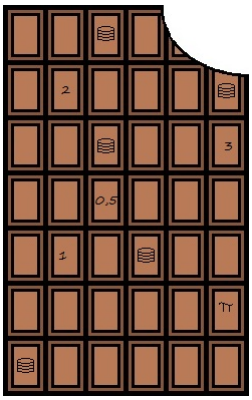
iFeeD

Interaktives Feedbacksystem zur Datenanalyse

Felix Bening, Erik Borker, Joshua Brutscher, Nico Denner, Robert Krause

15-10-2021



Inhaltsverzeichnis

1 Änderungen Pflichtenheft **5**

2 Architektur **7**

3 Backend **8**

3.1 Einleitung . . . 8

3.2 Paketdiagramm . . . 10

3.2.1 wsgi.py . . . 10

3.2.2 settings.py . . . 11

3.2.3 url.py . . . 11

3.2.4 manage.py . . . 11

3.3 Models . . . 12

3.3.1 Dataset . . . 13

3.3.2 Setup . . . 14

3.3.3 Session . . . 16

3.3.4 Person . . . 17

3.3.5 Admin . . . 18

3.3.6 User . . . 19

3.3.7 JSONDataset . . . 20

3.3.8 FeedbackModes . . . 21

3.3.9 HistoryModes . . . 22

3.3.10 QueryStrategies . . . 23

3.3.11 Labels . . . 25

3.4 Serializer . . . 26

3.4.1 ModelSerializer . . . 26

3.4.2 DatasetSerializer . . . 27

3.4.3 SetupSerializer . . . 28

3.4.4 SessionSerializer . . . 29
3.4.5 PersonSerializer . . . 30
3.4.6 AdminSerializer . . . 31
3.4.7 UserSerializer . . . 32
3.5 Views . . . 33
3.5.1 ListDataset . . . 34
3.5.2 ListSetup . . . 35
3.5.3 ListSession . . . 36
3.5.4 ListPerson . . . 37
3.5.5 ListAdmin . . . 38
3.5.6 ListUser . . . 39
3.6 URL . . . 40
3.7 JSON classes . . . 41
3.7.1 DatasetJSON . . . 41
3.7.2 SetupJSON . . . 42
3.7.3 SessionJSON . . . 44
3.7.4 PersonJSON . . . 45
3.7.5 AdminJSON . . . 46
3.7.6 UserJSON . . . 47

4 Frontend **48**
4.1 Einleitung . . . 48
4.2 Model . . . 49
4.2.1 SessionDetailComponent . . . 49
4.2.2 LoginComponent . . . 50
4.2.3 MenuComponent . . . 52
4.2.4 UserIterationComponent . . . 53
4.2.5 Label . . . 55
4.2.6 UserMainOverviewComponent . . . 56
4.2.7 AdminMainOverviewComponent . . . 57

4.2.8 AdminDatasetOverviewComponent . . . 58
4.2.9 AdminSessionOverviewComponent . . . 60
4.2.10 AdminSetupOverviewComponent . . . 61
4.2.11 AdminSetupCreateComponent . . . 63
4.2.12 AdminSetupDetailComponent . . . 64
4.2.13 AdminPersonCreateComponent . . . 66
4.2.14 AdminUserManagementComponent . . . 67
4.2.15 Ladezeiten . . . 69
4.3 View . . . 70
4.4 Controller . . . 71
4.4.1 Routing . . . 71
4.4.2 RESTService . . . 75
4.4.3 PersonService . . . 77
4.4.4 AdminGuard . . . 79
4.4.5 UserGuard . . . 80
4.4.6 JSONHandlerService . . . 81
4.4.7 PlotService . . . 82
4.4.8 StatisticsService . . . 83
4.4.9 CalcService . . . 84
4.4.10 SessionStatus . . . 85
4.4.11 ExportService . . . 86

5 Interaktionsabläufe 88

1 Änderungen Pflichtenheft

Die Änderungen im Pflichtenheft beziehen sich auf die Rückmeldung des letzten Kolloquiums.

- M2: Der User hat die Möglichkeit in einer Iteration verschiedene Heatmaps zu betrachten.
- M7: Die Oberflächenelemente in einer Session können im Setup deaktiviert werden.
- M8: Das System wird so modelliert, dass die Möglichkeit besteht den Code so zu erweitern, dass auch mehrere Punkte pro Iteration gewählt werden können.
- M9: Es besteht die Möglichkeit Ergebnisse eines Setups zu exportieren, hiermit ist nicht die Setupkonfiguration gemeint.
- K6: Ein Verweis zu F13 sowie zum GUI-Entwurf (Abb. 6) wurde hinzugefügt.
- K7: Ein Verweis auf F14 wurde hinzugefügt.
- K9: Der Admin kann die Ergebnisse zweier Sessions vergleichen.
- K10: Userkonten werden von einem Admin erstellt, es bedarf keiner Einladung.
- Produkteinsatz: Querystrategien sind unter Umständen besser für die Auswahl von Objekten geeignet als die Wahl durch den User.
- Produktdaten: Die finale Klassifikation entspricht der Klassifikation durch die

OcalAPI mit Hilfe der Labelung des Users in der Session.

- F3: Es wurde ein Verweis auf das Kapitel Produktdaten hinzugefügt.

- F4: Beim Löschen eines Datensatzes werden nicht beendete Sessions im System vermerkt.

- F5: Ein Setup kann per Button zu einem oder mehreren Usern zugewiesen werden (das heißt es werden gleichzeitig mehrere Sessions erstellt).

- F6.6: Subspaces werden zur Visualisierung verwendet.

- F6.9 Wenn der User in der definierten Zeit keine Labelung abgibt wird ein Objekt vom System ausgewählt und mit der nächsten Iteration fortgefahren.

- F12: Anzahl der Bearbeitungen wurde durch Anzahl der Revisionen ersetzt.

- F14: Kappawert wird zusätzlich angezeigt.

- NF: Die minimale Bildschirmgröße beträgt 10 Zoll.

- T3.3/F5/M4: Der Admin beendet explizit die Bearbeitung von Setups.

2 Architektur

Das Projekt ist aufgeteilt in Frontend und Backend, das Frontend ist dabei nach der MVC-Architektur strukturiert.Die Partitionierung in Modell, View und Controller gewährleistet Flexibilität und Portabilität im Entwurf, sodass zum Beispiel die Darstellung überarbeitet werden kann, ohne die Speicherung der Daten zu beeinträchtigen.

3 Backend

3.1 Einleitung

Die Hauptfunktionalität des Backends ist das Halten und Verwalten von Daten und die Kommunikation mit der OcalAPI. Als Framework dient Django welches in Python implementiert ist. Die Hauptunterstützung bietet Django durch die automatisierte Verwaltung der Datenbank und dem REST-Framework.

Datenbankunterstützung: Durch das Anlegen von Klassen in der Datei models.py und späterem Migrieren werden Tabellen in einer Datenbank angelegt. Die Klassen in models.py erben von models.Model. Diese Klasse bietet Methoden um neue Tupel in die Datenbank zu speichern beziehungsweise aus ihr auszulesen.

REST-Client-Framework: Desweiteren hilft das REST-Framework dabei Daten aus der Datenbank direkt als JSON auszuliefern. Hierfür benötigt man Serializer-Klassen welche von rest_framework.serializers.ModelSerializer erben. Hierin wird in einem Array definiert, welche Attribute das JSON-Objekt haben soll. Die gewählten Attribute müssen sich mit den Attributen, welche in der models.py definiert wurden, decken.
Als nächste Instanz wird die view.py entworfen. Die in dieser Datei entworfenen Klassen erben von der Klasse rest_framework.generics.ListAPIView. Diese Klasse bietet die Methode as_view(), welche das JSON-Objekt erstellt. Die Methode wird in der Datei url.py genau dann aufgerufen, wenn der Server mit einer passenden Url angesprochen wird.

