specified one value, the value of this field is the one indicated in the operation; but, if not, the value is the count of the total number of existing records without

paging.
    - **`data`**: contains the list of the records that belong to the requested page.

  When implementing a Select Page operation, it is normal to have to perform two queries to the database, one to bring back the records of the data page consulted and the other to get the number of existing records.

  It is usually useful, to give the possibility to consult the existing number of records without consulting a data page or to allow a way of not having to recount the number of existing records; for this you can add the following properties to the operation:

    - **`dataCount`**: indicates the total number of records. If you specify the value of this field in the operation, it causes that the operation not to have to consult the total number of records, instead it is assumed that the value is the one indicated here, and this is useful to avoid the additional query that counts the records.
    - **`onlyDataCount`**: if `true` is specified in this field it causes the operation to only check the total amount of the record, without bringing the data of any of the pages.

- **`Select Count`**: This operation returns the number of entities whose fields match the criteria indicated in the operation. A variant of this type of operation is one in which only the different registers are brought back, using the `distinct` clause in the SQL query.

- **`Select Entity By Id`**: This operation returns the entity whose id is the one indicated in the operation. A variant of this type of operation is one that returns only the entity that represents the first record found.

Entity modification operations:

- **`Insert Entity`**: This operation inserts a record with the values indicated in the entity contained by the operation. The operation returns as result, if desired and possible, the id of the inserted record. This behaviour is recommended, even though it is not used since it allows the operation to be reused in more places or to follow up on the logs.

- **`Update Entity`**: This operation updates a record with the values indicated in the entity contained by the operation; the record to be updated is the one that matches the id indicated in the entity.

- **`Delete Entity By Id`**: This operation deletes a record whose id is the one indicated in the operation.

- **`Save Entity`**: This operation inserts or updates a record with the values indicated in the entity contained by the operation; the record to be updated is the one that matches the id indicated in the entity, if it is not inserted. The operation returns as result, if desired and possible, the id of the inserted record. This behaviour is recommended, even though it is not used since it allows the operation to be reused

in more places or to follow up on the logs.

- **`Merge Entity`**: This operation updates a record with the values indicated in the entity contained by the operation whose values are different of null; the record to be updated is the one that matches the id indicated in the entity.

Data modification operations not restricted to entities:

- **`Insert`**: This operation inserts a record with the values indicated in the operation. The operation returns as result, if desired and possible, the id of the inserted record. This behavior is recommended, even though it is not used since it allows the operation to be reused in more places or to follow up on the logs.
- **`Update`**: This operation updates a record with the values indicated in the operation. It must be kept into consideration that in this type of operation, the fields of the operation can be used as part of the `where` of the query, or as values to be updated in the database, and in some cases, if desired, be omitted if the supplied value is null, thus leaving the current value in the database unchanged.
- **`Delete`**: This operation eliminates the records with the values indicated by the operation, whose criteria is defined in the implementation of this operation.

It should be noted that the operations indicated here are usually typical of database, but it is always possible to create more operations that escape this typology. This often happens in the backend of the application, where an operation could be "Send notice of collection". **The important thing is that the operation always maintains its meaning**, so it would be incorrect to use an invoice insertion operation in the invoice table of the database (of the module that represents the invoice table) as invoice issuance operation. This should be a different operation that should be in the billing module located in the backend layer.

Calls to stored procedures

Calls to stored procedures in the database can be expressed as an operation, preferably according to one of the typical types of operations discussed above, taking into account that any output that has the procedure must be returned as part of the result of the operation and never modifying the object of the operation itself. This is of special attention in the procedures with input and output parameters, where the input must be read from the operation, but the output must be written to the result (never in the operation or in the context).

In the invocation of stored procedures, as in any other action, the result of the execution must always be read from the object resulting from the operation execution. In the operation or in the context, under no circumstances, the result of the execution or any intermediate state should be stored.

Modeling the data

Entity views

In many circumstances, when consulting the database, you don’t want to extract an entire record, instead you want to extract some of that information, or even that information can be combined with information from another table; in those cases an entity can be created that only contains the desired information. This type of entity will be called an "entity view" and its construction is identical to that of an entity with the difference that it does not represent an entire record stored in a table.

Object-oriented modeling of data

The data of the database must be modeled in objects as they are in their origin, and in no case should transformations be applied to try to build an object-oriented adaptation.

