example

Table of Contents

1. Context

If you encounter any error, feel free to enter an issue on GitHub.

1.1. Context diagram

kafkatrain - System Context

Dot Executable: /opt/local/bin/dot
File does not exist
Cannot find Graphviz. You should try

@startuml
testdot
@enduml

or

java -jar plantuml.jar -testdot

Systems involved in train prediction

1.2. What is this softare system about ?

Getting "true" public transport delay is often impossible. Because the organism responsible for displaying delays is also the organism that allow transports to run. As a consequence, it is not always their best move to display effective transport delay. This system exists to overcome that.

1.3. What is it that’s being built?

We will build a system that allow us to easily compare official time table with effective transport

delay by

1. Asking users in transports to tell us if transport is late or not
2. Comparing effective transport location with the one provided by real-time location, when it exists

1.4. How does it fit into the existing environment? (e.g. systems, business

processes, etc)

As we're doing this as a startup, we have no internal context. However, there is an external context.

We will use services of navitia, which provides time tables for all France cities public transport systems, but also intercity train.

We will also use geolocation services provided by SNCF (for intercity trains) and other providers wher possible.

1.5. Who is using it? (users, roles, actors, personas, etc)

We currently envision two types of useers (who can in fact be the same person at different times)

1. The waiting user, to which we will send accurate crossing schedule information.
2. The user already in transport, which can inform waiting user if the train was on time or not

2. Functional Overview

If you encounter any error, feel free to enter an issue on GitHub.

This system allows a person waiting for a public transport to have informations on transport schedule, as provided by people upstream in the same public transport. Imagine that as a crowdsourced SMS from a friend.

Features are quite simple.

2.1. For a waiting person

When someone waits for a transport, kafkatrain detects (from user location and timetables) which are the possible transports the person wants to go in. If multiple transports matches, the application allows user to select which transport he is waiting for. Once a transport has been selected, information from upstream users is presented. This information will typically take the form of

> At stop "name of stop", transport was "on time|late by n minutes"

As of today, we don't envision the UI of application

2.2. For someone in a transport

Once the user is in the public transport, and the transport is moving, application simply sends a message notifying the system that transport is on way.

3. Quality Attributes

If you encounter any error, feel free to enter an issue on GitHub.

3.1. Common constraints

Performance (e.g. latency and throughput)

All users should have application available in 5 seconds.

Scalability (e.g. data and traffic volumes)

We expect a first deployment with 1.000 users (and 100 simultaneous connections of users)

Availability (e.g. uptime, downtime, scheduled maintenance, 24x7,99.9%, etc)

Application will initially be available on a 99.9% basis

Security (e.g. authentication, authorisation, data confidentiality, etc)

No user data should be stored by system. Authentication and authorization will be managed using OpenID Connect with the usual id providers (Google, Facebook, ...)

Extensibility

Application will be extended to various geographic areas and types of public transport systems (buses, trains, boats), but there will be no feature extensibility

Flexibility

Application is not supposed to be flexible

Auditing

We must be able to audit the timetables provided by Navitia as well as the real-time positions. We must also be able to audit what informations users in transit send to blacklist the ones that will (because shit happens) try to abuse the system.

Monitoring and management

Usual system monitoring will be used.

Reliability

Besides, we will monitor the number of users in transit and waiting and the delay between the time when one user starts his transit and another, on the same line, receive the delay information.

Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)

We should be able to recover data center loss in less than one day.

Business continuity

N/A

Interoperability

N/A

Legal, compliance and regulatory requirements (e.g. data protection act)

N/A

Internationalisation (i18n) and localisation (L10n)

Application will be deployed in all countries where public transport systems provide APIs (and have potential delays). For the sake of easiness, application will be first deployed in France.

Accessibility

Don't know how to validate that.

Usability

Don't know how to validate that.

3.2. Specifc constraints

These constraints maps to the relationships expressed in Context

3.2.1. Ingesting time tables

Ingesting time tables from Navitia should be done each day. The process should be monitored since there should be no day where this data is not ingested. THis ingestion should be done prior the first release (application has no interest without that).