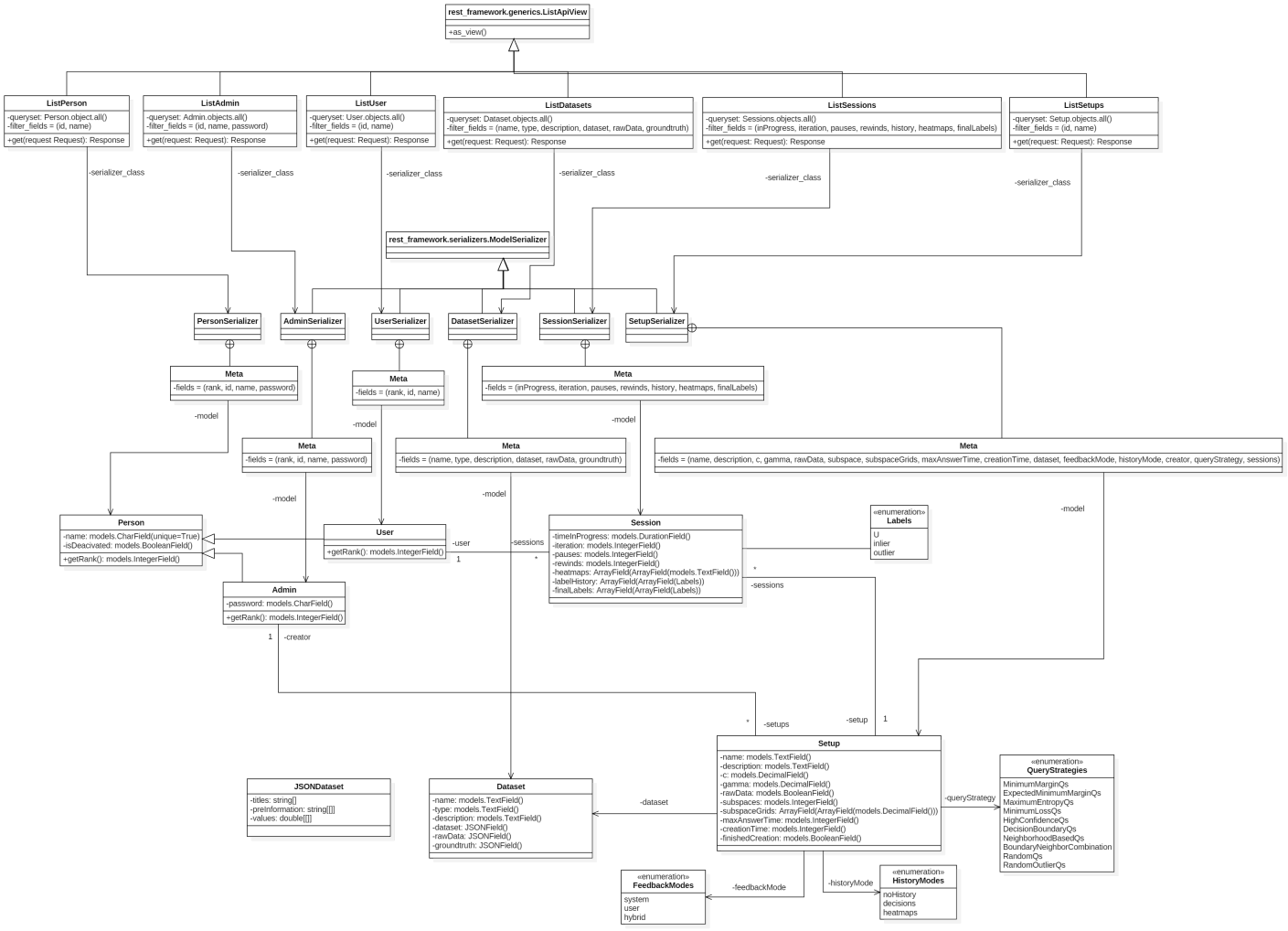


Abbildung 1: Übersicht aller Klassen im Backend

3.2 Paketdiagramm

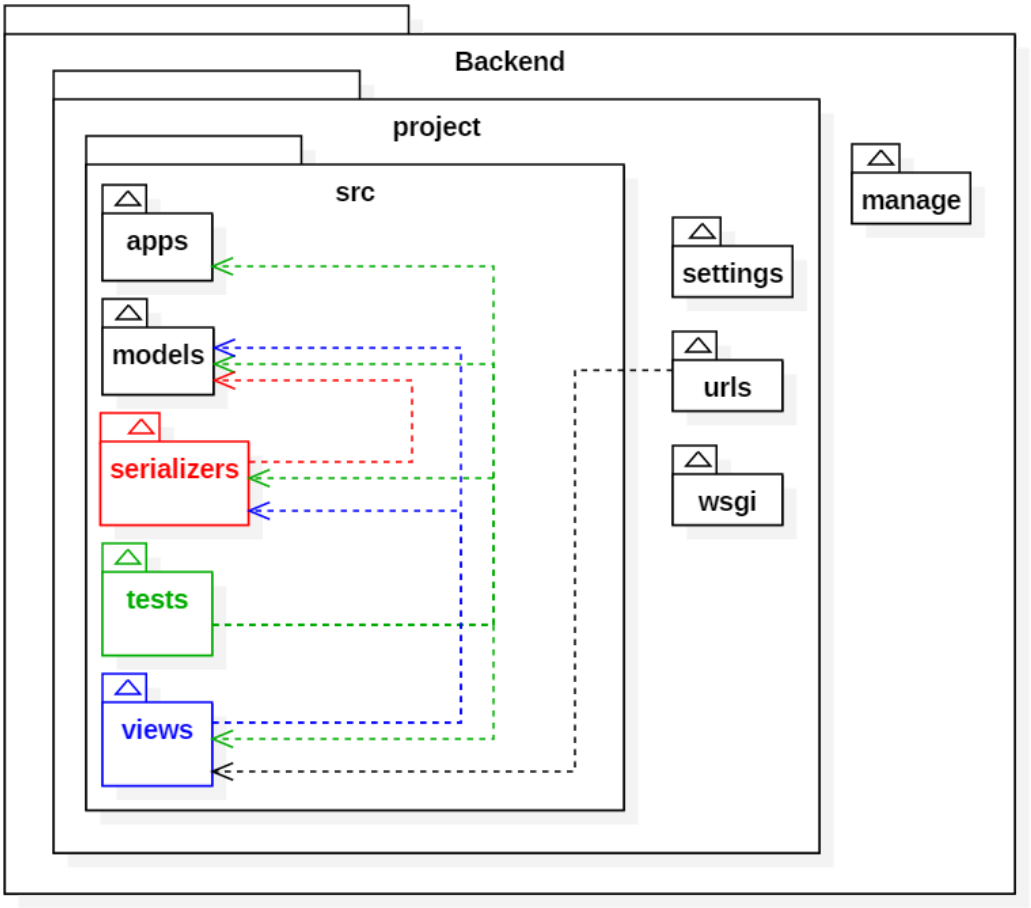


Abbildung 2: Paketstruktur des Backends

3.2.1 wsgi.py

Die Datei wsgi.py implementiert den WSGI-Python-Standard in das Django-Projekt. WSGI steht für Web Server Gateway Interface und ist im PEP 333 spezifiziert. PEP ist die Standard Python Konvention. Beim Starten eines Django-Projekts wird diese Datei angelegt und vom Entwickler nicht verändert.

3.2.2 settings.py

In der Datei settings.py werden die Grundeinstellungen des Systems beschrieben. Zu diesen gehören:

- benötigte Abhängigkeiten und Programme,
- Datenbankeinstellungen und
- Lokalitätseinstellungen.

3.2.3 url.py

In der url.py wird beschrieben über welche URL-Pfade welche Daten des Servers angesprochen werden. Genaueres hierzu folgt in Abschnitt 2.6 .

3.2.4 manage.py

Die Datei manage.py wird ebenfalls automatisch von Django angelegt und wird vom Entwickler nicht mehr modifiziert. Sie beinhaltet zentrale Funktionen zum Verwalten von Django. Darunter fallen Methoden wie beispielsweise die zum Starten des Servers.

3.3 Models

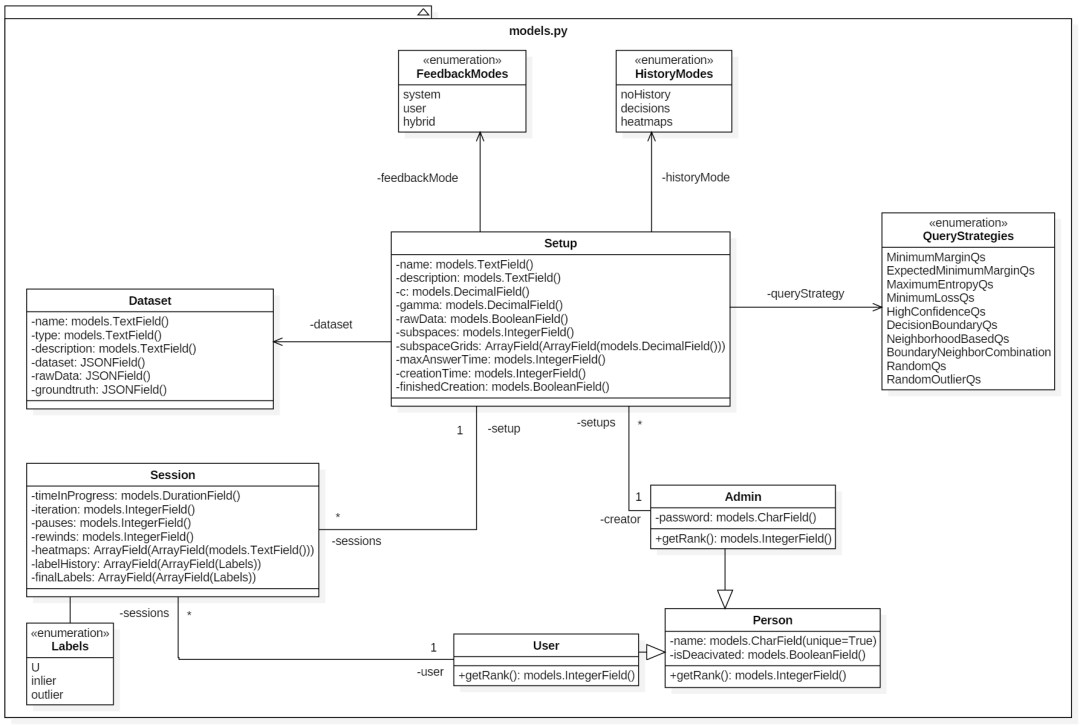


Abbildung 3: Übersicht aller Klassen in models.py

Die Hauptfunktion der Model-Klassen ist die Datenspeicherung. Sie enthalten die Daten mit denen das System arbeitet. Die einzelnen Klassen enthalten spezifische Werte der Daten welche gespeichert werden sollen. Dabei wird jedes Model als eine Datenbanktabelle materialisiert.

3.3.1 Dataset

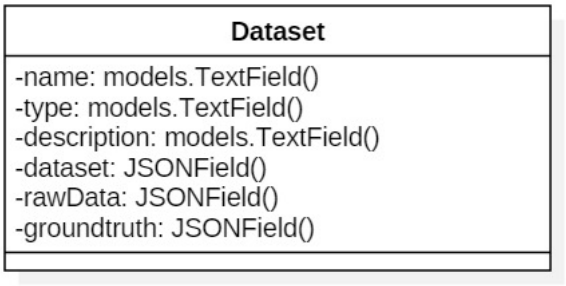


Abbildung 4: UML-Diagramm der Dataset-Klasse

Documentation:

Class Dataset describes a Dataset, its type, a description, its name and a groundtruth.

Attributes:

- *name: models.TextField()*: The name of the Dataset.
- *type: models.TextField()*: The type of the Dataset.
- *description: models.TextField()*: A description of the Dataset and what it represents.
- *dataset: JSONField()*: Feature data of the Dataset.
- *rawData: JSONField()*: The raw data of the Dataset. This is to help the User with his decisions.
- *groundTrouth: JSONField()*: A Dataset specific groundtrouth. This is used to determine whether or not active learning is practical.

3.3.2 Setup

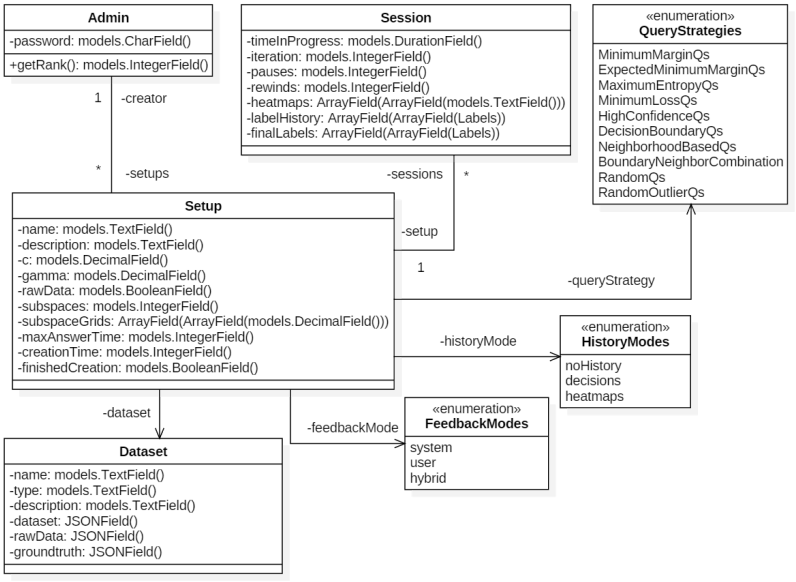


Abbildung 5: UML-Diagramm der Setup-Klasse

Documentation:

Class for modelling a Setup on the dataserver. The attributes are the parameters, which an Admin has chosen during the Setup-creation, the metadata and the average values over all Sessions, which are gathered automatically by the system and cannot be manipulated by the Admin. The following attributes are metadata:

- creationTime
- creator

The remaining attributes are the following parameters.

Attributes:

- *name: models.TextField()*: The unique name of the Setup .
- *description: models.TextField()*: A description of the Setup.
- *c: models.DecimalField()*: The c value needed for the OcalAPI.
- *gamma: models.DecimalField()*: The gamma value needed for the OcalAPI.
- *rawData: models.BooleanField()*: Determines whether or not the User is allowed to see the raw data.
- *subspaces: models.IntegerField()*: The number of subspaces the User is allowed to see.
- *subspaceGrids: ArrayField(ArrayField(models.DecimalField()))*: The scalingfactor of the heatmap grid.
- *maxAnswereTime: models.IntegerField()*: The maximum time for a User to select his label.
- *creationTime: models.IntegerField()*: The time when the Setup was created.
- *finishedCreation: models.BooleanField()*: whether the setup creation is final or not.

3.3.3 Session

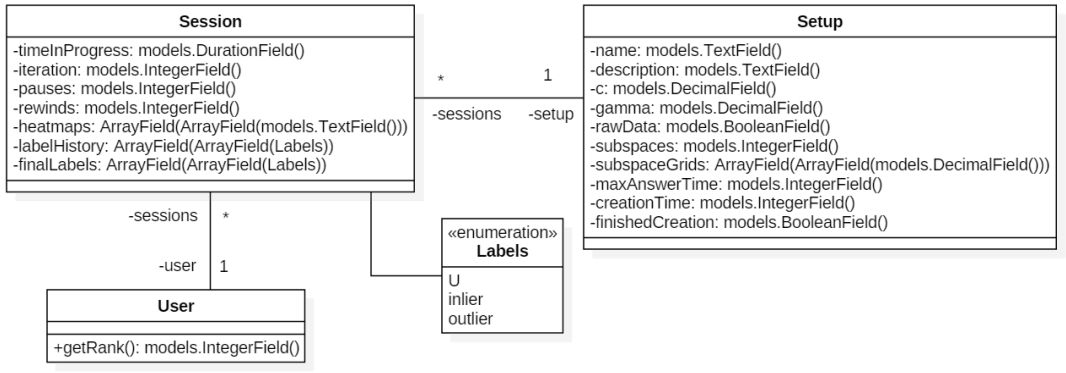


Abbildung 6: UML-Diagramm der Session-Klasse

Documentation:

A Session object represents a single Session. Each Session belongs to exactly one Setup and one User. The Session object contains all data collected for this Session: The time spent editing, the number of iterations, the number of pauses, the number of undone labels, the heatmaps of each iteration, as well as the chosen label of each iteration. The labels of each element at the end of a Session are also stored in the Session object.

Attributes:

- *timeInProgress: models.DurationField()*: The time passed since the Session has been started.
- *iteration: models.IntegerField()*: The current iteration.
- *pauses: models.IntegerField()*: The number of pauses the User has made.

