University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Sorting Hat

A Tool to Characterize the Architecture of Service-Based Systems

Erick Rodrigues de Santana

Final Essay

mac 499 — Capstone Project

Supervisor:	Prof. Dr. Alfredo Goldman vel Lejbman
Co-supervisor:	Prof. Dr. André van der Hoek (UC Irvine)
Co-supervisor:	MsC. João Francisco Lino Daniel (Unibz)

During the development of this work, the author will be receiving financial support from FAPESP – grant #2020/16577-9 and #2021/14240-0.

São Paulo
2022

The content of this work is published under the CC BY 4.0 license
(Creative Commons Attribution 4.0 International License)

Acknowledgments

I would like to thank my supervisors for guiding me to produce this amazing work in a field I love. The experiences and knowledge gained throughout the year mean so much to me. I would also like to thank my colleagues and friends for helping me during my time as a Computer Science student. Lastly, I thank my parents for always believing in me.

Resumo

Erick Rodrigues de Santana. **Sorting Hat:** ***Uma Ferramenta para Caracterizar a Arquitetura de Sistemas Baseados em Serviços***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Desenvolver grandes sistemas de software não é trivial, pois esses sistemas devem satisfazer atributos de qualidade como escalabilidade e manutenibilidade. Portanto, é importante que tenham uma arquitetura que favoreça o cumprimento desses atributos. Compreender a arquitetura de um software, conhecendo seus aspectos estruturais e padrões, é essencial para poder satisfazer os requisitos de qualidade desejáveis. No entanto, há pouco suporte para caracterizar e avaliar a arquitetura de sistemas, especialmente aqueles com arquitetura baseada em serviços. O Sorting Hat é uma ferramenta em desenvolvimento que auxilia no processo de caracterização da arquitetura de sistemas baseados em serviços.

Desde o início de 2021, o aluno trabalha no desenvolvimento dessa ferramenta. O desenvolvimento do Sorting Hat segue em duas direções complementares: a visualização e a coleta de dados. Durante o ano de 2021, o aluno desenvolveu um MVP (Protótipo Mínimo Viável) para a visualização. No entanto, a visualização teve alguns problemas relacionados à usabilidade e experiência do usuário, com muitas trocas de contexto devido ao grande número de pontos de vista e fluxo de navegação profundo. Além disso, as métricas que a ferramenta implementou não estavam atualizadas com o estado atual do modelo de suporte, CharM. Esta monografia final detalha a implementação da nova versão da visualização do Sorting Hat e da coleta de dados automatizada.

Palavras-chave: Arquitetura de Software. Microsserviços. Caracterização de Arquitetura. Coleta de Dados.

Abstract

Erick Rodrigues de Santana. **Sorting Hat: *A Tool to Characterize the Architecture of Service-Based Systems***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Developing large software systems is not trivial, as these systems must satisfy quality attributes such as scalability and maintainability. Therefore, it is important that they have an architecture that favors the fulfillment of these attributes. Understanding the architecture of a software, knowing its structural aspects and patterns, is essential to be able to satisfy the desirable quality requirements. However, there is little support for characterizing and evaluating systems architecture, especially those with service-based architecture. The Sorting Hat is a tool under development that assists in the process of characterizing the architecture of service-based systems.

Since the beginning of 2021, I have been working on the development of this tool. The development of the Sorting Hat goes in two complementary directions: the visualization and the data collection. During the year of 2021, the student developed a MVP (Minimum Viable Prototype) for the visualization. However, the visualization had some issues related to its usability and user experience, with many context switching due to the large number of viewpoints and deep navigation flow. Also, the metrics the tool had implemented were not up-to-date with the current state of the supporting model, CharM. This final monograph details the implementation of the new version of the Sorting Hat's visualization and the automated data collection.

Keywords: Software Architecture. Microservices. Architecture Characterization. Data Collection.

List of Figures

Contents

Introduction

Software architecture is an area that studies important concepts and practices to ensure the quality and success of a software system, hence improving its chances to succeed as project. A good architecture allows you to have a broad view on a software system and its future evolution, and to estimate the infrastructure costs and delivery times. It also influences the team organization and is the basis of system organization (Bass *et al.*, 2013). However, a software developed without a good architectural plan might be hard to keep up during its evolution. A poor architecture is a major cause of impediment for the developers to understand the software (Fowler, 2019).

Even with an architecture plan, it might be a challenge to architects and engineers to keep the system in accordance with it. Throughout the development process, architectural deviations can happen, that is, decisions during the implementation that hurt the planned architecture (Perry and Wolf, 1992). Therefore, it is valuable to keep the backlog of the current architecture and its evolutions, because it allows the architects to identify those deviations and to make the right decisions.

There are architectural styles that can guide an architecture. The microservice architectural style has been used frequently in the context of software development. Although this style has some benefits, there are challenges faced in its implementation. Those challenges are strongly linked to the complexity of microservice-based applications (Soldani *et al.*, 2018). As examples, there is the difficulty to determine the microservices' granularity and the distributed storage management (Soldani *et al.*, 2018). In that scenario, the deviations can be potentiated, in a way that characterizing the current architecture of a system can be difficult and expensive to the development team.

In order to help software architects in the process of decision-making in favor of evolution of their systems, it is being developed the Sorting Hat – a tool to characterize the architecture of service-based systems. In the MVP (minimum viable prototype), it displays the system's components, their characteristics and the relationship between them. It also displays some metrics based on the characterization model called CharM, which is under development by Rosa *et al.* (2020). The data available on Sorting Hat was collected manually, which was a strenuous and a computationally automatable process.

The main goals of this work for the Final Capstone are two. The first one is to automate the data collection of Sorting Hat, that can also apply the metrics of the model under development by Rosa *et al.* (2020) to data collected. The second one is to improve the existing visualization tool and update its information to the most recent metrics in the CharM model.

This work is part of a undergraduate research sponsored by FAPESP that has been ongoing since February 2021. From April 2022 to July 2022, I had been in an international internship at the Department of Informatics of the University of California, Irvine (UCI) advised by the Professor Dr. André van der Hoek, who has expertise in Software Engineering. Therefore, the first half of the work on this proposal was developed as part of the research internship.

The rest of this monograph is structured as follows: the Chapter 1 presents a conceptual base for this project. The Chapter 2 shows the tool first MVP, presenting its structural aspects. The Chapter 3 presents the goals of this project, as well as the methodology adopted to achieve them. The Chapter 4 shows the results achieved during the year. The Chapter 5 describes the personal experiences gained during the year. Finally, the Chapter 6 describes the conclusion and future work.

Chapter 1

Literature Review

Software architecture is a set of structures needed to understand and reason about a system, covering software elements, the relationship among them and their properties Bass *et al.*, 2013. It is also a discipline within software engineering that serves as a bridge between software requirements – functional and non-functional – and the software implementation. The development guided by a planned architecture is easier because it facilitates the understanding, maintenance, and evolution of a system.

An important role to be played by development team members is that of software architect. They are responsible for knowing the business domain, understanding the development process, knowing technologies and methodologies, as well as having good abilities with programming. That knowledge is essential, because the software architect is also responsible for making decisions in uncertain contexts that affect the structure and quality of a system.

To guide the software architects and engineers in the conception of software elements and their interactions, it is common the adoption of architectural styles.

> *"An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined."* Garlan and Shaw, 1994.

The microservice architectural style is described as an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, preferably asynchronous Fowler and Lewis, 2014. According to Newman, 2015, some main characteristics of that architectural style are the following:

- Small services and limited by business domain;
- Low coupling among services;
- Fault tolerant services;
- Technological heterogeneity;

- Automated deploy infrastructure.

However, developing systems following that architectural style is not trivial. Some questions raised by Newman, 2015 are:

- How to define the size and the responsibilities of each service?
- How to ensure the data consistency among services?
- How to keep a low level of coupling among services?

Those questions can lead to uncertainties when implementing the system. In this sense, architectural deviation can occur, that is, inconsistencies between the implemented software and the planned architecture Perry and Wolf, 1992. To restore the planned architecture, there is the concept of architectural recovery, which consists in using reverse engineer techniques to extract the implemented architecture from code artifacts Silva and Balasubramaniam, 2012.

To evaluate the current architecture of a microservice system, there is a characterization model called CharM Rosa *et al.*, 2020. That model, which is in constant evolution, currently has its core around four dimensions, each one being a composition of a set of metrics. The dimensions and their metrics are related to system components. A component can be either a module or a service. A module is defined as a deployment unit, i.e., a set of services that are deployed together. A service is defined as an application with cohesive and well-defined responsibility, that implements functionalities related to business tasks.

The CharM dimensions and its metrics are as follows:

- **Size**: The goal is to compare the organization of different components of a system. It has the following metrics:
  - Number of system's components;
  - Number of services per module;
  - Number of operations per component;
  - Number of services with deployment dependency.
- **Data source coupling**: The goal is to characterize the data source sharing strategy between the components of a system. It has the following metrics:
  - Number of system's data sources;
  - Number of data sources per component;
  - Number of data sources that each component shares with others;
  - Number of data sources where each component performs write-only operation;
  - Number of data sources where each component performs read-only operation;
  - Number of data sources where each component performs read and write operations.

- **Synchronous coupling**: The goal is to characterize the components synchronous interactions of a system. It has the following metrics:
  - Number of clients that invoke the operations of a given component;
  - Number of components from which a given component invokes operations;
  - Number of different operations invoked by each depending component;
  - Number of different operations invoked from other components.
- **Asynchronous coupling**: The goal is to characterize the components asynchronous interactions of a system. It has the following metrics:
  - Number of clients that consume messages published by a given component;
  - Number of components from which a given component consumes messages;
  - Number of different types of messages consumed by each depending component;
  - Number of different types of messages consumed from other components;
  - Number of components that consume messages from the queue;
  - Number of components that publish messages in the queue.

Chapter 2

Sorting Hat First MVP

In order to help software architects in the process of decision-making in favor of evolution of their systems, it is being developed the Sorting Hat – a tool to characterize the architecture of service-based systems, which displays metrics of the characterization model called CharM under development by Rosa *et al.* (2020).

The Sorting Hat's development is guided by evolutionary prototypes. The first MVP was the frontend focused on displaying a system's metrics.

The first MVP had only data from InterSCity – a platform following the microservice architectural style that assists smart-cities applications. The data were manually collected in a case study to validate the characterization model of Rosa *et al.*, 2020. This enabled to develop the frontend without having the automated data collector.

That frontend was composed by 3 levels of visualization: a system, a module of the system, and a service of the module. There was a page for each one of these levels. The pages of a system and a module have two views: the graph-based view, which showed the synchronous and asynchronous communications between the modules or services; and the metrics list, which showed all the metrics that belonged to that level. Figures 2.1 and 2.2 present these views.

