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 am ersten Hackathon teil. An der Hochschule St. Gallen findet der STARThack teil, bei dem Hauptsächlich an vorgegebenen Challenges das Können unter Beweis gestellt werden kann. In der gewählten Challenge von autoSense geht es darum Unfalldaten welche per JSON File zur verfügen Stehen zu analysieren und den Schaden zu visualisieren. Über zwei REST API Requests soll zum einen ein Bild und zum zweiten Daten zum Aufprall abrufbar gemacht werden. Die Challenge muss innerhalb gut 40h 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) hatte wir bereits ein breites Wissen von Teilaufgaben, wie Physik, Visualisierung, Präsentation und Kommunikation durch Webtechnologien. Das war dann auch unsere Motivation, uns in dieser Challenge zu messen.

2.2 Um was geht es?

Die Lösung empfängt JSON-Dateien 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 grundlegende 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 Arbeitpakete wurde 1:1 von dem 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 zu einer Person zuweisen.

4 Data Parser / Data Processing

4.1 Funktionsumfgang

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

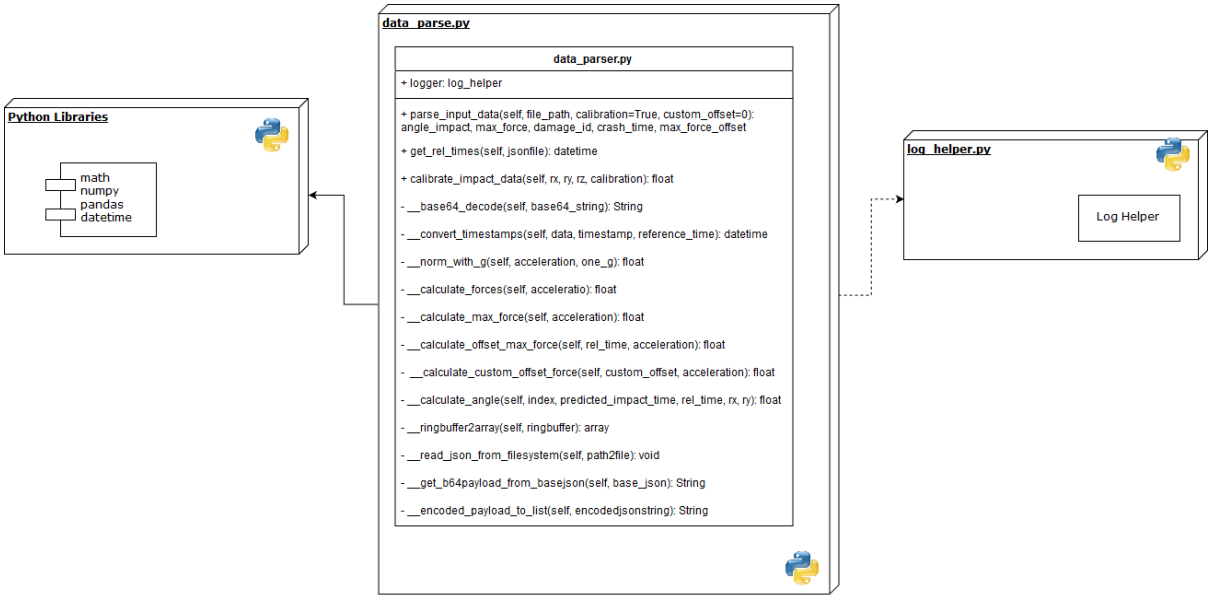
Zu Beginn werden die JSON Input Daten eingelesen und der Payload extrahiert. dieser wird mit base64 decodiert und danach die relevanten Key:Value Paare (Beuschlunigung x,y,z) ausgelesen. Diese Daten werden danach mit einer vordefinierten Kalibration transformiert und mit einem Referenz-G Wert in G-Kräfte 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 Realisation / Umsetzung

Klassendiagramm und Beschreibung der Funktionen der einzelnen Methoden.

