MDE ONLINE PLATFORM

Service-Oriented Software Engineering

Martina Nolletti
Giordano Tinella
11/07/2023

Summary

Description of Project

We designed a system to support the principles and practices of Model Driven Engineering (MDE). We provide a robust system that enables users to create 1 metamodel (M2) and 1 model (M1) that conforms to the defined metamodel. By operating within the MDE paradigm, the platform facilitates the development of domain specific modeling solutions, empowering users to define custom metamodels composed of packages, classes, and attributes. Additionally, the platform enables the instantiation of defined meta-classes consisting of objects and associated values, promoting higher abstraction and efficiency in system design and development.

System Limitations.

1) A class cannot extend and cannot be extended.

2) A class can only contain attributes.

3) An attribute type can be Boolean/String/Number/Char and its lower bound and upper bound are set to 1.

4) Enumerations are excluded.

5) Sub-packages are excluded.

6) A user can only work with 1 metamodel (M2) and 1 model (M1).

System Goals.

1) Handling user authentication and registration.
2) Handling metamodel:
    a. Create/delete/update package.
    b. Create/delete/update class.
    c. Create/delete/update attribute.
3) Handling model:
    a. Create/delete/update object.
    b. Automatic create/delete/update value.
    c. Check object conformance.

Architecture

We adopt a microservices architecture developed to provide a scalable and modular solution for managing various components in the system. The architecture consists of multiple providers, prosumers, and additional services, all communicating via RESTful APIs.
The MDE's service providers include:

- ePackage Service.
- eClass Service.
- eAttribute Service.
- eObject Service.
- eValue Service.

Each of these services is responsible for handling specific operations related to their respective components, such as managing packages, classes, attributes, objects, and values.
The service prosumers include:

- M2 Service.
- M1 Service.
- Aggregator.

These services play a crucial role in aggregating and processing data. The M2 service aggregates data from packages, classes, and attributes to provide a higher-level view. On the other hand, the M1 service aggregates data from objects and values, offering a comprehensive perspective on specific instances within the system. The aggregator service (**asynchronous service**) combines the data from the M1 and M2 services, running them in parallel for efficient processing. By merging the results, the aggregator service provides a unified view of the metamodel and the model, enabling further computations or transformations as needed. In addition to the core services, the architecture includes:

- Configuration Service.
- Discovery Service.
- Gateway Service.

The Configuration Service retrieves configuration settings from a Git repository. This service ensures consistent and up-to-date configurations across the entire system, enhancing flexibility and maintainability. To enable service discovery and load balancing, the architecture incorporates the Discovery Server, it acts as a registry for all the services, facilitating dynamic scaling, resilience, and efficient load distribution within the microservices architecture. For secure and centralized access to the microservices, the Gateway Service serves as an API gateway. It consolidates and routes requests from clients to the appropriate microservices. This gateway service simplifies client interaction, enhances security, and adds an additional layer of control. The architecture also includes one service provider for handling users:

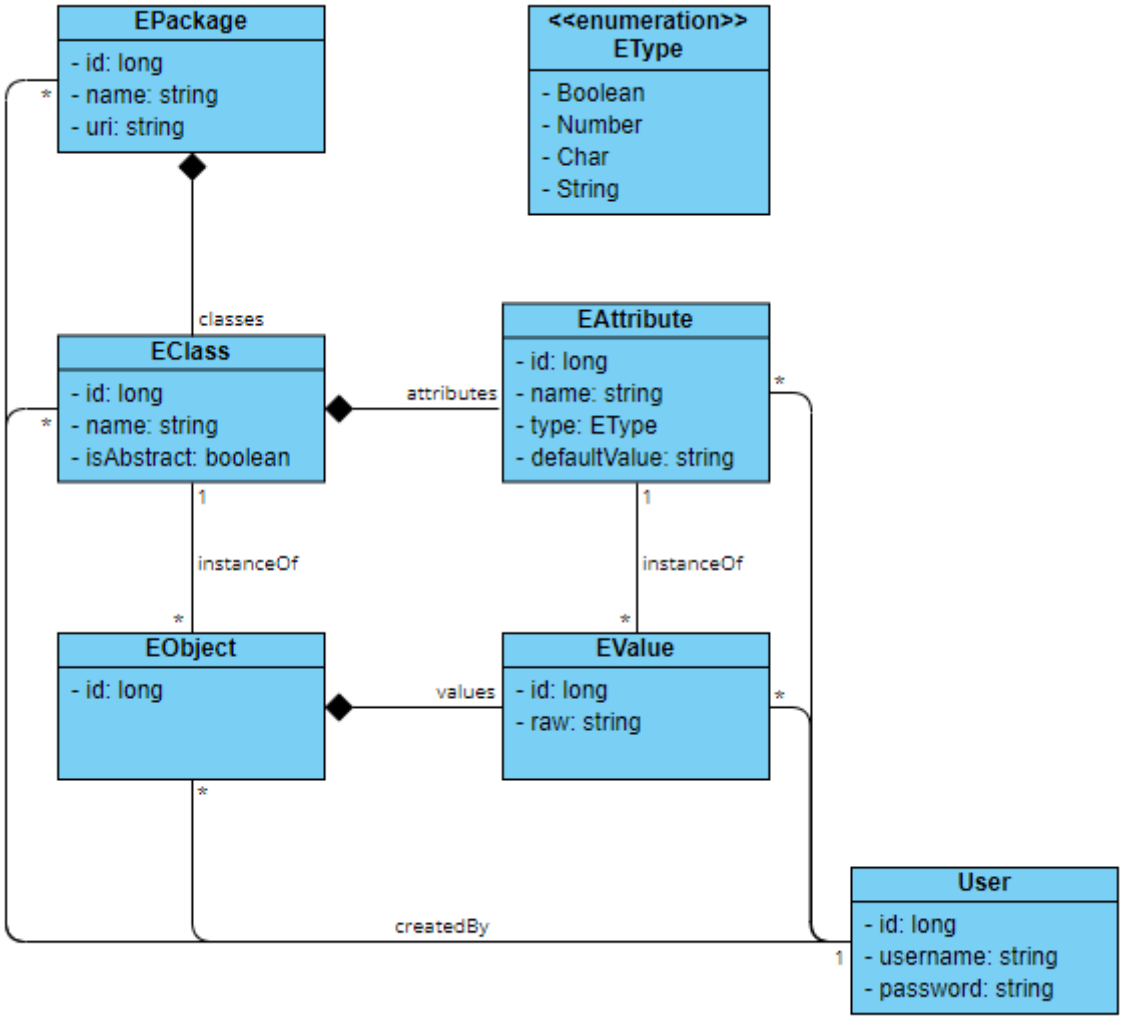
- User Service.

The User Service is responsible for user authentication and registration within the system. It allows users to authenticate themselves and register new accounts. This service ensures secure access and proper user management within the microservices architecture.

Diagrams

We have defined several diagrams that help more in understanding the activities and functionalities of our system.

