STARTHack Asimov

Dokumentation

D. Schafer, S. Hauri, S. Ineichen, R. Schwarzentruber

2019-04-09

Inhaltsverzeichnis

1 Management Summary

Das Team ‘asimov’ bestehend aus HSLU Informatik Studenten nimmt an ihrem ersten Hackathon teil. An der Hochschule St. Gallen findet der STARThack statt, bei dem hauptsächlich an vorgegebenen Challenges das Können unter Beweis gestellt werden kann. In der gewählten Challenge von autoSense geht es darum, Daten von Auto-Unfällen, welche per JSON File zur Verfügung stehen, zu analysieren und den Schaden zu visualisieren. Über zwei REST API Requests soll zum einen ein Bild und zum anderen Daten zum Aufprall abrufbar sein. Die Challenge muss innerhalb gut 40 Stunden gemeistert werden und anschliessend einer Jury präsentiert werden. Mit viel Koffein, regelmässigen Essen, viel Coding, aber wenig schlaf gelang es dem Team ‘asimov’ die Challenge zu meistern.

2 Getting Started

2.1 Inspiration

Die Challenge von autoSense (autoSense ist eine Tochtergesellschaft der Swisscom AG) bot Teilaufgaben wie physikalische Berechnungen, Visualisierung, Präsentation und Kommunikation durch Webtechnologien, zu welchen wir bereits ein fundiertes Wissen besassen. Das war dann auch unsere Motivation, uns in dieser Challenge zu messen.

2.2 Um was geht es?

Die Lösung empfängt JSON-Dateien von Auto-Unfall-Daten über eine REST API. Als Ergebnis soll die Beschädigung, die auf das Auto eingewirkt hat, visualisiert werden. Die Visualisierung soll noch mit zusätzlichen Informationen wie G-Force, Zeitpunkt des Crashs und dem Offset in der Zeitreihe angereichert werden.

2.3 Wie wurde es umgesetzt?

Als Fundament haben wir einen Webserver (sanic) verwendet, um die API-Aufrufe zu implementieren. Im Hintergrund sind alle Funktionen und Klassen mit Python implementiert. Für die Visualisierung haben wir uns für OpenCV 4.0 entschieden. Wir haben das gesamte Projekt auch als Docker Image umgesetzt, so dass es überall und jederzeit eingesetzt werden kann.

Um dem Projekt einige Extras hinzuzufügen, haben wir uns entschieden, ein kleines, übersichtliches Web UI zu erstellen, in dem wir grundlegendes HTML5, CSS und natürlich JavaScript verwendet haben.

2.4 Herausforderungen

Die Mathematik, um den Aufprall zu berechnen, hat definitiv sehr viel Zeit in Anspruch genommen. Wir haben manchmal einige komische Werte bekommen - dass lag zum Teil daran, dass wir auf ein Transformation-Problem gestossen sind. Nach der Behebung von diesen Problemen sahen dann die Visualisierungen einheitlicher aus.

2.5 Accomplishments

Wir sind stolz darauf: * Solide Mathematik * Maintainable Code * Clean Code * Visualisierung

2.6 Was wir gelernt haben

Wir haben viel Mathematik/Physik angewendet, das meiste haben wir in der Schule einmal gelernt aber nie richtig angewendet. Der Hackathon war für als Gruppe ausserdem ein wirklich gutes Training. Wir haben gelernt, wie man besser und effizienter miteinander zusammenarbeitet, um ein grösseres Projekt wie die autoSense-Challenge anzugehen.

2.7 What’s next?

Als nächstes wollen wir unsere Lösung aufpolieren. Wir würden gerne noch ein paar Crash-Daten und ein paar Realbilder einholen, um damit zu überprüfen, ob wir die richtigen Dinge berechnet haben. Außerdem gibt es einen Azure Webspace der darauf wartet, dass unsere App veröffentlicht wird.

