Web Applications, A.Y. 2020/2021
Master Degree in Computer Engineering
Master Degree in ICT for Internet and Multimedia

Homework 1 – Server-side Design and Development
Submission date: 23 April 2021

Last Name	First Name	Badge Number
Buratto	Alessandro	2019041
Caligiuri	Matteo	2019283
Galizio	Daniele	2022160
Lincetto	Federico	2022299
Tommasin	Giovanni	2022309
Varotto	Marco	2019169

Objectives

The purpose of the project, in continuation with the one developed during the Foundation of Database course, is to develop a web interface to manage a factory of agricultural machines and equipments. The project is mainly focused on aspects related to the management of employees, various phases of production and the warehouse. Moreover, it takes care of the economic aspects by managing costs and trying to optimize the production phase and minimizing the unnecessary waste of time and resources.

Main functionalities

The web application is used to allow stakeholders and employees at any level to interact with data collected into the database. They will be able to monitor all the phases and to having a global vision of the factory.

The website is divided into many areas, one for each user of the system that are:

- Administrator: will have full control of the system to supervise and manage all the operations and all the employees. He/she can also assign correct access privileges to employees' accounts and is the only one that can add or remove an employee from the system.
- Designer: will insert in the system products offered by the company specifying all the materials and processes needed to build the equipment that the company is going to produce.

- **Production planner**: will carry out purchase orders for raw materials and products and will manage and schedule the internal production. Furthermore, he monitors if the production chain performs like the estimated previsions.
- **Accountant**: will monitor all the expenses and the earnings of the company analyzing the material and the product orders, but also the employees' salary and the fixed costs. As well, he/she evaluates the performance of production compared to previous estimates. Then, using all of this information, the accountant generates the monthly financial report of the company.
- **Warehouse workers**: will access the system to keep update the warehouse catalogue and they manually manage the status of items and supplying as they enter or leave the warehouse.
- **Production line workers**: they will access the system in order to register the actual time needed for the item processing and to see which are the next tasks to perform and those one already performed.

For each role a different set of functionalities is granted. Every single operation is reserved to a different role, with the exception for the Administrator who can exploit any kind of functionality.

Presentation Logic Layer

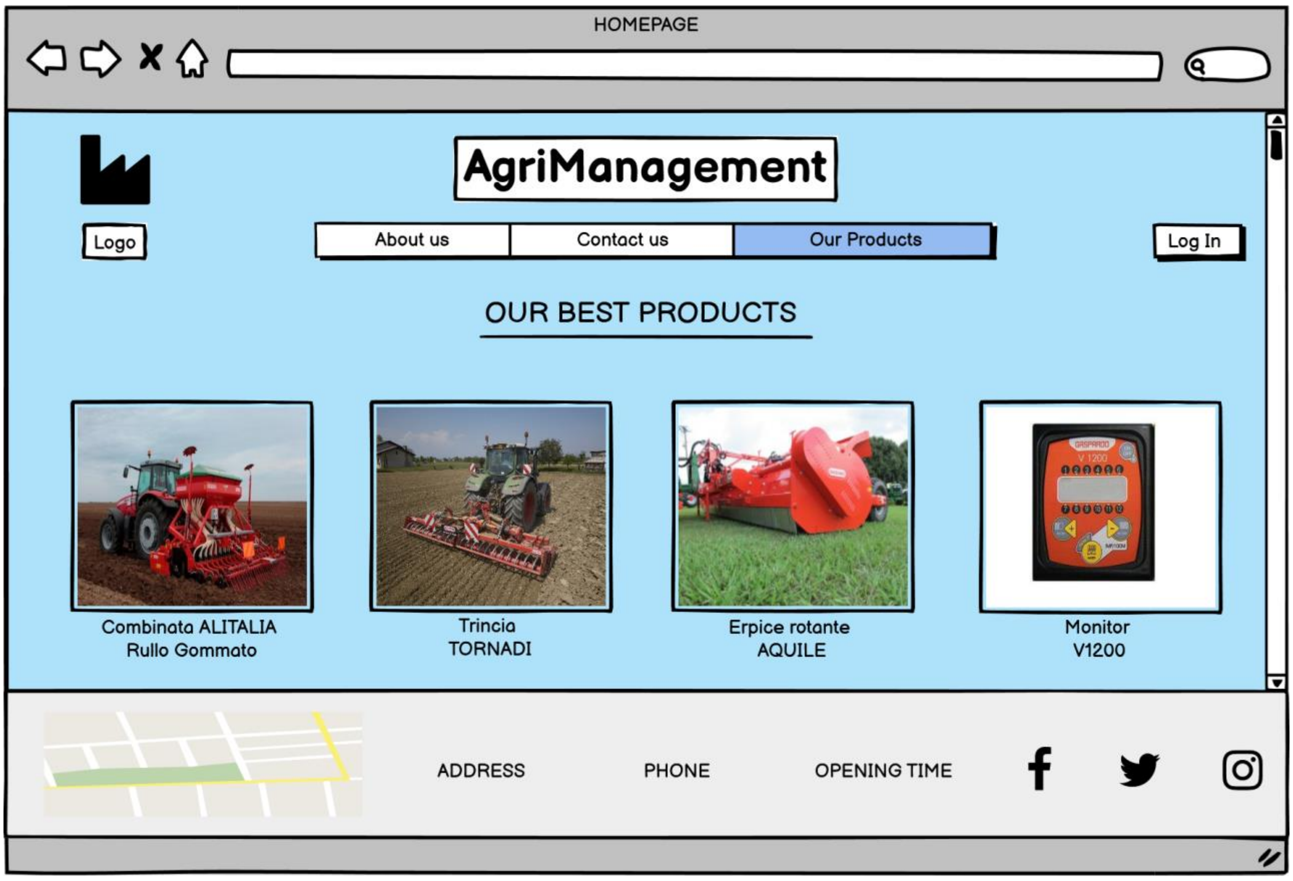
The project was planned to be divided into the following pages:

- **Homepage**: contains information about the catalogue of the factory, in which there are available products that the factory offers at the moment. This page also contains a section that allows registered users to login into the system, a section for consulting the information about the factory and one for the contact info of the company. These are the only pages reachable by anyone in the internet. Developed via jsp.
- **Login page**: allows the users login to the web application and be redirected to their own homepage where they can find access to all the functionalities he/she oversees. Developed via jsp.
- **Administrator page**: contains all links to the other role functionalities since he has the full control over everything. This page contains also a section dedicated to the exclusive operations that only the administrator can perform. For example, the insertion of the new employee, the update information of an employee or the delete of an employee from the database. This page is developed via jsp.
- **Warehouse worker page**: it contains many links that allow to navigate through the worker duties i.e., changing the status of an item, register the reception or the departure of raw materials and to look up the list of stored the items. Developed via jsp.
- **Designer page**: it contains all the functionalities for manage processes. In particular, there is a link for adding new processes with the contextual add of new products and raw materials if not already in the system. There are also other links which give the possibility of viewing the list of all raw materials or searching the processes to be executed in order to build a specific product. In addition, from here the designer can access the functionality to modify the estimated time of the processes and the attribute available of the products to define if those products are still offered and produced by the company. Developed via jsp.

- Production planner page: it contains all the links to the main functionalities divided by major categories: one link contains the links to insert a new production order and see a list of all orders completed or in production where it is possible to delete or see more information on them. Another link brings to the functionalities to see the status of the warehouse with raw materials and stored finished products, manage the list of suppliers and create new raw material order. There is also a link that lets the production planner manage the list of customers with similar functionalities previously mentioned for the suppliers. Furthermore, there is a link that allows checking the orders that are currently in production giving further detailed information such as the time actually taken to complete a production phase and the details of the worker that was assigned to that job. This page is developed via jsp.
- Accountant page: it contains some links related to the economic aspects of the company. In detail, he/she can manage the fixed costs the company has to pay, have a look to the financial overview and to the time difference analysis. Moreover, there is a page where each month the accountant has to upload the report of that period and there is also the possibility to download the old reports. This page is developed via jsp.
- Production line worker page: it shows all the employee's operations. In detail the page contains the running operations and a form in which the production line worker can insert the time needed for complete one of them. Moreover, the page shows the queued and the completed operations. This page is developed via jsp.

