BROCAN

A microservice-based CI

AGENDA

Preliminaries

CI system and messaging primer.

Architecture and Implementation

How Brocan is built up, and how the components fit together.

What happens when...

The way a build is carried out by Brocan.

PRELIMINARIES

PRELIMINARIES
topics

- What is continuous integration?
- Components of a CI system.
- Why use messaging and why use NATS?
- Message pattern matching.

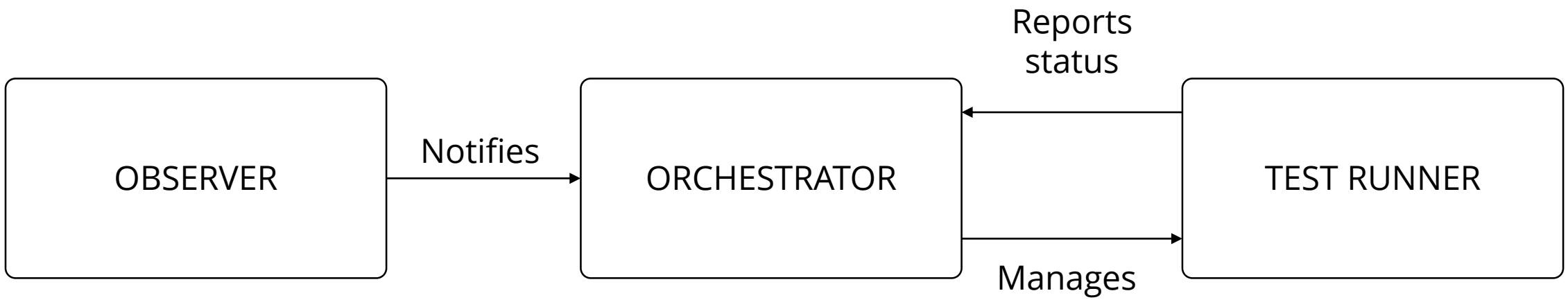
CONTINUOUS INTEGRATION

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Martin Fowler

TRADITIONAL CI ARCHITECTURE

- A traditional CI system contains three core parts:
  - Observer,
  - Orchestrator (or Dispatcher),
  - Test Runner (or Agent).



TRADITIONAL CI ARCHITECTURE
observer

- Checks a specified repository for changes.
- Notifies the Orchestrator upon a change.
- Does not care about
  - the way builds are carried out,
  - the result of the builds,
  - anything that involves more than the concept of code/changeset/repository.

TRADITIONAL CI ARCHITECTURE
orchestrator

- Manages the swarm of Test Runners.
- Handles incoming Observer notifications, queues if necessary.
- Dispatches build requests to Test Runners.
- Propagates build results to other services if necessary.
- Does not care about the way builds are performed.

TRADITIONAL CI ARCHITECTURE
test runner

- Does the heavylifting of the build process.
- Receives build requests from the Orchestrator.
- Reports build results to the Orchestrator.
- Needs some metadata to figure out how testing/building can be done.

MESSAGING
core ideas

- Message-first approach.
  - Messages are more important than the services themselves.
  - The actual sender and receiver do not matter, only the communication itself.
- A service is defined by its incoming and outgoing messages.
  - The way these messages are transported does not matter.
- Can be done with HTTP but works better with custom protocols.

MESSAGING
routing and discovery

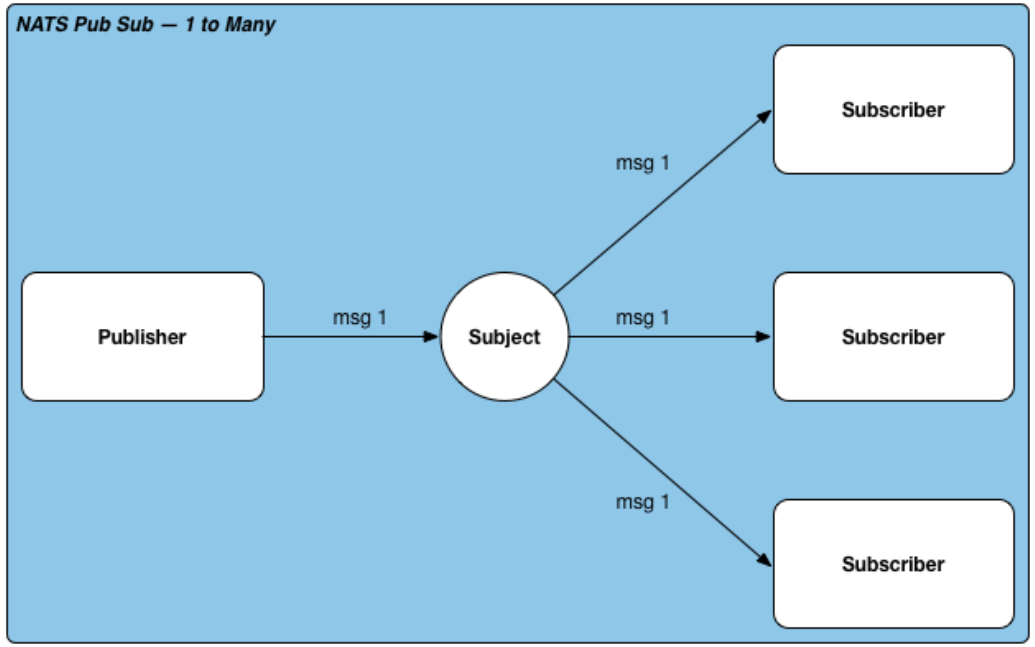
- How to find out where to deliver the messages?
- Many possible options with different trade-offs
  - Predefined IP address of single services.
  - Predefined domain name of single services.
  - Service discovery of single services.
  - Messages brokers (with service discovery or predefined IP/domain).
  - Gossip- or infection-style protocols.