It is important to note that, although it did not automatically collect data from systems, the views generated by the frontend could already be pertinent to the architects in future decisions. The graph visualization made explicit the synchronous connections. By observing this, the architects can make decisions to keep them or to migrate them to asynchronous, which are more consistent with the architectural style of microservices. Synchronous communications can generate dependency between services at runtime and increase coupling between them.

Viewing the list of metrics could also assist architects in making of decisions. For example, Figure 2.2 illustrates the metric of modules sharing database, which says the number of modules that share the same database. Sharing databases can generate inconsistencies, going against the adopted practices in the architectural style in question. Thus, if Figure 2.2 showed modules sharing databases, architects could decide to remove these shares or to keep them if necessary.

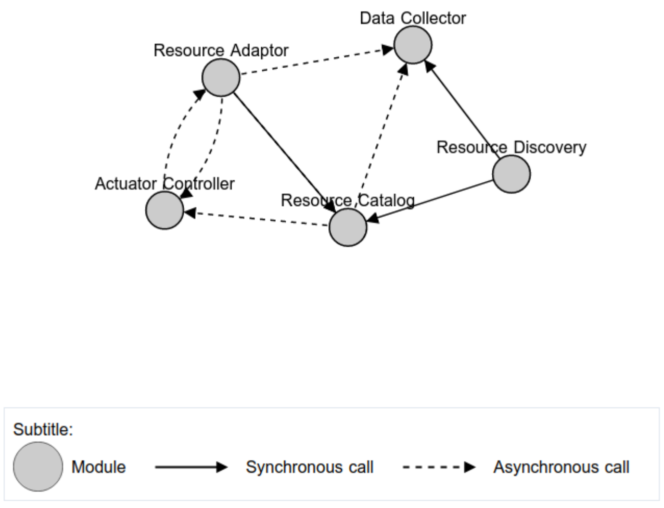


Figure 2.1: *Graph-based view of InterSCity's modules.*

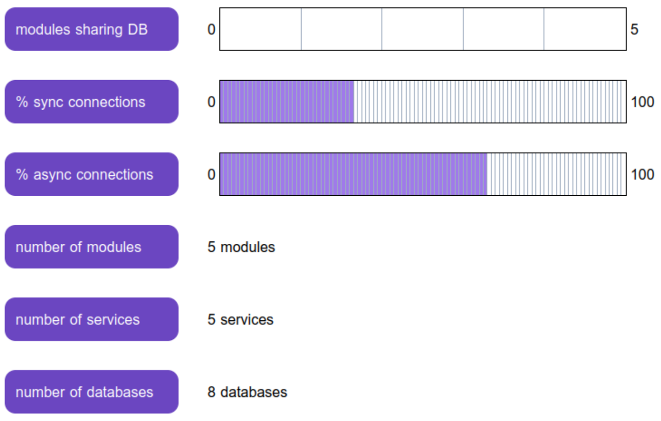


Figure 2.2: *Metrics list view of InterSCity.*

The first MVP architecture is represented in Figure 2.3. It illustrates an overview of the page structure, showing the user's navigation flow. Figure 2.3 also presents the components that the pages use, as well as the interaction with external elements, such as the backend, responsible for obtaining the data from the systems available on the platform.

To develop it, we used Nuxt.js, a framework for the construction of SPAs (single-page applications), a modern way of developing websites in which navigation between pages is made in a more fluid way, without the need for the browser to perform a new request to the server where the site is hosted with every page change. The construction of SPAs is also based on components, a part of the application that is common to several pages and therefore can be reused. Also, Nuxt.js allows the developer to use plugins – libraries or modules external to the application and that are made available to it.

In Figure 2.3, the three pages mentioned above are represented, which show the different viewing levels. The pages of a system and a module use the graph and metric components. For the graph visualization, we used the D3.js library as a plugin, which allowed the construction of customized graphs from the raw data that the application intends to display.

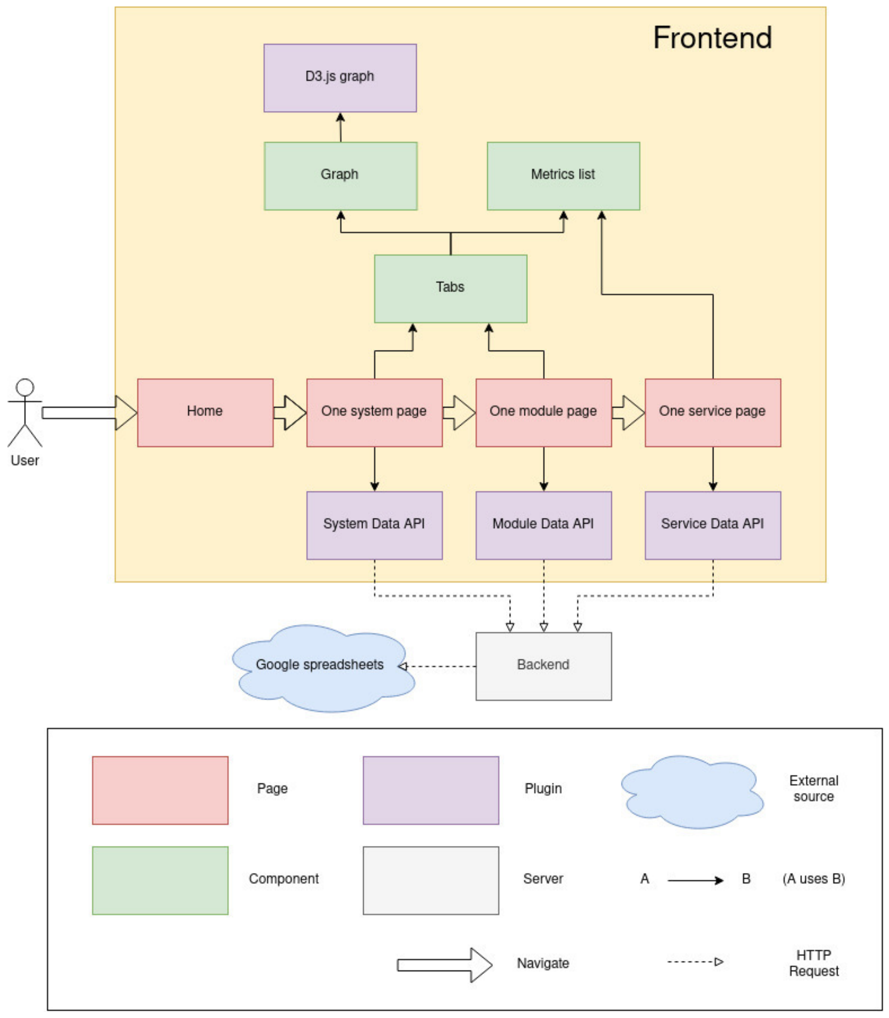


Figure 2.3: *Sorting Hat's frontend architecture.*

In this MVP, InterSCity data were available in Google spreadsheets. For the platform to get them, some abstractions were created: each of the three pages uses a plugin to request the data from the backend and transform it to the way the pages understand it. In this way, each of the plugins acts as an abstraction from the server. The backend has a similar goal: it requests data from Google spreadsheets and transforms it for objects that are understood by each of the plugins. All these abstractions were created with the objective of modularizing the application, providing testability gains, decoupling and scalability.

Chapter 3

Goals and Methodology

The data collection for the Sorting Hat was done manually, making that process harder. Also, there were some issues related to the frontend: its usability was a bit complicated, with many contexts switching, and the metrics of the model under development by Rosa *et al.* (2020) were not up to date. The goal of this work went in two different yet complementary directions: automate the data collection of Sorting Hat; and make the frontend more intuitive, easy to use and fluid to work with, as well as update its information to the latest metrics from the CharM model.

To achieve the goals of this work, we followed the methodology described in this chapter, illustrated by the Figure 3.1.

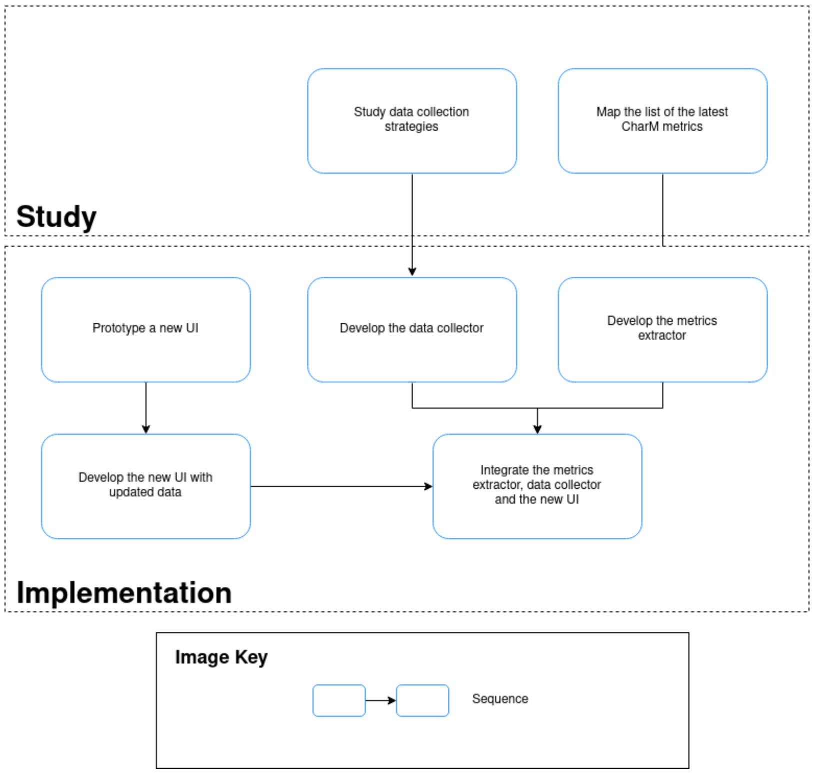

Study data collection strategies
Map the list of the latest CharM metrics
Study
Prototype a new UI
Develop the data collector
Develop the metrics extractor
Develop the new UI with updated data
Integrate the metrics extractor, data collector and the new UI
Implementation
Image Key
Sequence

Figure 3.1: *Methodology of the final work*

In the study stage, we started studying the CharM model and its characteristics to see the data we need to get in order to extract the information required by the model. We noticed that we can measure the CharM characteristics using structural information that can be static, which means that they do not change with the system in execution and, thus, it is not necessary to have the system running to extract those metric. For instance, CharM metrics such as "Number of operations per service", "Number of data sources", and "Number of data sources that each service share with others" are both structural information that can be collected using a static approach. These information depend on discovering the services of a system, their operations, the data sources of a system, and the usages of data sources by services.

With that in mind, we started studying some data collection strategies in order to have the basis to develop a data collector. We have seen that there were two strategies to be adopted: static and runtime analysis. Static analysis means to analyze source-code, configuration files, and so on. On the other hand, runtime analysis is a way of analyzing the behavior of a system that involves executing it, monitoring it by a certain period and instrumenting it to extract data. For instance, it is possible to analyze the accesses to databases, HTTP requests, and so on. The fact that most of the CharM characteristics can be measured using structural information that can be static led us to choose for static analysis rather than runtime analysis.

