User Manual

Real-time Geospatial Data Processor and Visualiser

Client: Werner Raath

COEUS

Members:
Nsovo Baloyi 12163262
Maluleki Nyuswa 13040686
Keletso Molefe 14222583
Kamogelo Tswene 12163555

09 September 2016
v0.3

Contents

Document History

Version	Date	Changed By	Summary
v0.1	27 May 2016	Nsovo Baloyi Maluleki Nyuswa Keletso Molefe Kamogelo Tswene	First Draft
v0.2	29 July 2016	Nsovo Baloyi Maluleki Nyuswa Keletso Molefe Kamogelo Tswene	Second Draft
v0.3	09 September 2016	Nsovo Baloyi Maluleki Nyuswa Keletso Molefe Kamogelo Tswene	Third Draft

1 Introduction

This purpose of this document is to provide a detailed summary of the architectural requirements for the Real-time Geospatial Data Processor and Visualiser system in a technology neutral design specification. The specification shows how the system components will communicate with each other and through what means.

2 Software Architecture Overview

Figure 1 shows a high-level overview of the software architecture. In particular, it shows the decomposition of the system into layers with abstract responsibilities, the core architectural components of the system and the concrete frameworks to be used when realizing these architectural components.

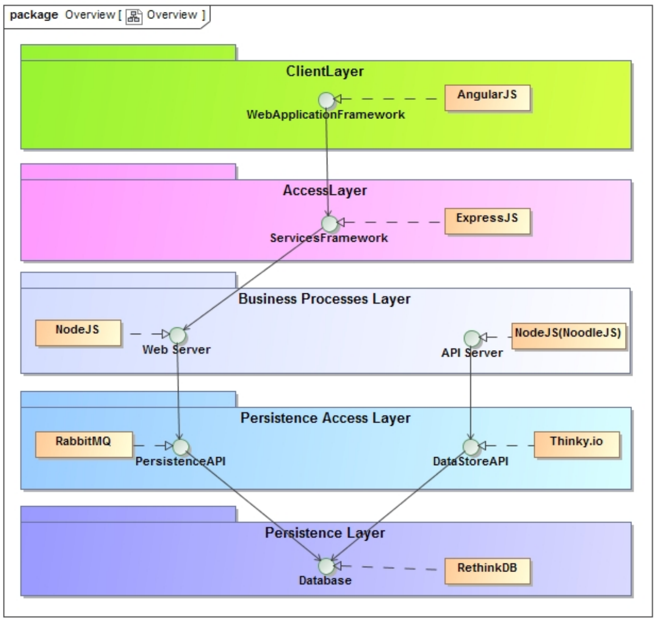


Figure 1: A high-level overview of the software architecture of the Real-time GeoSpatial Data Processor and Visualizer

3 Architecture Requirements

This section specifies the software architecture requirements and the software architecture design for the system as a whole. The output will be the high level software architecture components, the infrastructure between them and the tactics that will be used to realize the quality requirements for the system.

3.1 Architectural Scope

In this section discuss architectural responsibilities which need to be addressed by the software architecture. Typical examples include those of providing a persistence infrastructure (e.g. database), providing a reporting infrastructure, providing an infrastructure for process execution,

3.2 Quality Requirements

The quality requirements are the requirements around the quality attributes of the systems and the services it provides. This includes requirements like Flexibility, maintainability, scalability, performance, reliability, security, auditabilty, testability, usability, integrability and deployability requirements.

3.2.1 Flexibility

It is important that the system architecture is designed in such a way that one can easily add different access channels to the system as well a remove old unused or outdated access channels. Furthermore, persistence architectures are evolving at a great rate. The growth of NoSQL databases such as MongoDB and RethinkDB servers as proof of this. In this context it is important that the application functionality is not locked into any specification persistence technology and that one is able to easily modify the persistence provider.

3.2.2 Maintainability

Amongst the most important quality requirements for the system is maintainability. It should be easy to maintain the system in the future. To this end

- future developers should be able to easily understand the system,
- the technologies chosen for the system an be reasonably expected to be available for a long time,
- and developers should be able to easily and relatively quickly change aspects of the functionality the system provides.