MESSAGING WITH NATS

- NATS is a messaging system managed by a cluster of message brokers.
- High performance (much higher throughput than other brokers).
- Message persistence is not supported out-of-the-box.
- Clients can send messages and subscribe to subjects..
- Various messaging models are supported.

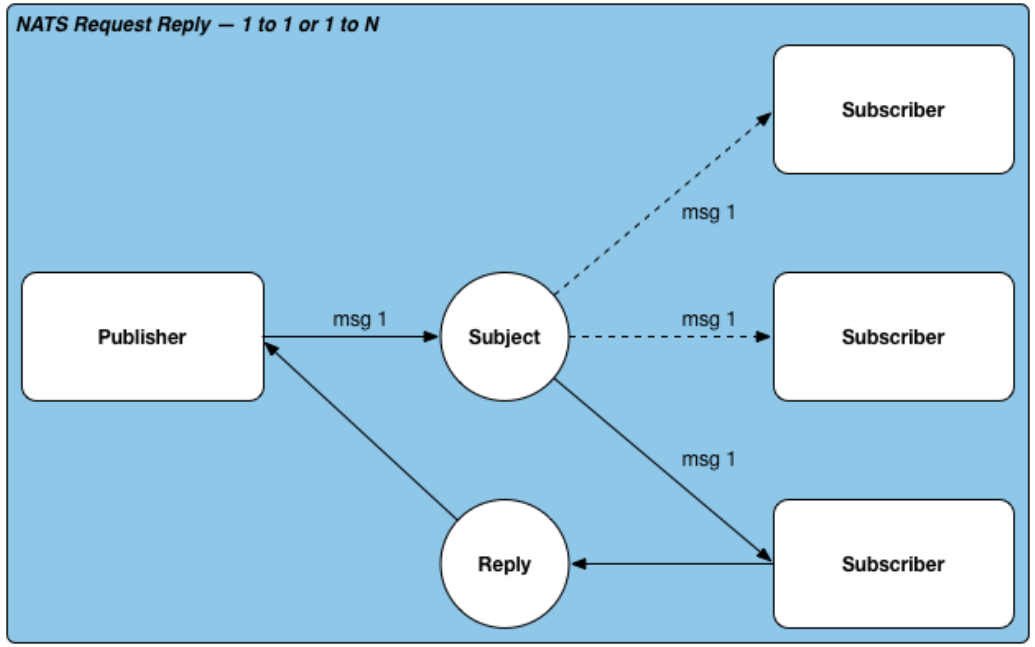
NATS MESSAGING MODELS
publish-subscribe

- One-to-many communication.
- Each subscriber of a specific subject receives the message.