This point especially affects the handling of relationships. The relationships in the database must be represented in the same way they are there; for example, if a record has the identifier of another record, when creating the entity that represents it, this must be maintained as is, the id of the other record will be included instead of having a reference to another object (or in other cases having a reference to a collection of objects).

The objective is that the entities represent the data as they are in the database, without making any tricks that alter the existing relational modeling.

Grouping results

If you want to perform an operation that returns several results, for example, a record of the database and its relationships, you must create an entity that will contain each of the records consulted in the database, so that they do not suffer alteration with respect to the query made to the database.

Primary key

The use of primary keys composed of a single field is recommended, whenever possible (it is not worth to artificially create a primary key of a single field), since this greatly facilitates its use in the upper layers, especially in the construction of the Interface.

The use of composed keys is usually very useful in the database, and these can be of two natures:

- **All the members of the primary key are also foreign keys**: in this case there is no other option than to handle all the members of the primary key in the operations. This situation is typical in the tables that control a relation many to many, where the table does not represent any concept in itself. Trying to create a unique identifier here does not make sense. In addition, it would only add complexity to the project,

especially in the interface, so the use of these artificial primary keys is discouraged.

- **Some member of the primary key is unique to the table**: this is the case in which one or several foreign keys are part of the primary key, and additionally a field that is specific to the table (it is not foreign). In this case, it is recommended to try to convert this field into an alternate key (marking it as unique in the database) and using it outside the database as if it were the primary key of the table. If it is not possible to convert this field into an alternate key, there is no other option than to handle all the members of the primary key in the operations, as explained in the previous point.

  An example of this situation **is the invoice line table**, **the primary key** of this table is usually **the identifier of the invoice**, which is the foreign key of the invoice table, **and an identifier of the invoice line**. To generate the identifier of the line of the invoice there are usually two options:

  - **The identifier of the line of the invoice is unique in the table**: and can be generated by a self-incremental field or a sequence, which is the desired situation since this field can be converted into an alternate key (marking it as unique in the database) and treat it as the primary key in the application.

  - **The invoice line identifier is unique within the invoice**: this value may be, for example, the number of the line in the invoice. These type of values are usually complex to manage, especially in the interface, and require a complex logic to be generated or maintained; for example, if an intermediate line is eliminated, so its use is discouraged, the desirable thing here would be to be able to transform this situation to the one exposed in the previous point, where the identifier of the line of the invoice is unique in the table. If not possible, there is no other option than to handle inside the operations all the members of the primary key and add the logic for the creation and maintenance of this identifier.

When creating operations, it is recommended to have preference for the keys composed of a single field on which they are composed of more than one field, without this being translated in trying to avoid the use of compound keys in the database, which are usually great utility, or the creation of artificial identifiers in some tables. The primary key used in the operations does not necessarily have to be the primary key of the database, an alternate key can perfectly identify the record unequivocally.

Execution permissions

The management of the security of the application is done by controlling the operations that the user can execute, without evaluating in any case the content of the operation. To achieve this, it may be necessary, in some cases, to have the same operation duplicated, one contextualized or another more general, or one that admits a series of parameters and others that additionally admit some more.

The definition of the operation must be consistent with the permits required, which must be invariable according to the content of the operation; if the required permits vary depending on the content, this operation must be divided until each operation has its own permits without depending on the content, even if that means having similar operations.

Optional fields in operations

The use of optional fields in the operations, that is, fields that support nulls, is a great help, and avoid having to create many operations that do the same, supersets of the previous one, where the next version allows adding one more field with respect to the previous one. The optional fields are used, for example, to have a operation that query to the database, which allows to have more demanding filtering criteria to the extent that the value is specified to those fields, that is, if the field has a value, the condition must be included in the `where` clause of the query.

The use of optional fields in the operations must be clearly indicated in its definition, by means of an attribute (of having automatic validations) or a comment, in such a way that the person who sees the definition of the operation can easily know which fields are optional without having to search the implementation.

The use of optional fields must respect the meaning of the action to be performed by the operation, so it is incorrect to have the same operation whose nature changes depending on the presence or absence of a value.

