Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Integração do Sorting Hat com usVision

Um relato de Engenharia de Software

Vinicius Pereira Ximenes Frota

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Prof. Dr. Eduardo Guerra
Cossupervisor: MSc. João Francisco Lino Daniel

São Paulo
2024

Resumo

Vinicius Pereira Ximenes Frota. **Integração do Sorting Hat com usVision: *Um relato de Engenharia de Software***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

A Arquitetura de Microsserviços (MSA) é amplamente adotada na engenharia de software. Contrapondo o monolito, ela propõe que o sistema seja dividido em microsserviços disponibilizados independentemente. Isso permite que uma única aplicação seja feita usando diferentes tecnologias em cada microsserviço, como bibliotecas, arcabouços, e até mesmo linguagens de programação. Como consequência, as aplicações que seguem a MSA são de alta complexidade, desafiando compreensão e depuração.

Para auxiliar o desenvolvimento ou até mesmo melhorar o desempenho de um software, é preciso evitar más práticas de arquitetura de microsserviços, avaliando as aplicações frequentemente. Atualmente, existem ferramentas que avaliam ou controlam questões específicas (ferramenta que limita o número de operações públicas de um serviço que utiliza determinado arcabouço, por exemplo). A usVision, por outro lado, foca-se na detecção de padrões e mau-cheiros de MSA, e pode ser utilizada para avaliar aplicações com diversas tecnologias por guiar-se apenas por métricas.

Antes deste trabalho, a usVision dependia que seu banco de dados já esteja populado para funcionar. Com este trabalho, dei suporte à inserção de dados na usVision, estendendo seu escopo para evolucionabilidade. Como resultado, fiz um módulo de inserção de dados por meio de uma interface REST. Assim, este trabalho possibilita uma integração do usVision com o Sorting Hat, uma ferramenta de visualização de arquiteturas distribuídas.

Palavras-chave: Arquitetura de microsserviços. Interface REST. Evolucionabilidade.

Abstract

Vinicius Pereira Ximenes Frota. **Integration of the Sorting Hat with usVision: *A Software Engineering Case Study***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Microservices Architecture (MSA) is widely adopted in software engineering. Unlike monolithic architecture, it proposes dividing the system into independently deployed microservices. This allows a single application to be built using different technologies for each microservice, such as libraries, frameworks, and even programming languages. Consequently, applications that follow the MSA approach are highly complex, making them challenging to understand and debug.

To support development or even improve the performance of software, it is essential to avoid poor microservices architecture practices by frequently evaluating the applications. Currently, there are tools that assess or control specific aspects (for example, tools that limit the number of public operations in a service using a given framework). usVision, on the other hand, focuses on detecting MSA patterns and anti-patterns, and can be used to evaluate applications with various technologies, as it relies solely on metrics.

Before this project, usVision required its database to already be populated to function. With this work, I provided support for data insertion into usVision, extending its scope to address evolvability. As a result, I developed a data insertion module through a REST interface. This work enables the integration of usVision with Sorting Hat, a tool for visualizing distributed architectures.

Keywords: Microservices Architecture. REST Interface. Evolvability.

Lista de figuras

Sumário

Introdução

A Arquitetura de Microsserviços (MSA) é amplamente adotada na engenharia de software. Contrapondo o monolito, ela propõe que o sistema seja dividido em microsserviços disponibilizados independentemente. Isso permite que uma única aplicação seja feita usando diferentes tecnologias em cada microsserviço, como bibliotecas, arcabouços, e até mesmo linguagens de programação. Como consequência, as aplicações que seguem a MSA são de alta complexidade, desafiando compreensão e depuração.

Para auxiliar o desenvolvimento ou até mesmo melhorar o desempenho de um software, é preciso evitar más práticas de arquitetura de microsserviços, avaliando as aplicações frequentemente. Atualmente, existem ferramentas que avaliam ou controlam questões específicas (ferramenta que limita o número de operações públicas de um serviço que utiliza determinado arcabouço, por exemplo). Especificamente, a usVision modela MSAs e foca-se na detecção de padrões e anti-padrões na arquitetura modelada, podendo ser utilizada para avaliar aplicações com diversas tecnologias por guiar-se apenas por métricas (Daniel *et al.*, 2023).

O Sorting Hat, por outro lado, também modela MSAs, mas possui como funcionalidade coletar as informações dos microsserviços de um sistema por meio de repositórios no GitHub a partir de heurísticas e análises do Docker. Além de ler o arquivo de Docker Compose, detectando os bancos de dados utilizados pelos microsserviços, ela também lê os arquivos de documentação da interface REST utilizando a especificação do openAPI, detectando todas as rotas HTTP expostas pelos microsseviços (Santana *et al.*, 2021).

Antes deste trabalho, a usVision não possuía módulo de inserção e dependia que seu banco de dados já esteja populado para funcionar. Por isso, os usuários desta ferramenta precisavam entrar em contato diretamente com o banco de dados, realizando as operações de inserção.

Inspirado pelo Sorting Hat, com este trabalho, criei um módulo de criação para o usVision, dando suporte à inserção de dados, para que sistemas como o Sorting Hat ou outro parecido possa se integrar a ele, populando seu banco de dados e até utilizando os relatórios fornecidos. Este módulo de criação foi integrado ao módulo REST pré-existente no usVision, permitindo que um sistema externo, como o Sorting Hat ou um futuro front end para o usVision possa se comunicar com ele. Além disso, este trabalho abre possibilidades para outras estratégias de comunicação além do REST a partir da criação de outro módulo que possui o módulo de criação como dependência.

Capítulo 1

Roadmap

A figura a seguir mostra o passo a passo seguido para a confecção deste projeto.

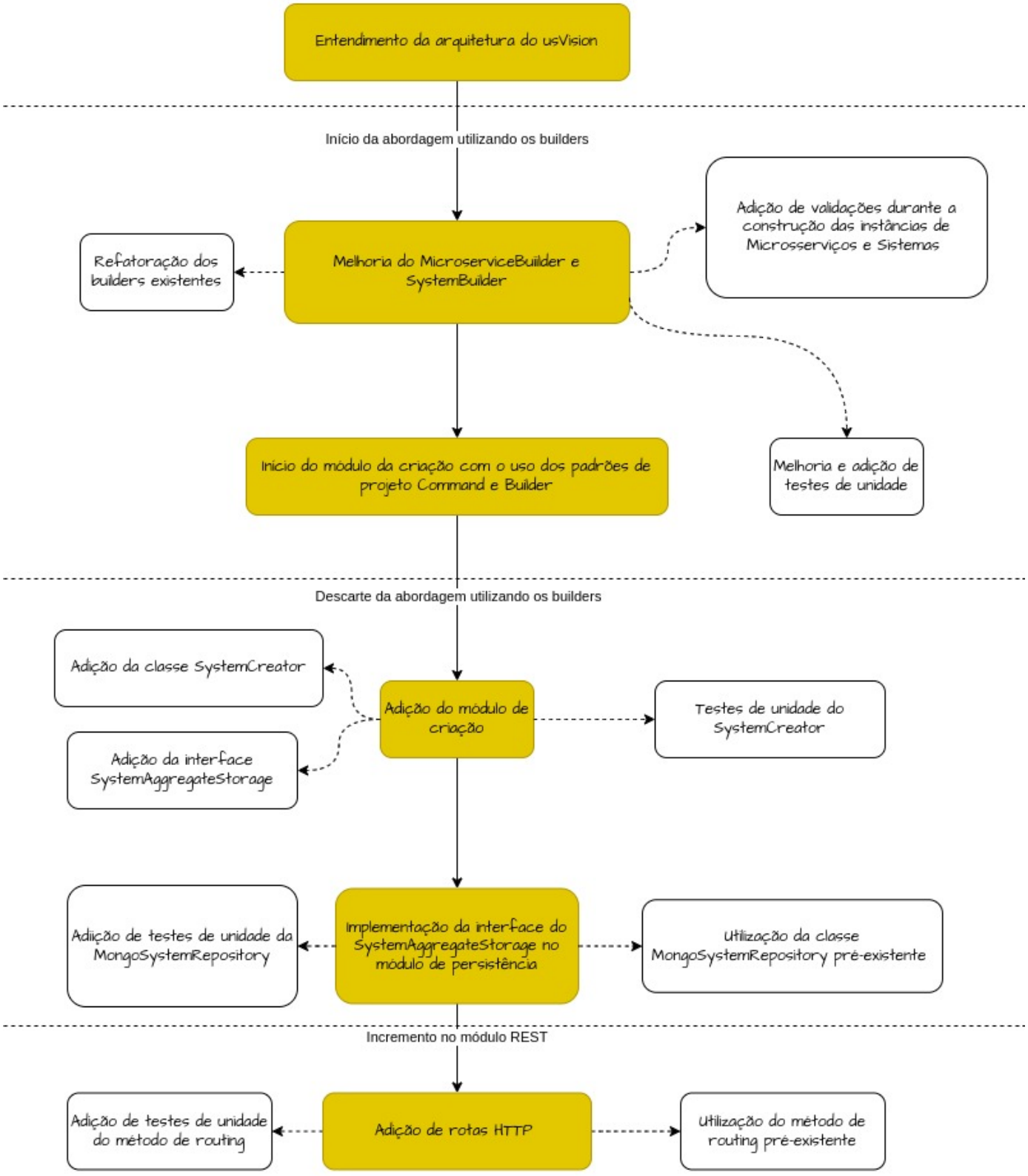

Figura 1.1: *Roadmap do projeto*

Capítulo 2

Entendimento da arquitetura do usVision

O usVision é uma aplicação escrita em Kotlin e possui o Gradle como gerenciador de dependências. Ele possuia diversos módulos, cada uma responsabilidade:

- app-reports: módulo responsável pela geração de relatórios.
- app-analyses: módulo responsável pela análise de cada padrão ou anti-padrão de arquitetura de microsserviços.
- app-persistence: módulo responsável pela persistência de dados. Ela possuia implementação específica para o MongoDB. Será aprofundado posteriormente, pois este trabalho consiste em deixar de apenas ler as informações dos sistemas que utilizam a arquitetura de microsserviços e escrever. Portanto, houve a necessidade de incrementar a funcionalidade dele.
- app-model: módulo responsável pela modelagem dos sistemas que utilizam arquiteturas de microsserviços. Será aprofundado posteriormente por fornecer justamente os modelos que serão salvos e recuperados através do módulo de persistência.
- app-web: é o módulo REST, responsável por expor rotas HTTP e permitindo que um servidor web seja levantado. Será aprofundado posteriormente, por ser a partir dele que sistemas externos terão acesso ao módulo de criação.

O app-model possui o pacote com.usvision.model.domain que possui as classes referentes necessárias para a modelagem de microsserviços:

- CompanySystem: cada instância dele representa um sistema que utiliza arquitetura de microsserviços. Possui um atributo para seu nome e outro atributo para seus subsistemas, que podem ser tanto uma instância de CompanySystem como uma instância de Microservice.
- Microservice: cada instância dele representa um microsserviço. Possui atributos para seu nome, canais de mensageria em que ele publica e em que está escrito, bancos de dados utilizados, operações que ele expõe ou consome e seu módulo.

