Generic Web Decision Maker System

A DegreeThesis

Submitted to the Faculty of the

Escola Tècnicad'Enginyeria de Telecomunicació de Barcelona

UniversitatPolitècnica de Catalunya

by

Isaac Ollé Gatell

In partial fulfilment

of the requirements for the degree in

Telecomunications Engineering In Telemeatics mention

ENGINEERING

Advisor: Marcel Fernandez

Barcelona, May 2021

Abstract

This thesis consist basically to make a web application where there is a data source connected, which feeds the application, through 2 protocols and is observed the different behaviours. A user can see this data, combine data, and put some conditions to the system. The system will notify the user if the conditions are passing or don't.

This project is build from the floor, so there is an integral building of a web application, frontend, backend, data base and data source. Also is focused on that it should work with all data sources that pass some conditions, generic system.

Resum

Aquesta tesis consisteix bàsicament en construir una aplicació web la qual rep informació d'un extractor de dades, a través de 2 protocols i es comparen com es comporten els protocols. L'usuari veu aquesta informació rebuda per pantalla, i l'aplicació permet que la combini i que hi posi les condicions que ell vulgui. El sistema avisarà a l'usuari quan aquestes condicions s'estan complint o no.

És un projecte creat des de zero i integral que conté totes les parts d'una aplicació web, frontend, backend, base de dades i extractor de dades. També s'ha tingut en compte que aquest projecte funcioni sempre que l'extractor compleixi algunes condicions, per tant podem dir que es un sistema genèric.

Resumen

Esta tesis consiste básicamente en construir una aplicación web la cual recibe información de una fuente de datos, a través de 2 protocoles y se compara como se comportan estos. El usuario ve esta información recibida por pantalla, y la aplicación permite que el usuario la combine y que ponga las condiciones que él/ella quiera . El sistema avisará al usuario cuando estas condiciones se están cumpliendo o no.

Es un proyecto creado des de cero e integral que contiene todas las partes de una aplicación web, frontend, backend, base d datos y extractor de datos. También se ha tenido en cuenta que este proyecto funcione siempre que el extractor cumpla algunas condiciones, por lo que tanto se puede decir que es un sistema genérico.

I would like to dedicate this thesis mainly to my tutor Marcel Fernàndez, for guide me to found a really thesis theme that I really enjoyed developing, and give my tips and advises to focus on what is really important, what project consist to, and not to work out of scope.

Last but not least, I'm really thankful to my brother Martí Ollé who helped me in some of the blocking points of the thesis.

Acknowledgements

The mainly person who has helped to develop this project is Marcel Fernandez, basically when the thesis was in a blocking point, he told which web pages can be found the solution, or a better explanation about the problem.

Revision history and approval record

Revision	Date	Purpose
0	10/01/2022	Document creation
1	22/01/2022	Document revision
2	22/01/2022	Document modification
3	22/01/2022	Document delivery

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Isaac Ollé Gatell	Isaac.olle@estudiantat.upc.edu
Marcel Fernandez	Marcel.fernandez@upc.edu

Written by:		Reviewed and approved by:	
Date	24/05/2021	Date	22/01/2022
Name	Isaac Ollé Gatell	Name	Marcel Fernandez
Position	Project Author	Position	Project Supervisor

Table of contents

Abstract....1
Resum....2
Resumen....3
Acknowledgements....5
Revision history and approval record....6
Table of contents....7
List of Figures....9
List of Tables:....11
1. Introduction....13
a. Statement of purpose(objectives)....13
b. Requirments and specifications....13
c. Methods and Procedures....14
d. Work plan with tasks, milestones and Gantt diagram....14
e. Description of the deviations....17
2. State of the art of the technology used or applied in this thesis:....18
3. Methodology / project development:....19
3.1. Configuration project on your local environment....19
3.2. Overview....20
3.3. Data Source Simulator....22
3.4. Database....24
3.5. HTTP Backend....26
3.5.1 Configuration Files....26
3.5.2 Scheduling and CRON....27
3.5.3 Business Logic....28
3.5.4 JPA Repositories....30
3.6. HTTP Frontend....31
3.6.1 Items Object....32
3.6.2 Components....33
3.7. WebSocketBackend....35
3.7.1 Business Logic....35
3.7.2 Configuration Class....37

3.7.3 Controller....37
3.7.4 Frontend....38
4. Results....40
5. Budget....43
6. Conclusions and future development:....44
Bibliography:....46
Appendices (optional):....48
Helping Figures....48
Demo....54
Glossary....60

List of Figures

3.1. First and Final system idea. Pàg.210

3.2. Autowired Intance Problem Scheme. Pàg.21

3.3. Set of Data Base Tables and the Actions Allowed. Pàg.25

3.4. Queries Management. Pàg.27

3.5. GeneralDto Scheme. Pàg.28

3.6. Frontend Architecture. Pàg.30

3.7. Frontend Object Compared to Backend Object. Pàg.31

3.8. Item Properties Scheme. Pàg.32

3.9. NewColumn Dialog Explaining FormArray. Pàg.32

3.10. WebSocket Full Architecture. Pàg.33

4.1. General architecture view. Pàg.36

Appendice Figures

1 Backend dependencies. Pàg.47

2 Frontend dependencies. Pàg.47

3 Application properties. Pàg.20

4 Final Project Structure. Pàg.21

5 WebSocketControllerListenerClass. Pàg.23

6 WeakReference Example. Pàg.24

7 Release Listener Function. Pàg.24

8 UserService Inside and Outside Compile Scope. Pàg.25

9 Generate Random Value to Sent to Frontend and Store on Data Base. Pàg.25

10 Security Folder with CORS Configuration. Pàg.28

11 Frontend HttpInterceptor. Pàg.28

12 Scheduled with FixedRate. Pàg.29

13 CRON Example. Pàg.29

14 CRON Example in Application Properties. Pàg.29

15 All CRON Possibilities. Pàg.29

16 NewColumns Table. Pàg30

17 Decisions Table. Pàg.31

18 GeneralDto Properties. Pàg.32

19 Custom Query Example. Pàg.32

20 Manual Connection Example. Pàg.34

21. All Backend and Database Architecture. Pàg.34

22 Home Routing Component. Pàg.38

23 Navigate Function. Pàg.38

24 WebSocket Configuration Error. Pàg.40

25 Open Two Ports Throught Tomcat. Pàg.41

26 WebSocket Header Failing. Pàg.42

27 WebSecurityConfig. Pàg.42

28 WebSocketConfig. Pàg.43

29. Socket Object Sent to Frontend. Pàg.44

30 Client Observable Scheme. Pàg.45

31 Rxjs Subject. Pàg.45

32 Switching Protocols with Socket Message. Pàg.46

33 ApiData Dashboard table. Pàg.57

34 NewColumn Dialog. Pàg.57

35 NewColumn Dialog with Several Objects. Pàg.58

36 UserColumns Asynchronous Tab. Pàg.58

37 Decisions Dialog. Pàg.58

38 Decisions Dialog with Several set. Pàg.59

39 UserColumns Tab with Decisions Columns Filled. Pàg.59

40 NewDecisions Dialog with Ids. Pàg.60

41 ApiData Synchronous Tab. Pàg.60

42 ApiData Asynchronous Tab. Pàg.61

List of Tables:

1.1. Work Packages. Pàg.12-14

1.2. Milestones. Pàg.15

1.3. Gantt. Pàg.15

3.1. NewColumns database. Pàg.25

3.2. Decisions database table with value. Pàg.26

5.1. Costs Table. Pàg.51

5.2. Average year cost of a web application. Pàg.51

5.3. App maintenance and improvement. Pàg 52

1. Introduction

a. Statement of purpose(objectives)

The idea of this project is to build user a powerful tool to analyze data, and notify the user when the data are passing certain conditions. The project not only consist in notifying the user, also it can be used to take some actions were the conditions are the expected.

To do that the project is developed with a web architecture, frontend, backend, and data base and a simulated data source. Make a good data source and good data extractor is out of scope, because we are focusing on user functionalities.

Frontend part has been developed on Angular language, backend part on Kotlin language and data base is SQL.

This project is focused on make it in a generic way, that should work independently the data source kind of data, but the data source has to comply two premises, first of them is that the data given has to be numeric, and our data base knows what is going to arrive through the data source.

The system is designed to be able to work on real time and take decisions on real time son there is no sense to storing old data on data base, so de table which stores the data source data has a maximum of columns of 100, once fully database starts to rewrite over the oldest data.

There is two ways to build a tool like that, asynchronous way and synchronous way; the first one is the tool that this projects consist of, the frontend part orders de new data to the data base, and the synchronous way should be that the data base sends the new data once it has arrived. Http protocol works asynchronous and WebSocket synchronous.

b. Requirements and specifications.

Project requirements:

- The project should be able to work on almost any system which needs to take decisions.
- He decisions have to be displayed in some understandable way.
- The user has to be able to sum, subtract, multiply, and divide columns.
- The backend should have a good architecture as frontend.
- Relational database

Project specifications:

- The database tables will be created through the backend code to synchronize the object get by the extractor and the object stored on the database.
- An angular frontend part will display the results with graphics.
- Hexagonal architecture will be the chosen architecture on the backend.
- The project will be carry out with MySQL a relational data base, that's a design decision because we don't need efficiency on database, but we need robustness, to make the project easier.

These were the requirements and specifications of the project at the beginning. The requirements were fully completed. But the specifications actually changed a little.

First of all, the fact to create the database tables through the code, was considered work of a good data extractor and that's not about this project is.

Another thing is that hexagonal architecture just makes sense with backend with more than two outputs/inputs. If the backend just have to connect to data base and frontend this will be hexagonal always. So if the application connected to another API taking into account the decisions, carry out actions regarding the decisions, then a hexagonal architecture would be a good practice to develop the project. It is true that protecting the services with Interfaces would be more accurate to a hexagonal architecture but this project is a little application and there is no need to do this.

c. Methods and procedures, citing if this work is a continuation of another project or it uses applications, algorithms, software or hardware previously developed by other authors.

As is said before this project is build from nothing so there is no work done by others.

d. Work plan with tasks, milestones and a Gantt diagram.

Work Packages:

Project: Creation and HTTP Scenario	WP ref: (WP1)
Major constituent: HTTP Protocol	Sheet 1 of 5
Short description: Make a system maker throught http protocol	Planned start date: 01/09/2021 Planned end date: 31/09/2021
	Start event: 01/09/2021

	End event: 31/09/2021	
Internal task T1: Creation of the database, backend and frontend Internal task T2: Create the communication with them Internal task T3: Make the logic on frontend and backend	Deliverables: The communications between backend and database	Dates: 31/09/2021

Project: WebSocket Scenario	WP ref: (WP2)	
Major constituent: WebSocket Protocol	Sheet 2 of 5	
Short description: Build the implamentation with WebSocket Protocol	Planned start date:01/10/2021 Planned end date: 10/01/2022	
	Start event: 01/10/2021 End event: 10/1/2022	
Internal task T1: Create communications throught WebScoket Internal task T2: Make the logit to make decisions	Deliverables: The table	Dates: 08/04/2021

Project: Comparisons and Put Together	WP ref: (WP3)	
Major constituent: HTTP vs. WebSocket	Sheet 3 of 5	
Short description: Comparison between this 2 systems and real life applications.	Planned start date:01/11/2021 Planned end date:31/11/2021	
	Start event:10/1/2022 End event:20/1/2022	
Internal task T1: Make a conclusion in which cases each protocol is better. Internal task T2: Look for real life applications which it's essential to use one of them.	Deliverables:	Dates: 31/11/2021

Project: Write Final Report	WP ref: (WP4)	
Major constituent: Write Final report	Sheet 4 of 4	
Short description: Write the conclusions	Planned start date:01/12/2021 Planned end date: 23/01/2022	
	Start event:18/02/2022 End event:23/02/2022	
Internal task T1: write the conclusions	Deliverables: Project itself.	Dates: 15/05/2021

Table 1.1 Work Packages

Milestones

WP#	Task#	Short title	Milestone / deliverable	Date (week)
1	1	Creation FE, BE and DB	Http	8
	2	Create communications	Http	8
	3	Http logic	Http	8
2	1	Create communications	WebSocket	9
	2	WebSocket Logic	WebSocket	9
3	1	Comparison	1st release	10
	2	Real life applications	1st release	10
4	1	Documentation	2nd release	11