After the study stage, we went to the implementation stage that gave us three contributions: the data collector, the metrics extractor and the improved visualization.

We developed a prototype for automated data collection. In that prototype, docker-compose and OpenAPI files were analyzed. Docker-compose files are useful because it is possible to know the services, databases and databases usages of a system by analyzing them. OpenAPI files are useful because they describe the API of a service. Those technologies were chosen because of the familiarity with them. It is important to highlight that the collector integrates with the data manually collected which are in use by the first MVP.

We also developed a metrics extractor, which is responsible for automatically applying the CharM metrics to each system in the platform, thus, characterizing the system according to the model. The results of the metrics extractor were compared to the metrics values of the InterSCity system, whose data were manually collected, to see whether the implementations were producing right results. This part was made during the period of the international internship supervised by the Prof. Dr. André van der Hoek.

To complete the implementation stage, we improved the Sorting Hat visualization. This part was also made during the international internship supervised by the Prof. Dr. André van der Hoek. First, we made a prototype in order to see if we improved the tool usability. The prototype was validated together with the supervisor. After prototyping, the next step was to develop the new frontend. In order to stress the view to validate the new frontend interface, we manually collected the data of a system called TrainTicket, a large benchmark based on microservice architecture which contains more than 40 microservices. The TrainTicket was recommended by a PhD student also advised by the Prof. André.

Finally, having the metrics extractor, the data collector and the new frontend implemented, the final next step to integrate these three parts.

Chapter 4

Results

The goals of this work were two: to automate the data collection of Sorting Hat; and to make the frontend more intuitive, easy to work with, as well as update its information to the latest metrics of the CharM model. The results of this work comprehend three contributions:

1. A data collector, responsible for collecting system data for the tool;
2. A metrics extractor, responsible for automatically extracting the latest metrics of the CharM model for every system in the tool;
3. An improved frontend, with a more interactive, elegant design that solves the usability issues.

The contributions 1 and 2 were implemented together in a component called backend, described in the Section 4.1. The contribution 3 is described in the Section 4.2.

4.1 Backend: Data Collector and Metrics Extractor

4.1.1 Data Collection Strategies

We defined a reduced scope for data collection in which we would collect a few data needed by the tool. The scope consisted of discovering the services of a system, their endpoints, the databases and the databases usages by services. With these data collected, it is possible to extract the metrics related to the Size and Data Coupling dimensions from the CharM model.

We developed a prototype for automated data collection with the defined scope. In this prototype, docker-compose files were statically analyzed. These files are used as a description of the containers that represent services and databases of a service-based system, and they are also often used in software development. The docker-compose files are obtained from a GitHub repository that contains a system data.

An example of a docker-compose file is illustrated in Figure 4.1. In it, there are four containers: `db`, `user_service`, `order_service` and `notification_service`. It is possible to note that there is a boot dependency between db and the other three through the property depends_on. Furthermore, user_service, order_service and notification_service have an environment variable MONGO_URL which represents a database connection with db. So, we may assume that user_service, order_service and notification_service probably use db at runtime. This hypothesis can be confirmed through static analysis of the source code of those services or through runtime analysis of it.

After analyzing the docker-compose represented in Figure 4.1, the collector prototype would return a service-based system as represented in Figure 4.2.

```
version: '3.7'

services:
  db:
    container_name: db
    image: mongo:4.2

  user_service:
    build: ./user_service
    environment:
      MONGO_URL: 'mongodb://db:27017'
    depends_on:
      - db

  order_service:
    build: ./order_service
    environment:
      MONGO_URL: 'mongodb://db:27017'
    depends_on:
      - db

  notification_service:
    build: ./notification_service
    environment:
      MONGO_URL: 'mongodb://db:27017'
    depends_on:
      - db
```

Figure 4.1: *Docker compose file example*

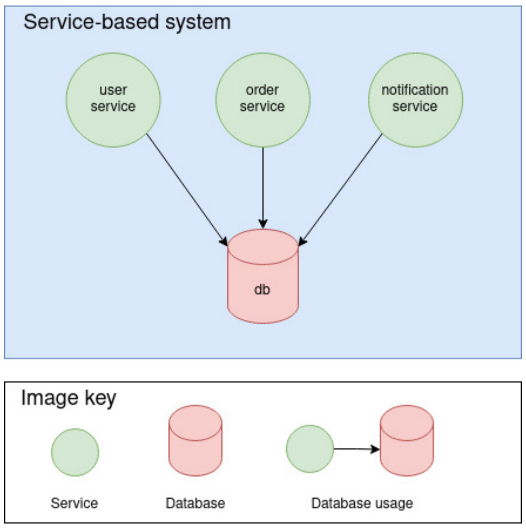
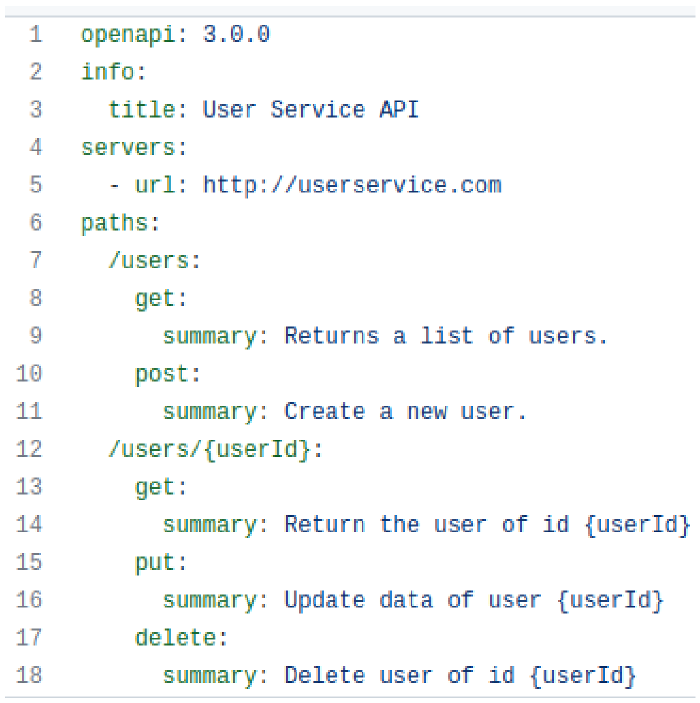


Figure 4.2: *Response of the data collector prototype*

Also, the collector analyzes OpenAPI files in order to collect the endpoints of each service in a service-based system. The OpenAPI Specification is a specification for a interface definition language for describing, producing, consuming and visualizing RESTful web services.

An example of a OpenAPI file is illustrated in Figure 4.3. In it, we have a list of endpoints described in the key `paths`, which has the `/users` and `/users/{userId}` paths. Each one of the paths has some HTTP Verbs (get, post, put or delete), making each combination of path and verb an endpoint. Thus, the file represents a service having five endpoints: `GET /users`, `POST /users`, `GET /users/{userId}`, `PUT /users/{userId}`, `DELETE /users/{userId}`.



```yaml
openapi: 3.0.0
info:
  title: User Service API
servers:
  - url: http://userservice.com
paths:
  /users:
    get:
      summary: Returns a list of users.
    post:
      summary: Create a new user.
  /users/{userId}:
    get:
      summary: Return the user of id {userId}
    put:
      summary: Update data of user {userId}
    delete:
      summary: Delete user of id {userId}
```

Figure 4.3: *OpenAPI file example*

4.1.2 Backend Architecture

An overview of the backend architecture is illustrated in Figure 4.4. Every time the backend is started, it fetches systems' data manually collected from a spreadsheet and saves those that have not yet been saved in the database. A client can access the content of the backend through its API, composed by the following endpoints:

- `GET /systems`: to get all systems registered in the tool.
- `GET /systems/{name}`: to get more detailed information of a specific system with name `name`.
- `GET /systems/{name}/metrics`: to extract and get the CharM metrics of a specific system with name `name`.
- `POST /systems`: to collect system data from a remote repository.

- `PUT /systems/{name}/endpoints`: to register services endpoints.

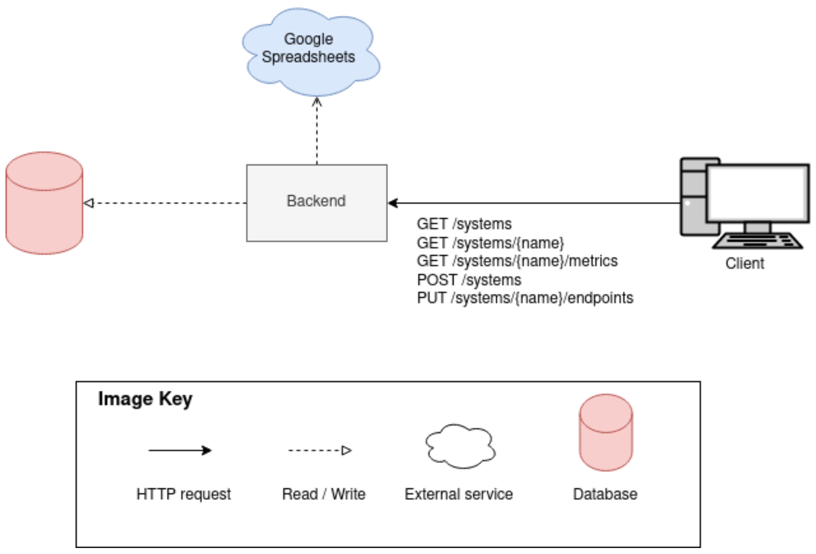


Figure 4.4: *Backend overview*

The backend packages structure is illustrated in Figure 4.5.

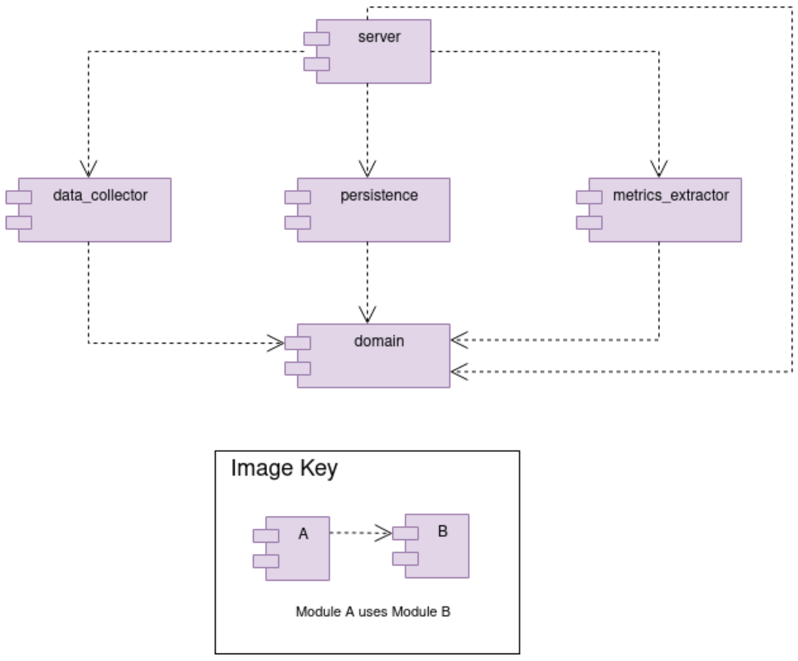


Figure 4.5: *Backend packages structure*

Internally, all the endpoints are handled by the server package, which uses the `metrics_extractor` and the `data_collector` modules. Both modules uses the `domain` module, which contain the model used to represent a service-based system in our tool. Also, there is the `persistence` module, which is responsible for saving and retrieving the system collected.