- *rewinds: models.IntegerField()*: The number of rewinds the User has made.

- *heatmaps: ArrayField(ArrayField(models.TextField()))*: A set of heatmaps for this and previous iterations.

- *labelHistory: ArrayField(ArrayField(Labels))*: A set of all previously made labels.

- *finalLabels: ArrayField(ArrayField(Labels))*: A set of the final labels.

3.3.4 Person

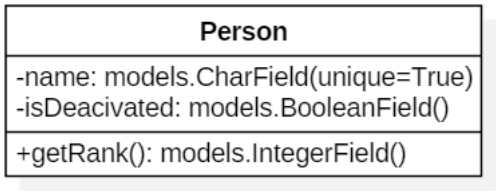


Abbildung 7: UML-Diagramm der Person-Klasse

Documentation:

Describes a Person interacting with the system.

Attributes:

- *name: models.CharField()*: The unique name of a Person. The uniqueness is guaranteed by the field ünique=True". If you try to create an object with the attribute name already existing a django.db.IntegrityError will be thrown.

- *isDeactivated: models.BooleanField()*: whether this Person is deactivated or not.

Methods:

- *getRank(): models.IntegerField()*: Returns the rank and therefore the rights of a Person.
  @return rank of a Person

3.3.5 Admin

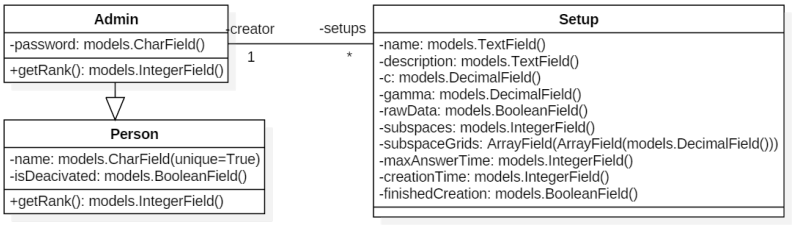


Abbildung 8: UML-Diagramm der Admin-Klasse

Documentation:
An Admin has control over the system. He can create, delete and edit Users, Sessions, Setups and Datasets.
Attributes:

- *password: models.CharField()*: The password for an Admin to log in to the system.

Methods:

- *getRank(): models.IntegerField()*: Returns the rank and therefore the rights of a

Person.

@return rank of a Person

3.3.6 User

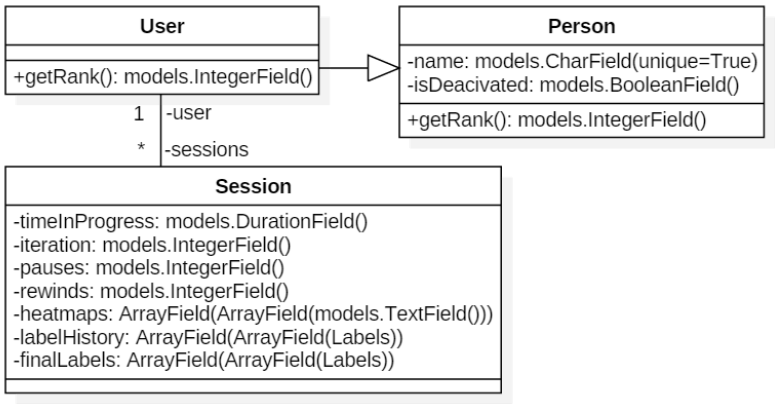


Abbildung 9: UML-Diagramm der User-Klasse

Documentation:

A User is an active participant of one or more Sessions.

Methods:

- *getRank(): models.IntegerField()*: Returns the rank and therefore the rights of a Person.

  @return rank of a Person

3.3.7 JSONDataset

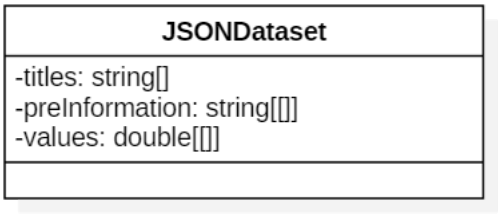


Abbildung 10: UML-Diagramm der JSONDataset-Klasse

Documentation:

The class JSONDataset describes the structure of the Dataset attribute in class Dataset.

Attributes:

- *titles: string[]*: Name of each dimension of the Dataset.
- *preInformation: string[[]]*: Metainformation for the data.
- *values: double[[]]*: The actual data of the Dataset.

3.3.8 FeedbackModes

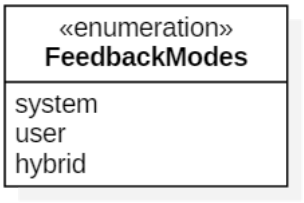


Abbildung 11: UML-Diagramm der FeedbackModes Enumeration

Documentation:

There are three different feedback modes. The feedback mode decides how the next object should be chosen.

Elements:

- *system*: The system decides which object should be labeled next (by using a certain query strategy stored in the attribute queryStrategy).
- *user*: The User chooses the next object himself.
- *hybrid*: The system gives a recommendation about which object should be labeled next, but the User can decide whether he follows the suggestion or chooses a different object. The query strategy in queryStrategy is used to give the recommendation.

3.3.9 HistoryModes

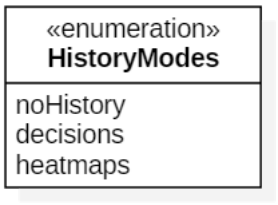


Abbildung 12: UML-Diagramm der HistoryModes Enumeration

Documentation:

There are three different history modes. The history mode decides if and what information should be given to the User about the past iterations.

Elements:

- *noHistory*: The User gets no information about his past iterations.
- *decisions*: The User can look up his past decisions (which label was chosen during which iteration).
- *heatmaps*: The User can look up the state of the heatmaps at the beginning of each iteration. He can also see how he labeled the object in this very iteration.

3.3.10 QueryStrategies

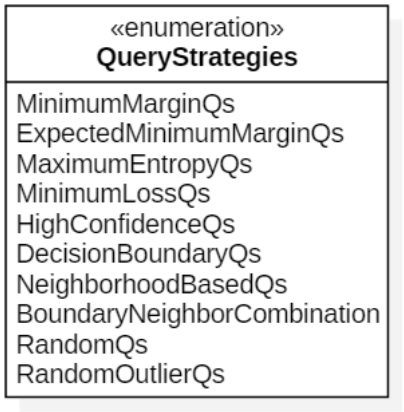


Abbildung 13: UML-Diagramm der QueryStrategies Enumeration

Documentation:

There are 10 different query strategies. The query strategy is used by the OcalAPI to determine which object should be labeled next. Only relevant if the system is choosing the object and not the User. The query startegies are implemented in the OcalAPI for more information read: *"An Overview and a Benchmark of Active Learning for One-Class Classification"*

Elements:

- *MinimumMarginQs*: This query strategy relies on the difference between posterior class probabilities.

- *ExpectedMinimumMarginQs*: This query strategy is almost equal to the *MinimumMarginQs*, but it assumes that the share of outliers is uniformly distributed.

- *MaximumEntropyQs*: This query strategy selects objects where the distribution of the class probability has a high entropy.

- *MinimumLossQs*: This query strategy selects objects by calculating the density for the cases where an object is an inlier or an outlier and then calculating the expected value.

- *HighConfidenceQs*: This query strategy selects objects which match the inlier class the least.

- *DecisionBoundaryQs*: This query strategy selects objects closest to the decision boundary.

- *NeighborhoodBasedQs*: This query strategy explores unknown neighborhoods in the feature space.

- *BoundaryNeighborCombination*: This query strategy selects objects by a linear combination of the distance to the hypersphere and the distance to the first nearest neighbor.

- *RandomQs*: This query strategy selects unlabeled objects randomly.

- *RandomOutlierQs*: This query strategy selects objects which are predicted to be outliers.

3.3.11 Labels

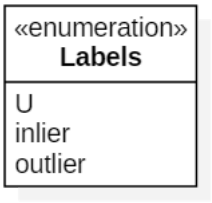


Abbildung 14: UML-Diagramm der Labels Enumeration

Documentation:

Represents the different possible labels of an element.

Elements:

- *U*: Elements labeled “U” have not yet been labeled.
- *inlier*: Elements labeled “inlier” have been labeled to be a “normal” element.
- *outlier*: Elements labeled “outlier” dont fit to the majority of other elements of the dataset.

3.4 Serializer

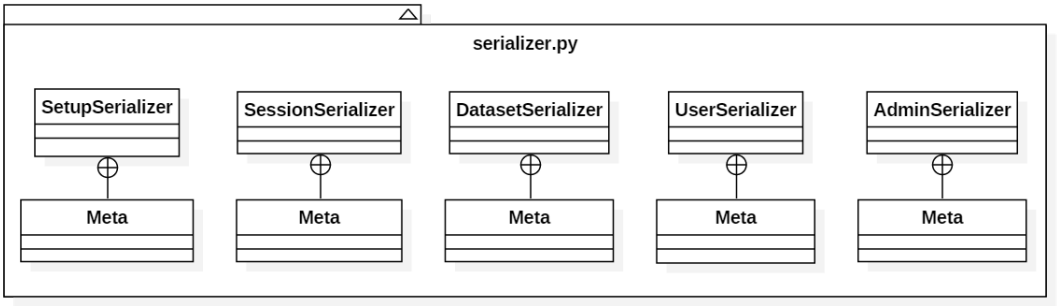


Abbildung 15: Simplifizierte Übersicht aller Klassen in serializer.py

Serializer werden dafür benötigt die Model Instanzen in das JSON Format zu konvertieren. Dabei erbt jeder spezifische Serializer von *ModelSerializer* und jeder hat eine eingebettete Klasse *Meta* welche jeweils ein Attribut *fields* hat. In *fields* sind die Attribute welche in der JSON Datei gespeichert werden sollen.

3.4.1 ModelSerializer

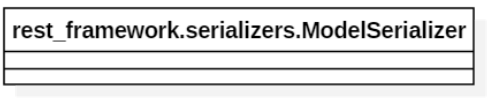


Abbildung 16: UML-Diagramm der ModelSerializer-Klasse

Documentation:

The ModelSerializer provides a regular serializer with elementary functions. All serializers used in this program are being extended from this class (further reading: http://www.cdrf.co/3.1/rest_framework.serializers/ModelSerializer.html)

3.4.2 DatasetSerializer

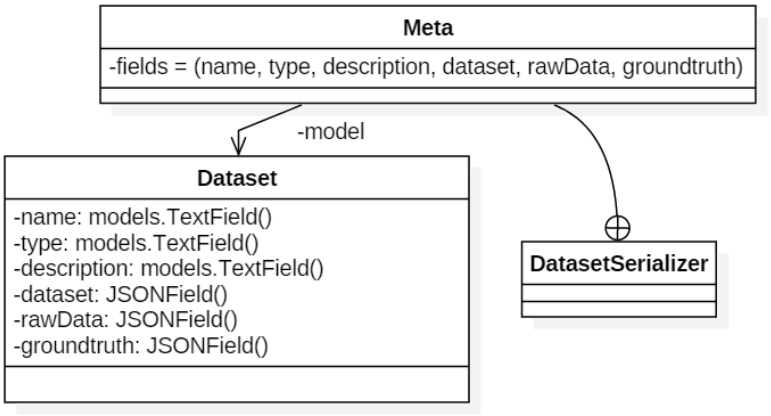


Abbildung 17: UML-Diagramm der DatasetSerializer-Klasse

Documentation:

Serializer for Datasets. Has a nested class **Meta** which has the following parameters:

- *fields = (name, type, description, dataset, rawData, groundtruth)*: The fields which are being serialized.

3.4.3 SetupSerializer

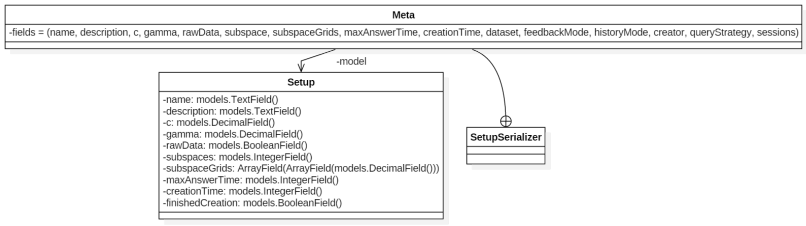


Abbildung 18: UML-Diagramm der SetupSerrializer-Klasse

Documentation:

Serializer for Setups. Has a nested class **Meta** which has the following parameters:

- *fields = (name, description, c, gamma, rawData, subspace, subspaceGrids, iterations, rewinds, maxAnswerTime, creationTime, avrgIterations, avrgPauses, avrgRewinds, avrgDuration, dataset, feedbackMode, historyMode, creator, queryStrategy, sessions)*: The fields which are being serialized.