Table 1.2. Milestones

Gantt

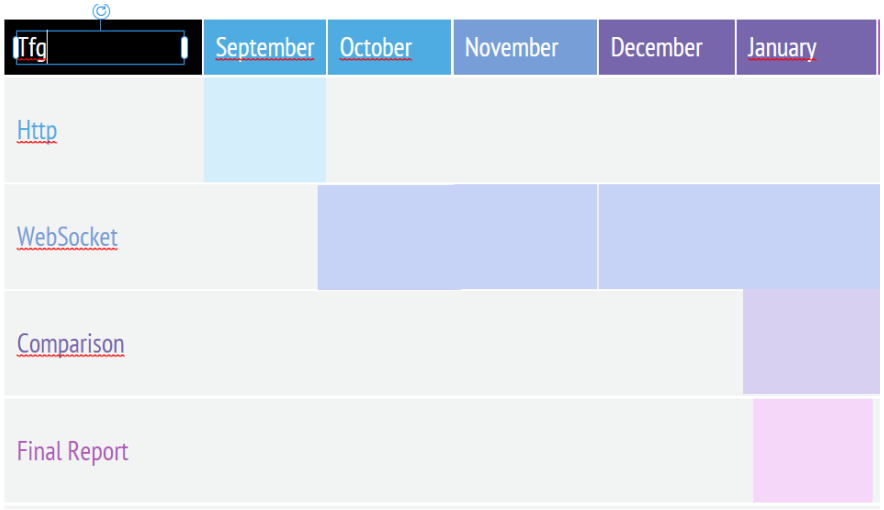


Figure 1.3 Gantt

e. **Description of the deviations from the initial plan and incidences that may have occurred.**

There are several things that didn't go as were expect at the beginning of the project.

First thing that was changes, was the database model. At the beginning I would like to work in a non relational database MongoDB, on that moment the project should have create the database tables through code so MongoDB presented a problem that SQL do not. This problem is that if the table is created with the kotlin codes, a non relational data base have to create the references as well, meanwhile on SQL all columns are referenced.

There are two things that changed because, were found a newer technology that makes it easier. These kinds of things are the repositories and simulate the data source. At the beginning the repositories should be all created manually, finally not all of them are, during the development was found on internet the JPA tool repositories which make easier this work. This tool is explained later.

The simulation changed because at the beginning it was a thread, but for make de code cleaner and for the possibility to develop on the future a sync system. Thread presented some inconveniency that @Scheduled and CRON tool does not.

Web socket protocol suffered a lot of mutations during his implementation and was the most difficult part on the project, taking into account that there is just 4 month to develop all and not all the problems had trivial solutions.

2. State of the art of the technology used or applied in this thesis:

Several technologies has been used to carry out this project:

- Visual Studio 2019: platform where frontend was developed, the best platform for frontend developers, really helpful.
- IntelIJ version 2020.3: platform supporting the backend, very powerful, easy an comfortable to develop. This project should work with others platforms like eclipse.
- Spring boot version 2.5.0: this is a tool that makes easy to create custom API rest. This project is just using the basic spring boot tools to make an API Rest
- Gradle version 6.8.3: gradle is a library repository importing certain libraries; the work on the backend can be quiet less. See the backend dependencies in the annex. Figure 1.
- MySQL version 8.0.23: One of the latest MySQL versions there is a change from the other versions that is affecting the project. Is the fact that from now MySQL does not allow tables without Id column. MySQL Workbench 8.0. To manage the database tables.
- Kotlin version 1.4.32: Backend language, the version before latest, there is no special that this project uses from Kotlin newest versions.
- Angular 11.2.12: Frontend Language is very powerful; this language is the chosen over react or view.js because is focused on integral applications like this project. By now is a little project but if in the future there new implementations and functionalities react or view.js presents failures.
- Rxjs library 7.1.0: Very important library to develop the project, gives a lot of tools and mechanism to create .Net functionalities easy.
- Angular Materials 12.0.1: Angular library used to the frontend part, plenty of tools used to make the user experience understandable.

3. Methodology / project development:

3.1 Configure project on your local environment

To configure this project on your local environment is needed to download the compress file and decompress it on your computer.

The frontend set up is very simple, just open the Visual Studio code, then search the folder download and look for the 'tfg' folder, open it, open a terminal and execute *npm install*. Make sure that the dependencies got are the same as the Annex Figure 2

If does not then search for install it on npmjs.com. There should be all of them.

Configuring the backend part is simple too, before run it you need to install Gradle, at least the version specified on Chapter 2 of this document, there is no guarantee about if it could work on older Gradle versions. Add Gradle in settings/environment variables. Once installed execute on terminal, first *gradle build* and then *gradle bootRun* or *gradlew bootRun* depending if its a windows or a linux which is working.

The database is the only part which cannot be uploaded on the internet so, to be configured you need to create it, if MySQL is not installed, download MySQL workbench, create a data base with 'root' user and 'root' password, and execute this 3 scripts:

- CREATE TABLE `apiinformation` (

  `id` bigint NOT NULL,

` atr1` double NOT NULL,

  `atr2` double NOT NULL,

  `atr3` double NOT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8

- CREATE TABLE `decision` (
  `id` int NOT NULL AUTO_INCREMENT,
  `var1` varchar(45) NOT NULL,
  `type` varchar(45) NOT NULL,
  `var2` varchar(45) NOT NULL,
  `value` int DEFAULT NULL,

```
  PRIMARY KEY (`id`)
 ) ENGINE=InnoDB AUTO_INCREMENT=28 DEFAULT CHARSET=utf8
```

- CREATE TABLE `newcolumn` (
  `id` int NOT NULL AUTO_INCREMENT,
  `col1` varchar(45) NOT NULL,
  `col2` varchar(45) DEFAULT NULL,
  `sign` varchar(45) NOT NULL,
  `name` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=29 DEFAULT CHARSET=utf8

This MSQL scripts creates the data base tables involved on the project. Then run MySQL through terminal with command *mysql start*.

You can execute it on a MySQL online server if there is access on it.

In the annex Figure 3 there is the Application properties needed to run connect the backend with database.

If, the configuration can't be set correctly on the annex there is a little demo of how it works.

3.2 Overview

It's highly recommended to see the demo on the Annex.

To explain how the project has been done, know how it works and understand the design decisions, this document is going to explain first a general overview and then explain each element of the project separately and deeply.

The project is a fully integrated web project with frontend, backend, and database created from nothing. Frontend is developed with angular because is a very powerful tool, and focused on full integrated applications like this project, meanwhile react or view.js are more useful on applications which UI is the most part. On backend is used Kotlin, because is one of the most moderns, powerful, completes languages on server part, another reason is because is over Java, is easy to understand and is a good chance to learn Kotlin. Kotlin server used is the one that JetBrains has that just translate Kotlin to Java. The database technology is MySQL because is a relational data base. The problems related with non relational database will be explained later.

The top level schema on the project just differentiates 4 elements. This was the first architecture idea, this design give a first view about in what consist this project.

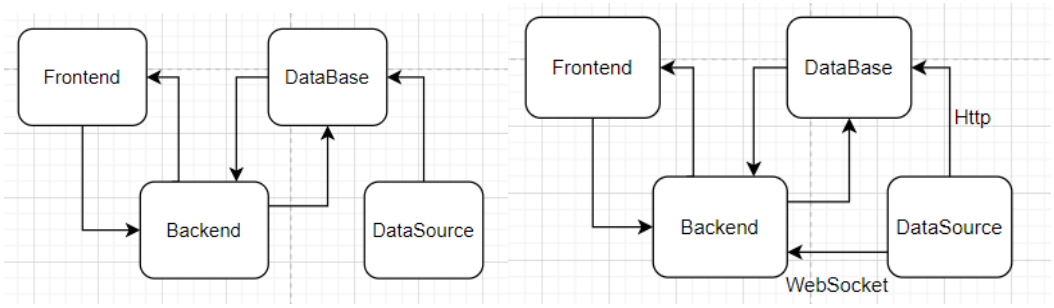


Figure 3.1 First and Last System Idea

This was not exactly the way that this thesis was developed but it's useful to create a first idea about the general architecture.

As the image show, there are 4 elements there is a data source (external API, sensors, etc) that provides data to a database, and that data base can be queried by a backend to process it and show on the frontend the user requests. All the elements can communicate each other, but the data generated in Data Source is stored in data base. Also is sent under WebSocket Protocol and that means that goes directly to the Backend. So the final system idea would look like as the right picture.

The data source part in this project is considered out of scope, so in this project it is mocked.

There are two premises to take into account. First is that the object sent by the data source is known in our system. And the second one is that the system considerate that the object received has numbers and not strings. It is because one of the project motivations is to make a system able to take decisions on crypto currency, and although is a general system, it has been implemented on that way, thinking in one day if the system is improved, it can be able to invest on crypto currency itself.

The last thing to know before starting the deep analysis of the architecture is that this system is build with an asynchronous and synchronous pattern. This means that frontend every X seconds (10' is good to test) asks to backend for the new data, this has inconveniences and advantages, an inconvenience is that if you can't estimate the time between 2 object generations or arrivals from/by data source, maybe the system is asking too much to get nothing, a little example: If there is a new Object on 'apiinformation' table every 2 minutes, if frontend is asking every 10' just 1 of 12 request will return with a new value, that can make the system inefficient. So an asynchronous system has to estimate how much time there is between object generations or arrivals on DataSource.

Let's divide the explanation in 2 parts, first of them the asynchronous there will be exposed the asynchronous part of the project, Http protocol and then the synchronous, WebSocket Protocol.

3.3 Data Source Simulator

Even though data source is simulated, what this project does is to generate a random object every X seconds and send it to the data base. On the test case this object is a three attribute object (atr1, atr2, atr3), and it's stored on a data base table with an id.

DataSoruce simulator contains a @Autowired of a WebSocketController class, this is because when a RandomEntity is generated, this will be sent through WebSocket to the frontend, and it's a bean because just can exist one WebSocketController in memory, due to that the Controller stores the socket sessions, and if simulator has a new Object () this won't have the stored sessions by Controller because there will be 2 WebSocketControllers in memory.

With this WebSocket @Autowired there were a hard problem.

This was that DataSourceSimulator was initializing WebSocketController as null, the @Autowired annotation was not working. This means that WebSocketController object which receives the connections and implements WebSocketHandler was not the same WebSocketController declared on DataSourceSimulator, because the first one had the connections and the one inside DataSource class does not.

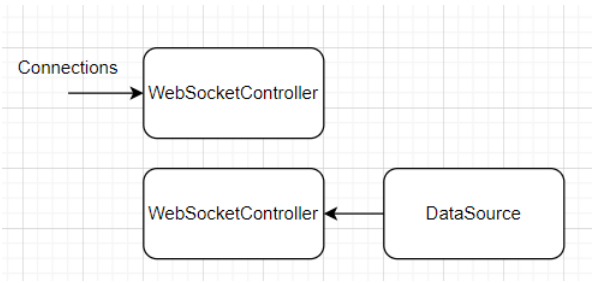


Figure 3.2 Autowired Instance Problem

At the first moment we thought that this was an architecture problem that this class was having a cyclic reference and to protect against this, it let WebSocketController as null, but this was not the question.

One solution purposed was instance a Kotlin listeners on WebSocketController, this is a powerful Kotlin tool through interfaces.

A class was created called WebSocketControllerListener implementing a custom interface called DataSourceListener. This class had a primary constructor with a WebSocketSession array because

when it receives a RandomEntity just created, is needed a list of WebSocketSessions to send this message. The interface implemented had a method called “onReleaseListener(entity: RandomEntity)” which is implemented over ‘override’ in this class, also it receives the entity created and the one that have to be sent.

Moreover, this class had a DataSourceSimulator object as property and it was also an @Autowired. On the init method had 2 functions. The annex Figure 4 references at this Class.

The implementation was on DataSourceimulator. So let’s take a look on that.

First of all the DataSourceSimulator had a Weak Reference, see on annex Figure 5 how it is implemented:

A weak reference is a reference that if there is some other reference on memory which same typed this will be removed from memory. In other words there just can be one reference of this class in memory.

