# Software Requirements Specification

## Sumary

# 1. Introduction

## 1.1 Purpose

The purpose of the product described here is to provide a web API that can handle the logic and data of a very simple to do app via HTTP. This app will be capable of managing request for every CRUD operation: create, read, update and delete. The functions of the app will be based on task, which are activities that must be done, a form of reminders. The user will be able to consult the tasks registered on the server, filtering them by different available criteria, so the user can use this data to his advantage.

This will be an HTTP server callable from any HTTP client that has access to it locally. It will depend on a MariaDB database.

## 1.2 Glossary

The following definitions are terms that will be used throughout the document.

- Task: An activity that the user has registered in the app. It has many possible states: pending, in process, completed, and abandoned. It has also a priority, description, a date of creation, date of desired completion, and a date of completion.
- User: A front-end app that can make HTTP requests and interchange JSON data.
- To-do server: the application being described here.
- The server: the application.
- DB: The database, in this case, a MariaDB database.
- HTTP: The HyperText Transfer Protocol that the server will use to communicate with the user.
- API: Stands for Application Programming Interface.
- JSON: Stands for JavaScript Object Notation.
- URI: Stands for Universal Resource Identfier.
- REST: Stands for REpresentational State Transfer.

# 2. Overall description

## 2.1 User needs

The users of this app are front-end applications that need a simple server to store and manage data related to a to-do list.

The needs that the tasks will cover for a possible end user are the following:

- Be reminded of pending activities that must be done
- Set a time limit for the completion of the task
- Save a history of the tasks registered, so it can be consulted after
- Set a priority level for each task

## 2.2 Assumptions and Dependencies

It is assumed that the user can make requests over HTTP and interchange information in JSON format. Also, no sensible information will be implied because of a lack of authentication in the application.

# 3. System Features and Requirements

## 3. 1 Functional requirements

- HTTP communication: Receive and send requests and responses over the HTTP protocol.
- JSON communication: The server will send and receive data through the JSON format only.
- Task managing
    - Consult all tasks: The server will return the information of all registered tasks, without any filter applied.
    - Create a task: The server will accept data of one task that will be registered on the database. The server will confirm if the registration was successful or not.
    - Modify a task: The server will accept data of one task that will be compared to the data stored in the database, and will be modified to sync with the new data received.

    - Delete a task by a task id: The server will receive an id of the desired task that will be deleted.
    - Consult a task with filters: The server will receive the information of a specific filter that will be applied to the data before sending a response.
        - By title containing a sub-string: The user will provide a sub-string. The server will return all tasks which title contain that sub-string of the description.
        - By description containing a sub-string: The user will provide a sub-string. The server will return all tasks which contain that sub-string on the description.
        - By date of creation: The user will provide a pair of dates, and all the tasks with a creation date within that time interval will be provided.
        - By date of desired completion: Same as date of creation, but taking into account date of desired completion.
        - By priority: The user will provide a priority level, and all tasks within that priority level will be returned.

      These filter can be applied in a cascading fashion. Then, the result returned by the server will be, in terms of Set theory, the intersection between the results of all filters.
    - Change the state of a task: Different from updating a task, the information consists in a new state for the task. If the new state is Completed, then a completed date will be generated by the server. If the state changes to another state different than completed, the completed date will be removed.

      All states can transition to any other state, except for the same state. For example, an abandoned task can transition into a pending task, but a completed task cannot transition to completed, nor pending to pending, etc.

## 3.2 Nonfunctional requirements

- Docker deployment: The server will be containerized in a docker container that will include the db and the
- Host OS: Linux (Debian) docker container
- Technology: Java with Spring boot

- Database: local MariaDB database
- Container: Docker container

## 3.3 System platform description

The system will be built using the Java programming language, making use of the Spring boot framework to simplify the development and deployment of the software. The server will be hosted on a Linux platform, on a Debian distribution.

