A-TO-MIC

APPLICATION
TO
MICROSERVICES

visit
www.a-to-mic.net
for download

Content

A-TO-MIC
Version 1.5.6

0. Motivation

Why should I use microservices?

Copilot

Certainly! Microservices are an architectural approach that breaks down applications into smaller, independent components. Here are some reasons why you might consider using microservices:

1. **Modularity and Scalability**:
    - Microservices allow you to split your application into smaller services, each handling a specific business function. This modularity makes it easier to scale individual services independently based on demand.
    - Unlike monolithic applications where scaling means replicating the entire system, microservices let you focus on the parts that need more resources.
2. **Developer Productivity**:
    - Developers work on smaller codebases for each microservice, which improves productivity and speeds up development.
    - Loosely coupled components mean that understanding the source code is easier, leading to faster development cycles.
3. **Clear Responsibilities**:
    - Each microservice has a well-defined responsibility, making it easier to assign teams to specific components.
    - Code reviews become more focused, and updates can be made without affecting the entire application.
4. **Technology Stack Flexibility**:
    - Microservices allow you to use different programming languages and technologies for different services.
    - You're not tied to a single language or library, enabling polyglot microservices.
5. **Continuous Delivery**:
    - Updates to a specific microservice don't require redeploying the entire application.
    - Changes can be isolated and deployed independently, improving agility.
6. **Reduced Risk during Updates**:
    - Microservices operate independently, reducing the risk of system-wide disruptions during updates.
    - You can modify or update one service without affecting the entire system4 5.

Remember that while microservices offer benefits, they also introduce complexities related to communication, deployment, and monitoring. Consider your specific use case and evaluate whether microservices align with your application's requirements. If you'd like more in-depth information, I recommend checking out the provided resources:

What are the challenges of implementing microservices?

Copilot

Certainly! Implementing microservices comes with its own set of challenges. Let's explore some of the common ones:

1. **Design Complexity**:
   - Compared to monolithic applications, designing microservices introduces increased complexity. Deciding on each microservice's size, optimal boundaries, and connection points can be challenging.
   - Developers must create microservices within a bounded context, clarifying specific responsibilities for each service. This often involves using a data-centric view when modeling the domain1.
2. **Security**:
   - Microservices are often deployed across multi-cloud environments, leading to increased risk and loss of control and visibility over application components.
   - Each microservice communicates with others through various infrastructure layers, making it harder to test for vulnerabilities.
   - Data security within a microservices-based framework remains a concern, especially since data is distributed across services1.
3. **Service Coordination**:
   - Microservices need to communicate with each other efficiently. Ensuring seamless service coordination and discovery can be challenging.
   - Coordinating interactions between services becomes crucial for maintaining overall system functionality4.
4. **Testing**:
   - Ensuring that all microservices are tested individually and as a system can be complex.
   - Comprehensive testing is essential to identify issues early and maintain system reliability4.
5. **Deployment and Scaling**:
   - Deploying and scaling microservices individually requires a robust infrastructure.
   - Managing the deployment and scaling of multiple services can be intricate, especially when dealing with varying resource demands4.
6. **Monitoring and Observability**:
   - Monitoring microservices in a distributed environment can be challenging.
   - Establishing effective observability practices to track performance, detect anomalies, and troubleshoot issues is crucial2.
7. **Service Resilience and Fault Tolerance**:
   - Microservices should be resilient to failures and faults. Implementing strategies like circuit breakers, retries, and fallback mechanisms is essential.
   - Ensuring that the entire system remains operational even if individual services fail requires careful planning2.

How do I handle data consistency across microservices?

Copilot

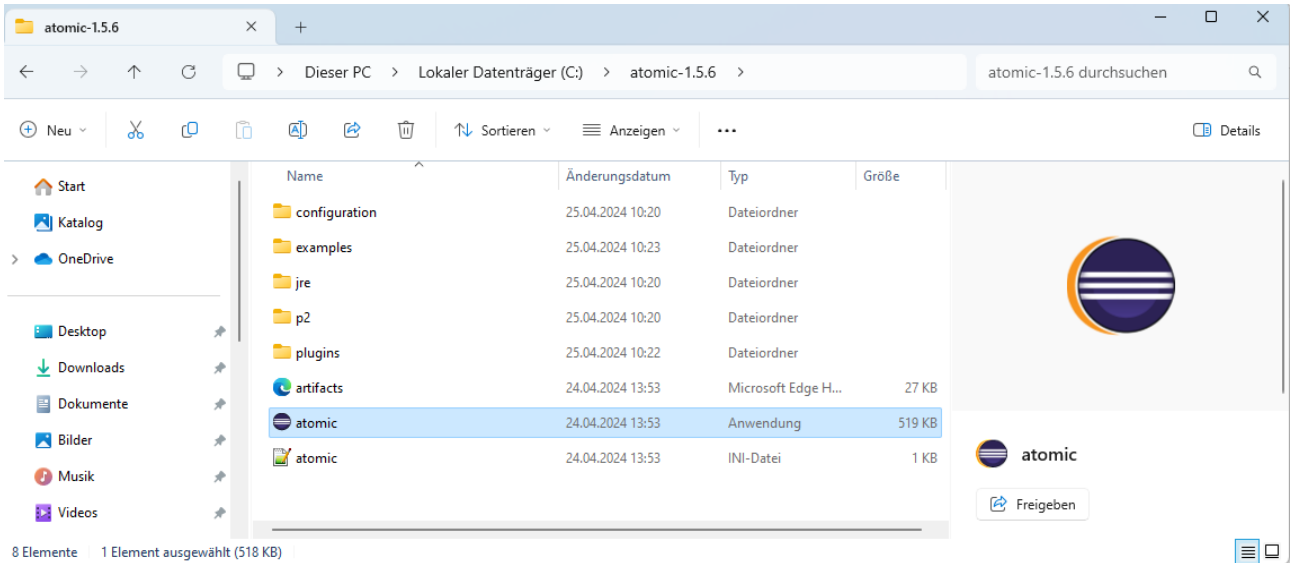
Ensuring data consistency across microservices in a distributed landscape can be challenging. Let's explore some approaches to address this:

1. **Service Interfaces (APIs)**:
    - Instead of direct database dependencies, express dependencies between microservices using service interfaces (APIs).
    - Each service team can change its internal data schema independently, focusing only on the interface design when dealing with dependencies.
    - This approach allows you to deploy new microservice versions independently as long as old APIs are still supported1.
2. **Saga Pattern**:
    - The **Saga pattern** is specifically designed to handle data consistency in distributed transactions within microservices architectures.
    - It breaks down long-running transactions into smaller, manageable steps (saga steps).
    - Each step corresponds to an action performed by a microservice.
    - If any step fails, compensating actions can be executed to revert changes made by previous steps.
    - Sagas ensure that data consistency is maintained even when multiple services are involved2.
3. **Event-Driven Architecture**:
    - Microservices can publish events when entities are created, updated, or deleted.
    - Other interested microservices can subscribe to these events and update their linked entities accordingly.
    - While this approach works, it requires careful coordination and programming effort across services1.

Remember that there's no one-size-fits-all solution, and the choice depends on your specific use case. Evaluate the trade-offs and select an approach that aligns with your system's requirements and constraints. 🚀

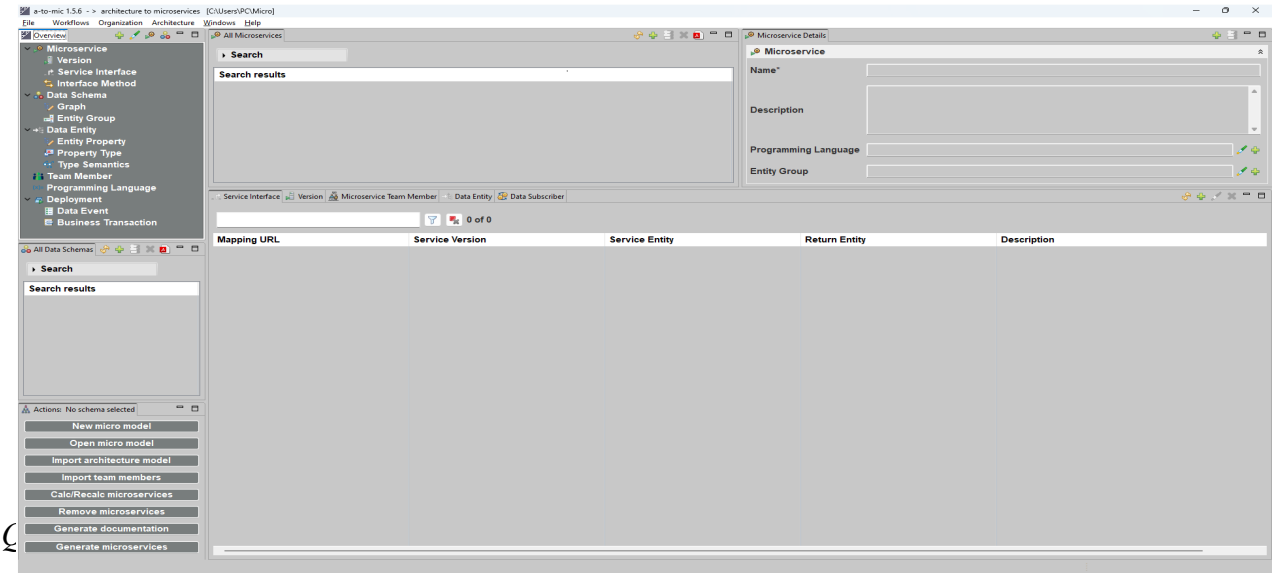
1. Download, Installation, Start

Go to www.a-to-mic.org/download the file atomic-1.5.6.zip to get the latest version of the desktop client app.
Extract the file to a file location on your hard disk. Choose a folder where you want the software and the corresponding example architecture files to be installed to. After Extracting the file, the folder should look like this:



Abbr. 1.2

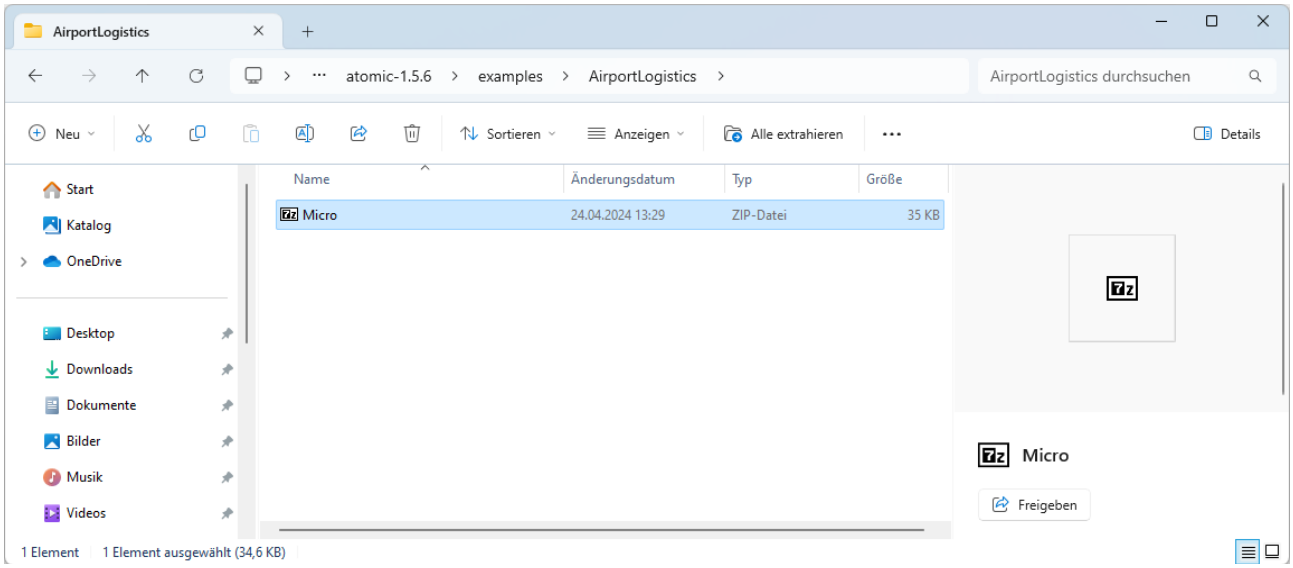
The file atomic.exe ist the application. A double click will start it on a Windows-64 system. When it is started for the first time, the application will come up like this:



Q

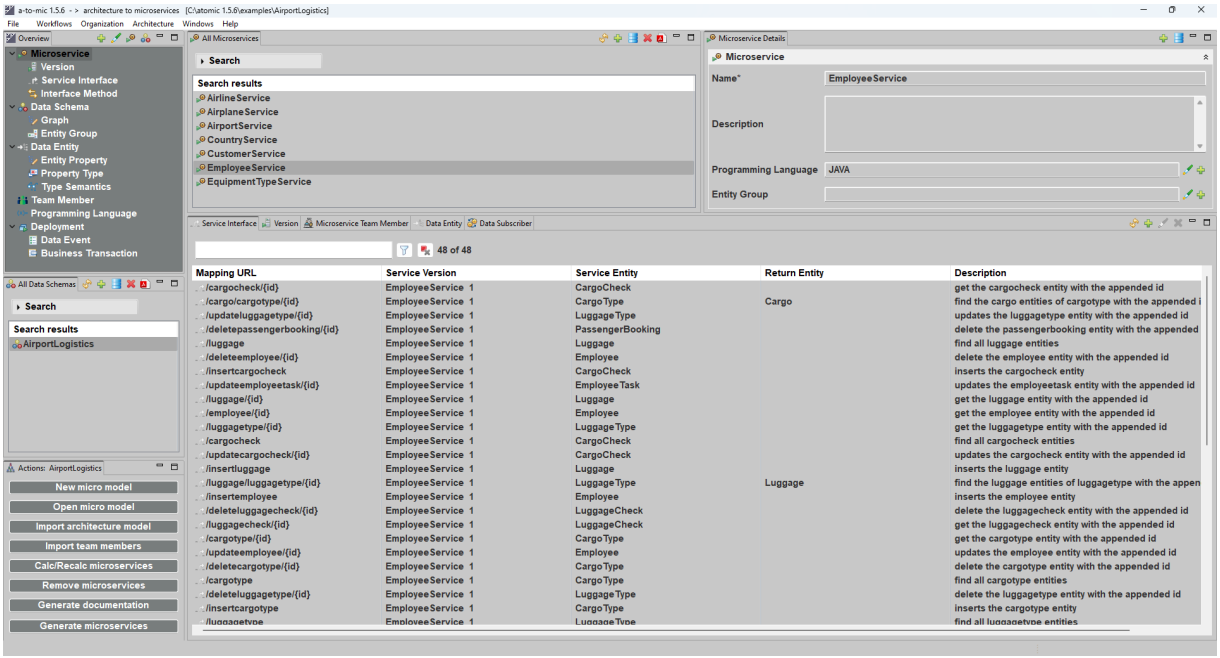
Abbr. 1.5

Open the „example“ folder, go into the folder „AirportLogistics“ and drag the file **micro.zip** into the tree view with the title **Overview**.



Abbr. 1.4

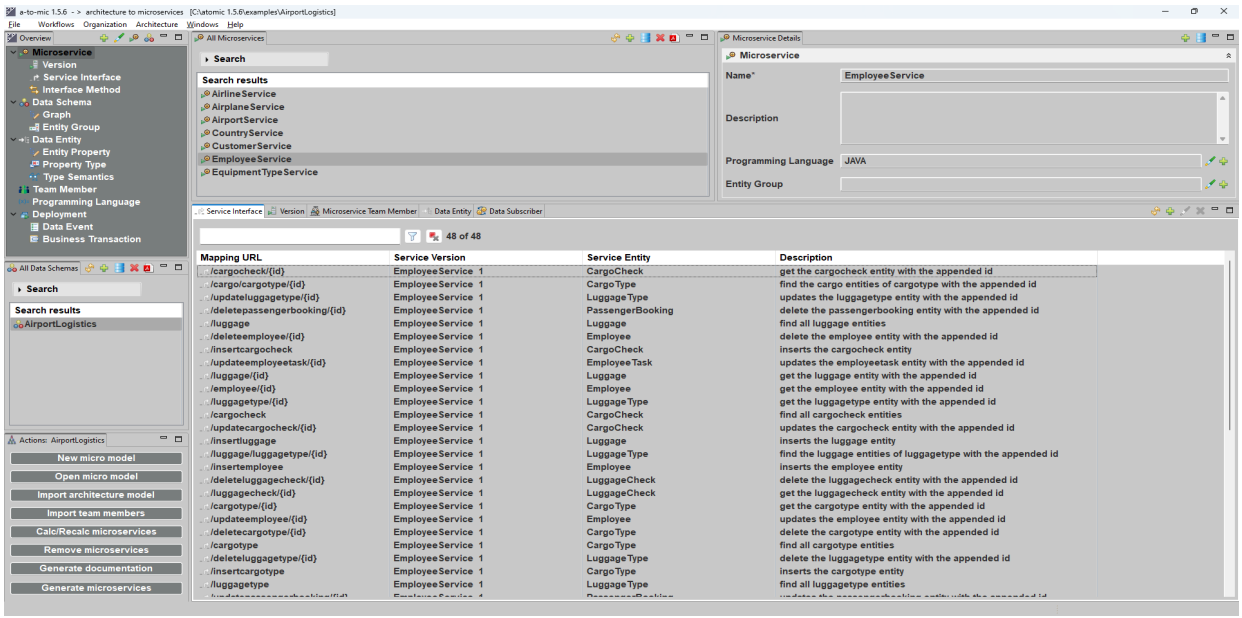
After dropping the file, the application will open the microservice architecture file and switch into the microservice perspective. Now, the perspective shows the content of the „AirportLogistics“ microservice model, that we are going to discuss in detail in the following chapters.