- MessageChannel: cada instância dele representa um canal de mensageria, possui um nome e um id.
- Module: cada instância dele representa um módulo.
- RestEndpoint: cada instância dele representa um endpoint REST. Esta classe implementa a interface Operation, isto é, ela é uma operação.
- PostgreSQL: cada instância dele representa um banco de dados PostgreSQL. Esta classe implementa interface Database, que por sua vez é a interface dos bancos de dados.

Este módulo também conta com o pacote com.usvision.model.systemcomposite, que possui as classes abstratas System, SystemOfComponents e SystemOfSystems. Elas são utilizadas pelas classes CompanySystem e Microservice e garantem que elas utilizam o padrão Composite (Gamma *et al.*, 1994), pois um CompanySystem é uma composição de outros ComapanySystems e Microservices.

Além disso, ela também possui o com.usvision.model.visitor, com as interfaces Visitor, implementado nas classes do módulo app-analyses. Neste pacote também há a classe Visitable, implementado nas classes do pacote com.usvision.model.domain (exceto nas classes RestEndpoint e MessageChannel). O padrão Visitor (Gamma *et al.*, 1994) é utilizado para cada nó ser visitado durante as análises de padrões de microsserviços fornecidas pela app-analyses.

Por fim, o app-model também possui o com.usvision.model.systembuilder, que por sua vez possui as classes SystemBuilder e MicroserviceBuilder. O SystemBuilder possui métodos para adicionar nome, adicionar subsistemas, adicionar microsserviços filhos e devolver uma instância de CompanySystem. Já o MicroserviceBuilder, possui métodos para adicionar canais de mensageria, operações, banco de dados, nome do microsserviço, e devolver uma instância de Microservice. Como os nomes delas já sugerem, estas classes implementam o padrão builder (Gamma *et al.*, 1994).

As classes principais do módulo app-persistence são:

- MongoSystemRepository: implementa a interface SystemRepository que está contida no módulo app-reports e é responsável pela leitura dos dados. Este trabalho faz ela implementar a interface SystemAggregateStorage do novo módulo de criação, que será detalhado posteriormente.
- SystemDocument: é a classe que representa Microservices e CompanySystems mapeados em um documento do MongoDB.
- DatabaseDocument: é a classe que representa um banco de dados mapeado em um documento do MongoDB.
- MessageChannelDocument: é a classe que representa um canal de mensageria mapeado em um documento do MongoDB.
- ModuleDocument: é a classe que representa um módulo mapeado em um documento do MongoDB.

- SystemMapper: é a classe que mapeia as classes dos modelos, como CompanySystem, Microservice, MessageChannel, entre outros em classes de Document.
- MongoDBRepositoryProvider: responsável por conectar com o banco de dados e fornecer a instância de MongoSystemRepository.

O módulo app-web utiliza o Ktor, um arcabouço para desenvolvimento de servidores web. Neste módulo, há funções de extensão da classe Application, que já é fornecida pelo arcabouço. Cada função representa um módulo a ser carregado pelo servidor. Algumas dessas funções ou módulos são:

- configureRouting: configura as rotas HTTP.
- configureReports: configura as dependências de geração de relatórios. Embora o foco do trabalho não seja a geração de relatórios, foi adicionado nesta seção, por haver a necessidade de alteração de seu código.
- module: faz as chamadas para os demais módulos.

Capítulo 3

Uso do padrão builder

Como o SystemBuilder e o MicroserviceSystemBuilder possuem a responsabilidade de criar CompanySystems e Microservices, acreditou-se que este padrão seria necessário. Inicialmente, para a construção do módulo de criação de sistemas, começou-se uma exploração no SystemBuilder através da leitura do código e testes de unidade.

Os testes de unidade (também chamados de testes unitários no contexto corporativo), são códigos que testam a funcionalidade da menor parte de um software. No caso de uma linguagem orientada a objetos, elas costumam testar métodos públicos. Para cada classe que se deseja testar, cria-se outra classe de teste, cujos nomes dos métodos descrevem o comportamento esperado para cada cenário.

3.1 Ajustando a construção de um sistema com dois microsserviços

A figura a seguir demonstra um código de um teste unitário que descreve o cenário da construção de um sistema que se chama “pingr” e possui um subsistema denominado “backend”. Este subsistema, por sua vez, possui microsserviços: um denominado “pings”, que expõe um endpoint POST e publica no tópico “pings-chan-id” e outro denominado “timeline”, que expões um endpoint GET e é inscrito nesse mesmo tópico.

```
@Test
fun `it builds an entire system`() {
    // given
    val pingsTopicId = "pings-chan-id"
    val pingsTopicName = "pings"
    val timelinePgId = "pg-one"

    // when
    val system = underTest
        .setName("pingr")
        .addSubsystems()
            .setName("backend")
            .thatHasMicroservices()
                .oneNamed("pings")
                .exposingRestEndpoint("POST", "/users/{uid}/pings", "create new ping")
                .thatPublishesTo(pingsTopicId, pingsTopicName)
                .and()
                .anotherNamed("timeline")
                .exposingRestEndpoint("GET", "/users/{uid}/timeline", "the user's timeline")
                .thatIsSubscribedTo(pingsTopicId, pingsTopicName)
                .accessingPostgres(timelinePgId)
            .endMicroservices()
        .endSubsystems()
        .build()

    // then
    assertIs<CompanySystem>(system)
    assertEquals(1, system.getSubsystemSet().size)
    val firstLevelSubsys = system.getSubsystemSet().first()
    assertIs<CompanySystem>(firstLevelSubsys)
    assertEquals(1, firstLevelSubsys.getSubsystemSet().size)
    assertIs<Microservice>(firstLevelSubsys.getSubsystemSet().first())
}
```

Figura 3.1: *Teste de unidade da criação de um sistema inteiro*

O resultado é salvo na variável system. Após essas atribuições, algumas asserções são feitas:

1. System é uma instância de CompanySystem.
2. O sistema “pingr” possui apenas um subsistema filho.
3. O subsistema filho também é uma instância de CompanySystem.
4. O subsistema possui apenas um microsserviço.
5. O filho do subsistema é uma instância de um Microservice.

Note que a asserção número 4, destacado na linha 49 da figura do código, está errada, pois, como descrito anteriormente, o subsistema criado possui dois microsserviços. Por isso, a primeira alteração foi ajustar isso, conforme ilustrado na figura posterior:

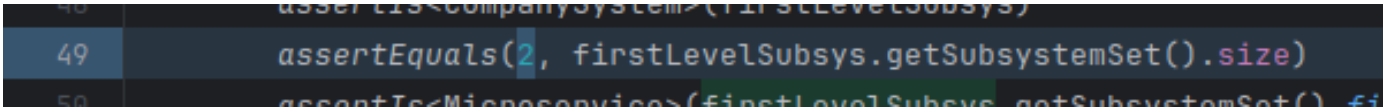

Figura 3.2: *Ajuste feito no teste de unidade*

Após essa alteração, o teste passou a falhar, indicando que o builder precisava de alterações para funcionar corretamente. Observou-se que o SystemBuilder possuia um atributo com a palavra-chave lateinit, isto é, ele não pode assumir o valor nulo, mas também não é inicializado durante a instanciação de um SystemBuilder. Esta classe também possuia um método denominado thatHasMicroservices, que inicializava o subsystems com um conjunto vazio e retornava um MicroserviceBuilder que recebia a própria instância de SystemBuilder que o invocou em seu construtor.

```
class SystemBuilder(private val parent: SystemBuilder? = null) {
    private lateinit var rootName: String
    private lateinit var subsystems: MutableSet<System>
```

Figura 3.3: *Atributos do SystemBuilder*

```
    fun thatHasMicroservices(): MicroserviceBuilder {
        subsystems = mutableSetOf()
        return MicroserviceBuilder(this)
    }
```

Figura 3.4: *Método thatHasMicroservices*

A investigação do problema continuou no MicroserviceBuilder. Em seu construtor, ele recebia um parâmetro parent, que, por conta das palavras-chave private val, também era um atributo. O método and fazia uma chamada para thatHasMicroservices do atributo parent. Conforme visto no parágrafo anterior, este thatHasMicroservices atribuía um conjunto vazio ao SystemBuilder novamente, portanto, acabava eliminando o Microservice construído. Assim, a solução para este problema foi remover esta chamada, substituindo-a pelo construtor do MicroserviceBuilder, passando parent como parâmetro.

```
class MicroserviceBuilder(private val parent: SystemBuilder? = null) {
    private lateinit var name: String
    private val exposedOperations = mutableSetOf<Operation>()
    private val consumedOperations = mutableSetOf<Operation>()
    private val databases = mutableSetOf<Database>()
    private val channelsPublished = mutableSetOf<MessageChannel>()
    private val channelsSubscribed = mutableSetOf<MessageChannel>()
```

Figura 3.5: *Atributos do MicroserviceBuilder*

```
fun and(): MicroserviceBuilder {
    endMicroservices()
    return parent!!.thatHasMicroservices()
}
```

Figura 3.6: *Método and antes das alterações*

```
86    fun and(): MicroserviceBuilder {
87        endMicroservices()
88        return MicroserviceBuilder(this.parent)
89    }
90
```

Figura 3.7: *Método and depois das alterações*

Pode-se relacionar este ajuste com o ciclo de TDD (Test Driven Development). Embora o código não tenha sido feita do zero, a etapa em que corrigimos o teste unitário, corresponde à etapa "escrever testes que falham", pois o teste passou a falhar assim que foi descrito o cenário corretamente no código. A etapa de correção corresponde à etapa "fazer o teste passar".

3.2 Comportamento do campo name para o MicroserviceBuilder

Ainda seguindo a abordagem do TDD, foi adicionado um teste unitário que verificava que, ao construir um Microservice com o MicroserviceBuilder sem atribuir um valor ao atributo name, a exceção SystemBuilderException é lançada. Assim, foi garantido que o microsserviço a ser construído sempre possuia um nome e teremos um comportamento no nosso controle caso isso não ocorra. Na figura, underTest é uma instância de MicroserviceBuilder. Como esse comportamento não acontecia, o teste passou a falhar novamente.

Para o MicroserviceBuilder passar no teste, a palavra-chave lateinit foi removida do atributo name, garantindo que ele possua algum valor logo na instanciação do builder. No caso, ele passou a se inicializar com o valor null.

Antes, o método build tentava construir o Microservice mesmo sem verificar se o campo name possui um valor associado, causando a exceção UninitializedPropertyAccessException caso o campo não tenha sido atualizado. Com a modificação citada no parágrafo anterior, passou a ser possível verificar se o valor do name é null ou teve um novo valor atribuído, fornecendo um maior controle ao desenvolvedor. Por isso, foi utilizado um safe call do let, que garantia que o bloco da instanciação do Microservice só fosse feita se o name não fosse nulo. Além disso, foi utilizado um elvis operator para garantir que, caso o name fosse nulo, a exceção SystemBuilderException seja lançada, satisfazendo assim o teste unitário escrito anteriormente.