If you use optional fields whose impact goes beyond the conditions of the `where` clause of the query, you have to evaluate its impact in terms of security. The permission of the application is based on the operations that can be executed by a user. The content of an operation must not be valued or questioned; so, if the presence of an optional field causes the operation to have two different permit levels (depending on the presence or absence of the value), it is highly advisable to divide the operation into two, in such a way that the principle that the execution permits of the operation do not depend on its content is respected.

Data validation

The use of a validation framework is recommended, which allows, through the use of annotations (preferably) that are added to the fields or properties of the entities and operations, to validate the data, in such a way that the rules expressed there contain all the restrictions, as far as the data refer, that exist in the database.

The idea with this type of framework is to complement the definition of the classes, by using attributes, which add restrictions to the fields; for example, in a property of type `int`, you can specify the range of valid values, or the length maximum of a string.

For example, the definition of the `Calendar` entity, when adding the validation attributes could look like:

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

    @NotNull
    @Length(50)
    private String title;

    @Length(200)
    private String description;

    // The @Data annotation automatically generates:
    // getters, setters, equals, hashCode, toString

    public Calendar() { }
    // Other constructors...
}
```

This type of framework usually allows to indicate the following type of validations:

- Mandatory or optional field, allowing or not allowing null values
- Range of valid values in a number
- Minimum and maximum length of a string
- Validate string using regular expressions, for example, to validate an email or a URL
- Validate a date be from the past
- Validate a date be from the future
- And more...

The frameworks of this type in Java are usually based on Java Beans Validations, the standard Java API for validations, and a de facto standard for this purpose, supported by practically all Java frameworks. Hibernate Validations is an implementation of Java Beans Validations (there are others, in order to use the API it is necessary to do it through an implementation). Web page: http://hibernate.org/validator/

The use of a validation framework is highly recommended, especially if its capacity is combined with the interface, so that you can take advantage of this information without having to manually program the validation.

By using a validation framework, it is possible to write an interceptor that validates all the operations that come from the interface on the server, in such a way that it ensures that all operations received by the interceptor meet the criteria before performing any other action.

Ordering

Sometimes it is useful to have operations that receive the `order by` to be used in the query; for this you can use a `string` type field that receives the ordering criterion.

The sort criterion is nothing more than a string that contains:

- Name of the field for which you are going to order.
- Optionally it can have the `order by` direction, to be: `asc` or `desc`.
- This can be repeated as many times as you want, separating it with a comma.

Example:

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

It is not recommended that the content of this field be passed directly to the database, but must be previously transformed and translated into the code.

Rules that must be taken into account in the ordering criteria:

- The name of the field must match the name of the property in the resulting entity, so that the value indicated here doesn't depend on the construction of the query. The idea here is that the invoker of the operation doesn't have to know the structure of the database or how the query is built.

- There is a known list with valid fields to order. If anything that does not comply with the format or an unknown field is included, an error must be raised, thus preventing the execution of the query. The idea here is to prevent any risk of SQL injection in this SQL fragment.

To translate ordering rule, you must:

- Normalize the upper and lower case, being able, for example, transforming everything to lowercase.
- Replace any occurrence of several blank spaces with one.
- Separate by commas, taking into account that, optionally, the comma may contain blank spaces before and after.
- Remove any blank space that is at the beginning or end of the fragment.
- For each resulting element, look for its translation to its equivalent in the database, bearing in mind that it has three variants:
    - *`field_name`* which translates to *`column_name`*
    - *`field_name`* `asc` which translates to *`column_name`* `asc`
    - *`field_name`* `desc` which translates to *`column_name`* `desc`
    - If it is not contained in any of these three variants, an exception is thrown, thus denying the execution of the operation.

- Once all the terms are translated into their database equivalent, they are put together in a single `string` separated by a comma.
- The resulting `string` is the one that can be used in the query.

The fact that the operation receives the order clause in this way allows:

- to make changes in the query without having to modify the code of the invocantes of the operation.
- to protect yourself from SQL injections, since typically this part of the query is not passed as a parameter of the database, but it is concatenated to the query itself.
- to facilitate the use of the operation, since the invoker doesn't have to know how the query is constructed, it only knows the name of the fields resulting from the operation, and with that can build the `order by` clause.
- to maintain the database logic within the database access layer, since in this way the query is not seen scattered along multiple layers.

The fact that the operation doesn't receive the `string` to be used directly in the query is highly recommended. It is strongly discouraged to receive in this string the `order by` clause as required by the database.

Implementation of access to the database

To implement access to the database there is no special requirement, so you can use any framework that you want. However, it is recommended to model the data as it is at its origin, and to avoid any attempt to transform the data in the database into an object-oriented model.

The benefits of modeling the database information in an object-oriented model are usually few when dealing with services that serialize the data by, for example, the service that sends the data to the web application, but the additional complications are many, such as managing the discrepancy between the two worlds, controlling the deferred loads of the data and its handling during serialization, more complex operations, data redundancy, etc. For these reasons and more, it is recommended not to try to model the database with an object-oriented model unless you have a good reason for that.

You can use any database access framework that you want, but to make an appropriate selection it is necessary to understand the kind of life the database has in relation to the project:

- **Database-centered design**: If you have an application whose data design is centered in the database, in which it is feasible to modify the database every time the application requires it, it is most appropriate to use a framework that allows access to the database as it is, and allows to validate the queries made in the application during the compilation. Some frameworks in this category are:
    - **jOOQ**: Very long trajectory framework that allows, starting from the database, to generate all the access code to it, so that the queries in Java can be written in the Linq style of .Net, where the queries are validated by the Java compiler,

instead of being a mere string. This framework is free for free databases and paid for not free databases. Web page: http://www.jooq.org

    - **Querydsl**: Framework similar to jOOQ but completely free software. In several aspects easier than jOOQ and a little less powerful, but with less documentation. It can be a very good option if you do not want to use jOOQ. This framework offers the possibility to write the query to generate the SQL directly (recommended) or write queries that generate the query JPA or JDO (not recommended as it adds more complexity without greater benefit). Web page: http://www.querydsl.com

- **Discrepancies between the database and the system**: If it is necessary to tolerate discrepancies between the database and the application; for example, when the design of the database is not well done, the database is difficult to change, etc.; it is best to use a framework that separates both worlds, which allows queries to be represented in string (we must be careful with SQL injection attacks). Some frameworks that can be used in this situation are:

    - **Apache DbUtils**: Very simple and very powerful framework to access the database, especially if you take advantage of the capabilities of the `BeanHandler` class. Do not confuse yourself, its simplicity does not limit its power. Web page: http://commons.apache.org/proper/commons-dbutils/

    - **MyBatis**: More complex database access framework, with some added features, and some complexities too, whose distinctive feature is that the queries are written in an XML file separated from the logic of the application and that it has added facilities for writing dynamic queries without major complications. Through the use of the `Result Maps` it offers powerful capabilities to adapt the queries result in Java objects, although the simplest, and least required code, is not to use them, and instead have the queries return columns whose name corresponds to the property in the Java object where they should be assigned. Web page: http://www.mybatis.org/mybatis-3/

In order to maintain the system in the long term, it is best to use JOOQ or Querydsl frameworks, which allow to detect, with a simple compilation, any change in the database that is incompatible with a part of the application.

It is perfectly possible to use JPA or JDO frameworks, such as Hibernate, although the benefit of its use must be assessed, since the use of its own query language adds extra complexity, a series of limitations, and a quantity of additional knowledge whose cost does not compensate your benefit. If what worries is to be able to use different databases without having to completely rewrite queries, SQL frameworks like jOOQ or Querydsl allow to easily change the database, and easily detect those queries that use very specific constructions, and impossible to emulate in another database, which need to be rewritten.

Implementation of REST or Web services

To implement web services or REST services you can use the standard frameworks of the JDK and in its implementation create an instance of the operation and send it to execute on the Bus.

The standard frameworks of the JDK, and that are widely used in other frameworks, are:

- **JAX-WS** for web services
- **JAX-RS** for REST services

It can also be useful, especially to handle AJAX requests from the browser, a service that receives the JSON from the operations and responds the JSON of the result of it, thus avoiding all the additional code of having to expose other types of services (such as REST services) to use from the browser. The code required to implement this service, and the entire security policy around it, is included below.

User interface implementation

To structure the backend of the application in operations is compatible with most user interface frameworks, so you can use the framework that most like.

Currently the most common at the time of making the interface of a system is a web page, which follows the philosophy Single Page Application, where all the logic necessary to build the interface is executed in the browser. In this type of interfaces, the server exposes one or several services to be consumed by the interface through AJAX, and in this case it may be convenient, if there is not a large separation between the backend and the interface, to have a service that receives the JSON of the operation to be executed and return the result in JSON format. The code required to implement this service, and the entire security policy around it, is included below.

Other elements

Distributed transactions

The use of distributed transactions is seriously discouraged; its use must be completely justified and there must be no other alternative to achieve the same result.

The use of distributed transactions causes serious performance problems in the database, and in many cases it is convenient to tolerate the risk of a certain degree of inconsistency and manually handle special situations.

Queues

To implement queue handling you can use JMS, which is the standard API of the JDK and used by practically all Java frameworks. One of the most common implementation of this API, and used in many frameworks, is Apache ActiveMQ. Web page: http://activemq.apache.org/

Scheduled tasks

In most cases, to do scheduled tasks or daemons, it is sufficient to use the Java `Timer` class. If the capabilities offered by that API are not enough, Quartz can be used; this framework is the most used beyond the Java Timer class. Web page: http://www.quartz-scheduler.org/

Aspect oriented programming

Through the use of interceptors, all the capabilities of aspect-oriented programming can be obtained in a simple way. All patterns for weaving the execution bus correspond to patterns of object-oriented programming.

Dependency injection

The usage of the inversion of control pattern and dependency injection is usually very attractive and convenient. Taking into account the design proposed here, the use of this type of patterns is not required in the implementation of the executors or operations, since the most critical elements, typically the user's identity, permissions and connection of the database, are already managed by using the execution context; additionally, the separation of layers is guaranteed by the design itself.

In the code included here, to make it simple, the singleton pattern is used to obtain the instance of the bus, which can be replaced and instead use the injection of dependencies if it is considered convenient.

The execution context usually contains other elements that are usually delegated to the dependency injection, so when the context object it created, it must be properly initialized.

Unit tests

Modeling the application of the form proposed here implies that the programmed code is very friendly to the unit tests, without requiring additional complications.

Some useful frameworks for unit tests are:

- **JUnit**: Unitary testing framework. Web page: http://junit.org/junit4/
- **Mokito**: Framework for creation of fake objects to be used in unit tests where it is really necessary. Web page: http://site.mockito.org/

How to do unit tests of the connection to the database will depend on the database framework that is chosen.

Framework code