Class diagram

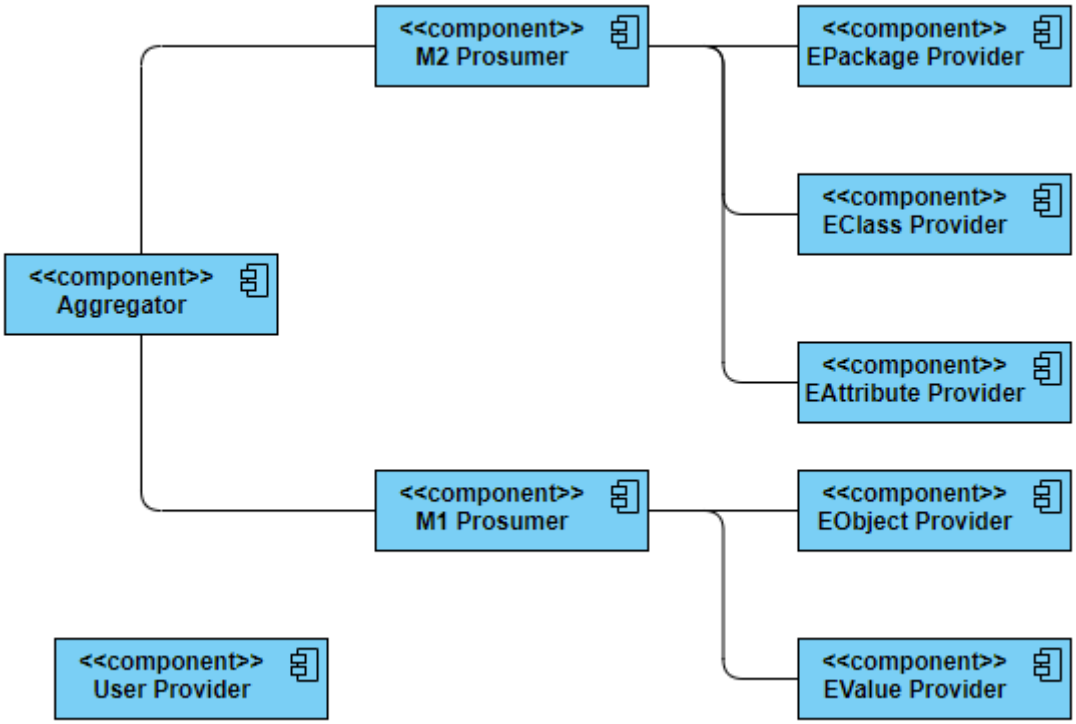


The main classes in our diagram are:

- **EPackage**. This class defines the package of our metamodel and, in addition to containing an id and a name, consists of a URI of type String and a userId of type Long, which stands for the user who created it.
- **EClass**. This class defines the classes present in our metamodel. It too, in addition to an id and a name, contains an isAbstract attribute of type Boolean (to figure out whether the class is abstract or not), a packageId of type Long (to know which package it belongs to), and a userId of type Long (to know which user created it).
- **EAttribute**. This class defines the attributes present in classes. In addition to an id and a name, it contains an attribute type of type String (indicates the type of attribute between String/Boolean/Char/Number), a defaultValue of type String (indicates the default value of an attribute, if no value is assigned to it), a classId of type Long (indicates the class to which it belongs) and a userId of type Long (to know which user created it).

- **EObject**. This class defines the objects in our model. In addition to an id, it contains an instanceOf attribute of type Long (indicates the class to which it belongs) and a userId of type Long (to know which user created it).
- **EValue**. This class defines the values present inside our objects. In addition to an id, it contains an instanceOf attribute of type Long (indicates the value of a class to which it belongs), an objectId (indicates the object to which it refers), a raw (indicates the value), and a userId of type Long (to know which user created it).
- **EUser**. This class defines the users in our system. In addition to an id, it contains a username attribute of type String (indicates the username under which the user can access the system) and a password of type String (indicates the password used by the user to be able to access the system).

Component diagram

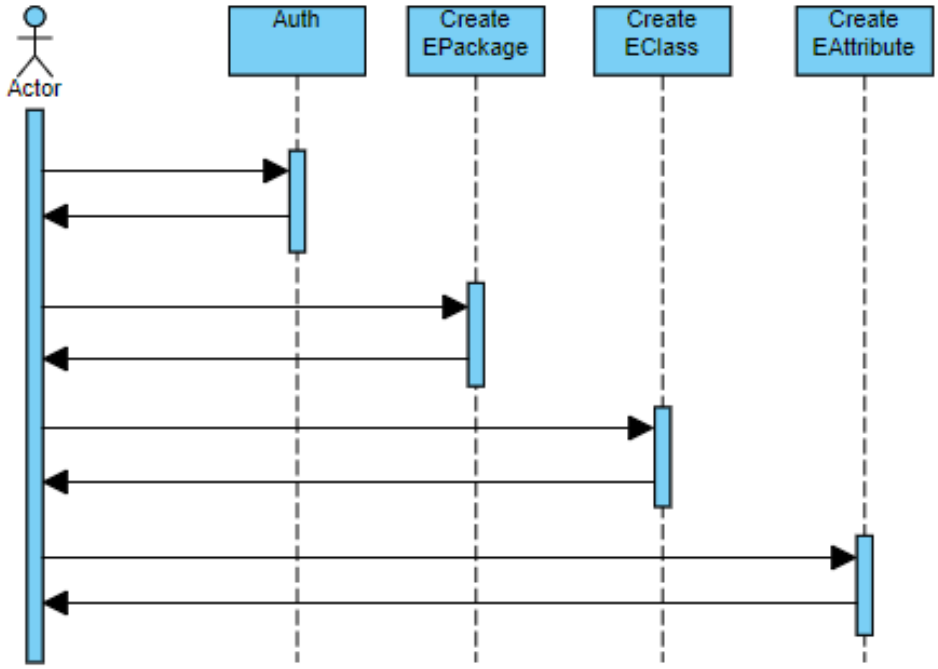


Our component diagram has four basic components:

- **Aggregator**. It takes the metamodel and model data and aggregates them together.
- **Metamodel**. Generated through data taken from the **EPackage** component, classes from the **EClass** component, and attributes from the **EAttribute** component.
- **Model**. Generated through the data taken from the **EObject** component, from the values of the **EValue** component. In addition, the **EObject** component is an instance of the **EClass** component.
- **User**. Through the username and password can enter the system and perform operations.

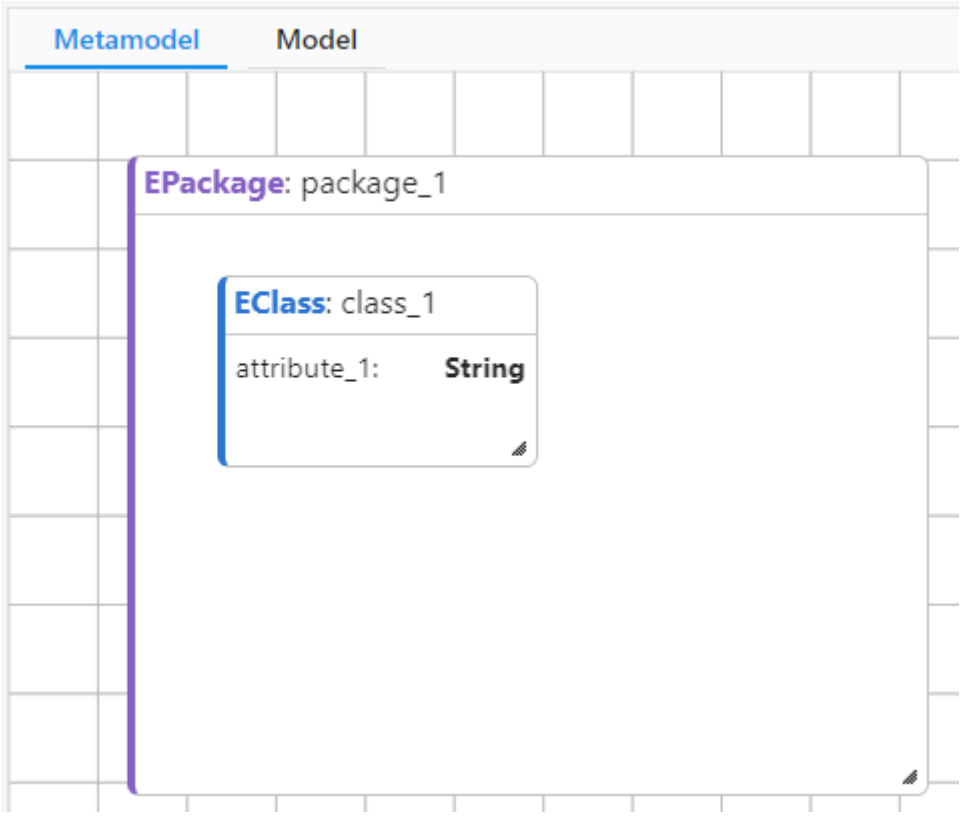
Sequence diagrams

First scenario

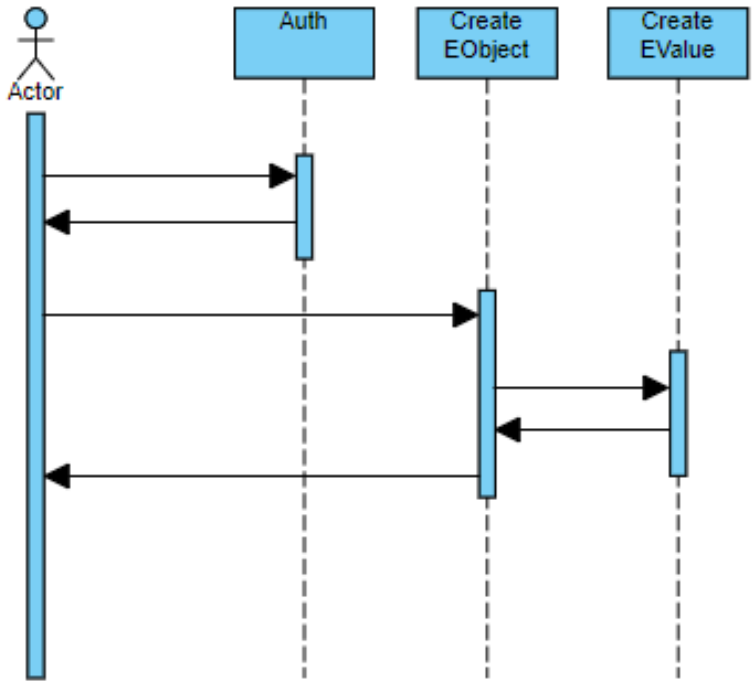


This sequence diagram represents the scenario where a user authenticates within the system and starts working on the metamodel by going to create a package, create a class within the package, and finally create an attribute within the class.

The result of these actions brings the user face to face with such a scenario:

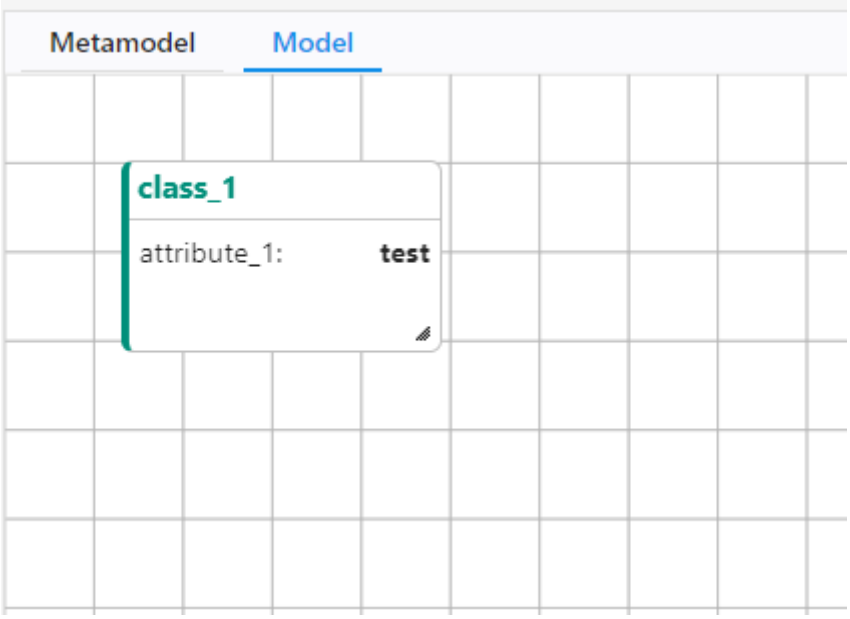


Second scenario



This sequence diagram represents the scenario where a user authenticates within the system and starts working on the model (we assume he has already define a meta-class) by going to create an object, and after this action the value will be created automatically.

The result of these actions brings the user face to face with such a scenario:



Used Technologies

The microservices architecture is developed using Spring leverages various technologies to enable the creation, update, deletion, and other operations on different components. Each service exposes endpoints to interact with the system. The following technologies were employed in the implementation:

- **Spring Boot**: The Spring Boot framework provides a simplified and opinionated approach to building microservices. It offers a range of features and tools that streamline development, configuration, and deployment processes.
- **Spring Boot Dev Tools**: This toolset enhances developer productivity by providing features such as automatic application restart, live reloading, and enhanced error reporting.