É importante ressaltar que este controle poderia ser feito mantendo o uso do lateinit e o método isInitialized para verificar se o name já possui valor atribuído e lançar o

SystemBuilderException. No entanto, esta abordagem não foi utilizada por decisão de estilo, preferiu-se evitar a utilização de reflexão do Kotlin de modo a deixar o funcionamento mais explícito no código. Semanticamente, deixar um atributo anulável, significa que null é um dos estados que o builder pode assumir. Por outro lado, usar lateinit significa que o atributo não possuir valor é uma situação atípica (o que não faz sentido para o contexto, já que é possível atribuir o valor do name após dar os valores de outros campos).

```
161     @Test
162     fun `it throws SystemBuilderException when building a Microservice with no name`() {
163         // given nothing
164         // when ... then
165         assertThrows<SystemBuilderException> {
166             underTest.build()
167         }
168     }
169
```

Figura 3.8: *Teste de unidade adicionado ao MicroserviceBuilder que testa quando não há o parâmetro name*

```
class MicroserviceBuilder(private val parent: SystemBuilder? = null) {
    private lateinit var name: String
```

Figura 3.9: *MicroserviceBuilder utilizando o atributo name com lateinit*

```
63  class MicroserviceBuilder(private val parent: SystemBuilder? = null) {
64      private var name: String? = null
```

Figura 3.10: *MicroserviceBuilder após ajuste do atributo name*

```
    fun build(): Microservice {
        return Microservice(name).also { msvc ->
            exposedOperations.forEach(msvc::exposeOperation)
            consumedOperations.forEach(msvc::consumeOperation)
            databases.forEach(msvc::addDatabaseConnection)
            channelsPublished.forEach(msvc::addPublishChannel)
            channelsSubscribed.forEach(msvc::addSubscribedChannel)
        }
    }
}
```

Figura 3.11: *MicroserviceBuilder antes das alterações no método build*

```
119    fun build(): Microservice {
120        return this.name?.let { name ->
121            Microservice(name).also { msvc ->
122                exposedOperations.forEach(msvc::exposeOperation)
123                consumedOperations.forEach(msvc::consumeOperation)
124                databases.forEach(msvc::addDatabaseConnection)
125                channelsPublished.forEach(msvc::addPublishChannel)
126                channelsSubscribed.forEach(msvc::addSubscribedChannel)
127            }
128        } ?: throw SystemBuilderException("Attempted to build a Microservice with no name")
129    }
```

Figura 3.12: *MicroserviceBuilder depois das alterações no método build*

3.3 Comportamento da atribuição dupla ao campo name do MicroserviceBuilder

Além do ajuste citado na seção anterior, foi adicionado um teste unitário que lança o SystemBuilderException caso o parâmetro name seja atribuído duas vezes via método named. Como já era esperado ao adicionar comportamento novo, o teste falhou, mas bastou adicionar essa validação neste método para corrigir isso.

```
301    @Test
302    fun `it throws SystemBuilderException when calling 'named' method twice`() {
303        // given
304        val name = "micro"
305
306        val result = underTest
307            .named(name)
308
309
310        // when ... then
311        assertThrows<SystemBuilderException> {
312            result.named(name)
313        }
314    }
315
```

Figura 3.13: *Teste de unidade adicionado ao MicroserviceBuilder que testa quando o name é atribuído duas vezes*

```
91     fun named(name: String) : MicroserviceBuilder = fluentInterface {
92         this.name?.also {
93             throw SystemBuilderException("Microservice already has a name: $name")
94         }
95
96         this.name = name
97     }
```

Figura 3.14: *Tratativa adicionada ao MicroserviceBuilder para o o name é sendo atribuído duas vezes*

3.4 Correção da elaboração de testes para o MicroserviceBuilder e comportamento da chamada dupla do método and()

O método and() do MicroserviceBuilder permite finalizar a instanciação de um Microservice, adicionando-o no SystemBuilder e logo em seguida retornando um novo MicroserviceBuilder para a construção de outro Microservice. Portanto, com as modificações citadas nas seções anteriores, que adicionam checagem na instanciação de um Microservice sem nome através do lançamento do SystemBuilderException, esperava-se que duas chamadas consecutivas do and() acarretaria o lançamento desta exceção. Na primeira chamada, ele instanciaria o Microservice que estava sendo construído com sucesso, retornando um novo MicroserviceBuilder, e na segunda chamada ele tentaria instanciar um Microservice através do último MicroserviceBuilder, mas isso causaria uma exceção ser lançada, pois seu nome não foi determinado. A figura a seguir mostra o código do teste.

```
@Test
fun `it throws SystemBuilderException when calling 'and' method twice without setting a name for the second microservice`() {
    // given
    val name = "micro"

    // when
    val result = underTest
        .named(name)
        .and()


    //then
    assertThrows<SystemBuilderException> {
        result.and()
    }
}
```

Figura 3.15: *Teste de unidade adicionado ao MicroserviceBuilder que testa uma chamada dupla do método and*

Após rodar este teste de unidade, a exceção SystemBuilderException foi lançada na primeira chamada do and() e não na segunda chamada, como era esperado. Além disso, havia um método de outro teste de unidade pré-existente que também falhou. A figura a seguir mostra o código dele:

```
@Test
fun `it returns its parent when closing a properly opened microservice env`() {
    // given
    val parent = spyk(SystemBuilder())
    underTest = MicroserviceBuilder(parent)

    // when
    val environment = underTest
        .named("something")
        .endMicroservices()

    // then
    assertIs<SystemBuilder>(environment)
    assertEquals(parent, environment)
    verify { parent.addMicroservice(any()) }
}
```

Figura 3.16: *Teste de unidade pré-existente que falhou*

Neste último caso, foi na chamada do método endMicroservices que foi responsável pelo lançamento de uma exceção. Portanto, foi necessária a investigação para saber o que ocasionou cada um desses erros.

A figura a seguir mostra a implementação dos métodos que lançaram exceção:

```
76    fun endMicroservices(): SystemBuilder {  João Daniel
77        if (parent == null)
78            throw SystemBuilderException("Attempted to close an environment that had not being opened")
79
80        val microservice = build()
81        parent.addMicroservice(microservice)
82
83        return parent
84    }
85
86    fun and(): MicroserviceBuilder {  João Daniel +1
87        endMicroservices()
88        return MicroserviceBuilder(this.parent)
89    }
```

Figura 3.17: *Métodos que lançaram exceção*

Quando o and é executado, e o endMicroservices também é, o que permite analisarmos os erros ocorridos em dois contextos diferentes. Exceções podem ocorrer em dois casos:

1. Atributo parent do MicroserviceBuilder é null, isto é, o Microservice não teve um valor atribuído ao seu pai.
2. Atributo parent é um CompanySystem instanciado, mas não possui uma lista de microsserviços inicializados, já que o atributo subsystems do SystemBuilder é lateinit, que possui inicialização tardia conforme explicado em seções anteriores. A tentativa de adicionar via addMicroservice desencadeia em um UninitializedPropertyAccessException, pois ele consiste apenas em chamar o método add deste atributo.

```
15  class SystemBuilder(private val parent: SystemBuilder? = null) {  João Daniel
16      private lateinit var rootName: String
17      private lateinit var subsystems: MutableSet<System>
```

Figura 3.18: *Atributos do SystemBuilder*

```
51      fun addMicroservice(microservice: Microservice) : SystemBuilder = fluentInterface {  João Daniel
52          subsystems.add(microservice)
53      }
```

Figura 3.19: *Método addMicroservice*

O teste que estava sendo escrito utilizava o atributo underTest, cujo valor é atualizado através do método anotado com a annotation BeforeTest. Esta annotation, com o próprio nome explicita, roda antes de cada teste. Conforme indicado no código, o underTest não possui parent. Para garantir o funcionamento correto, modificou-se este método, adicionando a instanciação de um CompanySystemBuilder para o underTest ter um parent.

```
@BeforeTest
fun `create clean, new instance of MicroserviceBuilder`() {
    underTest = MicroserviceBuilder()
}
```

Figura 3.20: *Método que roda antes de cada teste de unidade e atribui valor ao underTest antes da correção*

```
114     @BeforeTest
115     fun `create clean, new instance of MicroserviceBuilder`() {
            val parent = spyk(SystemBuilder())

            // when
            underTest = parent
                .thatHasMicroservices()
121     }
```

Figura 3.21: *Método que roda antes de cada teste de unidade e atribui valor ao underTest após a correção*

Isso foi suficiente para garantir que o teste rode corretamente.

Por outro lado, como visto na figura exposta anteriormente, o underTest do teste de unidade pré-existente possui um parent, já que ele é passado como argumento no construtor do MicroserviceBuilder. No entanto, o CompanySystemBuilder não tem o MutableSet subsystems instanciado, pois o método que faz isso (thatHasMicroservices) nunca é chamado, causando o erro listado em parágrafos anteriores. Por isso, a tratativa foi adicionada nesse teste, muito semelhante ao que foi feito no método anotado com BeforeTest.

```
132    @Test
133    fun `it returns its parent when closing a properly opened microservice env`() {
134        // given
135        val parent = spyk(SystemBuilder())
136        underTest = parent
137            .thatHasMicroservices()
138
139        // when
140        val environment = underTest
141            .named("something")
142            .endMicroservices()
143
144        // then
145        assertIs<SystemBuilder>(environment)
146        assertEquals(parent, environment)
147        verify { parent.addMicroservice(any()) }
148    }
```

Figura 3.22: *Código de teste pré-existente após ajuste*

3.5 Início do módulo de criação utilizando Command e o Builder pré-existente

Após ajustes nos builders e em seus testes unitários, o SystemBuilder foi renomeado para CompanySystemBuilder, pois o método build desta classe sempre retornava um CompanySystem. Além disso, um Microservice também é um System, o que poderia levar à confusão em relação ao que este Builder pode construir. O nome SystemBuilder passou a ser utilizado para uma interface para o CompanySystemBuilder e MicroserviceBuilder.

Com todas as modificações necessárias feitas, começou-se a elaboração do módulo de criação. Inicialmente, cada etapa da construção de um sistema utilizando os builders seria um comando diferente. Haveria um comando para o início da criação do CompanySystem, que implicava na instanciação do CompanySystemBuilder, outro comando para dar nome ao CompanySystem, que implicava em uma chamada para o método setName no builder instanciado e assim por diante. Portanto, foi-se utilizado o padrão Command (Gamma *et al.*, 1994), pois ele é o mais adequado para essas situações, já que separa, cada comando em uma instância diferente, além de dar suporte à criação de um histórico de comandos executados.

No contexto do nosso projeto, o Command possui um nome para representar qual comando é executado, possui uma sessão para identificar qual sistema está sendo construído naquele comando e por, fim, possui um parâmetro relativo a cada comando (no caso do comando para nomear o CompanySystem, ele seria o nome a ser atribuído ao CompanySystem). Cada implementação concreta do Command, possui um commandName relativo a ele e uma implementação do execute que instancia e/ou chama o método do builder.

O CommandInvoker possui uma lista de comandos que podem ser executados. O método público InvokeCommand recebe o nome do comando, o parâmetro e a sessão. Esses parâmetros são repassados para o método getCorrectCommand, que cria a instância do comando. A sessão é utilizada para recuperar o histórico de comandos realizados através do CommandHistoryDataProvider, sendo executados novamente em sequência,

acrescido à execução da nova instância do Command. Por fim, o comando é salvo no commandHistoryDataProvider, permitindo o processo se repetir múltiplas vezes, até que o sistema seja criado.

A figura a seguir ilustra os atributos e métodos de cada classe. Para simplificar a visualização, a imagem só possui algumas das implementações do Command, mas outros comandos (um para cada etapa de construção via builder) também são essenciais para o módulo de criação utilizando a abordagem do Command com Builder.