### 3.3.1 Main Paltforms and frameworks

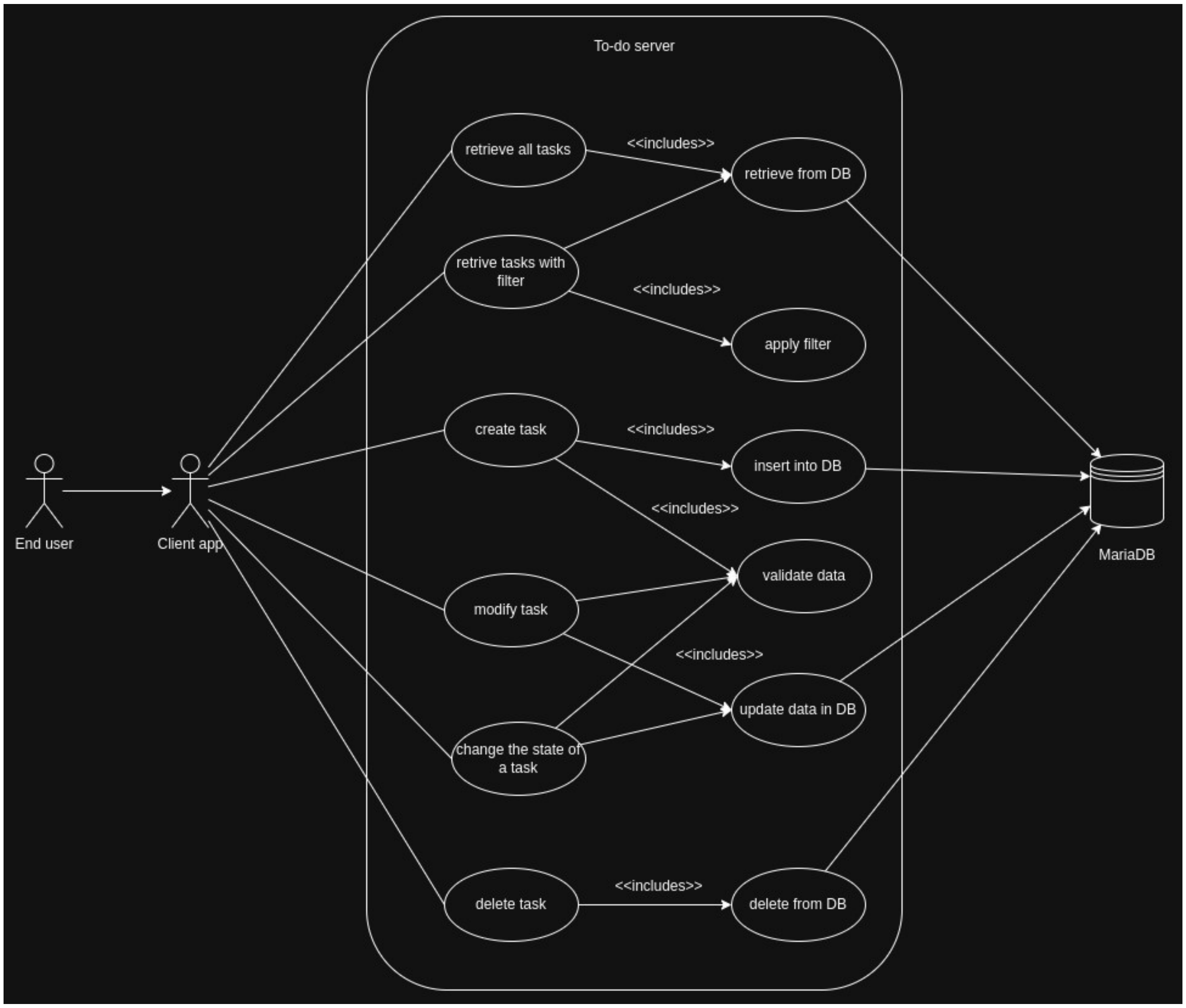
The whole system will be based on Java 17, working with the Spring framework. More preciselly, with the Spring Boot framework. This system will allow to build a type-safe application.

### 3.3.2 Persistance solution

The app will use Spring's JPA API to connect to the database, using MariaDB as the persistance provider via JDBC.

## 3.4 App interaction

### 3.4.1 Use case diagram



### 3.4.2 Retrieve all tasks

| Name | Retrieve all tasks |
|---|---|
| Description | The user retrieves all tasks. |
| Pre conditions | • The server is in correct conditions. |
| Post conditions | All the tasks registered have been sent to the user. |
| Basic path | 1. The user makes a requests.<br>2. The server retreieves data from the DB (Retrieve from DB use case).<br>3. The server sends the data to the user. |
| Alternate paths | None |
| Exception paths | None |

| | |
|---|---|
| Includes relationships | • Retrieve from db. |

### 3.4.3 Retrieve tasks with filter

| Name | Retrieve tasks with filter |
|---|---|
| Description | The user retrieves tasks with a filter applied. |
| Pre conditions | • The server is in correct conditions. |
| Post conditions | All the tasks registered that comply with the filter have been returned. |
| Basic path | 1. The user makes a requests that contains the filtering criteria.<br>2. The server retreieves data from the DB (Retrieve from DB use case).<br>3. The server filters the data (apply filter user case)<br>4. The server sends the data to the user. |
| Alternate paths | 1. The filtering criteria that the user provides desn’t match any data<br>2. The user is notified of this issue. |
| Exception paths | 1. The filtering criteria that the user provides doesn’t exist.<br><br>In case of time interval filterings:<br>1. The end date of the interval is prior to the ending date<br>2. The user is notified and the operation cancelled. |
| Includes relationships | • Retrieve from db.<br>• Apply filter. |

