Microsserviços, por que tão difícil?
Um catálogo de padrões para criar serviços bons de se operar

Leonardo Leite[1], **Alberto Marianno**[1]

[1]Serviço Federal de Processamento de Dados (Serpro)

{leonardo.ferreira-leite, alberto-marianno.santos}@serpro.gov.br

Resumo. *Apresentamos aqui um catálogo de padrões para aprimorar a operacionalização de serviços, o que reduz o tempo de reparo. Tal catálogo é baseado na prática de equipes do Serpro, estatal federal de tecnologia. Contudo, há um ônus considerável para se aplicar todos esses padrões. É possível também a má aplicação e o esquecimento deles. Assim, o catálogo traz reflexões sobre dificuldades em se produzir microsserviços (muitos serviços com constante atualização). Tais reflexões podem basear futuras pesquisas.*

Abstract. *Here we present a patterns catalog to improve service operation, thus reducing time to repair. This catalog is based on the practice of teams at Serpro, a federal state technology company. However, there is a considerable onus to apply all these patterns. There is also the possibility of misapplying or forgetting them. Thus, the catalog brings reflections on the difficulties in producing microservices (many services with constant updating). Such reflections can subsidize future research.*

Key words: microsserviços, serviços, APIs, REST, padrões

1. Introdução

Um sistema com uma arquitetura de microsserviços é composto por diversos serviços que se comunicam pela rede e são independentemente implantáveis [Lewis and Fowler 2014]. Independentemente da quantidade de serviços ou do tamanho desses serviços, a partir do momento que um sistema é composto por alguns poucos serviços e mais de uma base de dados, a equipe de desenvolvimento mantendo esses sistema já enfrenta os desafios tipicamente elencados para microsserviços.

No contexto do Serpro, desenvolvedores utilizam uma plataforma interna de infraestrutura, provida por um time de plataforma centralizado [Leite et al. 2020], para implantarem e operarem seus serviços de forma autônoma. Assim, desenvolvedores não precisam ter profundos conhecimentos em infraestrutura; mas, por outro lado, passam a operar seus serviços, o que basicamente significa serem responsáveis pelo atendimento de chamados e reestabelecimento dos serviços dada alguma indisponibilidade.

Nesse contexto, o atendimento de incidentes é uma tarefa indesejável para desenvolvedores, que preferem criar novas funcionalidades, atividade pela qual serão de fato reconhecidos. Além disso, o chaveamento de atenção entre diferentes projetos no mesmo dia reduz a produtividade do desenvolvedor [Vasilescu et al. 2016]. Ou seja, quanto mais rapidamente os incidentes forem resolvidos, mais previsível será o fluxo de entrega dos desenvolvedores, o que também agradará aos gerentes.

O tempo de recuperação de incidentes é uma das métricas que compõe o desempenho de entrega, importante construto relacionado à capacidade organizacional de se atingir tanto metas comerciais (rentabilidade, produtividade e participação de mercado), quanto não comerciais (eficácia, eficiência e satisfação do cliente) [Forsgren et al. 2018]. O trabalho de Forsgren et al. traz inclusive lições de que o tempo de recuperação importa mais do que a taxa de falhas para o desempenho de entrega.

Contudo, mesmo com facilidades trazidas por *frameworks* e plataformas, aprimorar a operacionalização de serviços requer grande esforço e atenção a detalhes por parte dos desenvolvedores. E este artigo é evidência dessa afirmação. Apresentamos aqui padrões [Meszaros and Doble 1997] para aprimorar a operacionalização de serviços utilizados por desenvolvedores no Serpro. Por questões de espaço, apresentaremos apenas alguns padrões de nosso catálogo e de forma bem resumida. A apresentação de cada padrão focará numa breve descrição, no benefício trazido ao desenvolvedor e na problematização da aplicação do padrão. Esse último ponto é que poderá subsidiar *insights* para que pesquisadores concebam futuras pesquisas.

2. Alguns padrões de nosso catálogo

Apresentamos agora de forma resumida sete padrões de nosso catálogo.

