Brocan

Motivation

University projects are among the toughest without question. Either because you're forced to implement something you're not genuinely interested in, or the other way around, the freedom of choice makes the initial stages of the project a nightmare. However, freedom allows you to think big, and make your long-awaiting dreams come true. That is the motivation behind Brocan - I've always wanted to create my own CI system, and now I had the opportunity to "sell" it as a uni project.

Features

Even the most basic functionality of a CI system - the ability of executing an arbitrary test set against a codebase - is so complex, I did not plan any other features on top of that. However the sheer execution and a YES/NO answer (whether the tests were green or not) would not be that informative, so when doing a requirements analysis, I identified some additional points as must-have requirements (instead of nice-to-have):

- The fundamental ones:
    - Brocan should be able to receive code change notifications from repository providers.
    - Based on changesets, Brocan should be able to run an arbitrary test suite or arbitrary piece of code against the repository.
    - Builds should be configurable through an in-repository metadata file.
- Build results should be stored and be queryable through an open API.
- All logs produced by executions should be captured, stored and exposed through an open API.

Processes

Apart from retrieving results, activities related to build can be summarized in two processes:

- build request,
- build execution.

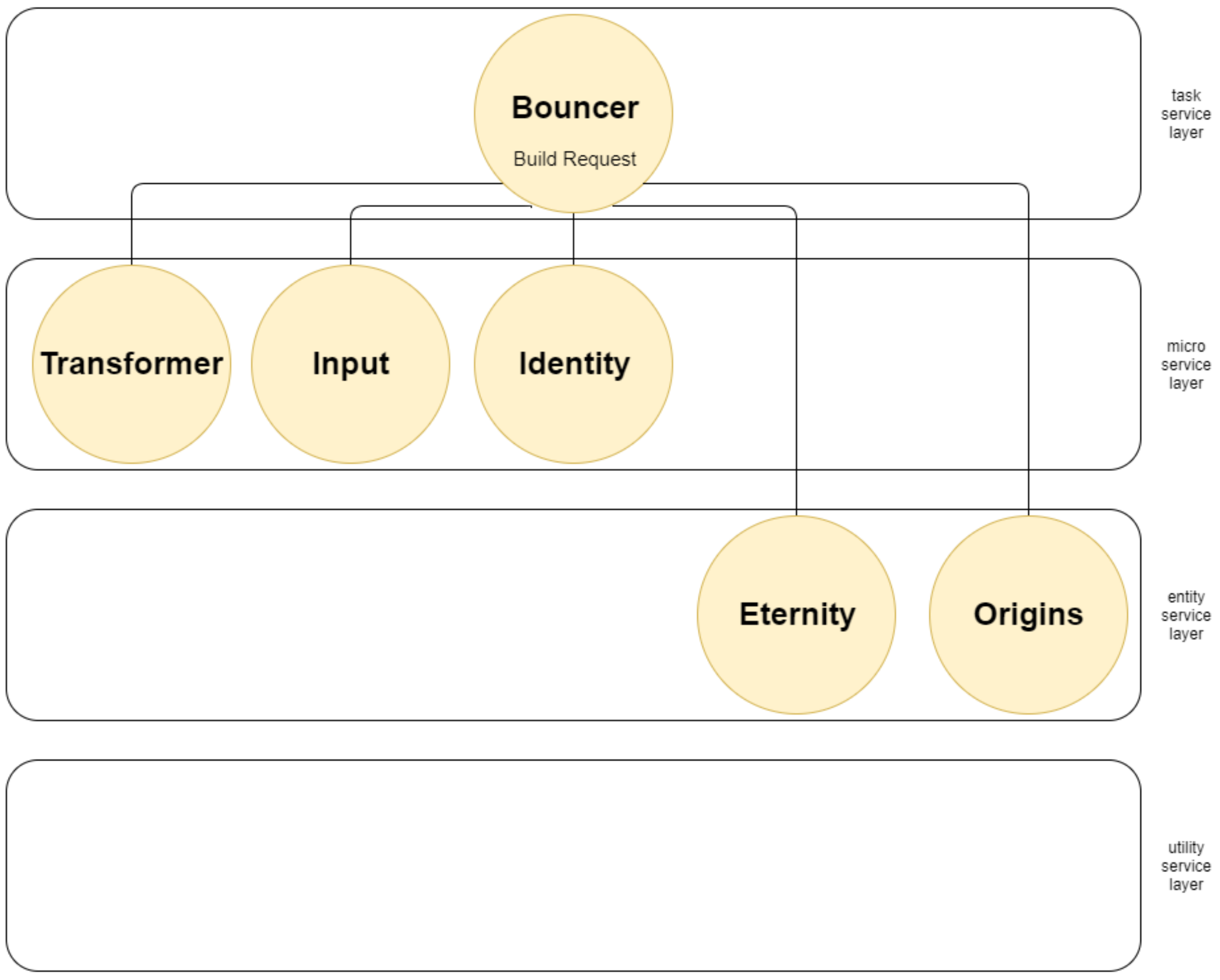
Conceptually, these processes are closely connected. On the other hand, as a consequence of the queued nature of CI systems, there can be large time gaps between the two, that's why they should be discussed as separate processes.