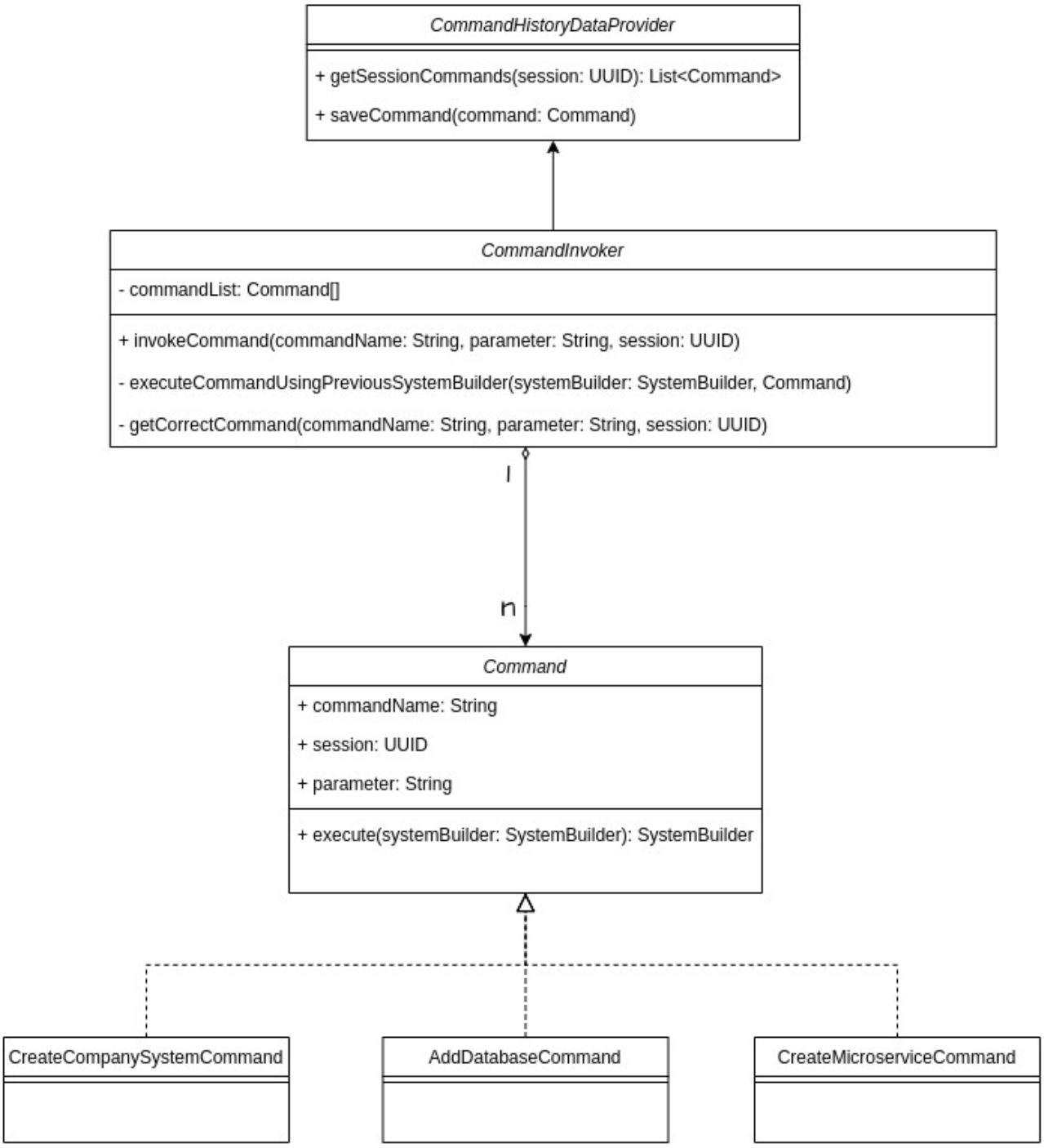

Figura 3.23: *Diagrama UML com o padrão Command*

Capítulo 4

Incompatibilidade do builder com o módulo de criação

No capítulo anterior, foi discutido o uso do padrão builder em conjunto com o padrão command. Com eles, os builders serão instanciados múltiplas vezes, bem como a execução dos seus métodos. A instanciação de um Microservice ou de um CompanySystem só serão concluídas ao invocar o Command que chama o método build. Esses fatores tornam a construção do sistema mais complexa.

O armazenamento do histórico de construção, em vez do objeto construído em si, também é um desperdício de processamento e de recurso, já que os builders e as classes possuem atributos muito parecidos, como mostram as figuras a seguir:

```
@Serializable  João Daniel
data class Microservice(
    override val name: String,
    var module: Module = Module.createWithId()
) : SystemOfComponents {
    private val exposedOperations: MutableSet<Operation> = mutableSetOf()
    private val consumedOperations: MutableSet<Operation> = mutableSetOf()
    private val databases: MutableSet<Database> = mutableSetOf()
    private val publishChannels: MutableSet<MessageChannel> = mutableSetOf()
    private val subscribedChannels: MutableSet<MessageChannel> = mutableSetOf()

```

Figura 4.1: *Atributos do Microservice*

```
class MicroserviceBuilder(private val parent: CompanySystemBuilder? = null): SystemBuilder  {  João Daniel +1
    private var name: String? = null
    private val exposedOperations = mutableSetOf<Operation>()
    private val consumedOperations = mutableSetOf<Operation>()
    private val databases = mutableSetOf<Database>()
    private val channelsPublished = mutableSetOf<MessageChannel>()
    private val channelsSubscribed = mutableSetOf<MessageChannel>()

```

Figura 4.2: *Atributos do MicroserviceBuilder*

Figura 4.3: *Atributos do CompanySystem*

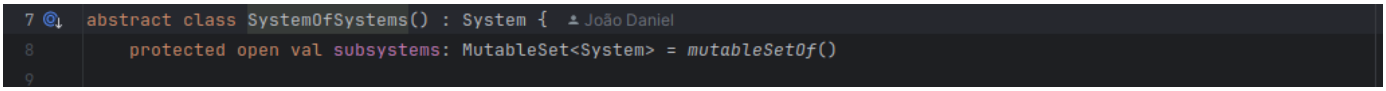

Figura 4.4: *Atributos do SystemOfSystems, implementado pelo CompanySystem*

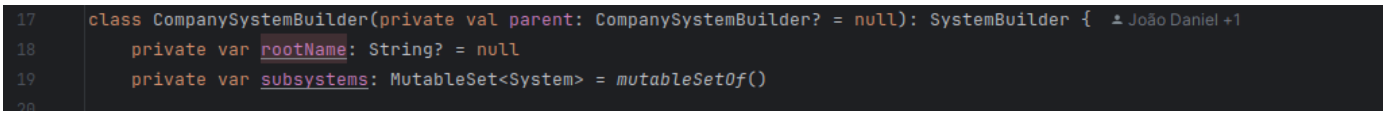

Figura 4.5: *Atributos do CompanySystemBuilder*

Além dos aspectos expostos nos parágrafos anteriores, também pode-se dizer que o caso de uso do builder é a construção de um objeto complexo cuja classe possui um construtor que recebe muitos parâmetros ou vários deles são opcionais (o que pode acarretar múltiplos construtores, dependendo de como será feita a modelagem). O builder permite que cada valor atribuído ao objeto fique explícito no código, o que não acontece no construtor do Java, por exemplo. Segue um exemplo de como o padrão builder pode melhorar o código:

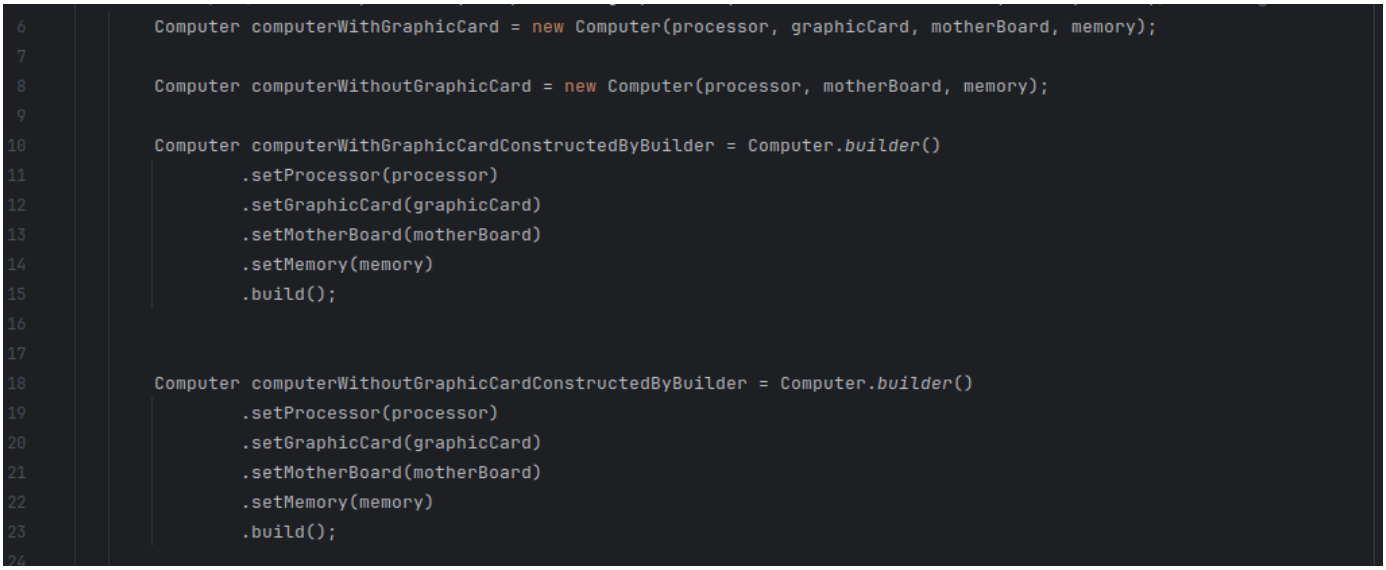

Figura 4.6: *Exemplo de uso do padrão builder*

Neste exemplo em Java, as duas primeiras instanciações de um objeto da classe Computer, utiliza seus diferentes construtores: um que passa um GraphicCard como parâmetro, outro não. Os nomes dos parâmetros necessários para instanciar um Computer não são

explícitos no código. Por outro lado, as duas últimas instanciações utilizam o builder do Computer, o que melhora a legibilidade do código para quem for construir o Computer, já que os métodos como setProcessor e setMotherboard deixam explícitos quais os nomes dos atributos a terem seu valor atribuído. Em vez de optar por um dos dois construtores, utilizando o builder, basta não invocar o método setGraphicCard caso esse campo não seja necessário.