```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); }
}
```

Useful exceptions

Validations not met

This exception serves to indicate that the validations of an operation or entity were not satisfied.

```
// 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;
    }

}
```

Insufficient privileges

This exception serves to indicate that the user is trying to execute an action for which he does not have permissions.

```
// 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);
    }

}
```

Useful interceptors

SQL Transaction Handling Interceptor

This interceptor allows controlling the scope of the connection and of the SQL transaction, being the one in charge of placing the connection to the database in the context.

If the execution of the operation causes an exception, a rollback of the database is done; if a result is returned, a commit is done.

If there is already a database connection in the context, this interceptor does nothing.

This interceptor is usually placed on the bus before any logic is executed in the application, thus causing all the operations that run after it to be in the same transaction.

```
// 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) {
            /*
             * There is already a database connection, it is not necessary to open another one
             */
            return next.execute(operation, context);
        }

        boolean rollback = true;
        Connection connection = null;
        /*
         * A new context is created, since the changes that are made on this can only be
         * published on the context received by argument when committing, in case of rollback
         * the changes must be discarded
         */
        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();
                        /*
                         * The changes made in the internal context are published in the
                         * context received by argument
                         */
                        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;
    }

}
```

MyBatis transaction management interceptor

If the project uses MyBatis to access the database, instead of having the SQL connection in the context, you are going to have the `SqlSession` object of MyBatis. So, instead of using the `SqlSessionInterceptor` class this is used, whose logic and behavior is analogous of the first one.

This implementation assumes that in the context, instead of the `databaseConnection` property with the JDBC connection to the database, there is the `myBatisSession` property of type `SqlSession` with the MyBatis session.

```
// MyBatisSessionInterceptor.java

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

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) {
        /*
         * There is already a database connection, it is not necessary to open another one
         */
        return next.execute(operation, context);
    }

    boolean rollback = true;
    SqlSession myBatisSession = null;
    /*
     * A new context is created, since the changes that are made on this can only be
     * published on the context received by argument when committing, in case of rollback
     * the changes must be discarded
     */
    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();
                    /*
                     * The changes made in the internal context are published in the
                     * context received by argument
                     */
                    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;
    }
}
```

Detailed writing interceptor in the log

This interceptor allows to write in the requested operations and its result and it does not write the errors since these are usually written in the log in the most external part of the application. This interceptor receives by arguments the logger, in which, a short name for the application is going to be written.