Build request

As the purpose of this document is to give a 50.000 feet view on Brocan, processes will be discussed with limited granularity only. When performing an analysis of processes and separating steps, I've come across the following ones as the backbones of the build request process (these are broken up into smaller steps, but we omit that level of detail here):

1. Receiving and accepting a new changeset/notification from a repository provider.
2. Transforming the previously received notification into an internal format, called Brocan Build Request Format (BBRF, as the kool kids are sayin').
3. Generating a build identifier. This might look like a small step, but actually, this is a crucial one. The build identifier is the unique identifier each build can be queried with. Therefore this step should be executed carefully.
4. Storing the build request.
5. Queueing the build request.

Talking about service layers, the aforementioned steps were materialized as the following services:



Task Service Layer

In the task service layer, orchestrators of several steps can be found. This is completely true for the *Bouncer* service which is responsible for marshaling data from repository providers, down to the persistence. Many services are involved in this process. As you can see, *Bouncer* works in a non-agnostic context.

Microservice Layer

Transformer, *Input* and *Identity* are great examples of microservices. Even though they operate in a non-agnostic context, they are atomic in the sense of functionality. They execute a small step in the business logic, providing the benefit of evolving steps independently from other ones.

Furthermore, they are indeed *micro*. For example, *Identity* is less than 90 LOC!

Discussing the service they provide, *Transformer* produces BBRF objects from repository provider changesets, *Input* queues builds and *Identity* generates build identifiers.

Entity Service Layer

Finally, we've reached the realm of agnostic services, that can be used among several different processes. In the case of build request, this layer is comprised of *Eternity* and *Origins*.

Eternity is responsible for storing and retrieving

- changeset related data (branch, repository, etc.),
- step/command-wise results,
- overall result.