A linguagem de programação Java foi escolhida para dar o exemplo anterior porque em Kotlin, o builder perde sua principal utilidade, pois instanciar um objeto com muitos campos passou a ser feito de forma mais legível, via construtor com parâmetros nomeados.

```
CompanySystem(name = "pingr").apply {
    addSubsystem(
        CompanySystem(name = "backend").apply {
            addSubsystem(
                Microservice(name = "pings").apply {
                    exposeOperation(
                        RestEndpoint(
                            httpVerb = "POST",
                            path = "/users/{uid}/pings",
                            description = "create new ping"
                        )
                    )
                    addPublishChannel(
                        MessageChannel(
                            name = pingsTopicName,
                            id = pingsTopicId
                        )
                    )
                }
            )
            addSubsystem(
                Microservice(name = "timeline").apply {
                    exposeOperation(
                        RestEndpoint(
                            httpVerb = "GET",
                            path = "/users/{uid}/timeline",
                            description = "the user's timeline"
                        )
                    )
                    addPublishChannel(
                        MessageChannel(
                            name = pingsTopicName,
                            id = pingsTopicId
                        )
                    )
                }
            )
        }
    )
}
```

Figura 4.7: *Exemplo de instanciação do sistema sem a utilização do builder*

No da figura anterior, um dos casos de teste do CompanySystemBuilder é recriado, só que sem o uso do builder. Todos os parâmetros obrigatórios no construtor, podem ser

passados ou não via parâmetro nomeado. No caso, o construtor do Microservice e do CompanySystem possuem o campo “name” como obrigatório. Com esse campo atribuído, uma instância de cada um dessas classes já está feita, bastando chamada de métodos para adicionar subsistemas, no caso do CompanySystem e chamada de métodos para adicionar operações e canais de mensageria, no caso de Microservice. É importante ressaltar que, até o RestEndpoint e o MessageChannel estão sendo instanciados sem nenhum builder, mas com seus parâmetros sendo explicitados no código via parâmetros nomeados.

Um questionamento que pode ser feito é em relação aos parâmetros opcionais neste contexto. O atributo id no MessageChannel é opcional com valor padrão null. Nesta ocasião, pode-se tanto passar o valor do id explícitamente em seu construtor, ou simplesmente não o passar para ele assumir o valor padrão.

Portanto, pode-se concluir que o builder cria uma complexidade desnecessária para o projeto, pois a própria sintaxe do Kotlin e métodos pré-existentes na modelagem do usVision são suficientes para a construção de um sistema e já resolvem o principal problema que o builder se propõe a solucionar.

Capítulo 5

Uma nova abordagem para o módulo de criação

Como usar o padrão builder criou uma complexidade desnecessária para a criação dos microsserviços, pensou-se em uma abordagem mais simples. Em vez de construir instâncias de sistemas ou microsserviços por partes e construí-las como no builder, a nova abordagem busca receber o sistema já instanciado e salvá-lo em seus métodos. Com ele, também é possível editar sistemas já existentes, adicionando subsistemas. No caso de microsserviços, também é possível editá-lo, adicionando banco de dados, operações, canais de mensageria, entre outros atributos. Criou-se um módulo denominado app-creation para esta tarefa.

A principal classe deste módulo é o SystemCreator, que possui diversos métodos com essa responsabilidade. O código a seguir ilustra os métodos para a criação das instâncias de Microservice e CompanySystem.

```
typealias CompanySystemDTO = CompanySystem
typealias SystemDTO = System
typealias MicroserviceDTO = Microservice
typealias DatabaseDTO = Database

class SystemCreator(  Vinicius Frota
    private val systemAggregateStorage: SystemAggregateStorage
) {

    fun createCompanySystem(companySystem: CompanySystemDTO, fatherSystemName: String? = null): CompanySystemDTO {
        checkIfSystemAlreadyExists(companySystem.name)

        fatherSystemName?.also {
            val fatherSystem = systemAggregateStorage.getCompanySystem(fatherSystemName)

            return@createCompanySystem fatherSystem?.let {
                fatherSystem.addSubsystem(companySystem)
                systemAggregateStorage.save(fatherSystem)
            } ?: throw Exception("A System Of Systems with name $fatherSystemName does not exist")
        }

        return systemAggregateStorage.save(companySystem)
    }

    fun createMicroservice(microservice: MicroserviceDTO, fatherSystemName: String? = null): SystemDTO {  Vinicius Frota
        checkIfSystemAlreadyExists(microservice.name)

        fatherSystemName?.also {
            val fatherSystem = systemAggregateStorage.getCompanySystem(fatherSystemName)

            return@createMicroservice fatherSystem?.let {
                fatherSystem.addSubsystem(microservice)
                systemAggregateStorage.save(fatherSystem)
            } ?: throw Exception("A System Of Systems with name $fatherSystemName does not exist")
        }

        return systemAggregateStorage.save(microservice)
    }
```

Figura 5.1: *Classe SystemCreator e seus métodos de criação de sistemas*

Os métodos createCompanySystem e createMicroservice funcionam de uma maneira muito parecida: a única diferença são os tipos que elas recebem e devolvem. Elas sempre começam checando se um sistema de mesmo nome já existe, através do método auxiliar checkIfSystemAlreadyExists. Se não existir, uma exceção é lançada. Depois, é feita uma checagem: se o parâmetro do nome do pai do Microservice ou CompanySystem foi passado como parâmetro e não é nulo, o sistema pai é buscado, a instância atual é adicionado como subsistema deste pai e por fim, tudo é salvo através do método save do systemAggregateStorage. Se o parâmetro do nome do pai do Microservice ou CompanySystem é nulo, a instância recebida pelo método é simplesmente salva utilizando o mesmo método do systemAggregateStorage.

O código a seguir ilustra os métodos responsáveis por adicionar bancos de dados, operações e canais de mensageria aos microsserviços:

```
49      fun addNewDatabaseConnectionToMicroservice(  Vinicius Frota
50          database: DatabaseDTO,
51          microserviceName: String
52      ) : Microservice  = getExistingMicroservice(microserviceName).let {
53          it.addDatabaseConnection(database)
54          systemAggregateStorage.save(it)
55      }
56
57      fun addOperationsToMicroservice(  Vinicius Frota
58          exposedOperations: List<Operation>,
59          consumedOperations: List<Operation>,
60          microserviceName: String
61      ) : Microservice  = getExistingMicroservice(microserviceName).let {
62          exposedOperations.forEach {
63              operation ->  it.exposeOperation(operation)
64          }
65
66          consumedOperations.forEach {
67              operation -> it.consumeOperation(operation)
68          }
69
70          systemAggregateStorage.save(it)
71      }
72
73      fun addMessageChannelsToMicroservice(  Vinicius Frota
74          publishMessageChannels: List<MessageChannel>,
75          subscribedMessageChannels: List<MessageChannel>,
76          microserviceName: String
77      ) : Microservice  = getExistingMicroservice(microserviceName).let {
78          publishMessageChannels.forEach {
79              operation ->  it.addPublishChannel(operation)
80          }
81
82          subscribedMessageChannels.forEach {
83              operation ->  it.addSubscribedChannel(operation)
84          }
85
86          systemAggregateStorage.save(it)
87      }
```

Figura 5.2: *Métodos responsáveis por adicionar bancos de dados, operações e canais de mensageria aos microsserviços*

O método auxiliar getExistingMicroservice busca um Microservice existente através de seu nome, e adiciona os atributos pelo qual o método chamado é responsável ao microsserviço, e salva através do método save do systemAggregateStorage. Escolheu-se passar listas aos métodos de canais de mensageria e operações, porque é relativamente comum existir microsserviços que são inscritos e publicam em mais de um canal de mensageria, assim como é comum um microsserviço expor ou consumir mais de uma operação. No entanto, raramente um microsserviço consumirá mais de um banco de dados, portanto, o método de adição recebe uma instância só de Database.

É importante ressaltar que é possível passar sistemas e microsserviços inteiros, inclusive com seus bancos de dados, canais de mensageria e operações através dos métodos createCompanySystem e createMicroservice. No entanto, os métodos de adição desses

atributos foram adicionados para tornar este módulo mais versátil, apto para atender futuras modificações dos microsserviços que já existiam no banco de dados.

O código a seguir ilustra os métodos privados, que servem como métodos auxiliares:

```
89      private fun getExistingMicroservice(  Vinicius Frota
90          microserviceName: String
91      ) : Microservice  = systemAggregateStorage.getMicroservice(microserviceName)
92          ?: throw Exception("A Microservice with name $microserviceName does not exist")
93
94      private fun checkIfSystemAlreadyExists(name: String) {  Vinicius Frota
95          systemAggregateStorage.getSystem(name)?.also {
96              throw Exception("A system with name $name already exists")
97          }
98      }
```

Figura 5.3: *Métodos auxiliares do SystemCreator*

Um ponto importante é que o método checkIfSystemAlreadyExists busca tanto Microservices como também CompanySystems através do método getSystem do systemAggregateStorage.

O SystemAggregateStorage não possui implementação concreta, ela é apenas uma interface para métodos que buscam ou salvam os Microservices e CompanySystems. Esta decisão foi tomada para ela ser extensível para qualquer implementação. Atualmente, ela foi implementada no módulo de persistência que utiliza MongoDB, mas por ser uma interface, poderia ser utilizado outro módulo ou outra implementação que utiliza PostgreSQL, por exemplo. O código a seguir ilustra as assinaturas dos métodos dessa interface:

```
7   interface SystemAggregateStorage {  Vinicius Frota
8       fun getSystem(name: String): System?  Vinicius Frota
9       fun save(microservice: Microservice): Microservice  Vinicius Frota
10      fun save(companySystem: CompanySystem): CompanySystem  Vinicius Frota
11      fun getCompanySystem(name: String): CompanySystem?  Vinicius Frota
12      fun getMicroservice(name: String): Microservice?  Vinicius Frota
13  }
```

Figura 5.4: *Interface SystemAggregateStorage*

A interface SystemAggregateStorage possui esse nome por utilizar o padrão Aggreate do Domain Driven Design (Evans, 2014) (Vernon, 2013). Os métodos para recuperar e salvar os sistemas, lidam com agregados inteiros: ao salvar um microsserviço, estamos salvando também suas operações, banco de dados e canais de mensageria. Analogamente, ao salvar um CompanySystem, estamos salvando também seus subsistemas, que podem ser microsserviços (com operações e outros atributos citados anteriormente) ou até outros CompanySystems.