The domain entities model in the package `domain` is illustrated in the class diagram in Figure 4.7, where all main classes of a service-based system are represented. It is important to note that all class diagrams follows the color key represented in Figure 4.6.

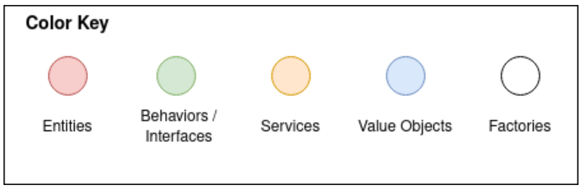


Figure 4.6: *Backend class diagram color key*

It is interesting to note that the `Service` class is the biggest entity, because it is the central entity of our model. It contains data such as exposed and consumed operations, used to identify synchronous communications between services, and channels publishing and subscribing, used to identify asynchronous communications. Other entities such as Module and Database are also represented in our model.

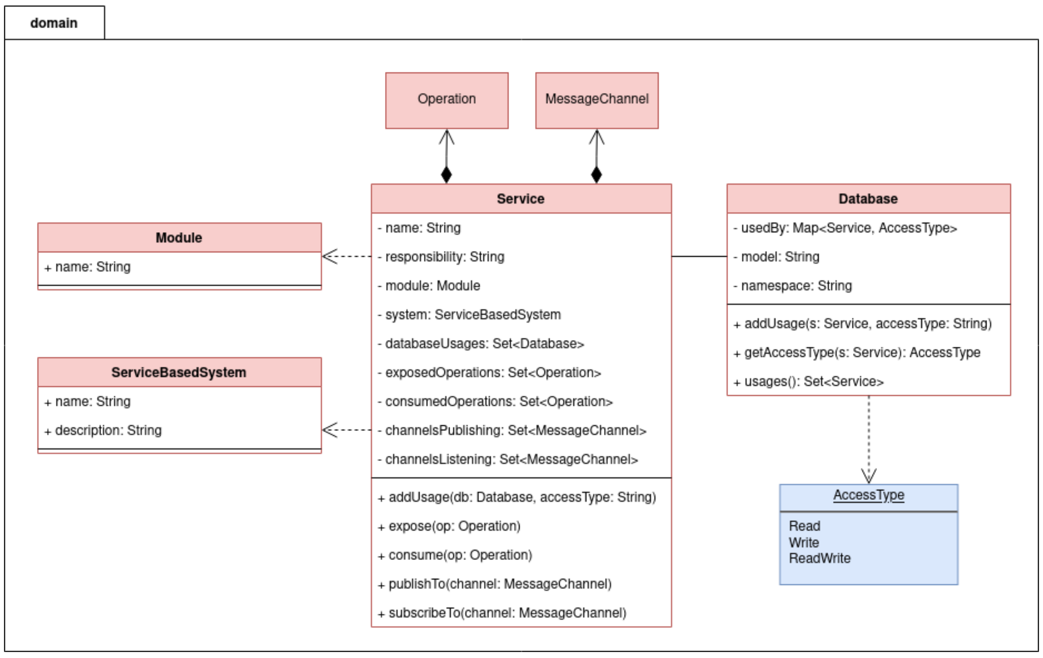

domain
Operation
MessageChannel
Service
- name: String
- responsibility: String
- module: Module
- system: ServiceBasedSystem
- databaseUsages: Set<Database>
- exposedOperations: Set<Operation>
- consumedOperations: Set<Operation>
- channelsPublishing: Set<MessageChannel>
- channelsListening: Set<MessageChannel>
+ addUsage(db: Database, accessType: String)
+ expose(op: Operation)
+ consume(op: Operation)
+ publishTo(channel: MessageChannel)
+ subscribeTo(channel: MessageChannel)
Module
+ name: String
ServiceBasedSystem
+ name: String
+ description: String
Database
- usedBy: Map<Service, AccessType>
- model: String
- namespace: String
+ addUsage(s: Service, accessType: String)
+ getAccessType(s: Service): AccessType
+ usages(): Set<Service>
AccessType
Read
Write
ReadWrite

Figure 4.7: *Domain entities class diagram*

Besides the entities, the domain has some behaviors, illustrated in Figure 4.8. It is interesting to note that the Domain uses the Visitor pattern to operate through the domain entities when the extraction is happening. This makes the code cleaner because the pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation. Also, the Domain has a `ServiceRepository` interface, which exposes the basic operations needed to perform data manipulations on the entity `Service` and its dependent classes.

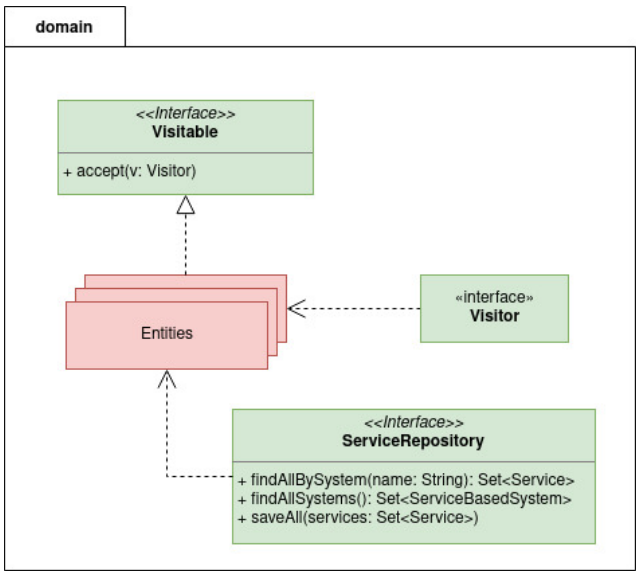

domain
<<Interface>>
Visitable
+ accept(v: Visitor)
Entities
«interface»
Visitor
<<Interface>>
ServiceRepository
+ findAllBySystem(name: String): Set<Service>
+ findAllSystems(): Set<ServiceBasedSystem>
+ saveAll(services: Set<Service>)

Figure 4.8: *Domain behaviors class diagram*

The class diagram of the `data_collector` module is represented in Figure 4.9.

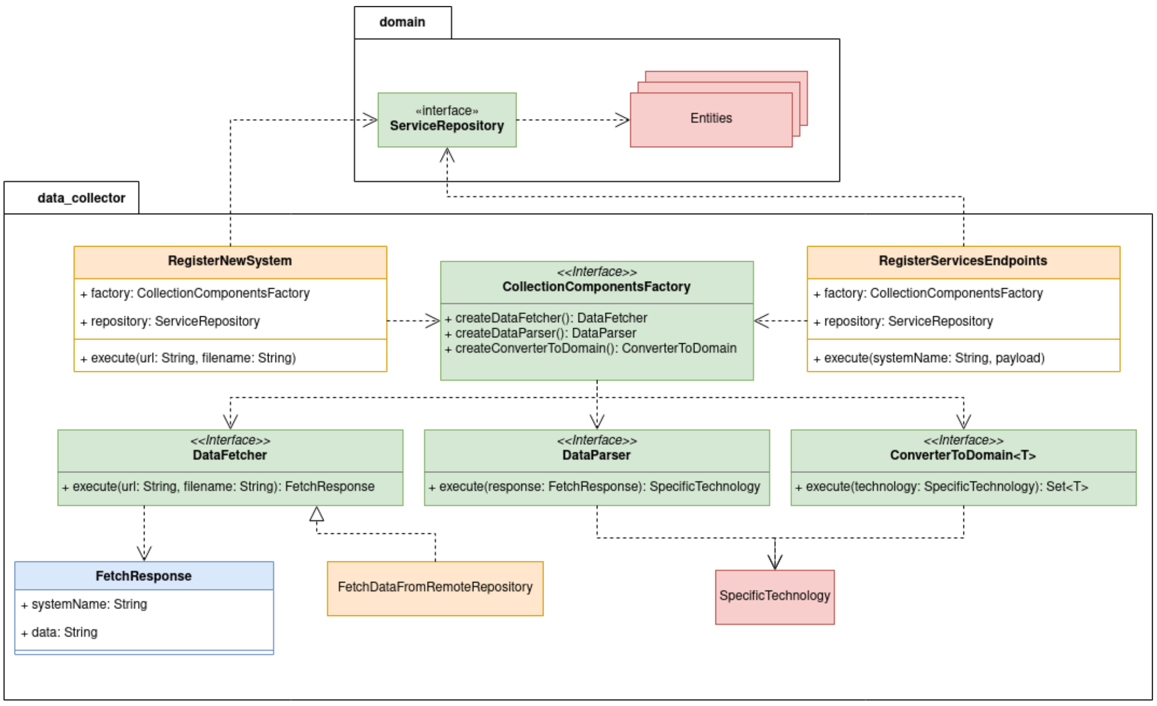

domain
«interface»
ServiceRepository
Entities
data_collector
RegisterNewSystem
+ factory: CollectionComponentsFactory
+ repository: ServiceRepository
+ execute(url: String, filename: String)
<<Interface>>
CollectionComponentsFactory
+ createDataFetcher(): DataFetcher
+ createDataParser(): DataParser
+ createConverterToDomain(): ConverterToDomain
RegisterServicesEndpoints
+ factory: CollectionComponentsFactory
+ repository: ServiceRepository
+ execute(systemName: String, payload)
<<Interface>>
DataFetcher
+ execute(url: String, filename: String): FetchResponse
<<Interface>>
DataParser
+ execute(response: FetchResponse): SpecificTechnology
<<Interface>>
ConverterToDomain<T>
+ execute(technology: SpecificTechnology): Set<T>
FetchResponse
+ systemName: String
+ data: String
FetchDataFromRemoteRepository
SpecificTechnology

Figure 4.9: *Data collector class diagram: main classes*

There are two important services in Figure 4.9. One is the `RegisterNewSystem`, which is responsible for registering a new system given an url and a filename. The other one is the `RegisterServicesEndpoints`, which is responsible for registering the endpoints of each service in a system. Both services use an abstract factory interface called `CollectionComponentsFactory`. The implementations of that interface create instances of three important objects to collect the data the collector needs:

- `DataFetcher`: an interface which its implementations are responsible for fetching the data from a remote source.
- `DataParser`: an interface which its implementations are responsible for parsing the data fetched.
- `ConverterToDomain`: an interface which its implementations are responsible for converting the data parsed to our domain model.

