Gorgeous Food

Monolith Decomposition

ARQSOFT 2019 / 2020

1131485 Pedro Coelho

1150863 Henrique Maciel

1160907 João Freitas

1161258 Vítor Melo

Content

Figures 2
ADD 4
Generic Aspects Related to all ADD iterations 4
Code 4
Architecturally evident coding style 4
Architecture Tradeoff Analysis Method 4
Design 4
Front-end 5
Decomposition Approach 5
Monolith decomposition strategies comparison 5
Why discard the Business Capabilities decomposition? 5
Services Granularity 6
Services Interface 6
Data Management strategies 6
Achieved Architecture Design 6
Module View 7
Component and Connector View 10
Allocation View 11
Research Topic 14
Introduction 14
Importing Data and Concepts 14
Algorithms 14
Results & Assessment 14
References 17

Figures

Figure 1 - Domain Model of the application as a UML class diagram 7
Figure 2 - Model Objects of the application as a UML class diagram 7
Figure 3- Aggregate Roots of the application as a UML class diagram 8
Figure 4- Meal Management Service Packages 8
Figure 5 - Item Management tService Packages 9
Figure 6 - PoS Handling Service Packages 9
Figure 7 - User Management Service Packages 10

Figure 8 - Report Management Service Packages 10
Figure 9 - Application Components Interaction in a Coarse View as a UML components diagram 11
Figure 10 - Components Physical Allocation as a UML deployment diagram.....12
Figure 11 -Use Cases Allocation as a UML use case diagram 12
Figure 12 - Domain Objects as a UML diagram 13
Figure 13- Result from Girvan-Newman algorithm with 5 services 15
Figure 14- Generated Service Composition with Leung´s algorithm.....16

ADD

Generic Aspects Related to all ADD iterations

In all ADD iterations, there are a set of architectural drivers that are mentioned in each iteration, being these the *use cases*, *quality attributes, concerns* and *constraints*. Use cases are defined as the *tangible* functionalities of the software which the users can interact with and quality attributes a set of metrics that evaluate the use cases in order to maximize the produced value of the software and user experience. Concerns and constraints are conceived as a set of worries and obligations in which the software must take into account.

The decision of which architectural drivers are to be addressed in each iteration is given by evaluative metrics. For example, use cases and concerns are evaluated with their prioritization/importance and difficulty, that can either be valued as low, medium and high. Constraints and quality attributes don't have a defined value of prioritization and difficulty, as these are obligatory and transversal to all software architecture.

Code

Architecturally evident coding style

The design for the Gorgeous Food application was based on the principles of Domain Driven Design, and as such, this choice is reflected in the team's coding style. This stylistic choice is clearly seen by the adoption of interfaces to represent the building blocks found in DDD, such as Entities, Value Objects, Services and Repositories.

When it comes to the adopted architectural pattern, MVC, it also influences the way the team writes code, by structuring each component into Model, View and Controller packages. This way, the code found in the Model package manages the data and business logic of the application; the code in the View package supports the presentation of the Model data; and finally, the Controller package is responsible for managing the user interactions with the application, passing the input to the model.

In terms of the communication between the GFAW and the Item service, the usage of an API Gateway is also reflected in the application's code. The API Gateway provides a single-entry point between these two components. This way, the GFAW doesn't need to know the Item service's URL and communicate directly with its API. Instead, the API Gateway provides the GFAW with an API that is appropriate for its requirements.

Architecture Tradeoff Analysis Method

Design

High service coupling in exchange for faster implementation time. For now, every service has a direct connection with another service. Implementing an API Gateway reduces coupling by aggregating service discovery and by providing a single-entry point for other connected services. (No risk)

Higher maintenance costs, in exchange for lower service coupling. By following a Database per Service strategy, we can make changes to a service's database without it affecting other services. On the other

hand, we also have a system that is harder to maintain and any transaction that spans multiple services is harder to implement. (No risk)

Service unavailability. It is possible that one or more services from the same responsibility to be down that compromises the data fetching. As the business grows, so the complexity of the services or the quantity of them, impacting the maintainability if those are managed by a single team. (No risk)

Harder deployment. There must be a plan and automatization to deploy the services, because while the number of services grow the more effort is needed for deployment. (No risk)

Front-end

Lack of appropriate and detailed error messaging, this makes a possible error harder to debug and therefore the GFAW harder to maintain, however, it also reduces the time spent on implementation. (Risk)

General performance issues. While for now the communication is synchronous it is normal to have asynchronous requests, so the front-end can have delays to get the information. (No risk)

Automatic synchronization after each backend operation leads to worse performance in exchange for instant data updates. (No risk)

Wrong stylization of elements may cause a poor application experience. The use of stable libraries reduces implementation time and increases consistency of presentation while using a bit more of system resources. (No risk)

Decomposition Approach

Monolith decomposition strategies comparison

Decomposition by Business capabilities: A business capability represents what a business does in a domain to fulfill its objectives and responsibilities. Each microservice expose an API that developers can discover and use freely.

Decomposition by Sub-domains: Start by finding the domain boundaries in an existing monolith. By applying domain driven design techniques to find the bounded contexts, we can define the microservices boundaries and their respective domain(s).

Decomposition by an external tool: Start by mapping a representation of the system specification and supply it as an input file to the tool, which will output a possible decomposition strategy.

The team ended up adopting the solution developed following the decomposition by Sub-domains (Bounded Contexts). This decision came down to a few decisive factors when comparing the three approaches.

Why discard the Business Capabilities decomposition?

By decomposing the monolith by its Business capabilities, a Point of Sale Handling service would be dedicated to managing the Item's purchase, since it's one of the application's main business capabilities. However, since in the context of this application the Item Purchase is a simple transaction which will

remove an item and register the buyer's User Type, the team decided that a dedicated service for the Item Purchase was an unnecessary and time-consuming solution.

