BROCAN

A microservice-based CI

PRELIMINARIES

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,
  - Dispatcher (or Orchestrator),
  - Test Runner (or Agent).

TRADITIONAL CI ARCHITECTURE
observer

- Checks a specified repository for changes.
- Notifies the Dispatcher 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
dispatcher

- Manages the cluster 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 Dispatcher.
- Reports build results to the Dispatcher.
- Needs some metadata to figure out how testing/building can be done.

MESSAGING WITH NATS

- NATS is a messaging system managed by a cluster of message brokers.
- Clients can send messages and subscribe to topics.
- Subscribers are free to process messages in a sync or async fashion.
- NATS supports out-of-the-box
  - Request-Reply (point-to-point, one-to-many)
  - Publish-Subcribe (one-to-one, one-to-many)

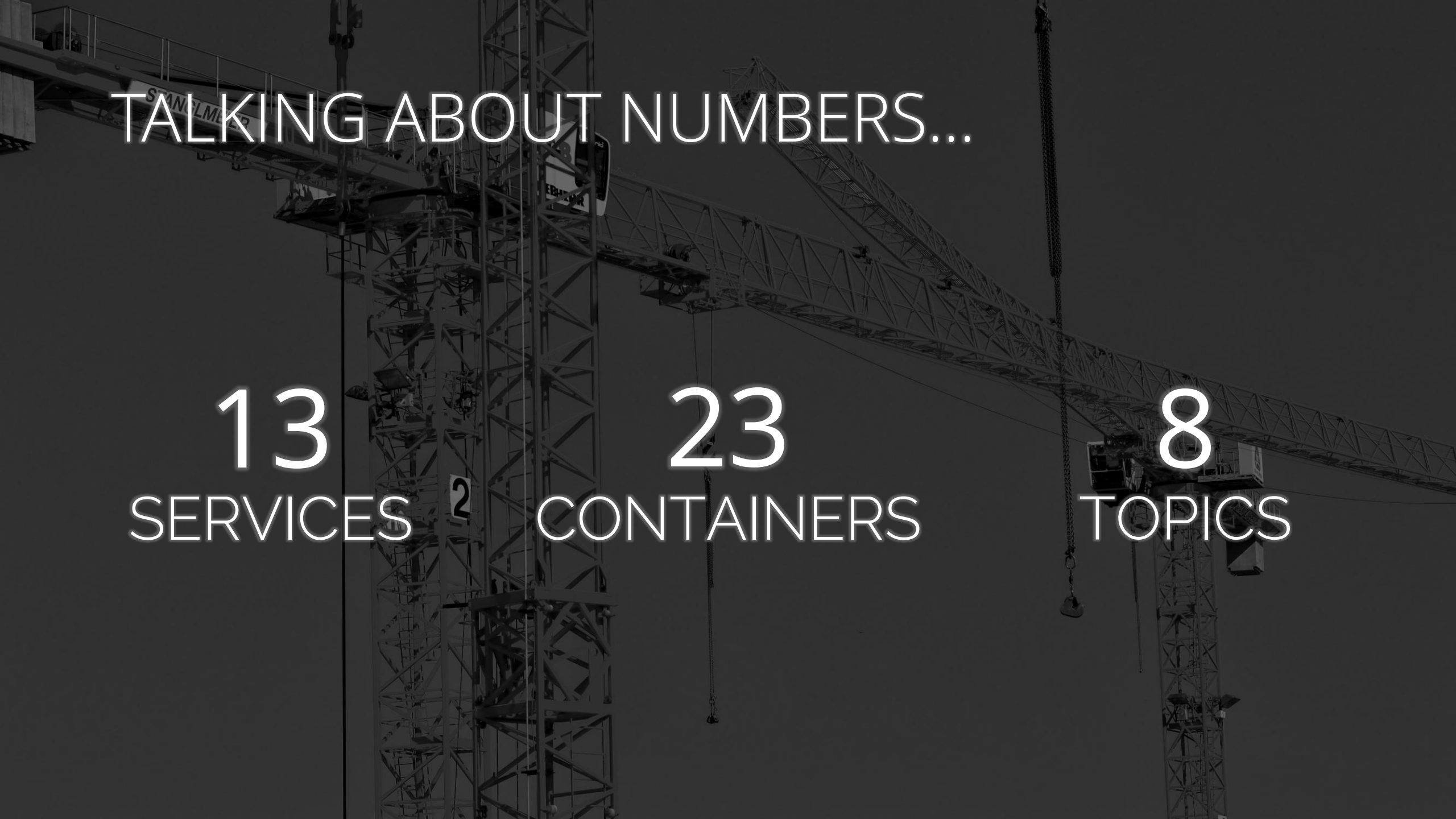
MESSAGING WITH NATS
pattern matching

- Actually provided by Hemera on top of NATS.
- Subscribers can supply a pattern they expect.
  - Messages will be matched against subscriber patterns.
  - Only matching messages will be delivered.
- Pattern matching makes it easy to incrementally introduce new functionality, mix and change behaviour.

