📘 AutoPartsPro Schema Rationale and ER Diagram

🎯 Purpose

This document explains the purpose of each table in the AutoPartsPro schema, the design rationale behind it, and how it relates to other tables. It also includes a complete dbdiagram.io-compatible schema.

📦 Tables and Rationale

1. **customers**
   - **Purpose**: Stores basic customer data.
   - **Rationale:** The workshop requires users to register their own vehicles, associate orders to them. For instance the customer might hear something weird in their vehicle, that triggers the event of them creating an order for reparation. To keep track of whose vehicle is from whom, we need a customers table. The relationships included allow the customer to have different vehicles registered, which is the case for a family where the husband normally takes both cars to the workshop.
   - **Relations**:
     - `vehicles.customer_id → customers.id`
     - `customer_orders.customer_id → customers.id`

2. **vehicle_types**
   - **Purpose**: Reference table for reusable vehicle configurations (make, model, year, version).
   - **Rationale:** It is common for shops to have several of the same type of vehicle, therefore this information can be reusable if it is stored in its own table. This table is related to that of the vehicle. For our example, vehicle type stores information such as model, company, year, color, and version, while the vehicles table stores mainly specifics of that car, for instance the VIN and License plate. Together this 2 tables represent the vehicle someone owns.
   - **Relations**:
     - `vehicles.vehicle_type_id → vehicle_types.id`

3. **vehicles**
   - **Purpose**: Represents a physical vehicle owned by a customer. Includes `license_plate` and optional `vin`.
   - **Relations**:
     - `customer_id`, `vehicle_type_id`, `customer_orders.vehicle_id`

4. **skus**
   - **Purpose**: Standard part identifiers that group interchangeable part instances. Includes `size` and `description` at the SKU level.
   - **Rationale:** SKUs allow to represent a general part, is kind of like the template for a part. Therefore it has the common information for a part, such as the storage size it requires.
   - **Relations**:

- `used by parts`, `job_parts`

5. **parts**
   - **Purpose**: Individual part instances with creation date and inventory status.
   - **Rationale:** Allows to represent the specific instance of a SKU. for instance this is useful to store the particularities of the part, such as when it was acquired (created), how many we have of each and combined with the table of part_market_data, even the price of acquisition and define a sell price. Basically it allows to represent an inventory
   - **Relations**:
     - `sku_id`, `used by part_market_data`, `vehicle_parts_catalog`

6. **part_market_data**
   - **Purpose**: Tracks economic data for each specific part.
   - **Relations**:
     - `part_id`
7. **vehicle_parts_catalog**
   - **Purpose**: Maps compatible parts to vehicle types
   - **Rationale:** Each Vehicle type is related to a set of sku, therefore this table is used to store those relationships. So for instance if a workshop employee needs to repair a car, it could easily search or relate for parts they own and that are related to a vehicle. The same is truth for customers trying to search a part for their own car. The system is thought to support for both repairs or direct sale to the customer.
   - **Primary Key**: Composite `(vehicle_type_id, part_id)`

8. **customer_orders**
   - **Purpose**: Captures the customer's description of what needs service.
   - **Rationale:** As previously mentioned, customers can generate an order for reparation of their vehicle. This is what is called an order. They can add their own observations of what is going on.
   - **Relations**:
     - `customer_id`, `vehicle_id`

9. **workshop_orders**
   - **Purpose**: Represents the workshop's formal repair/service response.
   - **Rationale:** While the customer might think they understand what the car requires, it is the actual experts the ones that can do that only, that is why thai table is necessary, it captures a general overview of what the car requires and also can be used to attach different specific jobs to it.
     This table also allows us to store the actual profit and costs related to the order. For instance, the car might need maintenance, which requires jobs like changing the engine oil, changing the transmission oil and changing the tires position.

     The workshop order table is used precisely to store all this information and also determine if the task is completed, therefore, collecting the profit.

     Note: a direct sale could also be registered as a type of order.
   - **Relations**:

        - `customer_order_id`, `order_jobs.order_id`

10. **jobs**
    - **Purpose**: Represents a unit of work to be performed.
    - **Rationale:** Jobs represent what is actually need to be done, by steps. For instance, the car might need maintenance, which requires jobs like changing the engine oil, changing the transmission oil and changing the tires position. Each of those jobs is associated to both a profit and a cost: fixed cost, such as the cost of a particular job to be done by the employees and variable costs such as the price of a part used in the job and that was bought in a particular time.
    - **Relations**:
        - `used in order_jobs`, `job_parts`

11. **order_jobs**
    - **Purpose**: Join table between `workshop_orders` and `jobs`. Supports many-to-many relationships.
    - **Rationale:** This table allows to relate a workshop order to a job, basically allows many to many relationships. That said, a workshop order can have multiple jobs to be done to be considered complete, while also those jobs being generic, can be related to multiple other workshops orders.
    - **Relations**:
        - `order_id → workshop_orders.id`, `job_id → jobs.id`

12. **job_parts**
    - **Purpose**: Links jobs to the SKUs they consume.
      Rationale: As mentioned, each job could require a particular part, therefore it is connected to the sku table, so that it is a generic part, not a particular the one required by a job.
    - **Relations**:
        - `job_id`, `sku_id`

13. **profit_loss**
    - **Purpose**: Tracks financial performance per workshop order.
    - **Rationale:** Basically allows to support total profit and loss of the company as time passes. For instance, each day we can estimate a variable cost loss due to the storage cost of each of the parts in our inventory. Or costs due to parts acquisition. Also, if an order is completed the profit of such could be tracked.
    - **Relations**:
        - `order_id → workshop_orders.id`

✅ Implementation and future work 🛠️

High level 🗺️

Current Implementation: ✅

My general overview is of a system that is thought to be maintainable, and future proof, so there are some considerations of the above defined system that not yet implemented in the current MVP. That being said, I will first mention what is currently implemented:

- [ ] Dashboard:
  Allows Employees to visualize relevant information of the business, such as :
  a. Next order to be completed -considering it as the optimal from a profit pov-
  b. Current profits
  c. When completing the optimized by profit workshop order, the profit is updated and the next most optimized order is calculated.
  d. 🛠️future:
     i. There are placeholders such as: estimated cost per day
     ii. Total inventory
     iii. Rent per month
     iv. Total workshop orders
     v. Workshop parallel capacity of orders
- [ ] Parts:
  Allows the employees to register skus and parts related to those
  a. 🛠️future:
     i. Register a part to the market price (price of acquisition and sell price)
- [ ] Orders:
  Allows both Customers and Employees to register orders
  a. Customers can register
  b. Customers can register their vehicle or select from already existing vehicles types
  c. Customers can submit an order for their or theirs vehicles.
  d. An Customer/Employee button was added to simulate different portals and what information would be presented to customers or employees
  e. Employees can create workshop orders
  f. Employees can create jobs
  g. Employees can assign jobs to workshop orders
  h. 🛠️future:
     i. Registering a job to a particular sku is left for future work. And costs are now only considered from fixed labor cost

Future Work: 🛠️

- [ ] **System to support costs associated to parts being stored.**
- [ ] **logging/signup/auth system**
- [ ] **Webhooks to keep data updated without interactions**
- [ ] **Multitenant**
- [ ] **Select different optimization strategies to select the next order**
- [ ] **Complete orders not associated to the most optimal one**
- [ ] **Have parallel workshop capacity supported, so multiple orders can be done at once**

- [ ] **Have orders with a limited max time to take. For instance “urgent” orders.**
    - [ ] **Having support to a scheduler algorithm so that orders don't “starve”**

Low Level Implementation 🔧

Tools/Stack:

- [ ] **DB:** Postgresql**:** I chose it mainly because I already had knowledge in it and is a common db for sql relationships
- [ ] **DB admin:** pgadmin
- [ ] **Backend:** FASTAPI/Python: I used this backend because it allows several advantages, such as self documented endpoints (Swager UI), industry proveen use, speed and because I am familiar with it. I implemented both the crud and the optimization endpoints with it.
- [ ] **Frontend:** Next.js: I used it because it is also industry standard react based framework, it supports several advantages such as the file based routing, and API routing capabilities, middleware and optimization for production ready environments for the continous maintainability of the project.
- [ ] **Containerization:** Docker: is industry proveen and can handle this project, also allows for easy creation of environments such as the one used for the backend, support to run the DB, the DB administration portal and also the frontend, all from the same mechanism 😉

🤓 Explanations

DB:

DB design Choises are mentioned on the first section “**AutoPartsPro Schema Rationale and ER Diagram**”

Backend:
For the CRUD operations and Services provided by the Backend Please refer to the attached document on Endpoints.pdf

In relation to my design choices, I went first by going full in into a MVC architecture, so that the data was independent of the user, not directly accessible and managed by the backend, which not only produced the CRUD endpoints, but also endpoints for the services such as the Optimal order to be performed.

My project looks like this:
```
-Src
---api
----public
-----customer_orders
--------------api.py
--------------crud.py
--------------models.py
--------customers
--------...
--------...
--------...
----services
----------------services.py
--------------api.py
----utils
--------------crud_base.py
--------------api_base.py
---database.py
```

---app.py
---config.py
---main.py
The idea behind splitting the project structure like this, was to keep the structure as suggested by best practices.

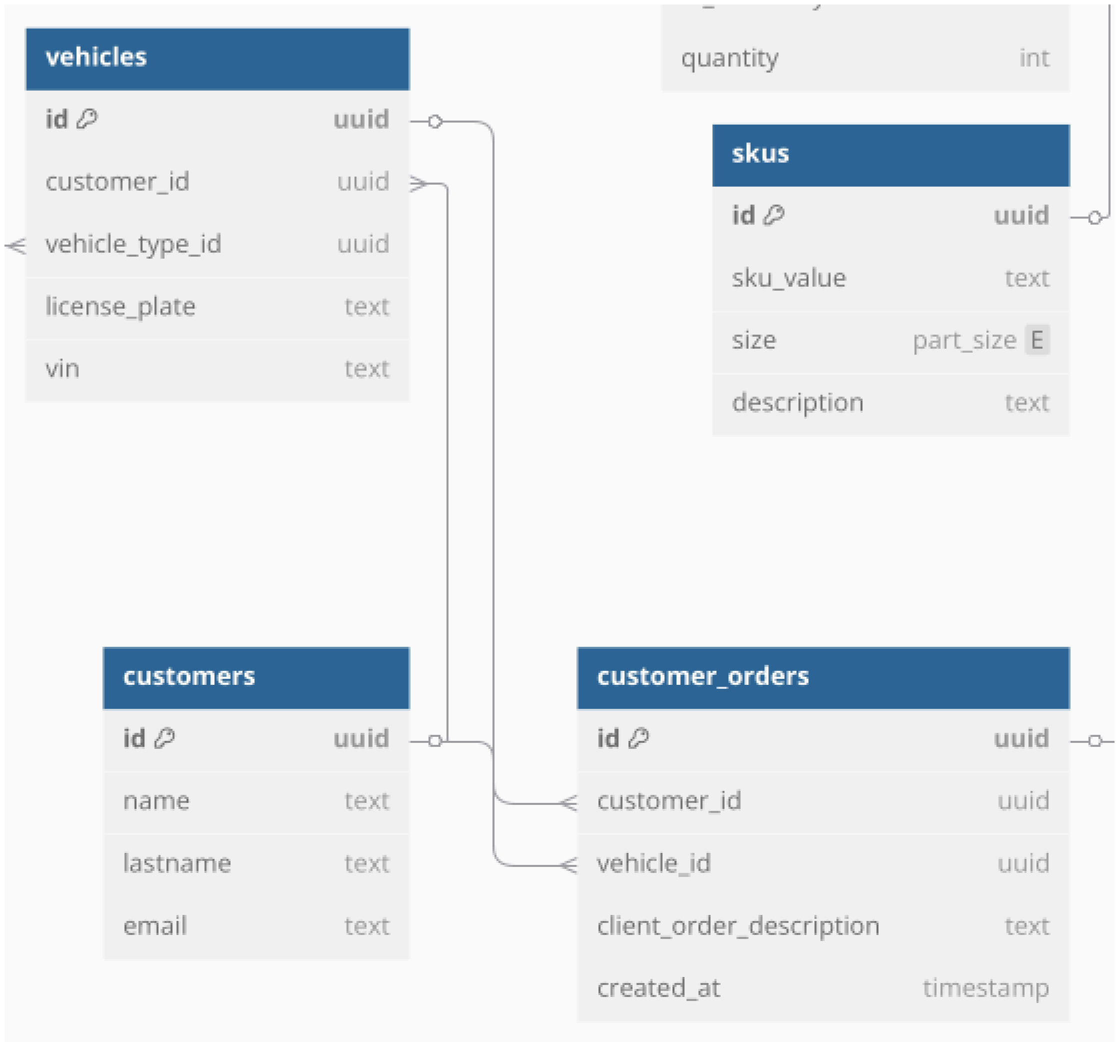
Each entity is represented by a model, that sets the supported values of table in the DB, therefore keeping consistency between both, and limiting what was visible by the user.