3.4.4 SessionSerializer

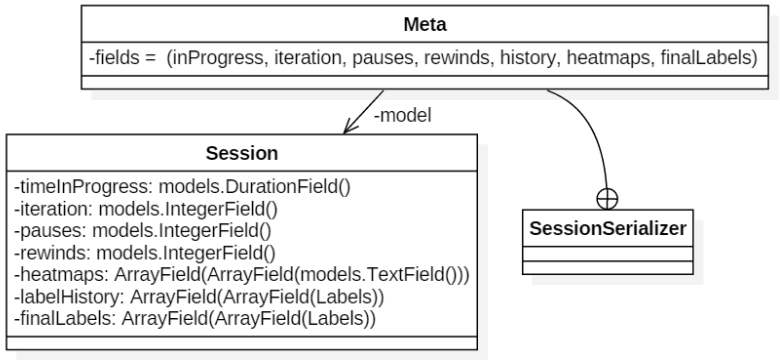


Abbildung 19: UML-Diagramm der SessionSerializer-Klasse

Documentation:

Serializer for Sessions. Has a nested class **Meta** which has the following parameters:

- *fields = (inProgress, iteration, pauses, rewinds, history, heatmaps, finalLabels)*: The fields which are being serialized.

3.4.5 PersonSerializer

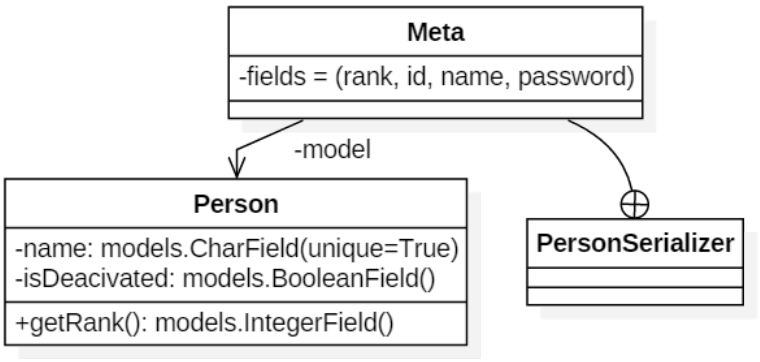


Abbildung 20: UML-Diagramm der PersonSerializer-Klasse

Documentation:

Serializer for Person. Has a nested class **Meta** which has the following parameters:

- *fields = (rank, id, name)*: The fields which are being serialized.

3.4.6 AdminSerializer

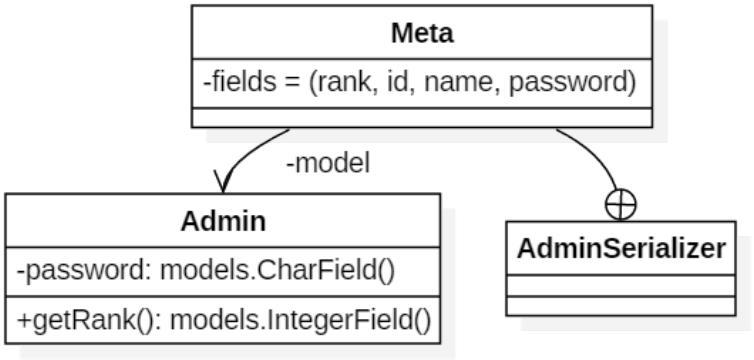


Abbildung 21: UML-Diagramm der AdminSerializer-Klasse

Documentation:

Serializer for Admins. Has a nested class **Meta** which has the following parameters:

- *fields: (rank, id, name, password)*: The fields which are being serialized.

3.4.7 UserSerializer

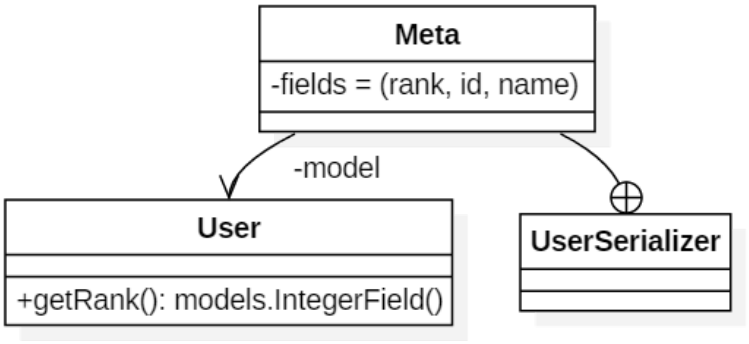


Abbildung 22: UML-Diagramm der UserSerializer-Klasse

Documentation:

Serializer for Users. Has a nested class **Meta** which has the following parameters:

- *fields = (name, type, description, dataset, rawData, groundtruth)*: The fields which are being serialized.

3.5 Views

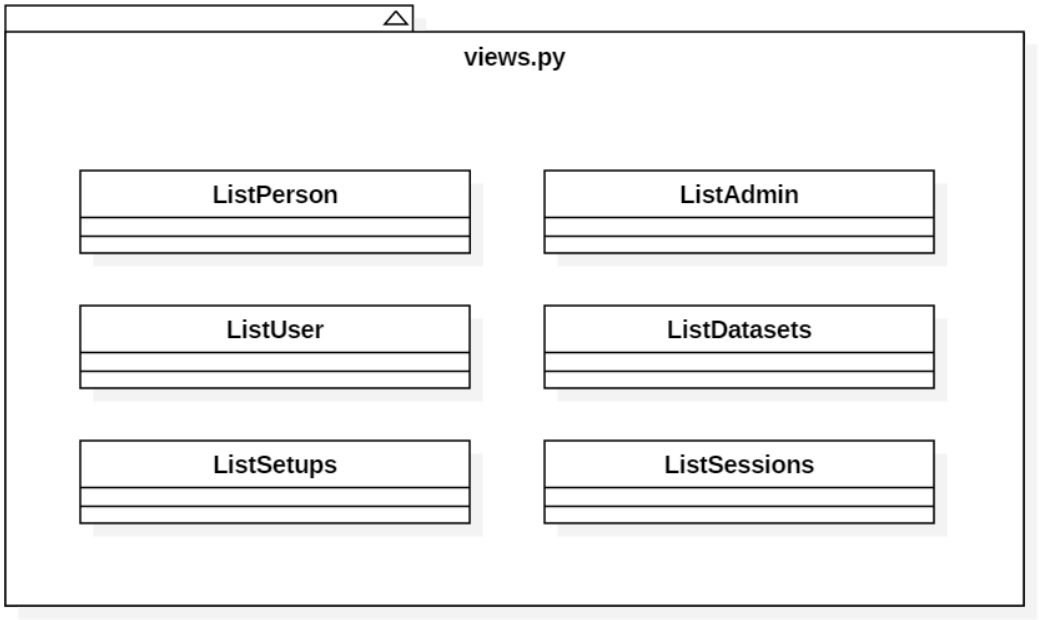


Abbildung 23: Vereinfachte Übersicht aller Klassen in views.py

Die folgenden Klassen sind zum Auflisten und Filtern von Models. Sie erben alle von der ListApiView Klasse aus rest_framework.generics. Von der ListApiView Klasse wird außerdem die Methode as_view() vererbt.

3.5.1 ListDataset

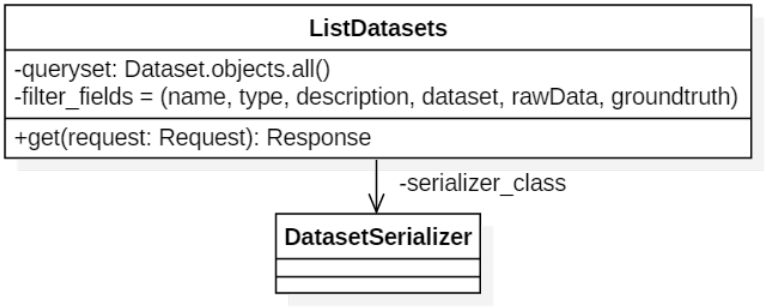


Abbildung 24: UML-Diagramm der ListDataset-Klasse

Documentation:

Class for listing and filtering Dataset objects.

Attributes:

- *queryset: Dataset.objects.all()*: List of all Dataset objects.
- *filter_fields = (name, type, description, dataset, rawData, groundtruth)*: The attributes the REST-API filters by.

Methods:

- *get(Request request): Response*: Performs changes to the database by requests from the API.
  @param request the request from the API.
  @return the response of the server.

3.5.2 ListSetup

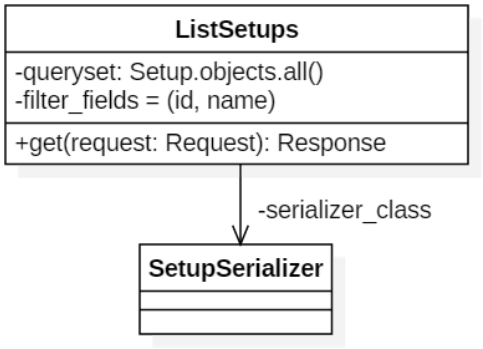


Abbildung 25: UML-Diagramm der ListSetup-Klasse

Documentation:

Class for listing and filtering Setup objects.

Attributes:

- *queryset: Setup.objects.all()*: List of all Setup objects.
- *filter_fields = (id, name)*: The attributes the REST-API filters by.

Methods:

- *get(Request request): Response*: Performs changes to the database by requests from the API.
  @param request the request from the API.
  @return the response of the server.

3.5.3 ListSession

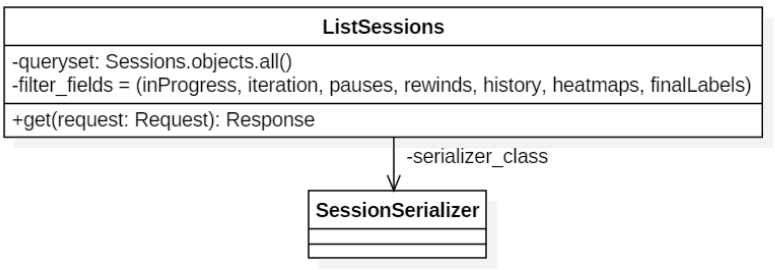


Abbildung 26: UML-Diagramm der ListSession-Klasse

Documentation:

Class for listing and filtering Session objects.

Attributes:

- *queryset: Session.objects.all()*: List of all Session objects.
- *filter_fields = (inProgress, iteration, pauses, rewinds, history, heatmaps, finalLabels)*: The attributes the REST-API filters by.

Methods:

- *get(Request request): Response*: Performs changes to the database by requests from the API.
  @param request the request from the API.
  @return the response of the server.

3.5.4 ListPerson

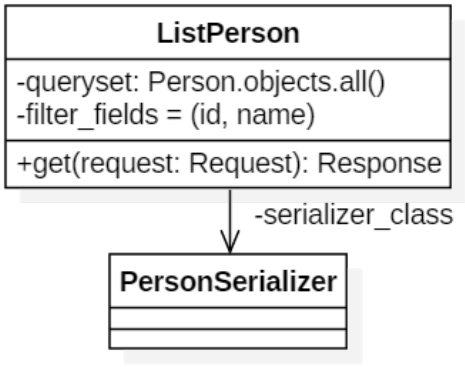


Abbildung 27: UML-Diagramm der ListPerson-Klasse

Documentation:

Class for listing and filtering Person objects.

Attributes:

- *queryset: Person.objects.all()*: List of all Person objects.
- *filter_fields = (id, name)*: The attributes the REST-API filters by.

Methods:

- *get(request: Request): Response*: Performs changes to the database by requests from the API.
  @param request the request from the API.
  @return the response of the server. If an error occurs while the request is being processed, it is intercepted and forwarded as a response.