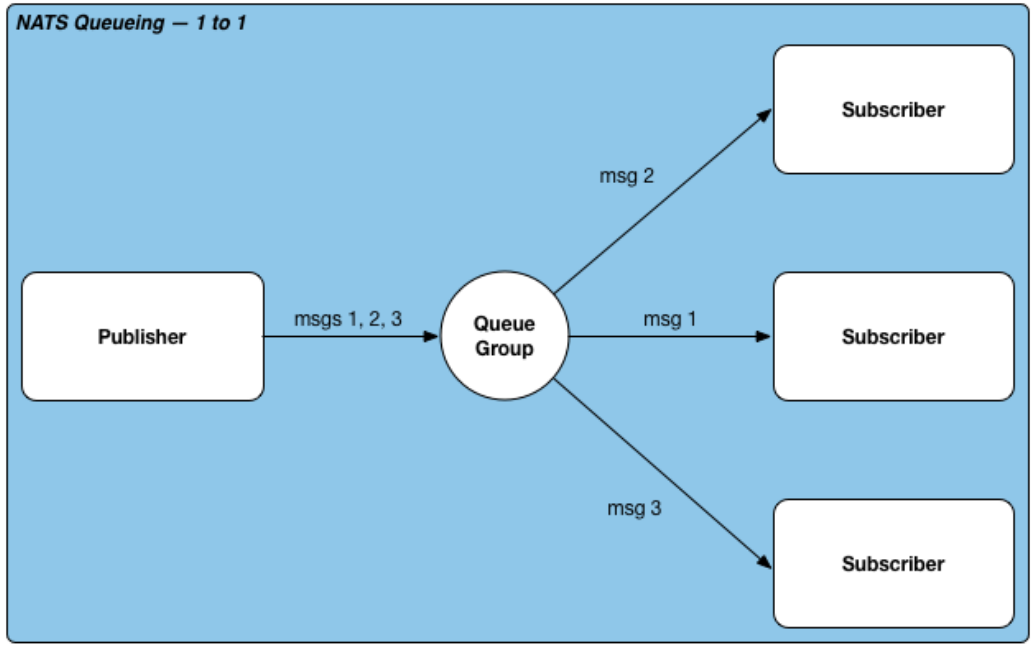
NATS MESSAGING MODELS
request-reply

- One-to-one or one-to-many communication.
- The first (or first N) response is returned.



NATS MESSAGING MODELS
queueing

- Subscribers form a queue group.
- Messages are load-balanced between the subscribers.



PATTERN MATCHING
core ideas

- Services can supply a pattern they expect.
  - Messages will be matched against subscriber patterns.
  - Only matching messages will be delivered.
- There are no endpoints anymore – only message patterns.
- Pattern matching makes it easy to incrementally introduce new functionality, to mix and change behaviour.

PATTERN MATCHING
example

- Let's say, we want to calculate the VAT for a shopping cart.
- Different categories of items have different VAT calculation rules.

PATTERN MATCHING
example

- Imagine, that we publish the following messages:

```
{
  name: The Tao Of Microservices
  category: [ non-food, book, IT ]
  …
}
```

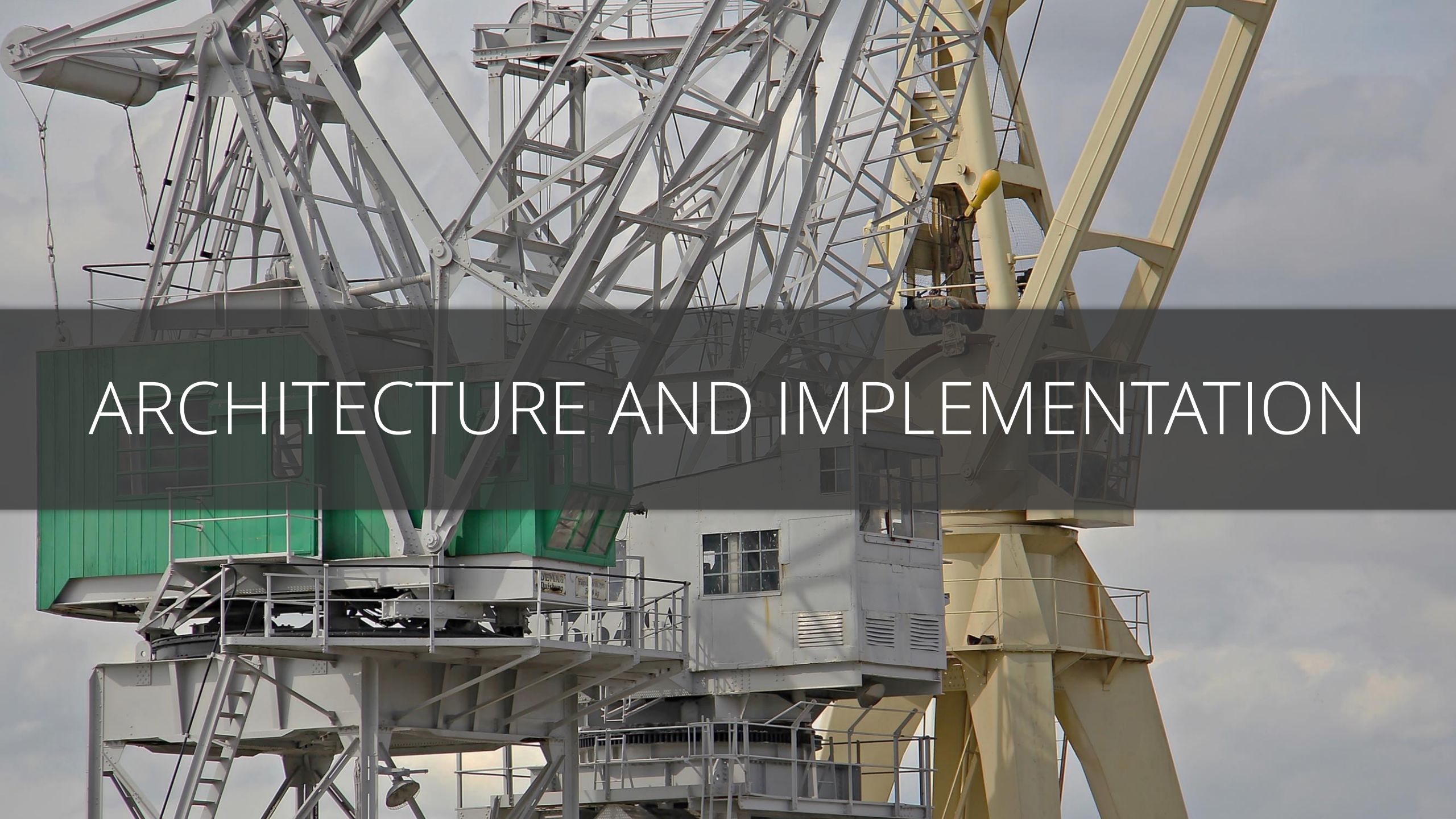
```
{
  name: Birra Moretti
  category: [ food, drink, alcoholic, beer ]
  …
}
```