For instance the customers table is represented with the following:

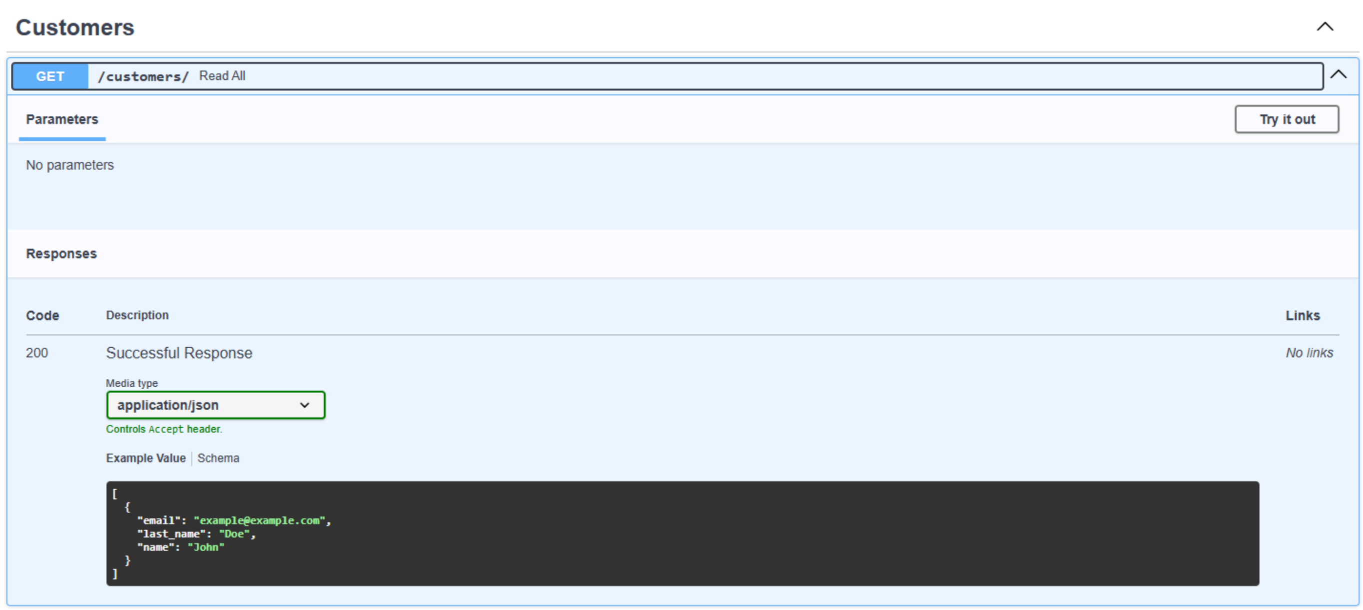
api > public > customers > models.py > ...

```python
class CustomerBase(SQLModel):
    name: str = Field(max_length=100)
    last_name: str = Field(max_length=100)
    email: EmailStr = Field(max_length=100, unique=True)
    model_config = {
        "json_schema_extra": {
            "examples": [
                {
                    "name": "John",
                    "last_name": "Doe",
                    "email": "example@example.com",
                }
            ]
        }
    }

class Customer(CustomerBase, table=True):
    id: uuid.UUID | None = Field(default_factory=uuid.uuid4, primary_key=True)
    created_at: datetime = Field(default_factory=lambda: datetime.now(timezone.utc))
    updated_at: datetime = Field(default_factory=lambda: datetime.now(timezone.utc), nullable=True)
    vehicles: List["Vehicle"] = Relationship(back_populates=None, cascade_delete=True)  # one-way relationship
    customer_orders: List["CustomerOrder"] = Relationship(back_populates=None, cascade_delete=True)

class CustomerCreate(CustomerBase):
    pass

class CustomerRead(CustomerBase):
    id: uuid.UUID
    name: str
    last_name: str
    email: EmailStr
class CustomerUpdate(CustomerBase):
    name: Optional[str] = None
    last_name: Optional[str] = None
    email: Optional[EmailStr] = None
```

Which does capture both the relationships and the fields of the table.



This system allows to limit users to interact with the created_at and updated_at fields only in read mode, which is safer, so in a way this classes hierarchy limit how the users are going to interact with the tables through the API, and also set the example schemas to use as they appear in the documentation of the API documentation.



One other design decision was to implement reusable logic as much as possible due to time constraints.