2.1. URL de verificação de saúde de serviço

Ao analisar um incidente de indisponibilidade, a primeira ação que o desenvolvedor deve tomar é acessar a URL de verificação de saúde do serviço afetado. Essa URL fornece a situação das integrações realizadas com outros serviços, exibindo um *stacktrace* que descreve os detalhes do problema ocorrido. Assim, essas URLs aceleram o tempo de diagnóstico durante a análise de incidente, assim como auxiliam a verificação de saúde do serviço logo após a implantação de uma nova versão (teste de fumaça).

A implementação dessas URLs envolve alguns detalhes, como a adequação para leitura por ferramentas corporativas de monitoração, adequação para uma exibição agradável do *stacktrace* a humanos, proteção contra acessos de fora da rede interna (aplicável a serviços acessíveis via Internet), padronização do *path* da URL entre diferentes serviços, etc. Infelizmente, a implementação dessas URLs exige um certo esforço manual e por vezes alguns desses e outros detalhes são esquecidos ou negligenciados. Como a padronização dos *paths* já não é seguida estritamente, é importante a documentação da URL no arquivo *readme* do projeto, mas mesmo isso é negligenciado por vezes.

2.2. Verificação de preenchimento de variáveis de ambiente

Por conveniência e para um bom *onboarding*, temos valores pré-definidos para variáveis de ambiente durante a execução dos serviços na estação de trabalho do desenvolvedor. Contudo, não toleramos o risco de que esses valores pré-definidos acabem sendo utilizados no ambiente de produção por engano, pois isso já nos causou problemas de demorado diagnóstico. Portanto, se alguma variável não estiver preenchida em produção (salvo exceções definidas explicitamente), o serviço não será inicializado. Esse é outro padrão que aplicamos de forma manual, copiando código de serviços antigos para novos serviços.

2.3. Auditoria

Para cada requisição feita à API, guardamos um registro de auditoria em bancos de dados. Temos um conjunto de vários campos que aplicamos à auditoria de cada serviço, com alguma variação para serviços internos e expostos na Internet. A solução completa de auditoria exige bastante codificação em cada serviço, com grande variação conforme a tecnologia (ex: SpringBoot ou Java EE). Vários detalhes devem ainda ser levados em conta, como o registro das falhas de autenticação e a supressão de dados binários do envio e resposta da requisição (para evitar estouro de disco). Além disso, uma certa engenharia é necessária para evitar que o acesso à auditoria se torne lento demais.

2.4. Tratamento de erro

Um bom tratamento de erros, útil para a operacionalização do serviço, requer convenções e disciplina. Definimos que erros 400 são culpa do programador do sistema cliente, erros 422 são culpa do usuário do sistema cliente e erros 500 são erros interno (falhas de programação do serviço ou intermitência na infraestrutura ou dependências). Porém, nem sempre é trivial diferenciar essas situações. Além disso, boas mensagens devem ser concebidas para que o destinatário saiba não só sobre o problema, mas principalmente que ação tomar. Outro complicador são os comportamentos que julgamos inadequados dos *frameworks* e que temos que sobrescrever; o SpringBoot, por exemplo, retorna erro 500 para o recebimento de um JSON mal formado, o que deveria ser um erro da família 400 (culpa é do sistema cliente, não um erro interno).

2.5. Manual

Valorizamos muito a geração automática de documentação executável (Swagger), contudo a mesma costuma não ser suficiente, principalmente para APIs de acesso pelo público externo. Investimos também bastante esforço na confecção de manuais que explicam os conceitos da API, regras de negócio, a semântica das operações, possíveis situações de erros, além de fornecer dicas mais gerais como sobre o uso de certificado digital para autenticação. Esse manual acelera o *onboarding* de desenvolvedores do sistema cliente e evita acionamentos devido a dúvidas (idealmente desenvolvedores devem receber acionamentos apenas sobre problemas em produção). Contudo, produzir essa documentação com qualidade, inclusive com um português bem escrito, possui um alto custo, não sendo todos os desenvolvedores que possuem perfil para esse trabalho de escrita.