The implementations of the two functions seen above are very simple:

- “this.dataSourceSimulator.addListener(this)” sets the listener WeakReference for the “this” param which is a class that implements the WeakReference “DataSourceListener” type as interface.
- Release() function gets the weakreference listener and triggers the “onReleasedListeners(entity: RandomEntity)”, this means that all classes implementing DataSourceListener will be triggered, in this case there will be just one class due to the WeakReference. Annex Figure 6 shows the implamentation.

This function is called every time that DataSourceSimulator generates one value.

But the problem here was that @Autowired DataSourceSimulator was initialized as null also. And this was not the final solution.

Finally, the problem root was that MainApplication file was not located on the right place and not all files were compiling as it should do. And a result of it they were not initialized because the compile was out of MainApplication file scope. The file was inside a “start” folder and relocating to the main project folder all problems with @Autowired were solved.

See in the Annex an Example of out of compile Scope and in of compile scope.Figure 7

On the first image UserService would not work with @Autowired annotation and in the second one yes.

The listener way would be solve the problem if the WebSocketControllerListener(..) class were located in the right folder.

So knowing it the code was rollback relocating all classes to the right place and did the first proposal for this functionality, declare a WebSocketController property to DataSourceSimulator Class and when a value is generated then call the function "webSocketController.sendMessage()". Se the annex figure 9 the implamentation of "generateOneRandomValue()".

Because of this problem and other that will be explained after, during some weeks the project was split in two different projects, the http protocol and WebSocket protocol.

On the other hand MySQL 8 requires an id for every table and regarding that it's important to store the time data when the object has been generated by the mock data source, the id and the generation time are mixed. So the id corresponds to a time stamp that indicates the current date in milliseconds. On frontend is important to print the time that sample has been generated.

The mechanism to generate the object every X seconds will be explained on the backend part.

To go deeper to the project, let's start to explain the data base part.

3.4 DataBase

The data base part consists in a MySQL database, MySQL is a relational data base. This project has been developed by this kind of database because they can be referenced by any column. Considering that the objects received can have different attributes, with a non relational data base it would be more complex regarding the index generation that a non relational data base needs to access on his data. If the system were implemented with a non relational database there should be three premises to take into account, the two explained before and knowing which references the data base need to get the data from data base properly.

This project itself has three database tables. One table the most important stores X samples about the data got through data source, this is done because if the data source is an external API, backend can call it to get older data but in case that the data source are sensors and there is no API storing data, the system have to save it in the local database. The system stores just X samples because if the system has to make decisions in real time there is no sense to store data

from some time ago that don't affect to the latest decisions. So when the table is full, it starts to rewrite the oldest columns.

The second table, stores what combine of data user wants to get, if the user get attribute A:Int and B:Int from the data source, the user can decide if to sum, multiply, divide, subtract A and B values. So if the user, for example, wants to get A+B values on the NewColumns data base there will be stored the first line, the other lines are for divide Atr3/Atr3 and for just return atr2 :

	id	col1	col2	sign	name
▶	1	atr1	atr2	SUM	NewColumn1 (atr1+atr2)
	2	atr3	atr3	DIV	NewColu,m3 (atr3/atr3)
	3	atr2	atr2	ASIGN	NewColumn2 (atr2 asign)
*	NULL	NULL	NULL	NULL	NULL

Table 3.1 NewColumns Database Table

Each line in this data table is a NewColumn Object.

"ASIGN" means that the attribute stays equals, this is done in this way because frontend needed it, to have a good view and a understandable experience.

The name column is the name that the new column will have on frontend.

Frontend can create, delete, modify and get the values stored on the table NewColums.

Finally, the last table is the decisions table it, stores the information to make decisions on the backend, it has a var1 and a var2 that will be the variables to compare a sign comparing equals to, bigger than, smaller than and a value.

Var1 and var2 values are corresponding to a column names on the frontend which has to be summed, multiplied etc. But if one of them has the value "value", the system replace that column for the number inside "value" column, comparing the other column with the value. So the comparison will be between a number and a column. And you can set a decision: A:Int > 5.

	id	var1	type	var2	value
▶	4	NewColumn1 (atr1+atr2)	<	NewColumn2 (atr2 asign)	NULL
	5	NewColumn2 (atr2 asign)	>	value	5
*	NULL	NULL	NULL	NULL	NULL

Table 3.2 Decisions Database Table with Value

Value column is a nullable column that only will be full if one of the var1 or var2 columns has "value" controlled by frontend. This last data base table corresponds to a CRUD service so the user can create, read, update and delete decisions as he/she wants.

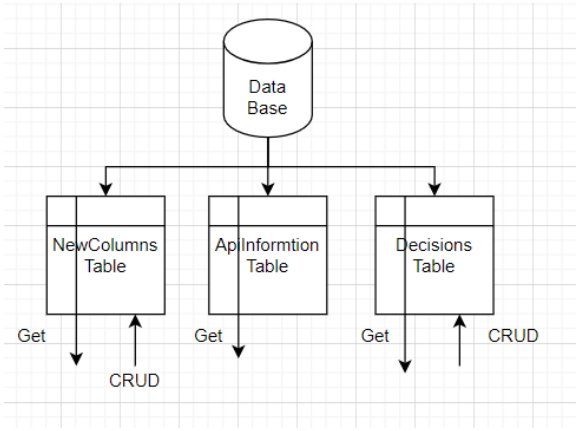


Figure 3.3 Set of Database Tables and Actions Allowed

3.5 HTTP Backend

3.5.1 Configuration Files

The third part of the project is backend; this is the most complex part on the project.

When de backend part was created, the first problem was the CORS policy. "Cross Origin Resource Sharing" CORS was blocking the entire frontend request to backend. There were two big problems caused by CORS. One of them was that the backend didn't have a security and a web configuration class, so these classes were added in following form:

When a request is received CORS check two things, first is that the request sender is a valid sender, in this case WebConfig in AllowOigins has a '*' it means that all origins are allowed, to be more safety it should be the @IP where the frontend is allocated. The second thing checked is that the request has a valid authorization, every request has to have on headers the backend user and password, this was the second problem caused by CORS that not all request had credentials and backend denied the access to Backend.

WebSecurityConfig defines the username and password for basic authentication, and tells that all HttpRequest need it mandatory. But the "antMatcher("/socket).permitAll()" means that all request to the "/socket/" endpoint don't need that. This is because web socket protocol doesn't support headers on the request.

It was solve to adding an interceptor on the frontend that catch every request, add on headers the necessary authorization credentials and send it to Backend.

Figure 9 on annex contains the Http Interceptor.

If the request is sent and Backend responds with an error, interceptor catch it and shows on the screen 'Servei no disponible' with a snackbar.

Once checked this two things, the connection should be safe. But on configure (http: HttpSecurity) method there is a ".crsf().disabled()" it means that this application is now protected against CRSF(Cross-site request forgery) attack. Considering that this Project does not have any personal data and is a local Project we should not be worried about that.

3.5.2 Scheduling and CRON

Before start explaining the business logic on the backend, let's take a look on the simulation of data generation mechanism.

On Kotlin, Java there is a very useful and powerful tool called "@Schedule", this tool can execute whatever the developer wants every X seconds, minutes, hours, days even execute a function every day at 15:00 or every 20th and 21th day of the month etc. Figure 10, 11, 12 and 13 in annex shows 2 example of Schedule tool.

Knowing that the DataSimulatorClass generates a new Object ready to send, every X seconds and store it in data base.

3.5.3 Business Logic

Once explain this kind of things, it's the turn of the business logic itself. The backend architecture is a Controller -> Service -> Repository architecture.

There is 3 controllers, one to get the Api mocked data, another that manage the CRUD decisions systems and the third one receive the request related with the generation, deleting, modifying and reading(getting) about NewColumns (user custom columns). Every controller has a Service, which is the part that makes all the system logic.

ApiDataService is the simplest service, this just get the API Data, and sends it to the controller which will send it to frontend when it asks for it, remember that this is an asynchronous solution.

Decision Service has 4 methods to modify, create, delete and get the data stored in decisions table on data base.

The last service is the most complex class on the project. NewColumnService is the service in charge of get the data from API, this service gets the data already summed or whatever and map it to the correct Object to pass to frontend and checks if the data is passing or not the user decisions stored on decisions table from database. And send it to the frontend. See Table3.1 on Pàg 25 to remember how the NewColum data is stored in database.

Firstly, to calculate the user NewColumns Values we got the data from database and do a several iterations and conditions to get the values summed, subtracted, etc. on the backend, later this

was known as a bad practice because the SQL language does it in a more efficient way, so it was decided to change it and move the NewColumn Values calculation to the database queries. In particular this way had an advantage, it was that angular Mat-table maps the elements by rows and not by columns (explained accurately later), and the frontend did not to do as many works as it does now, this mapping could be done on backend but in this case the frontend needs to do a little map also so to avoid splitting the map was decided to do it all in the frontend. and But this code was not removed at all because on the web socket part there is not able to build queries with SQL language.

To calculate the user NewColumns (A+B), the service calls ApiDataRepository class, this class builds custom queries with the corresponding sign and the columns involved. Let's take a look on that.

In NewColumnsService there is the function "getNewColumnsAndValues()" which gets the information to generate the new columns.

This is how the information for generate new columns is stored in database, what the function does is to call the "manageOperation()" function, this has several functions depending the sign of the operation and this calls ManualRepository Class to build the queries, execute them and receive the data already summed, subtracted, multiplied, divided or assigned from database.

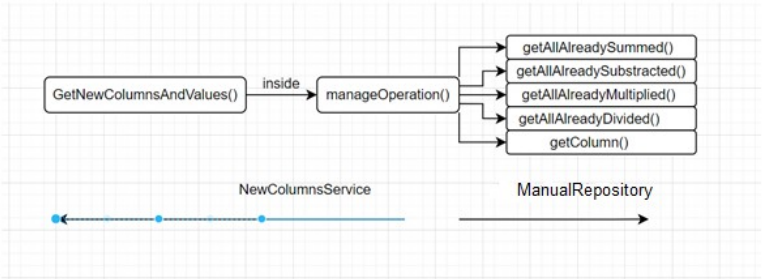


Figure 3.4 Queries Management

This queries returns a list of values, then they are all mapped inside a MutableMap with key corresponding to column name and values to all the array received. If the user wants X columns the map will have X keys with its respective arrays.

Once the values are mapped to the MutableMap, then it's turn to check if the decisions are passing or not, this is done by "setDecisions()" function on NewColumnsService, this function has two parameters, the first one is the mutable map with all the arrays made with ManageOperation() inside and respective keys and the decisions stored in database.

See Table3.2 on Pàg 25 to remember how the Decisions data is stored in database.

Each row contains a primary key generated by the backend, and the 2 columns involved, and if in some column field there is the value "value" that means that the comparison will be done with the number on the value field. In the backen there is an entity called Decision which contains all this properties.

Explained that, the setDecisions function iterates for all keys stored in the MutableMap and tries to find which key is equal to the var1 and var2 field of the decisions parameter also passed. Once the two columns of array are found then each value is compared between the 2 arrays and a decision is set throught the "manageComparison()" function which takes 2 numbers and a sign and determines if its true o false.

This is done for all the decisions passed as parameter. Finally all the decisions arrays with booleans inside are mapped to a MutableMap with key as the decisionId and value as the boolean decision array.

Once done, is called a method in RandomEntityRepository class which takes the ids, and the Id are map inside a generalDto.

GeneralDto is an object which contains all the information that frontend needs. See the Figure 13 on the Annex to see al properties.

Contains a map with keys and the NewColumns added by the user, a map with keys and if the decisions for every newColumn item inside are true o false and the Id. A schema for that, considering that Val1, Val2, etc. are numbers, D1,D2 are booleans, and ids long type:

listValueNames		Decisions		Ids
key1	key2	key1	key2	
Val1	Val1	D1	D1	Id1
Val2	Val2	D2	D2	Id2
Val3	Val3	D3	D3	Id3
Val4	Val4	D4	D4	Id4
Val5	Val5	D5	D5	Id5
Val6	Val6	D6	D6	Id6

Figure 3.5 GeneralDto Scheme

Take into account that key1 in the listValueNames is the column name and key1 in the decisions is the DecisionId in the database. Val1 are values which contains a Double object and D1 are

deicions which contains a boolean. The 3 columns are got seperately but there is one rull that the position 0 of any column corresponds to position 0 of the other columns.