3.5.5 ListAdmin

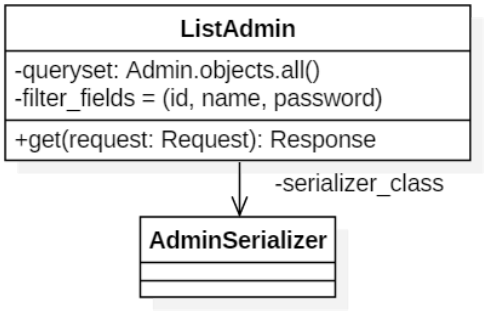


Abbildung 28: UML-Diagramm der ListAdmin-Klasse

Documentation:

Class for listing and filtering Admin objects.

Attributes:

- *queryset: Admin.objects.all()*: List of all Admin objects.
- *filter_fields = (id, name, password)*: The attributes the REST-API filters by.

Methods:

- *get(request: Request): Response*

3.5.6 ListUser

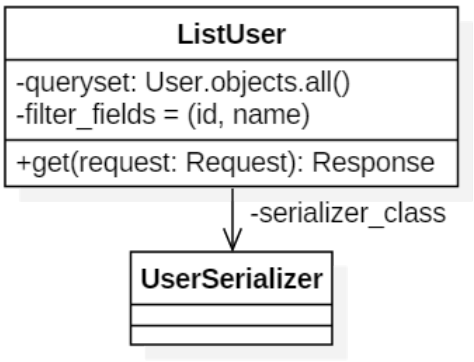


Abbildung 29: UML-Diagramm der ListUser-Klasse

Documentation:

Class for listing and filtering User objects.

Attributes:

- *queryset: User.objects.all()*: List of all User objects.
- *filter_fields = (id, name)*: The attributes the REST-API filters by.

Methods:

- *get(request: Request): Response*

3.6 URL

Für jede der im letzten Kapitel genannten Listen wird in der Datei url.py ein Pfad definiert. Der Pfad ist immer der Listenname in Kleinbuchstaben. Zusätzlich exisitiert noch der Pfad OcalAPI. Bei einem Aufruf von diesem Pfad wird eine Methode, welche in der url.py implementiert ist, aufgerufen. Diese Methode erhält ein JSON Objekt vom Client und stellt eine Anfrage an die OcalAPI. Zurück kommt die Klassifikation der API.

3.7 JSON classes

Die folgenden Klassen modellieren andere Klassen als .json-Datei. Also zum Beispiel welchen Aufbau die .json-Datei für ein Setup usw. hat.

3.7.1 DatasetJSON

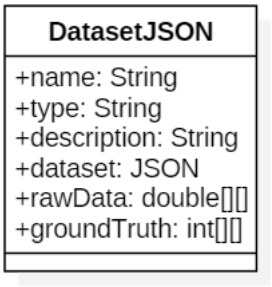


Abbildung 30: UML-Modelierung des Dataset.json-Objects

Documentation:

The style of the .json-file which represents a Dataset.

Attributes:

- *name: String*: The name of the Dataset.
- *type: String*: The name of the Dataset type.
- *description: String*: The description of the dataset.
- *dataset: JSON*: The Dataset itself.

- *rawData: double[[]]*: The rawData out of which the feature-data was calculated.

- *groundTrouth: int[[]]*: The groundtruth corresponding to the Dataset (if existent).

3.7.2 SetupJSON

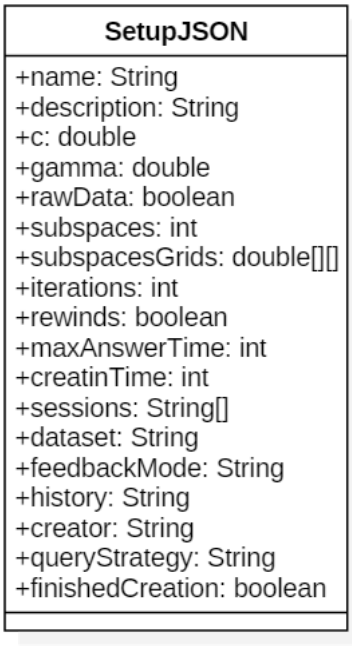


Abbildung 31: UML-Modelierung des Setup.json-Objects

Documentation:

The style of the .json-file which represents a Setup.

Attributes:

- *name: String*: The name of the Setup.

- *description: String*: The description of the Dataset.
- *c: double*: The c parameter passed on to the OcalAPI.
- *gamma: double*: The gamma-parameter passed on to the OcalAPI.
- *rawData: boolean*: The decision whether the user gets displayed the rawData or not.
- *subspaces: int*: The amount of subspaces which should be taken into consideration when evaluating an object.
- *subspacesGrids: double[[]]*: The scalingfactor of the heatmap grid.
- *iterations: int*: The maximum allowed iterations.
- *rewinds: boolean*: Whether the User is allowed to revise his last decision or not.
- *maxAnswereTime: int*: The maximum allowed time for deciding on a label.
- *creationTime: int*: The unix-timestamp when the Setup has been created.
- *sessions: String[]*: Names of all corresponding Sessions.
- *dataset: String*: Name of the Dataset used in this Setup.
- *feedbackMode: String*: The name of the feedback mode used in this Setup.
- *history: String*: The name of the history-mode used in this Setup.

- *creator: String*: The name of the account which has been used for creating this Setup.

- *queryStrategie: String*: The name of the queryStrategy used in this Setup.

- *fininshedCreation: boolean*: Whether the creation-process is finished.

3.7.3 SessionJSON

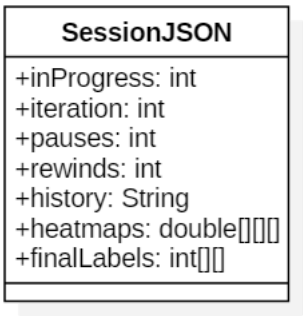


Abbildung 32: UML-Modelierung des Session.json-Objects

Documentation:

The style of the .json-file which represents a Session.

Attributes:

- *inProgress: int*: The time in which the Session has been worked on so far.

- *iterations: int*: The amount of iterations already completed so far.

- *pauses: int*: The amount of pauses by the User so far.
- *rewinds: int*: The amount of rewinds by the User so far.
- *history: String*: Whether and how much history the User gets shown. String represents the name of an enum-object of historyModes.
- *heatmaps: double[[[]]]*: All necessary heatmaps (if historyMode is set to heatmaps all heatmaps, else only the current one) represented by their points.
- *finalLabels: int[[]]*: The labeling of the objects so far.

3.7.4 PersonJSON

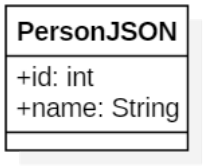


Abbildung 33: UML-Modelierung des Person.json-Objects

Documentation:

The style of the .json-file which represents a Person.

Attributes:

- *id: int*: The ID of the Person.

- *name: String*: The name of the Person.

3.7.5 AdminJSON

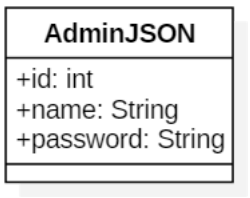


Abbildung 34: UML-Modelierung des Admin.json-Objects

Documentation:

The style of the .json-file which represents an Admin.

Attributes:

- *id: int*: The ID of the Admin.
- *name: String*: The name of the Admin.
- *password: String*: The password of the Admin.

3.7.6 UserJSON

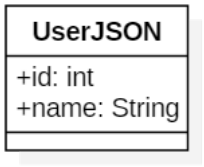


Abbildung 35: UML-Modelierung des User.json-Objects

Documentation:

The style of the .json-file which represents an User.

Attributes:

- *id: int*: The ID of the User.
- *name: String*: The name of the User.

4 Frontend

4.1 Einleitung

Das Frontend des Systems wird mit dem Framework Angular realisiert. Dieses wird in Typescript implementiert und nach JavaScript transpiliert.

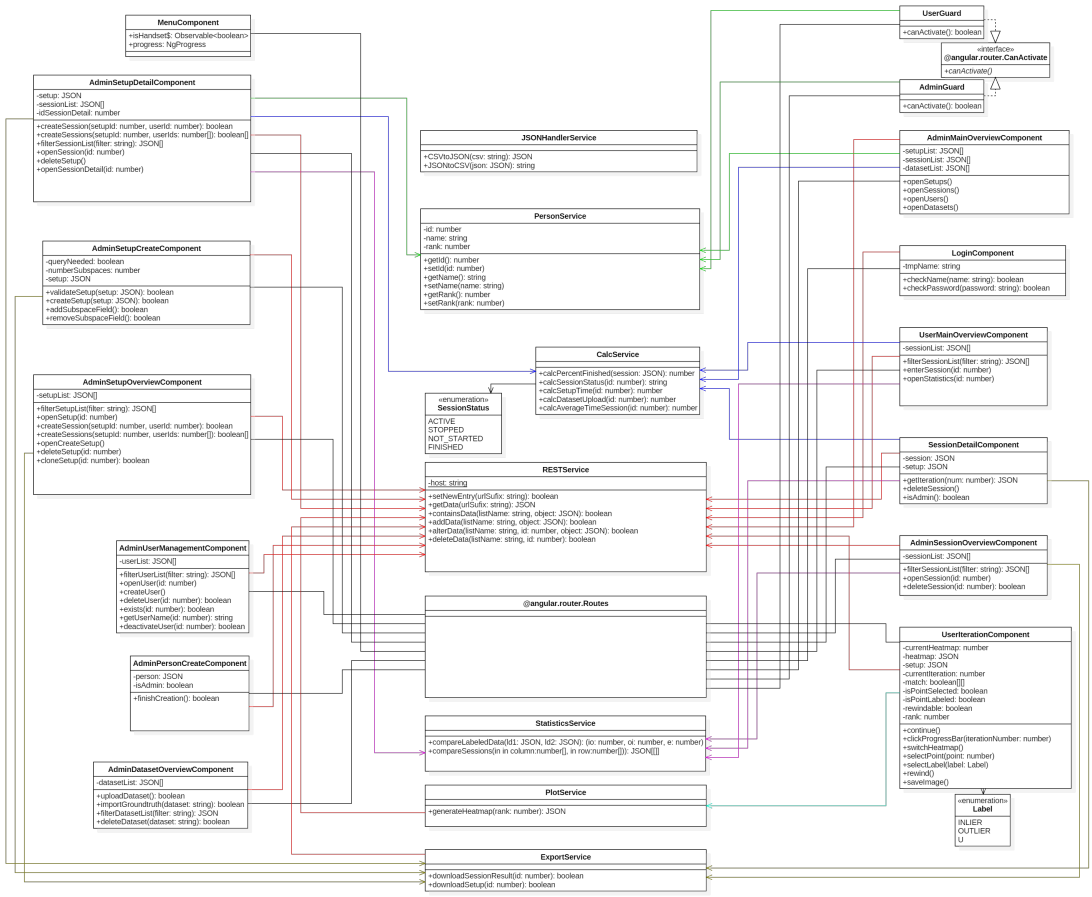


Abbildung 36: Übersicht aller Klassen im Frontend

4.2 Model

Im Model befinden sich die Components. Jedes Component definiert eine Seite des Frontends auf die Admins beziehungsweise User zugreifen können.

4.2.1 SessionDetailComponent

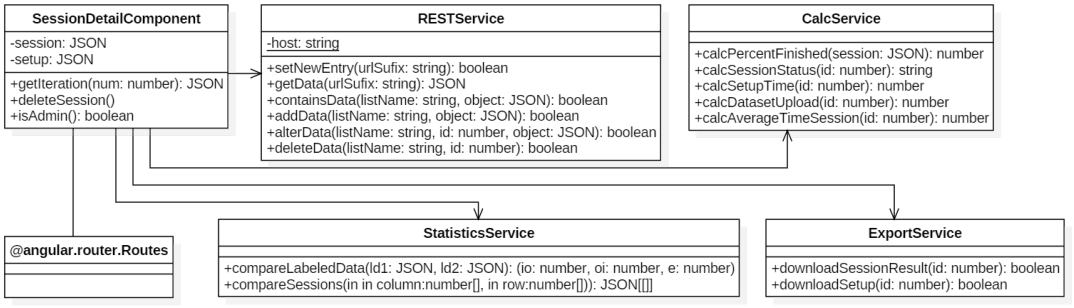


Abbildung 37: UML-Diagramm der Session-Detail-Komponente

Documentation:

Component for the detailed view of a Session. Provides information to this Session if its active, when its finished a detailed summary is shown.

Attributes:

- *session: JSON*: The JSON object containing all data to this Session.
- *setup: JSON*: The Setup configuration file for this Session.