3.2.2. Ingestion real-time positions

This should be done in a continuous stream. Ingestion should not have delay greater than one minute. Application should be able to work without this information.

3.2.3. Seeing train delays

Delays should be communicated to user in less than 1 s. If user connection to system dont allow that, delays will be sent to user with a notification indicating that network is not performant enough to have accurate timetables.

3.2.4. Informing application that transport is running

A transport is considered as moving after 5 seconds of continuous move. After this delay, signal should be sent in less than 1 s.

4. Constraints

If you encounter any error, feel free to enter an issue on GitHub.

4.1. Common constraints

Time, budget and resources

Project will be built by Nicolas Delsaux and Logan Hauspie. Budget is zero, as it is an example. Application is expected to be delivered … one day

Approved technology lists and technology constraints

Server-side components of application will be depolyed as containers.

Target deployment platform

Application will be deployed on Google Kubernetes Cluster.

Existing systems and integration standards

TODO

Local standards (e.g. development, coding, etc)

TODO

Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc)

TODO

Standard protocols

TODO

Standard message formats

TODO

Size of the software development team

Two persons at best

Skill profile of the software development team

Developers are skilled on server-side, less on front-end.

Nature of the software being built (e.g. tactical or strategic)

Strategic, as it is the only product of our startup.

Political constraints

TODO

Use of internal intellectual property

TODO

5. Principles

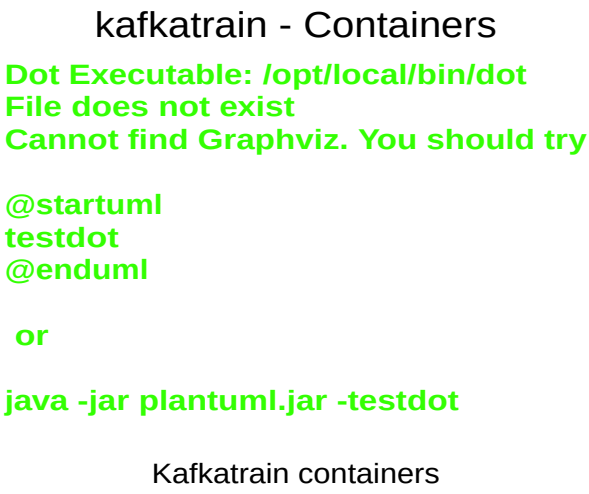
If you encounter any error, feel free to enter an issue on GitHub.

Team will adhere to the following set of principles.

- As it is an example project, we follow the programming, motherfucker methodology.
- There should be no operationnal management cost, so
  - Application should be auto-redeployed
  - Application should be self-healing
- Application should use messaging and async systems as much as possible
- Interfaces between components should be documented

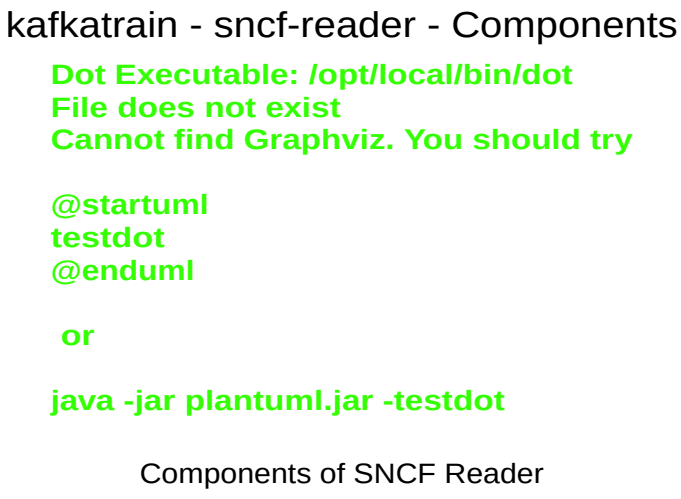
6. Software Architecture

If you encounter any error, feel free to enter an issue on GitHub.



We use Kafka to fully isolate load between sncfReader and storage. We use ElasticSearch to provide various search directions, as we will request search based on geographic criterais as well as proximity.