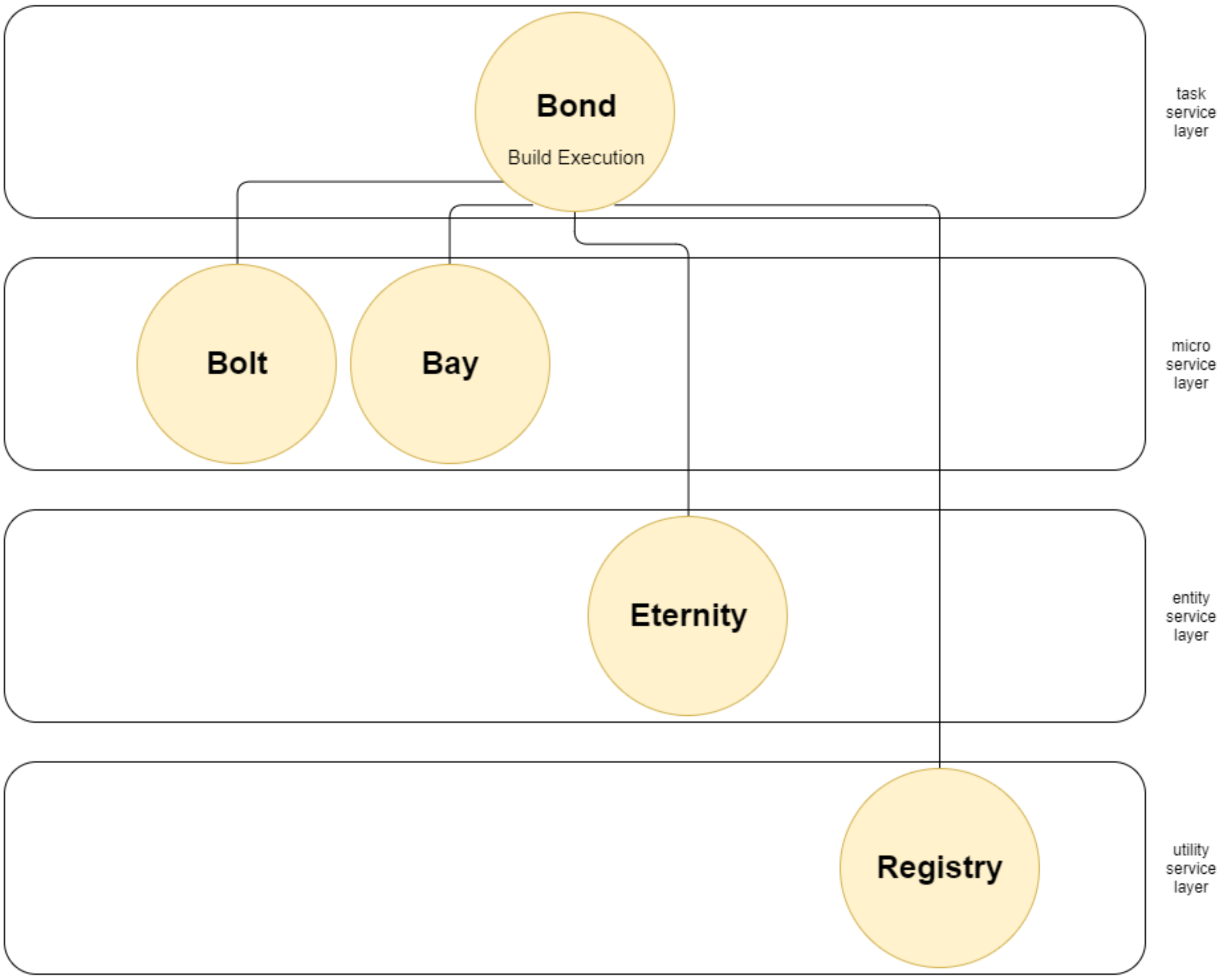
Origins is somewhat different as it was designed with a completely different mindset. Functionality provided by *Origins* could have easily been placed in *Eternity* but was factored out because of performance reasons. As changeset notifications from various repository providers tend to be quite large, and presumably won't be queried too often, it seemed to be a good idea to store them separately. This persistence separate called *Origins* to life.

Build execution

Build execution is much more complex when comparing it to build request. Therefore, I'll be using even broader strokes here, when painting up the big picture about its inner workings. The result of the step breakdown is the following:

1. Querying a new build from the build queue. The prerequisite of this step (and the whole process) is the presence of a free build agent.
2. Cloning the appropriate repository with the appropriate changeset.
3. Starting a new build container with the build runner deployed.
4. Propagating the results into a persistent storage.

Again, it can't be emphasized enough, that these are just some highlights, the key points of the process. Mapping these steps into services yielded the following result:



Task Service Layer

The whole process is supervised by the *Bond* build agent. Basically, the input of this whole task (and in turn, the service) is just a single build identifier. Using this small piece of information (and some other services), *Bond* can successfully carry out a complete build execution process.

Microservice Layer

More granular pieces of functionality locked to this process are provided by the *Bolt* and *Bay* microservices.

Inevitably, *Bolt* is not that micro-, as it's the build runner component, deployed into build containers. However, breaking up this one into smaller services would have impacted performance and would have placed a considerable performance overhead on build executions, there I've opted into writing a single service that's responsible for build running.

In contrast, *Bay* has a much smaller scope, translating user-given build environments into actual docker base image names. This translation is necessary to prevent abstraction leakage. Users should not know about the virtualization technology used in builds.

Entity Service Layer

Here we find *Eternity* again (this proves, that it's really useful in different processes), acting as a storage of the propagated build results, and as a source of build metadata required for repository cloning.

Utility Service Layer

Another important service, which is not even business-motivated, and is completely agnostic, is the *Registry* service, which is a custom docker registry, internal Brocan base docker images.