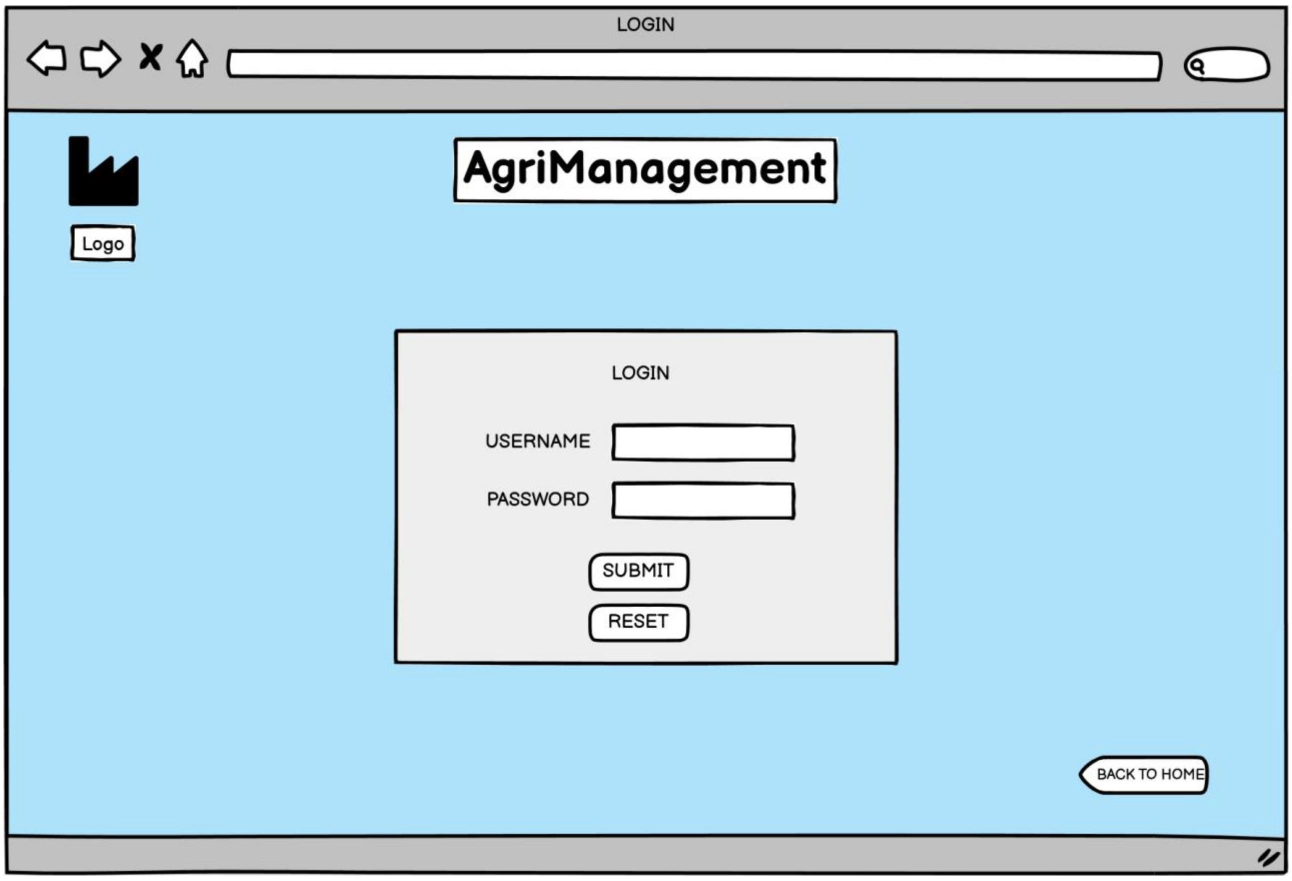
On each page above, a list is displayed containing all the information, name surname role and salary, which belong to the specific employee that logged into the system.

Home Page (Interface Mockup)



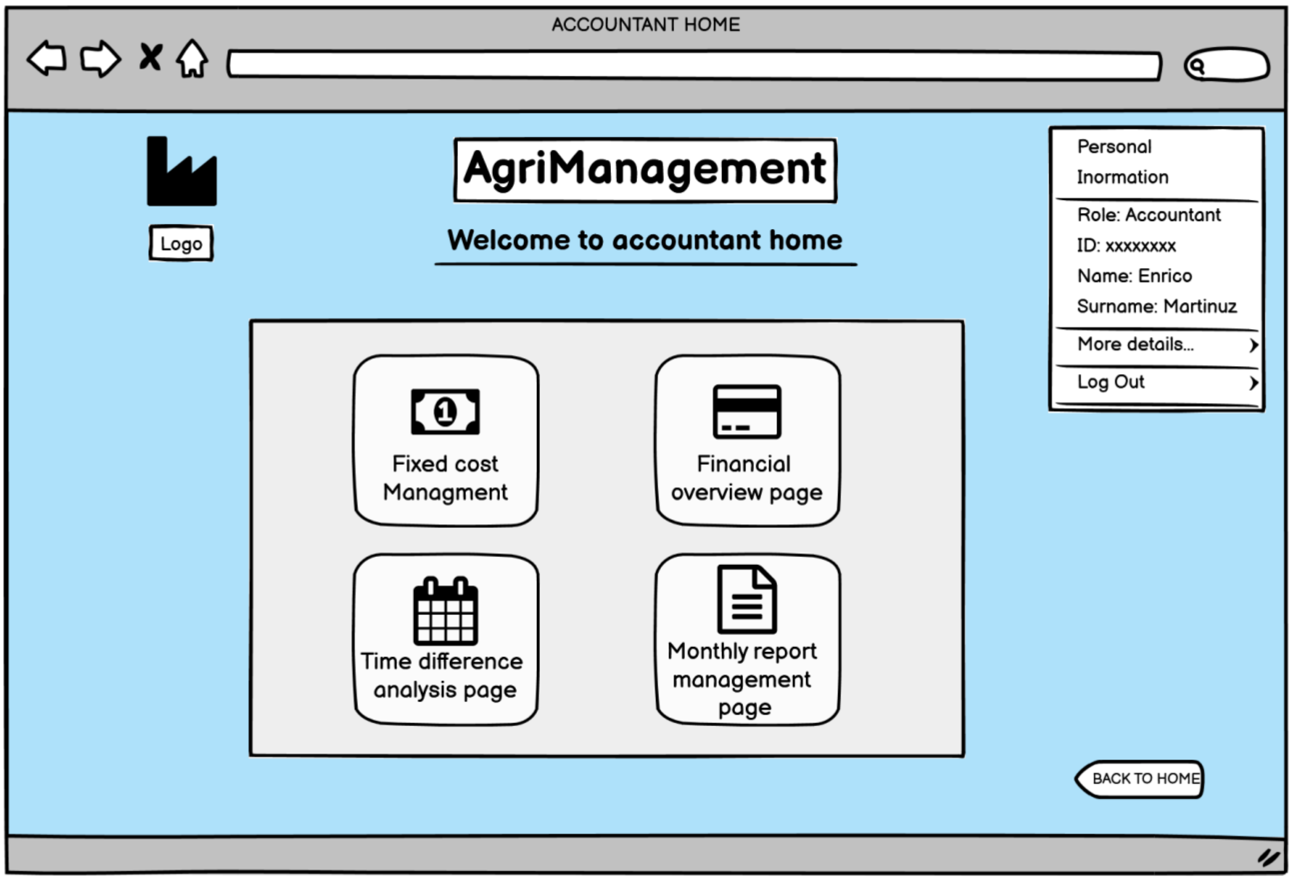
The homepage contains the main information about the factory and allows the user to discover what the factory produces and other details. On this page there are many links that lead to informative sections about the factory and to discover the catalogue of the factory, which contains all the products that the company is still producing. A list of the best-selling products is also shown.
There is also a part dedicated to the internal staff of the factory that allows to log into the system and access to the respective reserved area.

Login Page (Interface Mockup)



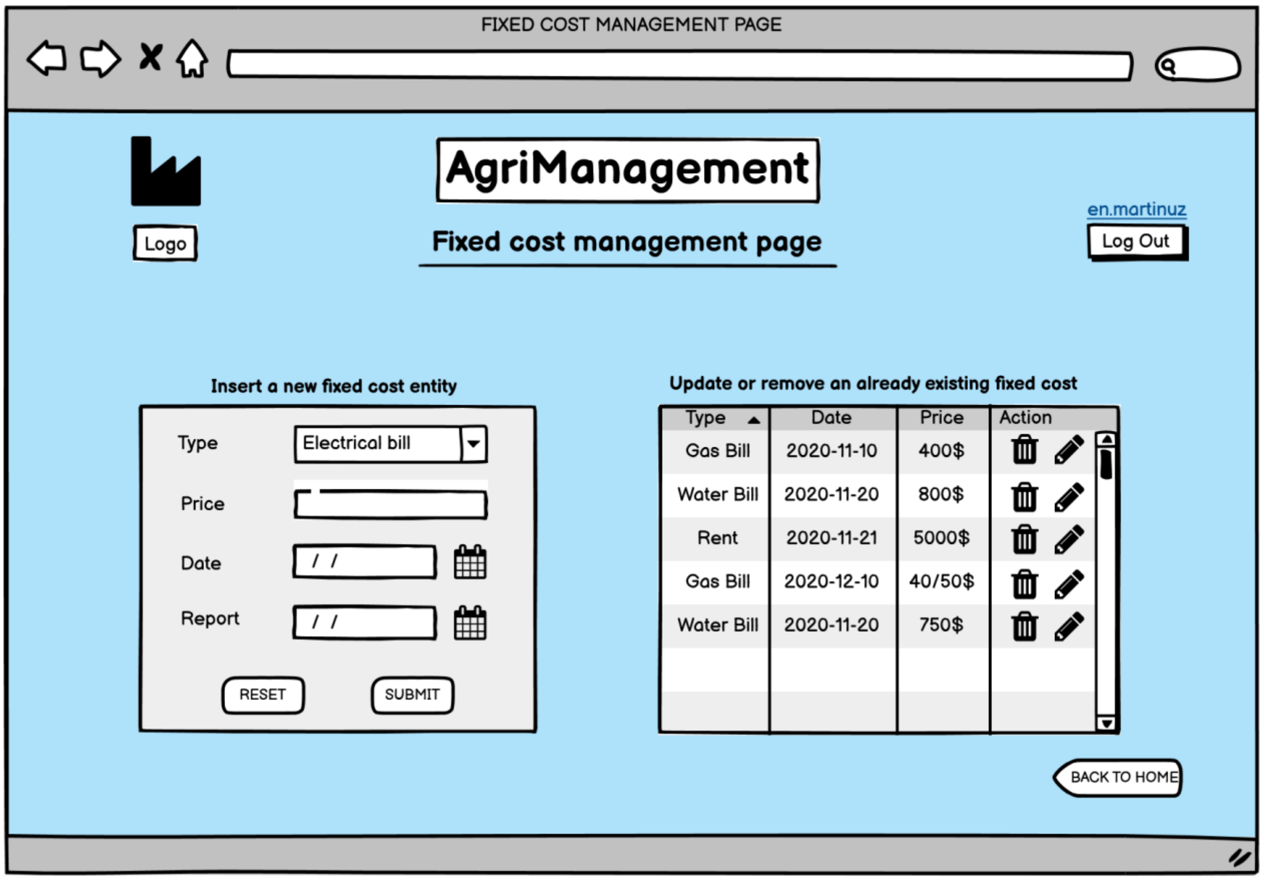
Our web app has functionalities that can be performed only by specific roles, so we had to create a login page. This page mainly consists of two fields, i.e., the username and the password, and two buttons, i.e., the submit and reset ones. When a user clicks the login button into the main homepage, he/she is redirected to this page so the employee can authenticate by just inserting the username and the password he/she was given in the respective text box. Now if the user clicks the submit button, he/she will be redirected to his/her specific homepage and he/she is allowed only to perform the determined tasks the employee is in charge of. If the user commits some errors in the insertion of the credentials, he/she will be asked to insert them again.