- **Spring Boot Actuators**: Actuator endpoints enable monitoring, health checks, and management of microservices. They offer valuable insights into the application's metrics and runtime information.
- **Spring Data JPA**: Spring Data JPA simplifies database access and management by providing a high-level abstraction over the underlying database. It offers convenient APIs for interacting with the database using Java Persistence API (JPA) standards.
- **Spring Web**: The Spring Web module provides the necessary components for building web applications and RESTful APIs. It includes features like request handling, routing, and response generation.
- **Spring Cloud Gateway**: Spring Cloud Gateway serves as an API gateway, allowing clients to access the microservices. It provides routing, filtering, load balancing, and other features for managing requests.
- **Spring Cloud Config Server**: The Config Server allows centralized configuration management for the microservices. It retrieves configuration settings from a Git repository and distributes them to the respective services.
- **Spring Cloud Netflix (Eureka)**: Eureka, from the Spring Cloud Netflix suite, serves as a service registry and discovery server. It enables service registration, discovery, and load balancing within the microservices architecture.
- **Spring Cloud Feign**: Spring Cloud Feign is a declarative HTTP client that simplifies service-to-service communication. It integrates well with Spring Cloud and enables easy and intuitive interaction between microservices.
- **MySQL Connector J**: MySQL Connector J is the official JDBC driver for connecting Java applications with MySQL databases. It facilitates seamless communication between the microservices and the MySQL database for data storage and retrieval.
- **Docker**: Docker is a containerization platform that allows to package applications and their dependencies into containers. It provides a consistent and isolated environment for running the microservices, enhancing portability and scalability.