Após a criação do SystemCreator, bem como a criação da interface SystemAggregateStorage, foram criados teste de unidade, muito parecido com os que foram feitos nos builders em uma classe chamada SystemCreatorTest. O código a seguir é um exemplo de teste unidade criado.

```
10  typealias CompanySystemDTO = CompanySystem
11  typealias SystemDTO = System
12  typealias MicroserviceDTO = Microservice
13  typealias DatabaseDTO = Database
14
15  class SystemCreator(  Vinicius Frota
16      private val systemAggregateStorage: SystemAggregateStorage
17  ) {
18
19      fun createCompanySystem(companySystem: CompanySystemDTO, fatherSystemName: String? = null): CompanySystemDTO {  Vini
20          checkIfSystemAlreadyExists(companySystem.name)
21
22          fatherSystemName?.also {
23              val fatherSystem = systemAggregateStorage.getCompanySystem(fatherSystemName)
24
25              return@createCompanySystem fatherSystem?.let {
26                  fatherSystem.addSubsystem(companySystem)
27                  systemAggregateStorage.save(fatherSystem)
28              } ?: throw Exception("A System Of Systems with name $fatherSystemName does not exist")
29          }
30
31          return systemAggregateStorage.save(companySystem)
32      }
33
34      fun createMicroservice(microservice: MicroserviceDTO, fatherSystemName: String? = null): SystemDTO {  Vinicius Frota
35          checkIfSystemAlreadyExists(microservice.name)
36
37          fatherSystemName?.also {
38              val fatherSystem = systemAggregateStorage.getCompanySystem(fatherSystemName)
39
40              return@createMicroservice fatherSystem?.let {
41                  fatherSystem.addSubsystem(microservice)
42                  systemAggregateStorage.save(fatherSystem)
43              } ?: throw Exception("A System Of Systems with name $fatherSystemName does not exist")
44          }
45
46          return systemAggregateStorage.save(microservice)
47      }
```

Figura 5.5: *Exemplo de teste de unidade para o SystemCreator*

Capítulo 6

Mudanças no módulo de persistência

Para atender as necessidades do módulo de criação, o MongoSystemRepository passou a implementar também a interface SystemAggregateStorage. O código a seguir ilustra isso.

```
class MongoSystemRepository(db: MongoDatabase) : SystemRepository, SystemAggregateStorage
```

Figura 6.1: *Interfaces implementadas pelo MongoSystemRepository*

Como citado anteriormente, o SystemRepository já era implementado por essa classe, pois o SystemRepository é uma interface do módulo de relatórios. Os códigos a seguir mostram cada uma dessas implementações.

```
override fun getSystem(name: String): System?  = runBlocking {  Vinicius Frota +1
    systemCollection
        .find(Filters.eq( fieldName: "name", name))
        .firstOrNull()
        ?.let { SystemMapper.fromDocument(it) }
}


override fun save(companySystem: CompanySystem): CompanySystem = runBlocking {  Vinicius Frota +1
    val insertedId = systemCollection.insertOne(
        companySystem.toSystemDocument()
    ).insertedId ?: throw SystemNotFoundException(companySystem.name)

    getSystemById(insertedId.asObjectId().value) as CompanySystem
}

override fun save(microservice: Microservice): Microservice = runBlocking {  Vinicius Frota
    val insertedId = systemCollection.insertOne(
        microservice.toSystemDocument()
    ).insertedId ?: throw SystemNotFoundException(microservice.name)

    getSystemById(insertedId.asObjectId().value) as Microservice
}

override fun getCompanySystem(name: String): CompanySystem? = try {  Vinicius Frota
    getSystem(name = name) as CompanySystem?
} catch (ex: ClassCastException) {
    null
}

override fun getMicroservice(name: String): Microservice? = try {  Vinicius Frota
    getSystem(name = name) as Microservice?
} catch (ex: ClassCastException) {
    null
}
```

Figura 6.2: *Métodos de get do MongoSystemRepository*

```
    private fun getSystemById(id: ObjectId): System?  = runBlocking {
        systemCollection
            .find(Filters.eq( fieldName: "_id", id))
            .firstOrNull()
            ?.let { SystemMapper.fromDocument(it) }
    }
}
```

Figura 6.3: *Método privado auxiliar do MongoSystemRepository*

Cada um dos métodos save converte um objeto de domínio para um objeto de documento do MongoDB e depois faz uma inserção no banco de dados através do systemCollection. Após a inserção, o objeto criado é recuperado via método auxiliar getSystemById e retornado.

Os métodos de get fazem a consulta, recuperam o dado e convertem ele do objeto de documento para o objeto de domínio e retornando-o. Os getters específicos de CompanySystem e Microservice fazem o casting para a classe correta e, caso não seja, possível, há um tratamento para esta exceção que faz eles retornarem nulo.

O mapeamento dos objetos de documento para os objetos de domínio já existia e estava na classe SystemMapper. No entanto, o mapeamento dos objetos da classe de domínio para a classe de documento não existia. Para resolver isso, no próprio arquivo que possui as classes de documento (SystemDocument.kt), foram criados funções de extensão para cada uma das classes. O código a seguir mostra algumas dessas implementações.

```
fun CompanySystem.toSystemDocument(  Vinicius Frota
    id: ObjectId = ObjectId()
): SystemDocument = SystemDocument(
    id = id,
    name = this.name,
    subsystems = this.getSubsystemSet().toSystemDocumentSet()
)

fun Microservice.toSystemDocument(  Vinicius Frota
    id: ObjectId = ObjectId()
): SystemDocument = SystemDocument(
    id = id,
    name = this.name,
    module = this.module.toModuleDocument(),
    databases = this.getDatabases().toDatabaseDocumentSet(),
    exposedOperations = this.getExposedOperations().toOperationDocumentSet(),
    consumedOperations = this.getConsumedOperations().toOperationDocumentSet(),
    publishedChannels = this.getPublishChannels().toMessageChannelDocumentSet(),
    subscribedChannels = this.getSubscribedChannels().toMessageChannelDocumentSet(),
)

private fun System.toSystemDocument(  Vinicius Frota
    id: ObjectId = ObjectId()
) : SystemDocument = when (this) {
    is CompanySystem -> {
        this.toSystemDocument(id)
    }
    is Microservice -> {
        this.toSystemDocument(id)
    }
    else -> {
        throw UnknownSystemClassException(this.name, "SystemDocument")
    }
}
```

Figura 6.4: *Funções de extensão que convertem classe de domínio para classe que representa um documento do MongoDB*

Por fim, a interface DBRepositoryProvider, que antes possuía apenas um método para fornecer um SystemRepository, passou a possuir um método para fornecer o SystemAggregateStorage. Como o MongoSystemRepository implementa essas duas interfaces, bastou retorná-lo como no getRepository na implementação concreta dessa interface.

```
interface DBRepositoryProvider : DBConnectionBuilder {  João Daniel +1
    fun getRepository(): SystemRepository  João Daniel
    fun getAggregateStorage(): SystemAggregateStorage  Vinicius Frota
}
```

Figura 6.5: *Interface DBRepositoryProvider*

Capítulo 7

Integração do módulo de criação ao módulo REST

O método responsável por carregar o módulo app-reports também carregava o app-persistence. No entanto, foi preciso delegar a lógica de instanciação das classes do módulo de persistência para outro método, responsável só por isso. Essa decisão foi tomada porque as dependências de persistência passaram a ser utilizadas tanto pelo app-reports, como também pelo módulo de criação. As figuras a seguir mostram o código responsável por carregar o app-reports antes e depois da modificação:

```
fun Application.configureReports(): ReportSupervisor {
    val host = environment.config.property("persistence.host").getString()
    val port = environment.config.property("persistence.port").getString()
    val user = environment.config.property("persistence.username").getString()
    val pass = environment.config.property("persistence.password").getString()
    val dbName = environment.config.property("persistence.database_name").getString()

    val presets = parsePresetsConfig(environment.config.config("reports.presets"))

    val repoProvider: DBRepositoryProvider = MongoDBRepositoryProvider()

    val systemRepository = repoProvider.run {
        connectTo(host)
        setPort(port)
        withCredentials(user, pass)
        setDatabase(dbName)
        getRepository()
    }

    return ReportSupervisor(
        systemRepository,
        presets = presets
    )
```

Figura 7.1: *Método configureReports antes da configuração*

```
8
9   fun Application.configureReports(systemRepository: SystemRepository): ReportSupervisor {
10      val presets = parsePresetsConfig(environment.config.config("reports.presets"))
11
12      return ReportSupervisor(
13          systemRepository,
14          presets = presets
15      )
16  }
```

Figura 7.2: *Método configureReports depois da configuração*

Além da remoção das linhas responsáveis por receber variáveis de configuração relacionadas com a conexão com o banco de dados, o método passou a receber o SystemRepository como parâmetro.

O método responsável por carregar o módulo de persistência possui as mesmas instruções que o método configureReports possuía para esta mesma finalidade. No entanto, em vez de retornar apenas o resultado do método getRepository do DBRepositoryProvider, ela retorna um Pair com o resultado deste método e com o valor obtido através da chamada de função getAggregateStorage. Desta forma, o SystemRepository, o qual é um requisito do módulo de relatórios, e o SystemAggregateStorage, o qual é um requisito do módulo de criação, são carregados por meio do configureDatabaseConnection. O código a seguir ilustra como este método foi construído.

```
fun Application.configureDatabaseConnection(): Pair<SystemRepository, SystemAggregateStorage> {
    val host = environment.config.property( path: "persistence.host").getString()
    val port = environment.config.property( path: "persistence.port").getString()
    val user = environment.config.property( path: "persistence.username").getString()
    val pass = environment.config.property( path: "persistence.password").getString()
    val dbName = environment.config.property( path: "persistence.database_name").getString()

    val repoProvider: DBRepositoryProvider = MongoDBRepositoryProvider()

    return repoProvider.run {
        connectTo(host)
        setPort(port)
        withCredentials(user, pass)
        setDatabase(dbName)
        Pair(getRepository(), getAggregateStorage())
    }

}
```

Figura 7.3: *Método configureDatabaseConnection*

A função configureSystemCreator foi criado para instanciar o módulo de criação. Ela recebe um SystemAggregateStorage como argumento. O código dela é extremamente simples, como mostrado na figura a seguir.

```
fun Application.configureSystemCreator(    Vinicius Frota
    systemAggregateStorage: SystemAggregateStorage
) : SystemCreator = SystemCreator(systemAggregateStorage)
```

Figura 7.4: *Método configureSystemCreator*

O configureRouting, método responsável por configurar as rotas HTTP, deixou de apenas receber um reportSupervisor e passou a receber também o SystemCreator. O código a seguir mostra a assinatura deste método.

```
fun Application.configureRouting(reportSupervisor: ReportSupervisor, systemCreator: SystemCreator)
```

Figura 7.5: *Assinatura do método configureRouting*

Por fim, o método module é responsável por carregar todos os módulos do Ktor, e também passou por algumas modificações devido às mudanças das assinaturas dos outros métodos descritos anteriormente. A figura a seguir mostram o código deste método depois da inclusão do módulo de criação ao módulo REST.

```
fun Application.module() {
    val (databaseRepository, databaseAggregateStorage) = configureDatabaseConnection()
    val systemCreator = configureSystemCreator(databaseAggregateStorage)
    val reportSupervisor = configureReports(databaseRepository)

    configureCORS()
    configureSerialization()
    configureRouting(reportSupervisor, systemCreator)
    configureExceptionHandling()
}
```

Figura 7.6: *Método module*

Após a configuração dos módulos carregados pelo Ktor, as rotas foram configuradas no configureRouting. O código a seguir ilustra rotas adicionadas para a adição de novos microsserviços, banco de dados e endpoints REST.

```
routing {
    route( path: "/microservices") {
        post {
            val microservice = call.receive<SystemRequestDTO>().toMicroservice()
            val createMicroserviceResult = systemCreator.createMicroservice(microservice)

            call.respond(
                message = createMicroserviceResult.toSystemResponseDTO(),
                status = HttpStatusCode.Created
            )
        }
        post( path: "/{name}/databases") {
            val microserviceName: String = call.parameters["name"]
                ?: throw MissingRequiredPathParameterException("name", "String")

            val databaseDTO = call.receive<PostgreSQL>()
            val addDatabaseResult = systemCreator.addNewDatabaseConnectionToMicroservice(databaseDTO, microserviceName)

            call.respond(
                message = addDatabaseResult.toMicroserviceResponseDTO(),
                status = HttpStatusCode.Created
            )
        }

        post( path: "/{name}/rest-endpoints") {
            val microserviceName: String = call.parameters["name"]
                ?: throw MissingRequiredPathParameterException("name", "String")
            val restEndpoints = call.receive<RestEndpointsRequestDTO>()

            val addOperationsResult = systemCreator.addOperationsToMicroservice(
                exposedOperations = restEndpoints.exposedOperations,
                consumedOperations = restEndpoints.consumedOperations,
                microserviceName = microserviceName
            )

            call.respond(
                message = addOperationsResult.toMicroserviceResponseDTO(),
                status = HttpStatusCode.Created
            )
        }
```

Figura 7.7: *Rotas referentes aos microsserviços*

- A rota POST /microservices recebe um SystemRequestDTO e converte este objeto para uma instância de Microservice. Esta instância é utilizada no método createMicroservice do SystemCreator. A rota devolve o resultado do createMicroservice (instância de microsserviço) convertido para um SystemResponseDTO.
- Na rota POST /microservice/<nome-do-microsserviço>/databases, o corpo da requisição é fornecido pelo próprio modelo do PostgreSQL, presente no módulo app-model. Esta instância é repassada para o método addNewDatabaseConnectionToMicroservice do SystemCreator, juntamente com o nome do microsserviço. A rota devolve o resultado deste método (o microsserviço com o banco de dados adicionado), é convertido para um SystemResponseDTO.