Worker Page (Interface Mockup)



This is a typical worker's homepage; in this example the accountant's one is displayed. On the central part this page contains the duties of the accountant which are links that lead to other pages where he can perform his works.
There is also a section on the upper right side for the personal information of the worker and there is the button that allows the worker to log out from the system.

Fixed Cost Management Page (Interface Mockup)



FIXED COST MANAGEMENT PAGE

AgriManagement

Fixed cost management page

Logo

en.martinuz

Log Out

Insert a new fixed cost entity

Type	Electrical bill
Price	
Date	/ /
Report	/ /

RESET SUBMIT

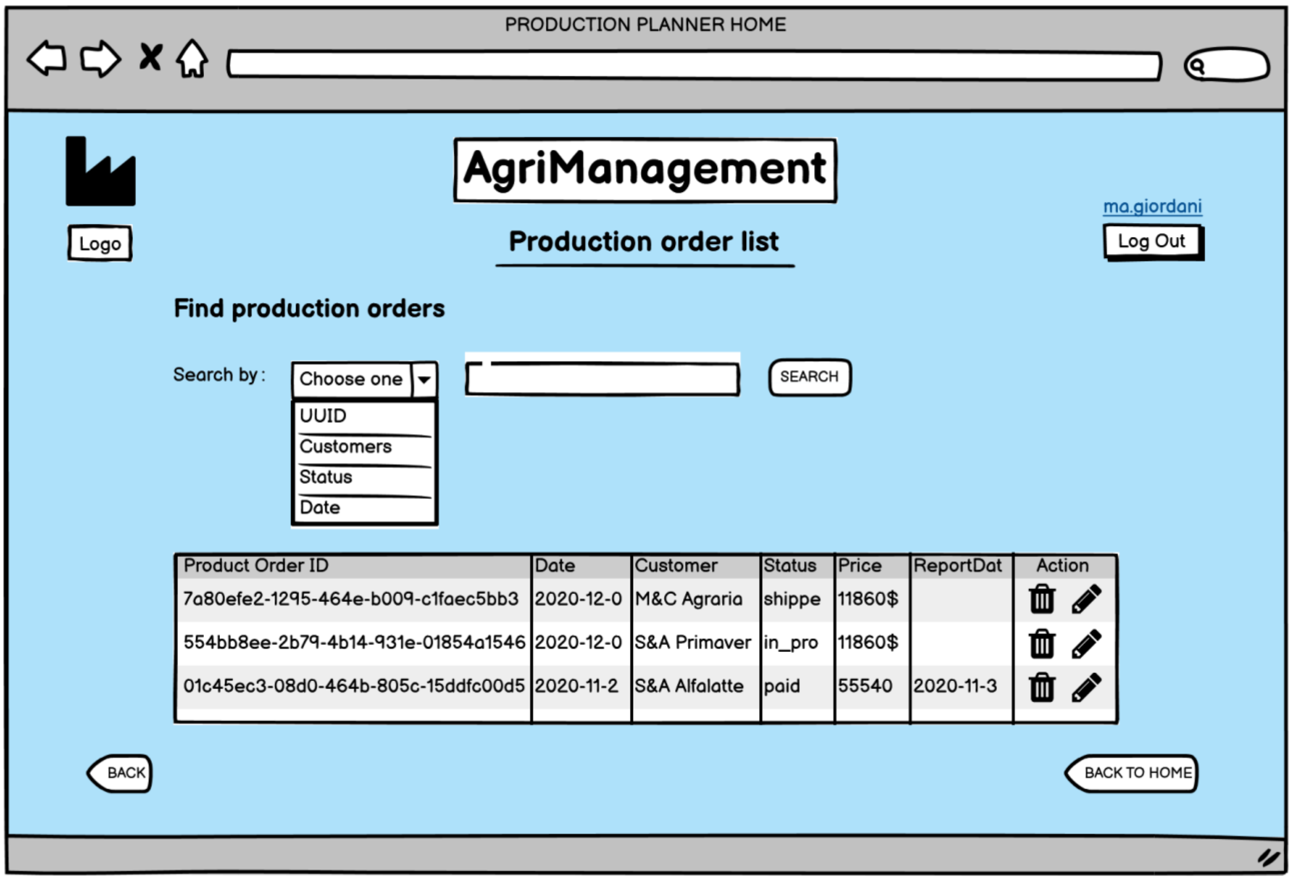
Update or remove an already existing fixed cost

Type	Date	Price	Action
Gas Bill	2020-11-10	400$	
Water Bill	2020-11-20	800$	
Rent	2020-11-21	5000$	
Gas Bill	2020-12-10	40/50$	
Water Bill	2020-11-20	750$	

BACK TO HOME

This is a page for the accountant's duty and in particular for managing fixed costs. It is divided in two sections: one for the insertion of a new fixed cost and the other for update or remove an existing fixed cost.
This is a dynamic page and so when the accountant decides to insert a new fixed cost and presses the submit button, the page automatically adds to the row of the table the new fixed cost entity. In this table there are two buttons for each fixed cost: one for deleting the current fixed cost from the database (bin) and one for updating the fixed cost (pencil) that when clicked produces the opening of a new page which allows to change the related values. Moreover, to be more precise, when the bin button is clicked the page will be automatically refreshed and the updated table will be visible.

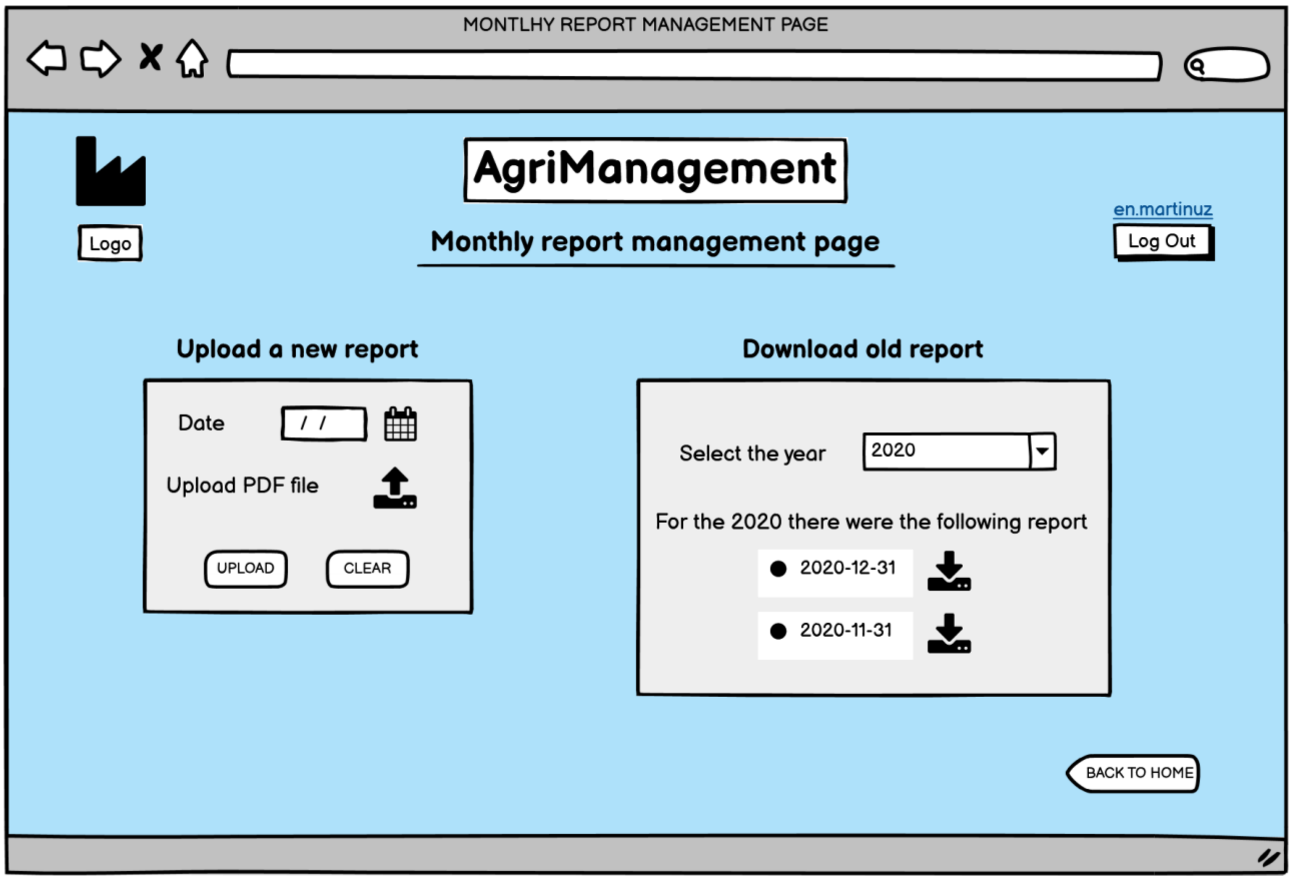
Production Orders Page (Interface Mockup)



