Software Architecture Description Document

By:

Pierre-Olivier Trottier (40059235)

Cedric Martens (40086877)

Camil Bouzidi (40099611)

Radley Carpio (40074888)

Bicher Chammaa (40096200)

Nimit Jaggi (40032159)

Matthew Kevork (40063824)

William Morin-Laberge (40097269)

Adrien Tremblay (40108982)

A report submitted in partial fulfillment of SOEN390.

Concordia University

March 17th, 2021

Table of Contents

1 Introduction

1.1 Identifying information

The Epic Resource Planner is an Enterprise Resource Planning application developed for the cycling industry. This application allows its users to keep track of the different parts and materials required to manufacture bicycle, keep track of finances and accounts due, as well as planning and predicting the future state of the company.

The purpose of this document is to provide details on the software architecture of the Epic Resource Planner. The document defines the Epic Resource planner as a client-server application, which uses networks to allow communication between the clients and servers.

1.2 Supplementary information

1.2.1 Scope

The scope of this document is to describe the architecture of the Epic Resource Planner. This document dives into the various aspects of this Enterprise Resource Planning application that have an impact on the architecture of the system. This includes the different stakeholders and their concerns, the different requirements for viewpoints and views, and the consistency of the requirements associated with the architecture.

1.2.2 Glossary

- Client: Software that communicates with the server
- COM: Serial port
- ERP: Enterprise Resource Planner
- Hot-Swap: Replacement while in operation
- Machinery: Industrial machines that can interact with the system
- MC: Model Controller Architecture
- Microservice: Microservice Architecture, where small services are loosely coupled.
- MVC: Model View Controller Architecture
- REST: Representational State Transfer

1.3 Other information

The client-server architecture model of the Epic Resource Planner consists of a client side and a server side. The client side consists of the user interface and allows users to interact with the product with ease. The server side is not accessible to the users unless they pass by the public REST API which is consequently access-restricted. The server contains the logic that allows the application to run smoothly. It also "hosts, delivers and manages most of the resources and services to be consumed by the client" [1].

[1] "What is Client/Server Architecture? - Definition from Techopedia," Techopedia.com, 25-Aug-2020. [Online]. Available: https://www.techopedia.com/definition/438/clientserver-architecture. [Accessed: 25-Jan-2021].

1.3.1 Architecture evaluations

Many architectures were evaluated before concluding that the client-server architecture would be the best for our use-case. The conclusions for each of the architectures evaluated were as follows:

a) **Microservices**

While microservices are currently very popular because they offer great scalability and maintainability, they also require a considerable amount of setup and hardware to be used to their full extent. It was also determined early on that, to reduce the hosting costs of the project, we would need to use as few physical resources as possible. It was therefore determined that since microservices would not be able to be used to their full extent, the advantages they brought to the table were simply not worth it over other patterns.

b) **Model-view-controller**

While pure MVC applications are great for desktop applications and sometimes even web, they often require more processing on the client-side which could lead to laggy and/or slow web pages when users have access to slower computers only which can lead to an undesirable user experience. Whilst it was decided that a pure MVC architecture was not the best choice for our application, most core concepts were kept in our final architecture choice (MVC concepts were used in the server part of the application).

c) **Monolithic**

Monolithic applications have become less and less popular in the last few years because of their less-than-ideal maintainability and extensibility. This therefore eliminated this architecture as a possibility since the Agile methodology works much better with architectures that can be easily extended because of its iteration-based approach.

d) **Client-server**

The client-server architecture was the last one to be evaluated. It was eventually decided that it would be the best choice for the Enterprise Resource Planning software since it allows us to incorporate the structured approach MVC offered, while also allowing a degree of flexibility. Even though the extensibility of a client-server application is not the same as that of a microservice application, it is much greater than that of a monolithic application.

1.3.2 Rationale for key decisions

As previously mentioned, the client-server architecture was chosen as the architecture pattern of choice for this project. That said, a few key changes or guidelines were added to make sure every requirement was covered, and that the architecture was as optimal as possible.

In fact, as the name suggests, the client-side and server-side code were split up to allow extensibility and allow for a lightweight client to be used. It also allows for a very easy deployment to the web with the possibility of easily integrating a progressive web application on the front-end, giving the users all the advantage of a native application at the same time as giving them the advantages of a traditional web page. This however leaves some freedom with how each part of the application are being implemented. We therefore chose to follow a loose version of the MVC model for the backend code implementation, where such execution was possible.