For example, if there is just one column called 'test' which is A+B, there will be one map inside listValueNames which contains as key 'test' and the list the N objects (which are A+B values summed),. If there is also one decision called 'decisiontest' which is A+B column > 3 then there will be just one map inside decisions with key as DecisionId assigned by the backend, comparing each item inside listValueNames item with the condicion proposed , and defining in the value list if the decision is passed or not for each value. To summrize all GeneralDto cointains column names and the values of the columns or the values of the decisions.

3.5.4 JPA Repositories

There is an interesting tool on spring boot, that has to be commented. This tool is the JPA repositories, this is a very powerful tool that makes the queries to database so simple, and there is no necessary to make repository implementations.

This tool consist on import on every repository the spring boot interface CrudRepository<Id type, Object stored>, one done that you can just write the query method as:

- FindById(id:Int)
- DeleteByName(name:String)
- Save(Object)
- FindAll()

And more complex queries:

-FindByOrderByNameAsc(columnDataBaseName:String). "Asc to get it in a ascendant way"

-DeleteByIdByName(id:Int, columnDataBaseName: String)

Id:Int in case that id type is Int but It can be Long, etc.

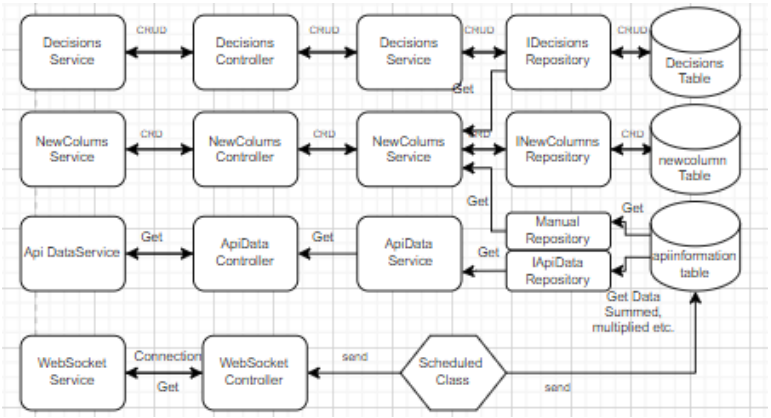
Just writing this to the interface repository, spring boot generates the query internally and sends it to the data base, giving to the system the data ordered.

You also can make custom queries writing on the backend repository @Query (MSQL code to order to database). Not all custom queries are allowed, just the ones which can be made by words, this is why the project has manually connections to database, because the queries has inside the '+','-','*','/' (summed, subtracted, multiplied, divided) operators that JPA doesn't

support. That's because there is a ManualRepository class which made the queries manually. The a custom query and a manual query in the Figures 14 and 15 in annex.

"This.connection()" method defines all the connection properties.

The connection to data base is on Application.properties file, that there is the credentials to make queries.



3.6 All Backend and Database Architecture

In the project the ApiInformationTable is divided in two repositories, first one is JPA and the second one the manual one. Manual repository just needs to access in Apiinformation table because of the '+','-','/','*' operators.

Finally, the last project part is the frontend part. This document doesn't explain every frontend component, just the most important thing, and general comments.

3.6 HTTP Frontend

Before start, it's important to take into account that the frontend component/module architecture is done on that way basically to make the code understandable and easy to work on it.

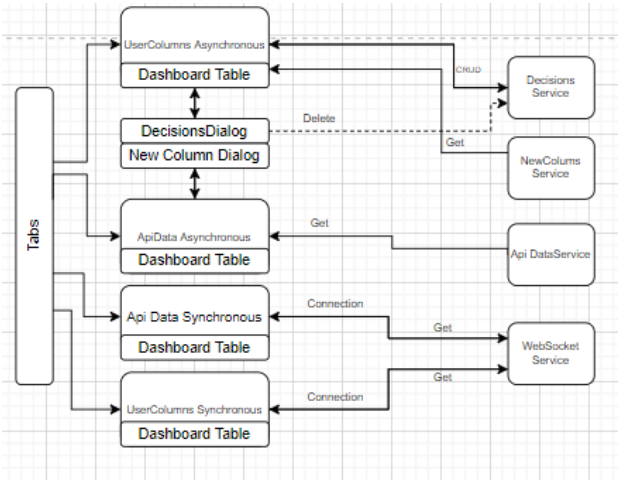


Figure 3.7 Frontend Architecture

3.6.1 Items Object

The most important component on the frontend is the dynamic dashboard table. This component is a dynamic custom mat table which enters an array of Items[] and this carry out the tables with the data printed (see it on Annex). It's important to know how is done the map between the GeneralDto in the Backend and items array in the frontend.

The problem is that mat-table works with rows, in other words there are 2 formulas to fill a table.

Imagine that there is an empty table, this has 2 parts the column names and the column values that you have to fill it.

The first one consist in pass a column Name and an array with the column Values. This can be done for the entire necessary column. Called column way.

The other one consist in firstly pass all the columns name, then an object with one value for the first column, one value for the second column, one value for the third column … one value for the N column. Then when an object has filled all its values to the table it pass to the second object in the same way, so is needed a list with objects which inside has different properties. Called row way.

If you check the GeneralDto on the backend it can be seen that it equals to the column way and It has to be map to the row way, and that is what Items[] object on frontend is.

This was done in this way because the database gives to the backend the values in columns and there is no way to avoid this. Usually it is done by a map function which relates the backend properties with frontend properties automatically but in this case this doesn't works because the properties name depends on the user.

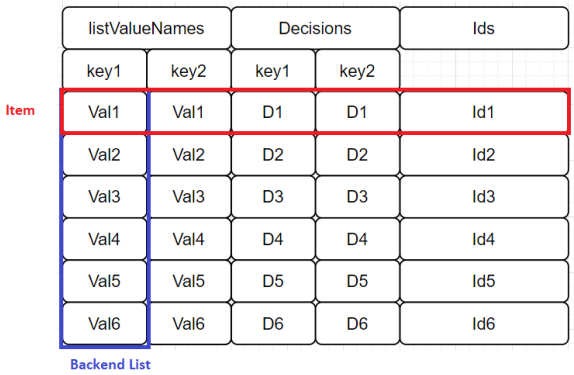


Figure 3.8 Frontend Object Compared to Backend Object

Red line represents an item on the frontend while at the blue object is a list of values in the backend. Each item has values which are the listValueNames(val1, val2) map in properties which its name is the key 1. Then the same for the decisions and finally the id.

See that a there is a vertical conversion to a horizontal conversion.

An item object with the above property would look like:

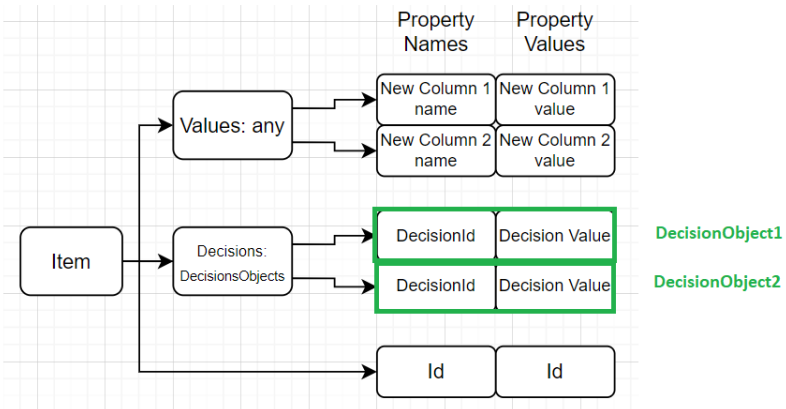


Figure 3.9 Item Properties Structure

This have to be clear because in the fronted Values are typed as any this is because the property name has to be the name entered by the user and each column created will have a different name. Decisions can be a defined Object because it will always be an Id number and a Boolean Value.

3.6.2 Components

So, going from top level (Tabs) and finishing on bot level(Services), let's explain the design.

Each tab is accessed by routing module that redirects to a different component depending the path passed to the URL. See the annex Figure 16 with the Routing component.

Empty path means by default. To navigate through components is needed to instance a router object and call the navigate function. See annex Figure 17 to see an example.

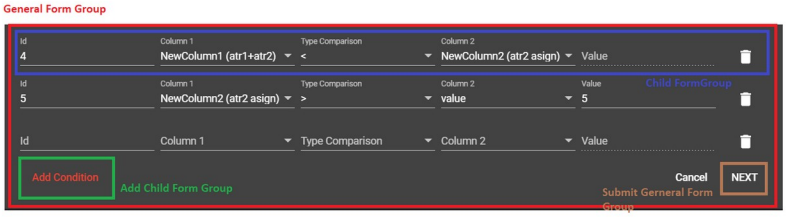
Asynchronous tabs in "ngOnInit()" ask the backend for the appropriate data to print. API Data tab is pointing to api-data.service.ts, Decisions Tab is pointing to new-columns.service.ts to get the correctly data to print on dashboard.

It's easy to think that new-columns-dialog.component.ts and decisions-dialog.component.ts are pointing on their respectively services (new-colum.service and decisions.service.ts) to send the CRUD request to the backend, but it's cleaner for the code to send the dialogs data to the parent component and the parent component will be the one which sends the information to backend. With this way all CRUD methods related with one dialog are on the same component. But there is just one exception on this project and it's that decisions-dialog.component.ts orders to backend when a decision is deleted.

In other words, the user maybe added and removed some of the existing ones, and the frontend should manage which were added and which were removed pointing to different endpoints. But if the dialog tells at the moment that delete button is clicked that this decision is removed. The frontend just have to manage creation decisions.

Moreover, there is two angular tools, used on the code that needs to be commented.

The first of them is FormArray tool, that tool has a FormGroup[] on his inside, and the user can add or delete FormGroups as he/she wants, and it's a dynamically behavior. That FormArray has to be a array from a general FormGroup in this case called "decisionsFormGroup" and NewColumnFormGroup respectively, this is the set of the all information about the decisions and NewColumns created, deleted, updated, and when the dialog is submitted you submit all the data inside the general FormGroup.



3.10 NewColumn Dialog Explaining FormArray

The second is the ForkJoin tool, that tool belongs to “rxjs” libray and allows to the developer to activate several observables together ant this function will not continue until all the observables are done. So this tool is used to be able to create several new columns or several new decisions at the same time.

3.7 WebSocket Backend

3.7.1 Business Logic

Protocol WebSocket was chosen because can sent data bidirectional, but in this project the data just will be sent from backend to frontend. The only data sent from frontend to backend is the connection event.

Also remember that WebSocket is not just an alternative to https, this protocol is build above http, the request and response are under http but offers some different features.

In this project the protocol web socket part is also called synchronous way because when a entity is generated all goes sequentially until the frontend.

This part caused several problems, starting for the configuration, it made that the connection was failing. See annex Figure 18 with an error example

When this part of the project was started, the most information found on the network was about the ebWSocketBrokers, so it was tried to implement one, brokers are a powerful tool if you need to manage some WebSocket endpoints and each endpoint has a determinate flow. But this was not the project case so this were like overtaking the necessities that project needs.

Broker is a tool over STOMP protocol (Simple Text Orientated Protocol) over Http, the main feature that broker has is to do as intermediate between 2 applications. In a few words imagine that you as user wants to receive the data that 3 users are sending to the broker (broadcasting) so tell to the broker that you are going to subscribe on it and broker opens a port to you to send the arriving data from the other 3 users.

That was too much for this application that just needs one channel to the frontend.

Knowing that the search was about just one Websocket channel without STOMP protocol.

Then was tried the WebSocket configuration with SockJS(). SockJS tries to establish a connection with a native WebSocket but if it is not able to then change the protocol for someone which front accept. But this was discarded quickly because it was need a WebSocket configuration at 100%.

It was already thought that WebSocket protocol and Http protocol could not go into the same backend port so it was tried to open a 2nd port in the backend. To do this is needed something like the figure 19 in the annex.