a-to-mic 1.5.6 - > architecture to microservices [C:\atomic 1.5.6\examples\AirportLogistics]
File Workflows Organization Architecture Windows Help
Overview
Microservice
Version
Service Interface
Interface Method
Data Schema
Graph
Entity Group
Data Entity
Entity Property
Property Type
Type Semantics
Team Member
Programming Language
Deployment
Data Event
Business Transaction
All Microservices
Search
Search results
AirlineService
AirplaneService
AirportService
CountryService
CustomerService
EmployeeService
EquipmentTypeService
Microservice Details
Microservice
Name*
EmployeeService
Description
Programming Language
JAVA
Entity Group
Service Interface
Version
Microservice Team Member
Data Entity
Data Subscriber
48 of 48
Mapping URL
Service Version
Service Entity
Return Entity
Description
/cargocheck/{id} EmployeeService 1 CargoCheck get the cargocheck entity with the appended id
/cargo/cargotype/{id} EmployeeService 1 CargoType Cargo find the cargo entities of cargotype with the appended i
/updateluggagetype/{id} EmployeeService 1 LuggageType updates the luggagetype entity with the appended id
/deletepassengerbooking/{id} EmployeeService 1 PassengerBooking delete the passengerbooking entity with the appended
/luggage EmployeeService 1 Luggage find all luggage entities
/deleteemployee/{id} EmployeeService 1 Employee delete the employee entity with the appended id
/insertcargocheck EmployeeService 1 CargoCheck inserts the cargocheck entity
/updateemployeetask/{id} EmployeeService 1 EmployeeTask updates the employeetask entity with the appended id
/luggage/{id} EmployeeService 1 Luggage get the luggage entity with the appended id
/employee/{id} EmployeeService 1 Employee get the employee entity with the appended id
/luggagetype/{id} EmployeeService 1 LuggageType get the luggagetype entity with the appended id
/cargocheck EmployeeService 1 CargoCheck find all cargocheck entities
/updatecargocheck/{id} EmployeeService 1 CargoCheck updates the cargocheck entity with the appended id
/insertluggage EmployeeService 1 Luggage inserts the luggage entity
/luggage/luggagetype/{id} EmployeeService 1 LuggageType Luggage find the luggage entities of luggagetype with the appen
/insertemployee EmployeeService 1 Employee inserts the employee entity
/deleteluggagecheck/{id} EmployeeService 1 LuggageCheck delete the luggagecheck entity with the appended id
/luggagecheck/{id} EmployeeService 1 LuggageCheck get the luggagecheck entity with the appended id
/cargotype/{id} EmployeeService 1 CargoType get the cargotype entity with the appended id
/updateemployee/{id} EmployeeService 1 Employee updates the employee entity with the appended id
/deletecargotype/{id} EmployeeService 1 CargoType delete the cargotype entity with the appended id
/cargotype EmployeeService 1 CargoType find all cargotype entities
/deleteluggagetype/{id} EmployeeService 1 LuggageType delete the luggagetype entity with the appended id
/insertcargotype EmployeeService 1 CargoType inserts the cargotype entity
/luggagetype EmployeeService 1 LuggageType find all luggagetype entities
All Data Schemas
Search
Search results
AirportLogistics
Actions: AirportLogistics
New micro model
Open micro model
Import architecture model
Import team members
Calc/Recalc microservices
Remove microservices
Generate documentation
Generate microservices

Abbr. 1.6

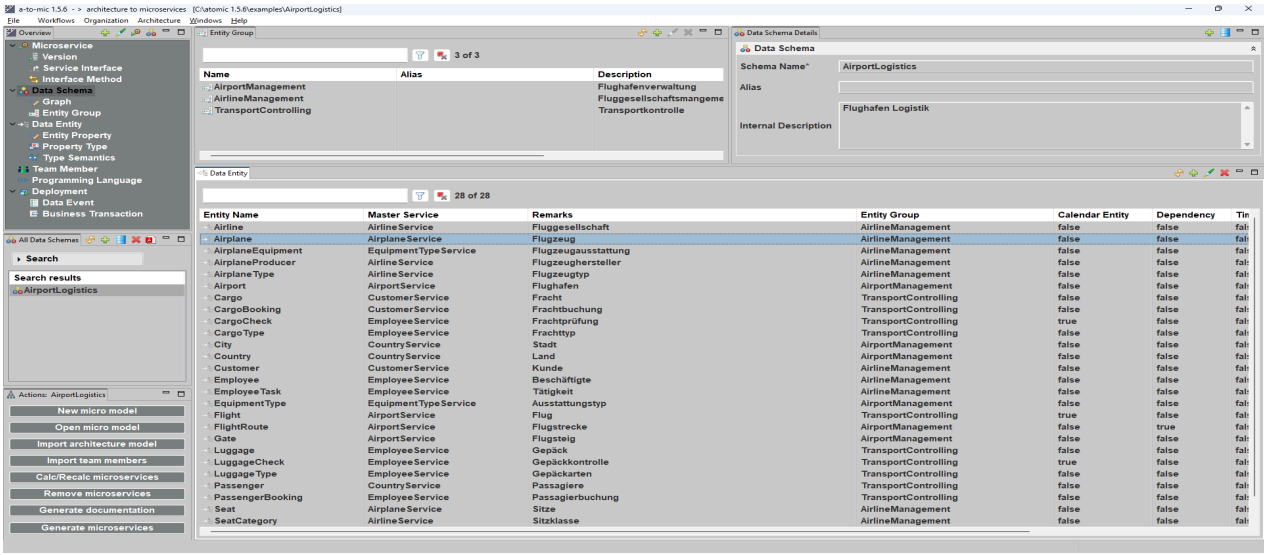
2. Overview

The left side shows the navigation. In the **overview** tree you can choose the perspective, where you want to work in. Below the tree is a list, that shows all the **data schemas** in the micro.zip file, that you have loaded. In the south are the **actions**. Some actions correspond with the selected data schema, because the calculation of the microservice structure takes the selected data schema as its parameter input.



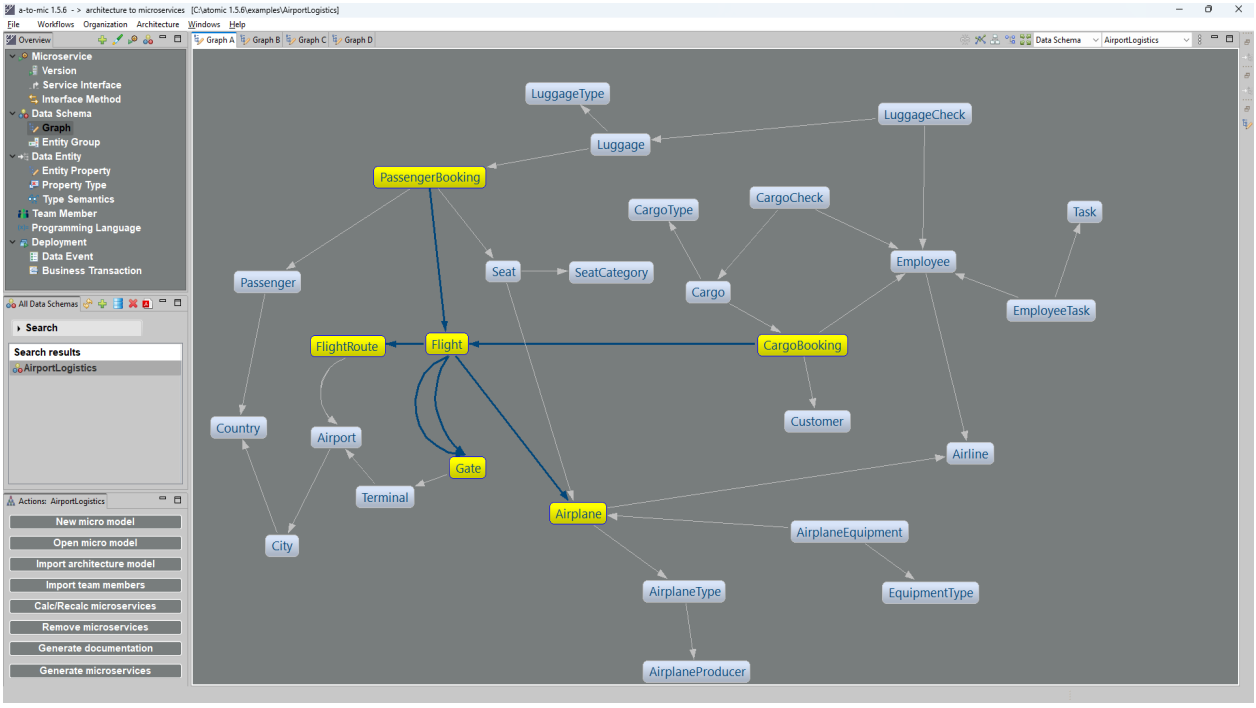
Abbr. 2.1

The data schema perspective shows all the entities of the selected data schema. To modularize the entities, they can be grouped to so called entity groups. The microservice calculator will use the information as a hint to put two entities of the same business aspect to the same microservice, if the complexity rule of the microservice calculator allows this.



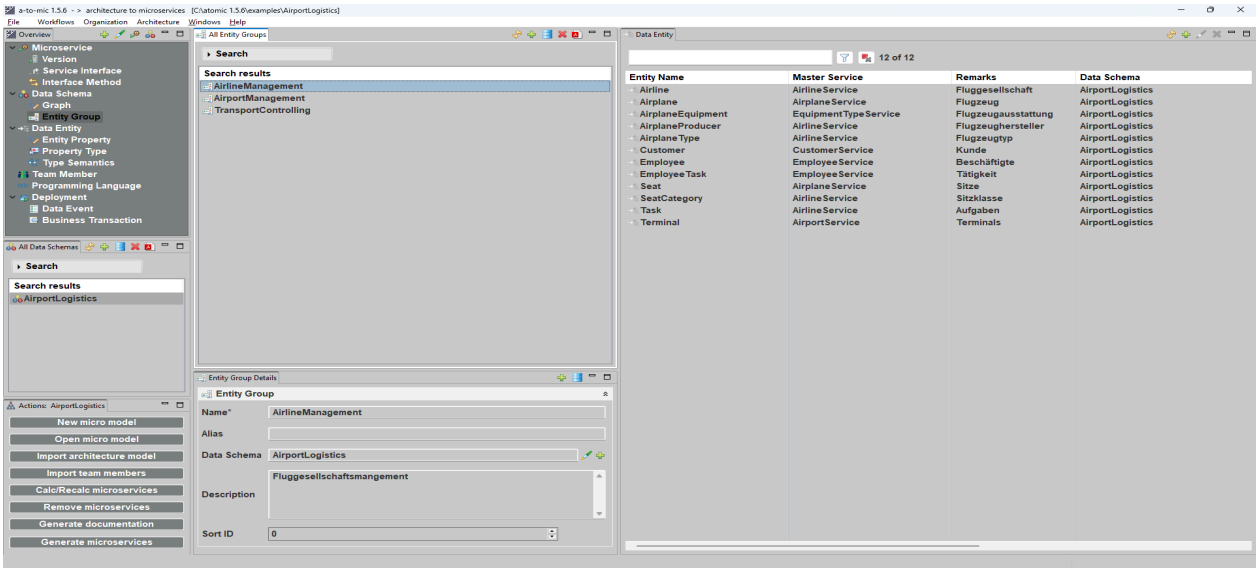
Abbr. 2.2

The existing data schema of the business domain „AirportLogistics“ can be shown in a graph model. Double clicking the entity node „Flight“ will also selected its neighbour entities, which are related to the „Flight“ entity. From a business point of view, these entities should be put into the same microservice. But it could also be the case, that the selected entity „Gate“ is more attracted by the entities that model a terminal or an airport. The microservice calculator will help you on your design decision.



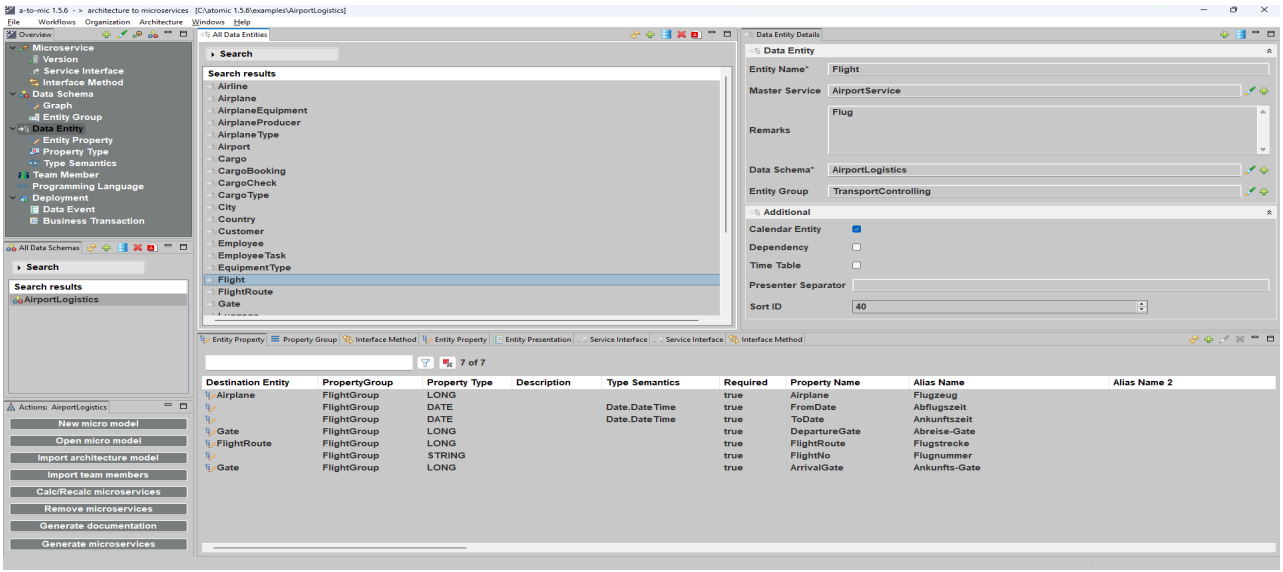
Abbr. 2.3

The entity groups should be defined by the experts of the „AirportLogistics“ domain. Where does an entity belong to can be a question, that is hard to be answered. Discuss this with a pool of experts.



Abbr. 2.4

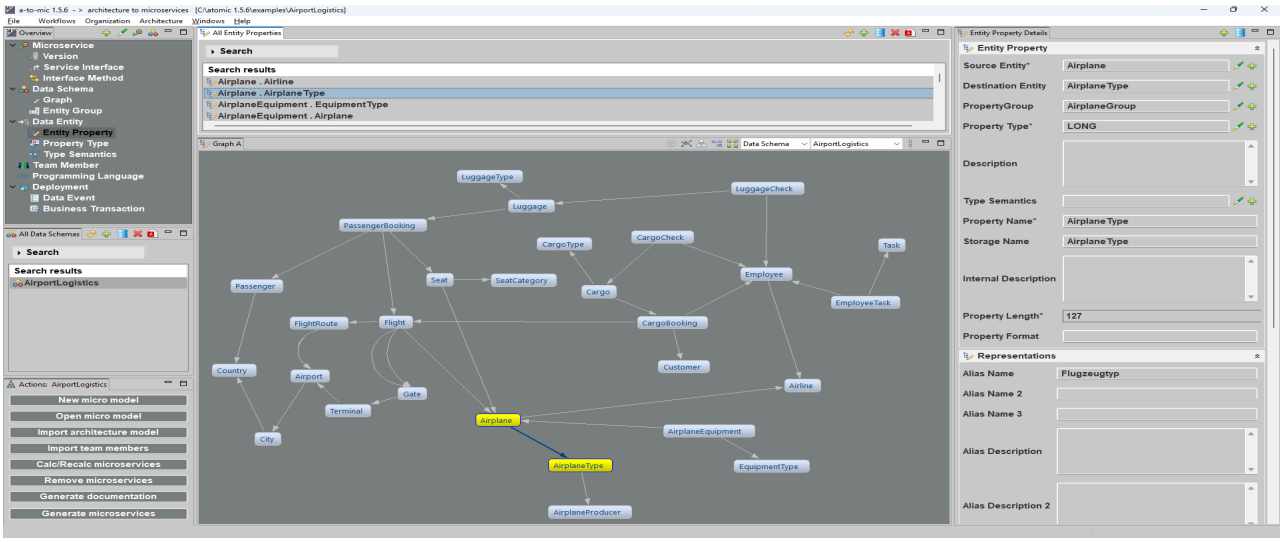
The data entity perspective shows you the details of each domain entity with its details and properties.



Abbr. 2.5

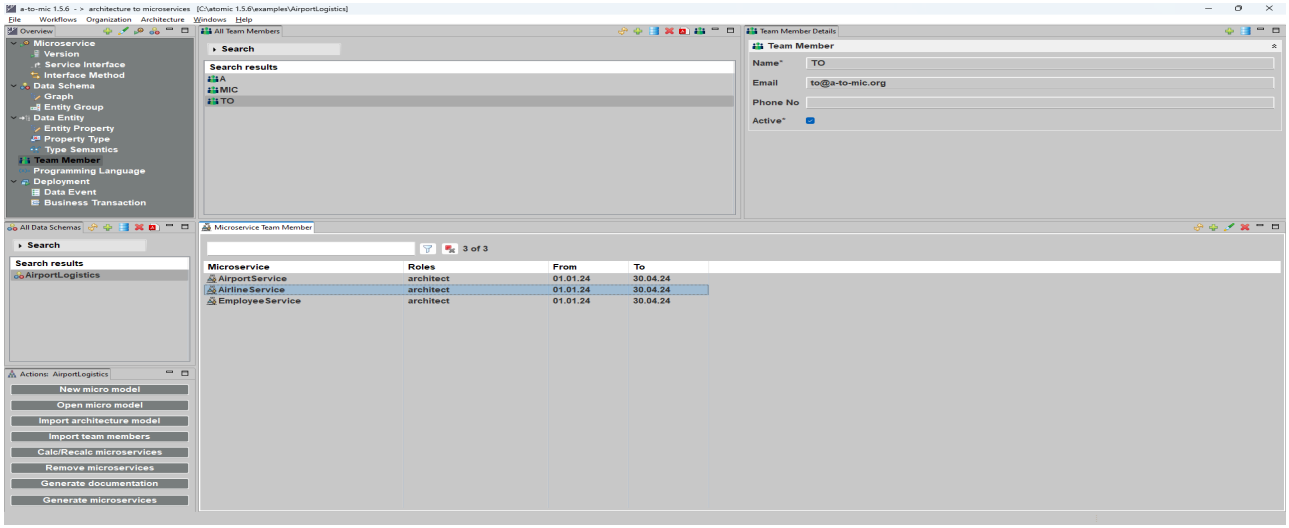
The atoms of every domain are the entity properties. Properties contain the

information, but they also build the bridges between different entities. The entity property perspective gives you insight to the bridges of the domain model.