One of those reusable logic comes from the CRUD service operations. Which support the actual communication with the DB and performing the operations in the tables.
For the particular customers table it looks like this:

api > public > customers > crud.py > ...

```python
from api.utils.crud_base import CRUDBase
from api.public.customers.models import *

crud_customer_type = CRUDBase[Customer, CustomerCreate, CustomerUpdate](Customer)

```

A Crud base class was created to minimize code writing, it accepts any generic Model type, and representations -update, read, and create- for the model.

```python
ModelType = TypeVar("ModelType", bound=SQLModel)
CreateSchemaType = TypeVar("CreateSchemaType", bound=SQLModel)
UpdateSchemaType = TypeVar("UpdateSchemaType", bound=SQLModel)

class CRUDBase(Generic[ModelType, CreateSchemaType, UpdateSchemaType]):
    def __init__(self, model: Type[ModelType]):
        self.model = model

    def get(self, db: Session, id: UUID, raise_not_found: bool = True) -> Optional[ModelType]:
        obj = db.get(self.model, id)
        if not obj and raise_not_found:
            raise HTTPException(status_code=404, detail=f"{self.model.__name__} not found")
        return obj

    def get_all(self, db: Session) -> List[ModelType]:
        return db.exec(select(self.model)).all()

    def create(self, db: Session, obj_in: CreateSchemaType) -> ModelType:
        try:
            obj = self.model(**obj_in.model_dump())
            db.add(obj)
            db.commit()
            db.refresh(obj)
            return obj
        except Exception as e:
            raise HTTPException(status_code=400, detail=f"Failed to create {self.model.__name__}: {str(e)}")

    def update(self, db: Session, db_obj: ModelType, obj_in: UpdateSchemaType) -> ModelType: ···

    def delete(self, db: Session, id: UUID, raise_not_found: bool = True) -> Optional[ModelType]:
```

All of the entities use this class as a base, and extend it as necessary.

Finally to keep the crud services decoupled from the api routes, those are handled in another file.

Again, for the customer entity, it looks like this:

api > public > customers > api.py > ...

```python

router = get_crud_router(
    model=Customer,
    create_schema=CustomerCreate,
    update_schema=CustomerUpdate,
    read_schema=CustomerRead,
    crud=crud_customer_type,
)

```

And the base implementation of the get_crud_router is generic enough to support all entities.

api > utils > api_router.py > ...

```python
ModelType = TypeVar("ModelType", bound=SQLModel)
CreateSchemaType = TypeVar("CreateSchemaType", bound=SQLModel)
UpdateSchemaType = TypeVar("UpdateSchemaType", bound=SQLModel)
ReadSchemaType = TypeVar("ReadSchemaType", bound=SQLModel)

def get_crud_router(
    *,
    model: Type[ModelType],
    create_schema: Type[CreateSchemaType],
    update_schema: Type[UpdateSchemaType],
    read_schema: Type[ReadSchemaType],
    crud: CRUDBase
) -> APIRouter:
    router = APIRouter()

    @router.get("/", response_model=list[read_schema])
    def read_all(db: Session = Depends(get_session)):
        return crud.get_all(db)

    @router.get("/{item_id}", response_model=read_schema)
    def read_one(item_id: UUID, db: Session = Depends(get_session)):
        obj = crud.get(db, item_id)
        if not obj:
            raise HTTPException(status_code=404, detail="Not found")
        return obj

    @router.post("/", response_model=read_schema)
    def create(item: create_schema, db: Session = Depends(get_session)):
        return crud.create(db, item)

    @router.patch("/{item_id}", response_model=read_schema)
    def update(item_id: UUID, item: update_schema, db: Session = Depends(get_session)):
        db_obj = crud.get(db, item_id)
        if not db_obj:
            raise HTTPException(status_code=404, detail="Not found")
        return crud.update(db, db_obj, item)

    @router.delete("/{item_id}", response_model=read_schema)
    def delete(item_id: UUID, db: Session = Depends(get_session)):
        obj = crud.delete(db, item_id)
        if not obj:
            raise HTTPException(status_code=404, detail="Not found")
        return obj
```

All of the entities use this class as a base, and extend it as necessary.

On the otherhand the services are a different type of interaction with the DB, therefore, it does not compel well enough with the CRUD operations, which forces it to have its own implementation but with the same principles in mind, **separation of concerns** between the routes and the services.

The routes look like this:

api > services > api.py > get_optimized_next_order

```python
@router.get("/optimized_order_by_expected_profit")
async def get_optimized_order_by_expected_profit():
    try:
        optimized_order = await calculate_optimized_order_by_expected_profit(False)
        return {"optimized_order": optimized_order}
    except Exception as e:
        return {"error": str(e)}

@router.get("/all_order_profits")
async def get_all_expected_profits():
    try:
        profits = await calculate_all_expected_profits()
        return {"order_profits": profits}
    except Exception as e:
        return {"error": str(e)}

@router.get("/order_state_counts")
async def get_order_state_counts():...

@router.get("/optimized/next_order")
async def get_optimized_next_order():
    try:
        optimized_order = await get_optimized_order()
        return optimized_order
    except Exception as e:
        return {"error": str(e)}
```

While the services are in another file

```python
async def get_optimized_order() -> dict:
    async with httpx.AsyncClient() as client:
        try:
            optimized_order_response = await client.get(f"{base_url}/services/optimized_order_by_expected_profit")

            if optimized_order_response.status_code != 200:
                raise Exception("Failed to fetch optimized order")


            optimized_order_data = optimized_order_response.json()
            optimized_order_id = optimized_order_data['optimized_order']['best_order_id']
            expected_profit = optimized_order_data['optimized_order']['expected_profit']
```