The problem here was that TomcatEmbeddedServletContainerFactory it was suppose to be a class inside this import:

implementation("org.springframework.boot:spring-boot-starter-tomcat:1.2.4.release")

But in my case this class was not detected by this Gradle import so it was decided to leave this solution.

At that point, it was started a 2nd project with just the socket part, that idea was the first one which worked with just the WebSocket part went well and it could be started the web socket logic implementation. So in this moment we got 2 projects one at port 8080 and the other one in the port 10101. This was a good partial solution because the implementation could be done.

But finally realized why there not could be the http connection and the WebSocket connection in the same port, the thing is that Google Chrome does not shows the WebSocketErrors as it should be but Microsoft Edge does. So for no reason, suddenly was tried to make the WebSocket connection through Microsoft Edge and Edge Showed the error more specifcly. See the annex Figure 20 with an example.

This was because the ws request went with headers and WebSocket request cannot contains header, so what it was tried is to embed the credentials into the request in the following way:

Request: ws://username:password@localhost:10101/socket

This finally worked and the 2 projects could be put together as one project.

But in terms of security you are exposing the username and password to the net world, in this application there is no problem because there is no important personal data but is not the best practice. To avoid WebSocket request to pass the authentication it was added to WebSecurityConfig see the sentence on the figure 21 on annex.

As it was explained before this disables CSRF adds a CORS and http basic authentication, but the “antMatcher” excludes the entire request to the “/socket” endpoint to be authenticated. So at this point the 2 projects could be put as one project.

The all scheme of the web socket part is:

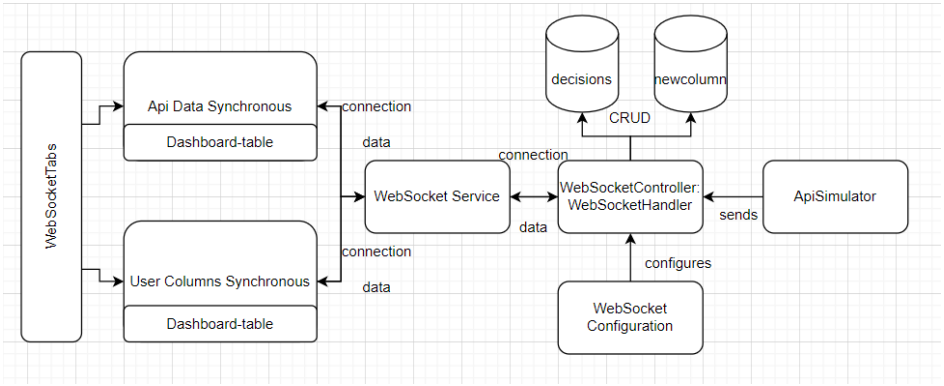


Figure 3.11 WebSocket Full Architecture

3.7.2 Configuration Class

WebSocket Backend is quite simple compared to http backend, first of all there is a class which configures the webSocket properties. This class is the webSocketConfig this class implements the WebSocketConfigurer interface.

See the all class implamentation in Annex Figure 22.

This method implements addHandler, as parameters are passed a class which implements WebSocketHandler interface and the path/endpoint that the handler will be listening to manage the messages.

SetAllowedOrigins(*) is for receive the request not depending in their origin. WebSocketHandler needs to be a bean because just can be a one WebSocketHandler because the sessions needs to be stored for DataSourceSimulator as it was explained before. Pag.24

3.7.3 Controller

WebSocketController class implements WebSocketHandler which has 5 member methods: “afterConnectionestablished()”, “handleMessage()”, “afterConnectionClosed()”, “handleTransportError()”, “supportsPartialMessage()”.

“SupportsPartialMessage()” returns a Boolean depending if the WebSocket can afford segmented messages or all of them have to be inside one package. In our case is false.

The other methods are event listeners activated when a connection is established, there is a message entering, a remote socket has closed or controller is receiving errors.

When a connection message arrive, is launched the “afterConnectionEstablished()” method which stores the session in a WebSocketSession array. When this connection is close “afterConnectionClosed()” remove this session.

There is a function called “sendMessage(msg: RandomEntity)” which is called from DataSource Simulator when a value is generated this send message to all sessions(in our case there will always be just one session), but before that transforms the object to a Gson object then to a string because WebSocket only supports BinaryMessage or TextMessages. This function is able to take the decisions and new columns from database and manipulate it as is needed (summing, multiplying, etc.) send the results to the frontend, this kind of data will be shown in the User Columns Synchronous.

The data treated as a user wants are sent to frontend inside SocketNewColumnsAndDecisions entity, this contains the following properties:

- ListNewColumns is an array list which contains as many Objects as newColumns created by user. If the user wants two new columns (A+B) and (A-B), then the size of this array will be 2. ValueName is an object with 2 string properties Value and Name, name contains the column name and value its respective value.

- Decisions goes in the same way that listNewColumns but with the decisions objects.

- Id is the time when the entity is created in milliseconds

See annex figure 23 the SocketNewColumnsAndDecisions Class.

All are initialized but this are values the default vales in case that the new object is not initialized.

3.7.4 Frontend

Message connection is sent to the backend when the user clicks on the API Data Synchronous tab, then the user starts to receive data create from DataSource.

The event connection is launched through a subscription object. A subscription object is an event handler which is triggered when some event occurs. A subscription object has several observables inside and each observable has an observer.

- The observer catches the data when this occurs. It has 3 methods, data event, error event info and complete event similar to a close event
- The observable manage the info when observer tells which kind of event is
- The subscription is the part which manages the data in the angular component.

See an schema of how it works in the annex Figure 24

Do not confuse this with a subject, a subject is an observable and observer together, they do not need to be defined.

See a subject schema on annex Figure 25

The connection message sends a Http request suggesting a protocol change and if the backend approves that returns a 101 with switching protocols message.

See in the annex Figure 26 an example of a succeful connection message.

After that the connection will be under WebSocketProtocol, until the frontend changes the tab and the connection is closed.

As a result that the web socket has a connection then backend can start to send data to the frontend, once data is received to the frontend in this case this don't need to be mapped horizontally (as was explained in Http part) because the data already arrives in the correct way, and it's just necessary a little map to be able to add it into the Items[] array.

To make the different observations about the time between the data is generated and is printed to the frontend it was added a clock which indicates how much time pass and a difference column which indicates the amount of difference of time between one object and the previous one.

4. Results

The final architecture scheme has a look like this.

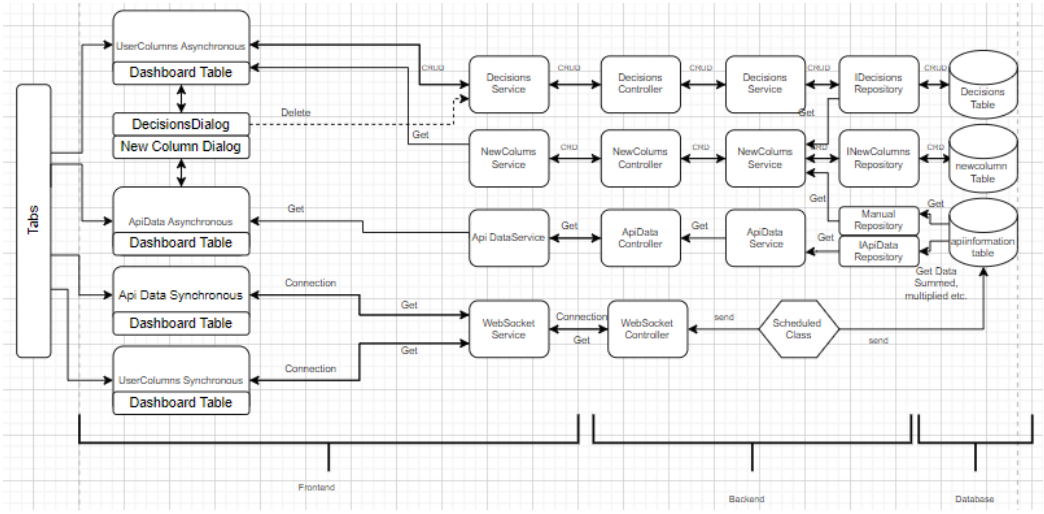


Figure 4.1 General architecture View

First of all, there are some things to clarify before start, first is that an arrow in just one direction means that useful data just can go in this direction.

But let's summarize all the architecture explained on the previous parts.

On the database side there are 3 tables, 1 for APIData arrived from DataSource, another one to store the NewColumns that user wants to set and the last one the information needed to set the custom user decisions.

To access in this database there are 4 repositories, the first of them is IDecisionsRepository which can make a CRUD service to Decisions Table, INewColumnsRepository which can make a CRUD service to NewColumn Tables. ApiInformtion table has two repositories able to access on data inside, the Manual Repository which is used to launch the queries for sum, subtract, multiply or divide columns and IApiDataRepository which is used just to get data from there.

On the service part there also 3 service, Decisions Service just make de necessary functions to ability the CRUD service, as well as the NewColumnsService but this service also gets the values from NewColumn table, Decisions table to calculate the new values wanted by user and it gives the data calculated inside a MutableMap together with Id data, it can access to all 3 repositories. And finally the ApiDataService which just take the data from ApiInformation table in database.

Finally there are 3 controllers which each one with its respective Service which are the doors to the frontend. The controllers have a CORS policy which denies the request with no basic http

authorization header. This is sett in the WebConfig and WebSecurityConfig class. This 3 controllers has 3 service on the frontend which makes the request.

Then there are 4 tabs, each tab has a component and inside this component there is a dynamic mat-table, dynamic mat-table means that giving a name columns and its respective values it will create a different table depending on that values, also remember that mat-table needs a list of values and the names in which column goes every property . APIData Asynchronous shows the API data through Http protocol, UserColumns Asynchronous shows the new data expressed by columns that user wants. The Synchronous tabs just make the same as asynchronous but through web Socket Protocol. The two dialogs to create and modify NewColumns and Decisions are located just in the asynchronous tabs, remember that decisions dialog has an special feature (discontinuous line) which can delete decisions inside it and the dialog is the component that tells to the service to send a DELETE request which the corresponding id. This was made in this way because instead of this it should be the parent component that has to compare with the initial decisions which were deleted, which were modified, which were created and the code goes a little overloaded.

Then the DataSource part is a class on backend called DataSourceSimulator (previously called ScheduledClass) that every X milliseconds create an object which is stored in database, and sent to the frontend through WebSocketController class.

To send the message, it is parsed to a JSON object and then converted to a TextMessage because is one of the typed objects supported by WebSocketMessage Interface. WebSocketController has a configuration class which tells the enabled endpoints for the connection. This WebSocket connection is not over STOMP and does not use a broker tool.

WebSocketController also manage the entering connections, launching one of the override functions depending the event arrived.

WebSecurityConfig also allows to WebSocket connection message without authorization request, because of the WebSocket protocol cannot contain headers in the request.

In frontend WebSocket connection is done when the user enters to the one of Synchronous tabs, and when the user leaves is closed.

Synchronous side on web Socket does not need a very huge mapping as asynchronous side because the object arrives as mat-table wants with names and properties.

The differences between protocols are that:

Http protocol is progressively slower when the number of new decisions and NewColumns increase, the point that protocol crashes is when there is approximately 10 NewColumns and 10 decisions, this is because the front mapping is complex and on every request there is 100 object to load. With memory cache work this could be improved.

If http protocol has a high rate of request/second and high object generation rate, the system is overloaded and there is some duplicated data, the maximum request/second to make the system work is approximately 1 request/second. This is because value data goes on the response message and the channel is spending resource to send the request petitions which does not have any substantial data.

WebSocket protocol does not have these problems because objects arrive one by one and can support a lot of package rate. Taking a look at the system flooded generating 1 object to send every millisecond, with a clock on frontend we got that sending in this rate the frontend got 1000 objects every 16.83sec. So doing some calculations:

$$\frac{1000}{16.83} = 59.41 \frac{paq}{s} \rightarrow \frac{59.41 * 0.001\, s}{milisecond} = 0.05941\ paq/milisecond$$

It's important to express it in milliseconds because the generation rate is indicated in milliseconds.