Abbr. 2.6

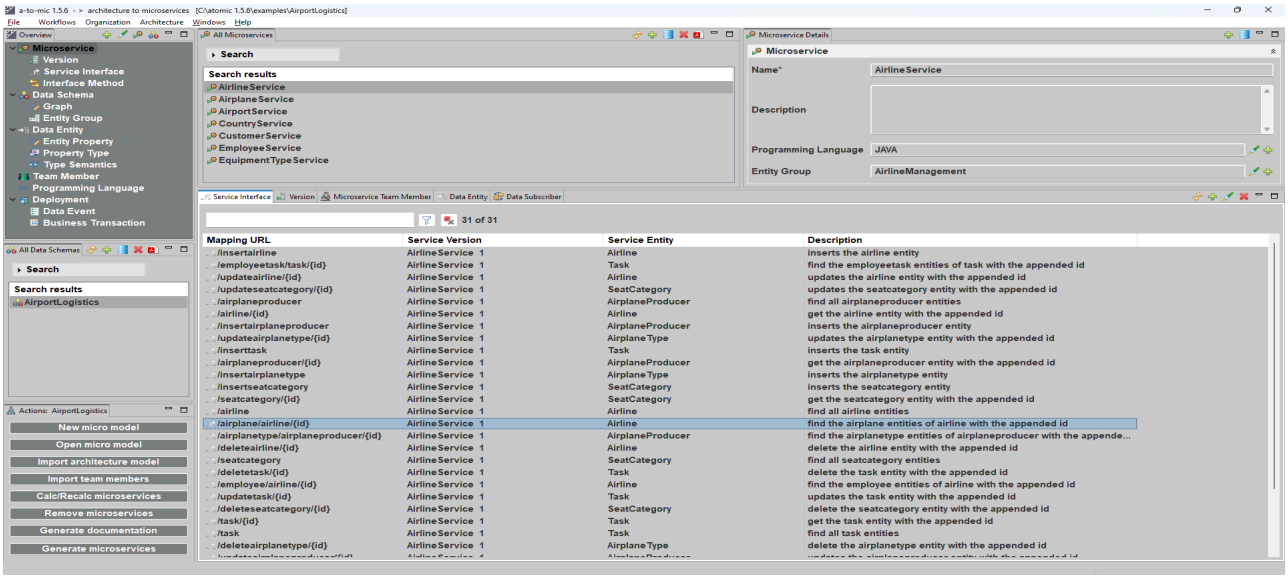
Every microservice is a deployment unit, that has its own lifecycle. The responsibility of each service can be assigned to members of the team.



Abbr. 2.7

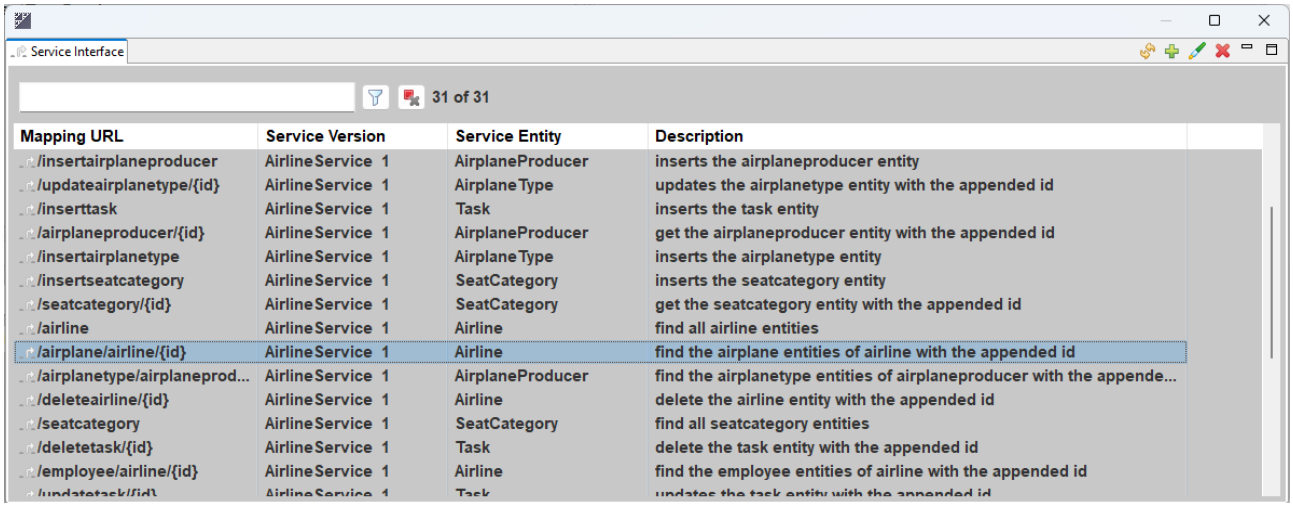
3. Microservices

The microservice perspective is the entry point into the application. This is the place, where the microservices, that have been calculated from the data schema, can be reviewed by the application architect. There are different aspects, that form a microservice.



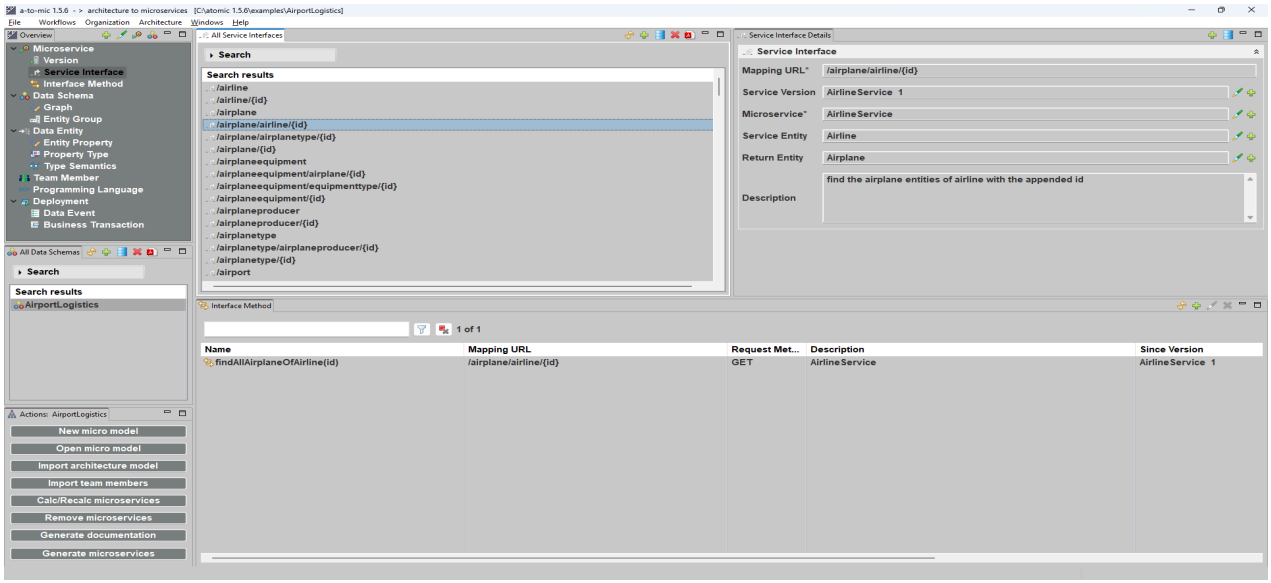
Abbr. 3.1

The boundaries to the „outside world“ of a microservice are the service interfaces. They build the skin of the service module body. Every communication with the microservice takes place by invoking one of its service interface methods.



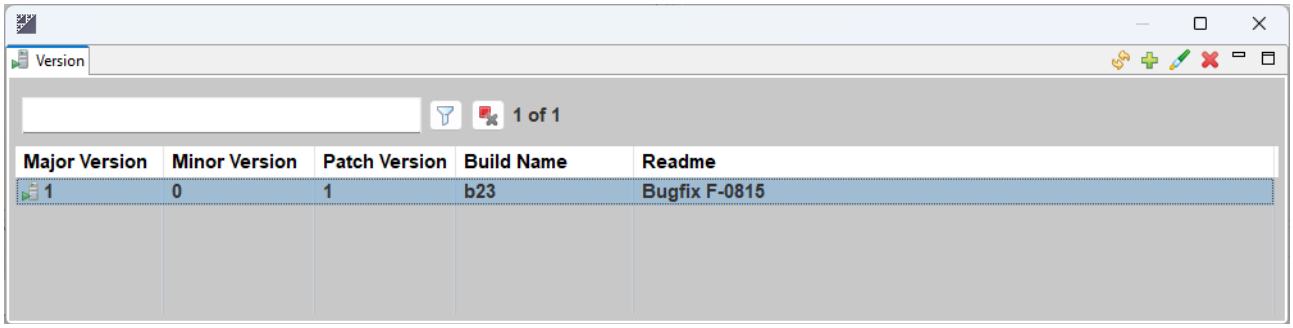
Abbr. 3.2

The mapping URL is a relative path to the REST-endpoint of a interface method. Every read and write access to the state of the microservice is handled over this endpoint. Every interface has one of the four REST request methods GET, PUT, POST and DELETE.



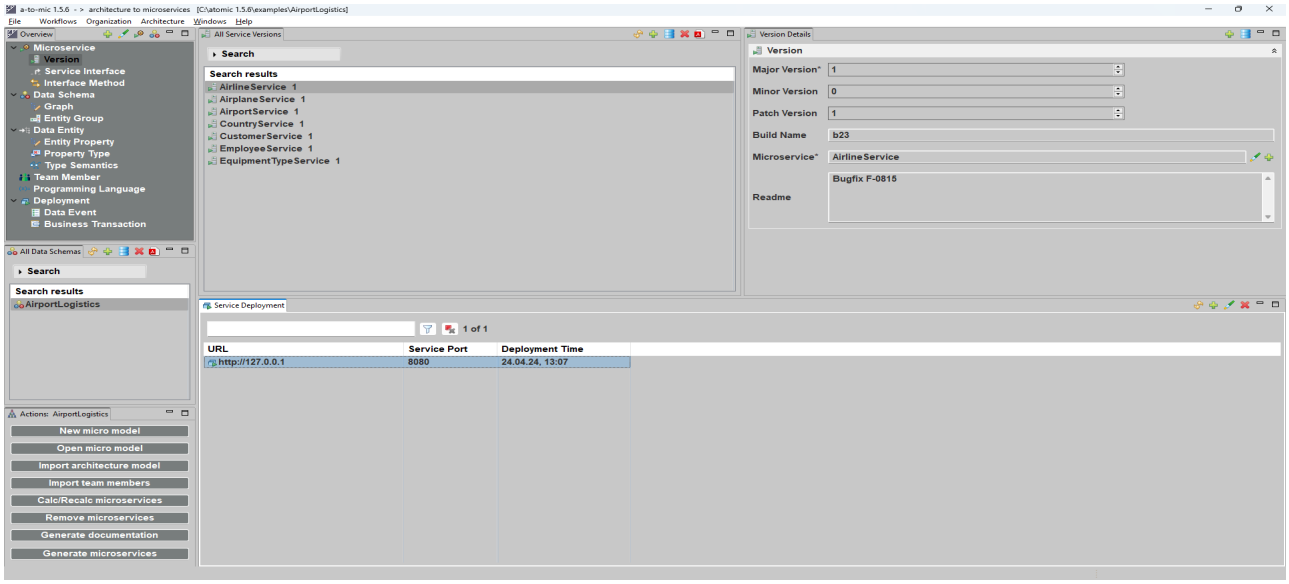
Abbr. 3.3

The different versions of the microservice, that evolve over time, can be traced in the versions view.



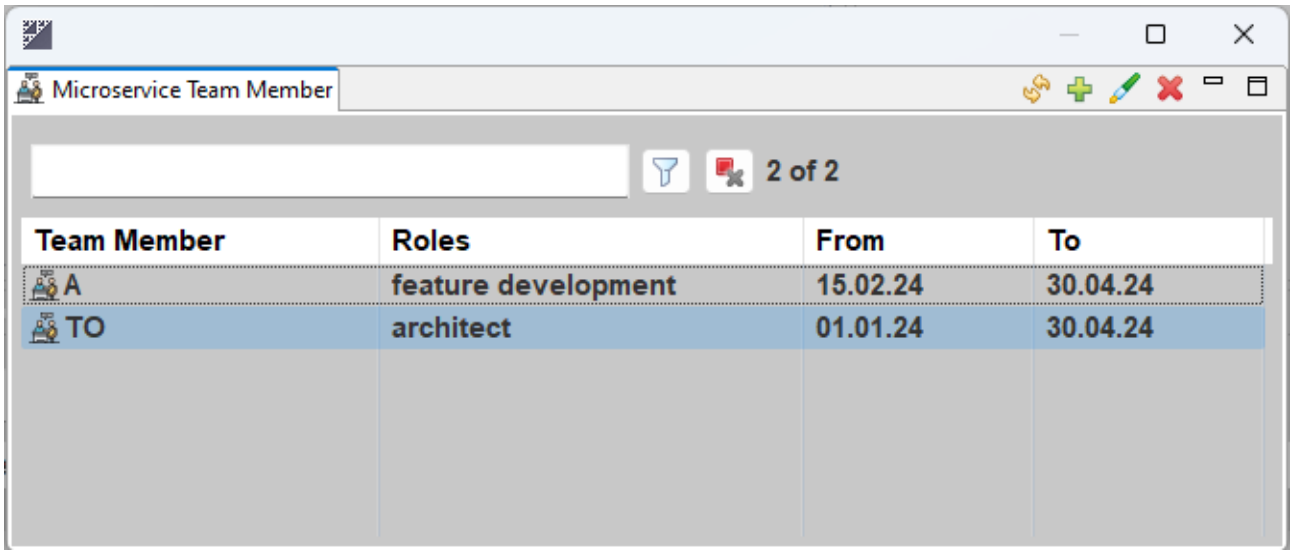
Abbr. 3.4

In the long term, new versions of the microservice will be created and deployed to the cloud. The evolution of the service can be tracked in the deployment history.



Abbr. 3.5

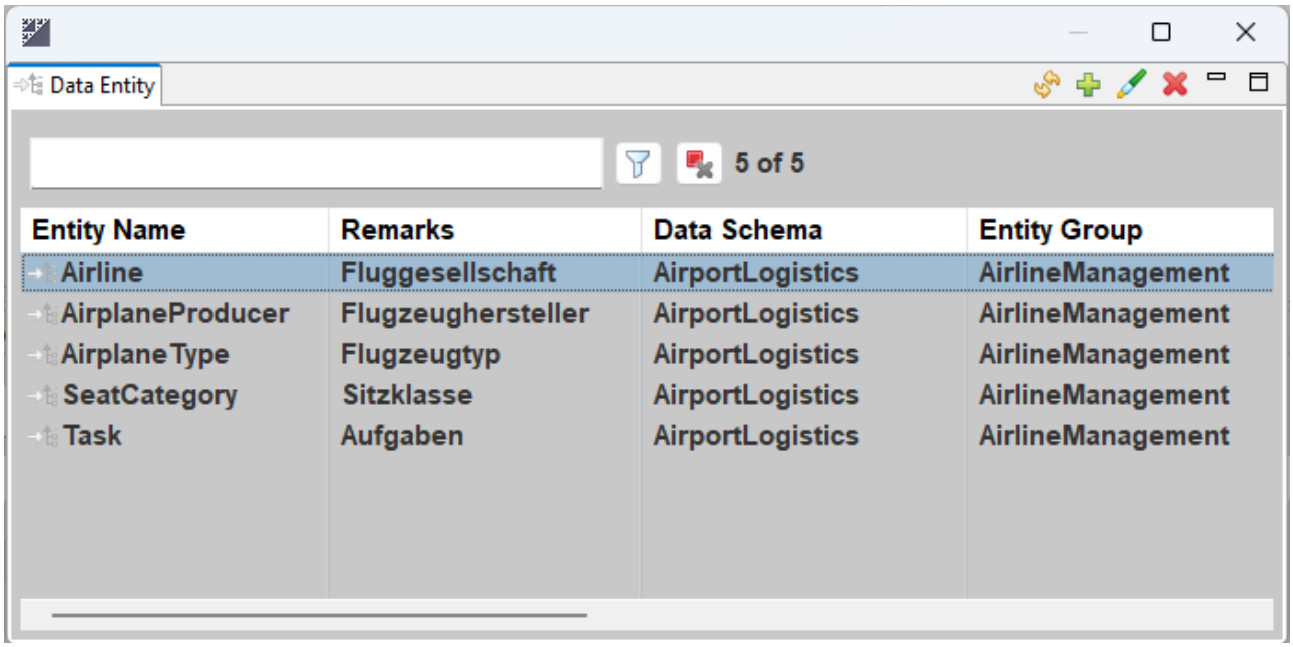
In the team members view there can be stored the people who are responsible for the service.



Abbr. 3.6

In the „Master entities“ view are the core entities, that are managed by this microservice exclusively. They have been assigned to this microservice after the calculation finished. Because of impacting the inter-communication between different micro services, they are clustered in a way to minimize the messaging traffic between the services. Other microservices that subscribe to the core entity of a microservice, are

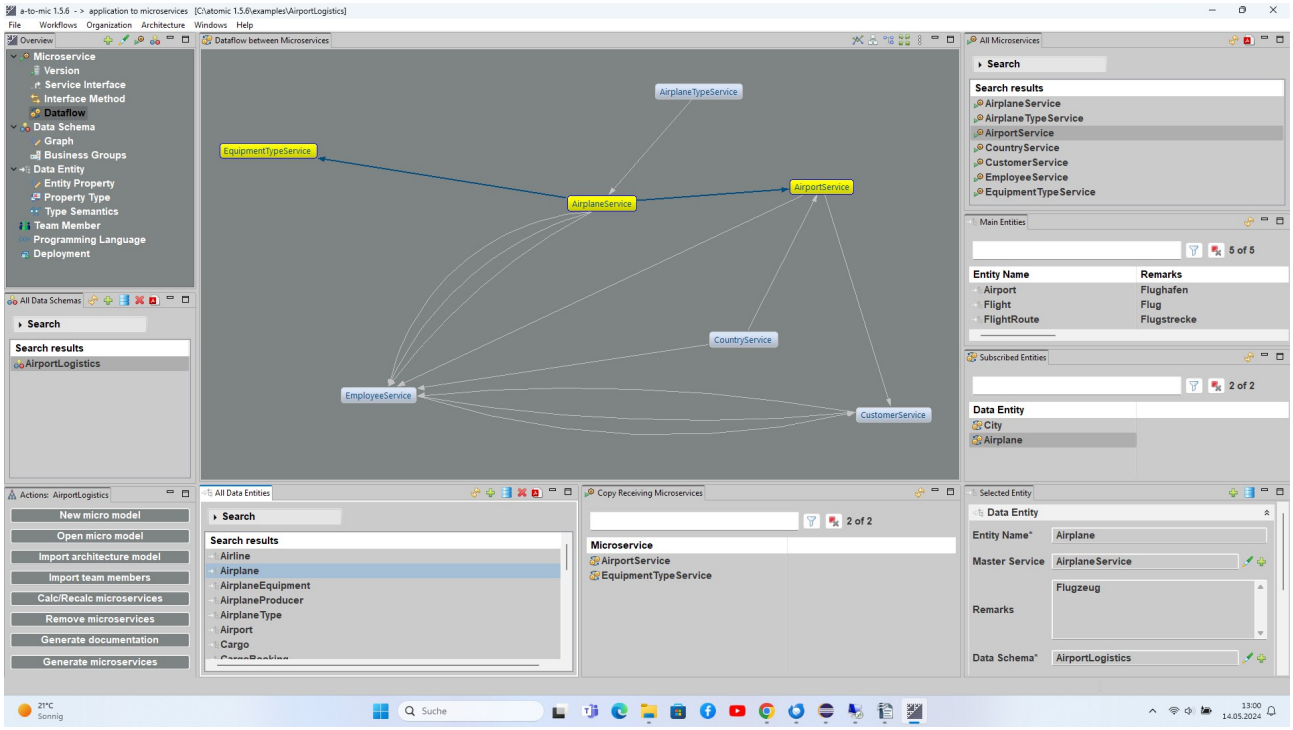
updated via messaging after the core entity in the master service has been modified.