Methods:

- *getIteration(number num): JSON*: Provides the data of the requested iteration.
  @param num the iteration number.
  @return a JSON object containing the data of this iteration.

- *deleteSession(): boolean*: Deletes a Session from the system. @return whether the Session was successfully deleted.

- *isAdmin(): boolean*: Checks if the current Person is an Admin.
  @return true if the Person is an Admin, false if not.

4.2.2 LoginComponent

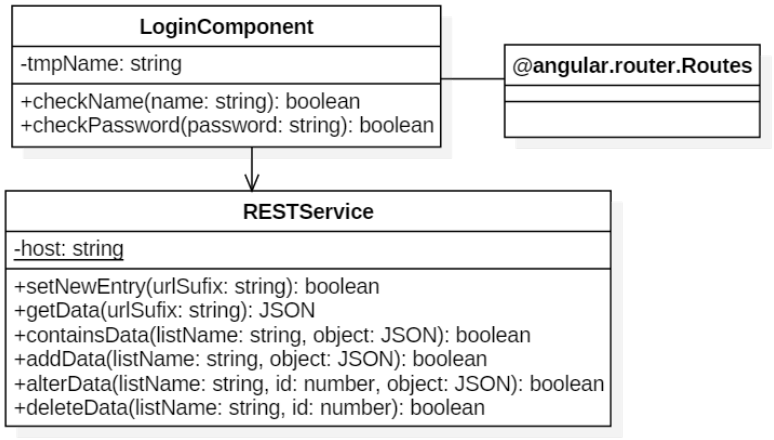


Abbildung 38: UML-Diagramm der Login-Komponente

Documentation:

This component serves the purpose of modeling the Login-Site, which is being shown to every Person who is accessing the iFeed website. If the Person is a User, the login process is finished after the name has been entered and the website is showing the User overview. Otherwise, if the Person is an Admin, a password field is being shown in addition and the Admin has to successfully enter his password to get to his overview page.

Attributes:

- *tmpName: string*: Needed to temporarily save an Admin's name to check validity of name and password combination.

Methods:

- *checkName(name: string): boolean*: This method checks if the entered name is referenced by a Person in this system. If the Person is an Admin, the name is being stored temporarily in tmpName.
  @param name the name which has been entered in the text field.
  @return true if the name is belonging to a person in this system, false otherwise.

- *checkPassword(password: string): boolean*: Checks if the entered name and password combination is valid.
  @param password the entered Admin password.
  @return true if the passwort is correct for this Admin, false if not.

4.2.3 MenuComponent

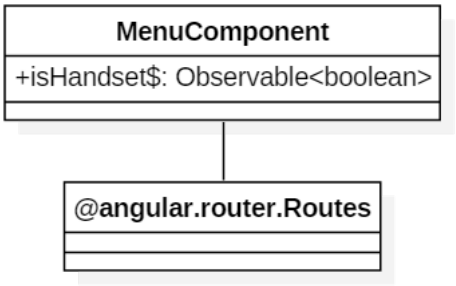


Abbildung 39: UML-Diagramm der Menu-Komponente

Documentation:

Component for the menu overview. This is for Admins and Users to navigate through the website.

Attributes:

- *isHandset$: Observable<boolean>*: Is used to observe components.

- *progress: NgProgress*: A loading bar which is used with @ViewChild in all components to visualize loading processes, while requesting data via the RESTService.

4.2.4 UserIterationComponent

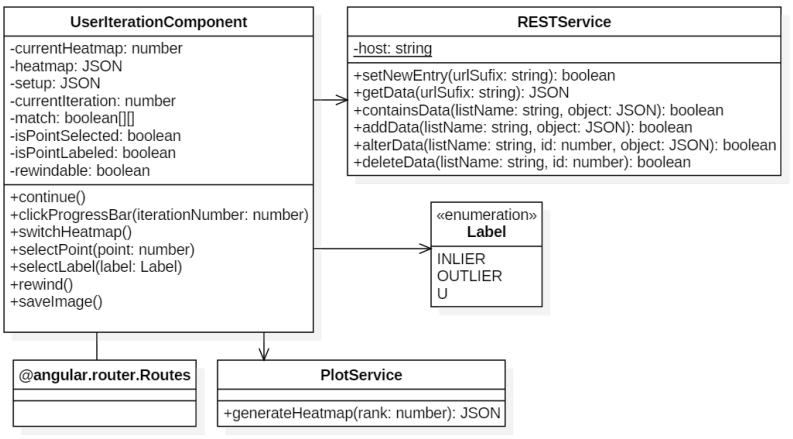


Abbildung 40: UML-Diagramm der User-Iteration-Komponente

Documentation:

Represents an iteration in a Session, executed by a User.

Attributes:

- *currentHeatmap: number*: The number of the heatmap which is currently shown to the User.
- *heatmaps: string[]*: An array of all heatmaps which can be displayed in this iteration.
- *setup: JSON*: The .json-file representing the Setup, which belongs to this Session.
- *currentIteration: number*: A counter for the current iteration.
- *match: boolean[[]]*: Field needed to save evaluations of previous iterations.
- *isPointSelected: boolean*: Indicates if the User has selected a point (only needed in

self choosing feedback mode).

- *isPointLabeled: boolean*: Indicates if the User has done an evaluation of a datapoint.

- *rewindable: boolean*: whether it should be possible to go back to the last iteration, or not.

Methods:

- *continue(): void*: Continues the Session with the next iteration, if a point has been selected.

- *clickProgressBar(iterationNumber: number): void*: Shows the specific subspace of the selected iteration.
  @param iterationNumber the iteration number (has to be smaller than currentIteration).

- *switchHeatmap(): void*: Switches between different heatmaps in one iteration.

- *selectPoint(point: number): void*: Selects a point in the heatmap. If required, detailed information (e.g. raw data) for this point is being presented.
  @param point the point in the heatmap to be selected.

- *selectLabel(label: Label): void*: Selects a label to a selected point.
  @param label one of the defined labels: inlier, outlier and u for default value.

- *rewind()*: lets the user redo the last iteration

4.2.5 Label

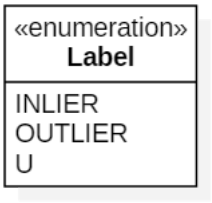


Abbildung 41: UML-Diagramm der Label Enumeration

Documentation:

Enumeration for the valid types of data labels.

Elements:

- *Inlier*: label for inlier
- *Outlier*: label for outlier
- *U*: unlabeled

4.2.6 UserMainOverviewComponent

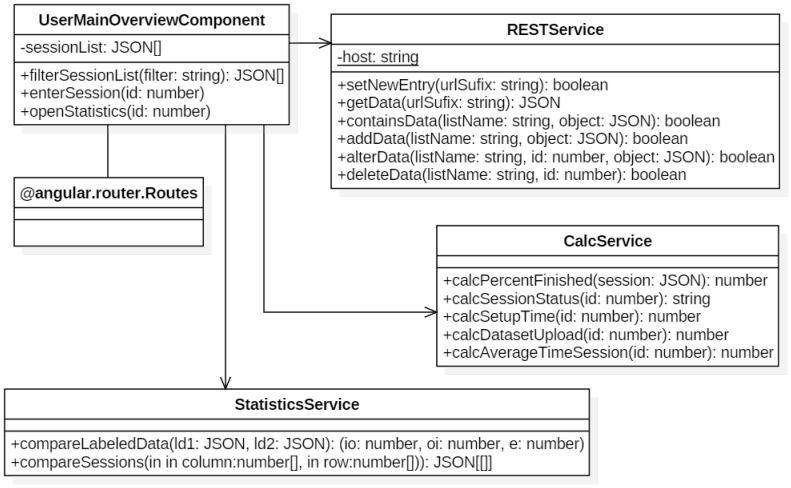


Abbildung 42: UML-Diagramm der main User-Overview-Komponente

Documentation:

Component for the main overview page of the User. The currently running Sessions and finished ones are being shown here.

Attributes:

- *sessionList: JSON[]*: A list of all Sessions, available to this User.

Methods:

- *filterSessionList(filter: String): JSON[]*: Filters the given Session list on behalf of the provided input. Serves as a search function in the User overview.
  @param filter the string through which the Sessions are being searched.
  @return an array of JSON objects, meaning all Sessions which contain the provided string in their names.

- *enterSession(id: number): void*: Enters the seleted Session, if it is not finished. If the Session hasn't been started before, the Session is being started.
  @param id the unique Session id, which the User has selected by clicking on the Session in the provided list.

- *void openStatistics(number id)*: Shows Session statistics to a finished Session.
  @param id the unique Session id, which the User has selected by clicking on the Session in the provided list.

4.2.7 AdminMainOverviewComponent

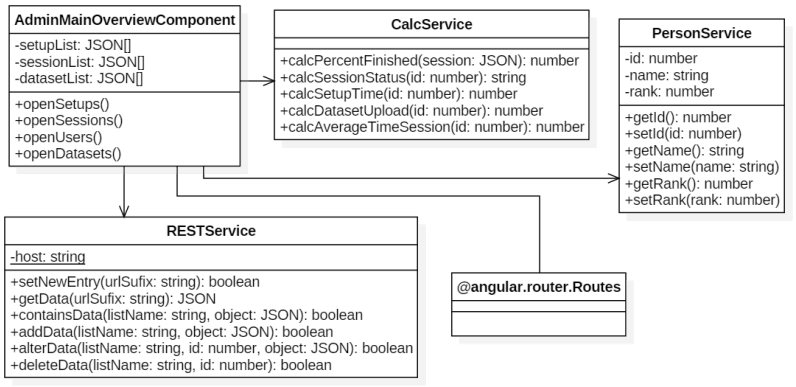


Abbildung 43: UML-Diagramm der main Admin-Overview-Komponente

Documentation:

This component implements the overview menu which is shown to the Admin after login. It provides an overview of the last created Setups, imported Datasets and currently running or finished Sessions.

Attributes:

- *setupList: JSON[]*: A list containing all Setups in the system.

- *sessionList: JSON[]*: A list containing all Sessions in the system.

- *datasetList: JSON[]*: A list containing all Datasets in the system.

Methods:

- *openSetup(): void*: Opens the Setup overview page.

- *openSession(): void*: Opens the Session overview page.

- *openUser(): void*: Opens the User overview page.

- *openDataset(): void*: Opens the Dataset overview page.

4.2.8 AdminDatasetOverviewComponent

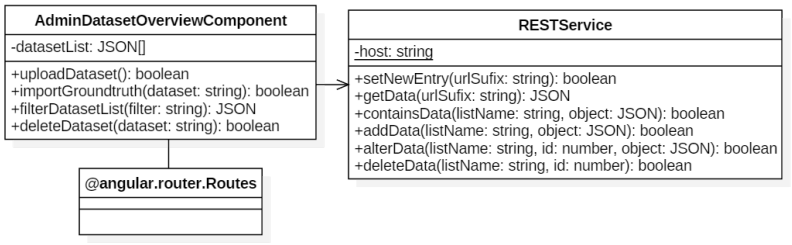


Abbildung 44: UML-Diagramm der Admindatensetübersicht-Komponente

Documentation:

Component for the Dataset management page. This page gives an logged in Admin the possibility to manage (upload/delete/import groundtruth/view) all Datasets.

Attributes:

- *datasetList: JSON[]*: List of all JSONs, with each representing a Dataset. Every Dataset is represented only once.

Methods:

- *uploadDataset(): boolean*: Opens a dialog, so the User can select a Dataset-file on his computer. Adds this Dataset to the datasetList and sends it via RESTService to the server.
  @return true: upload was successful and Dataset has been send to the server. false: An error occured during the uploading process.

- *importGroundTrouth(dataset: string): boolean*: Opens a file-explorer, so the User can select a groundtruth-file. Adds this groudtruth to the corresponding Dataset and uploads it via the RESTService to the server.
  @param dataset: The name of the Dataset to which a groundtruth should be imported.
  @return true: if the upload-process was successful and the groundtruth has been send to the server false: if an error occured during the upload-process.

- *filterDatasetList(filter: string): JSON*: Browses the datasetList for any dataset which correlates with the given filter String (the name contains/matches the filter).
  @param filter: the String for which the datasetList should be searched.
  @return JSON[]: contains all Datasets with the filter String in their name, as JSONs

- *deleteDataset(dataset: string): boolean*:Deletes a Dataset by removing them from the datasetList. Also sends the information about the deletion via the RESTService to the server.
  @param dataset: name of the Dataset which should be deleted.
  @return true: if deletion was successful. false: if an error occured during deletion.

