Composing a Release out of Microservices

由 Myrle Krantz创建于十一月 10, 2016

Vocabulary of the problem

What is a REST microservice?

A quick review by way of Wikipedia. If you need to know more, this is not the place:

REST is

- Client-Server
- Stateless
- Cacheable
- Uniform interface

A microservice is

- Elastic
- Resilient
- Composable
- Minimal
- Complete

In particular, a REST microservice is responsible for its own persistence. Other services interested in the same information should acquire that information from the REST microservice, and never directly from the persistence.

What kind of dependencies do I have in a landscape of REST microservices?

Runtime (client-server communication)

- Mediated by a protocol. In this case of REST: http.
- Dependency can be to a range of versions.
- Replacing it can be accomplished without a restart of the dependent service.

Static (included library)

- In C++, the library would be linked. In Java it might be included in a fat jar, or added to the class path.
- Consumer code is always compiled against a specific version.
- Replacing a static dependency requires at least a restart of the dependent process, and possibly a recompile.

Static dependencies really only get interesting in a deployment model when they have persistence

- They can have no persistence.
- They can have context-local persistence. This means for example that the library included in a REST service talks to a DB table created specifically for that REST service.
- They can have global persistence. In this case every REST service which includes the library would use it to communicate with a central DB.

What kind of release and development rhythm do we want?

Loosely coupled REST service development

In Open Source and in any project with more than five developers, development is distributed. The consequence of this is that someone may change Red, and update Purple, but ignore Orange. It may take a while before someone else gets around to updating Orange. But since Orange is part of the product, Red's deprecated API versions can't be removed until all the code dependencies to it have been removed. In some cases, it may take a while for all of the services in a product landscape to be upgraded to be compatible with the latest version of a service which is consumed by many of them. And in the worst case someone could remove Red's deprecated API version before the dependencies were removed. So how do we package a release in an environment like this?

A heartbeat release cycle, is a release at a regular, defined interval which disregards the state of development of any specific feature. Incomplete features are simply not integrated until they are complete. Releases are not held up for incomplete features Many projects have discovered the benefits of a heart beat release cycle in generating excitement with the user and developer base, and creating short feedback cycles and predictability.

What about a REST microservice needs to be compatible?

Persistence

- Configuration file
- Database schema
- Output data file formats
- Input data file formats
- Bearer tokens
- Copy/paste format

API

- Endpoints
- Parameters
- RequestBody
- ResponseBody
- Headers

SPI

- Events
- Calls to interfaces
- Any kind of inversion

How do I upgrade a REST microservice without downtime?

A/B deployment. (aka blue-green deployment, aka red-black deployment)

What is backwards compatible?

Changes which do not force a programmer to change or recompile consumer code, or migrate existing data are backwards compatible. Changes which do not prevent a side-by-side deployment of the “from” and “to” versions for an upgrade are also backwards compatible.

Persistency examples

- Add an optional database field.
- Make changes to a file format which do not prevent an older version from reading the file.
- Add an optional field to a security token.

API examples

- Add an optional field to a RequestBody, or an optional request parameter
- Add a field to the ResponseBody which is not critical to interpretation
- Add an optional header parameter

What is backwards incompatible?

Changes which do force a programmer to change or recompile consumer code, or which force data migrations are backwards incompatible. Changes which prevent a side-by-side deployment of the original and changed versions are backwards compatible.

Persistency examples

- Remove a required database field.
- Change a file format from xml to yaml

- Remove an endpoint
- Remove an expected field from a ResponseBody
- Remove a field from a RequestBody
- Add a required field to a RequestBody
- Add a required header parameter

How do I describe the compatibility of my components?

What is versioned?

For simplicity's sake, I will assume that versioning is performed on the persistence, the API, the SPI, and the artifact synchronously. There are even systems in which they are versioned separately. Since we can only release, deploy, and upgrade artifacts (or the source code which builds them), the artifacts must be versioned. That means that, for example if the persistence is not adjusted for several versions there will "holes" in the list of persistence versions. It also means that if you release a new artifact version which does not change the API at all, you will still have a new API patch version.