Abbr. 3.7

4. Dataflow

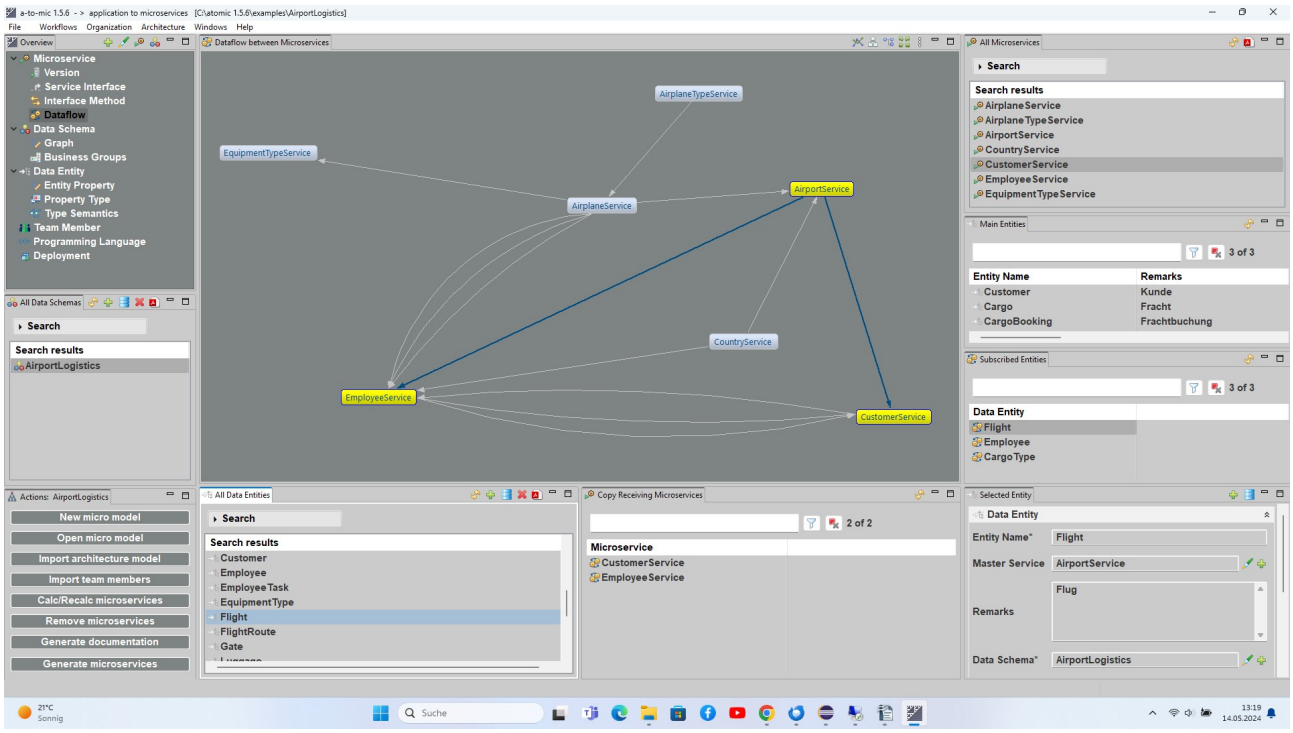
After A-TO-MIC has calculated the microservices, you can switch to the **Dataflow** perspective. A-TO-MIC has assigned each data entity of the schema to one microservice that writes modifications exclusively to its local data store. Services, that depend on this data entity, have to get a copy of that data entity. This copy of the data entity will be delivered via messaging. The logical dataflow perspective hides, that every data transfer between two microservice always happens over the message broker that dispatches incoming messages to the interested microservices.
The following example shows the dataflow perspective of the „AirportLogistics“ data schema.



Abbr. 4.1

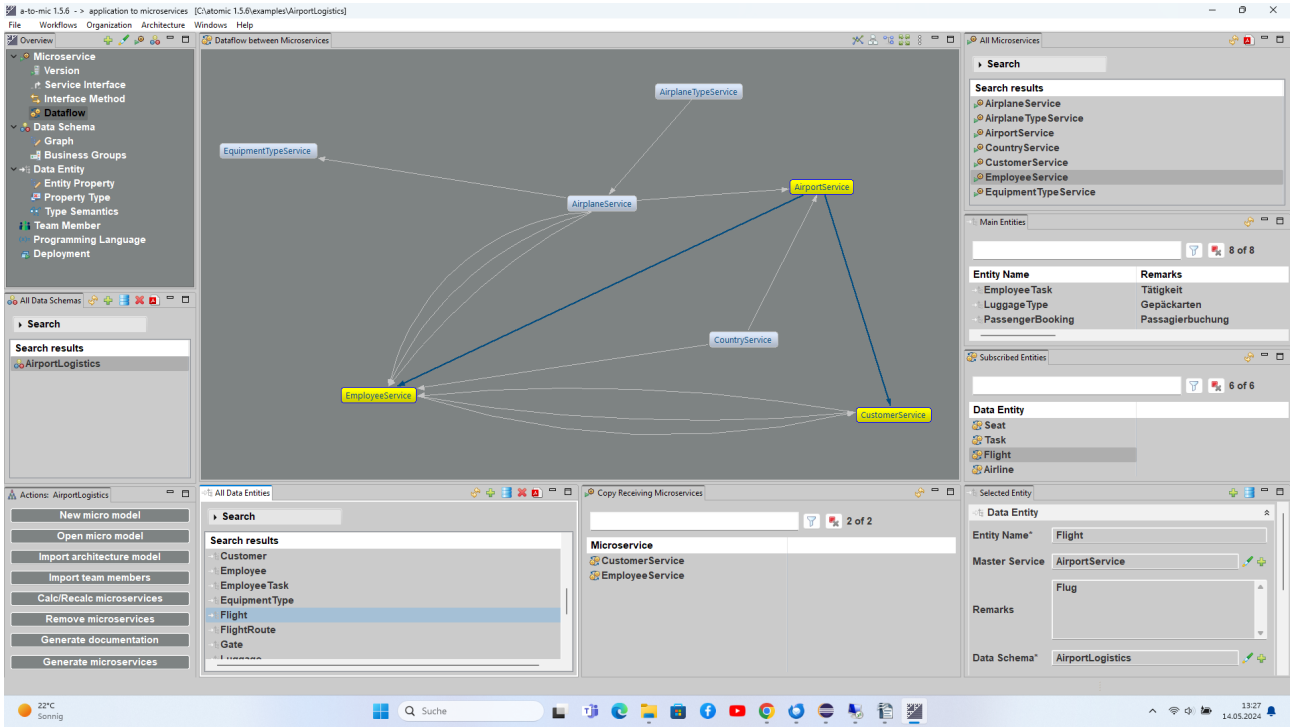
The dataflow of the entities results from the calculation of the microservices. By selecting the data entity in the **All Data Entities** view. the message flow of the entity copies is shown in the dataflow graph. The original version of the „Airplane“ entity is hosted by the „AirplaneService“. The two microservices „AirportService“ and „EquipmentTypeService“ depend on the „Airplane“ entity. The message

broker sends those services a copy of the „Airplane“ entity to keep them up-to-date. Selecting the „Flight“ entity, where the master service is the „AirportService“, the message broker sends copies of the „Flight“ entity to the „CustomerService“ and the „EmployeeService“ because one or more entities depend on the „Flight“ entity. On the right-hand side the „CustomerService“ is selected. In the list of **Subscribed Entities** we can see the „Flight“ entity. The „CustomerService“ gets a copy of the „Flight“ entity from the message broker, when the „AirportService“ modifies, creates or deletes a „Flight“ entity.



Abbr. 4.2

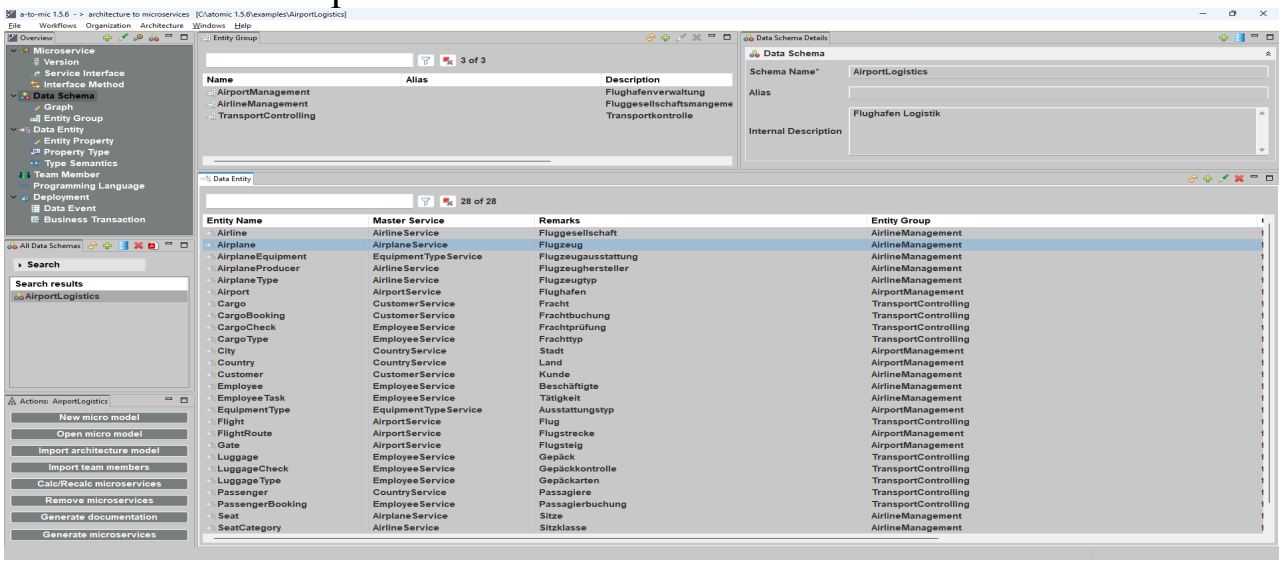
The same happens for the „EmployeeService“.


a-to-mic 1.5.6 - > application to microservices [C:\atomic 1.5.6\examples\AirportLogistics]
File Workflows Organization Architecture Windows Help
Overview
Microservice
Version
Service Interface
Interface Method
Dataflow
Data Schema
Graph
Business Groups
Data Entity
Entity Property
Property Type
Type Semantics
Team Member
Programming Language
Deployment
Dataflow between Microservices
AirplaneTypeService
EquipmentTypeService
AirplaneService
AirportService
CountryService
EmployeeService
CustomerService
All Microservices
Search
Search results
AirplaneService
AirplaneTypeService
AirportService
CountryService
CustomerService
EmployeeService
EquipmentTypeService
Main Entities
8 of 8
Entity Name
Remarks
EmployeeTask
Tätigkeit
LuggageType
Gepäckarten
PassengerBooking
Passagierbuchung
All Data Schemas
Search
Search results
AirportLogistics
Subscribed Entities
6 of 6
Data Entity
Seat
Task
Flight
Airline
Actions: AirportLogistics
New micro model
Open micro model
Import architecture model
Import team members
Calc/Recalc microservices
Remove microservices
Generate documentation
Generate microservices
All Data Entities
Search
Search results
Customer
Employee
EmployeeTask
EquipmentType
Flight
FlightRoute
Gate
Copy Receiving Microservices
2 of 2
Microservice
CustomerService
EmployeeService
Selected Entity
Data Entity
Entity Name*
Flight
Master Service
AirportService
Remarks
Flug
Data Schema*
AirportLogistics
Suche

Abbr. 4.3

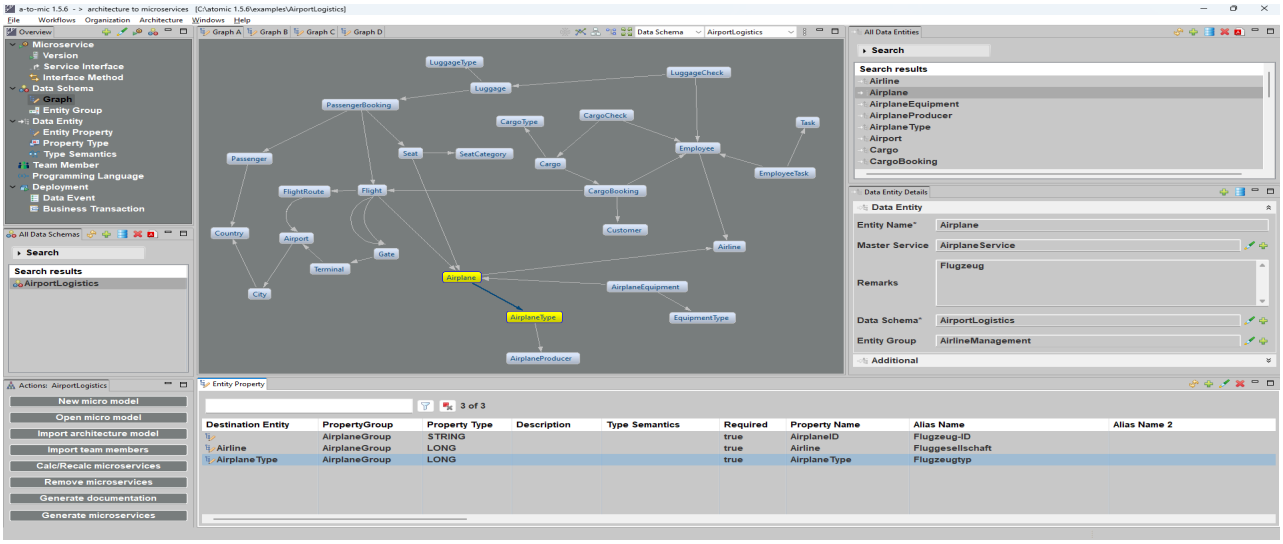
5. Data Schema

The data schema perspective paints the big picture of the whole „AirportLogistics“ model. All contained entities and their business groups are shown in a compact view.



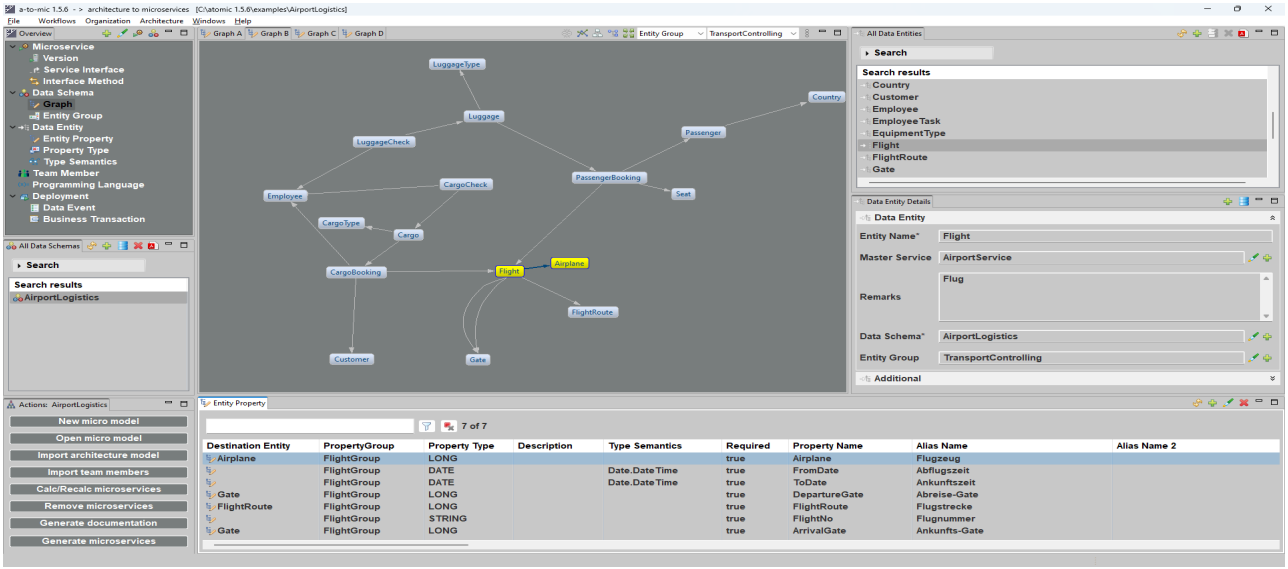
Abbr. 5.1

The graph view makes the complexity of the model more visible to the architect. The bridges between entities make the separation task more obvious to the architect. The microservices, that result from the calculation process, should have a maximum autarchy to minimize the necessary communication between each other. You can think of it as group of islands, that are organized in a way, that they need only a minimum of transport exchange to save resources. In the case of microservices, this means to reduce network bandwidth with reduced latency for data consistency of the system as a whole.