Additionally, the Business Capabilities approach described an Informational Data service, which would allow a user to consult the Meal's details, namely its Ingredients, Allergens and Descriptors. However, since the Meal Ingredients, Allergens and Descriptors aren't used by the application except when working with the Meal entity, the team decided that decomposing this functionality into its own service would be counterproductive.

Services Granularity

Meal Management service: Responsible for managing the Meals data and its Ingredients, Allergens and Descriptors.

Item Management service: Responsible for managing the Items data and the Identification number generator.

PoS Handling service: Responsible for managing the Points of Sale data, downtimes, Item transfers as well as Item purchases.

User Management service: Responsible for communicating with the school's user directory and managing the system users.

Report Management service: Responsible for generating reports on the user activities within the application.

Services Interface

In order to supply the GFAW with the functionalities it needs from the services, an API Gateway was created. By using the API Gateway, the GFAW won't need to know the URLs of the specific services it will use, instead communicating with the services via an API provided by the Gateway. This will result in a lower coupling between the services and their 'clients'. Additionally, it also allows the team to provide specific APIs depending on the needs of the 'client' application. For example, the GFAW and the PoS application might have different requirements, and therefore different APIs can be exposed.

Data Management strategies

For managing the data between the various services, the team settled on the Database Per Service strategy with a Schema-per-Service approach where each service has a database schema that's private to that service. Following this strategy will ensure the services are loosely coupled and can use the type of database best suited for its needs.

Achieved Architecture Design

The purpose of the new *Gorgeous Food Application* design was to decompose the existing monolith solution in microservices in order to provide more availability when using the application. Given the approached decomposition strategies, the following views illustrate the design that was conceived for the proposed solution:

Module View

This view illustrates all logic and implementation units that were considered for the architecture of the solution. In Figure 1, the domain concepts of the application and their relationships are exposed.

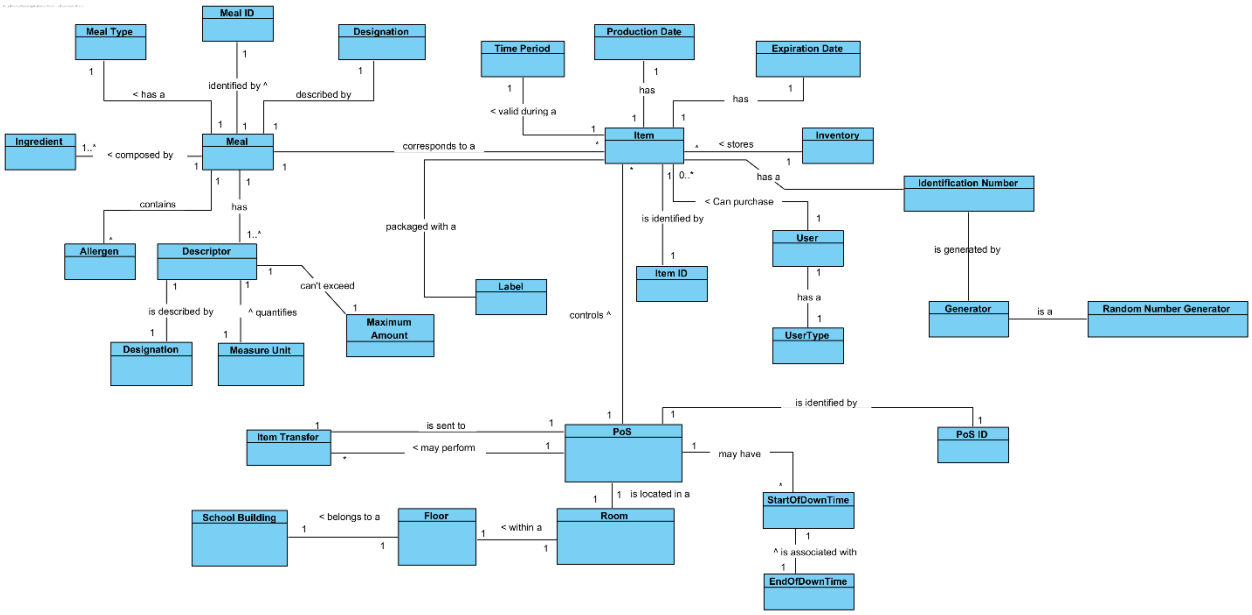

Figure 1 - Domain Model of the application as a UML class diagram

Figure 2 translates the domain concepts as *classes*, reflecting the behavior that is produced by these.

