Arquitetura de Microsserviços para um Sistema de Gerenciamento de Certificados

Matheus C. Andrade[1], Alexandro S. Silva[1], Pablo F. Matos[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA)
Av. Sérgio Vieira Melo, 3.150, Zabelê – 45.078-900 – Vitória da Conquista – BA – Brasil

matheusbdo1@gmail.com, {alexandrossilva, pablofmatos}@ifba.edu.br

Abstract. *In view of the variety of events that take place today, there was a need for online integration between participants and the organizing committee. For this, websites are used that gather user data, information about events and other functionalities, one of which is the generation of certificates. The Federal Institute of Bahia Campus of Vitória da Conquista has a certificate management system, developed in an architecture that is currently obsolete due to limitations in terms of scalability, performance and difficulty of updating. This paper presents a new approach for issuing and validating certificates, detailing how it was developed. We used the microservice architecture, which has advantages over the old system, in addition to the replacement of the relational database by the non-relational database. It is also proposed an approach to the infrastructure to be used by the project.*

Resumo. *Diante da variedade de eventos que ocorrem hoje em dia, verificou-se a necessidade de integração online entre participantes e comissão organizadora. Para isso, são usados sites que reúnem dados dos usuários, informações sobre os eventos e outras funcionalidades, sendo uma delas a geração de certificados. O Instituto Federal da Bahia Campus de Vitória da Conquista possui um sistema de gerenciamento de certificados, desenvolvido em uma arquitetura que atualmente está obsoleta devido às limitações quanto à escalabilidade, desempenho e dificuldade de atualização. Este artigo apresenta uma nova abordagem para a emissão e validação de certificados, detalhando como se deu seu desenvolvimento. Foi utilizada a arquitetura de microsserviços, que apresenta vantagens em relação ao sistema antigo, além da substituição do banco de dados relacional pelo banco de dados não relacional. Também é proposta uma abordagem para a infraestrutura a ser utilizada no projeto.*

1. Introdução

Em um mundo onde se tem uma variedade de eventos, sejam eles educacionais ou empresariais, sejam públicos ou privados, existe a necessidade de integração entre os participantes e a organização. Com as tecnologias atuais, existem plataformas online que fazem essa intermediação, gerenciando dados dos usuários e informações do evento. Algumas dessas plataformas, por exemplo, Doity[1], Even3[2] e Sympla[3], também oferecem a emissão de

[1]Doity: https://doity.com.br
[2]Even3: https://www.even3.com.br
[3]Sympla: https://www.sympla.com.br

certificados devido à necessidade de documentação comprobatória, principalmente para os eventos que ocorrem no meio acadêmico.

Para atender a essa necessidade, foi implantada uma plataforma online de emissão e gerenciamento de certificados no Instituto Federal da Bahia *Campus* de Vitória da Conquista. Tal plataforma foi desenvolvida na arquitetura monolítica, que apresenta algumas desvantagens frente a outras opções existentes atualmente, além de ter sido implementada em PHP[4] utilizando o Zend Framework[5], que foi descontinuado. Verificou-se, então, a necessidade de atualização dessa plataforma para melhoria da usabilidade do sistema e para oferecer um serviço com resposta rápida e otimizada às requisições por parte dos usuários e organizadores.

A arquitetura em monólitos não oferece tais vantagens, pois a aplicação em bloco único faz com que a sua atualização seja complicada, além de desperdiçar recursos de processamento, uma vez que de acordo com o escalonamento, há um aumento do uso de recursos computacionais [Richardson 2014b]. Outros modelos de arquitetura surgiram e tais problemas podem ser evitados com seu uso.

Também foi proposta a substituição do banco de dados relacional por um banco de dados não relacional. Identificou-se que o projeto possui a possibilidade de escalar seus recursos para que sejam utilizados em outras instituições ou em eventos avulsos, gerando assim uma grande base de dados. Tais características apresentam problemas no esquema relacional, pois ele enfrenta dificuldade em processar consultas complexas em grandes bases de informações gerando uma maior lentidão à medida que a aplicação escale horizontalmente [Hadjigeorgiou et al. 2013]. A abordagem não relacional tem como propósito solucionar essas demandas.

Nesse sentido, este artigo aborda o projeto de criação de uma plataforma de gerenciamento de emissão e validação de certificados, detalhando como se deu seu desenvolvimento, implementação e futuras manutenções. Apresenta a arquitetura utilizada, quais foram as mudanças propostas, bem como uma comparação com o sistema anterior e a configuração de sua infraestrutura.