The purpose of these abstractions is to give extensibility to the tool to obtain data from several specific technologies. In our case, we have docker-compose and OpenAPI as our specific technologies, but these abstractions give us the power to add more technologies if we want to expand our data collection strategies.

The Figure 4.10 shows the implementations of those abstractions in the case of docker-compose. It is important to note that new entities were created to represent the docker-compose file. Also, there is a concrete factory that implements `CollectionComponents-Factory` called `DockerComposeCollectionComponentsFactory`, which create instances of the services `ParseDockerCompose` and `DockerComposeToDomain`, which are implementations of the `DataParser` and `ConverterToDomain` interfaces, respectively.

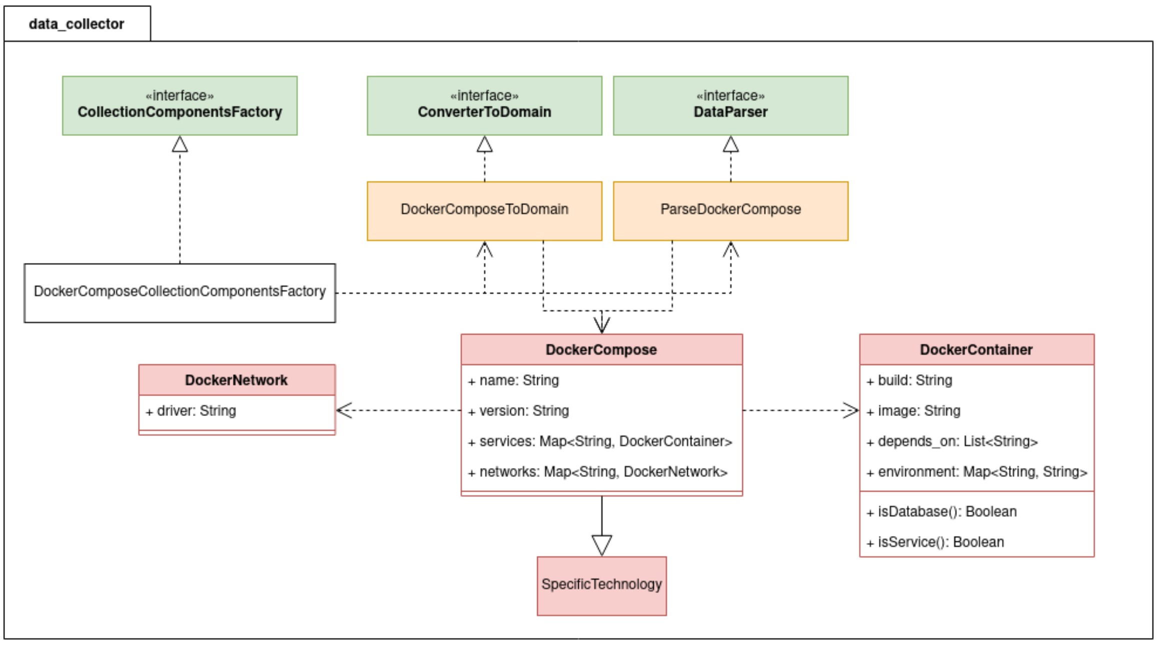


Figure 4.10: *Data collector class diagram: docker-compose*

The Figure 4.11 shows the implementations of the abstractions in the case of the OpenAPI. It follows the same structure as in the case of docker-compose: there is a model to represent a OpenAPI file, a concrete factory called `OpenApiCollectionComponentsFactory`, which implements `CollectionComponentsFactory` and is responsible for creating instances of the services `ParseOpenApi` and `OpenApiToDomain`, which are implementations of the `DataParser` and `ConverterToDomain` interfaces, respectively.

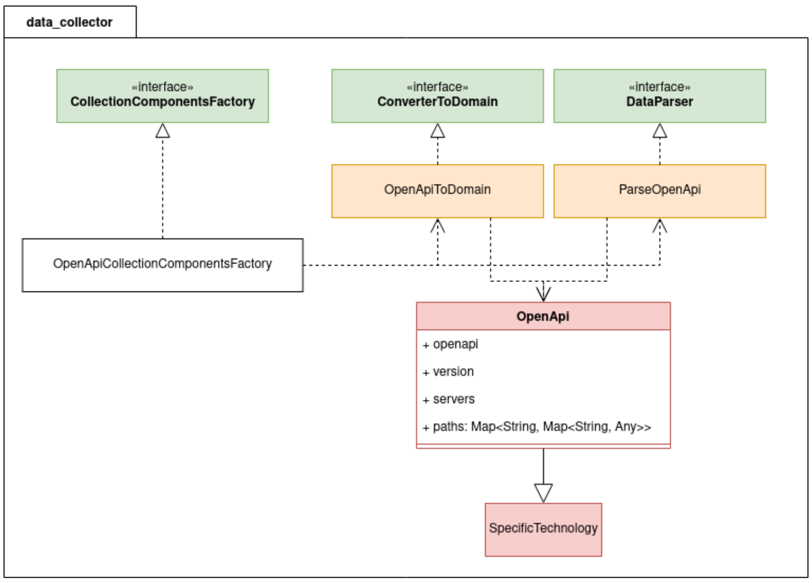


Figure 4.11: *Data collector class diagram: OpenAPI*

The `metrics_extractor` module in the backend is responsible for extracting all the CharM metrics for every system in the tool. The extraction results are available through the endpoint `GET /systems/{name}/metrics` in the backend server.

The Figure 4.12 illustrates the Metrics Extractor class diagram.

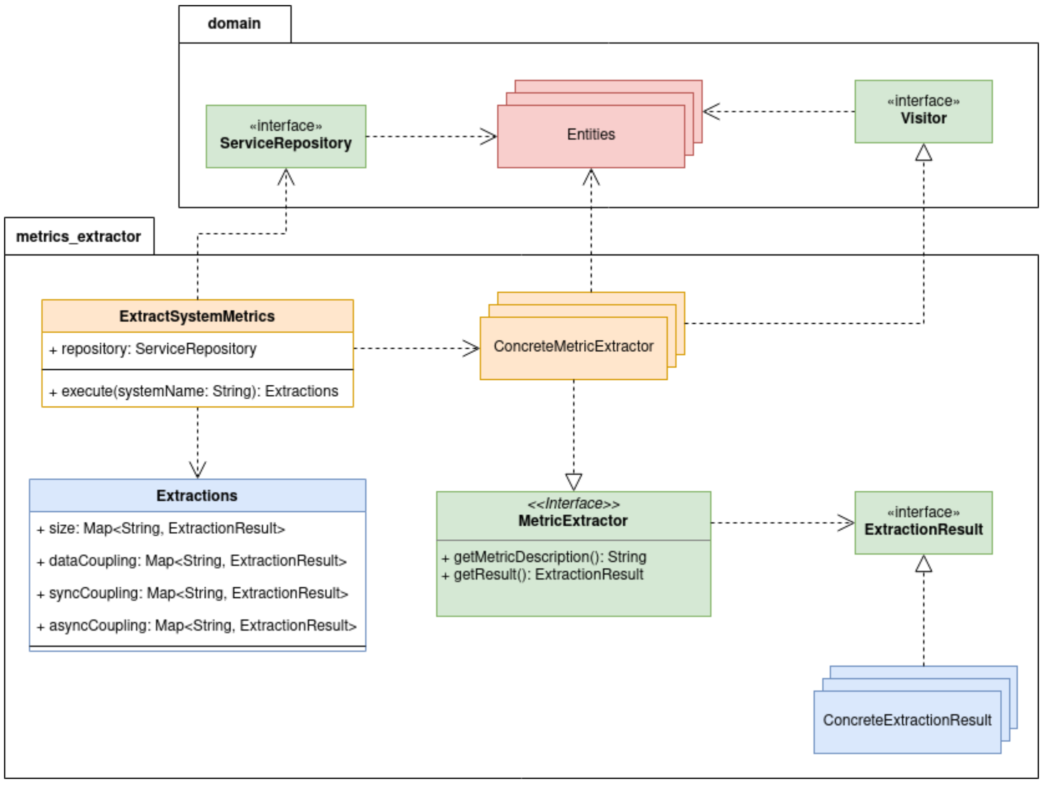

domain
«interface»
ServiceRepository
Entities
«interface»
Visitor
metrics_extractor
ExtractSystemMetrics
+ repository: ServiceRepository
+ execute(systemName: String): Extractions
ConcreteMetricExtractor
Extractions
+ size: Map<String, ExtractionResult>
+ dataCoupling: Map<String, ExtractionResult>
+ syncCoupling: Map<String, ExtractionResult>
+ asyncCoupling: Map<String, ExtractionResult>
<<Interface>>
MetricExtractor
+ getMetricDescription(): String
+ getResult(): ExtractionResult
«interface»
ExtractionResult
ConcreteExtractionResult

Figure 4.12: *Metrics extractor class diagram*

The `ExtractSystemMetrics` class is the entry point to execute the metrics extraction. When the `execute` method is called, all the `ConcreteMetricExtractor` classes are instantiated. A `ConcreteMetricExtractor` class is responsible for extracting one specific CharM metric. All of them implements the `MetricExtractor` interface, which provide methods for getting the metric description and the extraction result. Extraction results may differ for each `MetricExtractor` implementation, and because of this, `ConcreteExtractionResult` classes were created.

The backend is deployed in a Amazon EC2, which can be accessed through this link. It has almost 5300 lines of code written. To develop it, we used Kotlin and Spring as Web Framework. The code is open-source and can be accessed through this link.

4.2 New Frontend

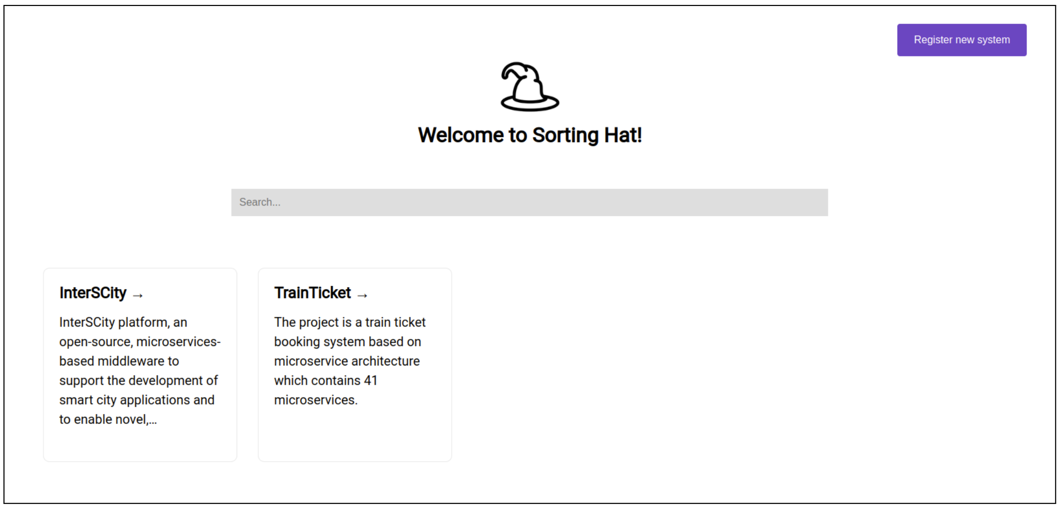
We noticed that the frontend had many context switching, which compromise the user experience by making the user perform a considerable number of clicks. For instance, we had to navigate through 3 links if we wanted to see the services and databases information of a system.