```
// 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 with the data validation of an operation

This interceptor verifies that the operation complies with all the requirements exposed by means of Java Bean Validations annotations, if any, it throws the `ConstraintViolationException` public exception. This interceptor is typically used to validate the entry of operations from external sources such as a JSON service used by JavaScript for AJAX requests and is usually placed even before handling the connection to the database.

```
// 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();
    }
}
```

Useful servlets

Servlet base for context management

The `ManagedHttpServlet` class provides an abstract implementation of a `Servlet` that provides:

- Management and recovery of the context from the user's session.
- Implementation of security mechanism to prevent XSFR attacks.
- Writing in the complete information log when the application fails.
- Errors related to the breakdown of communication with the browser are discarded.
- Handling of HTTP error codes: 419 (expired session), 401 (required login), 403 (insufficient privileges), 412 (unfulfilled preconditions) and 500 (internal error)
- Allows you to send from an AJAX request to the exact URL in which the browser is located, so that, in the event of an error in the server, you can see exactly on which page the AJAX request was generated. This is very useful in Single Web Applications.

Abstract methods:

- **`void process(HttpServletRequest request, HttpServletResponse response, Context context)`**: in this method, the action to be performed by the `Servlet` must be implemented.
- **`void logError(String requestInfo, String referrer, Context context, Exception ex)`**: this method must write the exception that occurred in the application log.

Methods with default implementation:

- **`boolean validateXsrfToken(HttpServletRequest request)`**: method that returns if the XSFR token must be validated. By default, all requests except the GETs are validated.

Methods to be used in the implementation:

- **`void processRequest(HttpServletRequest request, HttpServletResponse response)`**: By default, no request are processed, so it is necessary to overwrite the method `doGet`, `doPost`, etc. of the Servlet for the HTTP methods that you want to attend and in its implementation this method must be invoked.
- **`void saveContext(HttpServletRequest request, HttpServletResponse response, Context context)`**: by default the changes in the context are saved

once the processing of the `processRequest` method is completed, but this has a problem; if operations that can create or destroy the session are invoked, or alter the XSFR security token, this method must be invoked before the body of the page is sent.

```
// 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 {
        /*
         * The X-CURRENT header allows you to send the current page in the browser, especially
         * useful in AJAX requests from the browser in Single Page Application applications where
         * fragment, which is what is after the # symbol, is not included, and that's where it's
         * said really on what page it is.
         *
         * When building the AJAX request, you must manually set the X-CURRENT header, if you
         * want it to be used here to write in the log which is the url that caused the error.
         * The value of this header can be constructed as follows:
         *     '@' + 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 for AJAX requests

This class allows you to create an RPC service for AJAX requests from the browser that allows you to execute operations in JSON format, where the property of name "`type`" contains the name of the operation that you want to execute, and the rest of the properties correspond to the properties of the operation, and this class returns the result of the operation in JSON format.

Example of JSON to be sent to execute the operation:

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

The result of this operation could be:

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

The responses begin with the security prefix "`)]}',\n`" to prevent them from being used in XSRF attacks combined with JSON attacks.

Permission management

In the `JsonServiceExposeOperations` class (which must be implemented) the operations that can be executed for a particular request are exposed, for which the context is consulted if there is an authenticated user or if it has a particular role. The idea behind this control is to publish only those operations for which the user has permission to execute, thus preventing an attacker from executing operations for which the user from the interface would never use.

Example of the `JsonServiceExposeOperations` class

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

public class JsonServiceExposeOperations {

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

        /*
         * Operations available even without being authenticated
         */
```

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

if (context.isUserLoggedIn()) {
    // If it is not authenticated, it does not expose more operations
    return;
}