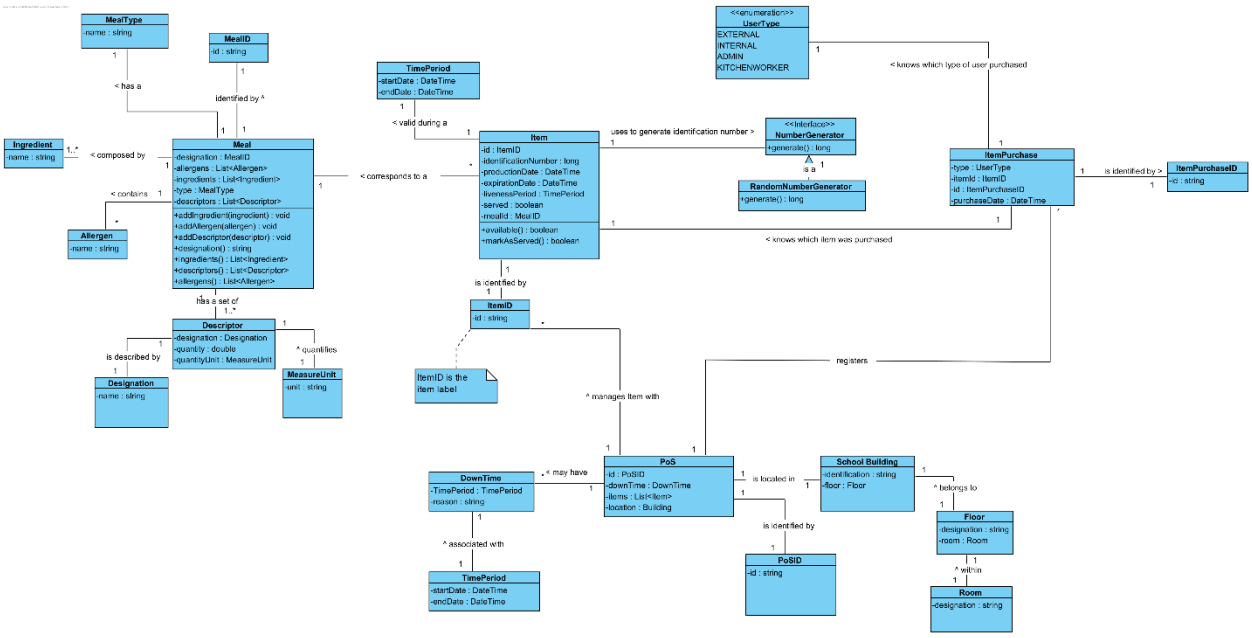

Figure 2 - Model Objects of the application as a UML class diagram

Figure 3 also translates the domain concepts as classes by exposing these using the DDD patterns that were adapted for the solution.

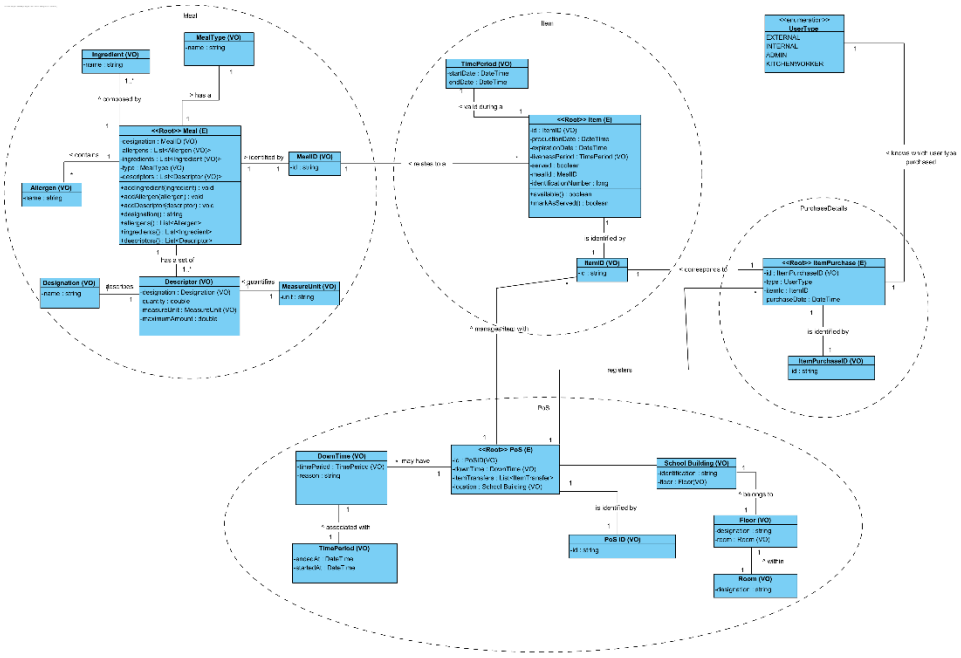

Figure 3- Aggregate Roots of the application as a UML class diagram

The following figures represent the structure of the proposed microservices packages.