3.2.3 Scalability

The purpose of this system is to be used in the business of disaster management, it should allow for easy scaling in all layers. The system must adhere to the following requirements

- The database should support high availability of data,
- messaging should succeed for every CRUD database operation,
- and front-end maps should be able to display the geo-spatial information queried by the user.

3.2.4 Performance

Performance is amongst the most important quality requirements for this system. The following requirements must be considered

- messages should be small and only supply the most important data, e.g. descriptions and primary keys that could be used by the client to pull data from the API,
- Front-end maps should react responsively without lag between frames,
- the information must be streamed to the user in real-time.

3.2.5 Reliability

A reliable system allows users to use the system with ease. Users should feel that the system is reliable, to this end

- client connections to messaging services should never break,
- server and client exceptions should be handled gracefully.

3.2.6 Security

Initially the system needs to support only

- Users should simply log in and see data relevant to their area of interest.
- HTTPS connections are optional.

In future the system is expected to also enforce confidentiality through encrypted communication and protection against man-in-the-middle attacks through hashing, protect against DOS and DDOS attacks that will stress the servers, and authentication against a chosen user repository (for users who will deploy troops)

3.2.7 Auditability

The system will log all messages processed by the system including all requests and all responses for all user services provided by the system. For each request and response entries the following will be logged

- Request entries:
  - an id for the log entry,
  - the userId of the user requesting the service,
  - the date/time stamp when the request was made,
  - the user service requested, and
  - the request object stringified as JSON with any sensitive information removed.
- Response entries:
  - an id for the log entry,
  - the id of the corresponding request entry,
  - the date/time stamp when the response is provided, and the response object stringified as JSON with any sensitive information removed.

The system will provide only services to extract information from the audit log and will not allow the audit log to be modified. Audit logs will be directly accessible to both, humans and systems.

3.2.8 Testability

All services offered by the system must be testable through

1. unit tests,
2. and integration tests

In either case, these tests should verify that

- all pre-conditions are met (i.e. that no exception is raised except if one of the pre-conditions for the service is not met), and
- that all post-conditions hold true once the service has been provided.

In addition to functional testing, the quality requirements like scalability, usability, auditability, performance and so on should also be tested.

3.2.9 Usability

Usability is an important quality requirement to consider. The system should be intuitive and efficient to use. Computer literacy is assumed. The time it takes users to find the disaster they're looking for or query a specific disaster should be kept to a minimum. Error handling messages should be self-explanatory and as much as possible of the input validation should be done on the client side.

3.2.10 Integrability

The system should be able to easily address future integration requirements by providing access to its services using widely adopted public standards. All use case which are available to human users should also be accessible from external systems.

3.2.11 Deployability

Deployability is an important requirement to consider when designing a system. The system should be able to run on any of the 3 most used platforms namely Linux, Windows and Mac OS. The system must:

- run on Linux OS,
- ultimately the system should be packaged as a Docker image which is deployable on a Docker container installed on a virtual or physical Linux server.

3.3 Integration and Access Channels Requirements

This section of the document outlines the operability of the system running on the server and demonstrates user access and the integration of different technologies. A Web interface will be the single access channel provided to the user. A detailed discussion follows.

Access Channel Requirements The web interface will be accessed typically through a personal computer using a browser such as Google Chrome, Mozilla Firefox and Internet Explorer to name a few.
The system will be accessed in the following manner:

1. The user will open a web browser of their choice.
2. Click on the web page where the system will be hosted.

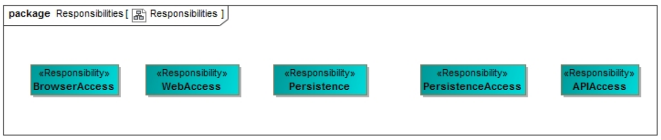
3. A login page will be displayed.

4. After user information validation, the user can then have full access to the information they wish to access, based on the privileges assigned to them during user creation.

Integration Channel Requirements Two servers would be used to achieve data persistence, one for interfacing with the website and a second for interfacing with the third-party API for data downloading. The system will also have to interface with a map database (Open Street Map) for downloading maps that will be overlaid with geospatial data. The web application will need to integrate with the services/business processes layer via a REST API.