We would like that a single view had as much access to different information as possible, so we prototyped a new frontend in a tool called Figma. This tool allows us to create professional prototypes and is commonly used by designers. It also allows us to create interactive prototypes.

A screen of the prototype is illustrated in the Figure 4.16. It is possible to see that everything is now in a single view. The idea of this view is to allow the user to see all system elements (services, databases, synchronous and asynchronous calls) and its metrics. The user can complement the initial visualization by selecting the CharM dimensions on the left side of the screen. For instance, if the user click on the data source coupling dimension, the databases and usages by services will be shown in the visualization. The interactive version of the prototype is available at this link.

After making the prototype using Figma, we developed the new frontend version. A view of the home page is shown in the Figure 4.13. It is possible to view the architecture of a system by clicking on it in the systems list. It is also possible to register a new system in the tool.

The system registration page is shown in the Figure 4.14. To register a new system, the user needs to inform the remote repository URL and a docker-compose filename, from which the tool will identify the services, databases and databases usages. After clicking on the register button, the view is changed to support endpoints registration for each service identified, as shown in the example of Figure 4.15. In that page, the user can optionally inform the OpenAPI files for each service identified. After that, the user can finish the system registration by clicking on the register button.



InterSCity →

InterSCity platform, an open-source, microservices-based middleware to support the development of smart city applications and to enable novel,...

TrainTicket →

The project is a train ticket booking system based on microservice architecture which contains 41 microservices.

Figure 4.13: *New frontend: Home page*

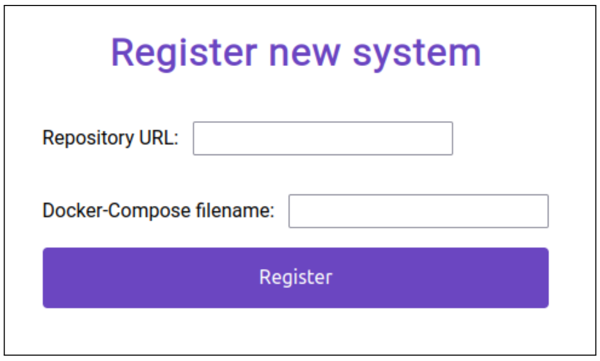


Figure 4.14: *New frontend: System registration page*

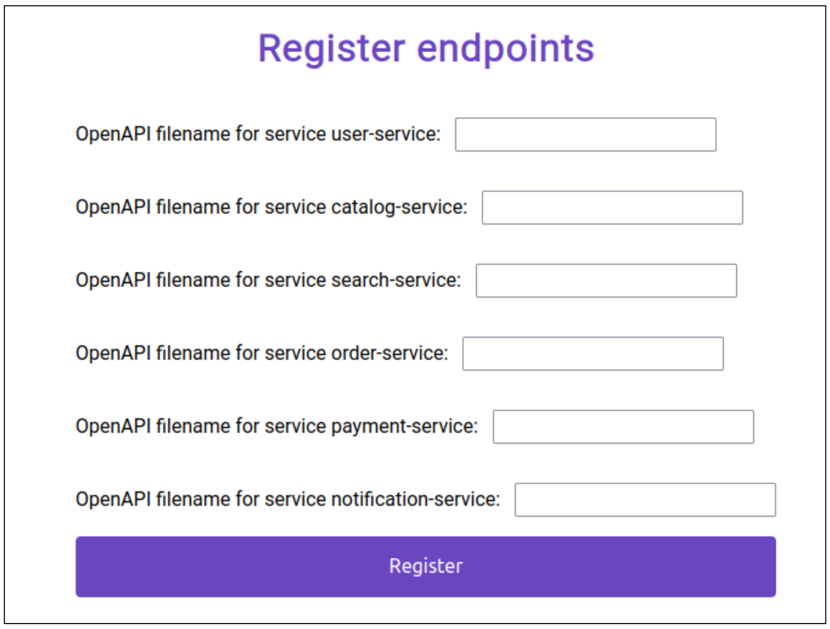
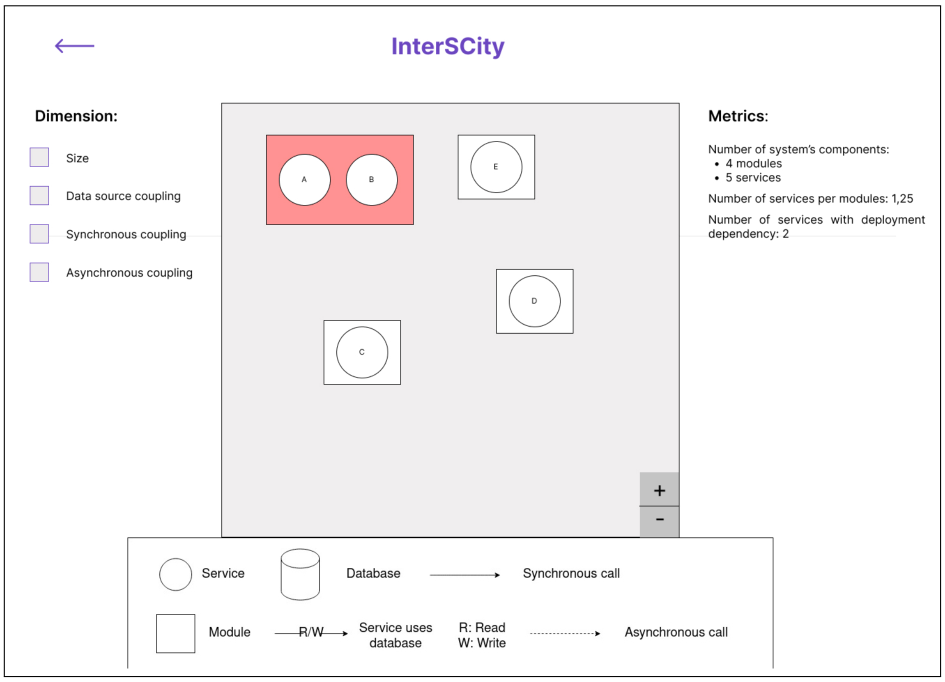

Register endpoints
OpenAPI filename for service user-service:
OpenAPI filename for service catalog-service:
OpenAPI filename for service search-service:
OpenAPI filename for service order-service:
OpenAPI filename for service payment-service:
OpenAPI filename for service notification-service:
Register

Figure 4.15: *New frontend: Endpoints registration page*

A view of the system page is shown in the Figure 4.17, which shows the page for a system called TrainTicket, a benchmark system based on microservice architecture which contains more than 40 microservices. The reason why we decided to add this system was to stress the view, i.e, to display a large amount of information, so we can validate the new frontend interface. The TrainTicket was the largest benchmark microservice system we found. Its data were collected manually from its documentation and put in the spreadsheet used as the data source of the Sorting Hat. The TrainTicket documentation is available at this link.



Metrics:

Number of system's components:
- 4 modules
- 5 services

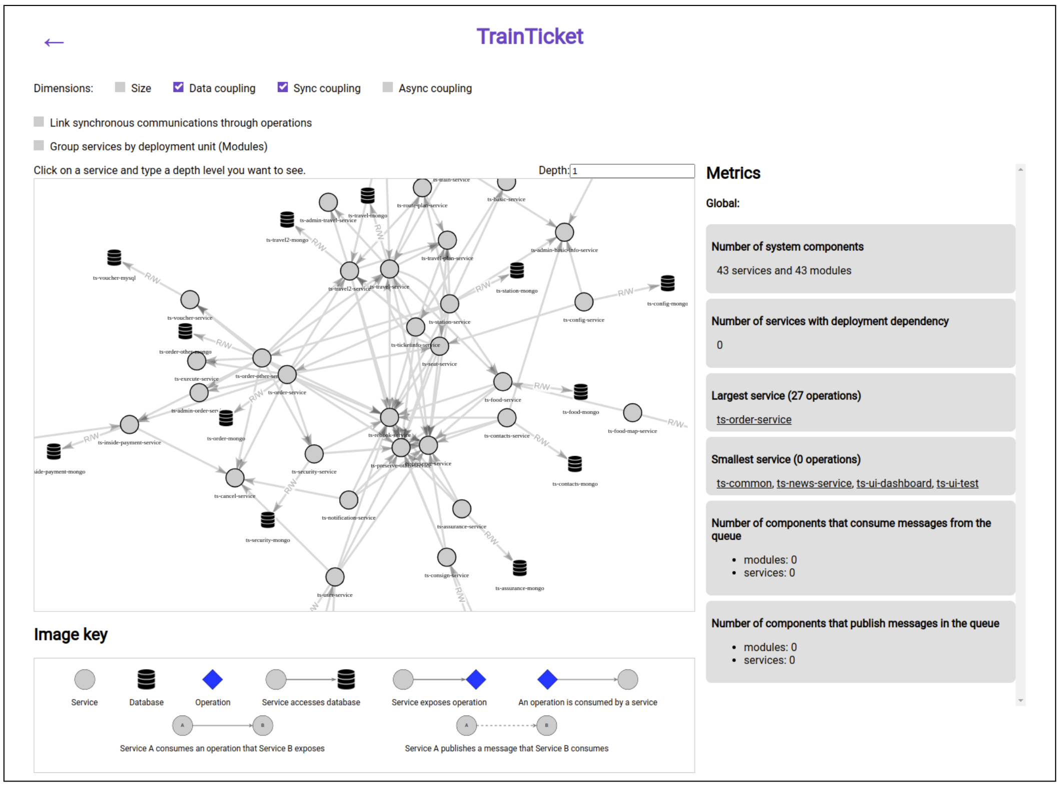
Number of services per modules: 1,25

Number of services with deployment dependency: 2

Figure 4.16: *New frontend: Prototype*

From Figure 4.17, it is possible to notice that there is a difference between the prototype version and the implemented version, as we decided to make some changes in the design and functionality as we were implementing. It is possible to see that there are no boxes representing modules in the visualization. Instead, we decided to add an option called "Group services by deployment unit (Modules)", which replaces the services in the visualization with modules, which is represented by the same object as services. We decided to make this change because it makes the view simpler, as it does not have the boxes coexisting with services.