For the backend, we decided to follow the default pattern offered by our framework of choice (NestJS). The opinionated framework is setup to accept a Module, Controller, Service architecture. That said, this design can be very closely compared to the MVC pattern. The controllers in both designs have the same roles, the services in NestJS are equivalent to models in MVC, and since there are no views for a backend application, the view component is not comparable. Instead, the modules in NestJS are used to link controllers and services whilst avoid circular dependencies. We therefore have a "MC" architecture implemented in the backend, since the views are self-managed by the client (frontend).

For the frontend, we decided to follow React guidelines and use the design patterns prescribed by the framework. The chosen architecture can be described as a component-based one, where some elements are hot-swapped while the client is running, depending on the different routes being visited. Such route management is possible thanks to a React plugin called the React Router. Components can also share data through a central store offered by React Redux. This allows all our components to have independent implementations while having the possibility to share common data and to be dynamically rendered based on the route being visited by the user.

Finally, the last important part in the description of the architecture is the communication model used to communicate between the client and server. Such connection will occur through the means of a REST API offered by the backend. The client will indeed send HTTP requests to the backend, which will then process the requested information and send data back in the form of an HTTP response. No server-to-client connection (web-sockets for example) will be established since it was determined that such a connection would not be useful for our application. Any client other than the web application that wishes to communicate with the server, will be required to make authenticated HTTP requests. This is mainly how industrial machines will communicate to the web server.

2 Stakeholders and concerns

2.1 Stakeholders

Individuals that wish to see the prosperity of Epic Resource Planner are the stakeholders. This includes investors, inventory managers, accountants, salespeople, and system administrators.

Stakeholder	**Stakeholder Type**	**Description**
Investor	Owner	The investor is the stakeholder that invested money in the project and want to make money off their investments.
Manufacturing Manager	Operator, User	Manufacturing managers are responsible for making sure that the factories always have enough raw materials to produce parts. They are also responsible for identifying which materials need to be added to the current collection. The marketing managers work in tandem with production managers to create new parts.
Production Manager	Operator, User	Production managers are responsible for designing new products from pre-existing parts, requiring new parts and consuming parts in order to produce said products. They are responsible to deliver batches of products on time but are not responsible for transactions with customers.
Accountant	User	Accountants track the company's finances and want to make sure the company is profitable.
Salesperson	User	Salespeople find clients for the company and sell some final products to said clients.
System Admin.	Maintainer	System administrators supervise application logs, allocated permissions to certain users, create new users, etc.
Developer	Developer	Developers code the ERP System and take the design and implementation decisions.

2.2 Concerns

The following concerns are fundamental to the architecture of Epic Resource Planner:

1. What is the suitability of the architecture for achieving the system-of-interest's purpose(s)?
2. How feasible is it to construct and deploy the system-of-interest?
3. What are the potential risks and impacts of the system-of-interest to its stakeholders throughout its life cycle?
4. How is the system-of-interest to be maintained and evolved?

2.3 Concern–Stakeholder Traceability

Table 2.1: Example showing association of stakeholders to concerns in an AD.

	Manufacturing Manager	Production Manager	Production	Accountant	Salesperson	System Admin.	Investor	Developer
What is the purpose of the Epic Resource Planner?	**X**	**X**		**X**	**X**		**X**	
What is the suitability of the architecture for achieving the system-of-interest's purpose(s)?						**X**		**X**
How feasible is it to construct and deploy the system-of-interest?						**X**		**X**
What are the potential risks and impacts of the system-of-interest to its stakeholders throughout its life cycle?	**X**	**X**		**X**	**X**		**X**	
How is the system-of-interest to be maintained and evolved?						**X**		**X**

3 Viewpoints+

3.1.1 Use Case View

Audience: All the stakeholders are concerned with the use cases of the system

Area: This view describes what the users need in the system. This is portrayed through the relations between the actors and the system within scenarios.

Related Artifacts: *N/A*

3.1.2 Logical View

Audience: Developer

Area: Functional Requirements. Describes the system's architecture, its operation, and the functionality it provides to the end users.

Related Artifacts: GitHub Issues

3.1.3 Data View

Audience: System Administrator

Area: Describes how the data is stored and retrieved in the system.