Also say that on the UserColumns Synchronous tab every object used to have a delay of 6 or 7 milliseconds from the generation to the frontend and on the APIData Synchronous tab used to be 0 or 1 milliseconds, this is basically due to the backend need to calculate the new values and make database queries.

Moreover, another interesting behaviour is that if the system is flooded and the backend is suddenly stopped, the frontend keeps showing new data some seconds later, this is because WebSocketHandler has a buffer to store the messages if they cannot be sent, and this buffer is not stopped or cleaned when the backend is stopped. I guess that the used buffer is on the emissary side because the AngularWebSocket object has a function called "this.amountBuffered()" which tells the user how much data is waiting on the buffer, and this function is returning 0 all time.

5. Budget

Considering that the project doesn't have a prototype itself. The cost of will be basically the amount of hours dedicated to develop this tool. Is going to be considerate that the project is submitted on the internet and their costs. As a junior developer, the amount of work done from the beginning at now have the following cost

Work	Hours	Cost/ hour	Total Cost
Frontend	75	9	675
Backend	100	9	900
Bug Fixing	30	9	270
Architecture	75	9	675
Total	280		2520

Table 5.1 Project costs

Regarding the web costs, this is the average cost for a web site including, first investment and annually costs. Considering that if this web site someday arrives on the public internet, just should paid the Domain, Certificate a Hosting and a maintenance.

Domain Name	$0.95 - $12
SSL Certificate	$0 - $1,500
Website Hosting	$24 - $10,000
Style or Theme	$2,000 - $15,000
Responsive Design	$3,000 - $25,000
Interactive Multimedia	$250 - $10,000
Content Management System (CMS)	$2,000 - $25,000
Ecommerce Functionality	$2,000 - $25,000
Database Integration	$2,000 - $25,000
Pages (1 to 250 pages)	$1,000 - $10,000

Table 5.2 Average year cost of a web application

Supposing that there is one developer working 4 hours a day on improve the product, the annually costs on that would be:

Costs	Price/hour	Hours/year	Total Price
Domain	-	-	12$
SSL Certificate	-	-	200$
Hosting	-	-	30$
Database	0.10 / Milion queries		0$
Developer	12$	240	12672$/year
Total			12914$/year = 850€/month

Table 5.3 App maintenance and improving cost

6. Conclusions and future development:

In this chapter, the results obtained are commented, and the way to get this results as well. No the satisfaction with the results itself, but the knowledge acquired developing this app.

The results obtained on this projects are satisfactory, the project do all the things functionalities expected at the beginning.

Another conclusion is that making a general architecture is complex, a lot of things has to be taken in account about the others parts of the project. One of the things that make the project in trouble was the mutability it has during his development. Is important to define the acceptance criteria well before start developing and don't change it, because change how thing works on one site affects on all project scheme.

Is a good practise on frontend that the minimum components talks with the services, to get data because in that way the data is controlled and not duplicated.

A very important thing when someone develops is to work in frontend independently from backend and vice versa, for development speediness, understandability, elegancy, clean code, comfort etc. Even is grateful be developing the frontend knowing that the backend is working good and you can think of it as a mocked one.

Is more important to apply the good practises on backend and be strict on it than on frontend which allows the developer more flexibility. It doesn't mean not to follow the principles of good architecture on frontend.

Future releases should include the following functionalities:

- Error management.
- Allow historical operations as average, variance of the last X samples
- Combined operations with more than 2 elements.
- Cache work to reduce the request size
- Charts Feature
- Improving the frontend mapping.
- Connection to a real API

Overall I'm very happy with the result I think that is a very useful tool in my personal project, and it is the first step that sometimes is the most hard to do because there are a lot of new things and now I can focus on improve the functionalities and not to work in project architecture.

Bibliography:

[1] Maxim Shafirov (JetBrains), Roman Elizarov (JetBrains),William R. Cook (University of Texas at Austin),Jeffrey van Gogh (Google) "Official Kotlin Documentation" https://kotlinlang.org/docs/home.html. [Accessed during all project]

[2] " Angular Material Official documentation" https://material.angular.io/components/categories. [Accesses during all project]

[3] "Angular Cli Official Documentation" https://angular.io/cli. [Accessed during work package 1]

[4] "MySQL official documentation" https://dev.mysql.com/doc/mysql-getting-started/en/ [Accessed during work packages 1, 4]

[4] Ascari Romo, "Tareas con spring scheduler" https://windoctor7.github.io/Tareas-con-Spring-Scheduler.html [accessed during work package 3]

[5] Alberto C. Ríos, "Accessing Data with JPA" https://spring.io/guides/gs/accessing-data-jpa/ [Accessing during work package 4]

[6] Kristiyan Kostadinov, "FormArray" https://angular.io/api/forms/FormArray

[7] Brian Troncone Oficial Documentation of rxjs https://www.learnrxjs.io/, [Accessed during all project]

[8] Brian Troncone "forkJoin" https://www.learnrxjs.io/learn-rxjs/operators/combination/forkjoin [Accessed during work package 5]

[9] MDN Contributors "Cross-Origin Resource Sharing (CORS)" https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS [Accessed during work package 2]

[10] ASFO "Usando HTTP Interceptors en Angular" https://asfo.medium.com/usando-http-interceptors-en-angular-a665ebe6350b [Accessed during work package 2]

[11] **Sandeep Jandhyala** "10 Microservices Best Practices for the Optimal Architecture 1Design" https://www.capitalone.com/tech/software-engineering/10-microservices-best-practices/ [Accessed during all project]

[12] AngularNYC – Working with WebSockets in Angular apps – Yakov Fain https://www.youtube.com/watch?v=mqFajN7kE4s&t=862s [Accesed during the webSocket part]

[13] Spring docs https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/websocket.html

[14]Creating custom listeners in kotlin https://www.youtube.com/watch?v=_a3axw2LF0A

Appendices (optional):

Helping Figures:

```
dependencies {
    implementation( dependencyNotation: "org.springframework.boot:spring-boot-starter")
    implementation( dependencyNotation: "org.springframework.boot:spring-boot-starter-data-jpa")
    implementation( dependencyNotation: "org.jetbrains.kotlin:kotlin-reflect")
    implementation( dependencyNotation: "org.jetbrains.kotlin:kotlin-stdlib-jdk8")
    implementation ( dependencyNotation: "org.webjars:bootstrap:3.3.7")
    implementation ( dependencyNotation: "org.webjars:jquery:3.1.1-1")
    implementation( dependencyNotation: "org.springframework.boot:spring-boot-starter-security")
    implementation( dependencyNotation: "org.springframework.boot:spring-boot-starter-web")
    implementation( dependencyNotation: "org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.1")
    implementation( dependencyNotation: "com.fasterxml.jackson.module:jackson-module-kotlin")
    implementation ( dependencyNotation: "org.springframework.boot:spring-boot-starter-websocket")
    implementation( dependencyNotation: "com.google.code.gson:gson:2.8.6")
    runtimeOnly( dependencyNotation: "mysql:mysql-connector-java")
    implementation( dependencyNotation: "org.jetbrains.kotlin:kotlin-reflect")
    testImplementation( dependencyNotation: "org.springframework.boot:spring-boot-starter-test")
}
```

Figure 1 Backend Dependencies

```
"dependencies": {
  "@angular/animations": "~11.2.1",
  "@angular/cdk": "^11.2.6",
  "@angular/common": "~11.2.1",
  "@angular/compiler": "~11.2.1",
  "@angular/core": "~11.2.1",
  "@angular/forms": "~11.2.1",
  "@angular/material": "^11.2.6",
  "@angular/platform-browser": "~11.2.1",
  "@angular/platform-browser-dynamic": "~11.2.1",
  "@angular/router": "~11.2.1",
  "@types/chart.js": "^2.9.32",
  "bootstrap": "^5.0.1",
  "canvasjs": "^1.8.3",
  "chart.js": "^2.9.4",
  "ng2-charts": "^2.4.2",
  "rxjs": "~6.6.0",
  "tslib": "^2.0.0",
  "zone.js": "~0.11.3"
},
```

```
"devDependencies": {
  "@angular-devkit/build-angular": "~0.1102.1",
  "@angular/cli": "~11.2.1",
  "@angular/compiler-cli": "~11.2.1",
  "@types/jasmine": "~3.6.0",
  "@types/node": "^12.11.1",
  "codelyzer": "^6.0.0",
  "jasmine-core": "~3.6.0",
  "jasmine-spec-reporter": "~5.0.0",
  "karma": "~6.1.0",
  "karma-chrome-launcher": "~3.1.0",
  "karma-coverage": "~2.0.3",
  "karma-jasmine": "~4.0.0",
  "karma-jasmine-html-reporter": "^1.5.0",
  "protractor": "~7.0.0",
  "ts-node": "~8.3.0",
  "tslint": "~6.1.0",
  "typescript": "~4.1.2"
}
```

Figure 2 Frontend Dependencies

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/defaultObjects
spring.datasource.username=root
spring.datasource.password=root
```

Figure 3 Application Properties

```
class WebSocketControllerListener(sessions: MutableList<WebSocketSession>): DataSourceListener{

    @Autowired
    lateinit var dataSourceSimulator: DataSourceSimulator

    init{
        this.dataSourceSimulator.addListener(this)
        this.dataSourceSimulator.release()
    }

    override fun onReleaseListeners(entity: RandomEntity){
        this.sendMessage(entity)
    }

    fun sendMessage(entity: RandomEntity){
        //Sends to all sessions
    }
}
```

Figure 4 WebSocketControllerListener Class

```
private val listener = WeakReference<DataSourceListener>( referent: null)
```

Figure 5 WeakReference Example Class

```
fun release() {
    this.listener.get()?.onReleaseListeners(this.dbEntity)
}
```

Figure 6 Release Listener Function

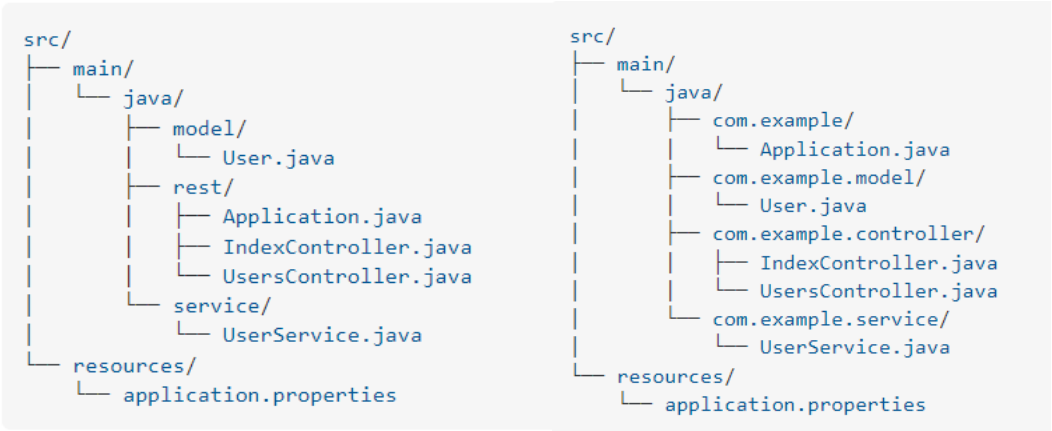


Figure 7 UserService Outside and Inside of Compiling Scope

```
@Scheduled(fixedRate = 10000)
fun generateRandomValue() {
    val entity = RandomEntity(
        floor(Math.random() * 10),
        floor(Math.random() * 10),
        floor(Math.random() * 10),
        currentTimeMillis()
    )
    this.webSocketController.sendMessage(entity);
    this.saveInformation(entity)
}
```

Figure 8 Generate Random Value to Sent to Frontend and Store on Data Base

```
intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
  let req = request;

  req = request.clone({
    setHeaders: {
      Authorization: `Basic ${btoa('isaacolle:isaacolle')}`
    }
  });

  return next.handle(req).pipe(
    catchError((e: HttpErrorResponse) => {
      if (e.error) {
        this.snackBar.open('Servei no disponible', 'Oculta', { duration: 3000 });
      }
      return throwError(e);
    })
  );
}
```

Figure 9 Frontend Http Interceptor

```
@Scheduled(fixedRate = 1)
fun generateRandomValue() {
    val entity = RandomEntity(Math.floor(Math.random()*10), Math.floor(Math.random()*10), Math.floor(Math.random()*10),
    webSocketHandler.sendMessage(entity)
    this.saveInformation(entity)
}
```

Figure 10 @Scheduled Example

```
@Scheduled(cron = "\${tfg.cron.start}")
fun generateRandomValue() {
    val entity = RandomEntity(Math.floor(Math.random()*10), Math.floor(Math.random()*10), Math.floor(Math.random()*10),
    webSocketHandler.sendMessage(entity)
    this.saveInformation(entity)
}
```

Figure 11 CRON Example

The @Schedule decorator allows you to declare a CRON or a fixed rate, a variable initialized on properties. “FixedRate” launches the function each X milliseconds.