2. Referencial Teórico e Tecnologias Adotadas

Esta seção apresenta a metodologia utilizada e detalha as tecnologias adotadas no desenvolvimento, implantação e execução da arquitetura proposta.

2.1. Banco de Dados Não Relacional

Os bancos de dados não relacionais surgiram para resolver os problemas relacionados à escalabilidade no armazenamento e processamento de grandes volumes de dados. Também conhecido pelo termo NoSQL, não é definido apenas pela ausência da linguagem SQL, mas também por outras características, a saber: ausência de relacionamento entre as entidades, arquitetura distribuída, disponibilidade na forma de código aberto, escalabilidade horizontal, ausência de esquema ou esquema flexível, suporte nativo à replicação e acesso via APIs simples. A importância do paralelismo para o processamento de grandes volumes de dados e a distribuição dos sistemas em escala global foram os fatores determinantes para o seu surgimento. Diversos bancos de dados NoSQL possuem a propriedade

[4]PHP: `https://www.php.net`
[5]Zend Framework: `https://framework.zend.com`

de consistência em momento indeterminado, sendo fundamental para atingir níveis maiores de escalabilidade [De Diana e Gerosa 2010].

2.2. Arquitetura Monolítica

A arquitetura monolítica é uma abordagem na qual todo código da aplicação está em uma única base de conjunto de códigos [Annett 2014].

Este modelo é bastante utilizado em aplicações pequenas por ser de fácil desenvolvimento, teste e implantação. Porém, conforme o crescimento da aplicação e/ou equipe, essa abordagem pode trazer várias desvantagens que poderão tornar inviável o seu desenvolvimento. Alguns exemplos são: com grande base de códigos, novos desenvolvedores terão dificuldade no entendimento e modificação, tornando o desenvolvimento mais lento e diminuindo a qualidade do código; IDE sobrecarregada; maior tempo de inicialização, afetando a produtividade e a sua implantação; dificuldade em escalar; compromisso de longo prazo com a linguagem e *frameworks* utilizados, dificultando ou até impossibilitando sua atualização, tornando sua aplicação obsoleta, além de gerar um grande desafio para migrar para uma estrutura mais nova e melhor [Richardson 2014b].

2.3. Arquitetura de Microsserviços

Na arquitetura de microsserviços, a aplicação é dividida em um conjunto de pequenos serviços, geralmente desenvolvidos em torno das regras de negócios e implantados de forma independente [Lewis e Fowler 2014].

Esta abordagem tem algumas vantagens em relação ao modelo monolítico, resolvendo alguns dos seus problemas, a exemplo da possibilidade de entrega e implantação contínua de aplicativos grandes e complexos, da manutenibilidade aprimorada, da melhor testabilidade e capacidade de implantação e da viabilidade de divisão da equipe em times menores responsáveis por um ou mais serviços. Além disso, elimina o compromisso de longo prazo com uma determinada linguagem e/ou tecnologia, facilitando a utilização e alteração de tecnologias durante o seu desenvolvimento. Porém, essa solução tem algumas desvantagens como uma maior complexidade para configurar sua implantação, tendo que utilizar algumas ferramentas para a sua gerência e tendo, por consequência, um maior consumo de memória e necessitando uma expertise adicional dos desenvolvedores [Richardson 2014a].

A Figura 1 ilustra cada uma das abordagens apresentadas, comparando as formas em que cada uma escala, enquanto a arquitetura monolítica replica toda a sua estrutura, a arquitetura de microsserviços replica apenas o serviço que tiver necessidade.

2.4. API Gateway

API Gateway é muito comum na arquitetura de microsserviços. Seu uso resolve o problema de como os clientes chamam microsserviços independentes, servindo de interface entre os clientes e os microsserviços. Além de poder fornecer ferramentas como verificação de permissão, balanceamento de carga, cache e monitoramento. O encapsulamento da estrutura interna do sistema facilita sua manutenção e disponibilização dos serviços, não necessitando que os clientes sejam alterados em caso de mudança nos serviços [Zhao et al. 2018].