2.8 Built with <3 and:

- Python
- Docker
- vanillaJS
- HTML5
- CSS
- Microsoft Azure
- OpenCV
- Web Framework ‘sanic’

2.9 Github Repository

STARThack Team ‘asimov’ Github Repository

3 Tasks

3.1 Arbeitspakete

Um die Arbeiten besser auf die jeweiligen Team-Mitglieder aufzuteilen haben wir am Freitag Abend folgende Arbeitspakete definiert. Die Arbeitspakete wurden 1:1 vom Projekt-Meeting am Freitag Abend übernommen

```
* Data Parsing (transform in more structured way -> acceleration, calibration)
    * define useful functions
    * implement functions
    * crash_record.py

* Webserver
    * create webserver (sanic)
    * implement requests
    * return some dummy data for the moment
    * webserver.py (rename main.py)
    * docker container

* Image
    * define interface
    * library to draw arrows
    * library to draw circles
    * image.py

* Visualization & Math
    * jupyter notebook visualization
    * define functions to calculate angles & impact
    * start crash_record_calculator.py
```

3.2 Zuteilung der Arbeitspakete

Nach dem wir die AP (Arbeitspakete) definiert haben, konnte jeder sein präferiertes AP wählen. Die Aufteilung hat bei uns sehr gut funkioniert und wir konnten auf Anhieb jedes AP einer Person zuweisen.

4 Data Parser / Data Processing

4.1 Funktionsumfang

Mit der Klasse Data_Parser werden alle Funktionalitäten im Zusammenhang mit der Datenverarbeitung, Auswertung und Konvertierung erledigt. Das beinhaltet das Einlesen der JSON Daten, Ausfiltern der relevanten Key:Value Paare, transformieren der relativen Werte sowie die mathematischen Umrechnungen auf die geforderten Output Daten (Kraft & Winkel des Einschlags)

4.1.1 Software Abhängigkeiten

Zum auslesen der JSON Daten wurde die Python Library `json` verwendet. Zur Decodierung und Encodierung wurde das base64 Format verwendet. Für die Mathematischen Umrechnungen wurden die Libraries `math`, `pandas` sowie `numpy` verwendet.

Für die Umrechnung der relativen Zeiten konnte auf die Standard Library `datetime` zurückgegeriffen werden.

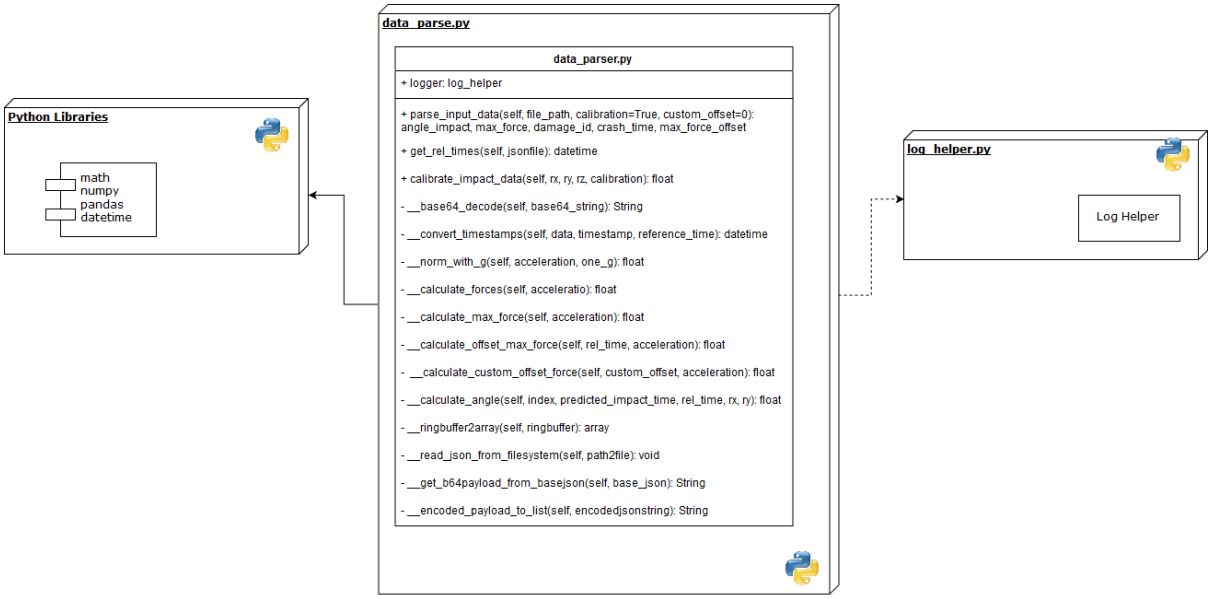
Um während dem Testing ein sauberes Logging zu erhalten, wurde auch hier unsere gemeinsame Log-Klasse `log_helper` eingebunden.