How do we describe a version?

We use Semantic Versioning for Fineract.

"Semantic versioning

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes."

Express a runtime dependency as an exact major version, and a minimum minor version.

Note that this is linear. You could describe a versioning scheme in which minor versions continue to be added after a major version shift. However this adds dimensionality to the problem of releasing that is more likely to confuse users, developers, and release managers. It's benefits are short-term at best. For the rest of this presentation I will assume linear versioning.

Declare the version API dependency

Common approaches in REST interfaces

- Make it part of your URI: "/service/v1/apples"
- Make it a parameter in your header, for example in the Accept header or in a custom header.
- Make the major version a URI segment and the minor version a header parameter.
- ...

Which approach you choose doesn't matter. For Fineract we choose the first approach. I'm sorry in advance for the pissing contests you'll discover on the internet on this topic when you go to do your research.

A Note on Continuous Development

There is no "big-bang" of feature development for the programmer, but for the code base there can be. Features can be introduced by merging a branch once the development and QA is completed. It is only at the time that a merge is completed that a final version number is defined for the feature. For the sake of this description I will be treating each feature introduction as having its own assigned semantic version. When you go to do this in real life, you can of course put multiple features in a release.

Scenarios

These scenarios cover only development and deployment. How we release is a consequence of how we develop and deploy.

Primary REST services

Primary REST services are REST services which may depend on static libraries but which do not depend on other REST services.

Backwards compatible changes to API or persistence

Backwards incompatible change to API

There are two possible approaches to bringing the change into production:

Side-by-side deployment

1. Implement the new API in a new artifact
2. Deploy both artifacts side-by-side indefinitely.

or…

Side-by-side implementation

1. Implement the new API side-by-side with the original API. Increment the artifact's major version.
2. A/B deploy the new artifact version.

Note that the side-by-side deployment approach means that you will either have to restrict persistence changes to changes which are backwards compatible, or you will have to update that older version of the artifact when you update your persistence. I will focus on the second approach.

Backwards incompatible change to the persistence

Backwards incompatible changes to persistence always require at least two upgrade steps to encapsulate migration without causing downtime.

The REST service has a new required field

1. Create a new REST service artifact "B" which creates the field and treats it as optional in read operations but required in write operations with an incremented minor version.
2. Create a migration artifact (or include migration in the artifact from step 1) which populates the field.
3. A/B deploy "B".
4. Create a new REST service artifact "C" which treats the field as required with an incremented minor version.
5. Wait until the migration is complete, then A/B deploy C.

The REST service dramatically changes its underlying persistence

(for example from an SQL DB to a NoSQL DB)

1. Create a new REST service artifact "B" which writes to both the old and the new persistence. Give it an incremented minor version.
2. Create a migration artifact (or include migration in the artifact from step 1)
3. A/B deploy "B".
4. Create a new REST service artifact which updates only the new persistence format, and give it an incremented minor version.
5. Wait until the migration is complete, then A/B deploy C.

Shared library

Without persistence

This one is easy. Create a new version. Include it where you need it.

With context-dependent persistence

Same as above for primary REST services.

With global persistence

Don't do this, unless you absolutely have to. If you do it, do not change the format of the shared data in a backwards incompatible manner. If you do change the format in a backwards incompatible manner, then you will need to continue to produce and consume both formats until all instances of all services are updated to consume the new format. In the case of a global database, that will mean that you need to keep the tables in the two schemas synchronized in both directions until all services are updated.

Dependent REST services

REST services Purple and Green consume functionality offered by the REST service Blue. REST services Purple and Orange consume REST service Red. REST service Brown consumes Purple, Orange, and Green.

Backwards compatible changes to primary service API

A new feature requires changes in Purple, and those changes in Purple place new requirements on Blue that can be solved with backwards compatible changes.