Each available route in services is well documented in the openapi standard

Services

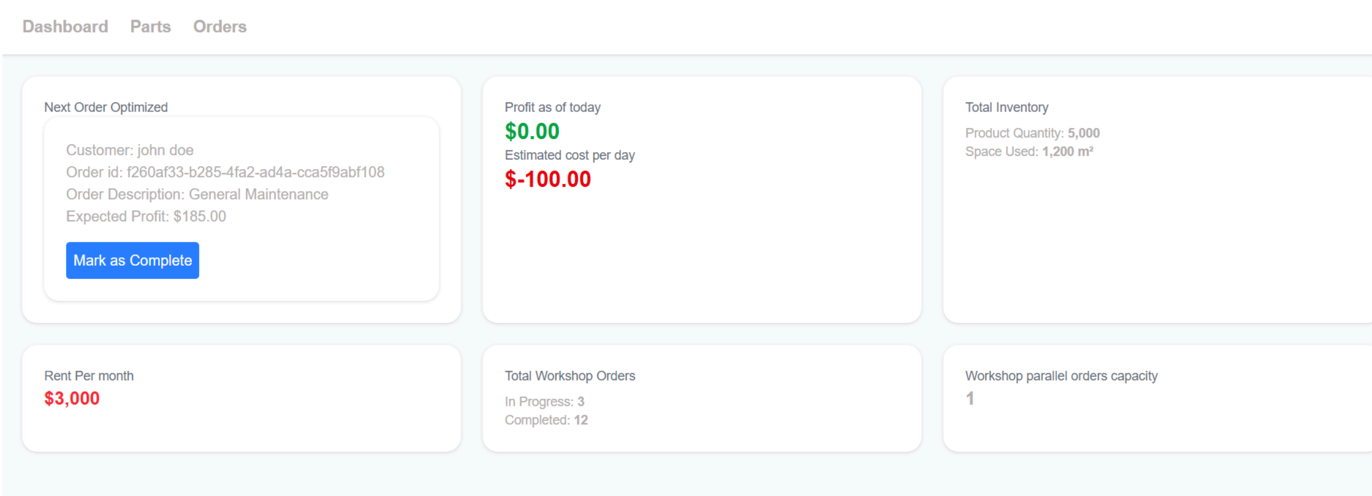
- GET `/services/order/{order_id}/revenue` Get Order Revenue
- GET `/services/order/{order_id}/fixed_costs` Get Order Fixed Costs
- GET `/services/order/{order_id}/total_profit` Get Order Total Profit
- GET `/services/total/revenue` Get Total Revenue
- GET `/services/total/fixed_costs` Get Total Fixed Costs
- GET `/services/total/profit` Get Total Profit
- GET `/services/optimized_order_by_expected_profit` Get Optimized Order By Expected Profit
- GET `/services/all_order_profits` Get All Expected Profits
- GET `/services/order_state_counts` Get Order State Counts
- GET `/services/optimized/next_order` Get Optimized Next Order

🤓 Frontend structure rationale

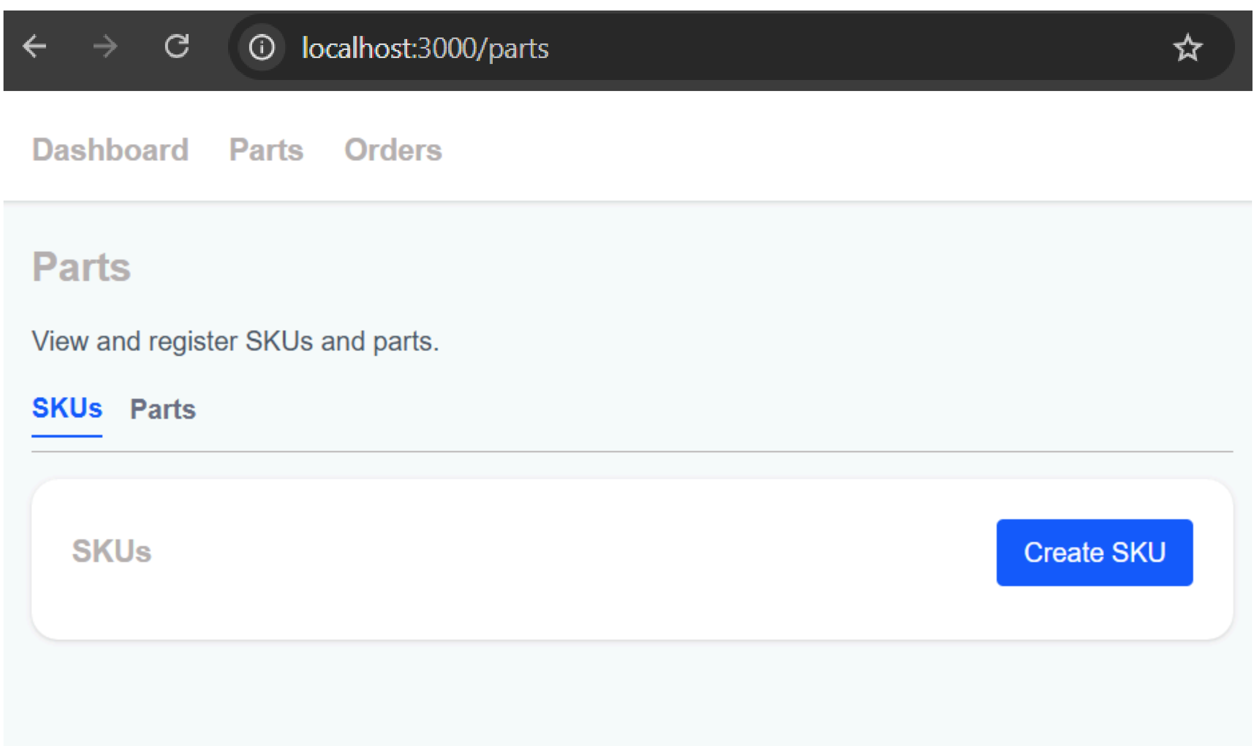
High Level Review

Modern looking styles:

Dashboard to see the next optimal order, the profit of the day and other parameters of the workshop:

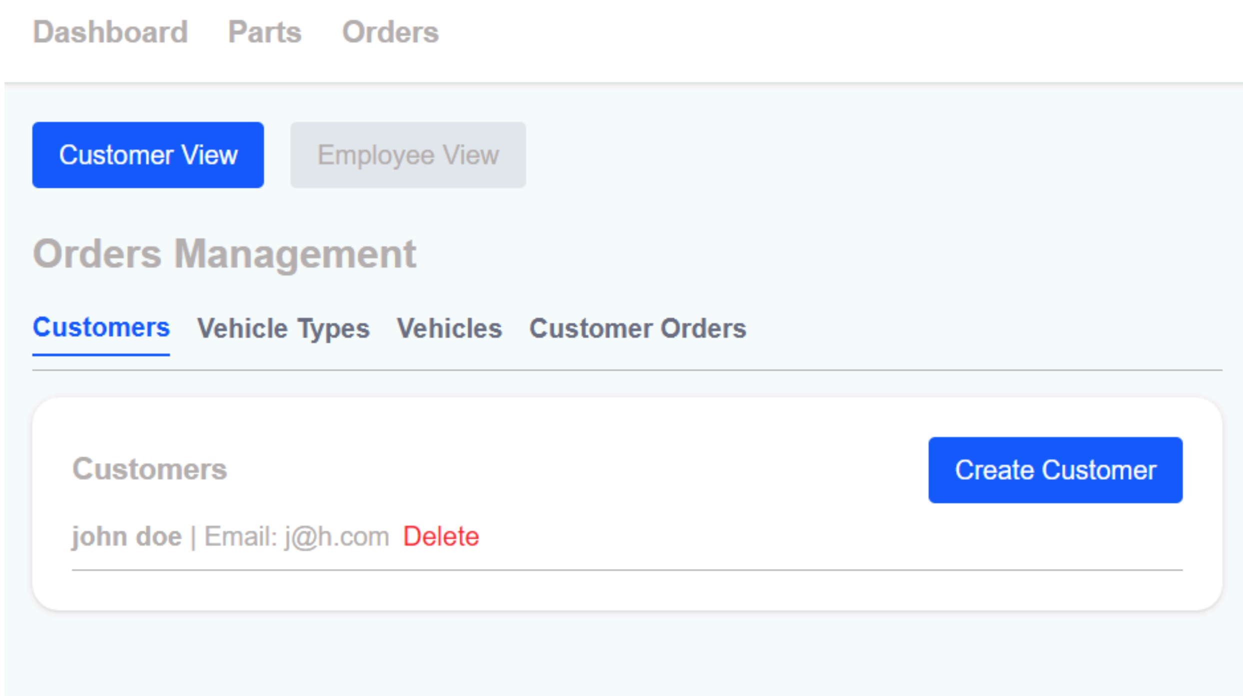


Parts page to see and register new parts and skus:

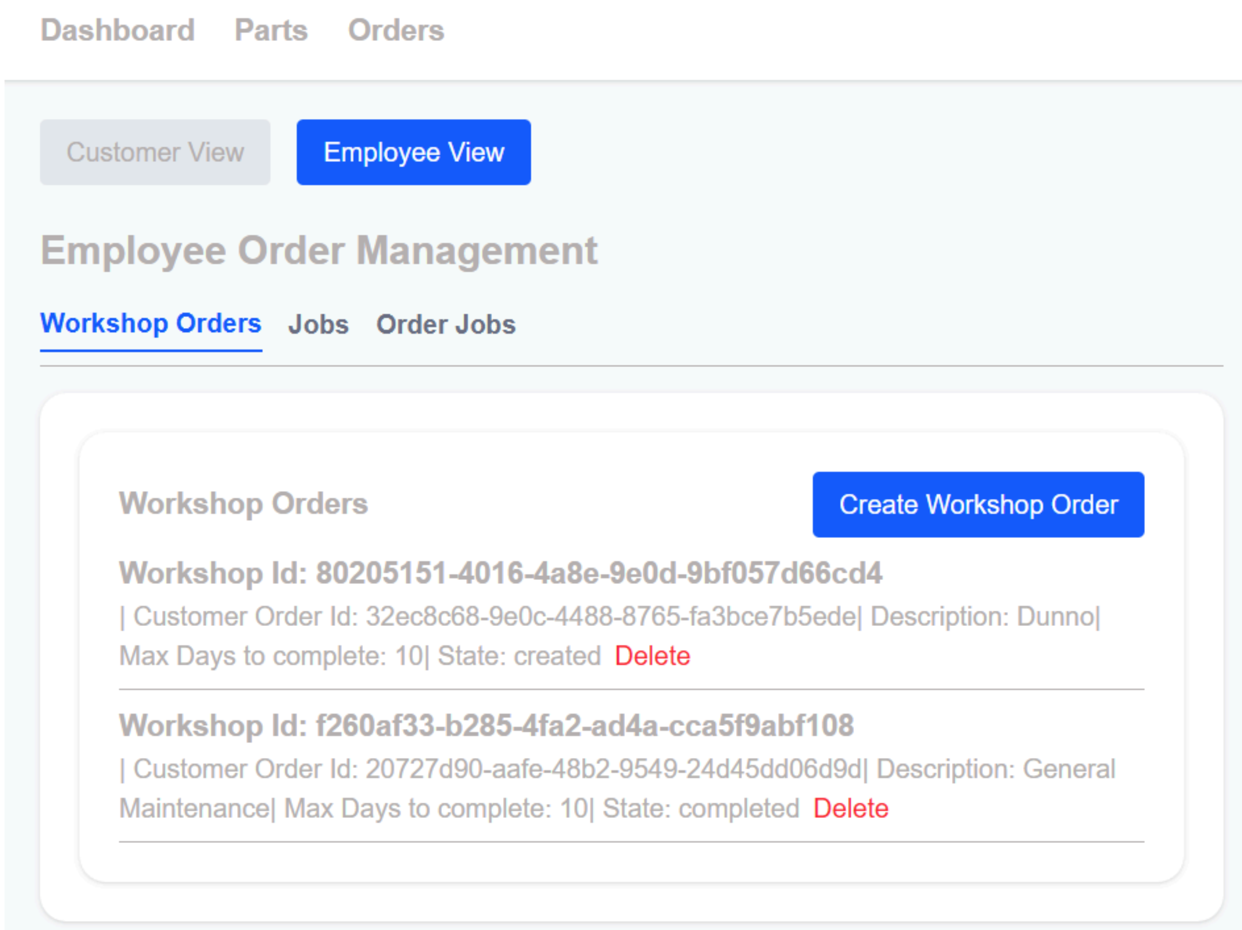


Selector between Customer or employee (simulate login and roles)

Customer page for registering orders:



Employee page for registering orders:



Low Level Review

For me the approach was to keep in mind the MVC architecture in mind, that is separation of concerns, so everthing related to UI, would be in the Next.js frontend.