4.3 Klasse DamageImage

Teilfunktion: Damage Image



4.3.1 Methode: parse_input_data

```
def parse_input_data(self, file_path, calibration=True, custom_offset=0):
```

Die Methode `parse_input_data` wird von extern verwendet und führt alle Teilfunktionen zusammen. Das JSON File wird entweder als Objekt oder als Filepath übergeben und danach verarbeitet. Zusätzlich besteht die Möglichkeit die Berechnungen ohne Kalibration auszuführen (nur für Testzwecke nützlich). Ausserdem kann via `custom_offset` ein beliebiger Offset in Millisekunden angegeben werden, standardmässig werd der Zeitpunkt der grössten Krafteinwirkung selbst berechnet.

Als Rückgabeparameter liefert die Methode: `angle_impact`, `max_force`, `damage_id`, `crash_time` und `max_force_offset`.

4.3.2 Methode: get_rel_times

```
def get_rel_times(self, jsonfile):
```

Die Methode `get_rel_times` wandelt die Relativen Zeiten Anhand des Referenzwertes zu realen Zeiten um. Diese Information wird benötigt um den genauen Zeitpunkt zurückzugeben, an dem am meisten Kräfte auf das Auto wirkten.

4.3.3 Methode: __base64_decode

```
def __base64_decode(self, base64_string):
```

Die Methode `__base64_decode` wandelt einen base64 Payload in einen UTF-8 json Paylod um.

4.3.4 Methode: __convert_timestamps

```
def __convert_timestamps(self, data, timestamp, reference_time):
```

Die Methode `__convert_timestamps` wandelt die relativen Zeiten der einzelnen Beschleunigungsmessungen in reale Zeiten um.

4.3.5 Methode: calibrate_impact_data

```
def calibrate_impact_data(self, rx, ry, rz, calibration):
```

Die Methode `calibrate_impact_data` wandelt die Arrays der x,y und z Beschleunigungen mit der von Autosense vorgegebenen Kalibration um, somit erhält man die Beschleunigungsvektoren relativ zur Auto Mitte.

4.3.6 Methode: __norm_with_g

```
def __norm_with_g(self, acceleration, one_g):
```

Die Methode `__norm_with_g` berechnet anhand der vorgegebenen G-Kraft die normierte G-Kraft des Beschleunigungsvektors.

4.3.7 Methode: __calculate_forces

```
def __calculate_forces(self, acceleration):
```

Die Methode `__calculate_forces` berechnet mittels Wurzelrechnung die effektiven Kräfte.

4.3.8 Methode: __calculate_max_force

```
def __calculate_max_force(self, acceleration):
```

Die Methode `__calculate_max_force` berechnet, wann die grösste Kraft in der Messung auftrat und gibt diese Kraft zurück.

4.3.9 Methode: __calculate_offset_max_force

```
def __calculate_offset_max_force(self, rel_time, acceleration):
```

Die Methode `__calculate_offset_max_force` berechnet, wann die grösste Kraft in der Messung auftrat und gibt den Zeitpunkt als Offset zurück.

4.3.10 Methode: __calculate_custom_offset_force

```
def __calculate_custom_offset_force(self, custom_offset, acceleration):
```

Die Methode `__calculate_custom_offset_force` berechnet, wann die Kraft zu einem beliebigen Offset in der Messung und gibt diese Kraft zurück.

4.3.11 Methode: __calculate_angle

```
def __calculate_angle(self, index,
                      predicted_impact_time, rel_time, rx, ry):
```

Die Methode `__calculate_angle` berechnet mittels euklidischer Distanz den Einschlags-Winkel anhand der Kräfte Vektoren.

4.3.12 Methode: __ringbuffer2array

```
def __ringbuffer2array(self, ringbuffer):
```

Da die Daten im JSON als Ringbuffer gespeichert werden, sortier die Methode `__ringbuffer2array` die Werte zuerst chronologisch.

4.3.13 Methode: __read_json_from_filesystem

```
def __read_json_from_filesystem(self, path2file):
```

Die Methode `__read_json_from_filesystem` liest die Input JSON Daten entweder vom Filesystem oder direkt aus einem String.