2.6. Monitoração no nível do serviço e do negócio

Bons painéis com métricas de negócio podem evitar que a área de negócio interrompa o trabalho dos desenvolvedores para pedir informações. A geração desse tipo de métrica requer algum trabalho manual, pouco propenso a automação, já que dependem de uma certa sensibilidade para definir as métricas de relevância.

Já as métricas no nível do serviço são mais facilmente automatizadas por *frameworks* e são importantes para servirem de base para alertar os desenvolvedores (usamos principalmente tempo de resposta e proporção de erros internos). Contudo, o desafio aqui é a calibração dos limiares que disparam os alertas. Um limiar muito baixo pode invisibilizar problemas; já um muito alto tende a gerar muitas notificações, que acabarão sendo ignoradas pelos desenvolvedores.

2.7. Resolução de inconsistências por meio de idempotência

Um grande problema de sistemas com bases de dados distribuídas é a geração de inconsistências nos dados diante de uma falha de infraestrutura. Esse cenário pode gerar um grande desgaste, no qual o cliente abre um chamado, o desenvolvedor deve investigar cuidadosamente a situação e, por fim, alterar os dados de produção para um estado consistente. Porém, entendemos que idealmente desenvolvedores não deveriam alterar dados de produção, pois um erro pode piorar a situação, além do que há a questão sobre a responsabilização da alteração do dado do usuário.

Diante do cenário exposto, procuramos criar operações idempotentes, de forma que diante de um erro interno, o usuário seja induzido a uma nova tentativa que possa levar os dados do estado inconsistente para um consistente. Porém isso requer uma implementação bem manual e com vários detalhes envolvidos. Por exemplo, se algum dado da segunda requisição vier diferente do da primeira, esse dado deve sobrescrever o dado anterior ou um erro deve ser lançado?

3. Conclusão

Apresentamos nesse artigo alguns padrões aplicados por desenvolvedores do Serpro para aprimorar a eficiência operacional de serviços, o que se aplica ao contexto de arquitetura de microsserviços. Porém, muitos desses padrões exigem um certo esforço manual, e por vezes desenvolvedores podem, com isso, cometer erros na aplicação do padrão, ou até se esquecerem ou negligenciarem alguns padrões ou detalhes de suas implementações. Acreditamos haver portanto espaço para futuras pesquisas que concebam formas de se aliviar para desenvolvedores o peso da implementação de sistemas baseados em microsserviços.

Nosso catálogo atualmente possui cerca de 33 padrões propostos, sendo alguns focados na manutenibilidade, e não na operacionalização de serviços. Alguns exemplos de padrões omitidos neste artigo contemplam discussões sobre *feature toggles* e *tracing*. Além disso, em nosso catálogo cada padrão é apresentado de forma extensa seguindo uma estrutura, podendo incluir as seguintes seções: título, nome curto, contexto, motivação, quando usar, benefícios, solução em alto nível, exemplo de implementação, serviços que usam esse padrão, "cuidados, detalhes, e desvantagens", alternativas e padrões relacionados. Como trabalho futuro, esperamos publicizar completamente nosso catálogo.

Referências

Forsgren, N., Humble, J., and Kim, G. (2018). *Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations*. IT Revolution Press.

Leite, L., Pinto, G., Kon, F., and Meirelles, P. (2020). Platform teams: An organizational structure for continuous delivery. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 505–511.

Lewis, J. and Fowler, M. (2014). Microservices. `https://www.martinfowler.com/articles/microservices.html`, acessado em abril de 2023.

Meszaros, G. and Doble, J. (1997). A pattern language for pattern writing. In *Proc. of International Conference on Pattern languages of program design 1997*, volume 131, page 164.

Vasilescu, B., Blincoe, K., Xuan, Q., Casalnuovo, C., Damian, D., Devanbu, P., and Filkov, V. (2016). The sky is not the limit: Multitasking across GitHub projects. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 994–1005. ACM.