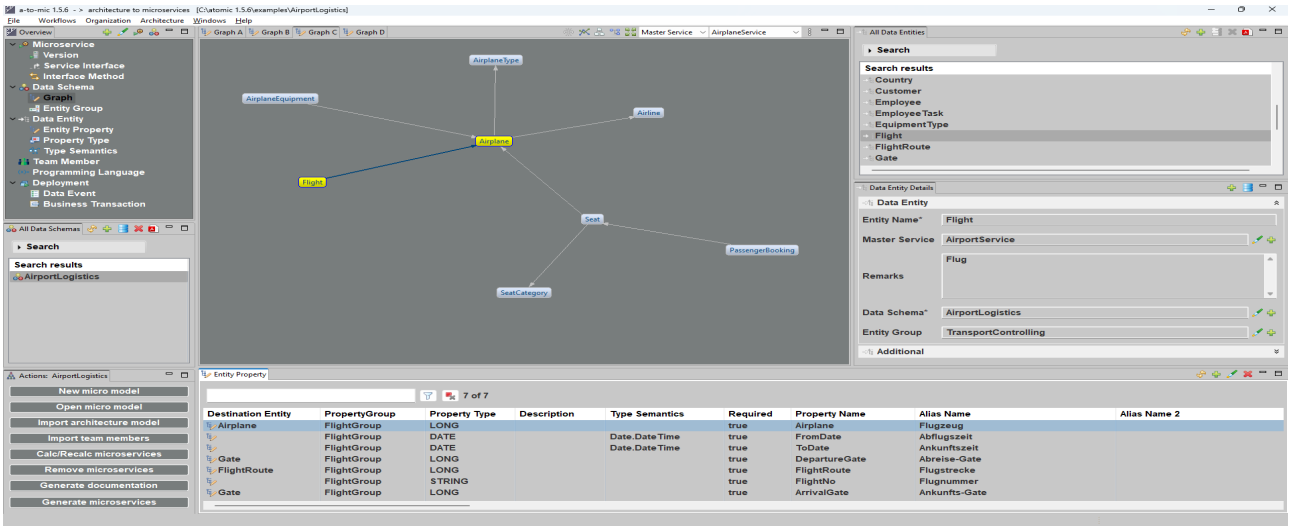


Abbr. 5.2

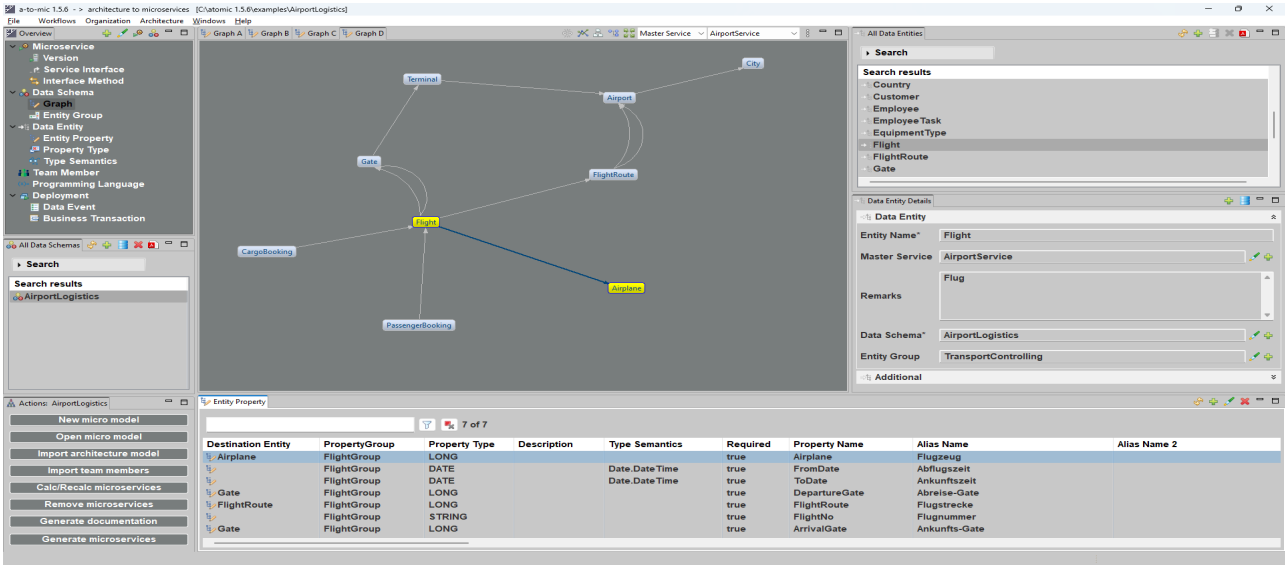
The graph can be split under business aspects in entity groups or under more technical aspects in microservices.



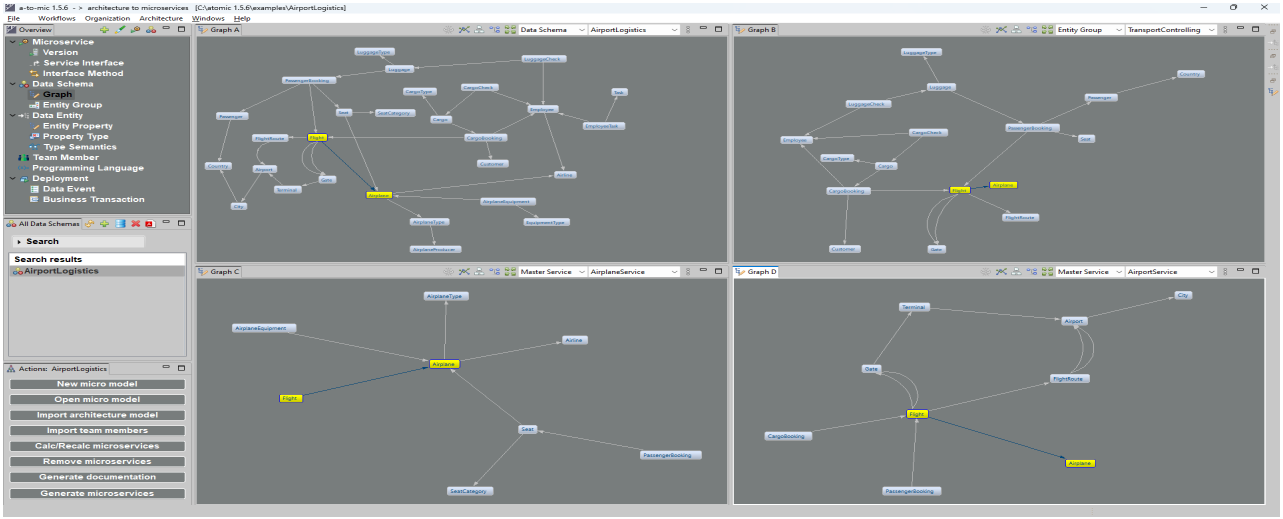
Abbr. 5.3



Abbr. 5.4



Abbr. 5.5

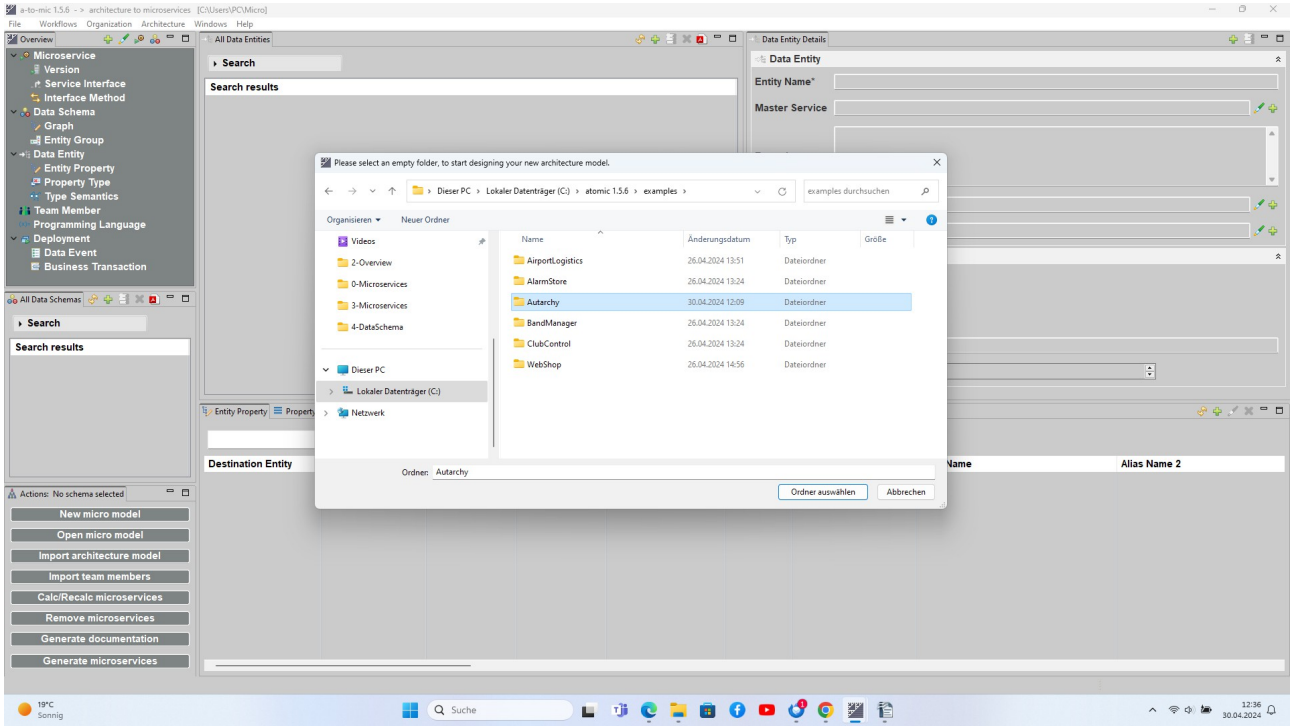


Abbr. 5.6

6. Starting from scratch

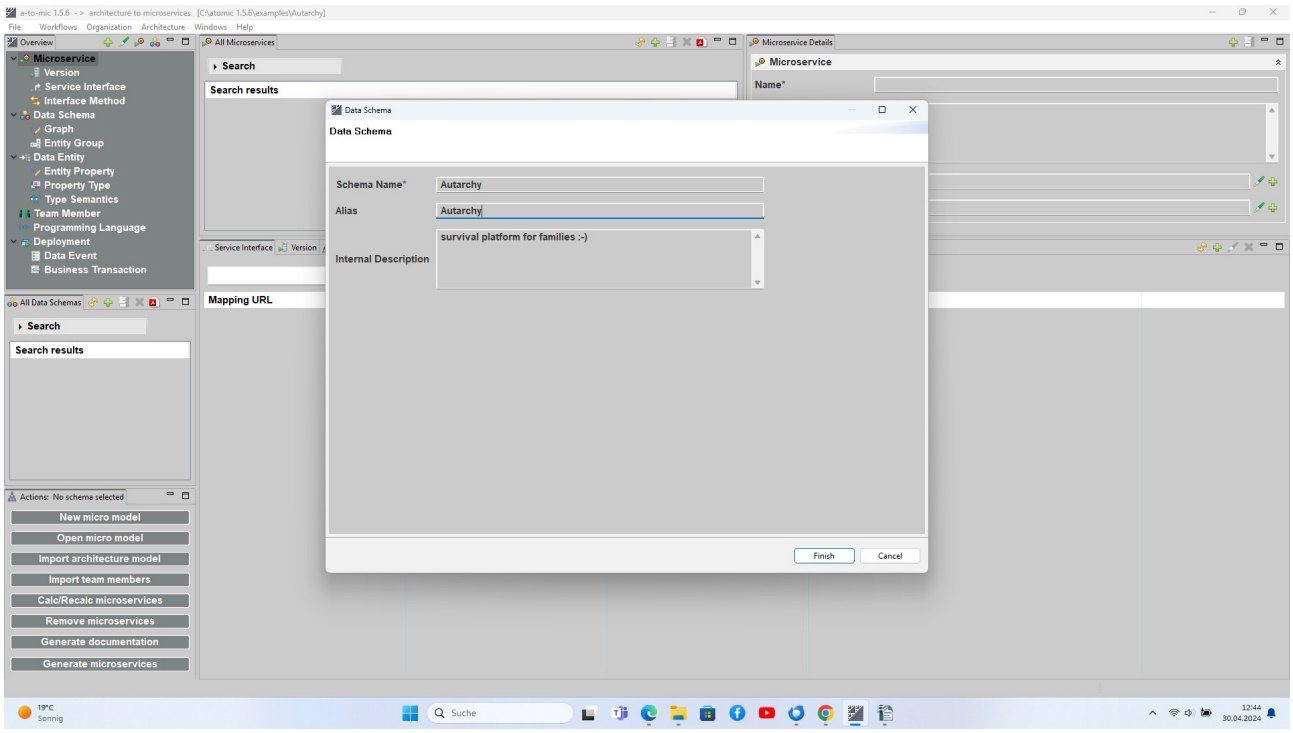
After introducting you to the different perspectives of the application with the „AirportLogistics“ example, we are going to design a new microservice architecture from scratch. The example models an application called „Autarchy“, that helps families to organize themselves in the case of a destroyed civilization, where nothing except the internet still works :-). Feel free to extend this model on your own and please write me a letter, where I can register me and my family in your application to participate from this private network.

We start the example with a fresh started application instance and click on the action button **New micro model**. We create a new empty folder in the example directory and call it „Autarchy“.



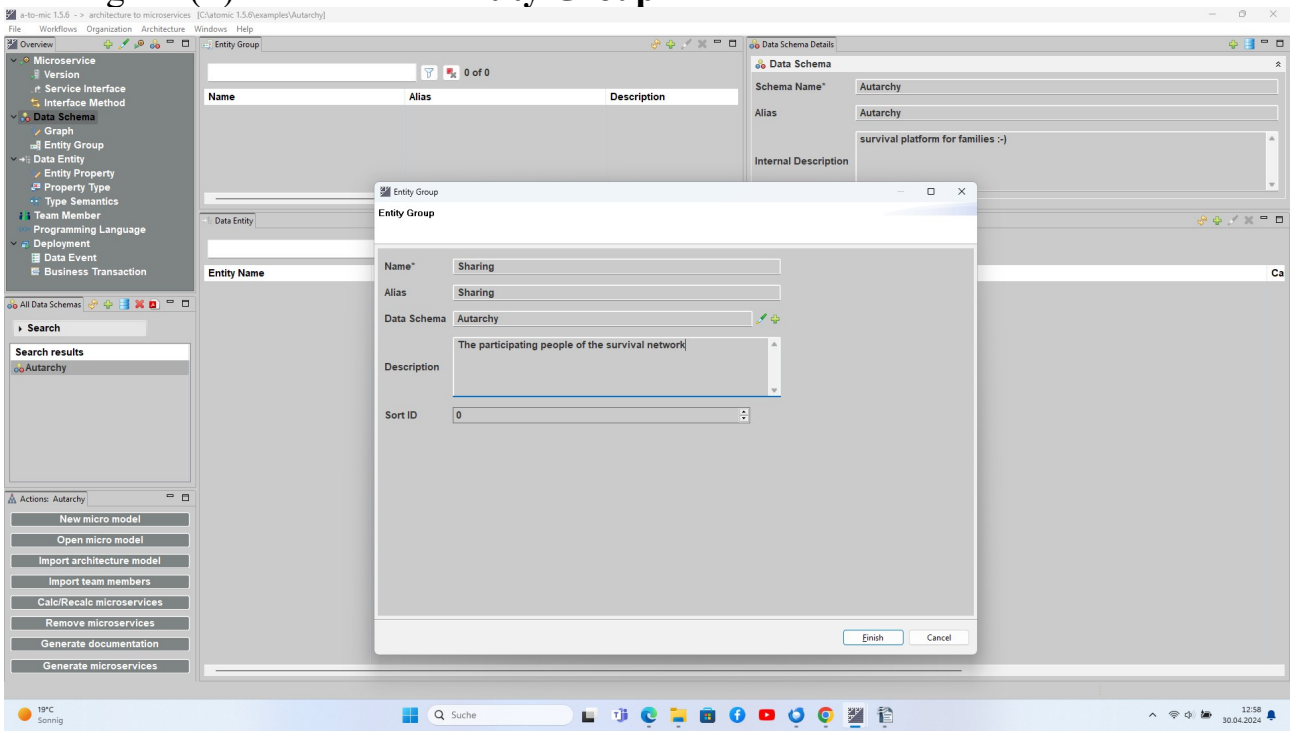
Abbr. 6.1

After the microservice model file **micro.zip** has been created in the new „Autarchy“ folder, we are going to create the data schema of the new application by clicking the (+) toolbar button in the **All Data Schemas** view.



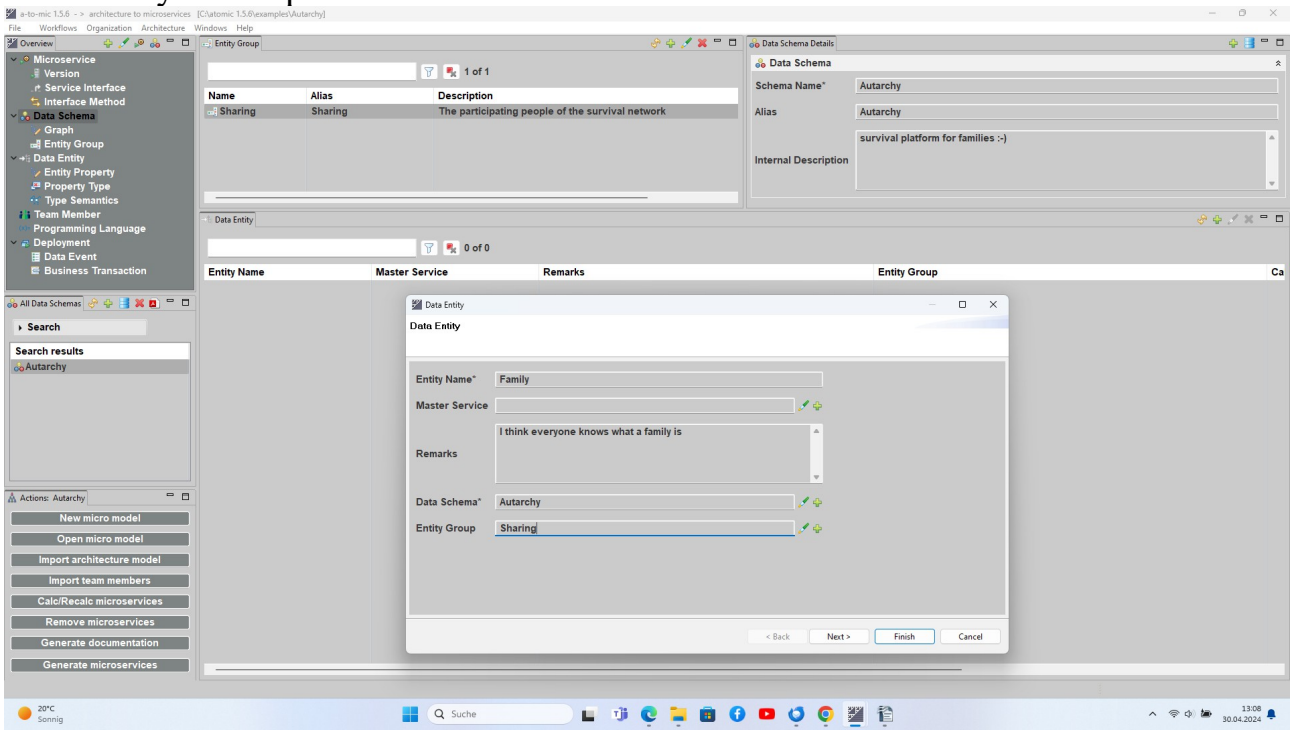
Abbr. 6.2

Now, we switch to the **Data Schema** perspective and create a new entity group with the name „Sharing“, that is associated with the data schema „Autarchy“. We are clicking the (+) button in the **Entity Group** view.