Product Order ID	Date	Customer	Status	Price	ReportDat	Action
7a80efe2-1295-464e-b009-c1faec5bb3	2020-12-0	M&C Agraria	shippe	11860$		🗑 ✏
554bb8ee-2b79-4b14-931e-01854a1546	2020-12-0	S&A Primaver	in_pro	11860$		🗑 ✏
01c45ec3-08d0-464b-805c-15ddfc00d5	2020-11-2	S&A Alfalatte	paid	55540	2020-11-3	🗑 ✏

This is one of the pages that the production planner can access. As can be seen from the image this page shows the list of the product orders in the system and then allows the user to filter them by a specific attribute type that can be chosen from a list. Once chosen the attribute type, the user just needs to insert it in the field and click search, so the page dynamically changes and displays only the filtered production orders. Moreover, the production planner has the possibility to delete or update the order by clicking on the action icons. If the user clicks the bin, after a confirmation the order will be deleted, the page immediately changes and does not show it anymore. In the other hand, if he/she clicks the pencil he/she will be redirected to another page where it is possible to insert the new details.

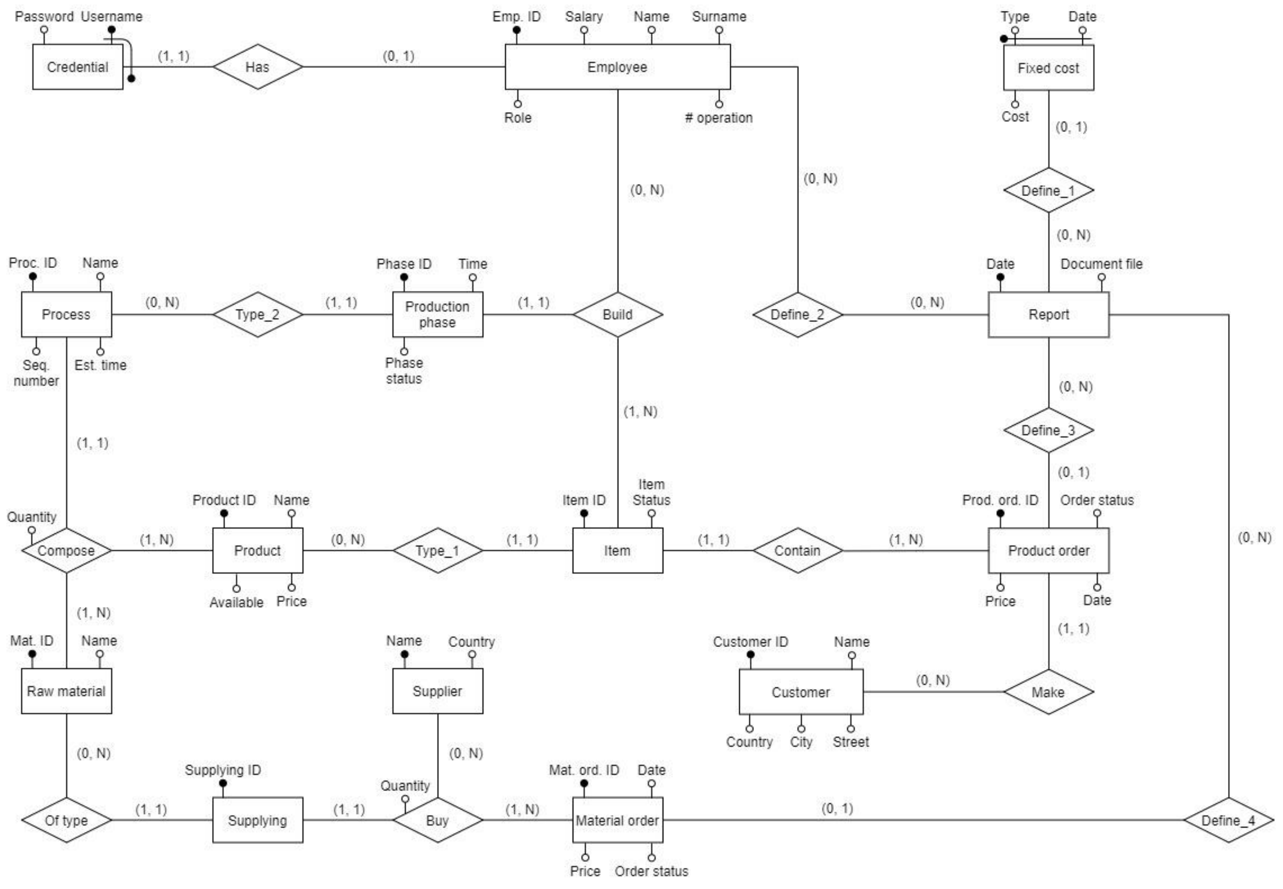
Monthly Management Report Page (Interface Mockup)



Through this page the accountant performs the task of uploading new reports and/or downloading the old ones. In particular, when the employee does this operation, he/she needs to compile in the left side the date field and insert the PDF document by clicking the upload button. Once the action is completed, he/she only has to click upload to confirm it. In the right side, by choosing the year, there will be dynamically shown the list of the reports made at that time and the user can decide to download one or more of them.

Data Logic Layer

Entity-Relationship Schema



The entity-relationship contains 14 main entities:

- **Credential**: contains data that allows users to login into the database and make operations. Its primary key is composed by the username, that is of type char with variable length (up to 100 characters) and the id of the employee, which is of type UUID and it is also the foreign key. There is also an additional attribute that is the password which is stored as char with variable length. The password is stored into the database in the md5 hash format.
- **Employee**: each employee is identified by an id that is of type UUID. For each employee we also record its name, surname (both of type varchar, up to 20 characters), the role, which is of type *role*, the salary, that is of type numeric (7,2) and also the number of operations that the employee has not yet completed, which is of type smallInt. Note that this value could be null if the role of the employee is not *'production line worker'*.
- **Production phase**: represents the actual manufacturing phase carried out for the production of an item. It is uniquely identified by an ID which is an UUID. It is also represented by other attributes like time, which is of type Interval, and it defines the actual time needed to complete the phase. Another one is the phase status, of type *phase_status* (custom enumeration), that represents the status of the production phase. In the related relational schema table, there is contained also the id of the

employee that has to complete the production phase since the production phase and the employee are involved into a 0-N relation.

- Process: describes the working phases of the production of products. It must be considered as a dictionary of all processes that the factory can make. The primary key is composed by the id of the process which is an UUID. Moreover, it contains the following attributes: name of the process, which is of the type char variable with length (up to 60 characters), the estimated time that is an estimation of the time needed to complete the process represented as an Interval, the sequence number that is a progressive numbering of the processes needed and it is saved as integer. In the related relational schema table, it is also saved the attribute quantity, which represents the same type of raw materials needed to carry out that process for a product and it is stored as integer. In the same table there are also the id of the product and the id of the raw material, which are both UUID, because process, product and raw material are involved in a ternary relationship and process has maximum cardinality equals to 1.
- Product: it is a stuff that can be produced and sold by the company. Each product is uniquely identified by an id that is an UUID and it is also represented by its name, that is of type char with variable length (up to 50 characters), its price (float) and the attribute available, which is a boolean value, that indicates whether the product is still produced by the company.
- Item: each item is a single piece of a customer's order manufactured by the factory, stocked in the warehouse and ready to be shipped. It is identified by an id, that is an UUID. It contains the attribute status that represents the actual status of the item and it is of the type *status_item* (custom enumeration). In the related relational schema table, it is also stored the product id of which it is referred to and the material order id, which are both UUID.
- Raw Material: represents the material in the warehouse needed to produce items. The primary key is composed by the id of the material, that is an UUID. The other attribute of this entity is the name of the material, which is of the type char with variable length (up to 50 characters).
- Supplying: it describes the supplying of a raw material. It Is identified by an id, which is an UUID. In the related relational schema table, is also stored the id and the quantity of the raw material for which we carry out the supplying alongside with the supplier's name, which refers to the seller of the raw material (saved as char with variable length), and the order id, which is an UUID.
- Supplier: this entity represents the company that sells raw materials to our factory. It is uniquely identified by the company's name, which is of the type char with variable length (up to 50 characters). It also stores the country where the supplier is located that is of type char with variable length (up to 100 characters).
- Material order: it represents the real order of raw materials that the factory needs to purchase. The primary key is composed by the id of the order, which is of the type UUID. In this entity is also stored the final price of the order, that is of the type numeric (8,2), the date in which the order was done, the attribute order_status, which is of the type *material_order_status* (custom enumeration). Each material order entity could be linked to a report one so, in the related relational schema table, it is present also the date of the report (that can be null if the correspondent report is not already present in the system) as a foreign key.
- Report: it describes the monthly incomes and outcomes and keeps track of all the products realized in the month. Its primary key is composed by the date in which the report was done, that is of the type date. It also memorizes the document file which is a PDF document that contains the economics report of the company.