/*
 * Operations that only need to be authenticated
 */

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

/*
 * Operations are added based on user roles
 */

// Operations for the Calendar table

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);
    [...]
}

// Operations for the Event table

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);
```

```
            [...]
        }
    }

}
```

Implementation of the service

To implement the transformation of Java objects to JSON and vice versa Gson (https://github.com/google/gson) is used in this implementation which indicates the list of operations allowed for the user. Note: this code uses the `RuntimeTypeAdapterFactory` class whose code can be downloaded from https://github.com/google/gson/tree/master/extras/src/main/java/com/google/gson/typeadapters

The handling of the date and time is done ignoring the differences of time zones between the browser and the server, so that the date and time indicated in the javascript correspond to the date and time of the server, as if they were in the same time zone. This behavior is implemented in the `DateTypeAdapter` class and may not be desired in some applications. The `TimeTypeAdapter` class does the same, but for the new date and time classes of the JDK 8, specifically: `LocalDateTime`, `LocalDate`, `LocalTime` and `Instant`, the representation of dates or times with time zones are not included in the current implementation.

Requirements:

- **`Executor getServiceBus()`** as a static method in the **`Bus`** class: this method returns the input executor to the application from the JSON service.
- **`void registerOperations(Context context, RuntimeTypeAdapterFactory<Operation> factory)`** as a static method in the **`JsonServiceExposeOperations`** class: this method initializes the operations that are allowed to be executed in the current request. To publish an operation, you must call the `registerSubtype` method of the `factory` and pass the argument of the type of the operation that you want to publish.

```
// 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);
        }
    }
}
```

Service access from the browser

The `rpc` function allows sending a request to the server with ease. The first argument is the JavaScript object with the operation information and the property "`type`" to be sent to the server. The second parameter is a function that executes on success and receives by argument the JavaScript object with the response of the server. And, the third parameter is a function that is executed in case of failure and receives the argument as the error.

This implementation sends to the server the current URL of the full browser in the `X-CURRENT` header and also sends in the `X-XSRF-TOKEN` header the value of the security token read from the `XSRF-TOKEN` cookie. Additionally, it transforms all the dates and times sent by the server into JavaScript Date objects.

```
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);
}

/*
 * Private functions
 */

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);
        }
    }
}
```

File upload servlet

To implement a Servlet that allows uploading a file you just have to use the capabilities offered by the Java Servlets. If it extends `ManagedHttpServlet`, you can take advantage of the capabilities already included by it.

Note: Do not forget to configure the file upload in the `multipart-config` section of the `web.xml` or use the `MultipartConfig` annotation.

File upload servlet example

```
// 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
    {
        /*
         * Check if the user has permission to perform this action
         */
        if (!context.canUploadFile()) {
            throw new InsufficientPrivilegesException();
        }

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

            /*
             * Process the uploaded file
             */

            [...]
        }
    }

    @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);
    }
}
```

File download servlet

To implement a Servlet that allows you to download a file you just have to use the capabilities offered by the Java Servlets. If you extend `ManagedHttpServlet`, you can take advantage of the capabilities already included by it.

```
// 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
    {
        /*
         * Check if the user has permission to perform this action
         */
        if (!context.canDownloadFile()) {
            throw new InsufficientPrivilegesException();
        }

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

            /*
             * If the file is not found, the 404 code is returned
             */

            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);
    }

}
```

Initialization servlet

If the application contains daemons, scheduled tasks, initialization logic, etc. it is most convenient to use a Java initialization servlet.

Note: Do not forget to include the class in the `listener` section of the `web.xml` or use the `WebListener` annotation.

Example of an initialization servlet that creates a scheduled task

```
// 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) {
        /*
         * Logic to be executed when the application starts
         */
        LOGGER.fine("Application starting...");
        SomeDaemon.startDaemon();
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        /*
         * Logic to be executed when the application stops
         */
        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) {
            /*
             * If the action to be executed by the scheduled task is too long, it can be
             * re-invoked when the previous invocation has not yet finished, so that the
             * new execution is not discarded while the other has not finished.
             */
            return;
        }

        try {
            isRunning = true;
            /*
             * Logic to be executed by the scheduled task
             */
            LOGGER.fine("The daemon is running...");
        } finally {
            isRunning = false;
        }
    }

    public static void startDaemon() {
        timer = new Timer("Some Daemon", true);
        long delay = 60000; // Wait 60 seconds for the first execution
        long period = 30000; // Run every 30 seconds
        timer.scheduleAtFixedRate(new SomeDaemon(), delay, period);
    }

}
```

Ω end Ω

Designed by: **Juan Luis Paz Rojas** - https://github.com/juanluispaz