Related Artifacts: Data Organization Diagram

3.1.4 Deployment View

Audience: Developers

Area: Describes the hardware the software will run on and the general physical layout of the system. It shows the physical interactions and connections between the different components.

Related Artifacts: Deployment Diagram

3.2 Concerns and stakeholders

3.2.1 Concerns

1. What is the suitability of the architecture for achieving the system-of-interest's purpose(s)?

   The architecture should be designed so that the web application loads under 5 seconds. This means that to achieve the goals it needs to be designed with performance in mind. Further, the architecture should be loosely coupled to increase maintainability.

2. How feasible is it to construct and deploy the system-of-interest?

   The deployment of the system is simple due to the advent of containers. The system can be deployed by simply uploading a new image or pulling from the version control repository and restart the server. The updates and maintenance can be done at night to avoid any disruptions.

3. What are the potential risks and impacts of the system-of-interest to its stakeholders throughout its life cycle?

   If the system of interest fails its task, it could result in loss to the investors because there might be delay and confusion within the company that uses the system.

4. How is the system-of-interest to be maintained and evolved?

   The ERP is planned to be used for many years to come. It is expected that updates and maintenance will periodically take place. Technical debt must be minimized, and clean code guidelines shall be followed in order to improve the maintainability. Further, a testable architecture shall be implemented with a unit test code coverage of over 50% for the webserver.

3.2.2 Typical stakeholders

- Users: Manufacturing Manager, Production Manager, Accountant, Salesperson
- Operators: Manufacturing Manager, Production Manager
- Owners: Investor
- Maintainers: System Administrator
- Developers: Developer

3.3 Architectural Goals and Constraints

The key requirements affecting the selection of the software architecture are the following:

1. The system is designed to handle almost every aspect of enterprise resource planning. If the system is down, it will result in financial lost to the company. For this reason, it is quintessential that the system operates within normal work hours. In other words, the system must be reliable. To ensure that the system is reliable, code coverage of at least 50% shall be enforced to avoid critical system failures.
2. The system shall be containerized for easier deployment. Docker was chosen to be appropriate as it is widely supported.
3. The system must interface with hardware such as machines using COM a port and support the upload of a .csv document to the current parts and materials. The webserver will only support a REST API. This means that the previously mentioned communications will require their own client that will make HTTP requests such as a Python script taking input from a COM port and create HTTP requests to the webserver.
4. The software is expected to have volatile and new requirements, this means that the system shall be flexible and extensible enough to accommodate requested changes while having a minimal impact on the current system.

4 Views

4.1 Use Case View

4.1.1 Actors

Manufacturing Manager

A manufacturing manager interacts with the system to create and update materials. If required, they will also interact with the manufacturing machines on the site of production.

Production Manager

A production manager interacts with the system to create and update materials.

Accountant

The accountant interacts with the system to keep track of accounts payable and the accounts receivable.

Salesperson

Salespeople are tasked to create client accounts and to sell them products.

System Administrator

The system administrator grants roles and monitors Epic Resource Planner

Manufacturing Machine

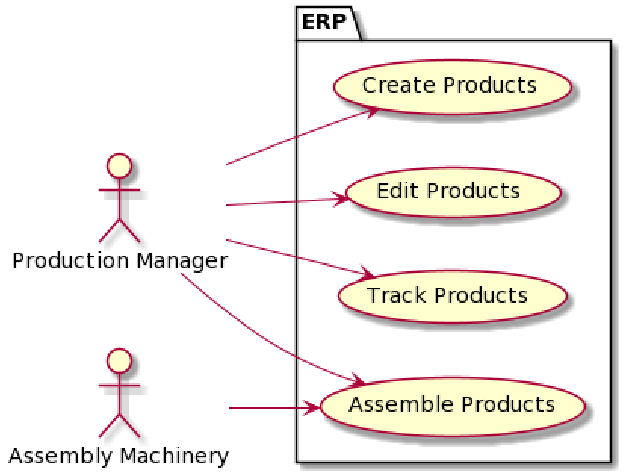
The manufacturing machines are an external subsystem that manufactures parts from raw materials. These machines can interact with the system via COM ports to update the materials and parts.