4.1.2 Prozess des Funktiondesigns

Zu Beginn werden die JSON Input-Daten eingelesen und der Payload extrahiert. Dieser ist im Base64-Format und muss daher zuerst decodiert werden; danach werden die relevanten Key:Value Paare (Beschleunigung, Timestamps) ausgelesen. Diese Daten werden danach mit einer vordefinierten Kalibration transformiert und mit einem Referenz-G Wert in G-Kräfte umgewandelt. Zum Schluss erfolgt die Umwandlung der Kräfte in einen Winkel, relativ zur virtuellen Bildmitte. Zeit, Kraft und Winkel werden danach zur Visualisierung an die Klasse `Damage_drawer` übergeben.

4.2 Klasse DataParser

Teilfunktion: Data Parser



4.3 Implementierung im Projekt

Innerhalb von dem Projekt wird die Funktion `parse_input_data` wie folgt benutzt:

Die Json Daten werden an die Klasse “data_parser” übergeben, welche danach die Rückgabewerte Kraft, Winkel und Zeit liefert. Diese Informationen werden einem drawer-Objekt übergeben, welches anhand dieser Parameter das Einschlagsbild zeichnet.

5 Damage drawer

5.1 Funktionsumfgang

Mit der Klasse DamageImage (File: `damage_image.py`) werden alle Funktionalitäten im Zusammenhang mit der Bild zusammengebündelt. Die Umfasst als Beispiel das Erkennen der Kulturen von dem Auto, das Zeichnen der Beschädigung wie auch das Beschriften der Milisekunden des Aufpralls, die Crash-ID und weiter Informationen.

5.2 Funktiondesign

5.2.1 Software Abhängigkeiten

Für die Erkennung der Kulturen wurde die Bildverarbeitungs Library OpenCV (Open Source Computer Vision Library) verwendet. Die Kulturen werden verwendet um den Eintrittspunkt der Beschädigung zu berechnen.

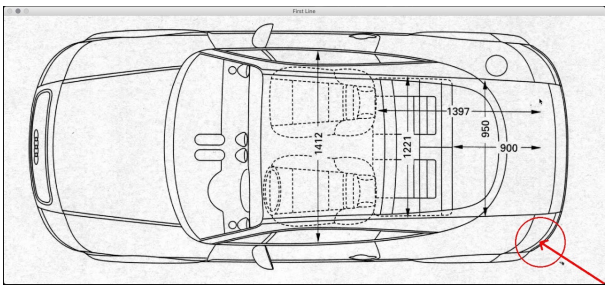
Für die Berechnung innderhalb der Klasse DamageImage wurde auf die bekannte Python Library Numpy (http://www.numpy.org) zurückgefriffen.

Die Python Standardbibliothek `os` / `math` / `shutil` werden für kleinere Funktionen benötigt.

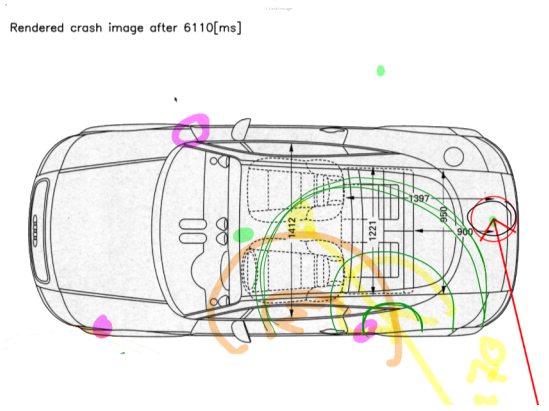
5.2.2 Prozess des Funktiondesigns

Zu Beginn wurde mittels OpenCV die Datei eingelesen und hardcoded ein Kreis und ein Pfeil gezeichnet. Mit dieser Version haben wir dann im Team das Zieldesign der Bilddatei mittels iPad und Pen gezeichnet.