1. Create a new version of Blue's artifact which contains the changes.
2. Create a new version of Purple’s artifact which contains the changes dependent on Red’s changes.
3. A/B deploy the new version of Blue.
4. A/B deploy the new version of Purple.

Because the changes are backwards compatible, a new version of the remaining services is not necessary.

Backwards incompatible changes to API

A new feature requires changes in Purple, and those changes in Purple place new requirements on Blue that can only be solved with backwards incompatible changes.

1. Create a new minor version of Blue's artifact which contains both the old and the new API.
2. Create a new version of Purple’s artifact which contains the changes dependent on Blue’s changes.
3. A/B deploy the new version of Blue.
4. A/B deploy the new version of Purple.
5. (optionally) Create a new major version of Blue’s artifact which removes the old API, and A/B deploy that.

Changes to persistence

As long as there is no API change involved, upgrades of REST services with changed persistence can be handled exactly as though the REST service were isolated. It doesn’t matter whether they are backwards compatible or incompatible. For migrations though the sequence of versions may matter.

Release Composition

Solutions

Given that our user base will have to follow deployment paths like the ones I’ve described, that our development may not be perfectly synchronized, but that we want to be able to produce regular releases how do we do this? There are several possible approaches

Developers who add features must upgrade the whole landscape

Pros

- Developers know their way around the entire code-base, and understand the impact of their changes.
- The latest features are always in the release.
- Release cycle management is very straightforward.

Cons

- Feature development is slower.
- Some users don’t need all the microservices, but they do need the new features.
- Some developers aren’t familiar with all the microservices, insisting that they get familiar reduces the number of potential developers.

This is the only available approach if all of the code is in one repository.

Releases are composed of the newest version of every service

Pros

- The latest features are always in the release.
- Release cycle management is very straightforward.

Cons

- In some cases we would be delivering product parts which can’t be assembled into a working whole.

Each service has its own release cycle

Pros

- The latest features are always in the release.
- Release cycle management is very straightforward.

Cons

- Lots of small releases need to be checked on a regular basis making release management time intensive.
- Composing the services into a working product becomes the user’s problem.

Releases contain some but not all not services

The community would identify the critical services which always need to be released. Only those services which have a version compatible to the newest version of those services would be included in the release.

Pros

- Users can take a release and use it, if it includes the services they need.
- The latest features in the most central components are always in the release.
- Developers can work more independently from each other when starting to introduce new features.

Cons

- Some services would be missing from some releases.
- Release management requires an analysis of version dependencies.
- Doesn’t address upgrade paths (which none of the above do either)

Releases contain all services, but not necessarily in the newest version

The community would identify the maximum compatible version of each service and compose a release out of those versions of those services.

Pros

- Users can take a release and use it as.
- Every release includes all the services created by the community.
- Developers can work more independently from each other when starting to introduce new features.

Cons

- Some new features which are already integrated won’t be released until all services are compatible with each other.
- Release management requires an analysis of version dependencies.
- Doesn’t address upgrade paths (which none of the above do either)

As you can see based on the above scenarios, upgrade of a running system without downtime is a non-trivial problem requiring considerable knowledge of the dependencies and the kinds of changes made in the services. Because of this, a helpful development community should include an upgrade guide between adjacent releases as part of each release. This should inform users which order to upgrade the services in. If the release includes a data migration, users need to know how to tell when data migration is complete. Formulating such an upgrade guide is also one way to make certain that the release *is* upgradeable.

Compatibility aspects I neglected above

- Event producers/SPIs
- Behavioral changes
    - Synchronicity
    - Time and memory performance

References

- “Practical API Design: Confessions of a Java Framework Architect 1st ed. Edition” Jaroslav Tulach
- https://en.wikipedia.org/wiki/Representational_state_transfer
- https://en.wikipedia.org/wiki/Microservices
- https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html
- http://semver.org/
- http://www.cio.com/article/2434640/developer/7-ways-to-improve-your-software-release-management.html
- https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/

无标签