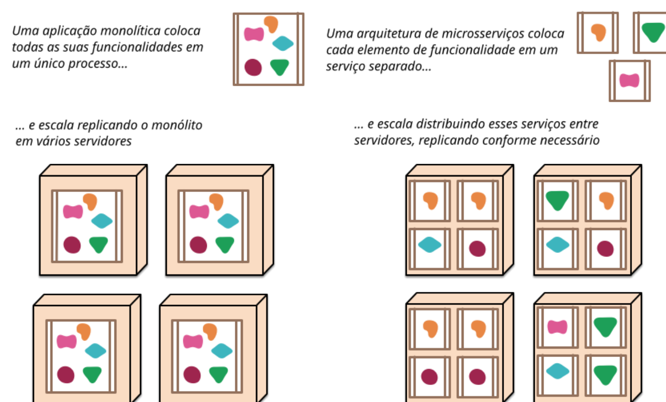

Figura 1. Monólitos e Microsserviços [Lewis e Fowler 2014]

2.5. Docker

Docker[6] é uma tecnologia de virtualização de contêiner. Ela foi criada com o objetivo de fornecer uma máquina virtual extremamente leve e ágil, facilitando o desenvolvimento, teste e implantação de aplicações, pois os códigos e serviços que forem colocados dentro dessa máquina virtual serão iguais em todos os estágios. Todo contêiner do Docker é construído sobre uma imagem, que é como um sistema de arquivos pré-configurados contendo uma camada de bibliotecas e arquivos binários necessários para a execução do aplicativo. São utilizadas duas partes da tecnologia do *kernel* Linux (*namespaces* e *cgroups*) para garantir que cada processo em um contêiner só possa ver outros processos no contêiner. Além disso, a tecnologia fornece um ambiente isolado para os endereços de rede e portas, expondo as portas apenas de maneira manual. Por conta desses recursos, essa ferramenta rapidamente se tornou extremamente utilizada na implantação de arquiteturas baseadas em microsserviços [Anderson 2015].

2.6. Docker Swarm

Docker Swarm[7] é um *cluster* nativo do Docker. É uma ferramenta que permite gerenciar vários contêineres como se fosse um único *host*. Ele permite a automatização de diversas tarefas, tal como escalonamento de acordo com a demanda e tratamento de falhas, criação de réplicas e restabelecimento de estado anterior a uma falha. Além de fornecer monitoramento dos serviços que estão em execução. O Docker Swarm apresenta vantagens sobre outros orquestradores de contêineres como a de ser nativo e integrado com ferramentas do ecossistema como Docker Machine e Docker Compose, já estando pronto para ser utilizado sem necessidade de nenhuma instalação adicional, sendo de fácil configuração e uso [Soppelsa e Kaewkasi 2016].

2.7. Traefik

Traefik[8] é uma API Gateway projetado para simplificar a complexidade quanto às operações de microsserviços. Ele fornece suporte às principais tecnologias de *cluster*.

[6] Docker: `https://www.docker.com`
[7] Docker Swarm: `https://docs.docker.com/engine/swarm/`
[8] Traefik: `https://doc.traefik.io/traefik/`

Ele tem as seguintes características: suporte aos atuais protocolos de camada de aplicativo, como HTTP/2, gRPC e REST; configuração dinâmica, podendo ser configurado automaticamente conforme o comportamento do seu sistema; *hot reloads*, executando as atualizações sem exigir reinicialização; observabilidade; métricas; terminação de TLS; *proxy* reverso; e balanceamento de carga [Sharma e Mathur 2021].

3. Estudo de Caso: Sistema de Gerenciamento de Certificados

O estudo de caso deste trabalho é baseado em um sistema de certificados desenvolvido e utilizado no IFBA *Campus* Vitória da Conquista/BA. Essa antiga versão tem como abordagem uma arquitetura monolítica utilizando a linguagem de programação PHP com o Zend Framework e o banco de dados relacional MySQL[9]. O sistema tem como objetivo fornecer a criação, edição e gerência da emissão e validação de certificados dos mais variados eventos de maneira rápida e prática, reduzindo o esforço e insumos na organização dos eventos da instituição. A versão antiga pode ser acessada pela URL `http://certificados.conquista.ifba.edu.br` e o código fonte está disponível em `https://github.com/joabepinheiro/certificados` [Ramos et al. 2018].

3.1. Arquitetura

A nova abordagem utiliza a arquitetura de microsserviços e é composta de alguns serviços e interfaces, conforme é apresentado na Figura 2. Os componentes estão descritos a seguir.