Assembly Machine (Machinery)

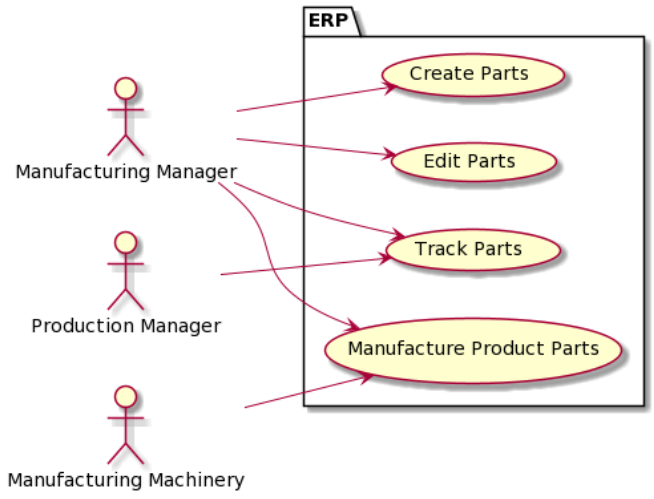
The manufacturing machines are an external subsystem that manufactures by combining parts. These machines can interact with the system via COM ports to update the number of parts and products available.

4.1.2 Use Case Realization

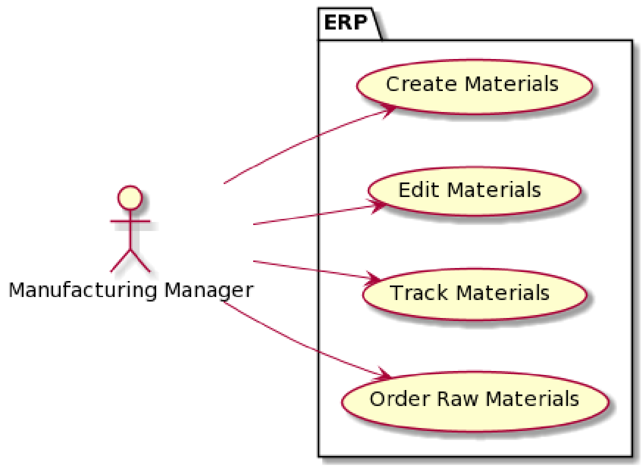
1. Manage Products



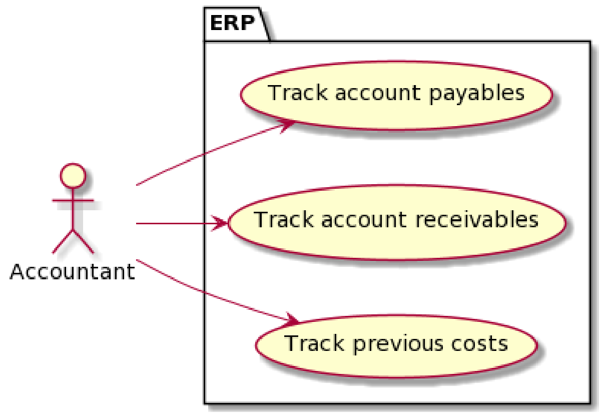
2. Manage Parts



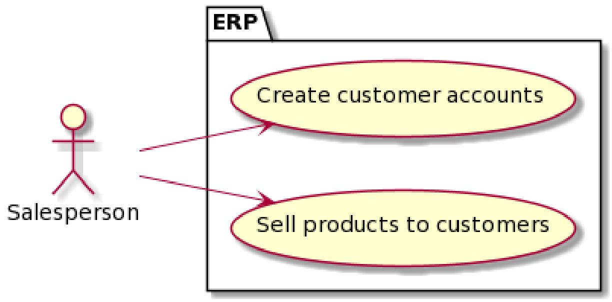
3. Manage Materials



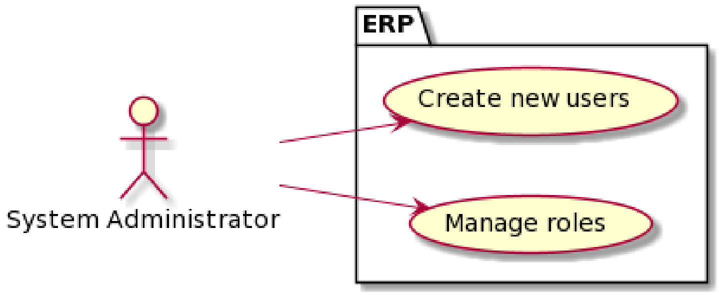
4. Track Finances



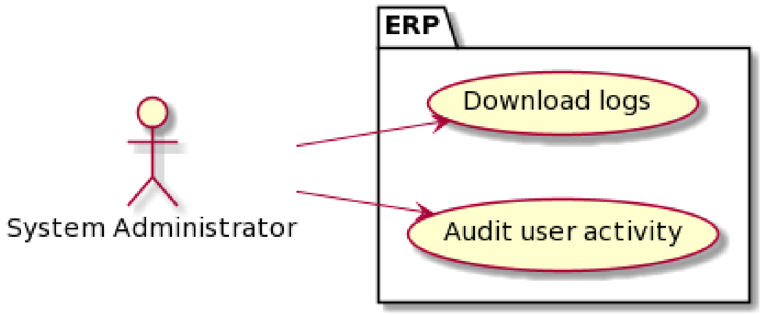
5. Manage Customers



6. Manage Users