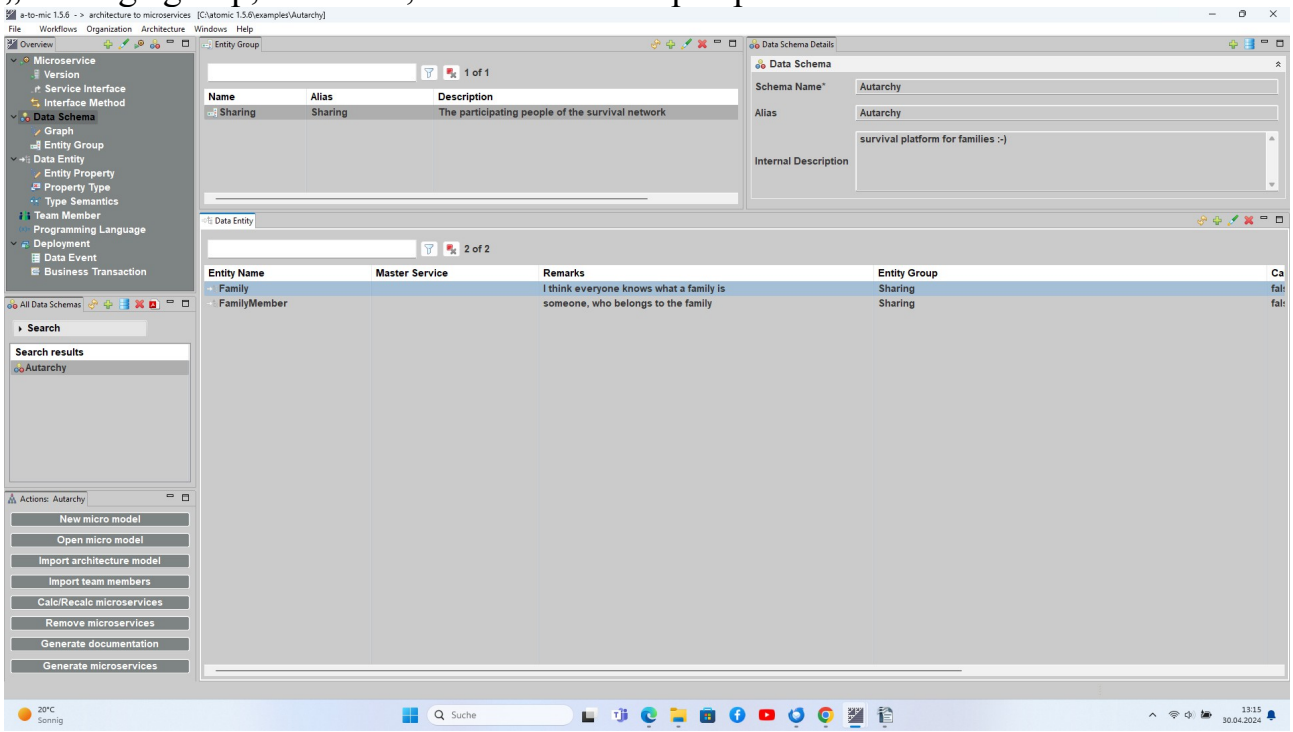
Abbr. 6.3

The next step is the creation of the first entity „Family“. Clicking the (+) button in the Data Entity view opens the wizard.



Abbr. 6.4

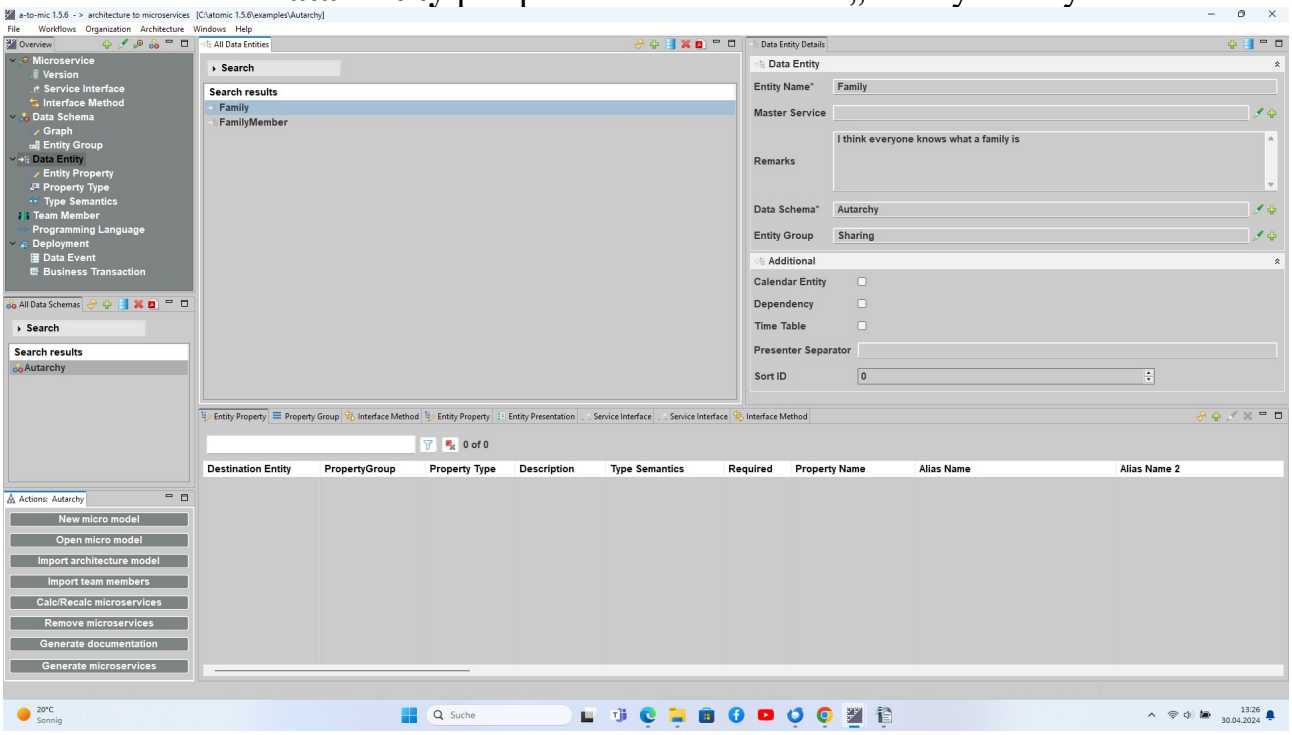
We put the „Family“ entity to the „Sharing“ group, that we just created a moment ago. Next, we create another entity called „FamilyMember“ and add it to the „Sharing“ group, too. Now, the data schema perspective looks like this:



Abbr. 6.5

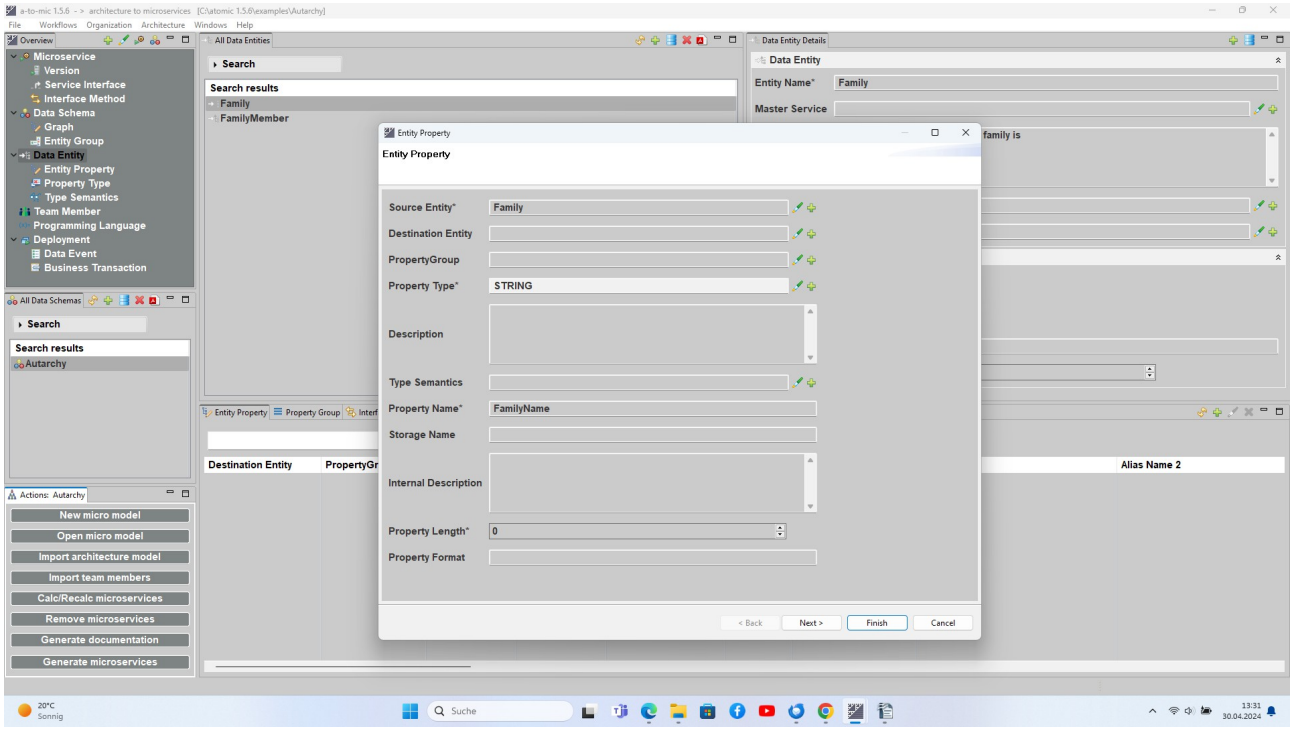
The entities are not assigned to a microservice (Master Service) until now. This will be done by the microservices calculator, after the complete data schema with all its entities, groups, properties and relationships between the entities has been defined by the application architect.

We switch to the **Data Entity** perspective and select the „Family“ entity:



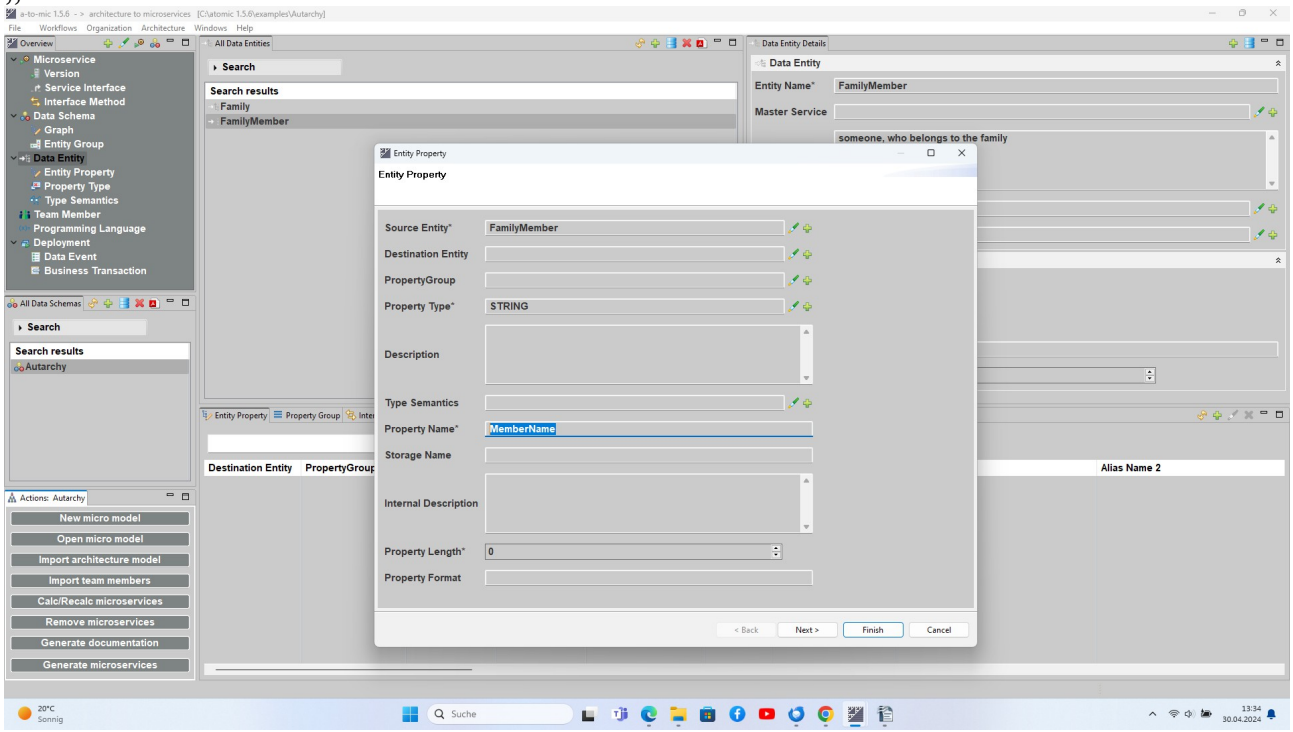
Abbr. 6.6

We want to add a property „FamilyName“ to the „Family“ entity. We click the (+) button in the Entity Property view and the entity property wizard appears on the screen:



Abbr. 6.7

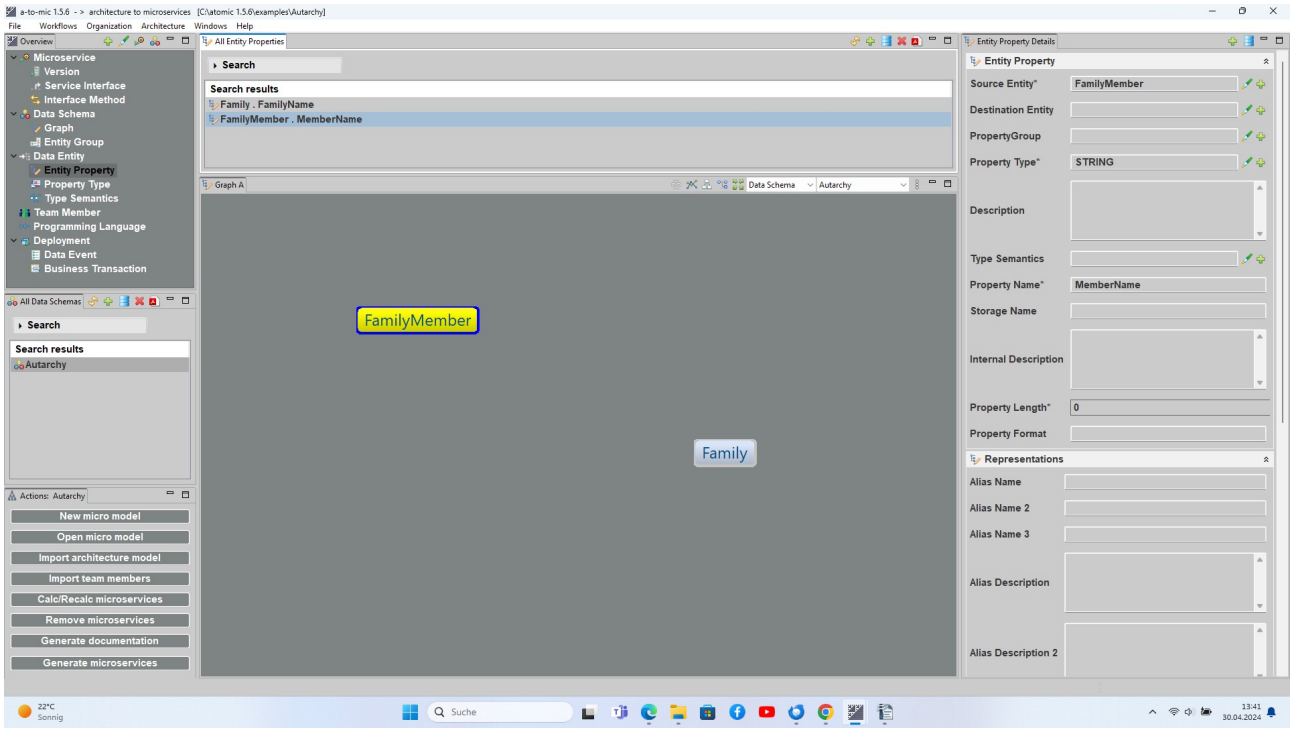
We select the „FamilyMember“ entity and repeat the workflow with the property „MemberName“:



Abbr. 6.8

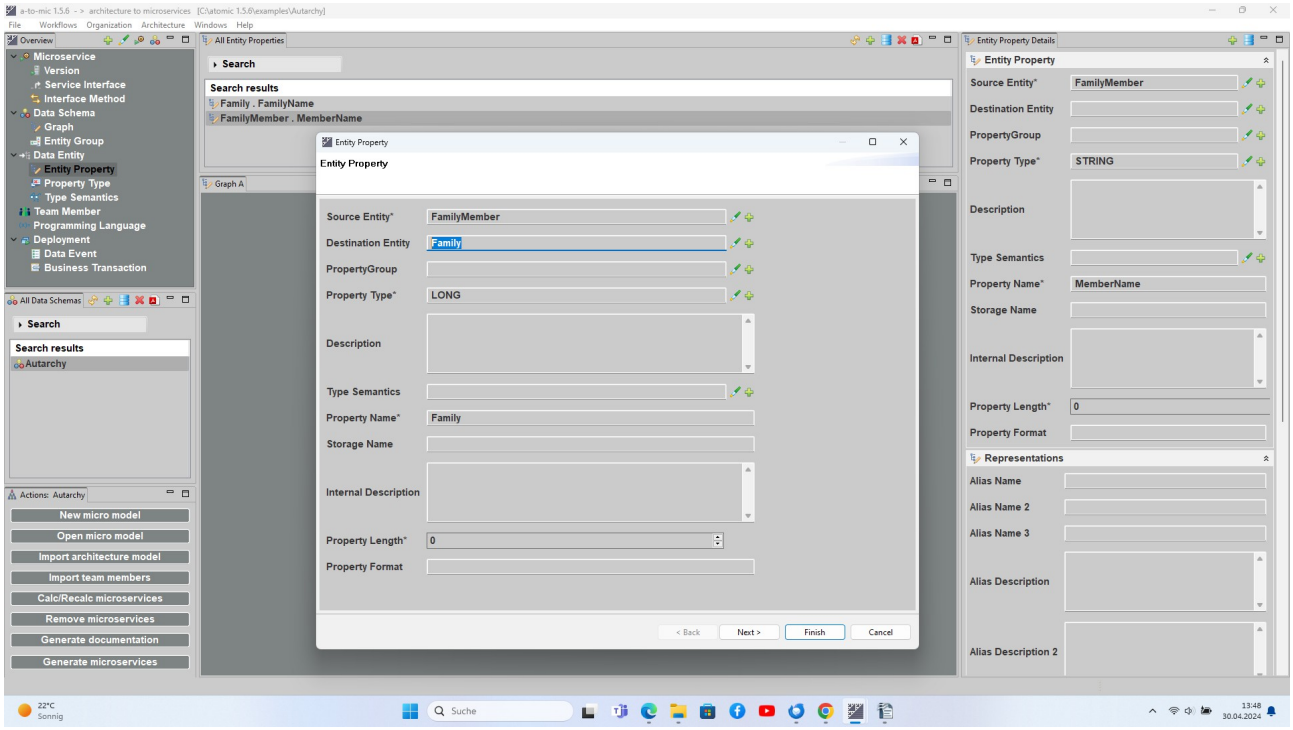
Next, we create a relationship between the two entities „Family“ and „FamilyMember“. The „FamilyMember“ entity is still selected.