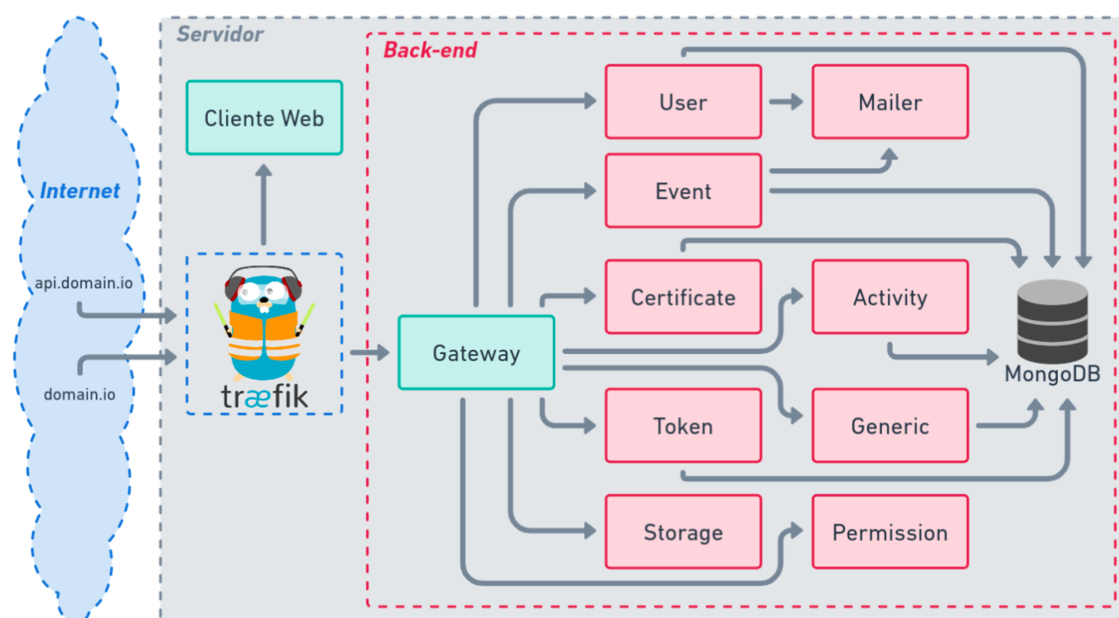

Figura 2. Arquitetura da nova aplicação para gerenciamento de certificados

- **Treafik:** o traefik é responsável pelo *Proxy* Reverso da aplicação, recebendo as solicitações do cliente web e da API, e as encaminhando para os serviços responsáveis pelo seu processamento. Foi escolhido pela sua fácil configuração (realizada através de um arquivo YAML[10]) e para ajudar a aumentar a segurança, desempenho e confiabilidade do sistema;

[9]MySQL: `https://www.mysql.com`
[10]YAML: `https://yaml.org`

- **Cliente Web:** serviço responsável por servir o *front-end* do sistema, tendo sido desenvolvido utilizando a linguagem Typescript[11] com o *framework* Next.JS[12], que utiliza como base a biblioteca React[13];
- ***Gateway*:** serviço que atua como *gateway* do *back-end*, sendo responsável por centralizar todos os serviços de forma a atuar como um *Back-end for Front-end* (BFF), garantindo um melhor desempenho e segurança;
- ***User*:** serviço responsável pela autenticação, listagem, exibição, criação, alteração e remoção do usuário, bem como pela confirmação, reenvio e recuperação da senha;
- ***Event*:** serviço responsável pela listagem, exibição, busca, alteração, criação e remoção do evento;
- ***Certificate*:** serviço responsável pela listagem, emissão, exibição, validação e remoção do certificado, bem como pela listagem, exibição, alteração, criação e remoção do modelo de certificado;
- ***Activity*:** serviço responsável pela listagem, exibição, alteração, criação e remoção da atividade;
- ***Generic*:** serviço responsável pela listagem, exibição, alteração, criação e remoção de informações genéricas do sistema, podendo ser um tipo de atividade ou função do participante;
- ***Storage*:** serviço responsável pelo *upload*, exibição e remoção das imagens utilizadas no modelo do certificado;
- ***Token*:** serviço responsável pela geração, remoção e validação dos *tokens* JWT[14], sendo utilizados na autenticação das requisições realizadas pelo usuário, garantindo uma maior segurança;
- ***Permission*:** serviço responsável pela validação das permissões de acesso de cada rota do *gateway*;
- ***Mailer*:** serviço responsável pelo envio de e-mail do sistema, podendo ser de confirmação e recuperação da senha, além de informar a disponibilização do certificado ao participante;
- **MongoDB:** banco de dados não relacional a documentos utilizado para persistência de dados.

Os serviços do *back-end* foram desenvolvidos utilizando o Nest.JS[15], um *framework* Node.JS cuja estrutura se inspira no *framework* Angular[16] e que utiliza um padrão de módulos para injeção de dependências. O Nest.JS fornece muitas ferramentas e integrações com as mais diversas tecnologias da atualidade, sendo muito utilizado na criação de microsserviços, pois possui fácil configuração. O *framework* suporta vários meios de comunicação entre os serviços, como Redis, MQTT, NATS, RabbitMQ, Kafka e gRPC. Por se tratar de um sistema que ainda não possui tanta demanda, foi optado por implementar a comunicação via protocolo TCP, podendo ser substituído facilmente no futuro por algum serviço de mensageria caso tenha necessidade.