Erste Version Damage Image

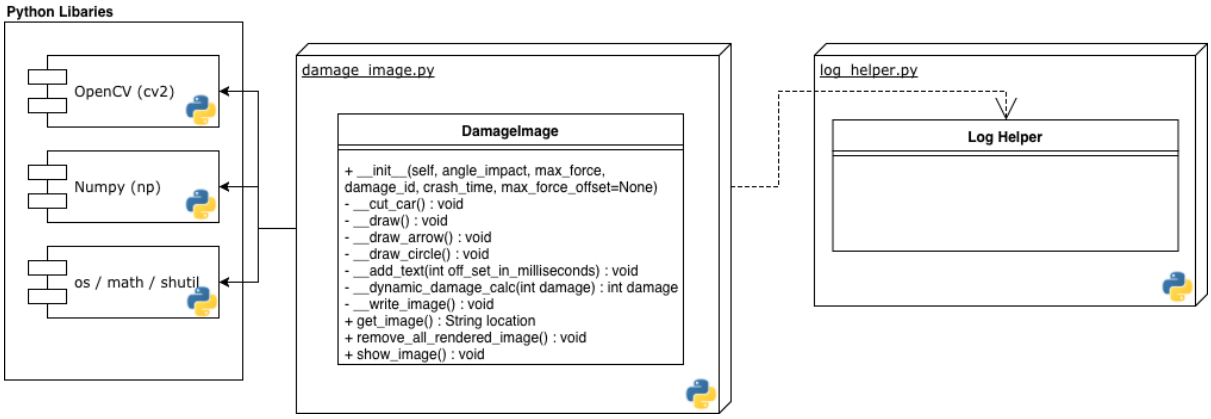


Skizze Damage drawer



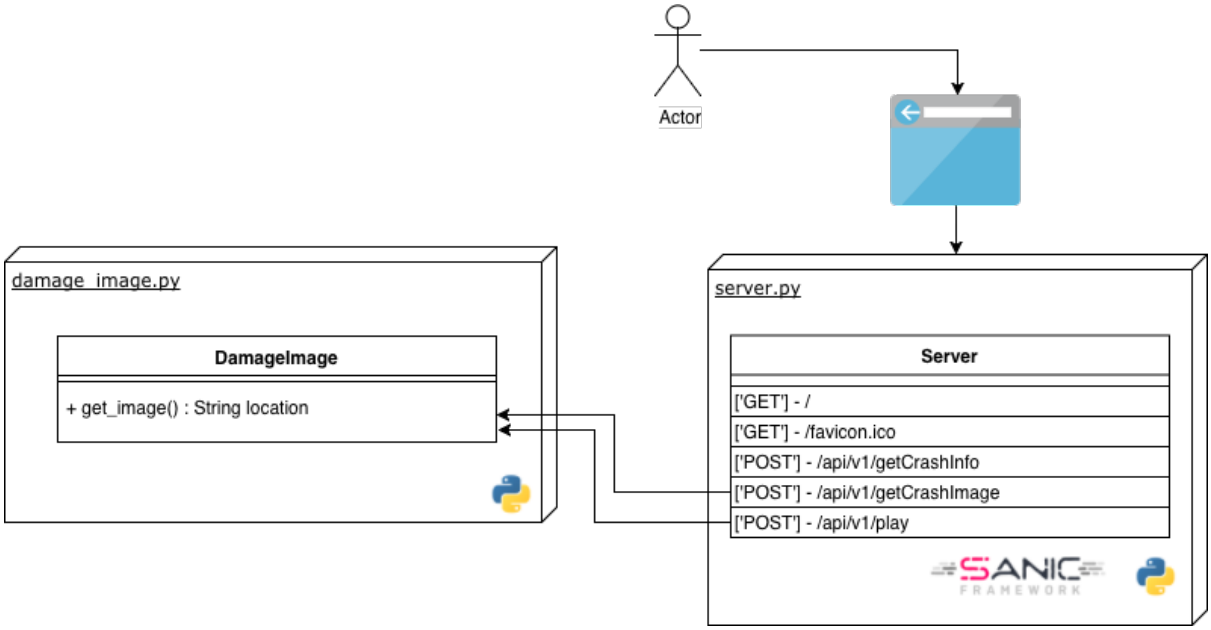
5.3 Klasse DamageImage

Teilfunktion: Damage Image



5.4 Implementierung im Projekt

Innerhalb von dem Projekt werden wir die Funktion `get_image` wie folgt benutzt:

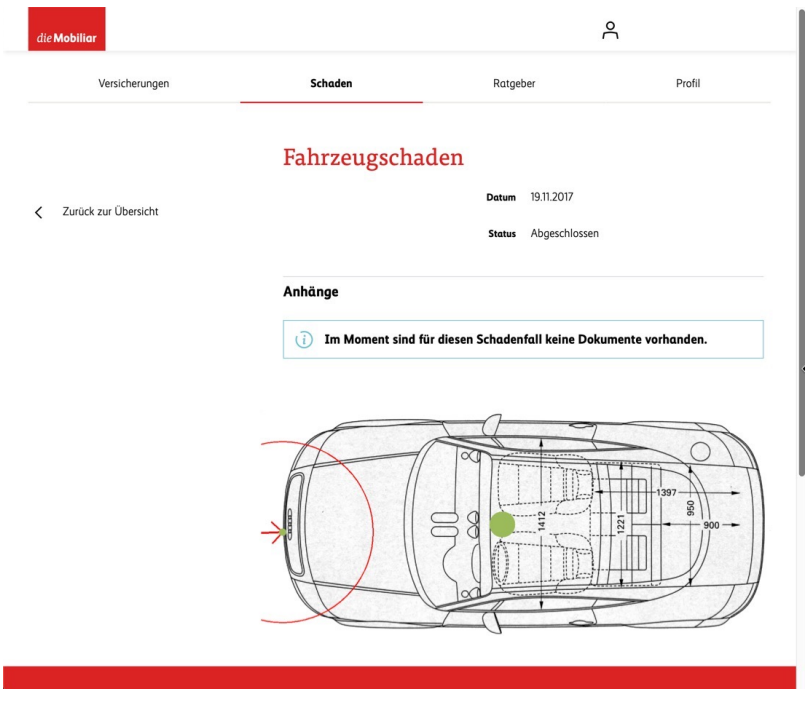


Innerhalb vom `server.py` wird ein Damage drawer Objekt erstellt und mittels der Funktion `get_image` die fertig gerenderte Bilddatei zurückgegeben und auf der Webseite dargestellt.

5.5 Mögliche Darstellung der Datei in einem Portal

Ein möglicher Einsatzbereich von unserem Projekt könnte ein Portal von einer Versicherung sein. Hier würde bei Autounfällen der Ort von dem Schaden wie auch das Ausmass der Beschädigung aufgezeigt werden. So kann ein Mehrwert in Form von mehr Informationen an dem Kunde einer Autoversicherung entstehen.

Anwendungsmöglichkeit in einem Portal



6 REST API und Frontend

Für die Challenge ist eine REST API notwendig. Diese ist genau spezifiziert und beinhaltet zwei vorgeschriebene Requests. Wir haben im Verlauf vom Projekt noch ein UI gemacht, um die eigentlichen Requests zu testen und dem User eine einfache Verwendung zu gewährleisten. Da in der Challenge auch noch erwähnt ist, dass das Hosting auch berücksichtig werden sollte, haben wir für diese Teilaufgabe Docker genutzt. Die Applikation kann so überall laufen gelassen werden, wo Docker installiert ist. Heutzutage ist das eine gängige Art und Weise etwas auf einem Server laufen zu lassen und bietet dazu auch die Möglichkeit eine Applikation zu skalieren

6.1 REST API Requests

6.1.1 Crash Info

Die ‘Crash Info’ API Request ist dazu da den ‘impactAngle’ (Winkel des Einschlags beim Crash) und den ‘offsetMaximumForce’ (die Maximalkraft die eingewirkt hat) zurückzugeben. Die Daten werden als JSON verpackt und zurückgegeben.