3.4 Architectural Responsibilities

The architectural responsibilities for the system as a whole are shown in Figure below:



The system requires an environment within which the business processes realizing the services are executed. These services need to be made available to humans and systems over the web. Humans need to access the services web browsers although it may be expanded to mobile devices. The business processes require access to a persistence provider (a database).

3.5 Architectural constraints

The choice of architecture components and technologies is mostly unconstrained. The development team may choose the architecture and technologies best suited to fulfil the non-functional requirements for the system subject to:

1. the system must be deployable in a Docker container, and

2. the system must use only open source frameworks and tools.

4 Architectural Patterns

The architectural pattern that will be used for the Real-time Geospatial Data Processor and Visualiser system will be the Layering architectural pattern. For the Layering pattern, the layers that will be used are

1. Client
   (a) This layer will use the lower layers to have data delivered to it so that it can display the different functionality the user requires in a user friendy manner, which could range from queries to just simply interacting with a map to view what is going on around it.
2. Access
   (a) This layer, which is a wrapping layer of the business processes layer, makes services available in a technology-neutral way over the Internet. ExpressJS, which is a NodeJS web application server framework, will be used.
3. Business Processes
   (a) The Business Processes layer will help process requests and will also contain an API server which will be used to extract data from Third Party APIs.
4. Persistence Access
   (a) This layer will be used to access data from the persistence layer, using RabbitMQ to queue messages from the database and it will also be used to write data pulled from public web APIs to the database using Thinky.io.
5. Persistence
   (a) The persistence layer will manage persistent real-time data which will have been acquired from Third Party APIs. It would in turn supply data, via the Persistence Access Layer, to the Business Logic layer upon request and when there are updates to the requested data.

The Layering pattern is used because:

1. it allows applications to be decomposed into groups of subtasks, each group of subtasks at a certain level of abstraction,

2. the layers are pluggable and replaceable,
3. complexity is reduced,
4. the is loose high-level coupling,
5. the is ability to mock out lower level layers, and
6. the is enhanced maintainability

5 Access and Integration Channels

5.1 Integration Channels

The Real-time Geospatial Data Processor and Visualiser system will integrate with:

1. different public web APIs using HTTP requests to pull data.
2. a database
   (a) The database will be used to store data which is pulled from the APIs.
   (b) The technology that will be used for the database is RethinkDB because it is scalable and will be able to store the massive amount of data that is pulled from the APIs; it allows for data from the APIs to be stored in real-time, meaning that the data stored would be the most up-to-date at all times (adding new data and updating data that already exists in the database) and it also allows the database to continuously push updated query results to applications in real-time.
3. a messaging service
   (a) Stores any messages to and from the web client, whether it is an update to the data the web client has already received, or is currently displaying or a new message consisting of different data to the one being currently shown.
   (b) In this case the system will integrate with technology, RabbitMQ, which will act as an intermediary for any incoming or outgoing data.
4. JavaScript Library

(a) The library will be used to display dynamic maps in any web page

(b) The technology that will be used will be OpenLayers 3, as it will help display map data in most modern browsers.The OpenLayers 3 will be pulling map data from openstreetmap.

5.2 Access Channels

The Real-time Geospatial Data Processor and Visualiser system will be accessed by different users via the web interface. The web interface will be accessible through web browsers such as Google Chrome, Mozilla Firefox or any other standard web browser.
The system has one kind of users, who is:

1. Any person interested in the data

    (a) A user is able to manage, view, hide or track disasters, they are also able to query information as well as perform a point or detailed query.

6 Technologies

Technologies and frameworks that will be used to build the system are as follows:

- Front-end
    - AngularJS, provide good supports RESTFul services and easy integration with other technologies.
    - RabbitMQ listener as Message bus, to collect real time data from the backend feeder.
    - Leaflet.js, as our map framework that will use the openStreetMap as its map source.
- Back-end
    - NodeJS for server-side Web applications.
    - ExpressJS is a framework for on node.js for building web-application, supports RESTful services.
    - Bluebird for promises, to complete asynchoneous tasks.
    - RabbitMQ feeder as Message bus, to collect real time data from the database.

- Database
  - RethinkDB, supports "changefeeds", which allow you to subscribe to changes on a table. This goes hand in hand with the real-time feature of the system
  - RabbitMQ, a framework for RethinkDB; is a natural choice for distributing notifications of change events on RethinkDB.

7 References