- A rota POST /microservice/<nome-do-microsserviço>/rest-endpoints recebe um RestEndpointsRequestDTO, que possui os endpoints que o microsserviço consome e expõe. Esses endpoints são repassados para o método responsável pela adição dessa operações, juntamente com o nome do microsserviço. Por fim, o resultado deste método (o microsserviço com as operações adicionadas), é convertido para o SystemResponseDTO.
- A rota POST /microservice/<nome-do-microsserviço>/message-channels não está na figura, mas possui um comportamento muito parecido com a rota dos endpoints REST. No entanto, recebe possui um DTO de canais de mensageria em vez de endpoint REST.

Adicionalmente, o código a seguir ilustra a criação de CompanySystems e adição de microsserviço a um CompanySystem.

```
route( path: "/systems") {
    post {
        val companySystem = call.receive<SystemRequestDTO>().toCompanySystem()
        val createCompanySystemResult = systemCreator.createCompanySystem(companySystem)

        call.respond(
            message = createCompanySystemResult.toCompanySystemResponseDTO(),
            status = HttpStatusCode.Created
        )
    }

    post( path: "/{name}/microservices") {
        val systemName: String = call.parameters["name"]
            ?: throw MissingRequiredPathParameterException("name", "String")

        val microservice = call.receive<SystemRequestDTO>().toMicroservice()
        val createMicroserviceResult = systemCreator.createMicroservice(microservice, systemName)

        call.respond(
            message = createMicroserviceResult.toSystemResponseDTO(),
            status = HttpStatusCode.Created
        )
    }

    post( path: "/{name}/companySubsystems") {
        val systemName: String = call.parameters["name"]
            ?: throw MissingRequiredPathParameterException("name", "String")

        val companySystem = call.receive<SystemRequestDTO>().toCompanySystem()
        val createCompanySystemResult = systemCreator.createCompanySystem(companySystem, systemName)

        call.respond(
            message = createCompanySystemResult.toCompanySystemResponseDTO(),
            status = HttpStatusCode.Created
        )
    }
```

Figura 7.8: *Rotas referentes à criação de CompanySystems e adição de microsserviços a um CompanySystem*

- A rota POST /systems é muito parecida com a /microsservices, a diferença é que ela converte o corpo da requisição para CompanySystem e utiliza o método do SystemCreator específico para criação de CompanySystem.

- A rota POST /systems/<nome-do-sistema-pai>/microservices recebe um SystemRequestDTO e converte este objeto para uma instância de Microservice. Esta instância é utilizada no método createMicroservice do SystemCreator junto com o nome do sistema pai. A rota devolve o resultado do createMicroservice (instância de microsserviço) convertido para um SystemResponseDTO.

- A rota POST /systems/<nome-do-sistema-pai>/companySubsystems possui a mesma lógica, só que cria um subsistema que é um CompanySystem em vez de um Microservice.

O pacote com os DTOs possuem o corpo de requisição e de resposta esperado pelas rotas cadastradas:

- O SystemRequestDTO é utilizado tanto nas requisições para a criação de Microservice, como também nas requisições para a criação de CompanySystem. Todos os campos são não obrigatórios, exceto o campo de nome que é comum às duas classes. Ela conta com os métodos toCompanySystem() e toMicroservice(), que converte esta classe para classes de domínio do usVision.

- O MessageChannelDTO e o RestEndpointsDTO possuem listas de MessageChannel e RestEndpoint, classes presentes no app-model. Não foi necessário criar DTOs específicos para cada um desses modelos, pois eles já contavam a annotation @Serializable, responsável pela serialização e desserialização de corpos de resposta e requisição.

- Não foi criado um PostgreSQLDTO para a rota de adição de banco de dados, pois o PostgreSQL presente no app-model também já possuía a Annotation @Serializable.

- Foram adicionadas as classes CompanySystemResponseDTO e MicroserviceResponseDTO, os quais são os corpos de resposta das requisições. Elas implementam a interface SystemDTO. No arquivo em que essas classes e interfaces foram definidas, há uma função de extensão para o Microservice e CompanySystem para convertê-los para o DTO de resposta adequado.

Para testar as rotas criadas, foi criada uma classe chamada ApplicationRoutingTest. Esta classe contém algumas funções de extensão para facilitar a escrita dos testes.

```
    private suspend inline fun <reified T> HttpClient.postRequest(
        urlString: String,
        body: T
    ) : HttpResponse = this.post(urlString) {
        url {
            protocol = URLProtocol.HTTPS
            contentType(ContentType.Application.Json)
            setBody(body)
        }
    }

    private fun ApplicationTestBuilder.getHttpClient(): HttpClient {
        val client = createClient {
            install(ContentNegotiation) {
                json()
            }
        }
        return client
    }

    private fun ApplicationTestBuilder.loadUnitTestModules() {
        application {
            configureCORS()
            configureSerialization()
            configureRouting(reportSupervisor, systemCreator)
            configureExceptionHandling()
        }
    }

}
```

Figura 7.9: *Funções auxiliares para o teste das rotas criadas*

A função de extensão postRequest do HttpClient recebe uma url e o corpo da requisição e faz um POST via HTTPS na url, garantindo que o conteúdo enviado possui o tipo json. Já a função de extensão getHttpClient do ApplicationTestBuilder, cria um cliente que se comunica via json. Por fim, a função de extensão do ApplicationTestBuilder loadUnitTestModules carrega todos os módulos do Ktor necessários para o teste. Todas essas funções de extensão servem para que os códigos de teste não fiquem carregados de instruções de configuração do cliente HTTP, que atua como testador, e do servidor, que atua como código a ser testado.

```
internal class ApplicationRoutingTest {  Vinicius Frota
    private val systemCreator = mockk<SystemCreator>()
    private val reportSupervisor = mockk<ReportSupervisor>()

    @Test  Vinicius Frota
    fun `it creates a new microservice`() : Unit  = testApplication {
        loadUnitTestModules()
        val client = getHttpClient()

        val microserviceName = "microserviceName"

        val microserviceRequest = SystemRequestDTO(
            name = microserviceName
        )

        val createdMicroservice = Microservice(name = microserviceName)

        val expectedResponseBody = createdMicroservice.toMicroserviceResponseDTO()

        every { systemCreator.createMicroservice(any()) } returns createdMicroservice

        val response = client.postRequest(
            urlString: "/microservices",
            microserviceRequest
        )

        val responseBody: SystemResponseDTO = response.body()

        verify(exactly = 1) {
            systemCreator.createMicroservice(createdMicroservice)
        }

        assertEquals(HttpStatusCode.Created, response.status)
        assertEquals(expectedResponseBody, responseBody)
    }
```

Figura 7.10: *Exemplo de teste de unidade para o módulo REST*

O código mostrado anteriormente mostra a simulação das classes SystemCreator e ReportSupervisor sendo criadas, necessárias para o carregamento do configureRouting, seguido do teste da rota que cria um Microsserviço. Uma observação importante é que o testApplication é um método que fornece o contexto necessário para a instanciação do servidor para os testes.

Pode-se resumir o trabalho de integração do módulo de criação com o módulo REST

em um diagrama de alto nível:

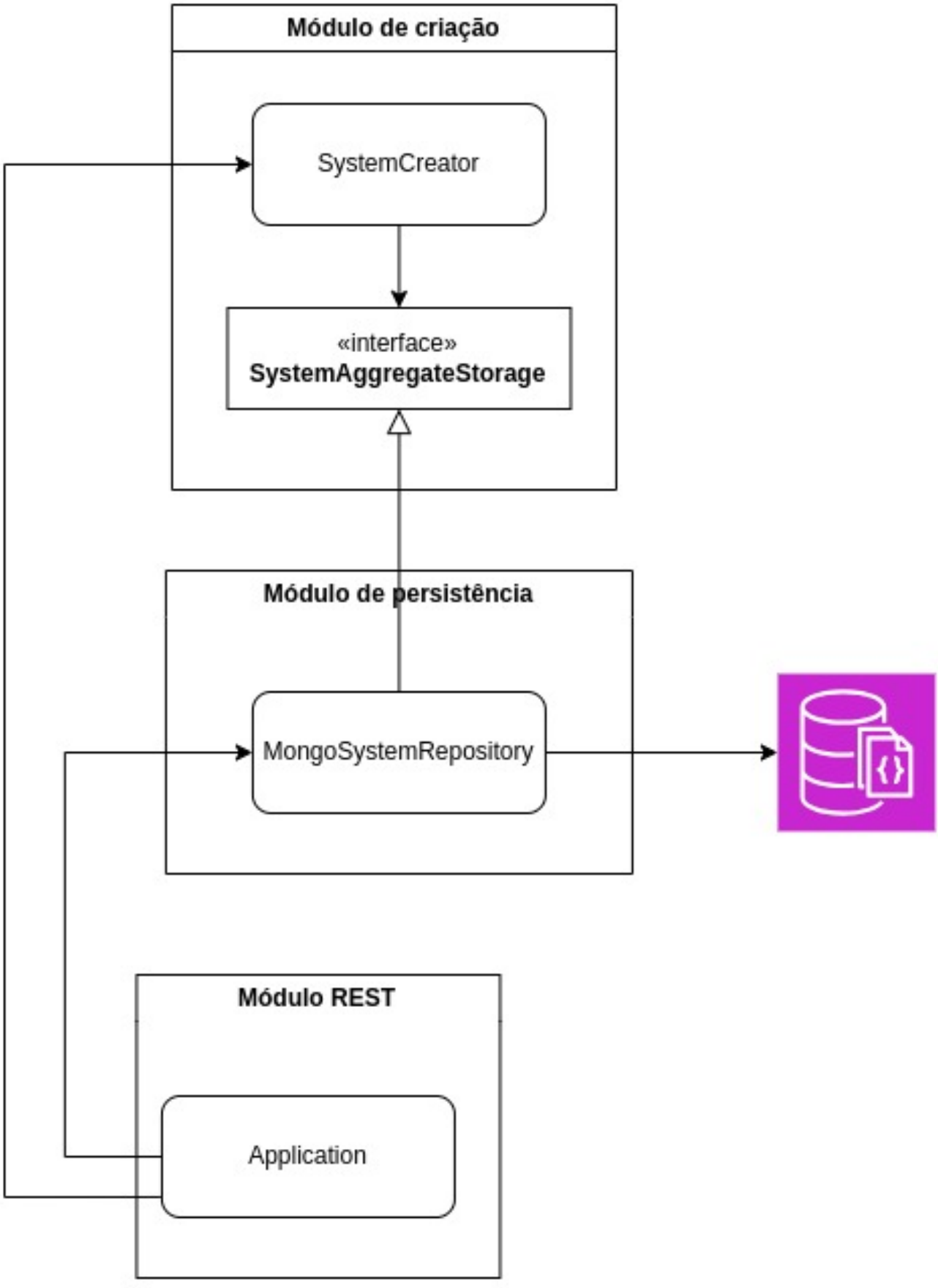
Módulo de criação
SystemCreator
«interface»
SystemAggregateStorage
Módulo de persistência
MongoSystemRepository
Módulo REST
Application

Figura 7.11

- O módulo REST tem como dependência o MongoSystemRepository, sendo uma implementação do SystemAggregateStorage, e o SystemCreator, que recebe este MongoSystemRepository em seu construtor.
- O SystemCreator possui como dependência uma implementação do SystemAggregateStorage.
- MongoSystemRepository se comunica com o banco de dados do usVision.