```
tfg.cron.start= */10 * * * * ?
```

Figure 12 CRON Property in Application Properties

```
data class GeneralDto(
    var listValuesName: MutableMap<String,MutableList<Double>>,
    var decisions: MutableMap<Long, MutableList<Boolean>>,
    var id: MutableList<BigInteger>
)
```

Figure 13 GeneralDto Properties

```
@Query( value: "SELECT c.name FROM NewColumn c")
fun findNames(): MutableList<String>
```

Figure14 Custom Query Example

```
private fun executeCustomQueryReturningDoubleList(query: String, operation: String): MutableList<Double> {
    var stmt: Statement? = null
    var resultset: ResultSet? = null
    this.connection()
    val result: MutableList<Double> = emptyList<Double>().toMutableList()
    try {
        stmt = conn!!.createStatement()
        resultset = stmt!!.executeQuery(query)
        while(resultset!!.next()){
            result.add(resultset.getDouble(operation))
        }
        return result
    } catch (ex: SQLException) {
        ex.printStackTrace()
        return emptyList<Double>().toMutableList()
    }
}
```

Figure 15 Manual Connection Example

```
path: '',
component: HomeSessionComponent,
children: [
  {
    path:'apiDataAsynchronous',
    component: ApiInformationComponent,
  },
  {
    path:'userColumnsAsynchronous',
    component: DecisionsComponent
  },
  {
    path:'apiDataSynchronous',
    component: ApiInformationSynchronousComponent
  },
  {
    path:'userColumnsSynchronous',
    component: DecisionsSynchronousComponent
  },
  {
    path:'coinMarketCapApi',
    component: CoinmarketcapApiComponent
  }
]
```

Figure 16 Home Routing Component

```
onClickTab(event: MatTabChangeEvent): void {
  this.router.navigate([this.navigate[event.index].link], {relativeTo: this.activatedRoute});
}
```

Figure 17 Navigate Function

```
▶ WebSocket connection to 'ws://localhost:10101/socket' failed:
▶ Event {isTrusted: true, type: 'error', target: WebSocket, currentTarget: WebSocket, eventPhase: 2, …}
▶ CloseEvent {isTrusted: true, wasClean: false, code: 1006, reason: '', type: 'close', …}
```

Figure 18 WebSocket Configuration Error

```java
@Configuration
public class EmbeddedTomcatConfiguration {

    @Value("${server.additionalPorts}")
    private String additionalPorts;

    @Bean
    public EmbeddedServletContainerFactory servletContainer() {
        TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFa
        Connector[] additionalConnectors = this.additionalConnector();
        if (additionalConnectors != null && additionalConnectors.length > 0) {
            tomcat.addAdditionalTomcatConnectors(additionalConnectors);
        }
        return tomcat;
    }

    private Connector[] additionalConnector() {
        if (StringUtils.isBlank(this.additionalPorts)) {
            return null;
        }
        String[] ports = this.additionalPorts.split(",");
        List<Connector> result = new ArrayList<>();
        for (String port : ports) {
            Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol
            connector.setScheme("http");
            connector.setPort(Integer.valueOf(port));
            result.add(connector);
        }
        return result.toArray(new Connector[] {});
    }
}
```

Figure 19 Open Two Ports Through Tomcat

```
▶ WebSocket connection to 'ws://localhost:1010 web-socket.service.ts:15
1/socket' failed: Error during WebSocket
handshake: Unexpected response code: 404
                                        decisions-synchronous.component.ts:24
▶ Event {isTrusted: true, type: 'error', target: WebSocket, currentTarget: W
ebSocket, eventPhase: 2, …}
                                        web-socket.service.ts:26
▶ CloseEvent {isTrusted: true, wasClean: false, code: 1006, reason: '', typ
e: 'close', …}
>
```

Figure 20 WebSocketHeader Failing

```kotlin
override fun configure(http: HttpSecurity) {
    http.csrf().disable().cors().and().httpBasic()
        .and().authorizeRequests()
        .antMatchers( ...antPatterns: "/socket").permitAll().anyRequest().authenticated()
}
```

Figure 21 WebSecurityConfig

```
@Autowired
lateinit var webSocketHandler: WebSocketController;

override fun registerWebSocketHandlers(registry: WebSocketHandlerRegistry) {
    registry.addHandler(webSocketHandler, ...paths: "/socket").
    setAllowedOrigins("*")
}
```

Figure 22 WebSocketConfig