7. Audit system



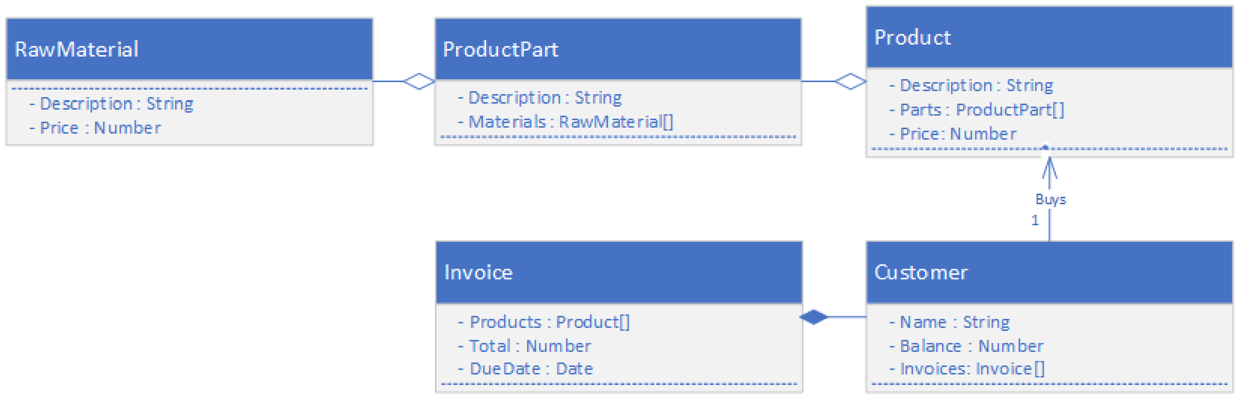
4.2 Logical View

4.2.1 Domain Model

While the domain model looks simple, it represents all the real-world components that are being translated into the software. As can be seen, the relationship between products and parts, and parts and materials are both aggregations as opposed to compositions. This is because the components are seen from a manufacturer's standpoint where materials can exist without being parts and parts can exist without being full products. However, it is also possible to build a product from a few parts and build parts from a few materials.

