E-AGLE Trento
Racing Team

Telemetry Exporter

Eugenio Berretta
eugenio.berretta@studenti.unitn.it

Academic year 2019/2020

Contents

1 Concept

After succeeding in making the telemetry work, we needed a quick way to safely export the gathered data. Initially we were used to manually export the data, using ssh and scp. When the number of the collections started to increase and we began loosing time to export the data during the tests, we started thinking to a better solution.

1.1 Introduction

The telemetry is a raspberry that resides in the car, plugged to the canbus. It listens to the can and gps messages and saves information on a local MongoDB. During the first tests we noticed that we were loosing a big amount of time to export the database data, that was done manually with ssh and scp. The purpose of the exporter is to provide a quick and productive way to export the data.

1.2 Evolution

The idea of a webapp as an exporter came only after some time. Before thinkind it was necessary, others solutions were thought and partially implemented.

1.2.1 Without exporter

Exporting the telemetry data manually was very long and annoying. The name of the MongoDB collections were made dinamically by the telemetry itself in order to provide useful information of their content, such as driver, type of race and timestamp. When exporting manually the collections, we needed to connect with ssh to the raspberry. After that we needed to list the collections with mongo and choose which were to be exported. Then we needed to use mongoexport for each of them, writing manually the collection name and the generated json path. Lastly we had to use scp to pass the json files from the raspberry to our pc.

1.2.2 Using bash

Initially we thought to write a simple bash script to automatize the job. But soon, before finishing it, we noticed that it would not have been enought to simplify the job. We would have stil had to connect via ssh to the telemetry.

1.2.3 Writing a command line program

After abandoning the idea of a bash script, we wrote a prototype of a command-line exporter. It was written with NodeJS (already installed in the raspberry) and allowed to export the collections as json (with mongoexport) or as csv. This solution was already quite good and quick: the user could select the collections from a list and it was run by the pc without using ssh.

1.2.4 Writing a webapp

The command line program was already good, but we thought that it would have been great to enhance the exporter providing an intuitive GUI, so that it would habe been easier to everyone to learn how to export the data. Furthermore, with the command line exporter the user had to have nodejs and mongodb-utils installed in his pc.

2 Decision

The exporter is now a webapp served by the raspberry. The memebers of the team can simply connect to the same net of the raspberry an use the exporter in any browser.

2.1 Why Vue.js?

We decided to implement the fronted using Vue.js, a javascript frontend framework. The frontend is a web frontend, because it is very easy and quick to develope and because it automatically grants support for most devices. The exporter is thought for desktop devices, we decided to avoid making it responsive because we did not need it. It is still usable by phones and with not many changes could be made responsive for all devices. Using vanilla js would have been too time-consuming and we did not need better performances. We thought that Angular was too over-engeenered for small projects like this and among us no one knew React. Hence, Vue.js was chosen as frontend framework because it is the most known among the members of the team.

2.2 Why javascript?

Initially I wanted to use Typescript, a language that transpiles to Javascript and allows type-checking. Type-checking, expecially with an editor that includes linters, can spare a lot of debugging time and makes the code more readable. Unfortunately, I was the only one that knew Typescript, so we decided to switch to Javascript because it was known by everyone. This makes the exporter more mantainable if other members of the team will have to deal with its code.

2.3 Why hosting it in the raspberry?

We could have made something that ran on the user's computer and that connected to the raspberry. The problem was that anyone should have had it in its own computer and furthermore they would have needed to install NodeJS (also mongodb-utils for export as json). We made the exporter as a normal webapp because now the user could be anyone near the telemetry with almost any device. The exporter impact on the telemetry performance is negligible, considering also the fact that it mostly used when the car is not running. The code does not change frequently and connecting via ssh to the raspberry in order to update it is not annoying.

2.4 Why NodeJS?

NodeJS is one of the most cutting-edge envinronment to develope server like this. It was already in the telemetry for other purposes, so we did not have to install it only for the exporter. The previous command-line version was already written in NodeJS and we kept big parts of that code for the backend. We used NodeJS also because a main.js file with a quite short amount of code is enough to serve efficiently a frontend. The backend has the purpose of serving the frontend and providing it the API to export the data.

2.5 Why json and csv?

The exporter allows the data to be downloaded as json or csv. These are the most flexible and useful formats for the team needs. The json is mostly used to pass the collections from the raspberry

mongodb to another pc's mongodb, this is also why the backend uses internally the mongoexport util. The csv is very useful for mathematical elaborations with MatLab or python and is used for example during the tests, when we want to check if the data that we are saving is valid. The csv data is taken with mongodb queries and parsing to csv their result.