4.3.14 Methode: __get_b64payload_from_basejson

```
def __get_b64payload_from_basejson(self, base_json):
```

Die Methode `__get_b64payload_from_basejson` extrahiert den Base64 Payload aus dem Input JSON.

4.3.15 Methode: __encoded_payload_to_list

```
def __encoded_payload_to_list(self, encodedjsonstring):
```

Die Methode `__encoded_payload_to_list` konvertiert den encodierten JSON String in eine Python List zur besseren weiterverarbeitung.

4.4 Implementierung im Projekt

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

Die Json Daten werden an die Klasse “data_parse” übergeben, welche danach die Rückgabewerte Kraft, Winkel und Zeit liefert. Diese Informationen werden einem drawer-Objekt übergeben, welches anhand dieser Parameter das EInschlagsbild zeichnet (siehe Kapitel 4.5 & 4.6)

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.

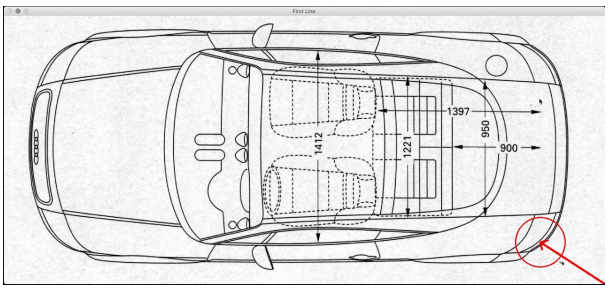
Für die Berechnung innderhalb der Klasse DamageImage wurde auf die bekannte Python Library Numpy (http://www.numpy.org) zurückgefriffen.

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

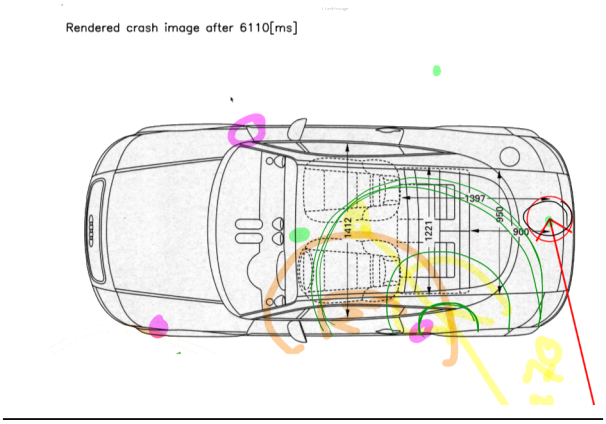
5.2.2 Prozess des Funktiondesigns

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

Erste Version Damage Image



Skizze Damage drawer

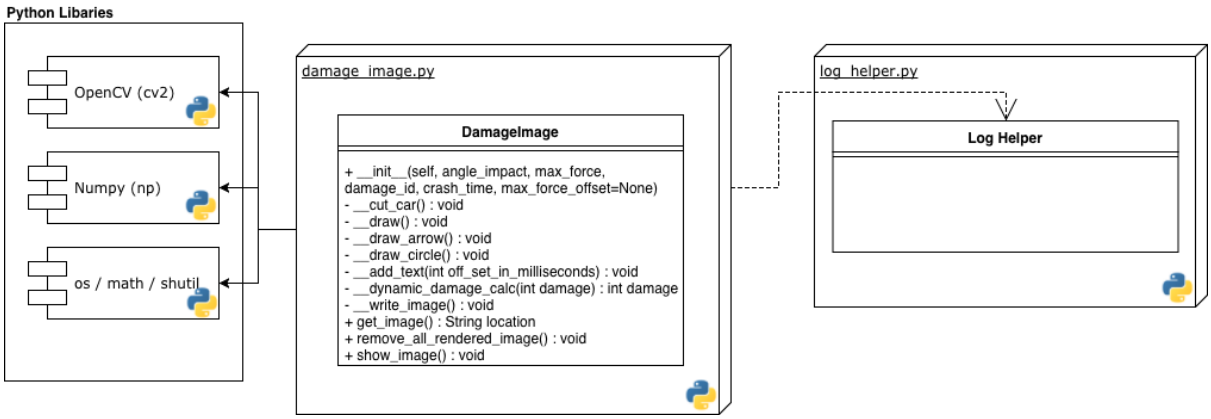


5.3 Realisation / Umsetzung

Klassendiagramm und Beschreibung der Funktionen der einzelnen Methoden.