Der Python Code dazu sieht folgendermaßen aus:

```
# POST request 1 - returns JSON:
# {"impactAngle": degrees, "offsetMaximumForce": millisecond}
@app.route('/api/v1/getCrashInfo', methods=['POST',])
async def crash_info(request):
    ''' crash info parses the crash record and returns a JSON object '''
    log.info("Handling '/api/v1/getCrashInfo'")

    angle, max_force_offset, _, _, _ =
    DataParser().parse_input_data(request.body.decode('utf8'))

    return json({'impactAngle': angle,
                 'offsetMaximumForce': max_force_offset})
```

Es ist eine Asynchrone Methode welche als ‘POST’ Request markiert ist und die ‘api/v1/getCrashInfo’ Route nutzt. Die Annotationen werden von dem ‘sanic’ Framework bereitgestellt. Ein Log Eintrag hilft beim analysieren. Die Hauptarbeit wird aber in dem ‘DatenParser’ gemacht welche alle relevanten Daten zurückgibt. Die Daten (autoSense JSON) für den ‘DataParser’ werden mithilfe der Request übergeben. Die letzte Zeile baut ein JSON Objekt und gibt somit die Antwort der Request an den Sender zurück.

6.1.2 Crash Image

Die Crash Image Methode gibt ein Bild zurück welches den Einschlag und die Maximale Kraft des Umfalls illustriert.

Der Python Code dazu sieht folgendermassen aus:

```
# POST request 2 - returns a rendered crash image (PNG)
@app.route('/api/v1/getCrashImage', methods=['POST',])
async def crash_image(request):
    ''' crash image parses the crash record and returns a Image '''
    log.info("Handling '/api/v1/getCrashImage'")
```

```
customOffset = 0
try:
    customOffset = int(request.args.get('timeOffsetMS'))
except Exception as e:
    log.error(e)

log.info("Set customOffset: " + str(customOffset) + "ms")

angle_impact, max_force, damage_id, crash_time, max_force_offset =
DataParser().parse_input_data(
                    request.body.decode('utf8'),
                    custom_offset=customOffset)

d = DamageImage(angle_impact, max_force, damage_id,
                crash_time, max_force_offset)
return await file(d.get_image())
```

Die Route der Request ist '/api/v1/getCrashImage'. Ein Offset zum Zeitpunkt des Aufpralls kann übergebene werden ('timeOffsetMS'). Zusätzlich muss wieder das JSON vom autoSense Sensor übergeben werden. Der 'DatenParser' übernimmt wieder die Hauptaufgabe von dieser Request. Zusätzlich wir die 'DamageImage' Klasse zum generieren des Bildes verwendet. Anschliessend wird das generierte Bild zurückgegeben

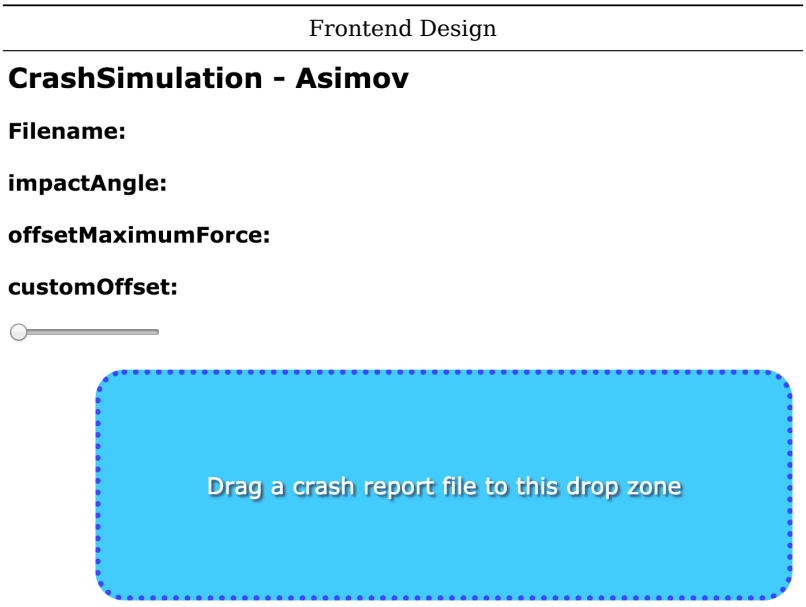
6.1.3 Play

Zusätzlich zur gegebenen Aufgabenstellung haben wir noch eine Request eingebaut, welche mehrere Bilder zurückgegeben um den Unfall genauer zu inspizieren. Es wird quasi das selbe gemacht wie bei der 'Crash Image' Request. Nur wird eine Liste von Bilder zurückgegeben, welche dann im Browser dargestellt werden können. Diese Methode ist nicht optimal da alle Bilder zuerst berechnet werden müssen und nicht gestreamt wird.