ARCHITECTURE

INVERTED CI ARCHITECTURE

- There is no Observer.
  - Notifications are supplied by repository providers (eg. GitHub).
- There is no Dispatcher, that's the inversion.
  - Test Runners are not given builds, but take them.
  - The Dispatcher is reduced into a build queue.

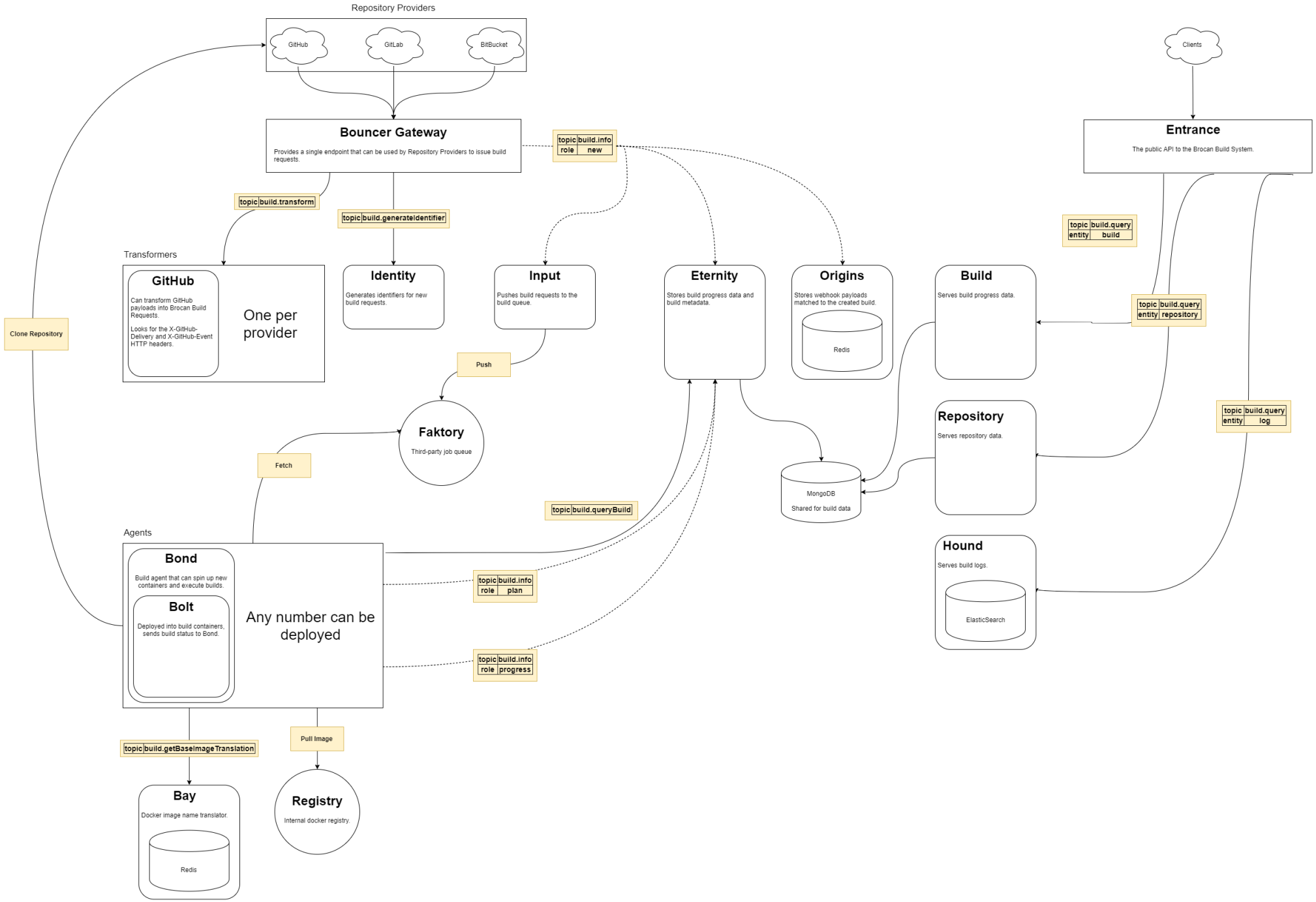


TALKING ABOUT NUMBERS...

13
SERVICES

23
CONTAINERS

8
TOPICS


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.

BUILD PROCESS

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.
- Exercises ~70% of the services.
- The number of concurrent builds is equal to the number of agents.

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
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.

PERSISTENCE

PERSISTENCE PROVIDERS

MongoDB

Stores build execution results and other build related metadata.

Redis

Used as a simple and fast K/V storage for image name resolution and request persistence.

Elasticsearch

The sink for every log produced by either the internal services or the build executions.

HOW MONGODB IS USED

- There is only one collection: builds.
- A custom ObjectId is used that is a hash of some unique fields.
- Nesting is used heavily because of a number of reasons:
  - Nested data won't ever change, therefore safe to inline.
  - In order to support presumably frequent queries.
  - Speed of development and ease of maintenance.

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