5.4 Klasse DamageImage

Teilfunktion: Damage Image



5.4.1 Methode: Init

```
def __init__(self, angle_impact, max_force, damage_id,
             crash_time, max_force_offset=None):
```

Die Init Methode wird aufgerufen bei der Instanzierung von einem Objekt. Als Parameter werden folgende Informationen benötigt: `self`, `angle_impact`, `max_force`, `damage_id`, `crash_time` und `max_force_offset`.

Mit den Methoden Parameter werden die jeweiligen lokalen / privaten Methodenvariabeln initialisiert und den Wert zugewiesen. Weiter wir de Pfad wo die Bilddatei gespeichert wird aufgrund der `damage_id` und der allfälligen `max_force_offset`. Die Dateine werden in einem speziell definierten Ort (`images_rendered/`) gespeichert. Dies hat den Grund, dass wenn die Bilddatei von den gleichen Crash-Report und dem gleichen `max_force_offset` nicht nochmals neu erstellen und schreiben muss, sondern direkt zurückgeben kann.

5.4.2 Methode: Kulturen Auto

```
def __cut_car(self):
```

Die Methode `__cut_car` findet mittels OpenCV die Kulturen (Randkulturen) von dem Auto-Bild. Diese Kulturen werden als Pixel-Matrix abgespeichert und später für den Eintrittspunkt der Beschädigung benötigt. Dazu wird das Auto-Bild in Schwarz/Weiss konvertiert und den Threshold für die Linien gesetzt. Mittels der OpenCV Funktion `cv2.findContours` können die äussersten, geschlossenne Linien abgefragt werden.

5.4.3 Methode: Zeichnen

```
def __draw(self):
```

Die generelle Zeichnungsmethode `__draw` ist als Wrapper Methode zu verstehen. Wenn gegebenenfalls noch weitere Beschriftungen auf die Bilddatei geschrieben werden soll, kann dies hier eingefügt werden. Hier passiert auch die Entscheidung ob der Text für den Offset der Millisekunden angezeigt wird oder nicht.

5.4.4 Methode: Zeichnen - Pfeil

```
def __draw_arrow(self):
```

Hier wird der Pfeil für die Bilddatei gezeichnet. Etwas unschön ist hier, dass auch noch der Nullpunkt des Koordinatensystem in dieser Methode gezeichnet wird. Eine entsprechendes `# TODO:` ist hier vermerkt.

5.4.5 Methode: Zeichnen - Kreis

```
def __draw_circle(self):
```

Der Kreis für die für die Grösse der Beschädigung wird hier auf die Bilddatei gezeichnet. Die Grösse von dem Kreis (Radius) wird mit Hilfe der Funktion `__dynamic_damage_calc` berechnet.

5.4.6 Methode: Text auf das Bild

```
def __add_text(self, off_set_in_milliseconds):
```

Auf der resultierende Bilddatei werden folgende Informationen dargestellt:

- Time-Offset von der maximalen Beschädigung
  - Text: `Rendered crash image after " off_set_in_milliseconds + "[ms]"`
- Eindeutige Bilddatei Beschriftung mit der entsprechender Uhrzeit aus dem Crashreport
  - Text: `crash identifier = " + self.damage_id + " - damage time = " crash_time`

5.4.7 Methode: Berechnung Beschädigung

```
def __dynamic_damage_calc(self, damage):
```

Diese Hilfsmethode berechnet aufgrund einem numerischen Wert die grösse der Beschädigung. Die maximale Beschädigung von 15 und die minimale Beschädigung von 2 wurden aus den Daten ermittelt.

5.4.8 Methode: Schreiben der Bilddatei

```
def __write_image(self):
```

Hier wird die PNG-Bilddatei auf den lokalen Storage von dem Server heruntergeschrieben.