4.2.9 AdminSessionOverviewComponent

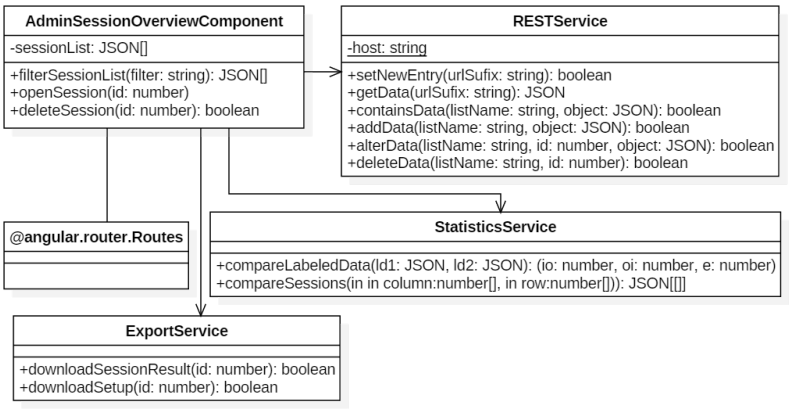


Abbildung 45: UML-Diagramm der Adminsessionübersicht-Komponente

Documentation:

Component for the Session overview page. This page is represening the entry point to the Session menu.

Attributes:

- *sessionList: JSON[]*: A list of all Sessions in this system.

Methods:

- *filterSessionList(filter: string): JSON[]*: Filters the given Session list on behalf of the provided input. Serves as a search function in the Session overview.
  @param filter the string on which the Sessions are being searched for.
  @return an array of JSON objects, meaning all Sessions which contain the provided string in their names.

- *openSession(id: number): void*: Opens a new window with detailed information to

the selected Session.
@param id the unique Session id.

- *void deleteSession(number id)*: Deletes an existing Session.
  @param id the unique Session id.

4.2.10 AdminSetupOverviewComponent

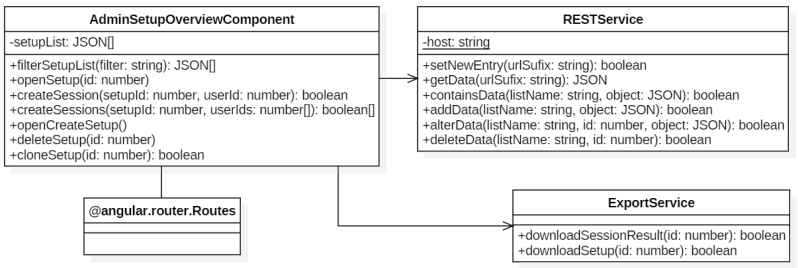


Abbildung 46: UML-Diagramm der Adminsetupübersicht-Komponente

Documentation:
Component for the Setup overview page. This page is representing the entry point to the Setup configuration. Setups can be edited/created/cloned and Sessions can be started.
Attributes:

- *setupList: JSON[]*: A list of all Setups in this system.

Methods:

- *filterSetupList(filter: String): JSON[]*: Filters the given Setup list on behalf of the provided input. Serves as a search function in the Setup overview.
  @param filter the string through which the Setups are being searched.

@return an array of JSON objects, meaning all Setups which contain the provided string in their names.

- *openSetup(id: number): void*: Opens a new window with detailed information to the selected Setup.
  @param id the unique Setup id.

- *boolean createSession(number setupId, number userId)*: Creates a new Session for a User to an existing Setup.
  @param setupId the unique Setup id.
  @param userId the unique User id.
  @return true if the Session creation was successful, false if not.

- *createSessions(setupId: number, userIds: number[]): boolean[]*: Creates multiple Sessions for multiple Users to an existing Setup.
  @param setupId the unique Setup id.
  @param userIds the unique User ids.
  @return a boolean array, its fields are true if the creation was successful, false if not.

- *openCreateSetup(): void*: Opens the create Setup page.

- *deleteSetup(id: number): void*: Deletes an existing Setup.
  @param id the unique Setup id.

- *cloneSetup(id: number): boolean*: Creates an identical copy of the provided Setup.
  @param id the unique Setup id. @return whether the the setup was successfully cloned.

4.2.11 AdminSetupCreateComponent

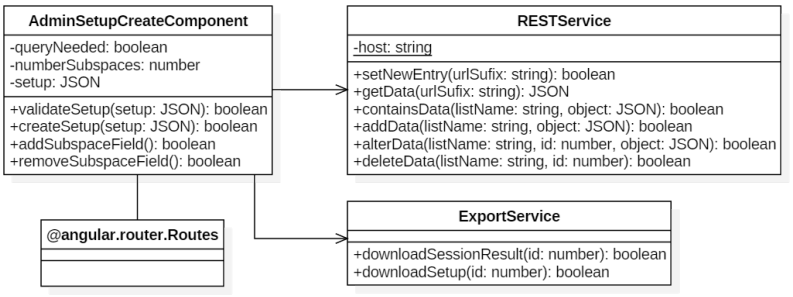


Abbildung 47: UML-Diagramm der Adminsetuperstell-Komponente

Documentation:

Component for the creation of a new Setup which is being done by the Admin.

Attributes:

- *queryNeeded: boolean*: Indicates if the Admin has selected a query strategy.
- *numberSubspaces: number*: Shows the amount of subspaces the Admin has chosen.
- *setup: JSON*: The Setup configuration file as a JSON object.

Methods:

- *validateSetup(setup: JSON): boolean*: Validates the correctness of an entered Setup.
  @param setup the entered Setup as JSON (defined in backend documentation)
  @return true if the Setup is valid, false if not.

- *createSetup(setup: JSON): boolean*: Creates a new Setup according to the entered configuration file.
  @param setup the entered configuration file.
  @return true if the creation was successful, false if not.

- *addSubspaceField(): boolean*: Adds a subspace field to the screen.
  @return true if creation was successful, false if not.

- *removeSubspaceField(): boolean*: Removes the subspace field from the screen.
  @return true if the removal was successful, false if not.

4.2.12 AdminSetupDetailComponent

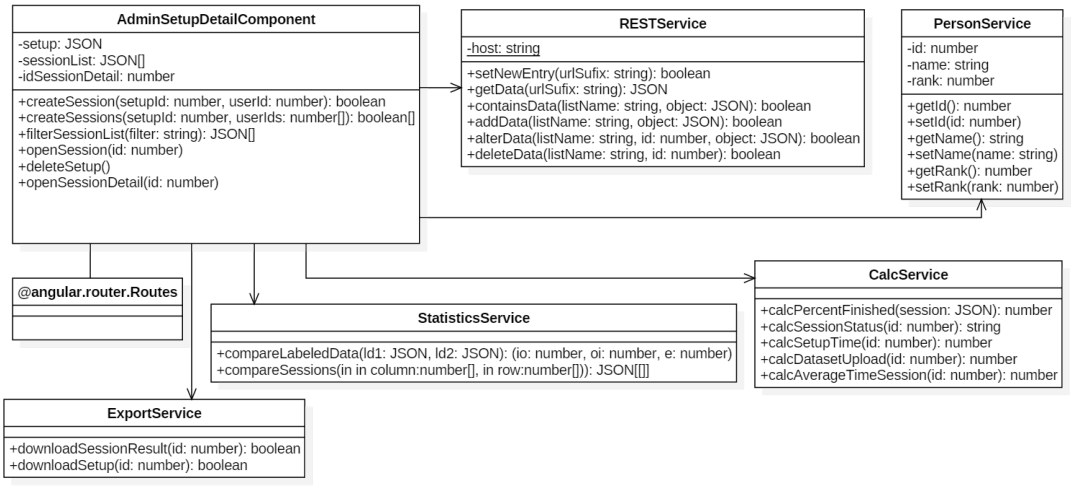


Abbildung 48: UML-Diagramm der Adminsetupdetail-Komponente

Documentation:

Component for the detailed view of a Setup. It serves its purpose in maintaining the Setup and the respective Sessions.

Attributes:

- *setup: JSON*: The configuration file to the Setup which is being displayed.

- *sesionList: JSON[]*: A list of all created Sessions to this specific Setup.

- *idSessionDetail: number*: The unique Session id, if one is selected.

Methods:

- *createSession(setupId: number, userId: number): boolean*: Creates a new Session for a User to an existing Setup.
  @param setupId the unique Setup id.
  @param userId the unique User id.
  @return true if the Session creation was successful, false if not.

- *createSessions(setupId: number, userIds: number[]): boolean[]*: Creates multiple Sessions for multiple Users to an existing Setup.
  @param setupId the unique Setup id.
  @param userIds the unique User ids.
  @return a boolean array, its fields are true if the creation was successful, false if not.

- *filterSessionList(filter: String): JSON[]*: Filters the given Setup list on behalf of the provided input. Serves as a search function in the Setup overview.
  @param filter the string through which the Setups are being searched.
  @return an array of JSON objects, meaning all Setups which contain the provided string in their names.

- *openSession(id: number): void*: Show a specific Session, when selected in the list.
  @param id the unique Session id.

- *deleteSetup(): void*: Deletes the Setup and, if demanded, also all Sessions belonging to this Setup.

- *openSessionDetail(id: number): void*: Shows details to the selected Session. If the Session is finished, the summary is shown.
  @param id the unqiue Session id.

4.2.13 AdminPersonCreateComponent

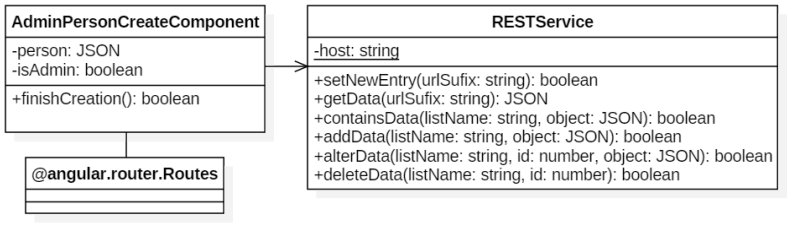


Abbildung 49: UML-Diagramm der Benutzererstellungs-Komponente

Documentation:

Component for the User creation page. Opens up when an Admin wants to create an new User/Admin. Contains all information currently given during the creation process. Also checks all input on validation.

Attributes:

- *person: JSON*: Contains the currently entered information about the User. Is updated everytime an information is changed. Will be send via RESTSerivce to the server when creation is finished.

- *isAdmin: boolean*: whether the current person is an Admin.

Methods:

- *finishCreation(): boolean*: Creates an new User with the information provided within the User JSON. If the JSON is not filled with sufficient information an error will be displayed.
  @return true: User creation was successful. Closes the page for User creation. false: error occured during User creation. Page will not be closed.

- *checkUsername(): boolean*: Checks the currently entered User name in the User JSON for correctness and uniqueness.
  @return true: The currently entered User name does not contain any forbidden characters and also is unique throughout the whole system. false: the User name either contains forbidden symbols or is already taken.

4.2.14 AdminUserManagementComponent

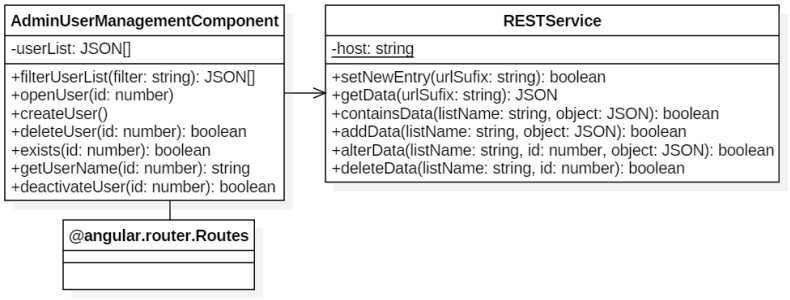


Abbildung 50: UML-Diagramm der Benutzerverwaltungs-Komponente

Documentation:

Component for the User management page. This page gives an logged in Admin the possibility to manage (create/delete/view) the registered Users.

Attributes:

- *userList: JSON[]*: List containing all registered Users.

Methods:

- *filterUserList(filter: string): JSON[]*: Browses the userList for any User which correlates with the given filter String (the name contains/matches the filter).
  @param filter: the String for which the userList should be searched.
  @return JSON[]: Contains all Users with the filter String in their name, as JSONs.