3 Implementation

The code of the exporter can be found in this github repo: https://github.com/eagletrt/eagletrt-telemetria-exporter

3.1 Frontend

The frontend is made with Vue.js and resides in the vuejs folder.

3.1.1 Vue cli configuration

The webapp is single-page, so vue-router was not added. We used vuex (https://vuex.vuejs.org/) as store for the data that was external from the components flow. We used the default linter to a correct and pretty code.

3.1.2 External modules

A part from the Vue.js ones, other external modules were imported. For the general layout it was used bootstrap-vue (https://www.npmjs.com/package/bootstrap-vue). For the beautiful scrollbar it was used vue2-perfect-scrollbar (https://www.npmjs.com/package/vue2-perfect-scrollbar). For the http requests it was used axios (https://www.npmjs.com/package/axios).

3.1.3 Configuration

The frontend has a file /src/config.js that contains a simple configuration object that specifies the url for the API calls.

3.1.4 Assets

In the /public folder, only a favicon and a logo were added as assets. The logo is displayed on error messages or at loading time. The logo is available both in png and webp, so that it can be downloaded by the browser in a short amount of time.

3.1.5 Structure

The /src folder contains only the usual main.js and App.vue, the config.js, the store.js for vuex, that contains the biggest logic/controlling part, and two other folders. /components is the usual folder containing the vue components /services contains simple javascript files containing utils and useful functions, such as the api calls and the function to download the zip file.

3.2 Backend

The frontend is a NodeJS server

3.2.1 External modules

There are lots of external modules used by the backend. express as framework, body-parser to parse the post requests bodies, chalk for coloured logs, compression to compress the served frontend, cors for cross-origin requests, helmet for security, morgan for the requests logs, rimraf to remove the temp files and folders, zip-lib to zip folders, dree for directory-tree inspections and mongodb to query the database.

3.2.2 Created modules

Some parts of the backend code were quite articulated and considered reusable for other projects. Hence, we decided to publish three npm modules and importing them in the exporter. The first is mongo-scanner (https://www.npmjs.com/package/mongo-scanner), to retrieve the structure of a mongodb database, such as the databases and the collections that it contains. The second is mongoback (https://www.npmjs.com/package/mongoback), to an easily and fully configurable export of data using mongoexport under the hoods. In the exporter it is used to export the data as json. The third is eagletrt-csv (https://www.npmjs.com/package/eagletrt-csv), which is a sort of analogue of mongoback, but it parses data supposing it is structured as our database stucture and in the exporter it is useed to export the data as csv.

3.2.3 Structure

The /main.js file is the root of the server, where express is imported and the server starts waiting for requests. the /config.js file contains a json object used as configuration of the server. It contains the server port, the mongodb urls, the location of the frontend to serve and the logger colours. The /frontend folder contains the built vue frontend, which is served statically. The /utils folder contains js files that provides useful function used in various files. The /routes folder contains the routes of the API. It contains a routes folder with .route.js files that provide the various routes and an index.js file that, by using dree, finds all the .route.js files in the routes folder and adds it to the express router.

4 User usage

This section supposes that the telemetry database structure is already known. To see an online demo of the exporter (first version), follow this link: https://telemetria-exporter-demo.herokuapp.com/.

The user usage is this:

1. The user types in a browser http://IP:PORT, where IP is the ip of the raspberry and PORT is the port of the server.
2. The webapp asks the server for the database schema and shows it.
3. Three columns are shown. The first shows the collections. When the user select a collection, in the second column appear the sessions of that collection. The user can select or unselect sessions by clicking on them. All the selected sessions appear in the third column, organized by collection. The selected sessions can be unselected also by clicking them in the third column.
4. Once selected the sessions to export, the user clicks the JSON or the CSV button, depending on the desidered format.
5. The webapp sends the request to the server and wait for a zipped file of the exported sessions.
6. After the server answers to the webapp, the zip file named with a human-readable timestamp is downloaded. Then the webapp comes back to point 2.

5 Developer usage

5.1 Deployment on the raspberry

1. Connect the raspberry to the internet.
2. Install NodeJs if it is not installed.
3. Clone the repository.
4. Open the terminal in the root directory of the repository.
5. Execute npm i.
6. Execute npm run start.

5.2 Changing the server port or the mongodb uri

1. Open the file config.json.
2. Set the property PORT as needed.
3. Set the property of the object MONGO as needed.

5.3 Changing something in the frontend

1. Open the terminal in the directory vuejs of the repository.
2. Execute npm i.
3. Make changes to the frontend source code.
4. See changes by executing npm run serve and open a browser in http://IP:8080. Start the server if backend api are needed.
5. Open the terminal in the root directory of the repository.
6. Execute npm run build:frontend.
7. Execute npm run start to start the server.