5.4.9 Methode: Pfad der geschriebene Datei

```
def get_image(self):
```

Diese Public Methode wird von Server verwendet umd die fertige Bilddatei zu erhlaten. Innerhalb dieser Methode werden die Zeichnungsmethoden `self.__draw()` und die `self.__write_image()` ausgeführt.

Als Rückgabewert erhält der Server den Pfad der Datei, welche auf dem Server geschrieben wurde.

5.4.10 Methode: Löschen von allen gerendert Dateien

```
def remove_all_rendered_image(self):
```

Diese Housekeeping Methode dient dazu, alle gerenderten Bilddateien auf dem Server zu löschen. Die Methode kann vor dem Start des Server wie auch nach dem Testing ausgeführt werden.

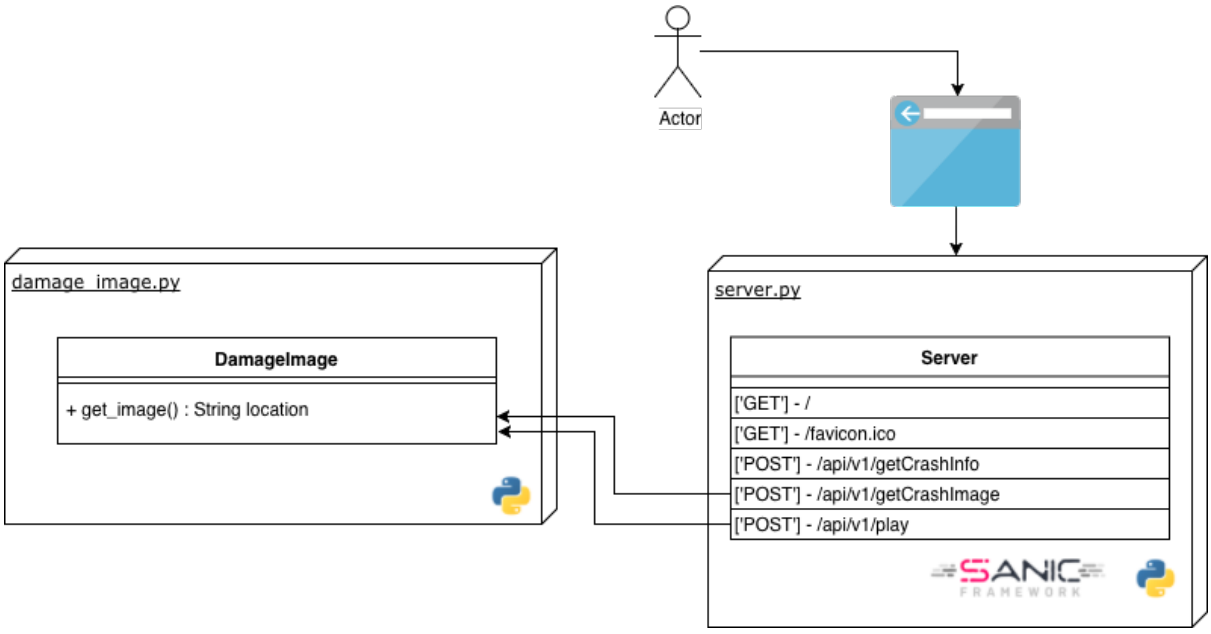
5.4.11 Methode: Anzeige der Datei auf dem Bildschirm

```
def show_image(self):
```

Die `show_image` dient für eine rasche Entwicklung ohne Server. Sie führt die gleichen Methoden aus, wie die Methode `get_image` mit dem Unterschied, dass die Bilddatei nicht an den Server zurückgegeben wird sondern mittels OpenCV an dem Display angezeigt wird. So kann Abhängigkeit vom Server neue Funktionen rasche getestet werden.