### 3.4.4 Create task

| Name | Create task |
|---|---|
| Description | The user posts data from a new task |
| Pre conditions | • The server is in correct conditions. |
| Post conditions | The task received is registered in the database, if valid. |
| Basic path | 1. The user makes a requests that contains data of a new task.<br>2. The server validates the data (validate data use case).<br>3. The data is stored in the databse (insert into DB use case).<br>4. The user is notified of the correct operation. |
| Alternate paths | None |
| Exception paths | 1. The data received is not valid<br>2. The user is notified |
| Includes relationships | • Insert into DB<br>• Validate data |

### 3.4.5 Modify task

| Name | Modify task |
|---|---|
| Description | The user sends data of an existing task that has to be modified. |

| | |
|---|---|
| Pre conditions | • The server is in correct conditions. |
| Post conditions | The task is correctly updated. |
| Basic path | 1. The user makes a requests that contains the new data, as well as the entity id.<br>2. The server retreieves data from the DB (Retrieve from DB use case).<br>3. The server filters the data (apply filter user case)<br>4. The server sends the data to the user. |
| Alternate paths | 1. The filtering criteria that the user provides desn’t match any data.<br>2. The user is notified of this issue.<br><br>3. The task id is not valid. |
| Exception paths | 1a. The new data is not valid.<br><br>1.b The user tries to update no-updatable information. |
| Includes relationships | • Update data in DB<br>• Validate data |

## 3.4.6 Change the state of a task

| Name | Change the state of a task |
|---|---|
| Description | The user sends a new state and the id of the task that is going to change. |
| Pre conditions | • The server is in correct conditions. |
| Post conditions | The task’s state is correctly updated. |
| Basic path | 1. The user makes a requests that the new state and the id of the task.<br>2. The server validates the new state.<br>3. The server updates the data in the database (Update data in DB).<br>4. The user is notified. |
| Alternate paths | None |
| Exception paths | 1a. The new state is not valid / doesn’t exist.<br><br>1.b The user tries to update to the current state. |
| Includes relationships | • Update data in DB |

## 3.4.7 Delete task

| Name | Delete task |
|---|---|
| Description | The user sends the id of the desired task. |
| Pre conditions | • The server is in correct conditions. |
| Post conditions | The task’s state is correctly deleted. |
| Basic path | 1. The user makes a requests that contains the id of the desired task.<br>2. The server updates the data in the database (Delete from DB).<br>3. The user is notified. |
| Alternate paths | None |

| | |
|---|---|
| Exception paths | 1. The selected task id doesn’t exist<br>2. The user is notified. |
| Includes relationships | • Delete from DB |

### 3.4.8 Retrieve from DB

| **Name** | **Retrieve from DB** |
|---|---|
| Description | The user sends the id of the desired task. |
| Pre conditions | • The server is in correct conditions.<br>• The database is in correct conditions. |
| Post conditions | The data required is returned to the server |
| Basic path | 1. The server makes a query to the database (via Spring’s JPA API)<br>2. The data is returned to the server. |
| Alternate paths | 1. The requested data doesn’t exist.<br>2. The server gets notified. |
| Exception paths | 1. The database is not accessible.<br>2. The server sends a notify to the user. |
| Includes relationships | None |

### 3.4.9 Apply filter

| **Name** | **Apply filter** |
|---|---|
| Description | The server applies a filter to the data collected |
| Pre conditions | • The server is in correct conditions.<br>• The data to be processed is valid. |
| Post conditions | The filtered data is routed to the nex step. |
| Basic path | 1. The server determines the kind of filters to be applied.<br>2. The data is filtered following the criteria.<br>3. The data is routed to the next step. |
| Alternate paths | None |
| Exception paths | 1. The data to be filtered is null.<br>2. The user is notified.<br><br>3. One of the filter fails to be applied.<br>4. The user is notified. |
| Includes relationships | None |

### 3.4.10 Insert into DB

### 3.4.11 Validate data

### 3.4.12 Update data in DB

### 3.4.13 Delete from DB

## 3.5 Data design

### 3.5.1 Class design

**Task**

A task is the central entity that the server will manage.