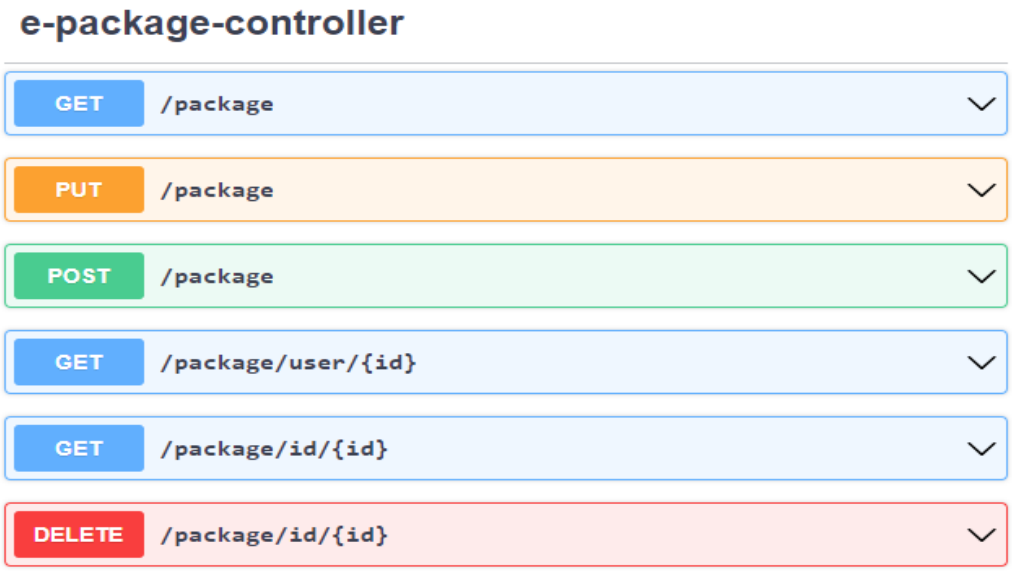
API Documentation

To facilitate API documentation, the system is well described by integrating OpenAPI and Swagger. These technologies enable comprehensive documentation of the exposed APIs, making it easier for clients to understand and interact with the services.

- **Rest**

REST services play a pivotal role in our system, enabling seamless interaction between client applications and the backend using HTTP methods and uniform resource identifiers.

[All the APIs are automatically documented. The API's auto-generated documentation can be found in the folder api/rest. To see the description, you need to import the API's JSON file into the Swagger editor.]