- Then the following patterns can be introduced:

```
{
  category: book
}
```

```
{
  category: beer
}
```

```
{
  // catch-all
}
```



ARCHITECTURE AND IMPLEMENTATION

ARCHITECTURE AND IMPLEMENTATION topics

- Inverted CI architecture.
- Implementation challenges.
- Pattern matching in action.

INVERTED CI ARCHITECTURE

- Externalized Observer.
  - Notifications are supplied by repository providers (eg. GitHub).
- The absence of a central Orchestrator, that's the inversion.
  - Test Runners are not given builds, but take them.
  - The Orchestrator is reduced into a build queue.

ARCHITECTURE ANGLES
message-first

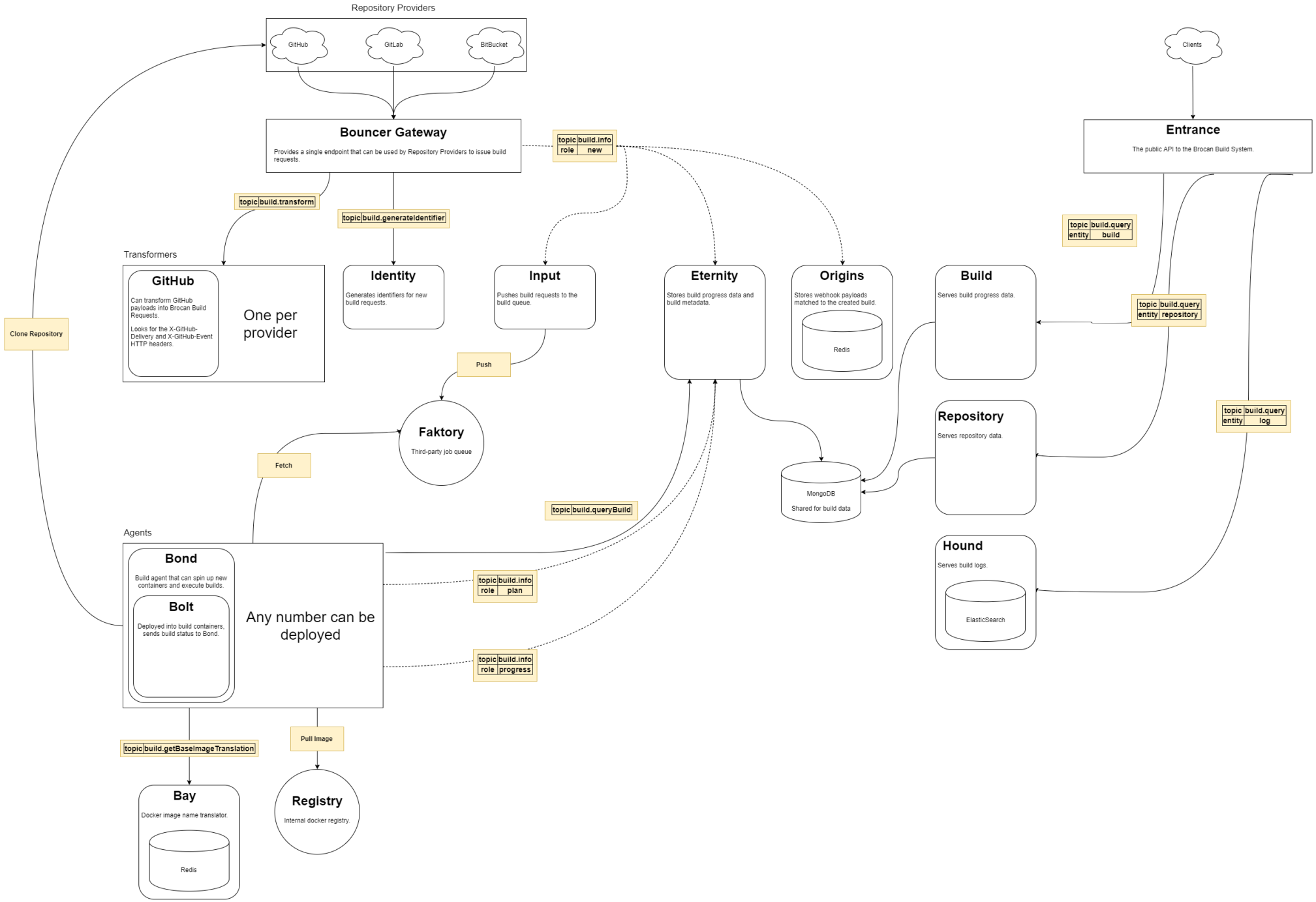
- There's no direct dependency between services.
  - They are just producers and consumers of messages.
- Processes are formed by message flows.
- In Brocan, the only services that have knowledge of each other are the Test Runner and the Agent services.
  - They utilize a JSON HTTP API.