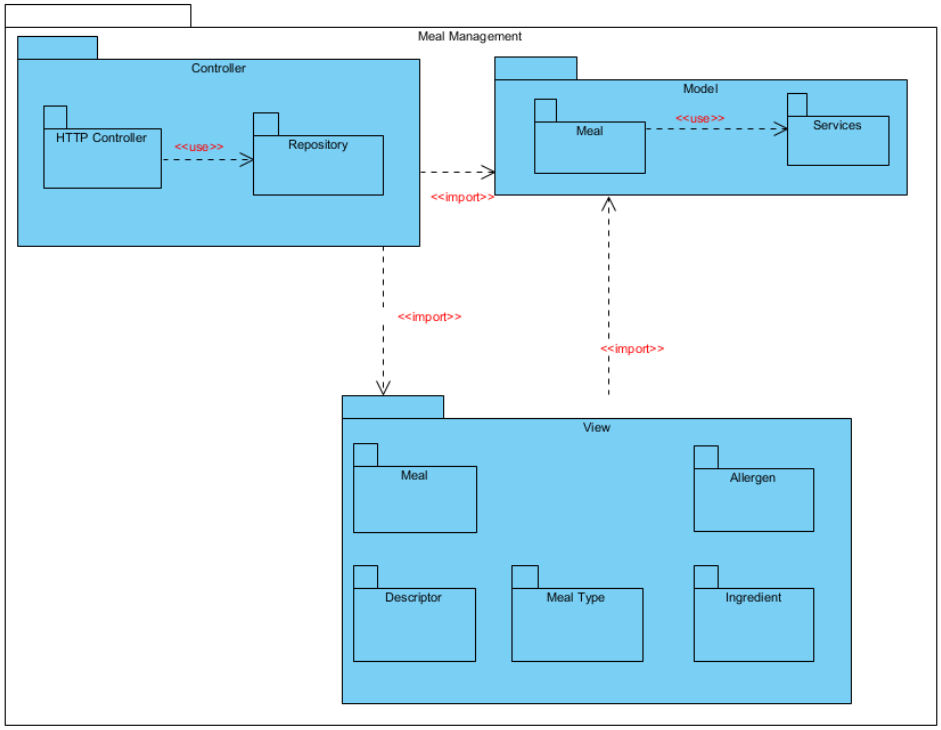

Figure 4- Meal Management Service Packages

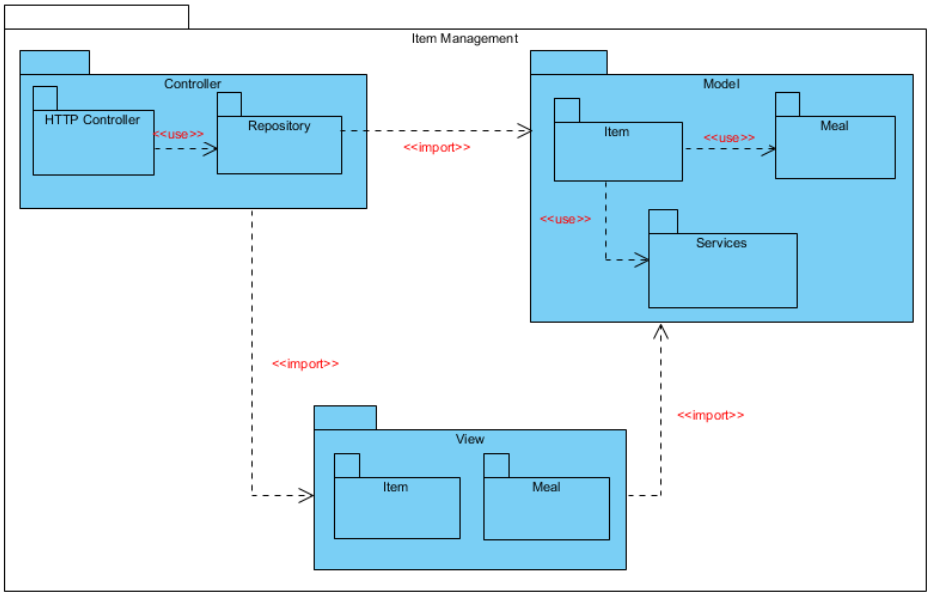

Figure 5 - Item Management tService Packages

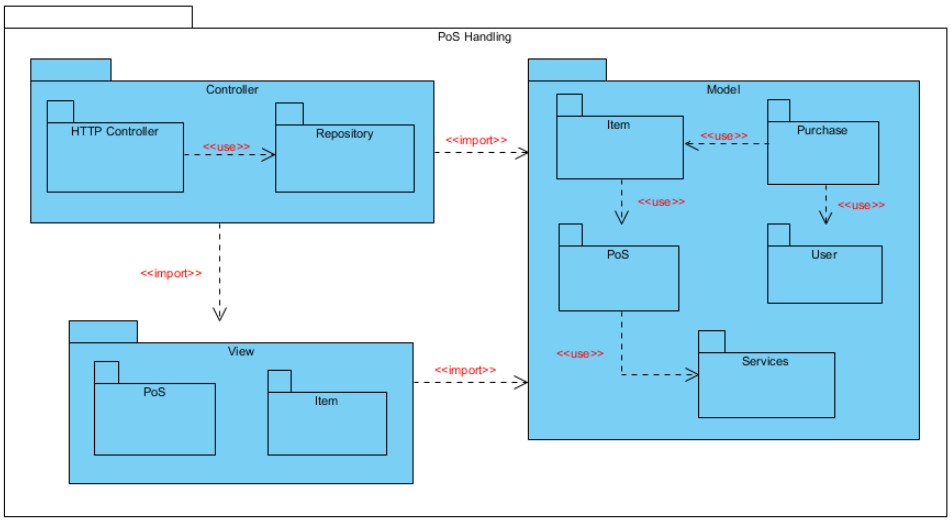

Figure 6 - PoS Handling Service Packages

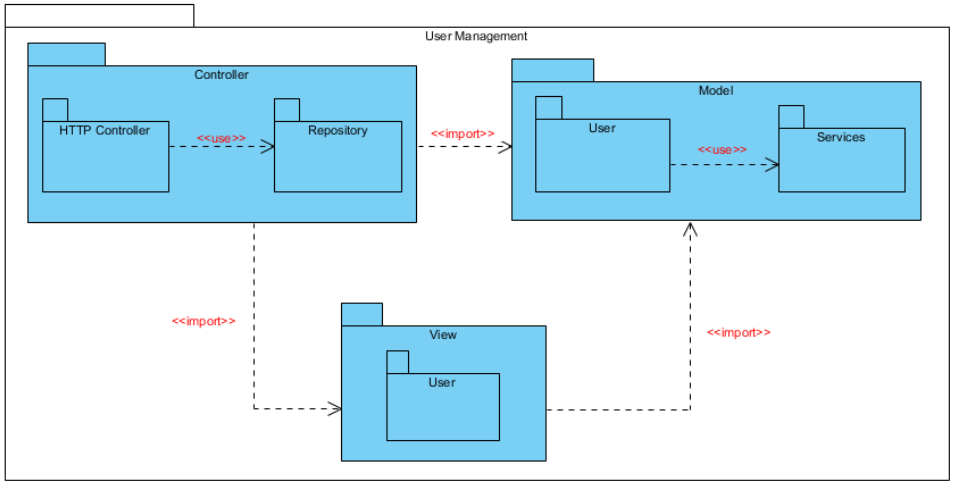

Figure 7 - User Management Service Packages

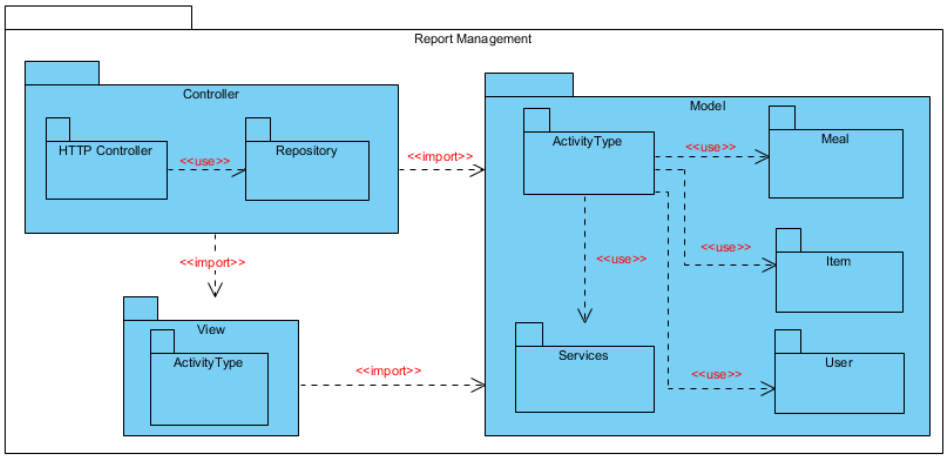

Figure 8 - Report Management Service Packages

Component and Connector View

This view illustrates all logic components interaction that were considered for the architecture of the solution. In Figure 9, the components that assemble the microservices can be viewed as well as the interfaces that these produce and consume.