6.1. Software architecture of sncfReader component



This one is quite simple : one verticle reads data from Navitia HTTP endpoint, send the obtained data through Vert.x event bus to another which outpus data to a Kafka stream.

7. Code

If you encounter any error, feel free to enter an issue on GitHub.

TODO

7.1. kafkatrain

See on GitHub

7.1.1. Avoir un train à l'heure, c'est kafkaïen

> Repository principal de notre présentation à Snowcamp 2019

What does this repository contains ?

- `src/build` contains various build scripts
  - `0-install.sh` installs the environment, provided the secrets are known
  - `1-write-reader-code.bat` copies reader code in its own repository
  - `2-write-web-ui.bat` copies web ui in its own repository
  - `delete.bat` deletes the cluster, and the various generated projects
- `src/k8s` contains all deployed into k8s cluster
  - `elastic` provides ingresses for Kibana and Elasticsearch (**DON'T DO THAT IN PROD**)
  - `kafka` installs all additionnal applications for kafka

Meta

- Logan Hauspie – @lhauspie

- Nicolas Delsaux – @Riduidel

Contributing

1. Fork it (<https://github.com/Riduidel/snowcamp-2019/fork>)
2. Create your feature branch (`git checkout -b feature/fooBar`)
3. Commit your changes (`git commit -am 'Add some fooBar'`)
4. Push to the branch (`git push origin feature/fooBar`)
5. Create a new Pull Request

7.1.2. sncf-reader

See on GitHub

sncf-reader application

This application allows us to inject SNCF timesheets into our Kafka engine, for later processing.

Configuration

This application requires the following environment variables to be set

- `SNCF_READER_TOKEN` access token for Navitia API
- `SNCF_READER_READ_AT_STARTUP` When set to true, immediatly start reading SNCF timesheet
- `SNCF_READER_KAFKA_BOOTSTRAP_SERVER` url of Kafka server to connect to
- `SNCF_READER_TOPIC_SCHEDULE` Topic where to post schedule. Defaults to `sncfReaderSchedule`

7.1.3. web-ui

See on GitHub

node

Simple Hello World that listens on localhost:8080

8. Data

If you encounter any error, feel free to enter an issue on GitHub.

TODO

9. Infrastructure Architecture

If you encounter any error, feel free to enter an issue on GitHub.

TODO

10. Deployment

If you encounter any error, feel free to enter an issue on GitHub.

TODO

11. Development Environment

If you encounter any error, feel free to enter an issue on GitHub.

TODO

12. Operation and Support

If you encounter any error, feel free to enter an issue on GitHub.

TODO

13. Decision Log

If you encounter any error, feel free to enter an issue on GitHub.

13.1. Use ADR to document architecture decisions

Date: 2020-01-14

13.1.1. Decision

We will store decisions as asciidoc documents in `src/docs/asciidoc/decisions`.

13.1.2. Status

Accepted

13.1.3. Context

The purpose of this section is to simply record the major decisions that have been made, including both the technology choices (e.g. products, frameworks, etc) and the overall architecture (e.g. the structure of the software, architectural style, decomposition, patterns, etc). For example:

- Why did you choose technology or framework "X" over "Y" and "Z"?
- How did you do this? Product evaluation or proof of concept?

- Were you forced into making a decision about "X" based upon corporate policy or enterprise architecture strategies?
- Why did you choose the selected software architecture? What other options did you consider?
- How do you know that the solution satisfies the major quality attributes?
- …

13.1.4. Alternatives

Use decisions in Asciidoc

PROS

- Decisions will be integrated in generated documentation
- Rendering will be optimal

CONS

- Decisions will be burried deep inside Asciidoc hiearchy

Use another format

PROS

- Decisions can be put in a specific folder
- We can use "tooling" to access those decisions

CONS

- Decisions will be isolate from architecture documentation
- Decision rendering will use another tooling (typically Markdown)

13.1.5. Consequences

Integrating decisions in Asciidoc requires changing a section of architecture.