O trabalho realizado com o módulo REST é um exemplo de como o módulo de criação pode ser utilizado. Uma interface HTTP foi criada para o módulo de criação, mas qualquer outro módulo que utiliza outros protocolos de comunicação, como mensageria com Kafka/RabbitMQ, entre outros, poderia ser criado baseando-se no que foi feito no app-web. Também podemos dizer que o trabalho realizado é um exemplo de arquitetura limpa (Martin, 2017), o que permite extensibilidade.

Além disso, a existência do módulo REST permite que o Sorting Hat, que inspirou este trabalho, ou qualquer outro sistema se integre ao usVision a partir de requisições HTTP. É possível criar sistemas e microsserviços através dessas rotas que foram criadas e obter relatórios de padrões de microsserviços através das rotas que já existiam.

Capítulo 8

Trabalhos futuros

A figura a seguir mostra um diagrama com os trabalhos futuros que podem ser feitos a partir do que foi desenvolvido.

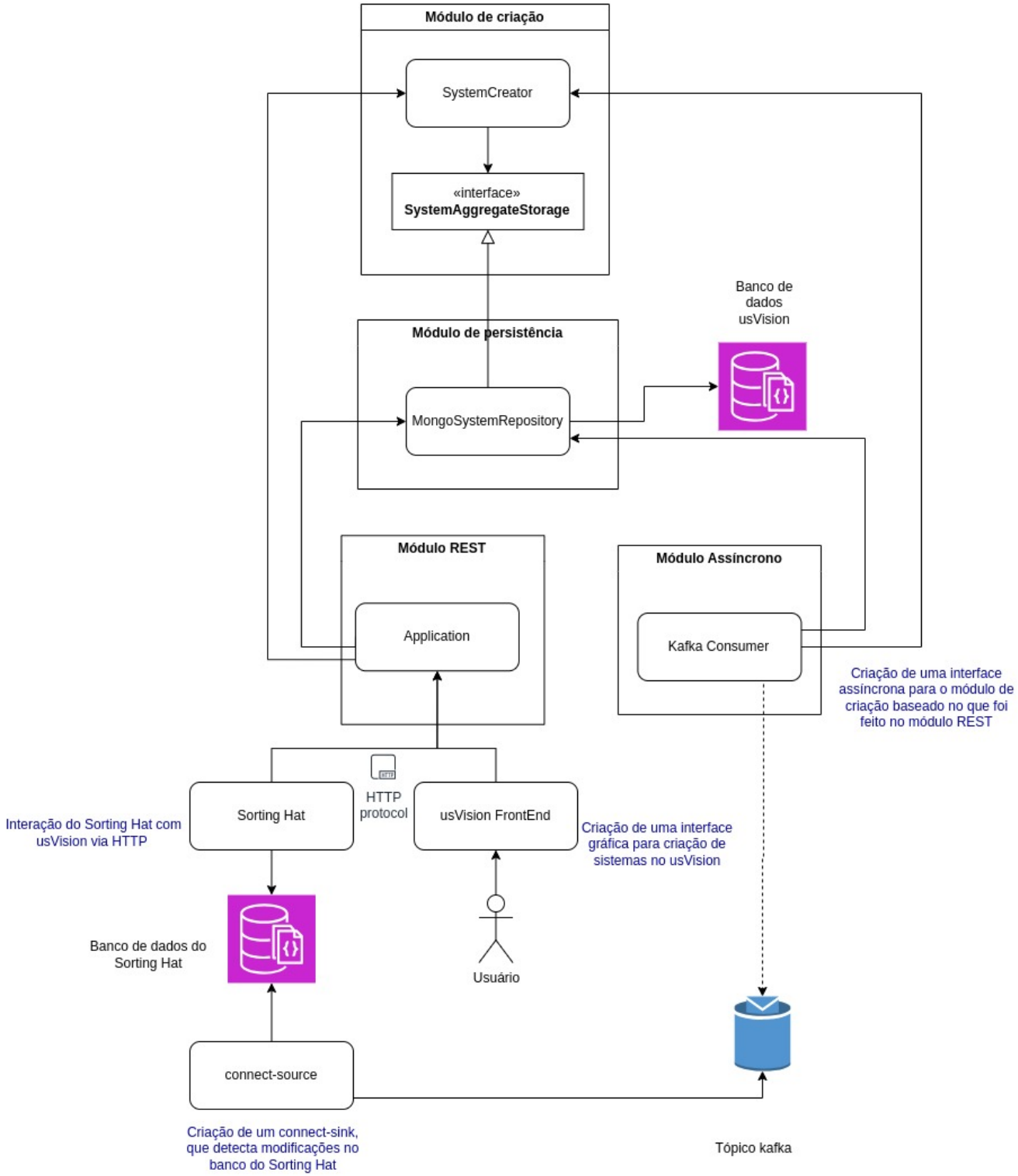

Figura 8.1: *Diagrama contendo as possibilidades*

O Sorting Hat, ferramenta que lê as informações dos microsserviços de um sistema por meio de repositórios no GitHub a partir de heurísticas e análises do Docker e de arquivos

de documentação com a especificação do openAPI, possui uma modelagem muito parecida com a do usVision. Portanto, é possível que o Sorting Hat se integre a ele via HTTP através da interface REST para o módulo de criação feita neste trabalho, populando o banco de dados do usVision. Dependendo das necessidades do Sorting Hat, é possível também utilizar as rotas HTTP já pré-existentes para a geração de relatórios desses sistemas com arquiteturas de microsserviços que foram criados e inseridos no banco do usVision.

Atualmente, o usVision não possui uma interface gráfica. A partir da interface REST para o módulo de criação, é possível criar um frontend que permite o usuário criar e inserir sistemas que utilizam a arquitetura de microsserviços no banco de dados do usVision. Adicionalmente, este frontend também poderia utilizar as rotas HTTP pré-existentes para a geração de relatórios desses sistemas que foram inseridos.

Com o exemplo do que foi feito no módulo REST, também poderia ser criado um módulo de comunicação assíncrona que utiliza Kafka. Cada classe com um consumidor de um tópico do Kafka poderia ter uma referência ao SystemCreator. Como o SystemCreator depende de um SystemAggregateStorage, é preciso que este módulo também instancie o repositório MongoDB ou outro repositório que siga a interface SystemAggregateStorage. Assim como houve a criação de DTOs para o módulo REST, bem como seus mapeamentos para classes de domínio, essa mesma etapa também aconteceria aqui, com a diferença que os DTOs seriam substituídos por classes que representam mensagens do Kafka.

Com o módulo de comunicação assíncrona com Kafka feito, também é possível integrá-lo ao Sorting Hat sem o uso de comunicação HTTP e sem que o Sorting Hat precise desenvolvimento. Para isso, é preciso criar um serviço com Kafka Connect, ferramenta do Kafka que interage com tópicos e banco de dados. Mais especificamente, a abordagem pensada envolve em utilizar um connect-source, instância do Kafka Connect que lê registros do banco de dados, para ler os dados do Sorting Hat e publicar em um tópico. Este tópico, por sua vez, é lido pelo módulo assíncrono. Um ponto importante sobre essa abordagem, é que o serviço do connect source também precisaria do acesso ao banco de dados do Sorting Hat.

Capítulo 9

Conclusão

Antes deste trabalho, era necessário popular o banco de dados do usVision diretamente via queries de inserção. Isto era uma má prática, pois requeria que o usuário final tenha acesso direto ao banco de dados. Além disso, o MongoDB, banco de dados utilizado pelo projeto, é orientado a documentos. Isso significa que o usuário poderia inserir dados inconsistentes, já que não há schema definido. Com o novo módulo de criação e sua exposição através do módulo REST, o usuário consegue fazer requisições para o usVision e popular o banco de dados, sem a necessidade de possuir acesso direto a ele. Além disso, a camada REST protege o banco de dados inconsistentes, pois se o usuário faz uma requisição com um corpo de requisição não esperado, o servidor do usVision responde com erro e não popula o banco de dados.

A criação do módulo REST pode servir de passo a passo para a criação de outros módulos que utilizam o módulo de criação, mas fazem o uso de outros protocolos. As etapas serão muito parecidas com o que já foi feito: instanciação do repositório MongoDB (ou outro repositório que siga a interface SystemAggregateStorage), instanciação do SystemCreator e por fim, mapeamento das classes relacionadas com o protocolo de comunicação para o domínio e vice-versa.

Ao utilizar um padrão de projeto, é preciso considerar se ele serve para o caso de uso e até se a linguagem de programação utilizada o torna desnecessário. A primeira tentativa da criação do módulo de criação envolveu o uso dos padrões builder e command, mas foram descontinuados pelo aumento da complexidade de arquitetura que isso iria trazer, bem como aumento do uso do processamento e de memória pelo usVision. O fato do Kotlin possuir parâmetros nomeados torna o builder desnecessário. O command é mais utilizado para guardar um histórico de comandos ou quando os comandos são utilizados em mais de um lugar, o que não era requisito para este trabalho. O uso do padrão aggregate, por outro lado, facilitou o desenvolvimento do módulo de criação, tornando a arquitetura muito simples.

Por fim, este trabalho abre possibilidades para projetos futuros. Os desenvolvedores do Sorting Hat pode utilizar as rotas criadas para se comunicar com o usVision, criando sistemas inteiros e gerando relatórios. Os desenvolvedores do usVision conseguem fazer um frontend, fornecendo uma interface gráfica para esta ferramenta utilizando essas

mesmas rotas.

Referências

[Daniel *et al.* 2023] João Daniel, Eduardo Guerra, Thatiane Rosa e Alfredo Goldman. "Towards the detection of microservice patterns based on metrics" (2023) (citado na pg. 1).

[Evans 2014] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 2014 (citado na pg. 28).

[Gamma *et al.* 1994] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 1994 (citado nas pgs. 6, 18).

[Martin 2017] Robert C. Martin. *Clean Architecture : a craftsman's guide to software structure and design*. Prentice Hall, 2017 (citado na pg. 49).

[Santana *et al.* 2021] Erick Rodrigues de Santana, Thatiane de Oliveira Rosa, Joao Francisco Lino Daniel e Alfredo Goldman. "Desenvolvendo o sorting hat: uma ferramenta para caracterização de arquitetura baseada em serviços" (2021) (citado na pg. 1).

[Vernon 2013] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013 (citado na pg. 28).