UDC 005

SERVICE VERSIONING IN MICROSERVICE ARCHITECTURE

YAUHENI ZHYDZETSKI, SERGEY SURTO
Polotsk State University, Belarus

Microservice architecture is all about integration and contracts. API versioning is extremely important to control compatibility of different parts of system and provides completed business features. There are different versioning techniques and it is important to select the most suitable for your project.

Microservices is a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services should be fine-grained and the protocols should be lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop and test. It also parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous refactoring. Microservices-based architectures enable continuous delivery and deployment [1].

The microservice architecture is not a silver bullet. It has several drawbacks. One of the most important – is integration complexity. Many different services work together in continuously changing environment to provide competed business feature. The probability of failure rapidly increases with number of service integration points, so it is extremely important to have a possibility to control compatibility of different parts of the system [2]. One of the most popular technique for that – is service versioning.

Service Versioning is the approach followed by service developers to allow multiple versions of the same service to be operational at the same time. To give an analogy, any re-usable software API library has multiple versions used by different applications. The same analogy applies to services [2].

The most popular versioning scheme today – is Semantic Versioning 2.0.0. It proposes a simple set of rules and requirements that dictate how version numbers are assigned and incremented. These rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software. For this system to work, first need is to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once public API identified, every service change will trigger specific increments to its version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version [3]. Before the service firstly published to production, major version may be zero, and in this case, minor and patch bump-up rules may be ignored.

Semantic versioning is very suitable for libraries, but in case of microservice architecture, classical semantic versioning Major.Minor.Patch format should be interpreted as Breaking.Feature.Fix API change. For a live production system, it is actually need to worry about making breaking changes to services (the first SemVer number). If developer wants to be able to make a breaking change to a service, he must provide a way of making that change while still supporting the old version of the contract. Feature part of version is also important to indicate left border of supported API versions for the consumer. As for the patch/fix part, it is internal service implementation detail, which not affects contract between the service and its consumer.

Breaking changes are any change to the contract, which is provided by the service, which is not backward compatible. There are many different types of changes, which could have the potential to be breaking. The most common places for breaking change are transfer schemas and endpoints. The transfer schema is the structure of the data you will either receive or emit in response to an external request. In HTTP this includes any response payload and the structure of posted content. In a messaging environment, this includes events that service emits and commands it receives. Endpoints are the place another service would go to connect to the service. In HTTP this is the URL, in messaging the source that service listening to commands on or publishing events to and its routing information. There are many different implementation techniques to provide backward compatibility, but it is impossible completely avoid breaking changes. API First method can help to pay due attention to contract design and reduce their quantity.

API First is one of engineering and architecture principles. Concisely, API First requires to define APIs outside the code first using a standard specification language and to get early review feedback from peers and client developers. Service APIs should evolve incrementally. Of course, API specification will and should evolve itera-

tively in different cycles; however, each starting with draft status and early team and peer review feedback. API may change and profit from implementation concerns and automated testing feedback. API evolution during development life cycle may include breaking changes for not yet productive features. API First does not mean that developer must have full domain and requirement understanding and can never produce code before have defined the complete API and get it confirmed by peer review. On the other hand, API First obviously is in conflict with the bad practice of publishing API definition and asking for peer review after the service integration or even the service productive operation has started. It is crucial to request and get early feedback — as early as possible, but not before the API changes are comprehensive with focus to the next evolution step and have a certain quality, already confirmed via team internal reviews.

There are different techniques and technologies for API definition. The most popular are OpenAPI Specification with Swagger implementation and Contract Testing with Pact implementation.

Swagger is an open source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services. While most users identify Swagger by the Swagger UI tool, the Swagger toolset includes support for automated documentation, code generation, and test case generation. It is extremely powerful framework, even cumbersome, but it is not suitable outside RESTful API [4].

Contract Testing is writing tests to ensure that the explicit and implicit contract of a service works as advertised. This methodology applicable for any kind of API: synchronous HTTP-based, asynchronous message-based, binary protocols. The Pact is a family of frameworks provide support for Consumer Driven Contracts testing. Consumer Driven Contracts is a pattern that drives the development of the Provider from its Consumer's point of view. It is Test Driven Development for services [5].

It is extremely important and relatively easy to implement contract testing as a part of the continuous delivery build pipeline. Continuous integration (CI) system should make cross validation for all contracts of the service against its consumers from production environment to check backward compatibility and validate version increment made by developer before real deployment. In addition, CI system should generate service manifest describing all external service contracts and supported versions. The manifest may be used by orchestration system on update and rollback operations to control cross compatibility of all services within a single environment. Except the manifest generation, CI system should assign unique identifier to the each built service to allow its simple identification in build history. For such simple cases, semantic versioning is overabundant, and recommended way is to use simple sequential counter or time-based generator.

There are different service versioning techniques, and each is with their own pros and cons. In our company, we use adapted semantic versioning approach for each separate service public API, and simple time-based unique versioning for a whole service. In addition, we use mandatory automated contract-based API testing to eliminate human error in version assignment. That allows us to use declarative description of microservice cluster structure, isolated testing of each service and automated control of cross-service consistency at the runtime environment, with the preservation of independent development and continuous deployment of each separate service.

REFERENCES

1 Microservices [Electronic resource] / Wikipedia, the free encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Microservices. – Date of access: 14.02.2018.
2 Harris, T. SOA Service Versioning - Best Practices [Electronic resource] / T. Harris // Become The Platform. – Mode of access: http://www.thbs.com/thbs-insights/soa-service-versioning-best-practices. – Date of access: 14.02.2018.
3 Semantic Versioning 2.0.0 [Electronic resource] / Semantic Versioning. – Mode of access: https://semver.org/spec/v2.0.0.html. – Date of access: 14.02.2018.
4 OpenAPI Specification [Electronic resource] / Swagger documentation. – Mode of access: https://swagger.io/specification. – Date of access: 15.02.2018.
5 Pact introduction [Electronic resource] / Pact documentation. – Mode of access: https://docs.pact.io. – Date of access: 15.02.2018.