5.4 Changing the api hostname in the frontend

1. Open the file vuejs/src/config.json of this repository.
2. Change the host and the port properties as needed.
3. Open the terminal in the root directory of this repository.
4. Execute npm run build:frontend.
5. Execute npm run start to start the server.

5.5 Automatize with SystemCtl

On the Raspberry runs Linux Ubuntu. This means that systemctl is available. To make the exporter run on boot:

1. Open a terminal.
2. Go to the /etc/systemd/system folder.
3. Create a file by executing touch exporter.service.
4. Copy this code into that file.

```
[Unit]
Description=Eagle-TRT telemetry exporter webapp
Wants=mongodb.service
After=mongodb.service
StartLimitIntervalSec=0
[Service]
Type=simple
Restart=always
RestartSec=5
User=ubuntu
ExecStart=/usr/bin/node /home/ubuntu/eagletrt-telemetria-exporter/main.js
WorkingDirectory=/home/ubuntu/eagletrt-telemetria-exporter
[Install]
WantedBy=multi-user.target
```

5. Execute sudo systemctl daemon-reload to make the changes effective.
6. Execute sudo systemctl start exporter. The exporter should start running at this step.
7. Execute sudo systemctl enable exporter to make the exporter start on boot.

6 Versions

There are two versions of the telemetry exporter. This is because after the first version, the way the mongodb database was structured changed and the exporter needed to be updated in order to work again.

6.1 What is a session

Every time there was a test or race, multiple "sessions" were saved. A session is a period of time were the car is driven by the same pilot, doing the same type of test/race, in the same circuit, stopping only for short amounts of time.

6.2 1.0

Initially, before changing structure, the mongodb was organized in different databases, such as one for each car (chimera and fenice). Each collection in a database corresponded to a session. The name of a collection was decided dinamically by the telemetry and consisted in the name of the pilot, the type of race and the time when the session started. Inside the collection there were only the data-documents of the session it corresponded. Version 1.0 of the exporter had the same ui of Version 2.0 and the only difference was that it reflected the old database structure. The first column showed the databases, the second the collections of the selected database and the third the selected collections. To code of the first version is still available on Github, in the branch 1.0 (https://github.com/eagletrt/eagletrt-telemetria-exporter/tree/1.0).

6.3 2.0

At some point the structure of the mongodb database changed and we needed to devlope version 2.0 of the exporter. The current mongodb database structure has only a fixed database "ealetrt". It contains various collections with arbitrary names. The collections have the same functions that the databases of the old structure had. They are currently four (chimera, fenice, chimera_test, fenice_test). Each collection contains two type of documents. The session-documents describe a session with pilot name, type of race, timestamp and a key. The data-documents are the same of the documents in the old structure, with the difference that now they contains a reference to the key of their session document. So, now, a collection corresponds no more to a single session, but can contain more than one. Version 2.0 has as first column the list of the collections, as second column the sessions that the selected collection contains and as third column, the selected sessions.

7 Test

7.1 Demo

The exporter was developed and tested on a local pc, using a mongodb exported with the old command line exporter. The ux changed in the very first moments of the developement. To be sure that the memebers of the team felt comfortable with the ux, we put a demo of the webapp online, using Heroku and Docker, to retrieve feedbacks in a short amount of time. The demo (https://telemetria-exporter-demo.herokuapp.com) is no more updated. It reflects the Version 1.0 of the exporter and the sample data has also session names that do not reflect the actual ones.

7.2 Current testing

Now that the exporter is quite stable and we do not need feedbacks or changing the ux, we simply test the exporter on our computers before testing it on the raspberry.

8 Conclusion

8.1 Summary

After developing succesfully the telemetry on a raspberry plugged to the canbus of the car, we bumped into an issue: exporting the gathered data. Thinking initially to trivial solutions such as a simple bash script, we came up to an articulated an better solution. We wrote initially a command line exporter, that evolved in a webapp served by the raspberry that is usable by everyone that is near to the car, with a browser as an only requirement.

8.2 Future

The project is now quite stable. The only problem is that, in order to connect to its API, the frontend needs to know the raspberry IP, that is not static. For now the IP is nonetheless almost always the same: 192.168.8.101. A minor solution has already been implemented, the user can pass url parameters to the webapp, such as http://IP:PORT?hostname=localhost:2323 or http://IP:PORT?host=localhost&p This solves the biggest problem, that makes the frontend call the right ip for the API in order to work properly. Another problem is that the user need to know the IP of the raspberry, and if it changes it can result to be annoying. A better and simple solution for the future will be configuring the raspberry to have always the same IP address.