My general rationale was as follows:
Have a general component folder to hold all of them. These would in turn render in the pages designated.
For instance since most of the pages required forms, I designed a general base form and try to use it as much as possible to reuse

code.

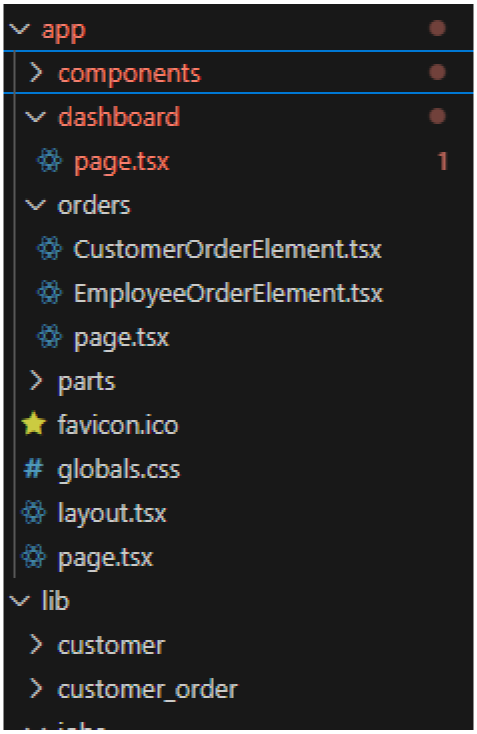
For instance:

app > components > BaseForm.tsx > ...

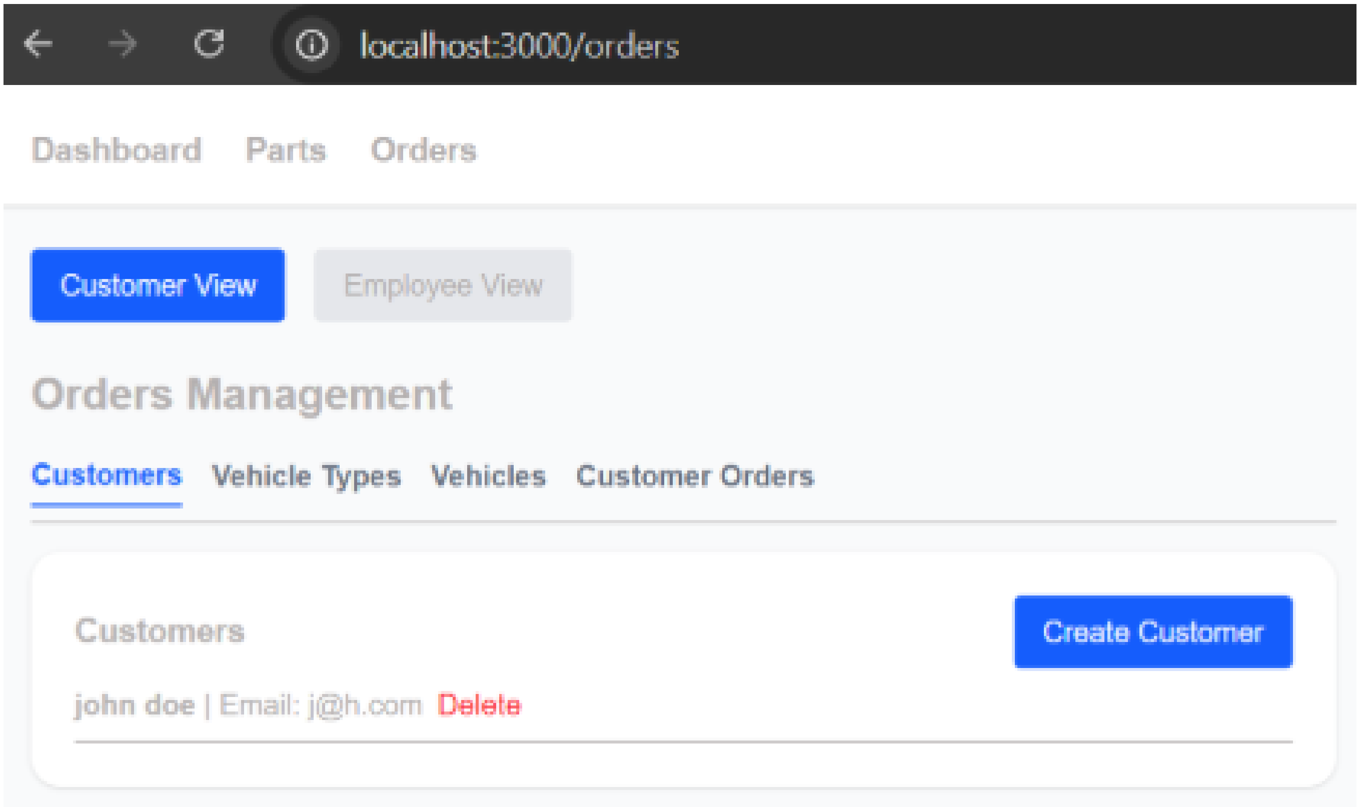
```tsx
interface BaseFormProps {
  fields: {
    [key: string]: string;
  };
  onSubmit: (fields: { [key: string]: string }) => void;
  loading: boolean;
  onCancel: () => void;
  fieldLabels: { [key: string]: string };
  placeholders: { [key: string]: string };
  enumOptions?: { [key: string]: string[] }; // Add enumOptions prop for enum-based fields
}

const BaseForm: React.FC<BaseFormProps> = ({
  fields,
  onSubmit,
  loading,
  onCancel,
  fieldLabels,
  placeholders,
  enumOptions = {}, // Default to empty object if not provided
}) => {
  const [formFields, setFormFields] = useState(fields);

  const handleFieldChange = (field: string, value: string) => {
    setFormFields((prev) => ({
      ...prev,
      [field]: value,
    }));
  };

  const handleSubmit = (e: FormEvent) => {
    e.preventDefault();
    onSubmit(formFields);
  };

  return (
    <form onSubmit={handleSubmit} className="space-y-4 mb-4">
      {Object.keys(fieldLabels).map((field) => (
        <div key={field}>
          <label className="block text-sm font-medium text-gray-700">{fieldLabels[field]}</label>
          {enumOptions[field] ? (
            // Render a select input for enum fields
            <select
              value={formFields[field]}
```