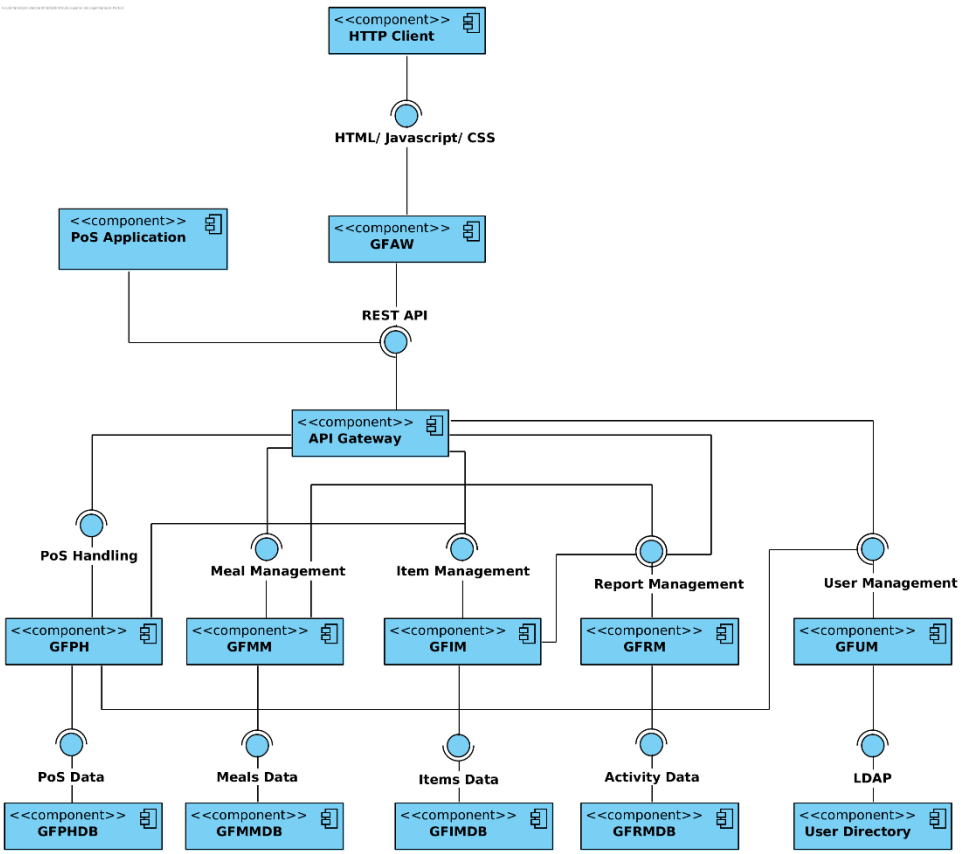

Figure 9 - Application Components Interaction in a Coarse View as a UML components diagram

Allocation View

This view illustrates all software and non-software allocations were considered for the architecture of the solution. Figure 10 represents the physical allocation of the application components. Figure 11 and 12 represent the allocation of the responsibilities that each application actor has regarding the existing functionalities.