5.5 Implementierung im Projekt

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



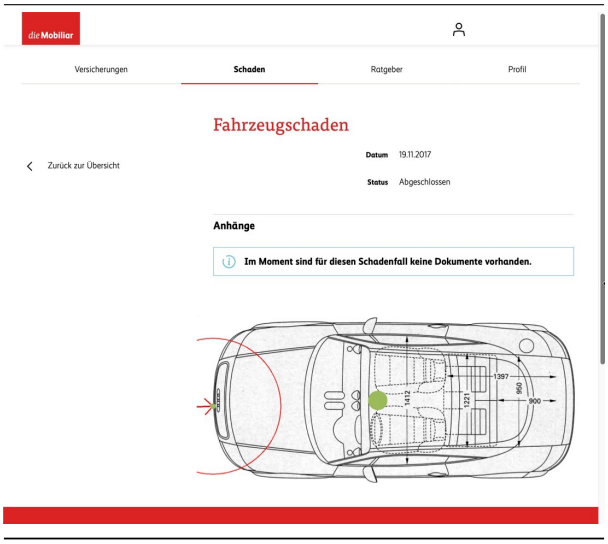
Innerhalb vom `server.py` wird ein Damage drawer Objekt erstellt und mittels der Funktion `get_image` die fertig gerenderte Bilddatei zurückgegeben und auf der Webseite dargestellt.

5.6 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.

Anzeige 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

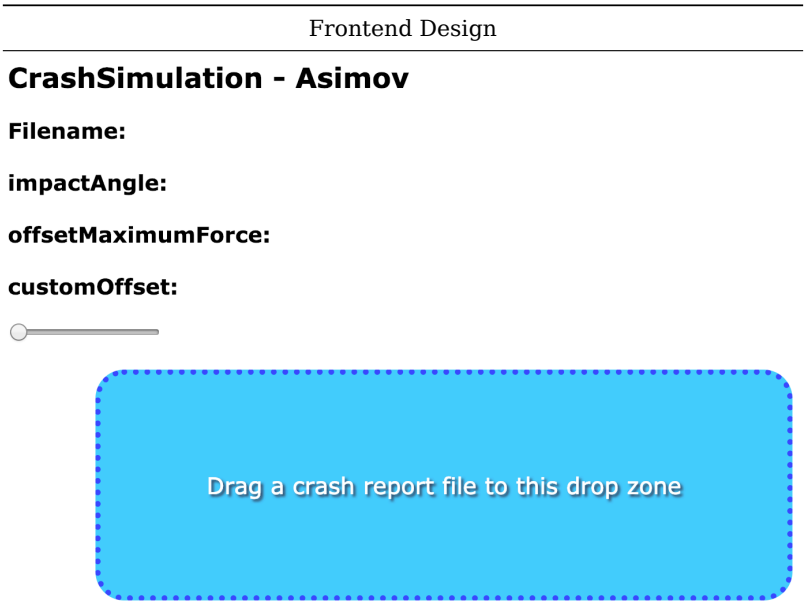
6.1.3 Play

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 va-

lides 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:

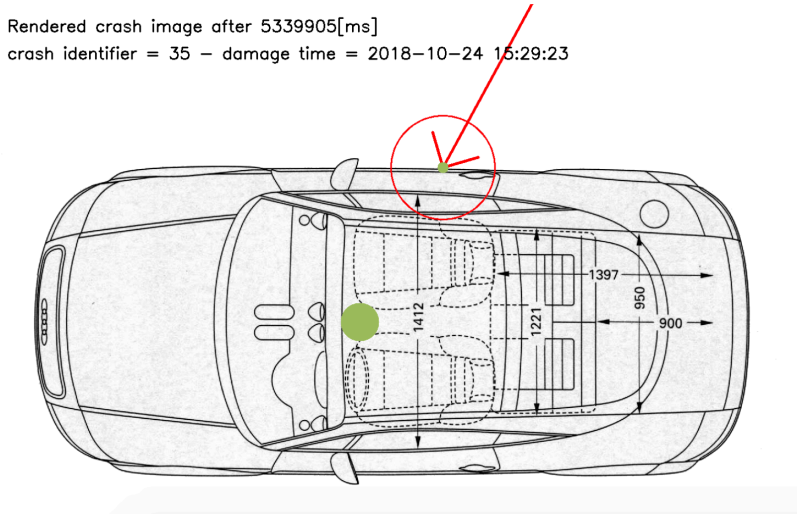
298.3973630356

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.