[11]Typescript: `https://www.typescriptlang.org`
[12]Next.JS: `https://nextjs.org`
[13]React: `https://pt-br.reactjs.org`
[14]JTW: `https://jwt.io`
[15]Nest.JS: `https://nestjs.com`
[16]Angular: `https://angular.io`

O banco de dados foi reestruturado, substituindo-se o modelo relacional pelo não relacional. Tomou-se essa decisão para otimizar o desempenho da aplicação e por conta da flexibilidade nos esquemas do banco (9 tabelas no esquema original foram reduzidas para 6 coleções). Também houve uma redução no tamanho, uma vez em que muitos campos no esquema antigo eram setados como *NULL* por não serem preenchidos. Outro motivo foi a fácil integração com a arquitetura adotada, já que os documentos do banco são muito semelhantes ao JSON[17]. Existem várias soluções que fornecem um banco de dados não relacional, porém foi optado pela utilização do MongoDB por ser o mais popular[18] e ter um ótimo suporte.

Foi adotado o padrão de banco de dados compartilhado, no qual todos os serviços irão utilizar a mesma instância do banco. Este padrão se mostrou melhor para o projeto, pelo fato de ser mais simples de implantar, além de ser de fácil manutenção [Taibi et al. 2018].

A Figura 3 representa o esquema conceitual do sistema antigo; logo em seguida a Figura 4 demonstra as interfaces (descrição da estrutura do objeto) nas quais as novas coleções foram implementadas.

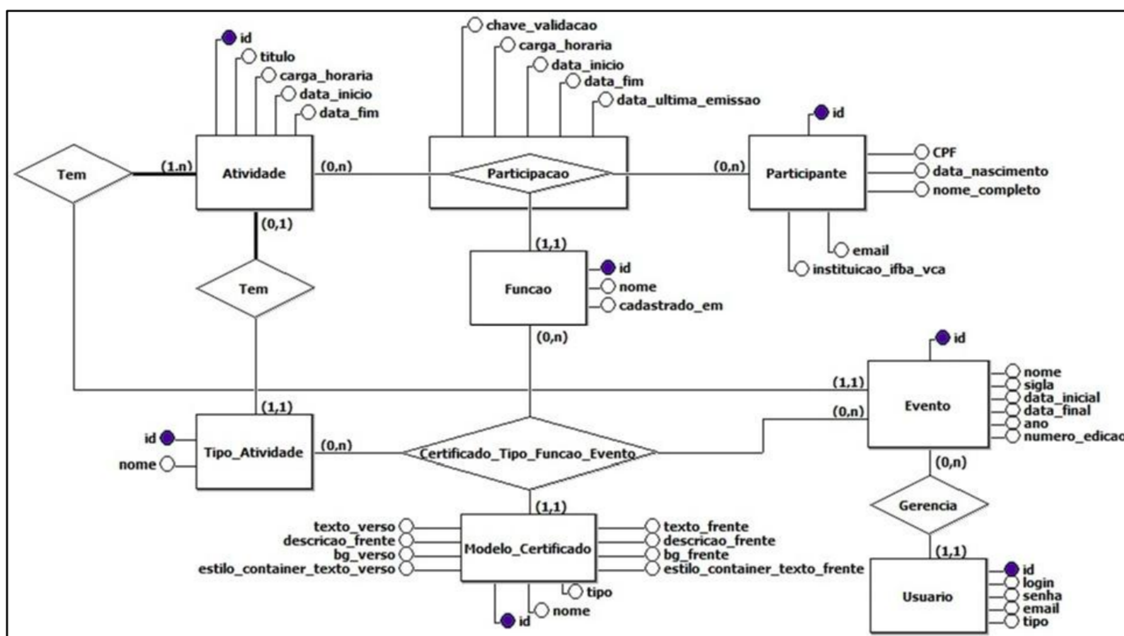

Figura 3. Esquema Conceitual do sistema certificados antigo [Ramos et al. 2018]

```
1 interface User {
2   name: String
3   email: String
4   password: String
5   role: "ADMIN" | "COORDINATOR" | "PARTICIPANT"
6   is_confirmed: Boolean
7   last_login: Date
8   personal_data?: {
```