It is also possible to see that there is an option called "Link synchronous communications through operations", which changes the default behavior of the sync coupling dimension – link services directly – to show the operation from which the given synchronous call is being made, as shown in the Figure 4.18. This can be a valuable information for the user to understand more the synchronous communications between services. The operations objects were not in the prototype version, but we decided to add it as we noticed that the Size dimension is about the amount of operations a system component has.



←

TrainTicket

Dimensions: ☐ Size ☑ Data coupling ☑ Sync coupling ☐ Async coupling

☐ Link synchronous communications through operations

☐ Group services by deployment unit (Modules)

Click on a service and type a depth level you want to see.

Depth: 1

Metrics

Global:

Number of system components

43 services and 43 modules

Number of services with deployment dependency

0

Largest service (27 operations)

ts-order-service

Smallest service (0 operations)

ts-common, ts-news-service, ts-ui-dashboard, ts-ui-test

Number of components that consume messages from the queue

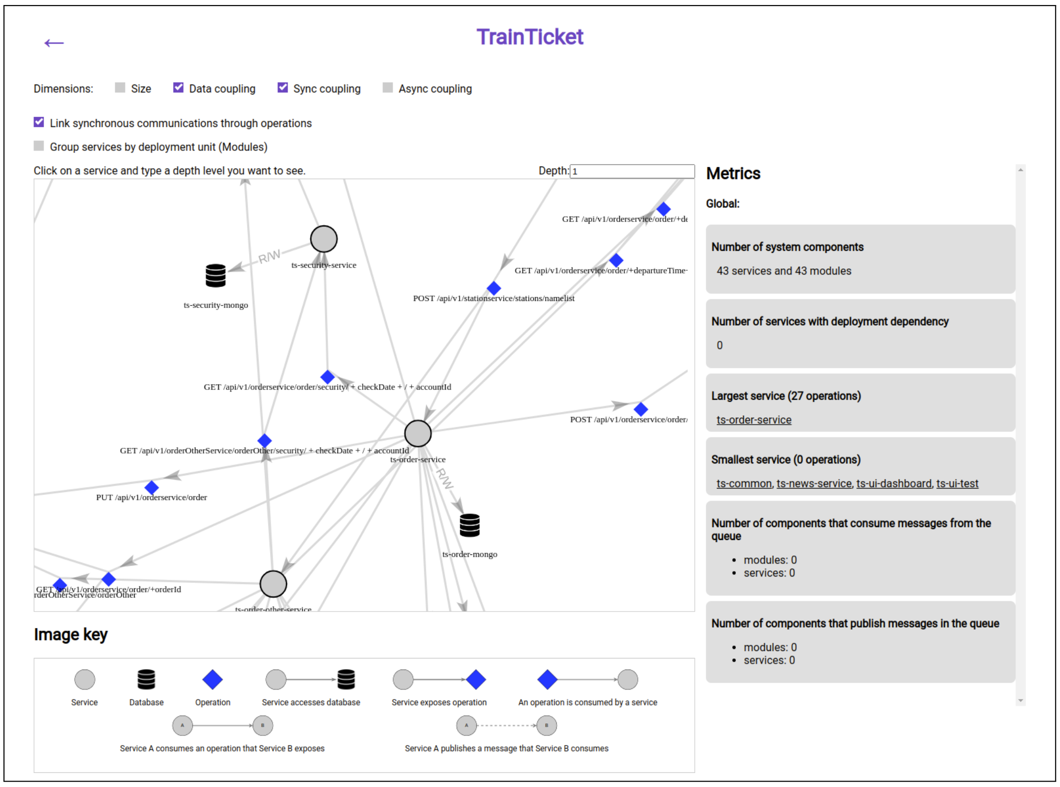
- modules: 0
- services: 0

Number of components that publish messages in the queue

- modules: 0
- services: 0

Image key

Figure 4.17: *New frontend: System page*



←

TrainTicket

Dimensions: ☐ Size ☑ Data coupling ☑ Sync coupling ☐ Async coupling

☑ Link synchronous communications through operations

☐ Group services by deployment unit (Modules)

Click on a service and type a depth level you want to see.

Depth: 1

Metrics

Global:

Number of system components

43 services and 43 modules

Number of services with deployment dependency

0

Largest service (27 operations)

ts-order-service

Smallest service (0 operations)

ts-common, ts-news-service, ts-ui-dashboard, ts-ui-test

Number of components that consume messages from the queue

- modules: 0
- services: 0

Number of components that publish messages in the queue

- modules: 0
- services: 0

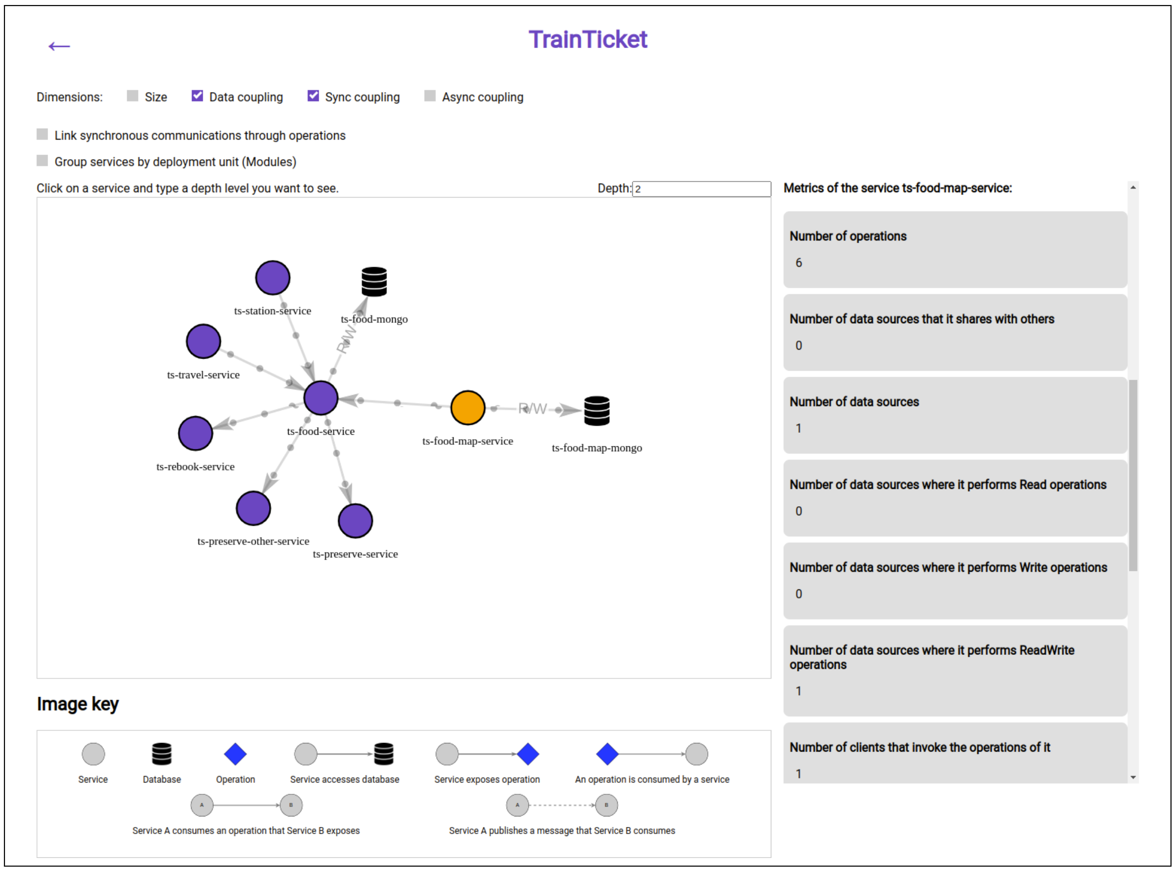
Image key

ts-security-service
ts-security-mongo
R/W
GET /api/v1/orderservice/order/+d
GET /api/v1/orderservice/order/+departureTime
POST /api/v1/stationservice/stations/namelist
GET /api/v1/orderservice/order/security/ + checkDate + / + accountId
GET /api/v1/orderOtherService/orderOther/security/ + checkDate + / + accountId
POST /api/v1/orderservice/order
ts-order-service
PUT /api/v1/orderservice/order
ts-order-mongo
GET /api/v1/orderservice/order/+orderId
rderOtherService/orderOther
ts-order-other-service
Service
Database
Operation
Service accesses database
Service exposes operation
An operation is consumed by a service
Service A consumes an operation that Service B exposes
Service A publishes a message that Service B consumes

Figure 4.18: *New frontend: Synchronous calls through operations*

Sometimes we can be analyzing a large system with a huge amount of information, which makes it difficult to understand its architecture in the Sorting Hat. For instance, we can only be interested in a set of services and their interactions. With this situation in mind, we added a filter option, which can be used by clicking in a service in the graph and typing a depth level. Then, the view will be filtered to display only the interactions with the clicked service until it reaches the depth level typed, as shown in the Figure 4.19.

It is possible to see in the Figure 4.19 that the microservice `ts-food-map-service` was selected and the depth level 2 was typed, which made the view be filtered and easier to analyze. When a service is selected by clicking on it, the CharM metrics specifics for that service is shown, so we can understand more that specific service and the other services and databases liked to it.



←

TrainTicket

Dimensions: ☐ Size ☑ Data coupling ☑ Sync coupling ☐ Async coupling

☐ Link synchronous communications through operations

☐ Group services by deployment unit (Modules)

Click on a service and type a depth level you want to see. Depth: 2

Metrics of the service ts-food-map-service:

Number of operations	6
Number of data sources that it shares with others	0
Number of data sources	1
Number of data sources where it performs Read operations	0
Number of data sources where it performs Write operations	0
Number of data sources where it performs ReadWrite operations	1
Number of clients that invoke the operations of it	1

ts-station-service
ts-food-mongo
R/W
ts-travel-service
ts-food-service
ts-food-map-service
R/W
ts-food-map-mongo
ts-rebook-service
ts-preserve-other-service
ts-preserve-service
Image key
Service
Database
Operation
Service accesses database
Service exposes operation
An operation is consumed by a service
A
B
Service A consumes an operation that Service B exposes
A
B
Service A publishes a message that Service B consumes

Figure 4.19: *New frontend: View filtered by depth level*

The components architecture of the new frontend is illustrated in the Figure 4.20, which shows the pages the user can navigate to, as well as the web components used by these pages and the interactions of components with hooks and services – responsible for handling the logic in the application.

When the user goes to the system page, it uses the `SystemService` – a separate file that handles fetching the data from an external API, in our case the `Backend`. The pages fetches the system data and its metrics through the System service. After that, the page distributes the data to the `Graph` component – which handles displaying the system data as a graph – and the `MetricsWrapper` component – which takes the system metrics and display them. The page also uses the `DimensionSelector` component, which handles the selection of CharM dimensions.