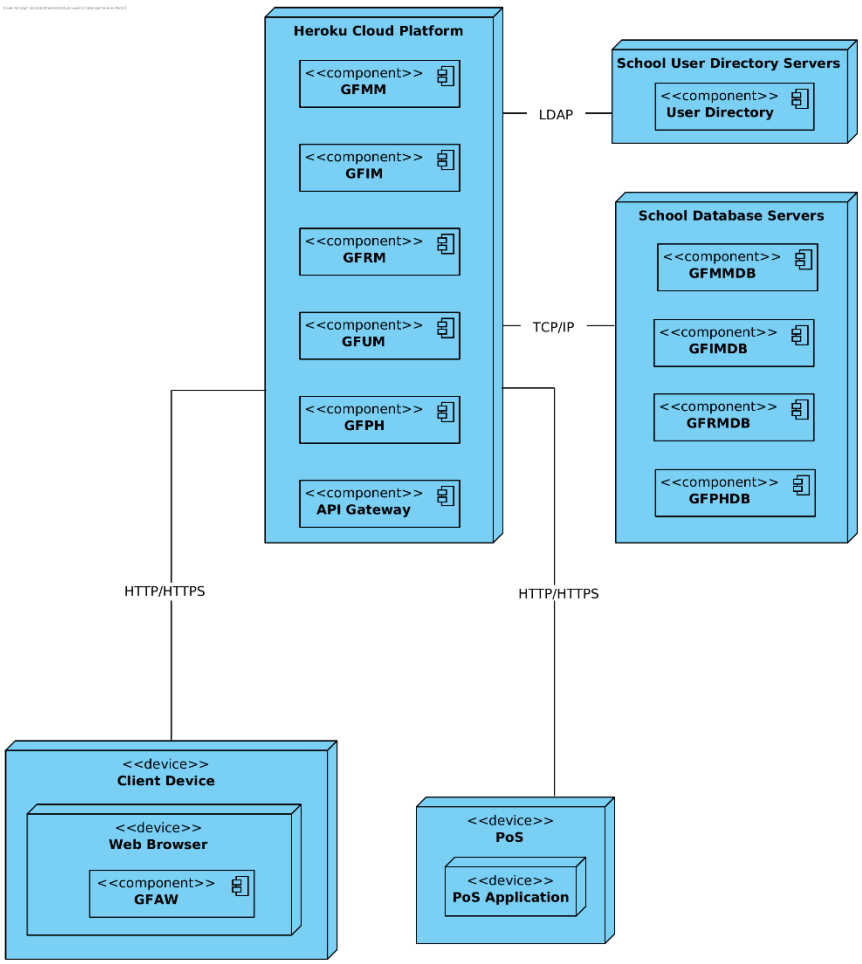

Figure 10 - Components Physical Allocation as a UML deployment diagram

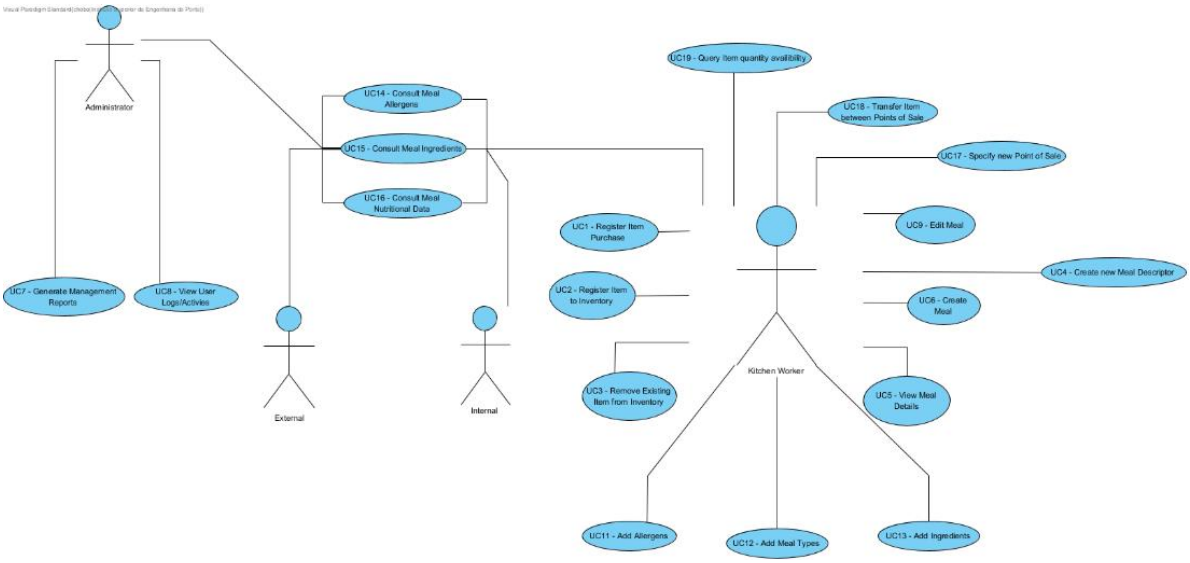

Figure 11 -Use Cases Allocation as a UML use case diagram

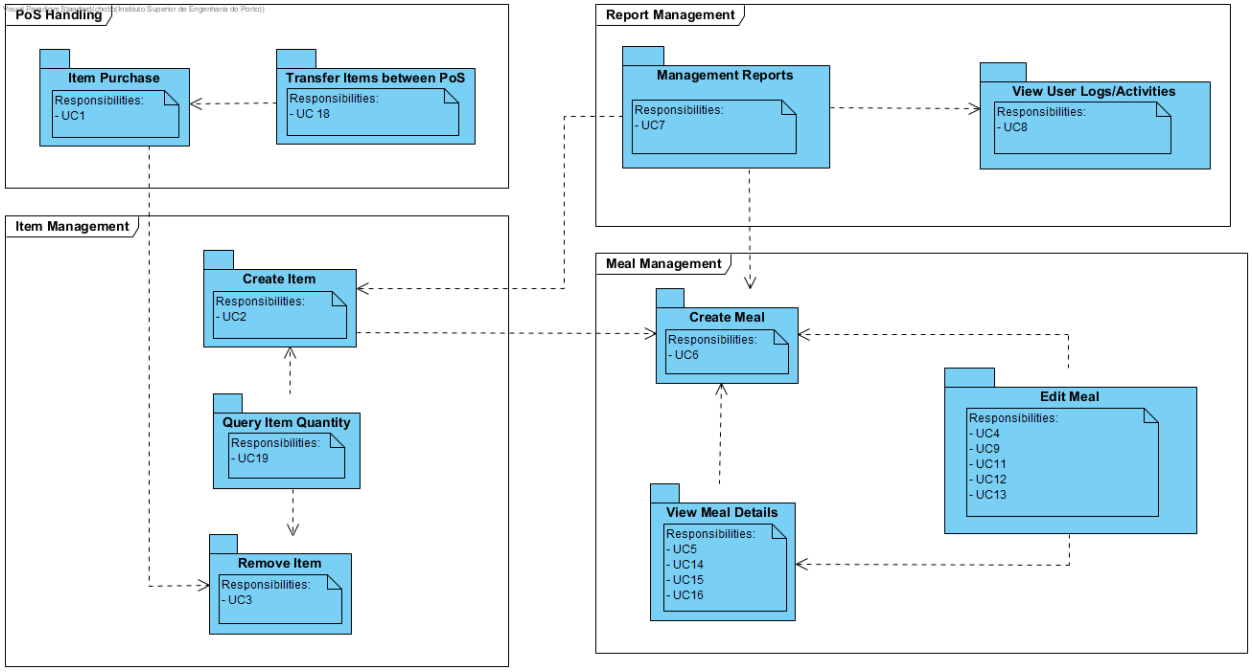

Figure 12 - Domain Objects as a UML diagram

Research Topic

Introduction

In this research topic it will be presented a chosen tool which can support the monolithic decomposition into microservices as it is the focus in this project. The group decided to choose a tool named Service Cutter, created by Lukas Kölbener and Michael Gysel, which analyzes a system´s specification and suggests candidate service cuts in order to optimize loose coupling between services and high cohesion within services, by utilizing an entity-relation model and use cases.[1]

Importing Data and Concepts

The tools needs a specific user representations to define the domain model and use cases of the system so the decomposition can be done in base of these artifacts, which used to feed criteria information into Service Cutter.[1] As mentioned before this tool needs a entity-relation model which use a user representation of the domain model which consists of data fields imported as nanoentities (which are resources of a service) [2], entities and relationships. In order to import this data to Service Cutter the user must generate a .json file by following a schema that can represent the domain model and the relations between the entities.