We switch to the **Entity Property** perspective and select in the first combo of the graph view the item „Data Schema“ and choose the „Autarchy“ schema in the second combo. The two existing entity nodes appear in the graph view.



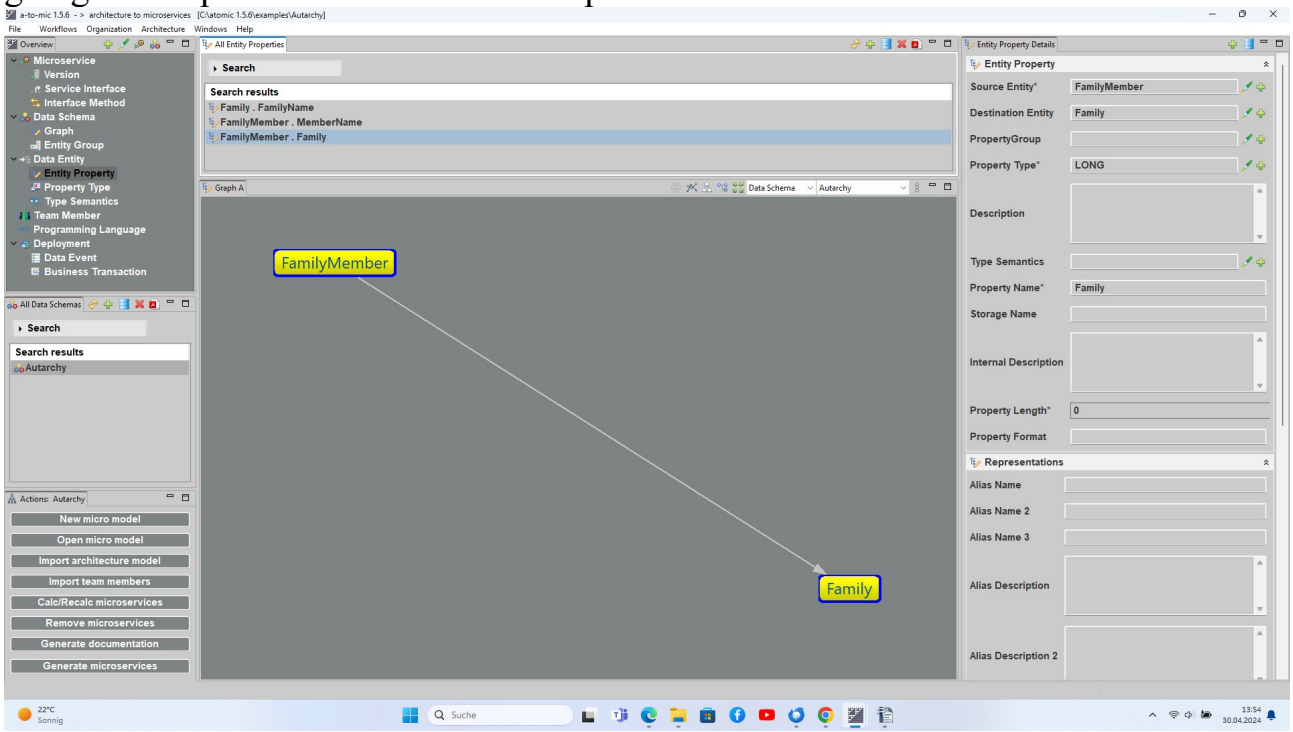
Abbr. 6.9

Clicking on the (+) button in the **All Entity Properties** view opens the entity property wizard once more.



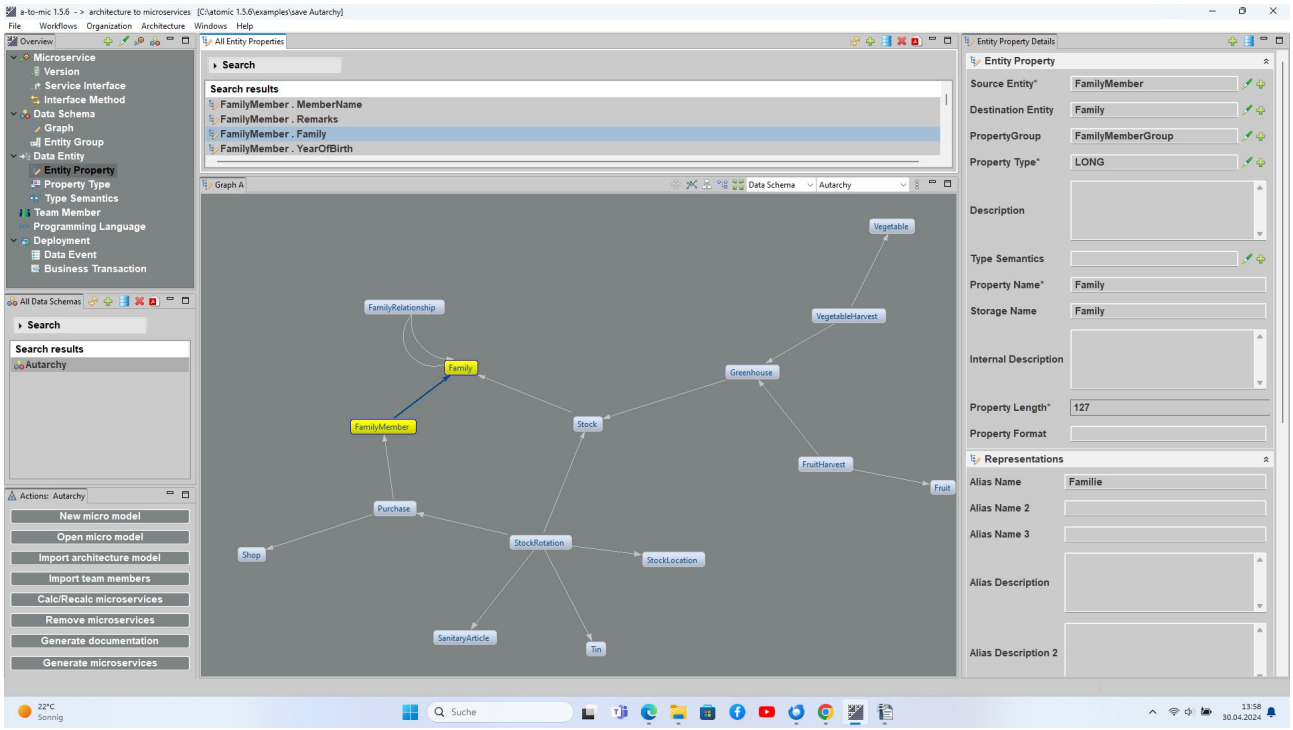
Abbr. 6.10

We select „Family“ as the destination entity. This is equivalent to the creation of a foreign-key-relationship between the „FamilyMember“ and the „Family“ entity. As the property type, we choose type „LONG“, because all of the technical primary keys in our schema should have this data type. After finishing the wizard, the graph view is going to be updated and the relationship becomes visible.



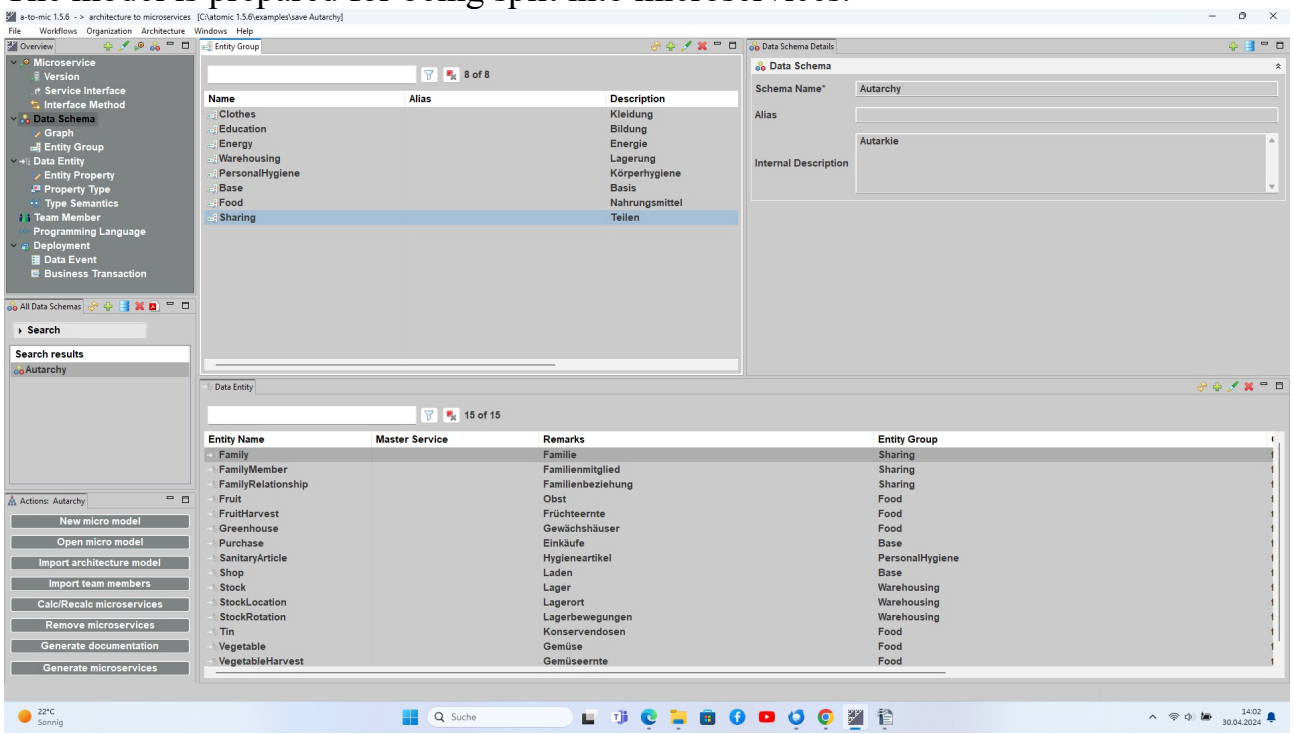
Abbr. 6.11

In the same way, we expand our data schema more and more, until the schema has been expanded to the final data model state.



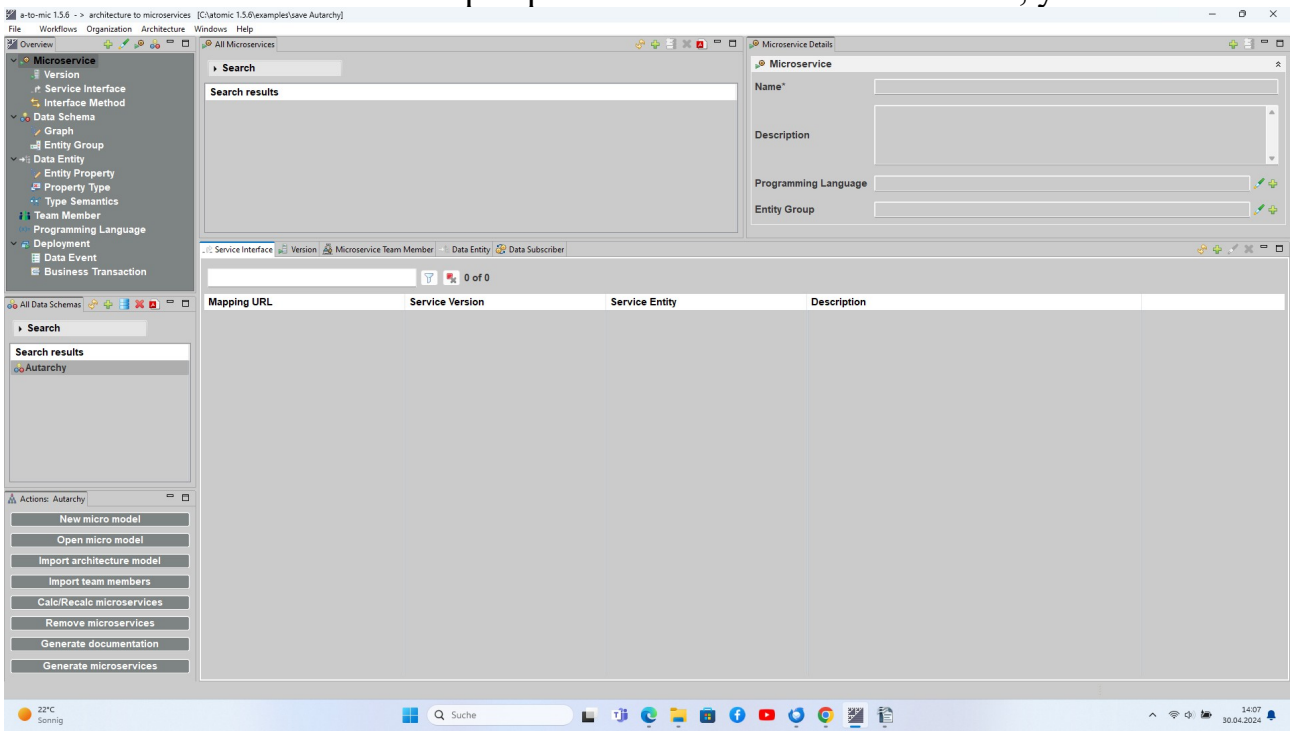
Abbr. 6.12

The model is prepared for being split into microservices.



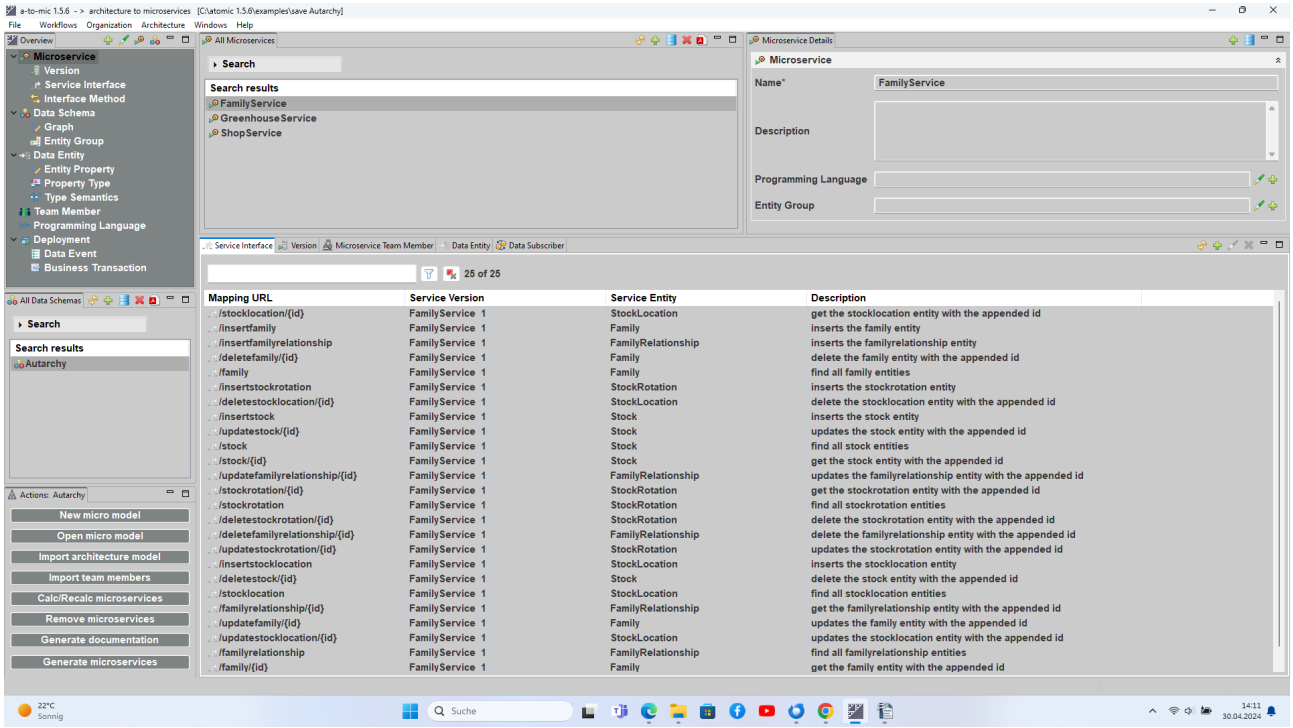
Abbr. 6.13

We switch to the microservice perspective and see no microservices, yet.