```
data class SocketNewColumnsAndDecisions (
    var listNewColumns: MutableList<ValueName> = emptyList<ValueName>().toMutableList(),
    var decisions: MutableList<DecisionObject> = emptyList<DecisionObject>().toMutableList(),
    var id : Long = Long.MAX_VALUE
    )
```

Figure 23 Socket Object Sent to Frontend

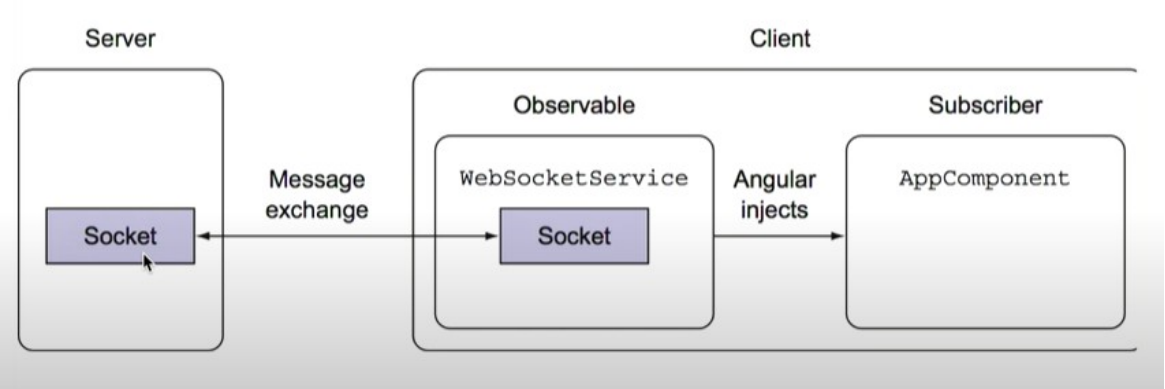


Figure 24 Client Observable Scheme

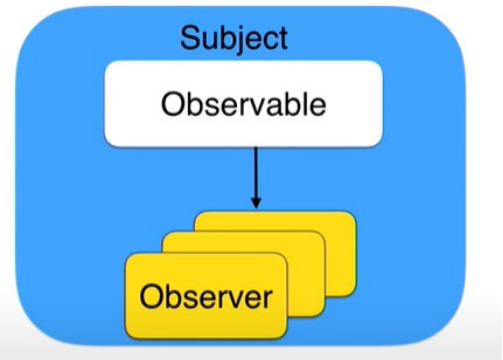


Figure 25 Rxjs Subject

Solicitar URL: ws://localhost:4200/sockjs-node/098/izouajnd/websocket

Método de la solicitud: GET

Código de estado: 101 Switching Protocols

Figure 26 Switching Protocols WebSocketMessage

Demo:

In this appendix is going to be include a little technical demo to see and understand what the project does for an user.

This demo is done in local so the URL is http://localhost:4200. Let's enter to this web page.

The first page showed is a tabs where inside the first tab there is this table:

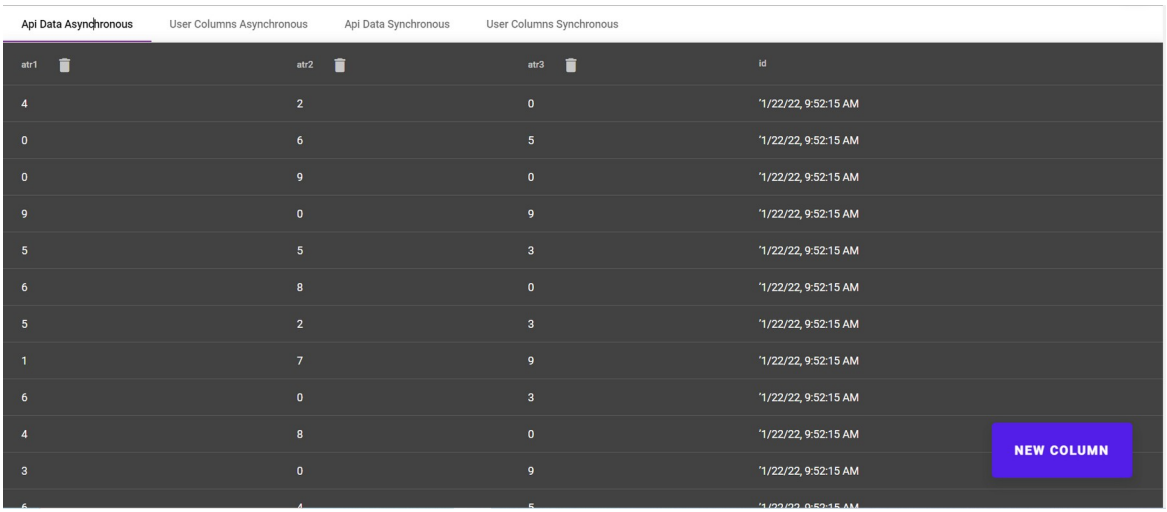


Figure 27 Api Data dashboard table

This table is not modifiable, and indicates to the user all the data received by the data source, with the data timestamp transformed to hh:mm:ss.

And the other button opens another dialog which allow to the user to create an Array of NewColumns Objects (or just one object).

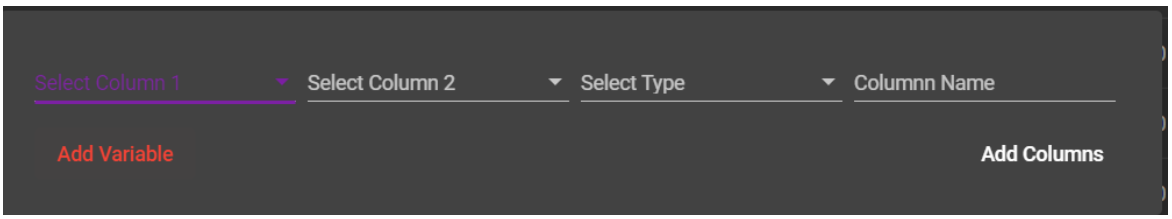


Figure 28 New Column Dialog

On the drop down there is a list with all the columns on APIData dashboard table, a user can choose whatever he/she wants. On select type there is a drop down with Sum, Subtract, multiply, division and assign. "Asign" type just take the column on the first select column drop down and copy it. All this variables constitute a NewColumn Object.

A user can click on add variable to create more than one NewColumn Object and submit all with Add Columns button together.

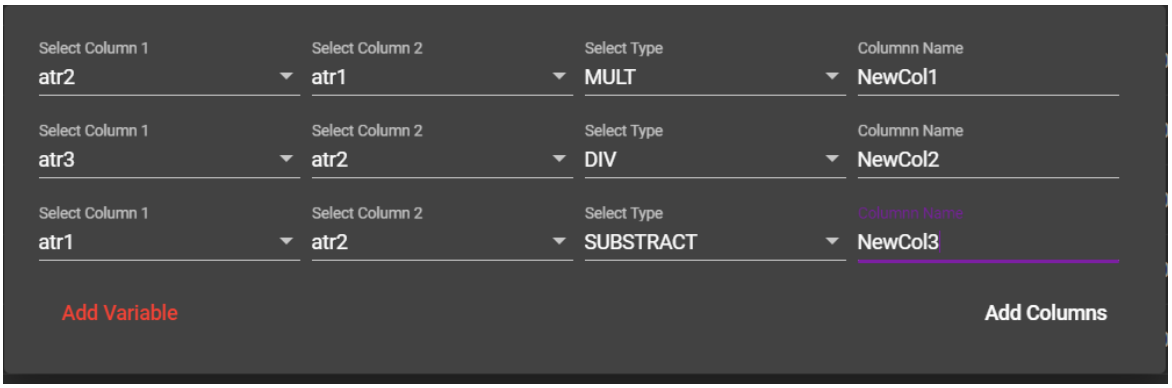


Figure 29 New Column Dialog with several objects.

Once submitted the dialog closes. The user will see the columns created on the other tab. User Columns and

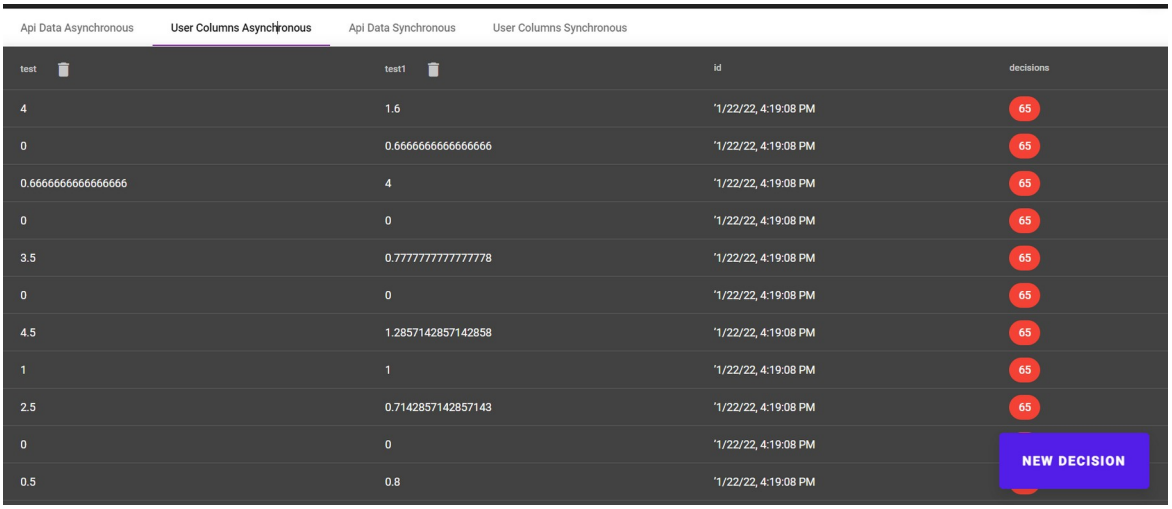


Figure 30 User Columns Tab

The user can delete each column with the trash button located next to column name.

And the other button opens the decision dialog, it allows to the user, that the system will notify to the user if the decision is complied or doesn't. Let's take a look.

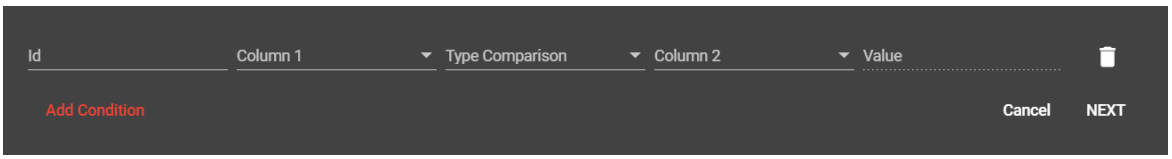


Figure 31 Decisions Dialog

The id field is read only, explained later why, the column one is a dropdown with all the UserColumns dashboard column names, type comparison is '<', '>', '=', and column two has the column names as well.

For example if the user submits one decisions like {Column1: NewCol1, Type Comparison: '>, Column 2: NewCol2}, then the system will calculate when the NewCol1 is bigger than NewCol2.

On Column 1 and Column 2 dropdowns there is a value called 'value', if a user choose this option, the value field will be triggered, so the user have to write some number on it and the comparison will be between the column chosen and the value chosen.

For example, a decision like {Column1: NewCol1, Type Comparison: '=', Column 2: Value, Value: 5}, will tell to the user when NewCol1 is equals to 5.

Add condition allows to the user to submit more than one decision at the same time.

Figure 32 Decisions Dialog with Several Set.

Trash button is to delete decisions. Then if the user submits the dialog the decisions will be created. And the dashboard will look like this.

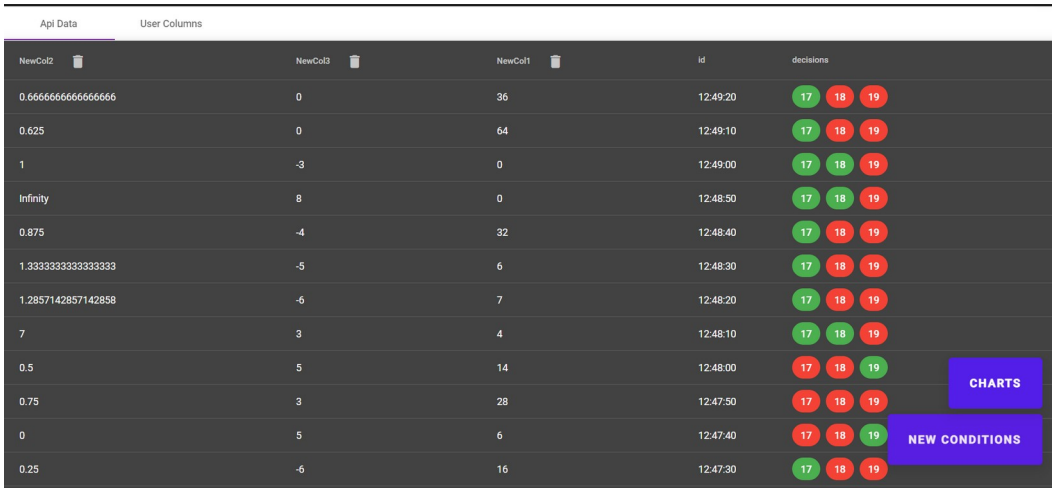


Figure 33 UserColumns Tab with Decisions Columns Filled

Decisions columns has been filled of Mat-chips, green mat-chip means that the condition is passing as true an red as false. Each mat-chip has a number, this is the decision id given by the backend. So the user can reopen the dialog and will the de decisions created before with id.

Figure 34 New Decisions Dialog with Ids

A user can modify and delete decisions, but to translate it to the data base this dialog always have to be submitted after a modify or delete, if don't the changes will not apply.

The 3rd tab shows the same data as ApiDataSynchronous data but in this case the data is sent through WebSocket Prototocol.

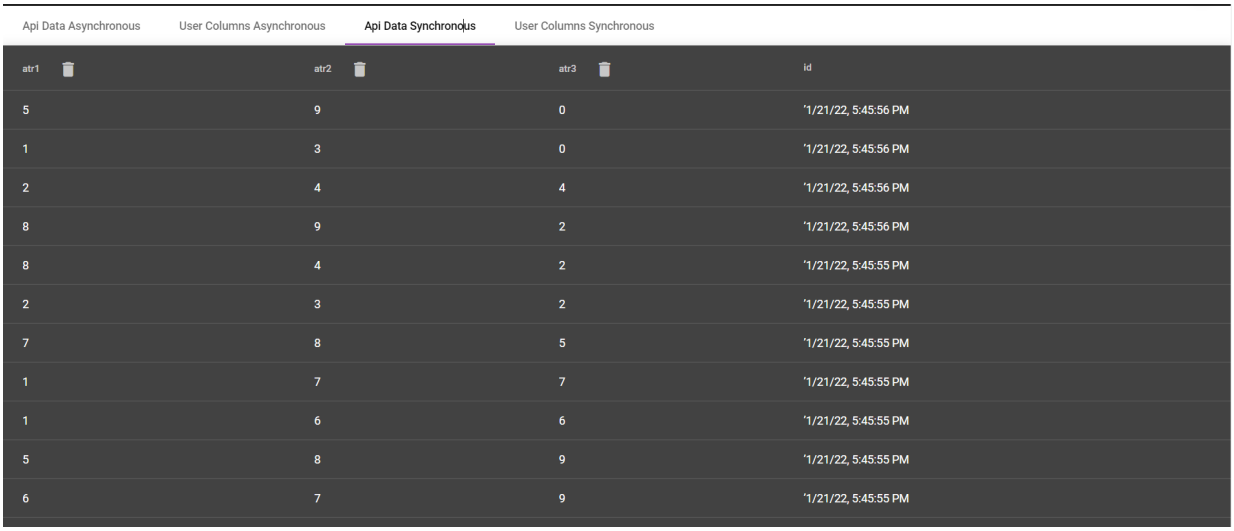
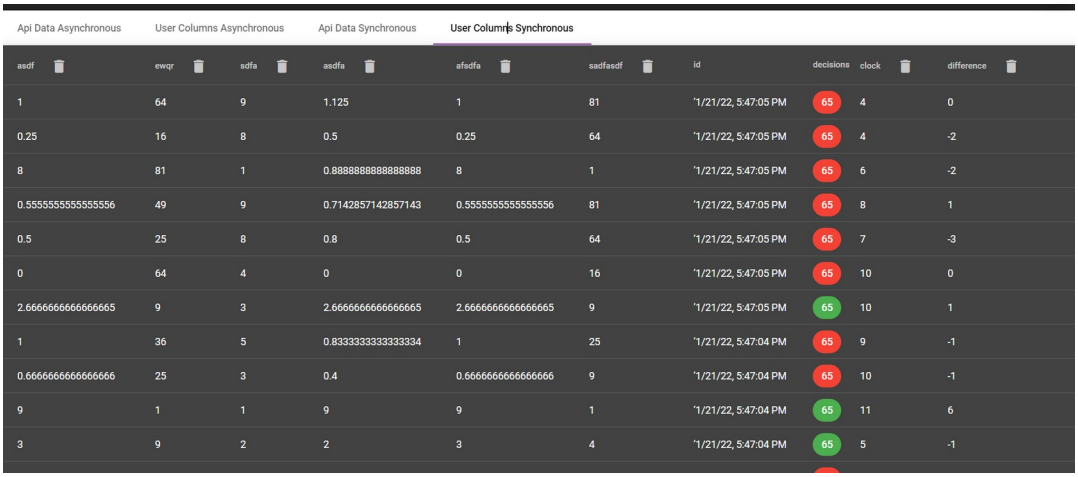


Figure 35 Api DataSynchronous Tab

And finally the 4th tab shows the same data as UserColumns Synchronous but also with web socket protocol.



asdf	ewqr	sdfa	asdfa	afsdfa	sadfasdf	id	decisions	clock	difference
1	64	9	1.125	1	81	'1/21/22, 5:47:05 PM	65	4	0
0.25	16	8	0.5	0.25	64	'1/21/22, 5:47:05 PM	65	4	-2
8	81	1	0.8888888888888888	8	1	'1/21/22, 5:47:05 PM	65	6	-2
0.5555555555555556	49	9	0.7142857142857143	0.5555555555555556	81	'1/21/22, 5:47:05 PM	65	8	1
0.5	25	8	0.8	0.5	64	'1/21/22, 5:47:05 PM	65	7	-3
0	64	4	0	0	16	'1/21/22, 5:47:05 PM	65	10	0
2.6666666666666665	9	3	2.6666666666666665	2.6666666666666665	9	'1/21/22, 5:47:05 PM	65	10	1
1	36	5	0.8333333333333334	1	25	'1/21/22, 5:47:04 PM	65	9	-1
0.6666666666666666	25	3	0.4	0.6666666666666666	9	'1/21/22, 5:47:04 PM	65	10	-1
9	1	1	9	9	1	'1/21/22, 5:47:04 PM	65	11	6
3	9	2	2	3	4	'1/21/22, 5:47:04 PM	65	5	-1

Figure 36 User Columns Synchronous Tab

Glossary

CRUD service: Create, Read, Update, Delete functionalities.

Apiinformation, ApiData: Is the same database table.

setNewValues(): is a method located on backend.

DecisionObject, NewColumn: Different programming objects used in several parts of the project.