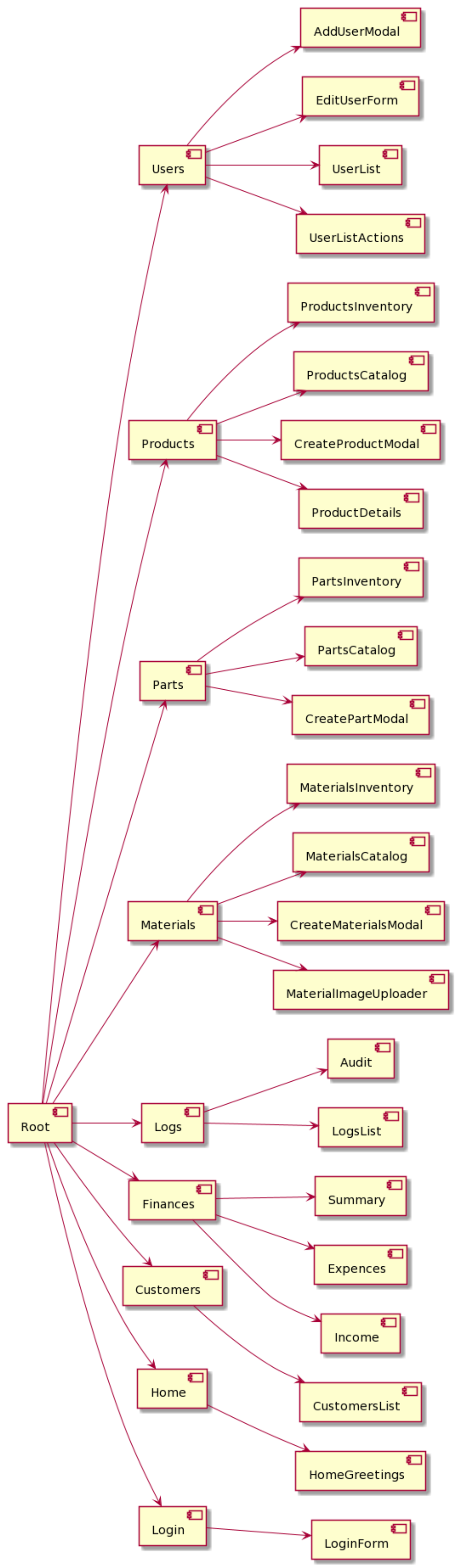
On the other hand, invoices and customers are linked through a composition relationship because invoices only exist if the customer they are attached to exists as well.

Lastly, the only association present is the one between the customers and products, where a customer can buy a product. This action will generate a new invoice and change the customer's balance.



4.2.2 Logical Component Diagram (Frontend)

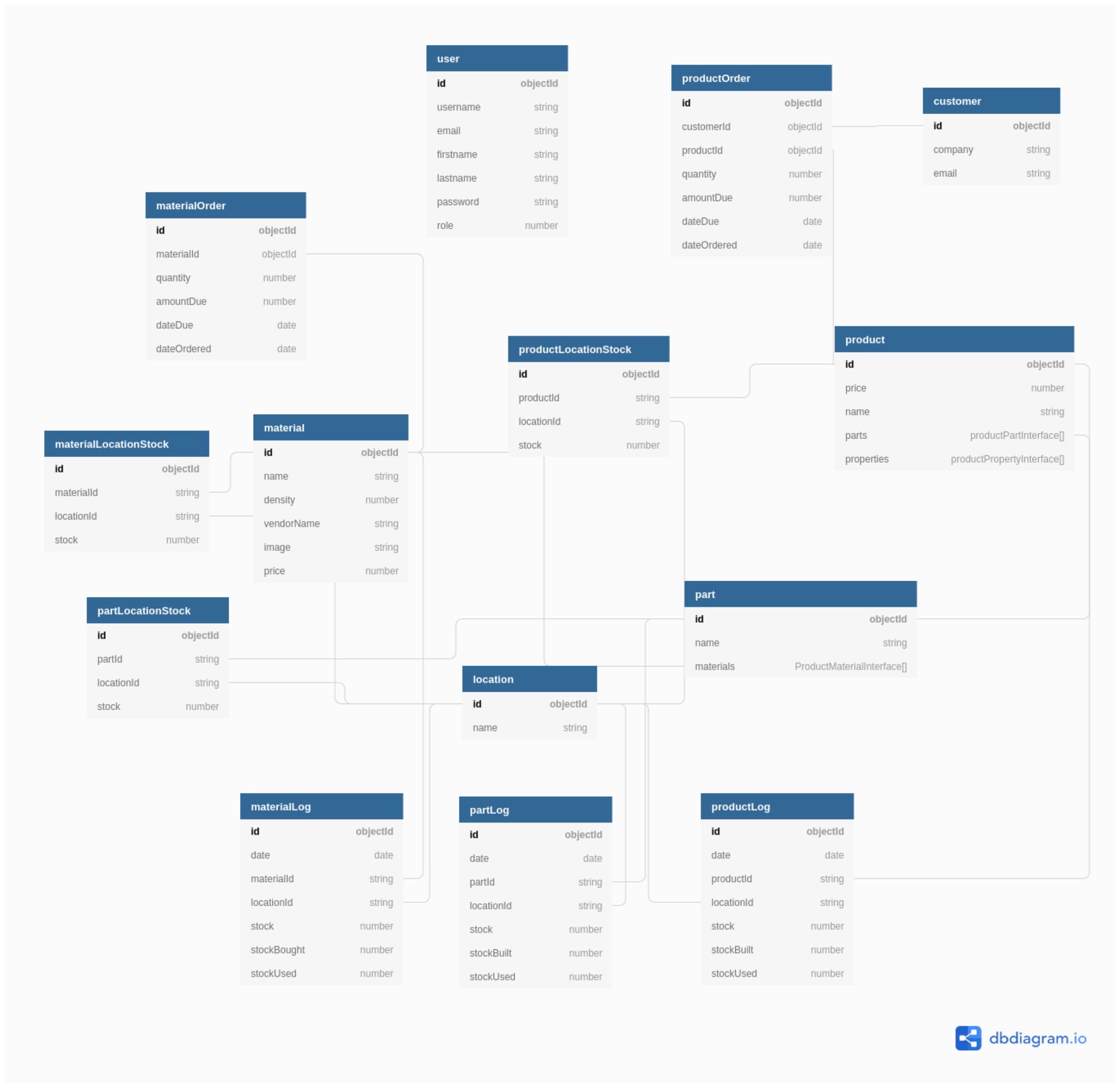
The diagram below describes the component structure of the React frontend used for the project.