ARCHITECTURE ANGLES
disposable

- Inspiration from Richard Rodger's book, The Tao of Microsevices.
- Services are designed to be disposable.
- Every service is
  - small,
  - concentrated,
  - and atomic.
- The largest service is 805 LOC!

ARCHITECTURE ANGLES
evolution and dynamism

- Being message-first and disposable embraces that individual services are not important.
- This leads to the importance of the processes and flows.
- As business requirements and processes change
  - the system can be easily adjusted and thus evolve,
  - services can be thrown out, merged and introduced on demand.
- In conclusion, the system becomes a living thing, with the messaging system as its nervous system.


Repository Providers
GitHub
GitLab
BitBucket
Clients
Bouncer Gateway
Provides a single endpoint that can be used by Repository Providers to issue build requests.
topic build.info
role new
Entrance
The public API to the Brocan Build System.
topic build.transform
topic build.generateIdentifier
topic build.query
entity build
Transformers
GitHub
Can transform GitHub payloads into Brocan Build Requests.
Looks for the X-GitHub-Delivery and X-GitHub-Event HTTP headers.
One per provider
Identity
Generates identifiers for new build requests.
Input
Pushes build requests to the build queue.
Eternity
Stores build progress data and build metadata.
Origins
Stores webhook payloads matched to the created build.
Redis
Build
Serves build progress data.
topic build.query
entity repository
Clone Repository
Push
Faktory
Third-party job queue
Fetch
MongoDB
Shared for build data
Repository
Serves repository data.
topic build.query
entity log
topic build.queryBuild
Agents
Bond
Build agent that can spin up new containers and execute builds.
Bolt
Deployed into build containers, sends build status to Bond.
Any number can be deployed
topic build.info
role plan
topic build.info
role progress
Hound
Serves build logs.
ElasticSearch
topic build.getBaseImageTranslation
Pull Image
Bay
Docker image name translator.
Redis
Registry
Internal docker registry.

IMPLEMENTATION CHALLENGES
achieving distilled business logic

- Distilled business logic refers to services that
  - are only concerned with business tasks,
  - do not care about circuit breaking, service discovery, log transportation etc.
- Nearly impossible to achieve but worth to strive for.
- Resolution for Brocan:
  - Passed with issues, as some non-business code is contained within each service.

IMPLEMENTATION CHALLENGES
multilingualism

- A system should not be framework- or language-locked
  - That would break the „Use right tool for the right job.” principle.
- Regarding the transport, Brocan is locked to NATS.
  - This is not an issue, since NATS has client libraries in multiple languages.
- However, pattern matching and other advanced features are provided by HemeraJS.
  - Therefore Brocan is locked to HemeraJS.