By importing the data represented in the mentioned file it is possible to Service Cutter decompose the system specifications in microservices though it cannot link or represent the services dependencies which are important but with the another .json file with the representation of the functionalities as use cases these dependencies and connections between services can be represented.

Algorithms

For the service decomposition algorithms Service Cutter offers two choices:

1. Girvan-Newman: A algorithm named after Michelle Girvan and Mark Newman[3], which detects communities (a network which its nodes can be easily grouped into sets of nodes that are densely connected internally[4]) by progressively removing edges from the original network. The algorithm then reconstructs the network by focusing on edges that are located between communities.
2. “Epidemic Label Propagation” by Leung: A algorithm that mimics epidemic contagion by spreading labels [5] that uses the network structure alone as its guide and requires neither optimization of a pre-defined objective function nor information about the communities [6].

Results & Assessment

As mentioned before, the group produced the required files to represent the systems specifications (domain model and uses cases/functionalities) and generated a decomposition diagram with the result given by both algorithms described in the last section.

As illustrated in Figure 13, that shows the result after specifying in Service Cutter to use the Girvan-Newman algorithm and that was intended to have 5 microservices, which almost looks identical to the result of decomposition by bounded context but with some slight problems. To start, having just one

microservices responsible for the MealID (***Service A***) is unnecessary because it already exists ***Service B*** which represents the bounded context *Meal Management* therefore it's a waste of a resource and doesn´t comply to the Single Responsibility Principle that states that every module, function or class should be responsible by just a single part of the functionality provided by the software [7], in other words it means the result is great but not optimal and doesn't meet the group desires.

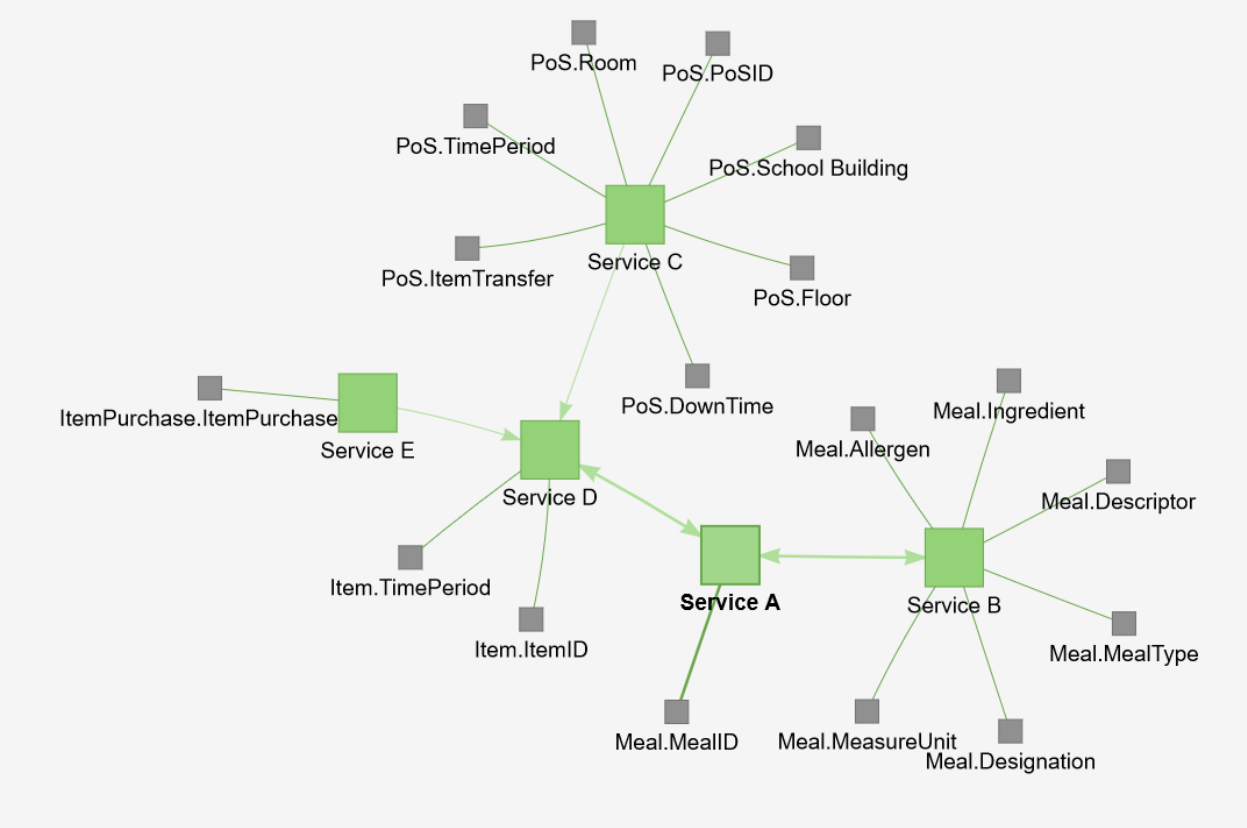

Figure 13- Result from Girvan-Newman algorithm with 5 services

By specifying that we want to use Service Cutter and the algorithm designed by Leung, as mentioned before the "Epidemic Label Propagation", which unlike the Girvan-Newman algorithm doesn't accept the parameter to choose how many services the user wants and so it generated the result shown in Figure 14. This algorithm chooses to cut the system into only two microservices, these two services being ***Service B*** that is responsible for handling with all data from the Meal and represents well the Business Context identified as *Meal Management* although the second service designed as ***Service A*** has the other nanoentities and this result on a oversized microservice and that's not what we want. The microservices should be created in focus with business capabilities of the knowledge that the domain gives and because there is a low number of dependencies within the ***Service A*** then it shouldn't be a single service.[8]

After either of these results show a good suggestion for the decomposition of the services in regard of the domain model created by the group therefore these presented solutions were discarded and it was decided to follow one decomposition pattern to define the microservices for the new system.