| Field name | Field description | Field type | Default |
|---|---|---|---|
| id | The identifier of the task. | Integer | Value, automatic |
| title | The title of the task. It describes shortly what the task is about. | String | Null (must be provided) |
| state | The state of the task. | String (Pending, In process, Completed, Abandoned) | Value (Pending) |
| description | A longer description of the task, or some text that is related in some way to the task. | String | Null |
| priority | The priority level of the task, as declared by the user. | Integer | 1 |
| creation_date | The time of creation of the task. It's assigned automatically at runtime by the server. | DateTime | Value, automatic |
| completion_date | The time when the task was completed. Pending tasks will not have this field set. | DateTime | Null, automatic |
| due_date | The time when the task must be completed, set by the user. | DateTime | Current date time. |

**Custom DateTime class**

Given the current limitations of the JSON parsing system of Spring,  Jackson library, to parse Java's native DateTime objects, a custom DateTime class will be created.

| Field name | Field description | Field type | Default |
|---|---|---|---|
| year | The year of the date represented | Integer | null |
| month | The month in numbers | Integer | null |
| day | The number of the day | Integer | null |
| hour | The hour | Integer | null |
| minute | The minute | Integer | null |
| second | The second | Integer | null |

Given that this class doesn't have all the mechanisms provided by Java's DateTime or LocalDateTime classes, this class will only be used as a way of bypassing Jackson's limitations. For example, hour values can be then values ranging from 0 to the maximum Integer value, or even negative numbers. The validation of this data will be performed later in the server, so invalid DateTime values aren't inserted into the database.

### 3.5.2 JSON response object design

The Task JSON object will be designed as follows, and this structured will be used by the server responses. Note that in some operations where not all values are known (like inserting  a new Task, where the id is unknown), these null fields can be ignored when posting.

```
user_id: {
   title: String,

   state: String,

   description: String,

   priority: integer,

   creation_date: formatted String,

   completion_date: formatted String,

   due_date: formatted String

 } ?
```

### 3.5.3 Conceptual database design

Following the class design, the database will consist of only one table.

## 3.6 API design

### 3.6.1 Endpoint specification

For every use-case in which the user is involved, there is an API endpoint that corresponds.

In cases where *formatted String* is used it places like dates, it refers to a String representing a date and a time without zone in a format compliant with the ISO-8601 standard.

**GET /tasks (Retrieve all tasks use-case)**

This endpoint returns all the tasks registered in the database.

**JSON response:**

An array containing objects, which will contain the data of one task each.

**Input format:**

This endpoint doesn't accept input.

**GET /tasks?query_params (Retrieve tasks with filters use-case)**

This endpoint allows the user to retrieve tasks that comply with the selected filters and filter values.

The following is a list of accepted filters.

| Filter name | Values |
|---|---|
| title_contains | The string that the title must contain. |
| desc_contains | The string that the description must contain. |
| priority | An Integer representing the priority number. |

Some filters will need two values to be applied, and the two values must be provided simultaneously.

| Filter name | Values |
|---|---|
| desired_completion_start | An ISO-8601 date String indicating the start of the firltering interval. |
| desired_completion_end | An ISO-8601 date String indicating the end of the firltering interval. |

|  |  |
|---|---|
| creation_date_start | An ISO-8601 date String indicating the start of the firltering interval. |
| creation_date_end | An ISO-8601 date String indicating the end of the firltering interval. |

In these cases, the end date must be higher than the start date. Otherwise, this will cause an error.

These filters and the values are passed in the query parameters of the request, like the following example:

```
GET /tasks?title_contains=recipes&priority=2
```

This will return all the tasks which contain the sub-string “recipes” in the title and have a priority level of 2.

**JSON response**

The Server will return a JSON array containing objects which contain the data of the tasks that were filtered. If note, it will be an empty array.

**POST /tasks (Create task use-case)**

This endpoint allows the user to create new tasks, posting it’s information.

**JSON data format**

The input JSON must provide the data necessary to create a new entity. This excludes some fields that are created automatically by the server.  Also, some fields are optional, and not providing them will allow the server to set default values. The JSON must contain the next fields:

| Name | Value type |
|---|---|
| title | String |
| description* | String |
| priority* | Integer |
| due_date* | Custom Date class |

The fields marked with an * are optional, and if no value is provided, then the server will assign a default following the class *design section*.

**JSON response**

The response will consist of a simple JSON object with a message, depending on the outcome of the operation:

{message: “success”} – for a successful insertion

{error: error message} – for some kind of error notification

### PUT /tasks/{id} (Modify task use-case)

This endpoint allows to modify all properties of a task.

**JSON data format**

The user must provide a complete description of the task entity that is going to be modified, not only the modified fields.

**PATCH /tasks{id} (Change the state of a task use-case)**

### DELETE /tasks/{id} (Delete task use-case)

This endpoint allows the user to provide an id number for the entity that the user wants to delete. This id will be provided in the URI