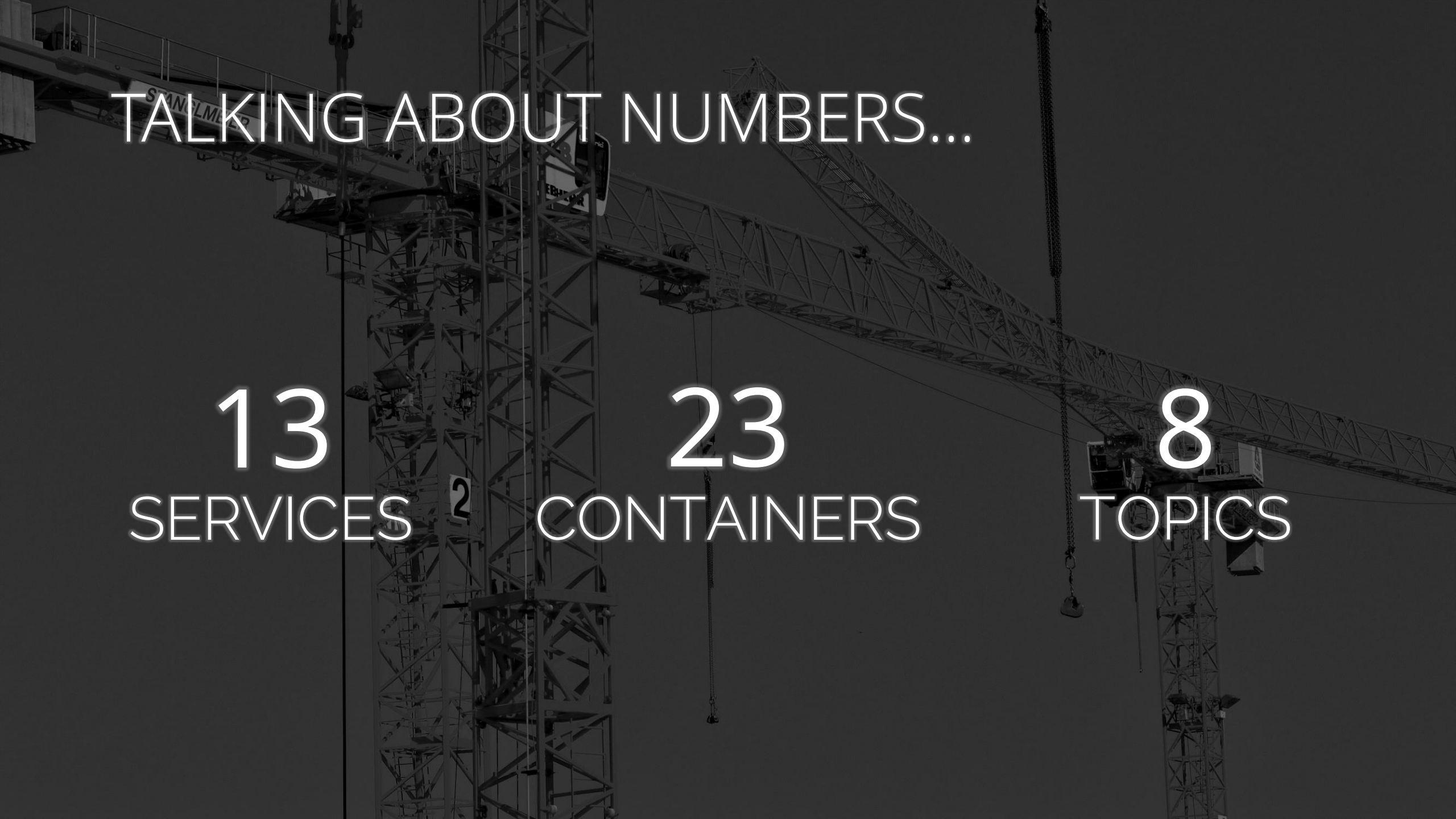
PATTERN MATCHING IN ACTION
webhook transformers

- Repository providers (such as GitHub) can be integrated with transformers.
- Transformers produce the internal BBRF format.
- New transformers can be added with zero infrastructure change thanks to pattern matching.

PATTERN MATCHING IN ACTION
webhook transformers

- Currently only GitHub is integrated, using the following pattern:

```
{
  webhookRequest: {
    headers: {
      x-github-delivery: /.*/,
      x-github-event: /.*/
    }
  }
}
```



TALKING ABOUT NUMBERS...

13
SERVICES

23
CONTAINERS

8
TOPICS

WHAT HAPPENS WHEN...

WHAT HAPPENS WHEN...
topics

- How a build is performed.
- The brocanfile format.
- What gets stored in the DB.

BUILD PROCESS

- It can be seen as two distinct processes:
  - Build Request – Processing and putting a notif into the queue.
  - Build Execution – Actually performing a queued build.
- Prerequisites:
  - The targeted repository contains a brocanfile.
  - A proper webhook is set up for push events.