- **Soap**

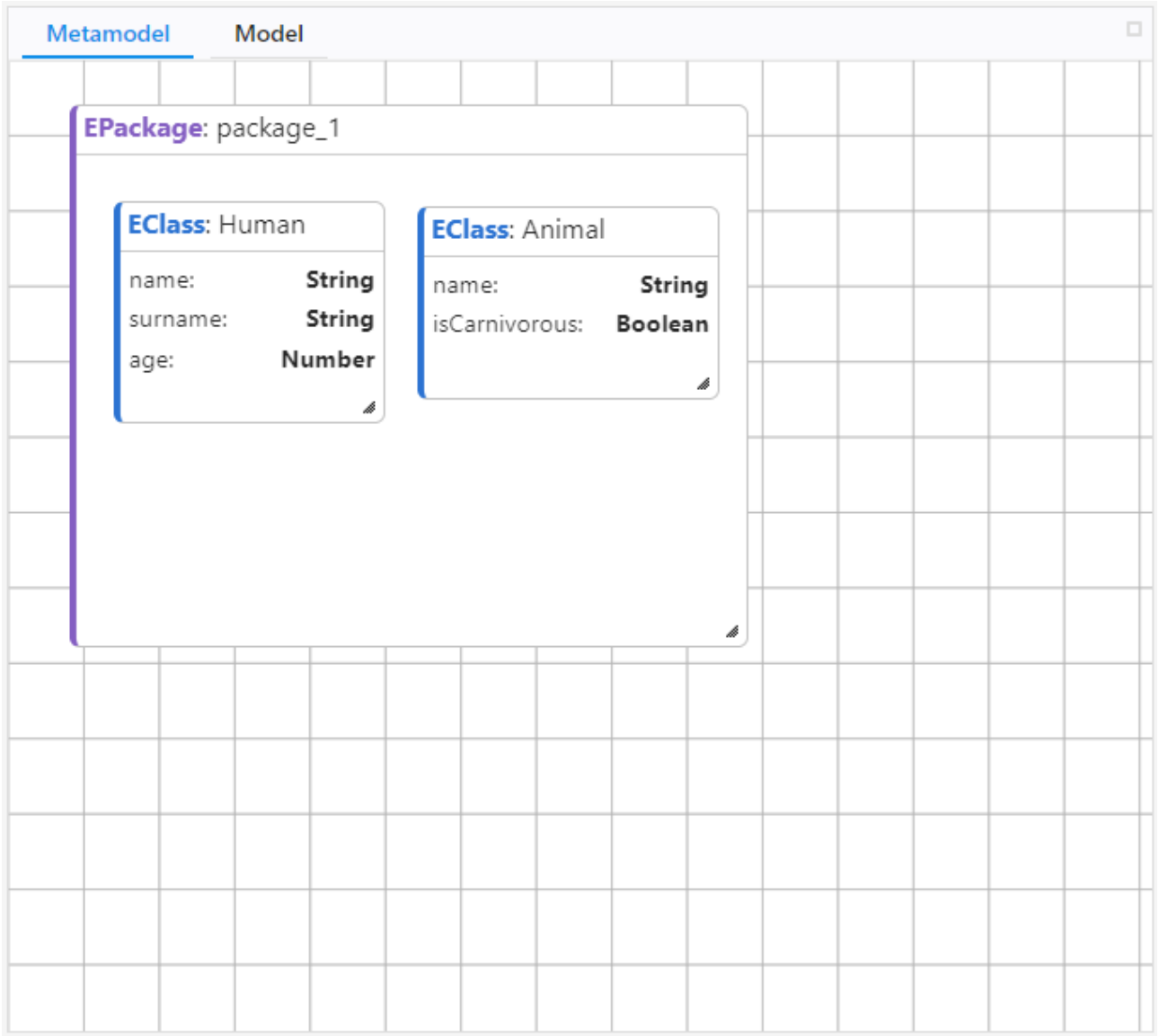
In our system, we have implemented a service designed to provide comprehensive information about packages. This service follows the SOAP protocol. The purpose of this service is to retrieve package data based on a given package ID. By accepting the package ID as input, the service performs a lookup in our system and retrieves the relevant package information. Alongside essential package data, the service also includes a textual description that offers a detailed and contextual overview of the package. Utilizing SOAP ensures that our service can seamlessly integrate with clients across various platforms and programming languages.