[17]JSON: https://www.json.org
[18]Ranking de banco de dados orientado a documentos: https://db-engines.com/en/ranking/document+store

```
9       cpf: String
10      dob: Date
11      phone: String
12      institution: Boolean
13    }
14  }
15  interface Event {
16    user: User
17    name: String
18    local: String
19    initials: String
20    year: String
21    edition: String
22    start_date: Date
23    end_date: Date
24    status: "DRAFT" | "PUBLISHED" | "REVIEW"
25  }
26  interface Generic {
27    type: "type_activity" | "function"
28    name: String
29  }
30  interface Activity {
31    event: Event
32    type: Generic
33    name: String
34    workload: Number
35    start_date: Date
36    end_date: Date
37  }
38  interface Certificate {
39    activity: Activity
40    function: Generic
41    participant: User
42    event: Event
43    key: String
44    workload: Number
45    start_date: Date
46    end_date: Date
47    authorship_order?: String
48    additional_field?: String
49  }
50  interface Model {
51    event: Event
52    name: String
53    pages: Array<{
54      type: String
55      text: String
56      image: String
57      layout: {
58        padding: String
59        horizontal_padding: Number
60        vertical_padding: Number
61        position: String
62        horizontal_position: Number
63        vertical_position: Number
64      }
```

```
65    }>
66    criterions: Array<{
67      function: Generic
68      type_activity: Generic
69    }>
70  }
```

Figura 4. Interfaces das coleções do banco de dados

Pela natureza do sistema, é esperado que ele tenha grandes picos de acesso em determinado período, como ao final de grandes eventos, uma vez que muitos participantes irão acessá-lo para emitir seus certificados. Com isso, o Docker Swarm aumentará o número de réplicas apenas dos serviços que tiverem necessidade, garantindo uma melhor otimização de custos e desempenho.

É possível visualizar na Figura 5 uma comparação entre a tela de *Dashboard* do sistema antigo e do sistema novo. A nova abordagem apresenta uma tela mais limpa, trazendo uma maior consistência. Além de aplicar conceitos de *UI/UX Design*, como navegabilidade, IHC, usabilidade, acessibilidade, responsividade, entre outros.

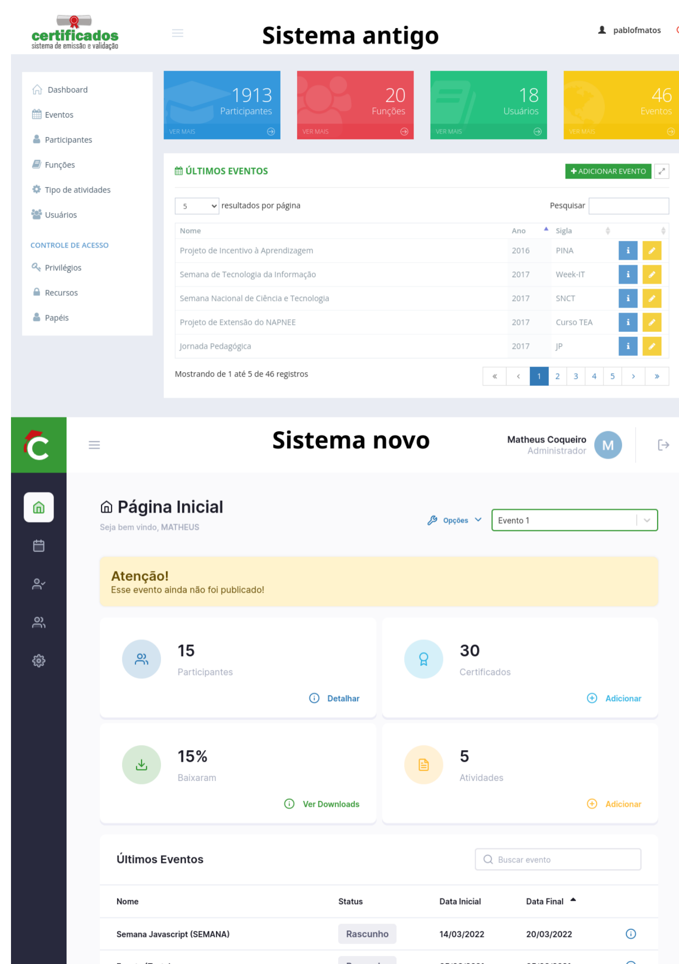

Figura 5. Tela de *Dashboard* do sistema antigo e como ela ficou no novo sistema

3.2. Implementação e Manutenção

Em função de limitação de recursos disponíveis para o trabalho, atualmente há apenas um ambiente de testes. Após conclusão de trâmites burocráticos, espera-se que o sistema esteja implantado em servidores do IFBA *Campus* Vitória da Conquista, de modo a substituir, dessa forma, a plataforma antiga de certificação de eventos apresentada anteriormente.