- **Fixed Cost**: it represents the costs that the company has to sustain every month. Its primary key is composed by the date of payment and the type which is of the type *fixed cost type*. Then this entity memorizes the price of the payment, which is of the type numeric (2,8). In the related relational schema table is also stored, as foreign key, the date of the report in which the cost has been reported.
- **Product Order**: it describes the orders for the production of items. Every product order is identified by an id of type UUID. For each order there are other attributes: the date in which is done, the status of the order, which is of the type *product_order_status* (custom enumeration), the price that represents the overall cost of all items in the order and it is represented as type numeric (2,8). In the related relational schema table there are also present two foreign keys: one related to the date of the report that includes that order and another one that represents the customer id which is of the type UUID.
- **Customer**: It represents who wants to buy items from the company. It is uniquely identified by its id, which is of the type UUID. Each customer has a name, of type char with variable length (up to 50 characters), a country, a city and a street, all three of type char with variable length (up to 100 characters)

Other Information

The interaction with almost all the entities can be performed through insertions, deletions and updates via interface. While, the interaction with the Report entity has to be done by the upload of the report through the user interface.

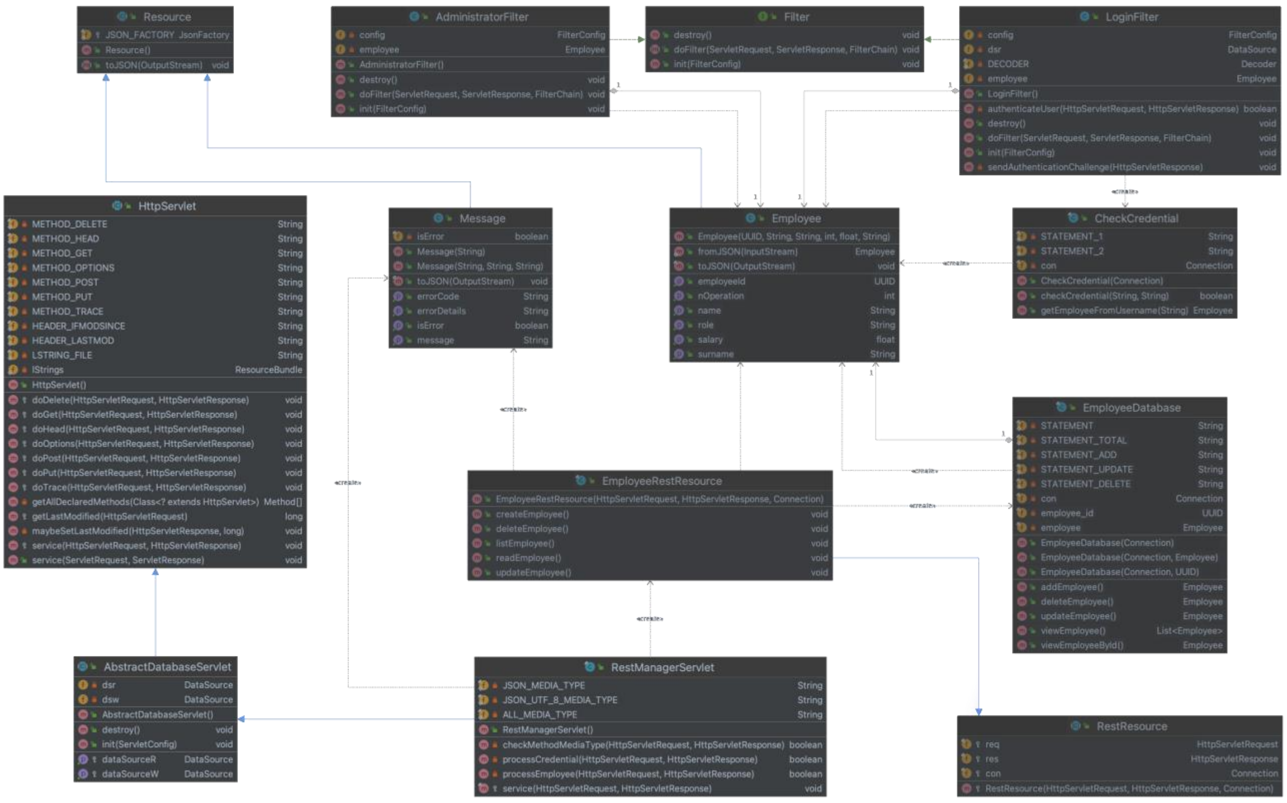
There are 6 custom enumerations:

- **Role**: contains a value for each possible role of the employees (i.e., Administrator, Warehouse worker, Production line worker, Accountant, Designer, Production planner);
- **Phase_status**: contains a value for each possible state of the production phase (i.e., queued, running, completed);
- **Item_status**: contains a value for each possible state of the item (i.e., in_construction, stored, shipped);
- **Product_order_status**: contains a value for each possible state of the production order (i.e., in_production, completed, shipped, cancelled, not_paid, paid);
- **Material_order_status**: contains a value for each possible state of the material order (i.e., completed, received);
- **Fixed_cost_type**: contains a value for each possible type of the fixed cost (i.e., electrical_bill, gas_bill, water_bill, rent, tax).

Business Logic Layer

Class Diagram

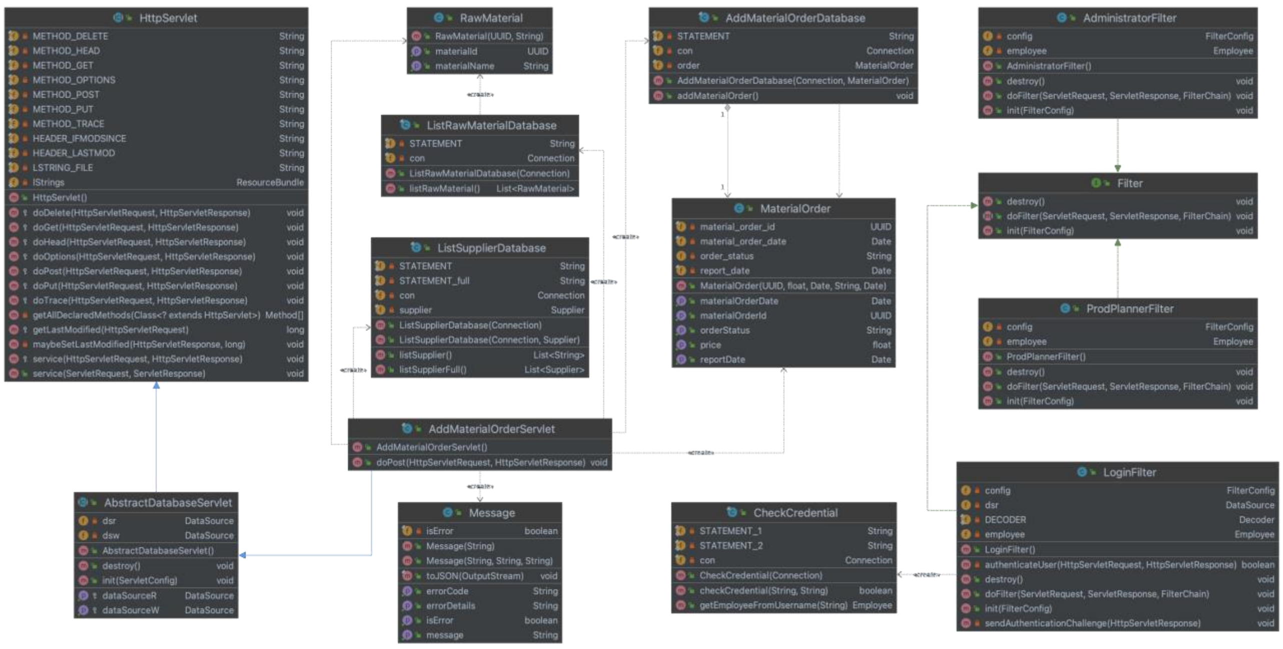
EmployeeRestResource Diagram:


Resource
JSON_FACTORY JsonFactory
Resource()
toJSON(OutputStream) void
AdministratorFilter
config FilterConfig
employee Employee
AdministratorFilter()
destroy() void
doFilter(ServletRequest, ServletResponse, FilterChain) void
init(FilterConfig) void
Filter
destroy() void
doFilter(ServletRequest, ServletResponse, FilterChain) void
init(FilterConfig) void
LoginFilter
config FilterConfig
dsr DataSource
DECODER Decoder
employee Employee
LoginFilter()
authenticateUser(HttpServletRequest, HttpServletResponse) boolean
destroy() void
doFilter(ServletRequest, ServletResponse, FilterChain) void
init(FilterConfig) void
sendAuthenticationChallenge(HttpServletResponse) void
HttpServlet
METHOD_DELETE String
METHOD_HEAD String
METHOD_GET String
METHOD_OPTIONS String
METHOD_POST String
METHOD_PUT String
METHOD_TRACE String
HEADER_IFMODSINCE String
HEADER_LASTMOD String
LSTRING_FILE String
lStrings ResourceBundle
HttpServlet()
doDelete(HttpServletRequest, HttpServletResponse) void
doGet(HttpServletRequest, HttpServletResponse) void
doHead(HttpServletRequest, HttpServletResponse) void
doOptions(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
doPut(HttpServletRequest, HttpServletResponse) void
doTrace(HttpServletRequest, HttpServletResponse) void
getAllDeclaredMethods(Class<? extends HttpServlet>) Method[]
getLastModified(HttpServletRequest) long
maybeSetLastModified(HttpServletResponse, long) void
service(HttpServletRequest, HttpServletResponse) void
service(ServletRequest, ServletResponse) void
Message
isError boolean
Message(String)
Message(String, String, String)
toJSON(OutputStream) void
errorCode String
errorDetails String
isError boolean
message String
Employee
Employee(UUID, String, String, int, float, String)
fromJSON(InputStream) Employee
toJSON(OutputStream) void
employeeId UUID
nOperation int
name String
role String
salary float
surname String
CheckCredential
STATEMENT_1 String
STATEMENT_2 String
con Connection
CheckCredential(Connection)
checkCredential(String, String) boolean
getEmployeeFromUsername(String) Employee
EmployeeDatabase
STATEMENT String
STATEMENT_TOTAL String
STATEMENT_ADD String
STATEMENT_UPDATE String
STATEMENT_DELETE String
con Connection
employee_id UUID
employee Employee
EmployeeDatabase(Connection)
EmployeeDatabase(Connection, Employee)
EmployeeDatabase(Connection, UUID)
addEmployee() Employee
deleteEmployee() Employee
updateEmployee() Employee
viewEmployee() List<Employee>
viewEmployeeById() Employee
EmployeeRestResource
EmployeeRestResource(HttpServletRequest, HttpServletResponse, Connection)
createEmployee() void
deleteEmployee() void
listEmployee() void
readEmployee() void
updateEmployee() void
AbstractDatabaseServlet
dsr DataSource
dsw DataSource
AbstractDatabaseServlet()
destroy() void
init(ServletConfig) void
dataSourceR DataSource
dataSourceW DataSource
RestManagerServlet
JSON_MEDIA_TYPE String
JSON_UTF_8_MEDIA_TYPE String
ALL_MEDIA_TYPE String
RestManagerServlet()
checkMethodMediaType(HttpServletRequest, HttpServletResponse) boolean
processCredential(HttpServletRequest, HttpServletResponse) boolean
processEmployee(HttpServletRequest, HttpServletResponse) boolean
service(HttpServletRequest, HttpServletResponse) void
RestResource
req HttpServletRequest
res HttpServletResponse
con Connection
RestResource(HttpServletRequest, HttpServletResponse, Connection)
«create»
1

The above diagram shows the class diagram of the `EmployeeRestResource` class that is the one used to manage the update, insertion and deletion of an employee using the REST paradigm.
From the above schema we can notice that we have a general class (`AbstractDatabaseServlet`) sub-class of the `HttpServlet` one which main roles is to implement the servlet initialization method and establish the connection with the database. This class is then extended by all the servlet classes in the project. Following this behavior also the `RestManagerServlet` class does that.
The `RestManagerServlet` class corresponds to the servlet in charge of parsing the received URI to determine the type of resource that the user wants to interact with. Once this servlet has processed the URI, without errors, it forwards the request to the correspondent REST resource class that is able to correctly handle the interested resource. In our case the class in charge of processing the requested resource is the `EmployeeRestResource` class, that is a sub-class of the general `RestResource` class and implements all the method required to handle proper resource.
As we can understand from its name the aim of the `EmployeeRestResource` class is to manage all the operation that can be performed on the `Employee` entity in the database (update, insert, delete, search). To accomplish all of the required operation these classes need the support of two other classes: the `Employee` and the `EmployeeDatabese` ones. The former is a sub-class of the `Resource` class and essentially stores all the information

concerning an employee that is present in the database; the latter, instead is the class in charge of communicating with the database (using SQL statement) and so the only one that can modify the `Employee` table stored in the database.
Other than that, in the diagram we can also see all the classes needed to perform the filtering operation to ensure that only the administrator can access these functions. For performing this control, we use two filter classes (both subclasses of the general `Filter` class): `LoginFilter` and `AdministratorFilter`. The former is in charge of checking that the user is authorized to login in the system, in other words it checks if the inserted credentials is valid using an instance of the CheckCredential class. The latter, instead checks if the just logged in user is an administrator or not.
The last element in the diagram that remains to be analyzed is the `Message` class, that is used by a lot of other classes to create messages entity that are then displayed to the user in order to give him useful information about the success, or the failure of the requested operations. Also this class, like the `Employee` one is a sub-class of the `Resource` class.

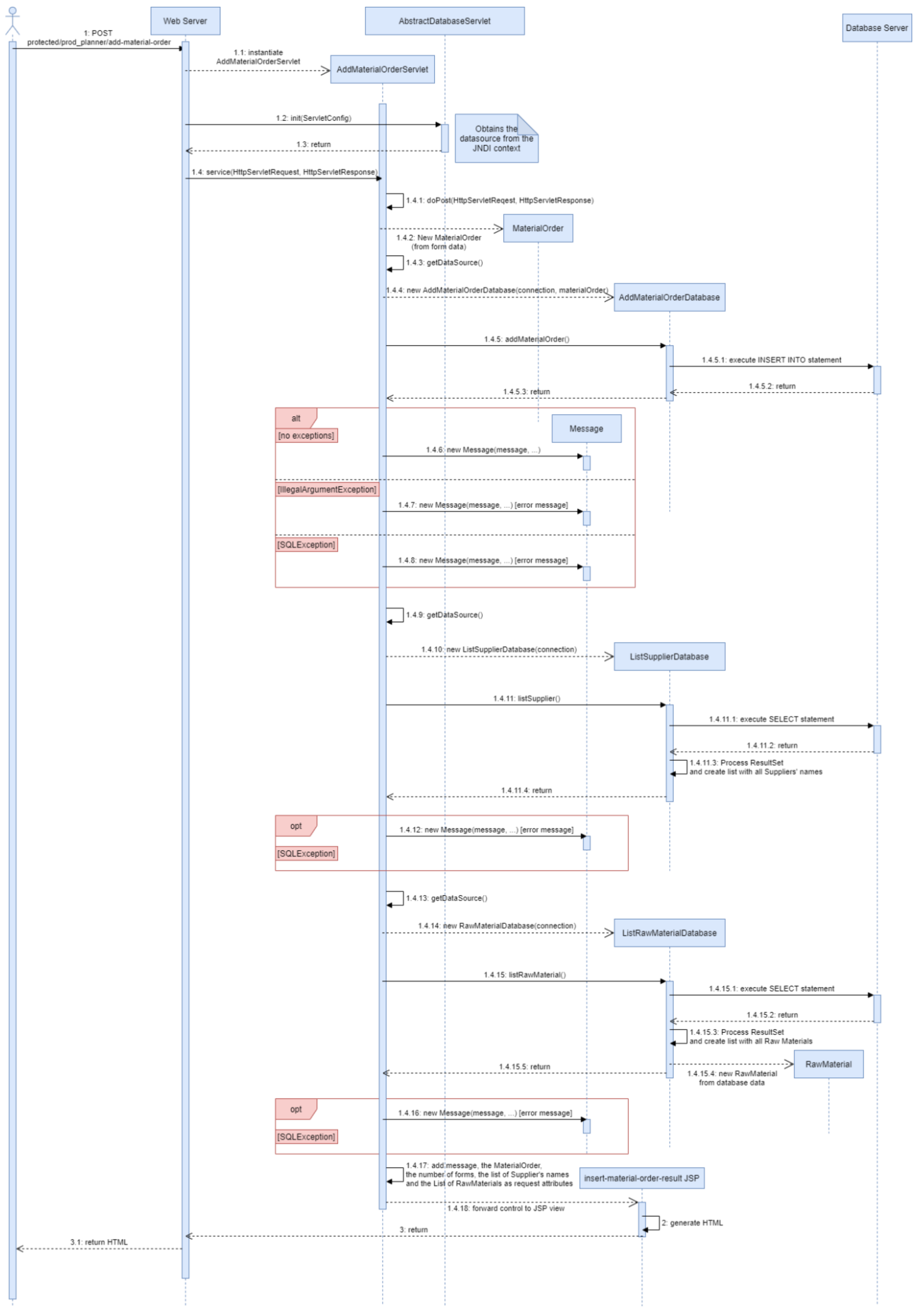
ADDMATERIALORDERSERVLET DIAGRAM:



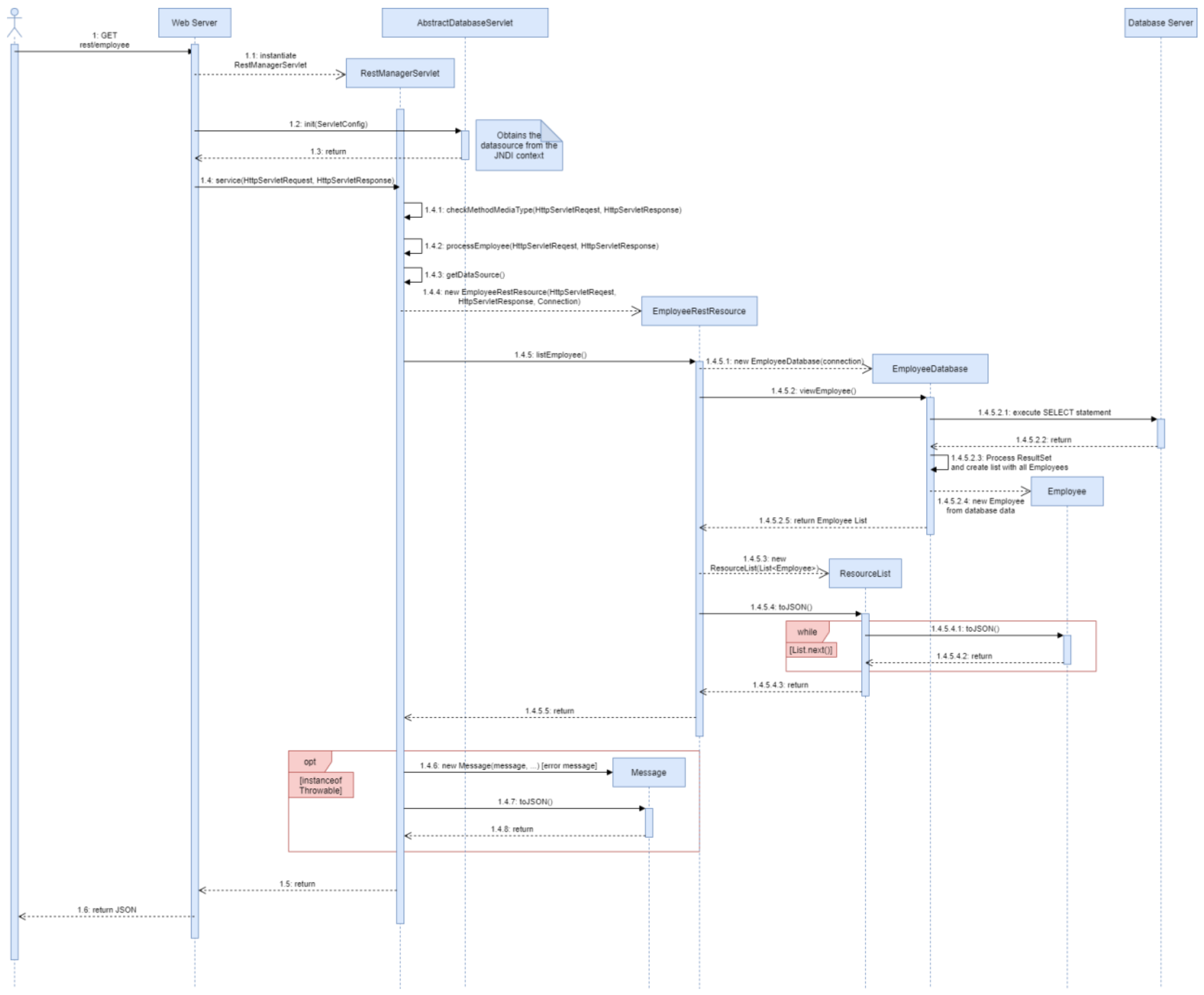
The above diagram shows the class diagram of the `AddMaterialOrderServlet` class that is the one used to manage the insertion of a new material order in the database.
From the above schema we can notice that we have a general class (`AbstractDatabaseServlet`) sub-class of the `HttpServlet` one which main roles is to implement the servlet initialization method and establish the connection with the database. This class is then extended by all the servlet classes in the project. Following this behavior also the `AddMaterialOrderServlet` class does that.
The `AddMaterialOrderServlet` class to accomplish its role needs the support of five other classes: the `RawMaterial`, `MaterialOrder`, `ListRawMaterialDatabase`, `ListSupplierDatabese` and the `MaterialOrderDatabase`. The first two classes essentially store all the information concerning a raw material or a material order that is present in the database. The third and fourth classes, can establish a connection with the database and retrieve all the possible raw material or supplier. The fifth class is in charge of

communicating with the database (using SQL statement) and actually insert a new material order entity in the database.
Other than that, in the diagram we can also see all the classes needed to perform the filtering operation to ensure that only the administrator and the product planners can access these functions. For performing this control, we use three filter classes (both subclasses of the general Filter class): `LoginFilter`, `ProdPlannerFilter` and `AdministratorFilter`. The former is in charge of checking that the user is authorized to login in the system, in other words it checks if the inserted credentials are valid using an instance of the `CheckCredential` class. The last two classes, instead checks if the just logged in user is an administrator or a product planner and so if he is allowed to use this specific function.
The last element that remains to be analyzed in the diagram is the `Message` class, that is used by a lot of other classes to create messages entity that are then displayed to the user in order to give him useful information about the success, or the failure of the requested operations.

Sequence Diagram


Web Server
AbstractDatabaseServlet
Database Server
1: POST protected/prod_planner/add-material-order
1.1: instantiate AddMaterialOrderServlet
AddMaterialOrderServlet
1.2: init(ServletConfig)
Obtains the datasource from the JNDI context
1.3: return
1.4: service(HttpServletRequest, HttpServletResponse)
1.4.1: doPost(HttpServletReqest, HttpServletResponse)
MaterialOrder
1.4.2: New MaterialOrder (from form data)
1.4.3: getDataSource()
1.4.4: new AddMaterialOrderDatabase(connection, materialOrder)
AddMaterialOrderDatabase
1.4.5: addMaterialOrder()
1.4.5.1: execute INSERT INTO statement
1.4.5.2: return
1.4.5.3: return
alt
[no exceptions]
Message
1.4.6: new Message(message, ...)
[IllegalArgumentException]
1.4.7: new Message(message, ...) [error message]
[SQLException]
1.4.8: new Message(message, ...) [error message]
1.4.9: getDataSource()
1.4.10: new ListSupplierDatabase(connection)
ListSupplierDatabase
1.4.11: listSupplier()
1.4.11.1: execute SELECT statement
1.4.11.2: return
1.4.11.3: Process ResultSet and create list with all Suppliers' names
1.4.11.4: return
opt
[SQLException]
1.4.12: new Message(message, ...) [error message]
1.4.13: getDataSource()
1.4.14: new RawMaterialDatabase(connection)
ListRawMaterialDatabase
1.4.15: listRawMaterial()
1.4.15.1: execute SELECT statement
1.4.15.2: return
1.4.15.3: Process ResultSet and create list with all Raw Materials
1.4.15.4: new RawMaterial from database data
RawMaterial
1.4.15.5: return
opt
[SQLException]
1.4.16: new Message(message, ...) [error message]
1.4.17: add message, the MaterialOrder, the number of forms, the list of Supplier's names and the List of RawMaterials as request attributes
insert-material-order-result JSP
1.4.18: forward control to JSP view
2: generate HTML
3: return
3.1: return HTML

Here is reported the sequence diagram for the operations executed by adding the material order. The actor executes a POST request to the web server specifying the protected/prod_planner/add-material-order URI. Additionally, the information required for creating the material order and inserting the right amount of related different raw materials is passed to the web server. The web server instantiates the `AddMaterialOrderServlet` and calls the `service`() method passing as parameters the `HttpServletRequest` and `HttpServletResponse`. The servlet then analyzes the request and calls its `doPost`() method. This method initializes a new `MaterialOrder` object using the data from the form, gets a connection to the database and then it initializes a new `AddMaterialOrderDatabase` instance which is responsible for inserting the newly created `MaterialOrder` into the database. If no exceptions are encountered then an accomplishment `Message` is created, otherwise an error message is generated. Before sending back a response to the client the servlet needs to retrieve from the database other two lists that will be used in the JSP page generation to allow an easier data insertion in the subsequent forms. Firstly, a new connection to the database is obtained and it is passed as a parameter in the constructor for the `ListSupplierDatabase` constructor. When the `listSupplier`() method is called the `ListSupplierDatabase` object executes the `SELECT` query to the Database Server and by processing the `ResultSet` creates a list with all the `Employee`'s names which is subsequently returned to the servlet. If an exception was raised during this phase, an error message is created, otherwise a similar process is repeated for the `ListRawMaterialDatabase` which returns a list of all the `RawMaterial` objects in the database.
Once all this data is retrieved from the database, the message generated, the `MaterialOrder` object and the two lists are added to the request as attributes and the control is forwarded to the JSP compiler which generates the HTML page that is ultimately returned to the client.