6.2 Frontend

Das Frontend ist sehr simple Aufgebaut:

Frontend Design



Das wichtigste ist die Drag & Drop Zone um ein JSON File von autoSense hochzuladen. Sobald man ein

valides JSON hochgeladen hat werden im Hintergrund die beiden API Requests an das Backend gemacht. Anschliessend wird das Bild und die Daten (Filename, impactAngle & offsetMaximumForce) angezeigt. Optional kann noch der ‘customOffset’ eingestellt werden nicht die MaximalKraft sondern einen anderen Zeitpunkt des Aufpralls darzustellen.

Frontend mit hochgeladenem JSON:

Frontend mit geladenem JSON

CrashSimulation - Asimov

Filename:

4.json

impactAngle:

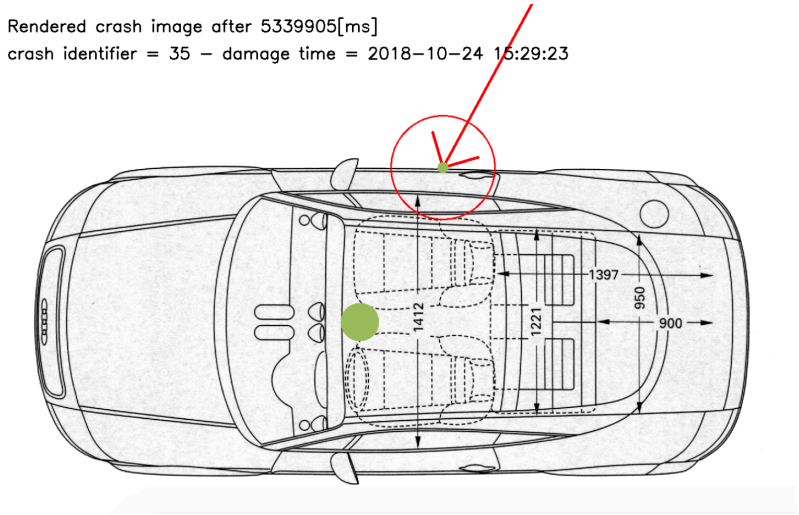
298.3973630356

offsetMaximumForce:

4.7887611696

customOffset:

Rendered crash image after 5339905[ms]
crash identifier = 35 – damage time = 2018–10–24 15:29:23



6.3 Docker

Docker ist ein Tool um Images zu kreieren, welche dann in einem Container ausgeführt werden können. Somit kann eine Applikation unabhängig vom Betriebssystem ausgeführt werden. Ausserdem wäre es möglich eine Applikation zu skalieren indem mehrere Container auf verschiedenen Systemen ausgeführt werden und ein Proxy dazwischen geschaltet wird.

Unser Dockerfile, welches das Image beschreibt, sieht folgendermassen aus:

```
FROM python:3.7-slim

#Install libs and tools needed for building python wheels
RUN apt-get update
RUN yes | apt-get install build-essential
RUN yes | apt-get install cmake git libgtk2.0-dev \
          pkg-config libavcodec-dev libavformat-dev libswscale-dev
RUN yes | apt-get install python-dev python-numpy libtbb2 libtbb-dev \
          libjpeg-dev libpng-dev
```

```
#Install python dependencies
COPY requirements.txt /app/
RUN cd /app && pip install -r requirements.txt

#Copy application to /app
COPY data/* /app/data/
COPY frontend/* /app/frontend/
COPY helper/* /app/helper/
COPY images/* /app/images/
COPY *.py /app/

#Change working directory to /app
WORKDIR /app

#Run server
ENTRYPOINT [ "python", "server.py" ]
```

Unser Basis Image ist ein Python3.7 Image von DockerHub (einer öffentlichen Registry bei der Images hochgeladen werden). Wir installieren zuerst alle Abhängigkeiten damit die zusätzlichen Python Libraries (wie zum Beispiel numpy) installiert werden können. Anschliessend werden die Python Abhängigkeiten installiert bevor schlussendlich alle Files der Applikation in das Image kopiert werden. Zum Schluss wird die Working Directory (WORKDIR) und der Entrypoint (der Webserver) spezifiziert.