Como ferramenta de versionamento de código, foi utilizado o Git e a plataforma GitHub[19]. O processo de integração contínua e entrega contínua (CI/CD) é realizada através do GitHub Actions[20]. Sua configuração é realizada através de um arquivo YAML e é necessário instalar e configurar um *runner* no servidor que será utilizado. Uma vez feita essas configurações, todo o processo de integração e entrega será realizado de maneira automática, facilitando a gestão do sistema.

Através do Traefix, cada serviço será inicializado com 1 instância, podendo ser escalado horizontalmente conforme a sua demanda e necessidade. Pelo fato de ter suporte ao Docker Swarm, sua configuração é muito simples, também sendo realizada através de um arquivo YAML. A maior vantagem identificada nesse conjunto de abordagens é poder realizar a configuração via código, deixando tudo pré-configurado. Como resultado, o desenvolvedor não precisa se preocupar com o *deploy* das suas aplicações. Também foi configurado o Portainer[21], uma interface visual para gestão dos contêineres Docker. Por ser bastante intuitiva, ela reduz a complexidade operacional associada ao gerenciamento de vários contêineres, fornecendo uma interface amigável para quem realiza sua gestão. Por fim, foi configurado uma instância do *registry*, onde serão armazenadas as imagens da aplicação, garantindo que elas só poderão ser acessadas pelos desenvolvedores internos.

Uma vez que a infraestrutura esteja pronta, três passos são suficientes para o *deploy* do sistema, a saber: a) construção das imagens Docker utilizadas; b) envio das imagens construídas para a instância do *registry*; e c) inicialização de serviços a partir de contêineres gerados pelas construção das imagens. Todos esses passos são realizados de maneira automática, podendo ser conferido no *pipeline* do GitHub.

As manutenções podem ser feitas utilizando o Portainer e o Traefik, sem que haja necessidade de acessar remotamente o servidor. Caso aconteça alguma falha catastrófica com o servidor, toda a infraestrutura pode ser construída novamente em questão de minutos, já que ela se encontra pré-configurada, necessitando apenas a instalação e configuração do Docker e do *runner* do GitHub. Neste caso, restauram-se os backups do banco de dados e as imagens utilizadas na geração do certificado.

3.3. Código Fonte do Sistema

Toda a codificação do sistema abordado neste artigo está disponível gratuitamente para ser acessado através do endereço `https://github.com/Certificados-Ifba/certificates`. Já a codificação da infraestrutura utilizada pode ser acessada através do endereço `https://github.com/Certificados-Ifba/infra`.

[19]GitHub: `https://github.com`
[20]GitHub Actions: `https://docs.github.com/pt/actions`
[21]Portainer: `https://www.portainer.io`

4. Lições Aprendidas

A implementação da arquitetura proposta permitiu um maior reaproveitamento do código no desenvolvimento dos microsserviços, já que a estrutura de cada um é muito similar, necessitando mudar apenas as regras de negócio e esquemas. Essa abordagem também forneceu uma melhor comunicação entre os desenvolvedores, uma vez que o foco desse projeto foi na infraestrutura, arquitetura e desenvolvimento do *Back-end*, enquanto o desenvolvimento do *Front-end* coube a outro time de desenvolvedores. Como todos os desenvolvedores são experientes na linguagem Typescript, toda a aplicação utilizou-se da mesma linguagem. Porém, caso outros desenvolvedores tivessem expertise em outras linguagens, isto não seria um empecilho. Caso futuramente haja necessidade de realizar alguma mudança e substituição de alguma parte do sistema, é possível promovê-las de maneira incremental sem que seja necessário recomeçar o projeto do zero.

Com o isolamento dos módulos em contêineres, a tarefa de escalar a aplicação horizontalmente se tornou muito simples, permitindo a configuração de diversos *worker nodes*, criando um *cluster* para realizar o balanceamento de carga entre mais de um servidor. Por falta de orçamento e de diversas outras limitações quanto ao servidor fornecido pela instituição para realizar a implantação, essas abordagens foram simuladas apenas em ambiente de desenvolvimento, com a utilização de algumas máquinas virtuais locais. Porém, ficou nítido a facilidade que esta abordagem oferece em relação à escalabilidade.

Por ainda não ter uma base de dados robusta, não se pôde comparar o desempenho entre o banco de dados relacional e o não relacional. Porém, a utilização de um banco de dados NoSQL facilitou o desenvolvimento, uma vez que sua integração e utilização é simples e combina perfeitamente com o protocolo REST, que foi a arquitetura de comunicação utilizada entre o *Front-end* e o *Back-end*, já que ambas utilizam o formato JSON para representação dos dados.