BUILD PROCESS
brocanfile

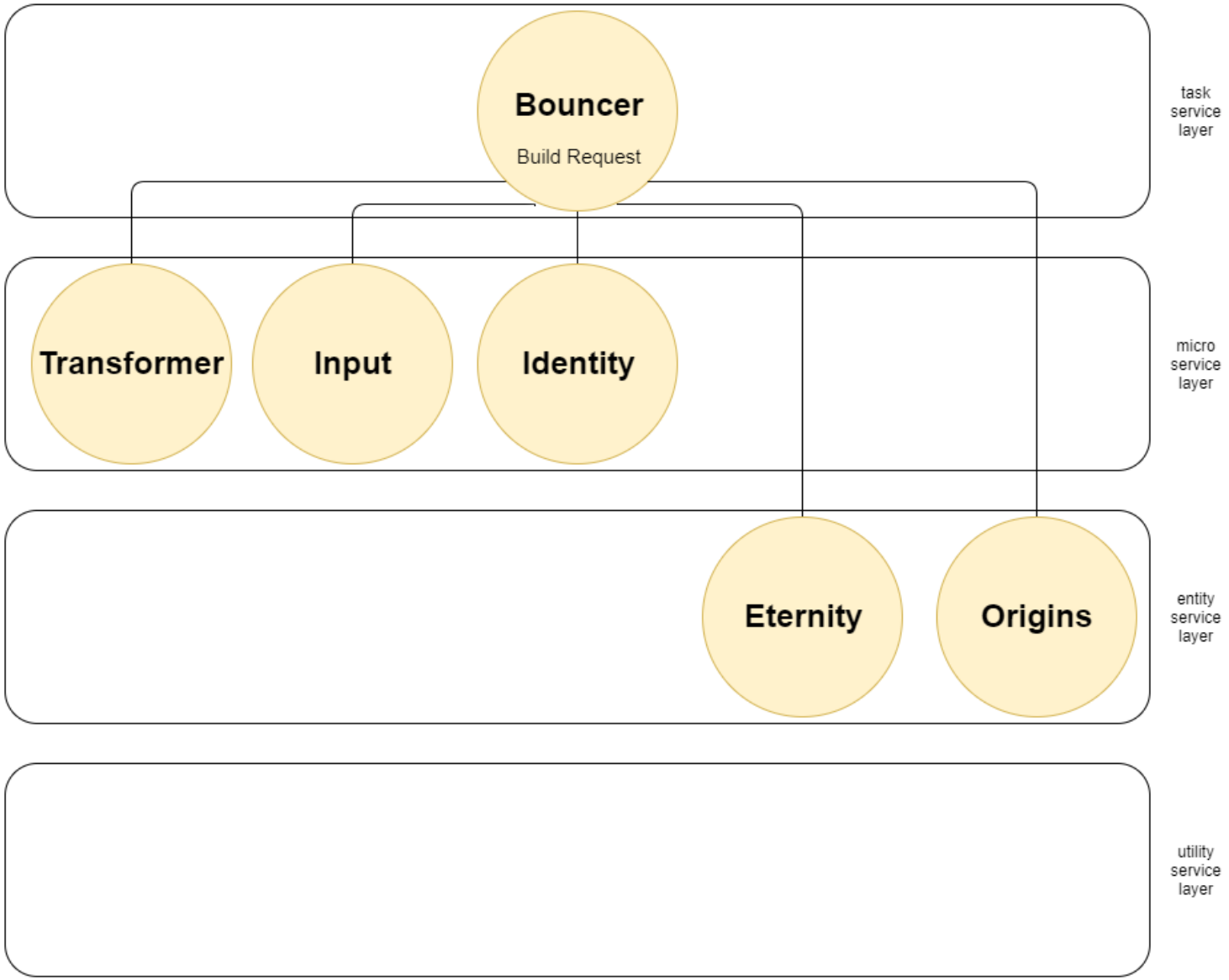
- Based on the HJSON format.
- Properties
  - Base: The environment to run the build in.
  - Owner: The owner of the project.
  - Steps: Step definitions with commands.

```
{
  base: maven
  owner: battila7
  steps: [
    {
      name: compile
      commands: [
        echo start
        mvn clean compile
      ]
    }
    {
      name: test
      commands: [
        mvn test
      ]
    }
  ]
}
```

BUILD PROCESS
request

1. New notification received from a repository provider.
2. Notification is transformed into an internal BBRF object.
3. A new Build Identifier is generated for the BBRF.
4. The build is stored and is put into the build queue.