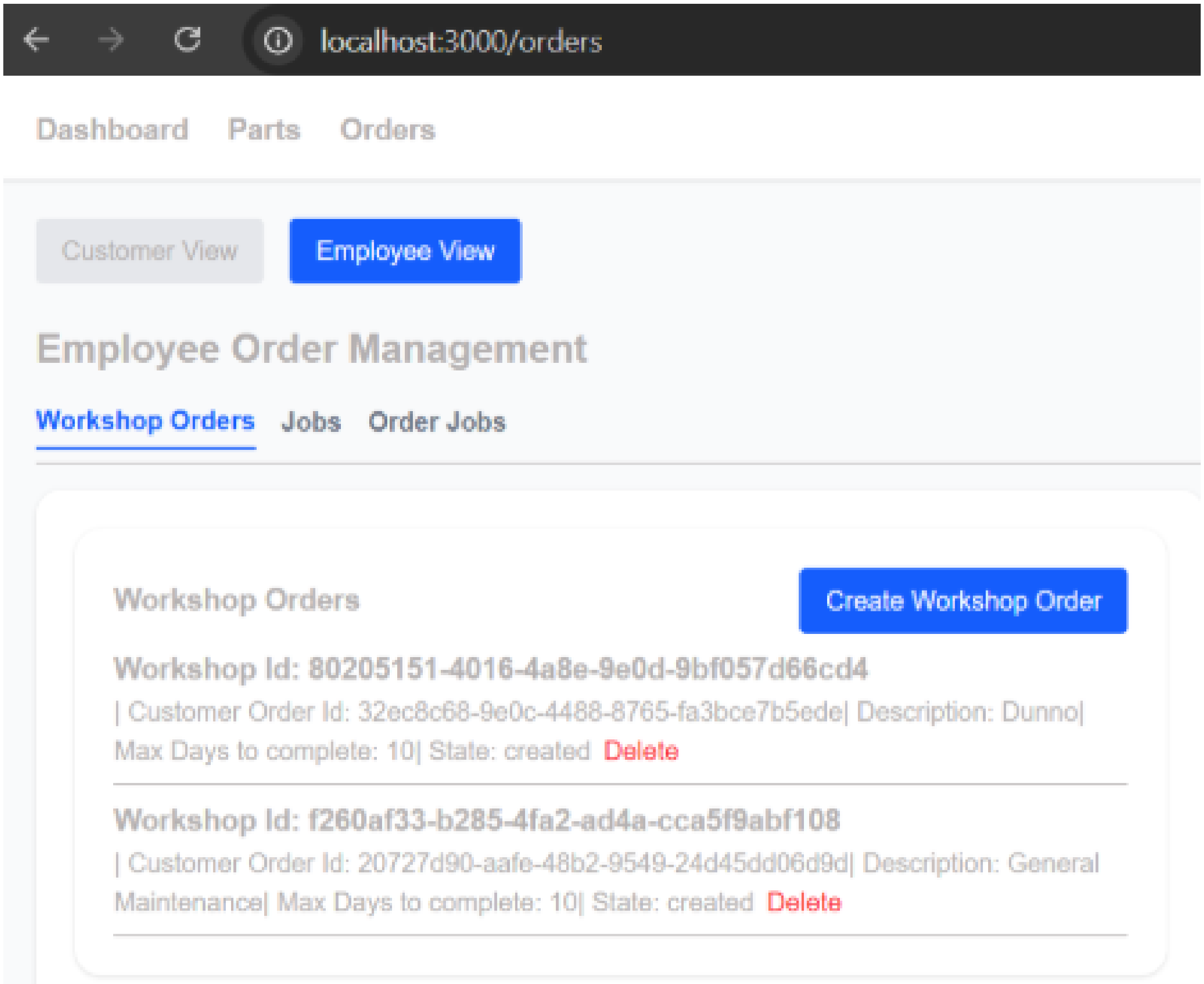
The API interaction layer was in a different folder and was called when needed from the pages general components.

As mentioned the pages in next.js can hold the components and with that form more complex components to be rendered in routes with the same name as the ones in the folder:



As can be seen, the orders folder has a page component with CustomerOrderElement and EmployeeOrderElement inside that are used to render the Orders route.


localhost:3000/orders
Dashboard
Parts
Orders
Customer View
Employee View
Orders Management
Customers
Vehicle Types
Vehicles
Customer Orders
Customers
Create Customer
john doe | Email: j@h.com
Delete



🤓 Optimization endpoint for repair order selection

The optimization endpoint algorithm is basic right now, but the system design and DB allow for future changes in the optimization strategy.
As of right now, the basic idea is to obtain the workshop order which's jobs expected revenue and costs add up to the maximum profit. **That would be the next order to do.**

To accomplish this, the following is done:
- retrieve all the workshop orders from the DB
- for each job in it calculate the profit
- determine the workshop order with the biggest profit
- set the endpoint to send that workshop's id

That´s it!

😐 Tradeoffs of my design

- Different languages imply more work, so it is not completely ideal to work the backend in a different language as the frontend, specially if the team is small. For larger projects or heavily decoupled projects it is not a big issue
- Self hosting is challenge if money is not a big problem, probably a better approach would have been using a serverless db such as supabase instead of my own postgres instance
- Python is not the fastest executing, so probably for super heavy applications GO would have been a better idea, but for simplicity python was fine.

😎 Business challenge solutions

Definitely this project was a challenge especially considering that it required so much in so little time:

- Designing a scalable DB
- Complete the CRUD and services endpoints
- Complete a working frontend
- Document everything

As I see it, the best approach to handle this kind of problems is to look for patterns to generate reusable code as much as possible, which was what I intended.
Other important thing that worked for me is try to put on the users shoes.