A arquitetura descrita na Seção 3.1 será mantida internamente pelos funcionários da instituição, sendo utilizado um servidor local com uma máquina virtual criada dedicada para o projeto. Foi configurado o backup diário do banco de dados e também dos arquivos do serviço *storage*, que são os únicos dados necessários para a restauração da aplicação.

Existe diversas melhorias que podem ser implementadas futuramente, como as seguintes ferramentas de observabilidade: Kibana, para a visualização dos *logs* dos serviços; Prometheus, para fornecer diversas métricas importantes, por exemplo, consumo de CPU/memória, quantidade de requisições, entre outras; Jaeger, para rastreamento dos erros gerados; e Grafana, para gerar *dashboards* que permitirão monitorar todos os serviços e o servidor, além da criação de alertas em caso de problemas. Também é interessante a utilização de algum serviço de mensageria, pois fornece muitos recursos que são importantes conforme a aplicação cresça, como filas e tratamento de erros caso ocorra alguma falha durante a comunicação dos serviços, fornecendo uma alta disponibilidade. Essas melhorias não foram implementadas por limitações e falta de recursos, mas podem ser facilmente implantadas caso seja necessário. Outra mudança que pode ser feita é configurar uma instância do banco de dados para cada microsserviço, deixando os serviços ainda mais desacoplados e potencializando a atualização do banco de dados não relacional valendo-se de sua arquitetura distribuída.

5. Considerações Finais

Este trabalho apresenta uma nova abordagem para o sistema de certificados que o IFBA *Campus* Vitória da Conquista utiliza atualmente. Esta abordagem é baseada na arquitetura de microsserviços, na substituição do banco de dados relacional pelo não relacional e na infraestrutura utilizada para a sua implementação. Essa nova estrutura foi pensada a partir da necessidade de atualizar o sistema atualmente utilizado.

O presente artigo explica em detalhes a estrutura proposta, descrevendo as tecnologias utilizadas e quais foram as decisões tomadas. É realizada uma comparação com a estrutura em vigência. Também é explicado como foi a implantação do sistema e como foi organizado a sua manutenção.

Por fim, o estudo relata as lições aprendidas e as dificuldades enfrentadas durante o seu desenvolvimento, bem como melhorias que podem ser implementadas em trabalhos futuros.

Referências

[Anderson 2015] Anderson, C. (2015). Docker [software engineering]. *Ieee Software*, 32(3):102–c3.

[Annett 2014] Annett, R. (2014). What is a Monolith? Disponível em: `http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html`. Acesso em: 21 julho 2022.

[De Diana e Gerosa 2010] De Diana, M. e Gerosa, M. A. (2010). Nosql na web 2.0: Um estudo comparativo de bancos não-relacionais para armazenamento de dados na web 2.0. In *Workshop de Teses e Dissertações de Bancos de Dados do Simpósio Brasileiro de Bancos de Dados WTDBD2010*.

[Hadjigeorgiou et al. 2013] Hadjigeorgiou, C. et al. (2013). Rdbms vs nosql: Performance and scaling comparison. *MSc in High*.

[Lewis e Fowler 2014] Lewis, J. e Fowler, M. (2014). Microservices. Disponível em: `https://martinfowler.com/articles/microservices.html`. Acesso em 21 julho 2022.

[Ramos et al. 2018] Ramos, L. L., da Silva, J. P., Sobreira, A. D., e Matos, P. F. (2018). Sistema web e open source de gerenciamento de emissão e validação de certificado nos institutos federais. In *XII Congresso Norte Nordeste de Pesquisa e Inovação*, Recife, PE.

[Richardson 2014a] Richardson, C. (2014a). Pattern: Microservice Architecture. Disponível em: `https://microservices.io/patterns/microservices.html`. Acesso em: 27 julho 2022.

[Richardson 2014b] Richardson, C. (2014b). Pattern: Monolithic Architecture. Disponível em: `https://microservices.io/patterns/monolithic.html`. Acesso em: 21 julho 2022.

[Sharma e Mathur 2021] Sharma, R. e Mathur, A. (2021). *Traefik for Microservices*. Springer.

[Soppelsa e Kaewkasi 2016] Soppelsa, F. e Kaewkasi, C. (2016). *Native docker clustering with swarm*. Packt Publishing Ltd.

[Taibi et al. 2018] Taibi, D., Lenarduzzi, V., e Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress.

[Zhao et al. 2018] Zhao, J. T., Jing, S. Y., e Jiang, L. Z. (2018). Management of api gateway based on micro-service architecture. In *Journal of Physics: Conference Series*, volume 1087, page 032032. IOP Publishing.