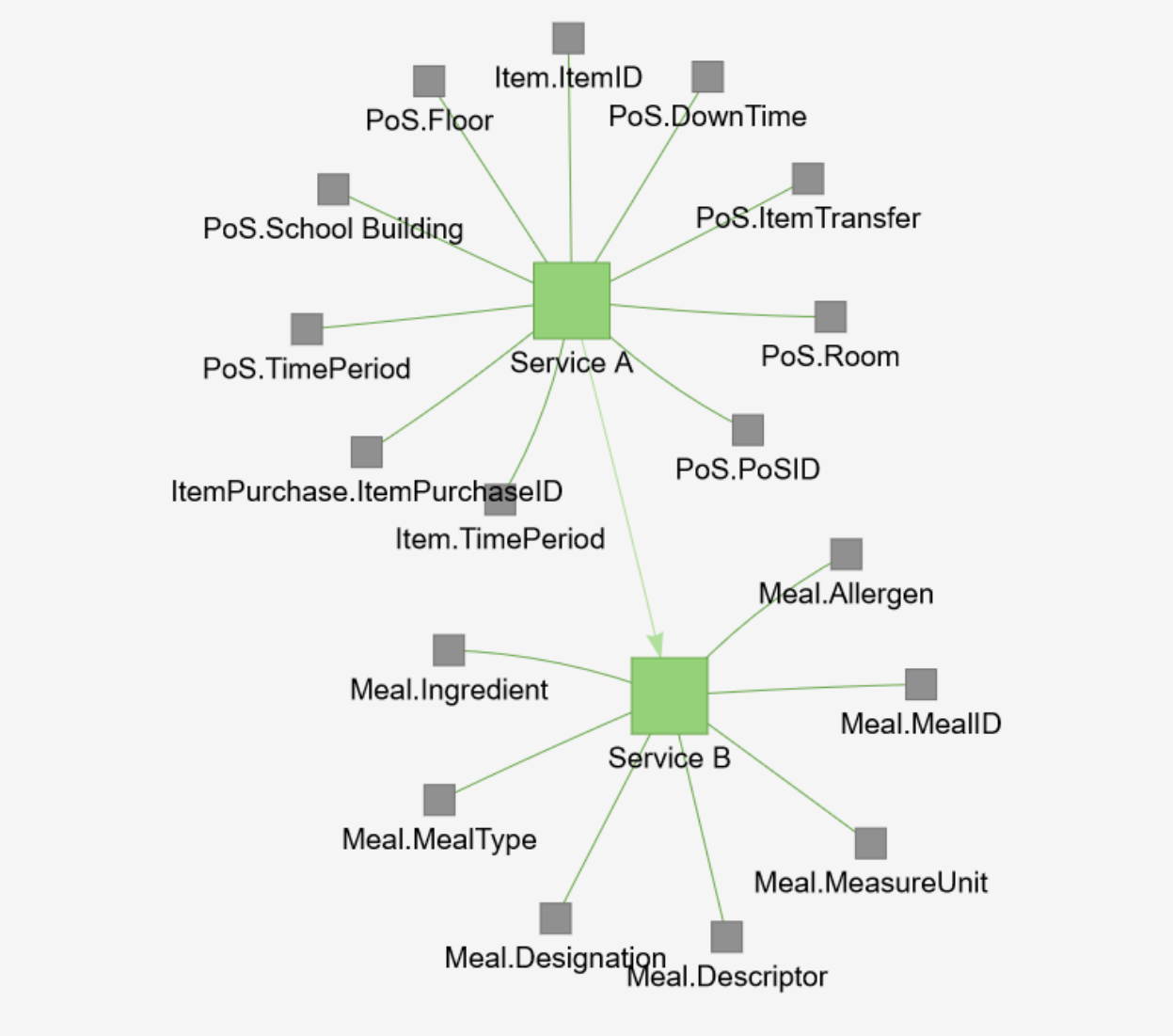

Figure 14- Generated Service Composition with Leung´s algorithm

References

[1] K. Lukas and G. Michael, 'Service Cutter', Bachelor, University of Applied Sciences of Eastern Switzerland (HSR FHO), Department of Computer Science in Rapperswil, 2015.

[2] 'Nanoentity - SC', *Concepts*, 16-Dec-2015. [Online]. Available: https://github.com/ServiceCutter/ServiceCutter/wiki/Concepts#nanoentity. [Accessed: 15-Dec-2019].

[3] M. Girvan and M. E. J. Newman, 'Community structure in social and biological networks', *Proc. Natl. Acad. Sci.*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.

[4] J.-R. Xie, P. Zhang, H.-F. Zhang, and B.-H. Wang, 'Completeness of Community Structure in Networks', *Sci. Rep.*, vol. 7, no. 1, pp. 1–6, Jul. 2017.

[5] S. E. Garza and S. E. Schaeffer, 'Community detection with the Label Propagation Algorithm: A survey', *Phys. Stat. Mech. Its Appl.*, vol. 534, p. 122058, Nov. 2019.

[6] U. N. Raghavan, R. Albert, and S. Kumara, 'Near linear time algorithm to detect community structures in large-scale networks', *Phys. Rev. E*, vol. 76, no. 3, p. 036106, Sep. 2007.

[7] R. C. Martin, J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2003.

[8] nishanil, 'Identifying domain-model boundaries for each microservice'. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/identify-microservice-domain-model-boundaries. [Accessed: 15-Dec-2019].