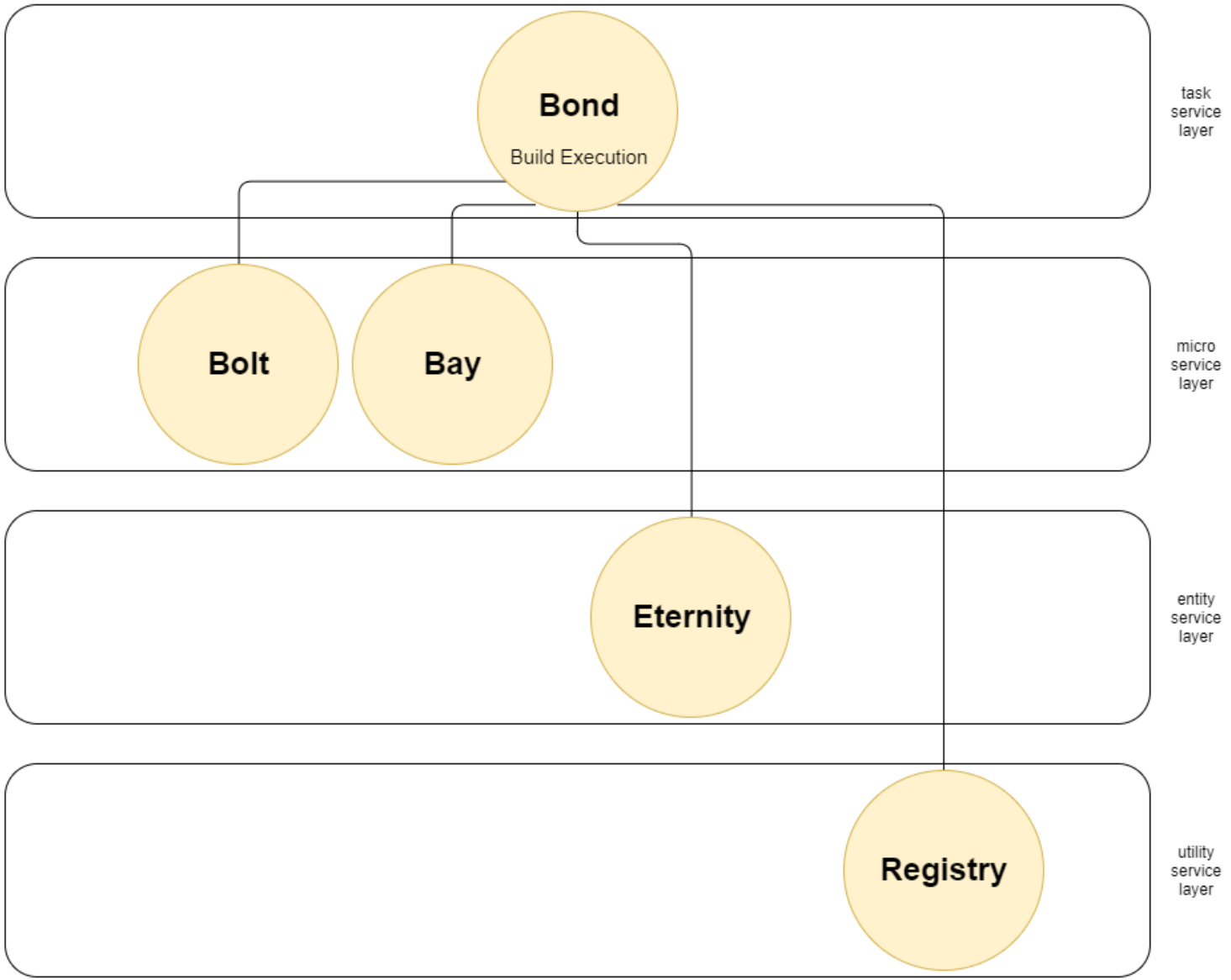
BUILD PROCESS
request



BUILD PROCESS
execution

1. Once an agent frees up, it queries a build from the queue.
2. The appropriate repository (at the state of the changes) is cloned.
3. A new build container is started, with the runner deployed.
4. As the build container exits, results are propagated into the database.

BUILD PROCESS
execution



MONGODB
document structure

_id	Calculated from the commit hash, timestamp and branch name.
author	The user/entity who has pushed the changes.
branch	The branch the changes reside on.
commit	Head commit of the changes.
repository	Data related to the containing repository of the changes.
execution	Results of the step and command executions.

MONGODB example document


```
{
	"_id" : "1abb95fd469c67969e8f6a02c705a71c0ea0e1ab4410625338e94a93383935ca",
	"timestamp" : 1511208107436,
	"author" : {
		"name" : "battila7",
		"username" : "battila7",
		"uri" : "https://github.com/battila7"
	},
	"branch" : {
		"name" : "master",
		"uri" : "https://github.com/battila7/cd2t-100/tree/master"
	},
	"commit" : {
		"hash" : "30d7dca8936847b83c2e908b3f0c2e38473866a7",
		"message" : "Updated the brocanfile to test Brocan",
		"uri" : "https://github.com/battila7/cd2t-100/commit/30d7dca8936847b83c2e908b3f0c2e38473866a7"
	},
	"repository" : {
		"name" : "cd2t-100",
		"uri" : "https://github.com/battila7/cd2t-100"
	},
	"execution" : {
		"steps" : [
			{
				"name" : "compile",
				"commands" : [
					{
						"command" : "echo start",
						"status" : "successful"
					},
					{
						"command" : "mvn clean compile",
						"status" : "successful"
					}
				],
				"status" : "successful"
			},
			{
				"name" : "test",
				"commands" : [
					{
						"command" : "mvn test",
						"status" : "failed"
					}
				],
				"status" : "failed"
			}
		],
		"status" : "failed"
	},
	"startedAt" : 1511208127653,
	"finishedAt" : 1511208194076
}
```


DEMONSTRATION

Q&A


THANKS FOR YOUR ATTENTION!