Web Server
AbstractDatabaseServlet
Database Server
1: GET rest/employee
1.1: instantiate RestManagerServlet
RestManagerServlet
1.2: init(ServletConfig)
Obtains the datasource from the JNDI context
1.3: return
1.4: service(HttpServletRequest, HttpServletResponse)
1.4.1: checkMethodMediaType(HttpServletReqest, HttpServletResponse)
1.4.2: processEmployee(HttpServletReqest, HttpServletResponse)
1.4.3: getDataSource()
1.4.4: new EmployeeRestResource(HttpServletReqest, HttpServletResponse, Connection)
EmployeeRestResource
1.4.5: listEmployee()
1.4.5.1: new EmployeeDatabase(connection)
EmployeeDatabase
1.4.5.2: viewEmployee()
1.4.5.2.1: execute SELECT statement
1.4.5.2.2: return
1.4.5.2.3: Process ResultSet and create list with all Employees
Employee
1.4.5.2.4: new Employee from database data
1.4.5.2.5: return Employee List
1.4.5.3: new ResourceList(List<Employee>)
ResourceList
1.4.5.4: toJSON()
while
[List.next()]
1.4.5.4.1: toJSON()
1.4.5.4.2: return
1.4.5.4.3: return
1.4.5.5: return
opt
[instanceof Throwable]
1.4.6: new Message(message, ...) [error message]
Message
1.4.7: toJSON()
1.4.8: return
1.5: return
1.6: return JSON

Now it is reported the sequence diagram for the REST API call to retrieve the list of employees. The actor sends a GET request to the Web Server specifying the URI /rest/employee. The Web Server instantiates a new `RestManagerServlet` and initializes it, then it calls the `service()` method passing as parameters the `HttpServletRequest` and `HttpServletResponse`. The servlet then calls the `checkMethodMediaType()` method passing as parameters the request and response which is responsible for checking that the request method and MIME media type are correct. If both of them are correct then the servlet calls the `processEmployee()` method passing as parameters the `HttpServletRequest` and `HttpServletResponse`. This method checks the URI to see if the request is for an employee resource. If it is the case then it calls the appropriate Database object to execute the correct query on the Database Server. In this case the given URI with the GET method is used to retrieve a list with all the Employee resources in the database. The servlet then gets the connection to the database, instantiates a new `EmployeeRestResource` passing as parameters the `HttpServletRequest` and `HttpServletResponse` and the newly created connection and calls the `EmployeeRestResource's listEmployee()` method. This method creates a new `EmployeeDatabase` instance passing as parameter the connection to the database and by calling the `viewEmployee()` method this `EmployeeDatabase` object executes the `SELECT` statement and processes the `ResultSet` creating a list of all Employees contained in the database. This list is returned to the `EmployeeRestResource` that creates a new instance of `ResourceList` passing as parameter the list of `Employee`s. By calling the `toJSON()` method of this class the `toJSON()` method of `Employee` is called for every object in the list passed in the previously mentioned constructor. The resulting JSON is then returned to the `RestManagerServlet` and an error message (which is promptly converted to JSON) is generated if any exception occurred in the previous process. The resulting JSON (the `Employee` list or the error message) is then returned to the Web Server and finally to the Client.

REST API Summary

The REST API has been implemented for the management of employees and credentials associated to each of them. These operations are reserved to the Administrator only, for this reason they must pass through a filter to ask for authentication. For this reason, the URI presented in the table must be completed by the prefix `protected/administrator/rest/`.
All these endpoints are developed in a traditional way. For each of them getting, adding, removing and updating operations will be possible.

URI	METHOD	DESCRIPTION
employee	GET	Return a list of all employees stored in the database.
employee/{employeeID}	GET	Return the employee identified by the UUID specified.
employee	POST	Add an employee to the database with the specified UUID and details specified in the body.
employee/{employeeID}	PUT	Update an existing employee in the database defined by the UUID with the details specified in the body.
employee/{employeeID}	DELETE	Delete the employee specified by the UUID from the database.
credential	GET	Return a list of all credentials stored in the database.
credential/{employeeID}	GET	Return the credential identified by the UUID of its owner specified.
credential	POST	Add the credential to the database with the specified UUID of its owner and details specified in the body.
credential/{employeeID}	PUT	Update existing credential in the database defined by the UUID of its owner with the details specified in the body.
credential/{employeeID}	DELETE	Delete the credential specified by the UUID of its owner from the database.

REST API Error Codes

This is the list of error codes used in the application. As we can see, there are two classes of errors: class E4 and class E5. The class E4 refers to errors involving a wrong composition of the request, often due to user's behaviour, like wrong inputs, or errors in the web application frontend. Class E5 errors refers to failures occurring when the request is being processed and usually depend on the state of the database or of the web application backend.

Error Code	HTTP Status Code	Description
E4A1	400 Bad Request	Output media type not specified.
E4A2	406 Not Acceptable	Unsupported output media type.
E4A3	400 Bad Request	Input media type not specified.
E4A4	415 Unsupported Media Type	Unsupported input media type.
E4A5	405 Method Not Allowed	Unsupported operation.
E4A6	404 Not Found	Unknown resource requested.
E4A7	400 Bad Request	Wrong URI format.
E5A1	500 Internal Server Error	Unexpected error while processing a resource.
E5A2	409 Conflict	Resource already exists.
E5A3	404 Not Found	Resource not found.
E5A4	409 Conflict	Cannot modify a resource because other resources depend on it.

REST API Details

Here are reported 3 functionalities exploited with the use of REST API. All are referred to the employee management and carried by the administrator.

List of Employees

This functionality is exploited by the administrator in the employee management page. When called, it retrieves the list of all employees hired by the company with the respective attributes of the Employee resource.

- URL
  employee/

- Method
  GET

- URL Params
  No parameters are required in the URL.

- Data Params
  No parameters are required in the body of the request.

- Success Response
  Upon success, the servlet returns a JSON object with the required information.

  Code: 200
  Content:
  ```
  {"resource-list":
     [{"employee":{
       "employeeId":"497f2fbc-0f54-4785-ad42-5a36592dd946",
       "surname":"Bedini",
       "name":"Marian",
       "nOperation":6,
       "salary":1100.0,
       "role":"Production line worker"
       }
     },

     …

     ]
  }
  ```

- Error Response

  Code: 500 INTERNAL_SERVER_ERROR
  Content: {“Cannot list employees: unexpected error.”: {“code”: E5A1, “message”: null}}
  When: if there is a SQLException or an empty list is returned, this error is returned.

  Code: 401 UNAUTHORIZED
  When: The URI is accessible only to logged users with role “administrator”.

Insert an Employee

This functionality is exploited by the administrator in the employee management page. When called, it inserts a new employee in the database.

- URL
  employee

- Method
  POST

- URL Params
  No parameters are required in the URL.

- Data Params

  Required:
  - `employeeID = {UUID}`
    The identifier of the employee
  - `surname = {string}`
    The surname of the employee
  - `name = {string}`
    The name of the employee
  - `role = {string}`
    The role of the employee
  - `salary = {double}`
    The salary of the employee
  - `nOperations = {integer}`
    The number of assigned operations for the employee. It must be `null` if the role is different from `Production line worker`.

  Optional:
  - `none`

- Success Response
  Upon success, the servlet returns a JSON object with the required information.

  Code: 201
  Content:
```
{"employee":
   {"employeeId":"da5ca713-c15c-4a19-b91b-e408c6af7d11",
   "surname":"Bedini",
   "name":"Marian",
   "nOperation":9,
   "salary":1500.0,
   "role":"Production line worker"
   }
}
```

- Error Response

  Code: 500 INTERNAL_SERVER_ERROR
  Content: {“Cannot create the employee: unexpected error.”: {“code”: “E5A1”, “message”: null}}
  When: if there is a SQLException is thrown, this error is returned.

  Code: 409 CONFLICT
  Content: {“Cannot create the employee: it already exists.”: {“code”: “E5A1”, “message”: null}}
  When: if there is a SQLException, in particular if the specified employee ID is already assigned to an existing employee, this error is returned.

  Code: 400 BAD_REQUEST

Content: {“Input media type not specified.”: {“code”: “E4A3”, “message”: “Content-Type request header missing.”}}
When: if the header does not contain a specification for the content type of the request body.

Code: 415 UNSUPPORTED_MEDIA_TYPE
Content: {“Unsupported input media type. Resources are represented only in application/json.”: {“code”: “E4A4”, “message”: “Submitted representation is {something}.”}}
When: if the header does not contain the JSON specification for the content type of the request body.

Code: 401 UNAUTHORIZED
When: The URI is accessible only to logged users with role “administrator”.

Delete an existing Employee

This functionality is exploited by the administrator in the employee management page. When called, it deletes an existing employee stored in the database.

- URL
  employee/{employeeID}

- Method
  DELETE

- URL Params

  Required:
  - `employeeID = {UUID}`
    The identifier of the employee

  Optional:
  - `none`

- Data Params
  No parameters are required in the body of the request.

- Success Response
  Upon success, the servlet returns a JSON object with the required information.

  Code: 200
  Content:
```
{"employee":
      {"employeeId":"da5ca713-c15c-4a19-b91b-e408c6af7d11",
      "surname":"Bedini",
      "name":"Marian",
      "nOperation":9,
      "salary":1500.0,
      "role":"Production line worker"
      }
 }
```

- Error Response

  Code: 404 NOT_FOUND
  Content: {“Employee {ID} not found.”: {“code”: “E5A3”, “message”: null}}
  When: if the user inserts an employee ID which is not stored in the database.

  Code: 409 CONFLICT
  Content: {“Cannot delete the employee: other resources depend on it.”: {“code”: “E5A4”, “message”: null}}
  When: if there is a SQLException, in particular if the specified employee ID is assigned to a resource which can’t be deleted due to other dependencies, this error is returned.

  Code: 400 BAD_REQUEST
  Content: {“Wrong format for URI /employee/{ID}: {ID} is not a UUID.”: {“code”: “E4A7”, “message”: “Requested operation DELETE.”}}
  When: if the inserted ID is not a UUID, this error is returned.

  Code: 500 INTERNAL_SERVER_ERROR
  Content: {“Cannot delete the employee: unexpected error.”: {“code”: “E5A1”, “message”: null}}
  When: if there is a SQLException is returned, this error is returned.

  Code: 401 UNAUTHORIZED
  When: The URI is accessible only to logged users with role “administrator”.