The Graph component uses a hook called `useGraph`, responsible for handling the logic of the interaction with the graph and for configuring an auxiliary library to display the graph. The library used to display the graph is the force-graph, which is really easy to configure and use. The `useGraph` hook receives the raw data of the system, and it needs to transform them into a data structure that can be used by the library. It does this by using a service called `GraphDataProcessor`.

Finally, the `MetricsWrapper` component uses a hook called `useMetrics`, which handles the logic of filtering and displaying the metrics list.

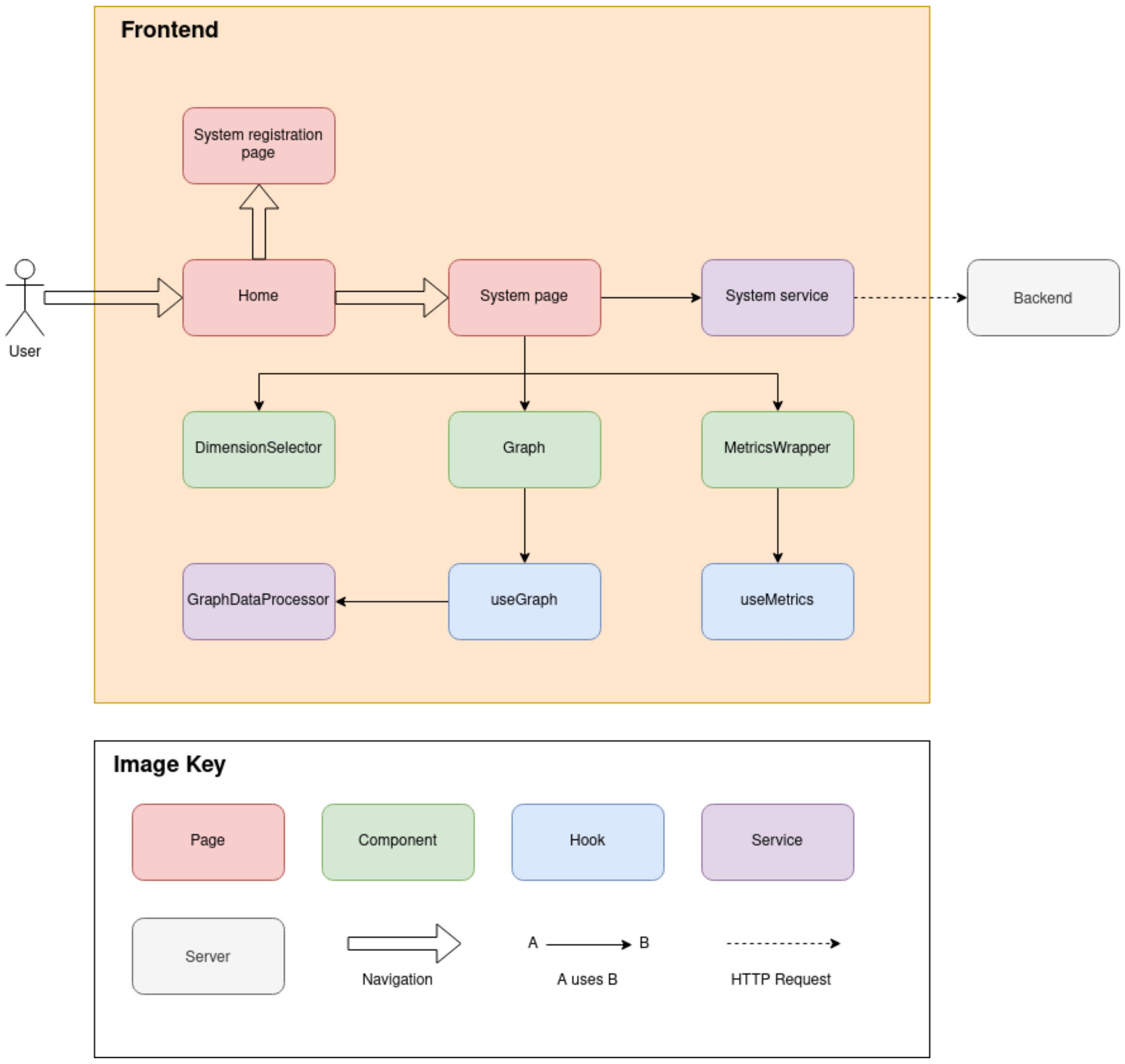


Figure 4.20: *Frontend components architecture*

The new version of the frontend was built using Next.js, a React framework that allows the construction of SPAs (single-page applications), a modern way of developing websites in which navigation between pages is done more fluidly, without the need for a browser make a new request to the server where the site is hosted at each page change.

The new version of the frontend is deployed at this link, and the code is open-source and is available at this repository link. It has more than 1700 lines of code written.

Chapter 5

Personal Experiences

Since the beginning of 2021, I have been working on the development of this tool. In that year, we had the opportunity to write a paper about the first MVP and we presented the work in the CBSoft's Undergraduate Research on Software Engineering Competition, which we were awarded as the third best Undergraduate Research on Software Engineering.

With the great results achieved, the Prof. Dr. Alfredo Goldman talked to the Prof. Dr. André van der Hoek, from the University of California in Irvine (UCI), about the project, which was received with interest by him. Then, we discussed about an opportunity to do an exchange abroad from April 1, 2022 to July 31, 2022, so Prof. André and I could continue developing the tool.

Professor Dr. André van der Hoek is co-author of the books "Software Design Decoded: 66 Ways How Experts Think" and "Studying Professional Software Design: a Human-Centric Look at Design Work", the only published books that detail the practices of software design professionals. He was recognized as Outstanding Scientist by the ACM in 2013 and received the Award for Excellence in Engineering Education Courseware in 2009. Additionally, in 2005, he was honored Professor of the Year at UCI for his outstanding and innovative educational contributions. Additional information about the work of Professor André van der Hoek can be found in his profile at Google Scholar and in DBLP.

Therefore, considering the vast experience that Professor Dr. André van der Hoek has in the areas of Software Architecture and Experimental Software Engineering, he was able to contribute to the enrichment of our project. Through his in-depth knowledge of the day-to-day life of software architects, he was able to contribute to making the Sorting Hat platform a really practical and useful guide for software development professionals. Furthermore, through the values cultivated by the UCI Department of Informatics and the Software Design and Collaboration Laboratory, our project was enriched by exchanging experiences and ideas with other students.

I arrived in California on March 29th, and the first in-person meeting with Prof. André happened on April 1st. We got along very well during the internship period, and his collaborations were very rich to the project. Our interactions used to happen twice a week: one in-person and another one virtually by text messages.

We used to do a group meeting in-person at the university on Mondays, with his other advisees, composed mostly of graduate students. Those group meetings were really valuable to me, because I had opportunities to interact with other students and learn about other researches, as well as I had the opportunity to explain my research to them, which helped me to improve my communication skills in a foreign language.

I would go to the university twice a week, usually on Mondays and Fridays. On the other week days, I would go to a library next to the house I was living in. I preferred to do that because I was living in a city called Santa Ana, which was almost one hour by bus far from the University of California, Irvine. Thus, I could save some time of my day. On weekends, I used to meet some places in order to explore California, live the American culture and improve my English skills.

During the beginning of the internship, I faced some difficulties regarding the foreign language, getting used to a new culture, and living with other people. I stayed in a shared bedroom in a shared house, and the house had many people, which often put me in situations that required greater language skills. Because of that, I got in touch with different cultures such as Mexican and from other Latin countries. Other difficulties faced were related to daily life, such as going to supermarket, taking a bus, and so on.

The lessons I learned from this international internship experience were many. I had the opportunity to work together with a renowned professor in my area of interest, and interact with other graduate students, which made me improve my communication skills, and gain confidence. Also, I learned a new way of thinking, a new way of looking at problems and solving them. Professor André brought new ideas for the project and sometimes put me in situations that required me to think in other ways.

By living in another country on my own, with different people and a different culture, I could learn how the real life is. I learned how to deal with people who think differently from me, and I learned how to deal with daily problems, which is part of everyone's life

All these lessons learned really make me feel proud of what I did professionally, how I dealt with the daily situations, and most important, the person I became after this international internship experience.

Chapter 6

Conclusion

6.1 Contributions

In this final essay, we showed the Sorting Hat, a tool to characterize the architecture of service-based systems. This tool has been under development since 2021. In that year, we developed a MVP for the visualization of the tool. Despite achieving great results in that time, we noticed that the visualization had some issues regarding its usability, not being intuitive to understand, and the CharM metrics were not up-to-date. In addition, the data available in the tool were manually collected, in an arduous process that is computationally automatable. So the goals of this final work were (1) to provide an automated data collection to the Sorting Hat, and (2) to improve the visualization of the tool and update its information to the most recent CharM metrics.

In this work we have presented the improvements and updates in the Sorting Hat, which reach the proposed goals. We have provided some data collection to the tool. Now, the tool analyzes docker-compose and openapi files to collect the services, endpoints, databases and databases usages of a new service-based system registered in the tool. We have also solved the issues related to the tool usability: now, all information take place in a single page, with a reformulated, more elegant design. Also, there are many filter options in order to make the visualization more interactive. In addition, the CharM metrics are up-date and automatically extracted for every new system in the tool.

6.2 Future Work

Although we have achieved the goals proposed, there is still a need to improve the tool in several aspects. It is still not possible to collect data related to communications between services (synchronous and asynchronous). Therefore, for a future work, it is possible to study new data collection strategies to collect the remaining data the tool needs.

References

[Bass *et al.* 2013] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. 3rd. Addison-Wesley, 2013 (cit. on pp. 1, 3).

[Fowler 2019] Martin Fowler. *Software Architecture Guide*. Url: https://martinfowler.com/architecture/, 2019 (cit. on p. 1).

[Fowler and Lewis 2014] Martin Fowler and James Lewis. *Microservices*. Url: https://martinfowler.com/articles/microservices.html, 2014 (cit. on p. 3).

[Garlan and Shaw 1994] David Garlan and Mary Shaw. "An introduction to software architecture". In: *Carnegie Mellon University* CMU-CS-94-166 (1994) (cit. on p. 3).

[Newman 2015] Sam Newman. *Building Microservices: Designing Fine-Grained System*. 1st. O'Reilly Media, 2015 (cit. on pp. 3, 4).

[Perry and Wolf 1992] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes, 1992 (cit. on pp. 1, 4).

[Rosa *et al.* 2020] T. Rosa, A. Goldman, and E. Guerra. *Modelo para Caracterização e Evolução de Sistemas com Arquitetura Baseada em Serviços*. Workshop de Teses e Dissertações do CBSoft - WTDSoft 2020, 2020 (cit. on pp. 1, 4, 7, 11).

[Silva and Balasubramaniam 2012] Lakshitha de Silva and Dharini Balasubramaniam. *Controlling software architecture erosion: A survey*. Journal of Systems and Software, 2012 (cit. on p. 4).

[Soldani *et al.* 2018] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: a systematic grey literature review". In: *Journal of Systems and Software* 146 (2018) (cit. on p. 1).