AddUserModal
EditUserForm
Users
UserList
UserListActions
ProductsInventory
ProductsCatalog
Products
CreateProductModal
ProductDetails
PartsInventory
PartsCatalog
Parts
CreatePartModal
MaterialsInventory
MaterialsCatalog
Materials
CreateMaterialsModal
MaterialImageUploader
Audit
Root
Logs
LogsList
Summary
Finances
Expences
Customers
Income
CustomersList
Home
HomeGreetings
Login
LoginForm

4.3 Data View

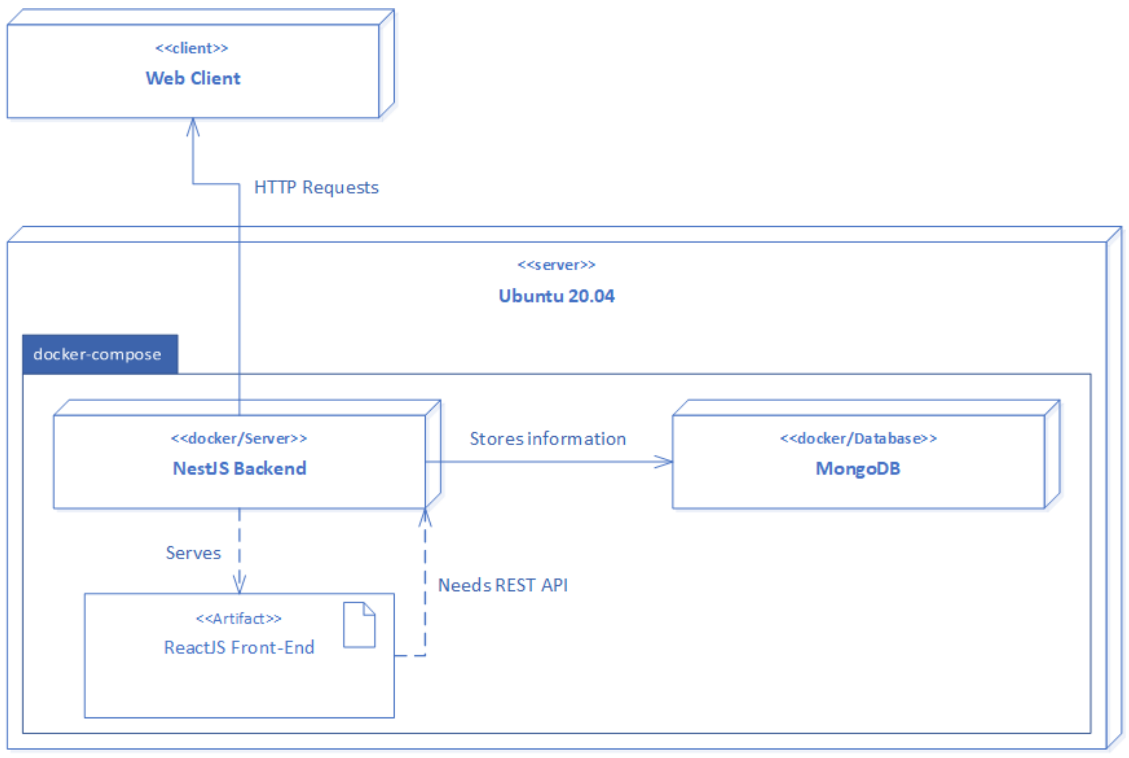
Data organization diagram

The diagram illustrates the data's organization within the system. The different objects and their multiplicity are shown.