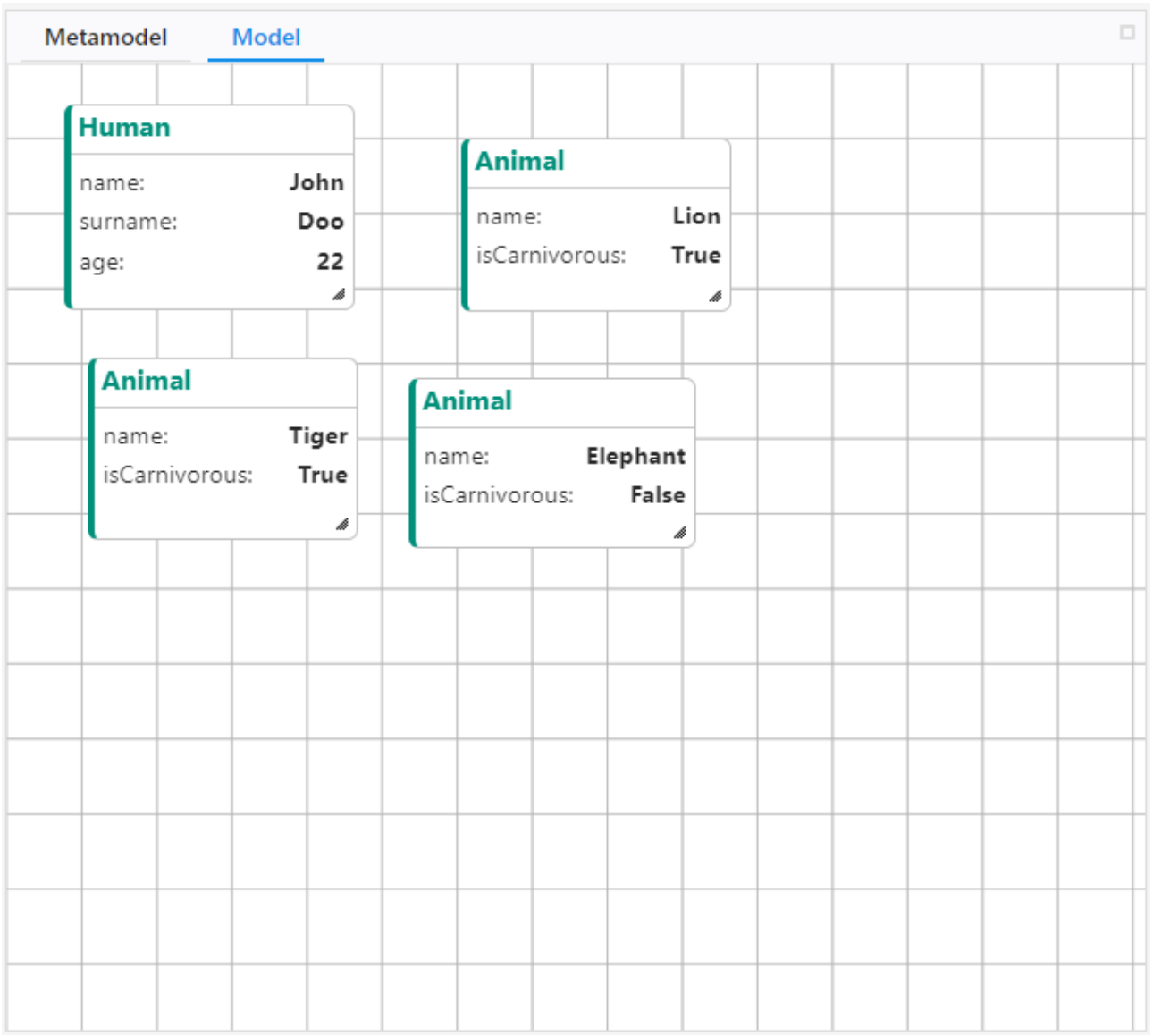
[The WSDL file can be found in the folder api/soap.]

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <SOAP-ENV:Body>
      <ns2:getPackageResponse xmlns:ns2="http://disim.it/univaq/sose/autogenerated/epackage">
         <ns2:name>package_800</ns2:name>
         <ns2:createBy>admin</ns2:createBy>
         <ns2:uri>it.disim.univaq.sose</ns2:uri>
         <ns2:description>Inside the specified package there are 3 classes.</ns2:description>
      </ns2:getPackageResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Client

To create a user friendly and responsive interface for interacting with our system, we built a frontend client using React, a popular JavaScript library for building user interfaces. React's component-based approach allowed us to break down the UI into reusable components, simplifying development and maintenance. To handle communication between the frontend and backend, we utilized Axios, an HTTP client library. Axios made it easy to send requests from the frontend to the backend endpoints, enabling operations such as retrieving data, creating new resources, updating existing ones, and deleting resources.




Metamodel
Model
Human
name: John
surname: Doo
age: 22
Animal
name: Lion
isCarnivorous: True
Animal
name: Tiger
isCarnivorous: True
Animal
name: Elephant
isCarnivorous: False