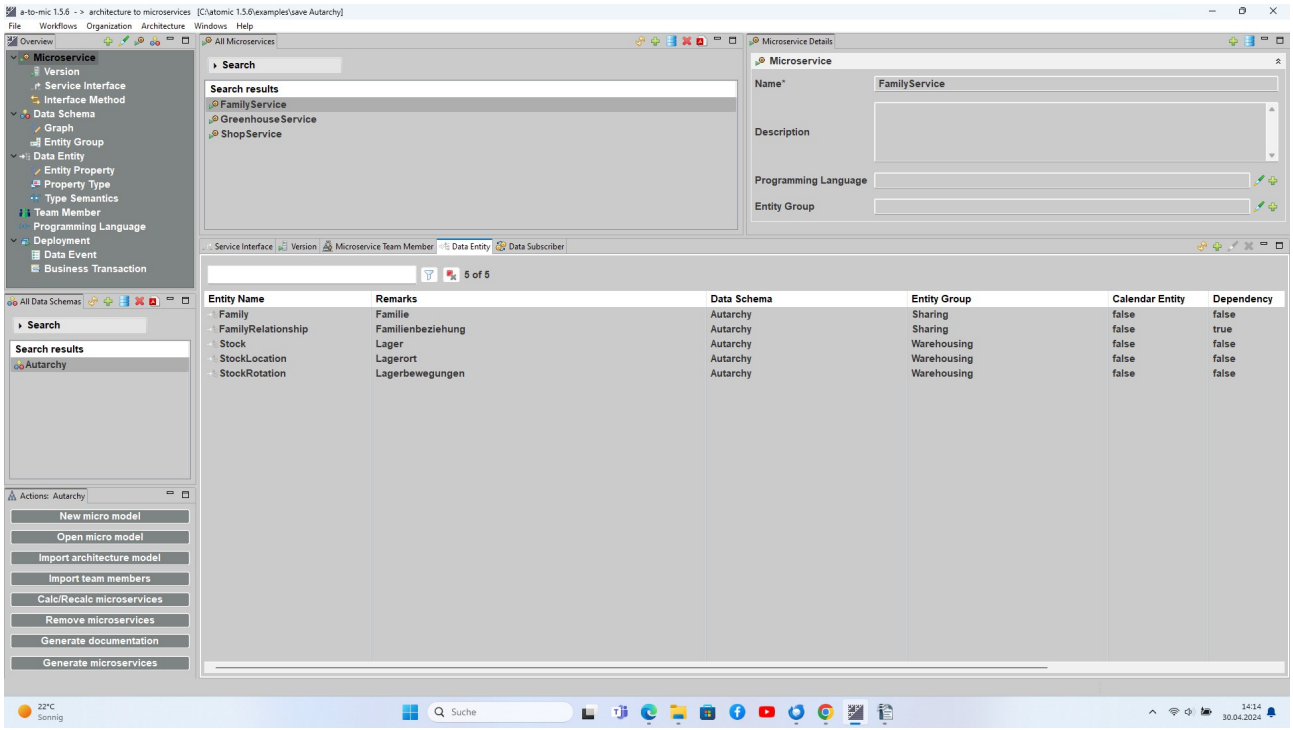
Abbr. 6.14

We click on the **Calc/Recalc microservices** button. Three microservices have been created by the microservice calculator. We select the „FamilyService“ and analyse, what has been changed in the architecture model.



Abbr. 6.15

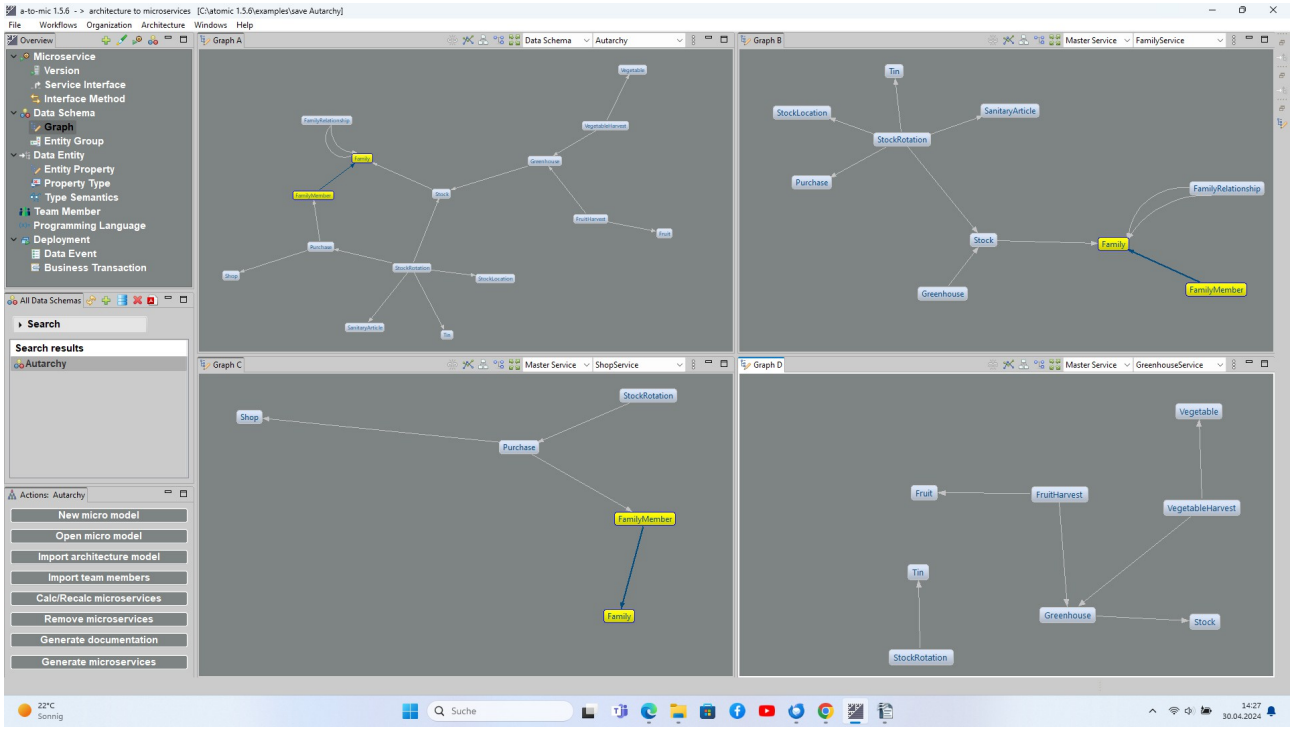
The calculation algorithmn has created a huge amount of REST interfaces for the „FamilyService“. In the **Data Entity** view, we can see, for which entities the „FamilyService“ is responsible to keep data consistency.



Abbr. 6.16

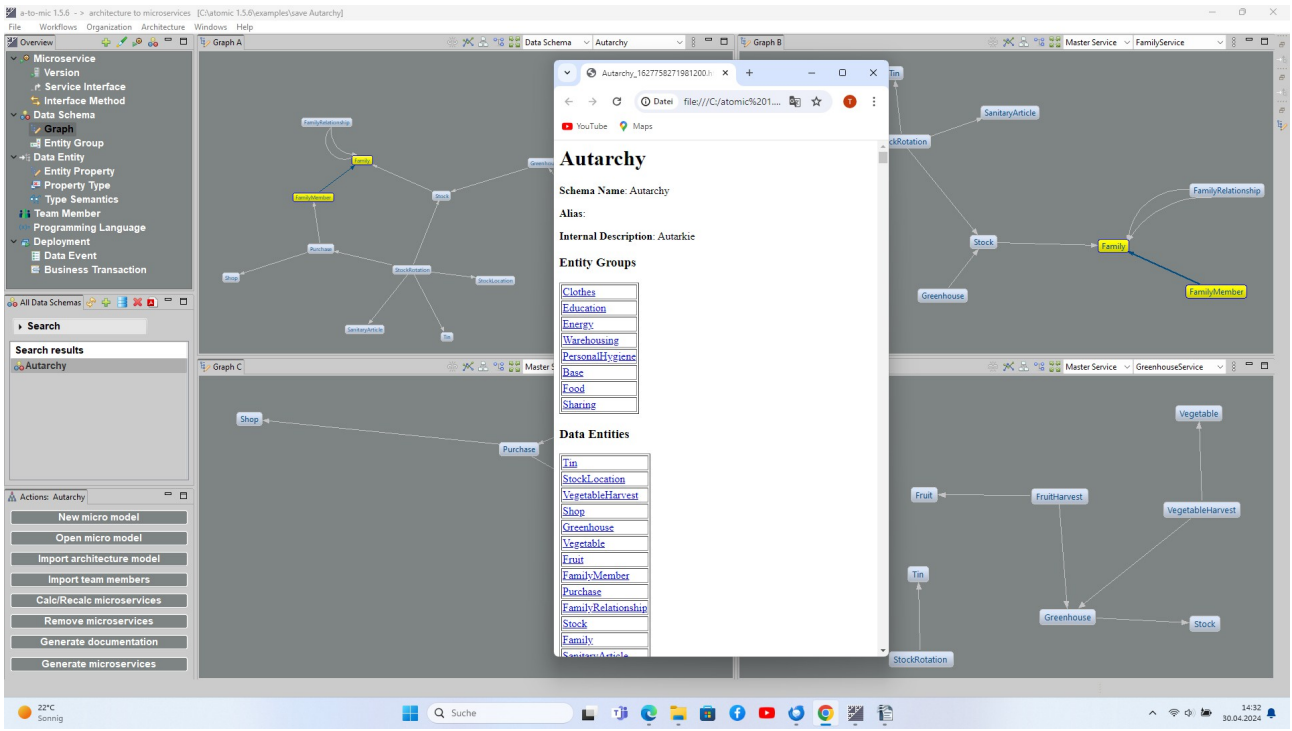
In the **Data Subscriber** view, there can be found the entities, that are hosted in other microservices, but are needed by the entities of the „FamilyService“. This information is important for the messaging system, that has to update those entities in the local data source of the „FamilyService“ with as little latency as possible (Eventual Consistency).

In the **Graph** split view, we can see, what the microservices calculator has done with our „Autarchy“ data schema:



Abbr. 6.17

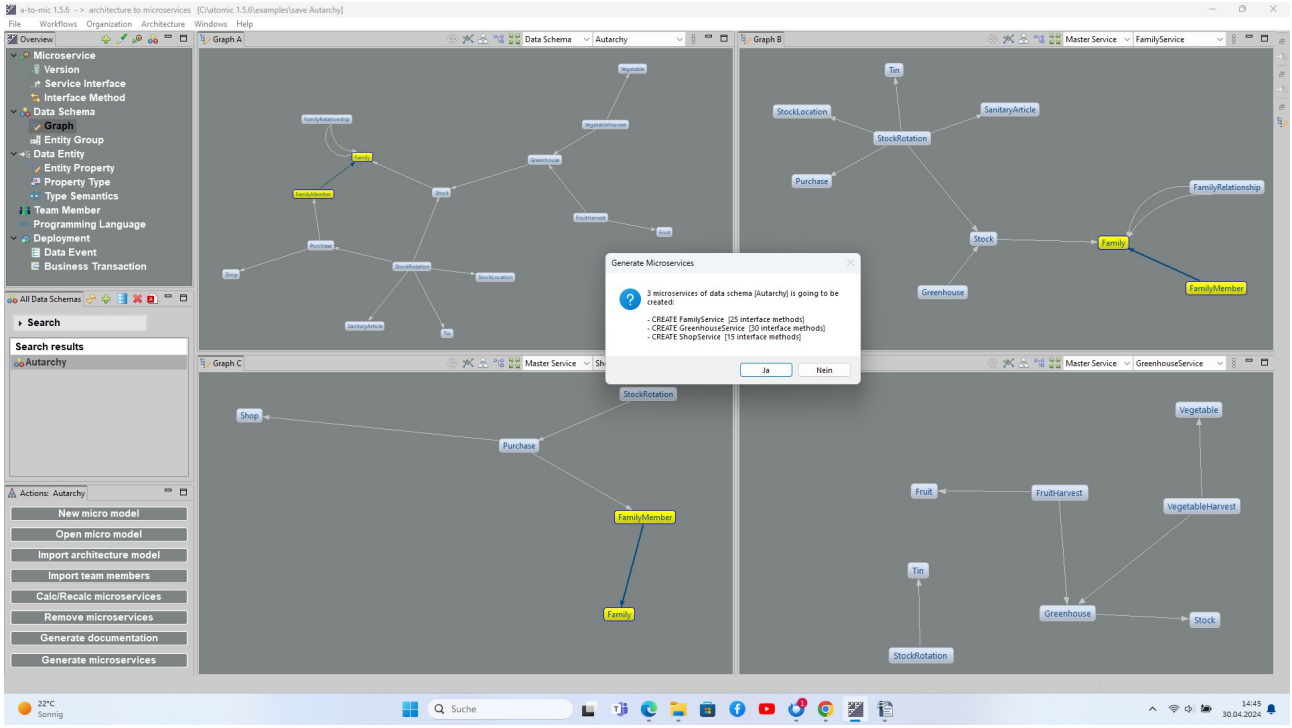
If the architect has been satisfied with the microservice structure, the documentation can be generated by clicking the **Generate documentation** button in the **Actions** view.



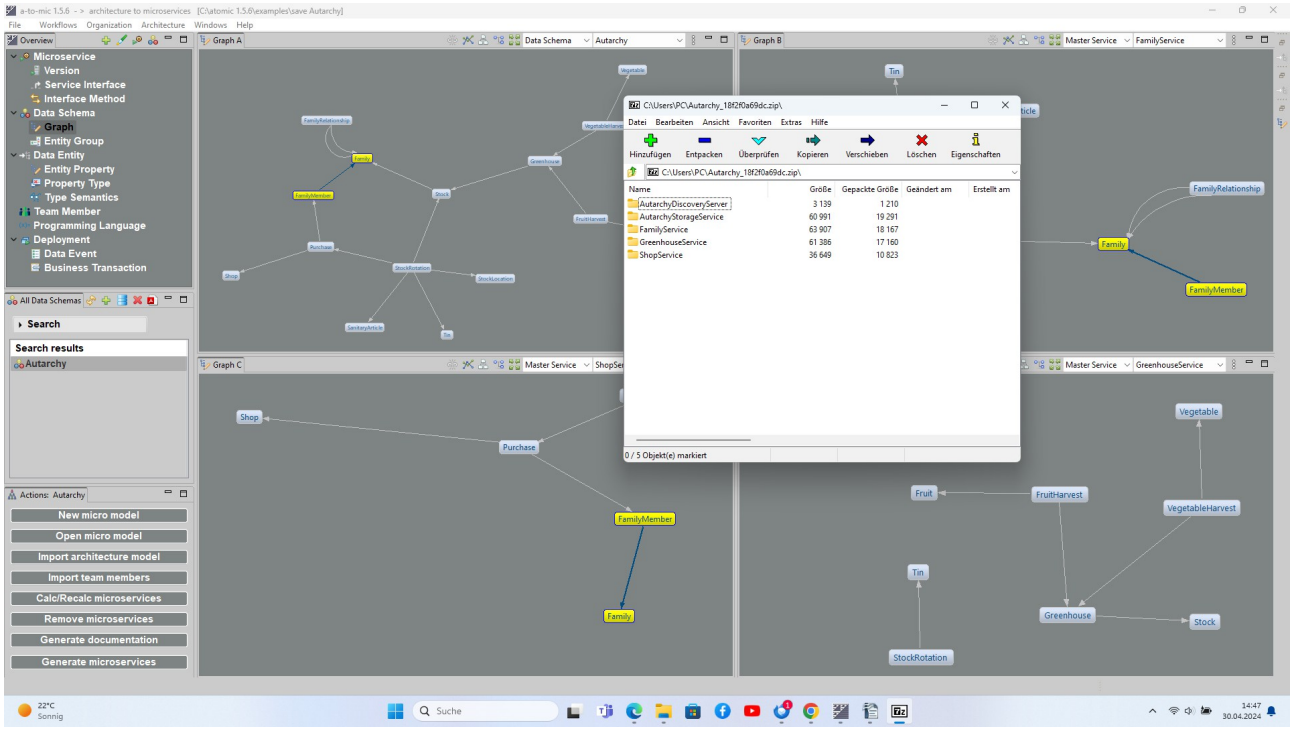
Abbr. 6.18

The generated documentation is a single HTML file with internal links to the relevant architecture artifacts. In this way, the is easy to send it by email to other architects for discussion with people that do not have A-TO-MIC installed on their device.

The final step is the generation of the distributed microservices application for deployment. The generated meta-model microservices can be transformed to initial JAVA Spring-Boot projects for the build process. To use this feature, you must have a licence. Please take a look at the www.a-to-mic.com/licences for further information. After clicking the **Generate Microservices** button in the Actions view, a zip file is going to be created that contains all microservices organized in Spring-Boot projects.

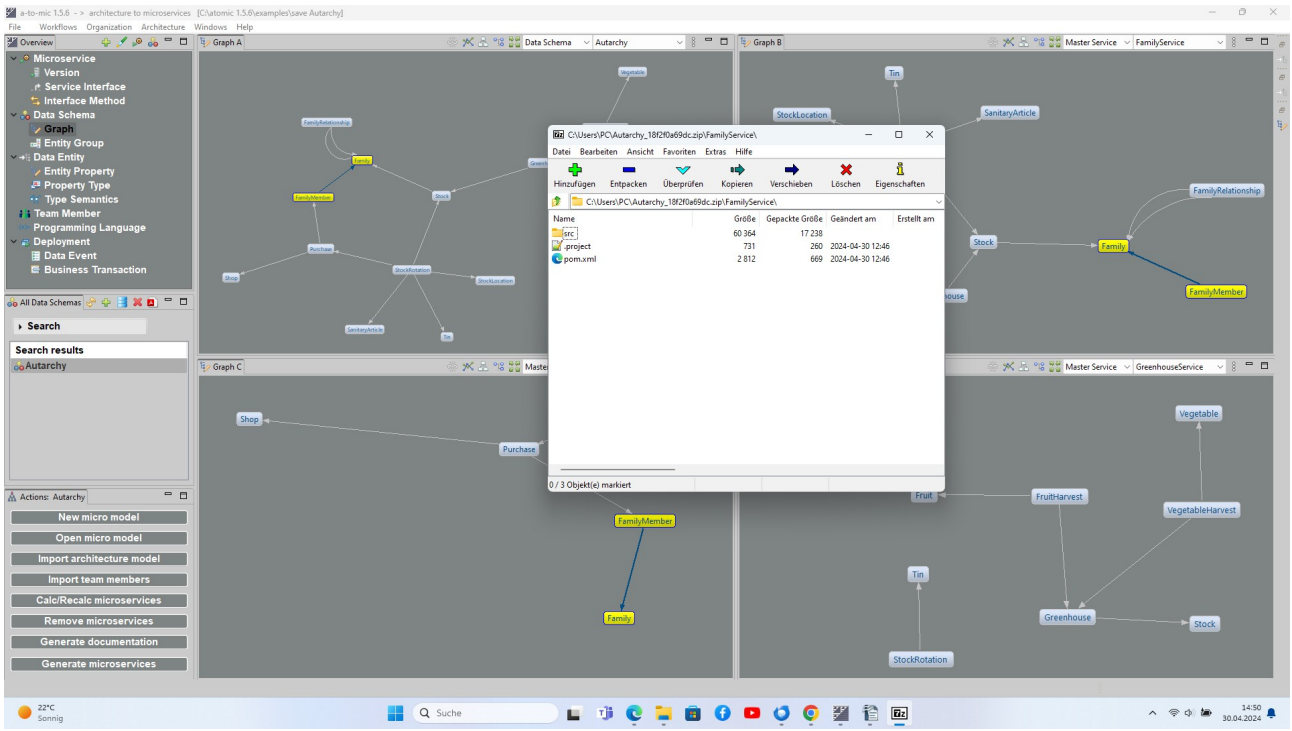


Abbr. 6.19



Abbr. 6.20

Opening one of the Spring-Boot projects shows the project root folder of the „FamilyService“ project:



Abbr. 6.21

Thank you for reading, please visit www.a-to-mic.net for more information.