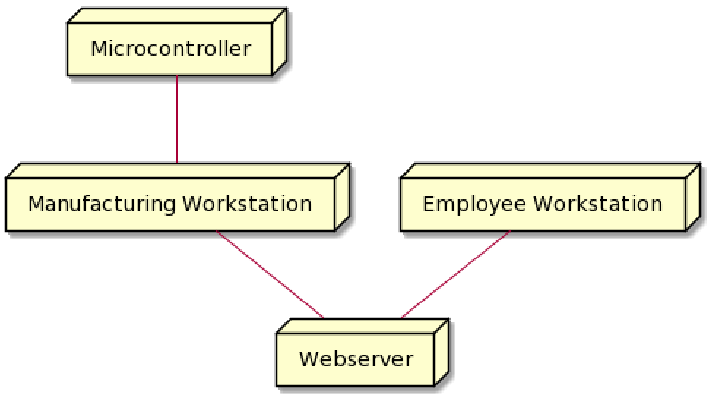


4.4 Deployment View

The system is composed of two main parts. The server and the web client. The webserver runs on the operating system Ubuntu 20.04 deployed by Docker using docker-compose. HTTP Requests are sent from the web client to interact with the webserver.



Additionally, it is possible for a machine to communicate with the webserver using COM ports. This is done by relaying information through the COM port to a computer referred as the "Manufacturing Workstation" and this computer sends HTTP Requests to the webserver.



5 Consistency and correspondences

5.1 Known inconsistencies.

The following inconsistencies were found in the document. They are marked as “Resolved” if they have been fixed.

- **Resolved (February 2021): inconsistencies between user stories and user personas**. After the original submission of the document, it was noticed that the user stories required more than 4 personas (salesperson, administrator, inventory manager, accountant). The main issue was that the inventory manager’s role was too broad, as they were in charge of both the manufacturing of parts from raw materials and the production of products. As a fix, this persona was split into the production manager and manufacturing manager personas in the Github repo and the SAD had to be updated as well.
- **Resolved (February 2021): Confusion in machinery available**. It was noticed that a single type of machinery was available as an actor for the Use Case Diagrams of this document, which would not be accurate: in reality. We need two categories of machines, the first one being for manufacturing, and the second one for production. This was updated in the SAD.
- **Resolved (March 17 2021)**: **Inconsistency between logs, orders, stocks and locations in Views**. After deliberation within the team, it was decided that additional data schema had to be implemented between sprint 2 and 3. In order for documentation to be update, the new schema (orders, updated logs and locations) were added in the SAD.

5.2 Correspondences in the AD

Data view and Logical Component View: The data view and the logical component view have a decent amount of overlap, most concepts like parts, products, customers and more are found in the two views. This means that the abstraction for these entities is similar in both the front-end architecture and the backend webserver.

Domain Model and Component Diagram: There are some overlapping parts between the domain model and the component diagram. Especially concerning the idea of products and parts. This overlap shows that we have modeled the components close to how it is in the physical world, using appropriate object-oriented programming.

Overlapping parts in deployment view: In the deployment view, the two diagrams have overlapping parts where the webserver is shown in more depth in the first diagram while in the second diagram it is simply called “webserver”. The reason that the second diagram has less information is for simplicity, to understand how the main physical components are laid out. Additionally, the data to be transferred between the backend and front-end (from the data view) is represented as an HTTP request.

5.3 Correspondence rules

Concerning the data view and logical component view’s overlap in terms of concepts, this is not an issue and should be part of the project. This makes it easy for the programmer to understand both architecture and the objects sent from the front-end can be like the back end. This also increases cohesion between the front-end and back-end, thus ensuring better and quicker feature implementation. Additionally, it must be noted that the data (explained in the data view) to be transferred is only highlighted as HTTP requests in the Deployment View in order not to overcrowd that view with unnecessary details/