- *openUser(id: number): void*: Opens a new window which displays all detailed information(such as average of amount of Session pauses, amount of working time for a Session and how many times the User revised there decision) about the User with the id provided as parameter.
  @param id: The id of the User whose information should be displayed.

- *createUser(): boolean*: Creates an new User and adds it to the userList. Also sends the new User via the RESTService to the server.
  @return true: if the creation was successful. false: if an error occured during creation. A possible error could be that the name is already assigned. After an error has been detected, the admin is informed that this could be his error.

- *deleteUser(id: number): boolean*: Deletes a User by removing them from the userList. Also sends the information about the deletion via the RESTService to the server.

@param id: id which clearly identifies the User who should be deleted.
@return true: if deletion was successful. false: if an error occurd during deletion.

- *exists(id: number): boolean*: Checks the userList for a User with the id given as parameter.
  @param id: id which should be checked.
  @return true: if User with id exists. false: if user with id does not exist.

- *getUserName(id: number): boolean*: Searches through the userList for the name of User with the id.
  @param id: id to which the User name should be searched.
  @return the User name of the User with the id. Returns a empty String, if there is no such User.

- *deactivateUser(id: number): boolean*: deactivates the User with the given id.
  @param id The unique id of an User.
  @return true, if the User was successfully deactivated, else false.

4.2.15 Ladezeiten

Die Datenverwaltung und die Kommunikation mit der OcalAPI wird mittels des Backends implementiert. Die Anfragen an den Server können unter Umständen in längeren Ladezeiten resultieren. Um den User von Ladezeiten zu informieren, wird ihm bei jeder Anfrage mittels des RESTService ein Ladebalken angezeigt.
Hierfür wird ein NgProcess Module verwendet, welches die oben genannten Funktionalität bietet. Da jede Seite eine Menüleiste zur Navigation benötigt, wird der Ladebalken hierüber von den einzelnen Components angesprochen.

4.3 View

In der View wird definiert, wie die Components aus dem Model angezeigt werden. Dies wird in .html- und .css-Dateien beschrieben. In den .html-Dateien wird über Platzhalterkomponenten festgelegt an welcher Stelle die einzelnen Bestandteile (z.B. Diagramme) der Components angezeigt werden sollen. Die .css-Dateien spezifizieren das Aussehen der Bestandteile.

4.4 Controller

Der Controller besteht aus Services und dem Routing und kümmert sich so um die Kommunikation zwischen den einzelnen Komponenten. Services bieten Funktionalitäten die z.T. von mehreren Komponenten benötigt werden. Dadurch wird die Komplexität der Angularkomponenten reduziert. Diese sollten nicht die Daten mit denen sie arbeiten selbst speichern, sondern diese ausschließlich zur Visualisierung verarbeiten. Services helfen dabei einen einfachen Austausch zwischen verschiedenen Klassen herzustellen, bzw. die Schnittstelle zwischen Server und Client zu modellieren.

4.4.1 Routing

Der zentrale Bestandteil des Controllers in Angular bildet die Klasse Routes. Ein Objekt dieser Klasse wird über Angular als Singleton gehalten. Initialisiert wird das Objekt durch eine Liste an JSON-Objekten der Form:

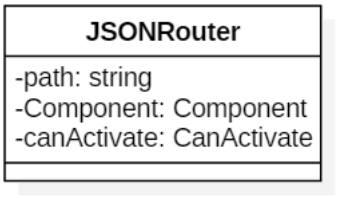


Abbildung 51: UML-Diagramm des JSONRouters

Das letzte Attribut canActivate ist hierbei optional. Diesem wird, wenn es gesetzt ist, ein Objekt des Interfaces IsActivate übergeben, welches überprüft ob der Nutzer die url öffnen darf.

Das Attribut path gibt an über welche URL das Attribut Component erreichbar ist. Für diese Objekte gilt:

- path: datasets, component: AdminDatasetOverviewComponent, canActivate: AdminGuard
- path: adminmain, component: AdminMainOverviewComponent, canActivate: AdminGuard
- path: session, component: AdminSessionOverviewComponent, canActivate: AdminGuard
- path: setup/create, component: AdminSetupCreateComponent, canActivate: AdminGuard
- path: setup/detail/:id, component: AdminSetupDetailComponent, canActivate: AdminGuard
- path: setup, component: AdminSetupOverviewComponent, canActivate: AdminGuard
- path: user/create, component: AdminUserCreateComponent, canActivate: AdminGuard
- path: user, component: AdminUserManagementComponent, canActivate: AdminGuard
- path: usermain, component: UserMainOverviewComponent, canActivate: UserGuard

- path: iterate/:id, component: UserIterationComponent, canActivate: UserGuard
- path: login, component: LoginComponent
- path: session/detail/:id, component: SessionDetailComponent

In den folgenden Diagrammen werden die Routingpfade zwischen den einzelnen Komponenten beschrieben.
Jeder Durchlauf startet mit dem Login. Nach dem Login kommt der Nutzer auf die erste Seite. Die Pfeile signalisieren wie die Route durch die Applikation weitergeführt werden könnte. Im ersten Diagramm werden die Routing-Pfade des Users und im zweiten die des Admins visualisiert.

User-Routing

Jede Nutzung des Programms startet auf der Loginseite. Nach der Eingabe des Nutzernamens wird der User auf die Seite UserMainComponent weitergeleitet. Hier sieht er eine Übersicht aller für ihn wichtigen Informationen (siehe: UserMainComponent). Von dieser Seite kann er über einen Button auf die Sessionübersicht kommen. In dieser hat er die Möglichkeit entweder zur SessionDetailComponent oder zur UserIterationComponent zu navigieren. Aus beiden Komponenenten heraus kann er wieder zurück zur Sessionübersicht gelangen.

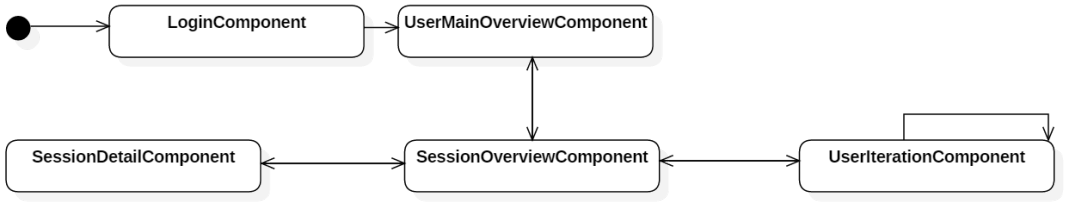


Abbildung 52: Aktivitätdiagramm zum User-Routing

Admin-Routing

Auch die Navigation des Admins startet auf der Loginseite auf welcher er sich als Admin einloggt. Nach dem Login wird er auf die AdminDatasetOverviewComponent weitergeleitet. Von dieser hat er die Möglichkeit sich die Übersicht über die

- Sessions (AdminSessionOverviewComponent),
- Setups (AdminSetupOverviewComponent),
- Datensätze (AdminDatasetOverviewComponent) und
- Nutzer (AdminUserCreateComponent)

anzeigen zu lassen. Von diesen Komponenten besteht auch die Möglichkeit zurück zu navigieren. Desweiteren kann er über die AdminSessionOverviewComponent zu der SessionDetailComponent hin und zurück steuern. Das gleiche gilt auch zwischen der AdminUserManagementComponent und AdminUserCreateComponent, ebenfalls zwischen der AdminSetupOverviewComponent und der AdminSetupDetailComponent. Desweiteren kann man über die AdminSetupOverviewComponent zu der AdminSetupCreateComponent navigieren. Von dieser aus gelangt man nach erfolgreichem Anlegen eines neuen Setups auf die AdminSetupDetailComponent.

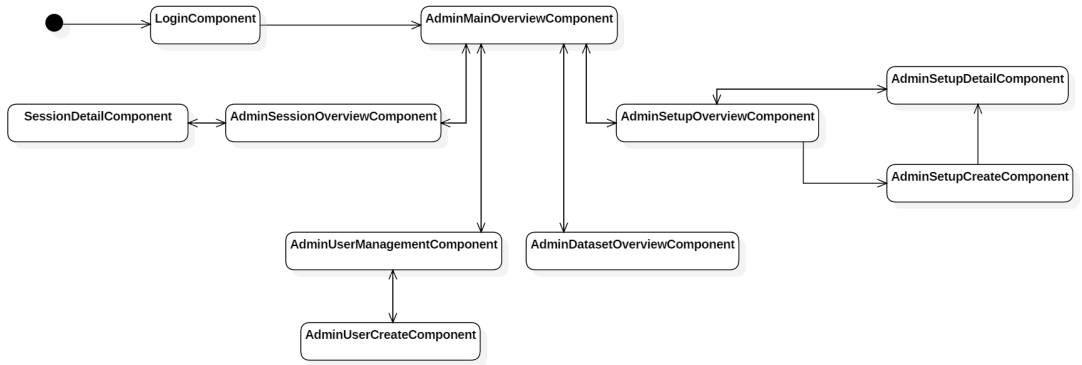


Abbildung 53: Aktivitätdiagramm zum Admin-Routing

4.4.2 RESTService

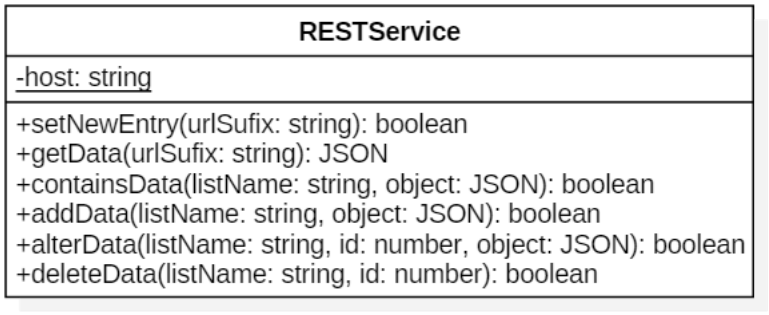


Abbildung 54: UML-Diagramm des REST-Service

Documentation:

Manages the interaction between server and client. It is responsible for addressing the REST (Representational State Transfer) API to perform CRUD (create, read, update, delete) requests.

Attributes:

- *host: string*: The host address through which the REST API is being approached.

Methods:

- *setNewEntry(urlSufix: string): boolean*: Creates a new resource on the server. Implements the create request of CRUD.
  @param urlSufix the created url endpoint which is being concatenated to the end of the provited host string.
  @return true if the creation was successful, false if not.

- *getData(urlSufix: string): JSON*: Returns the data provided at the requested url

endpoint. Implements the read request.
@param urlSufix the name of the requested list at the provided url endpoint.
@return the JSON object which contains the requested data.

- *containsData(listName: string, object: JSON): boolean*: Checks if the entered data is existing.
  @param listName the list to be checked. Meaning the url sufix which is being concatenated to the host address to access the data.
  @param object the JSON object to be checked.
  @return true if the list is existing, false if not.

- *addData(listName: string, object: JSON): boolean*: Adds data to an existing url endpoint. Implements the post request.
  @param listName the list where the data is to be added.
  @param object the JSON object which is being added.
  @return true if post was successful, false if not.

- *alterData(listName: string, id: number, object: JSON): boolean*: Updates an existing datapoint. Implements the update request.
  @param listName the list where the data is being updated.
  @param id the unique id of the datapoint in the respective list.
  @param object the object which contains the update.
  @return true if update was successful, false if not.

- *deleteDate(listName: string, object: JSON): boolean*: Deletes a datapoint from the provided list. Implements the delete request.
  @param listName the list where the data is to be deleted. Meaning the url sufix which is being concatenated to the host address to access the data.
  @param the JSON object of the data to be deleted.

@return true if removal was successful, false if not.

4.4.3 PersonService

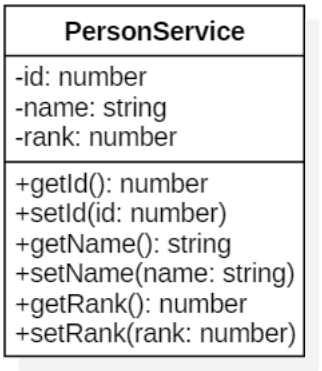


Abbildung 55: UML-Diagramm des Person-Service

Documentation:

Manages the Person which is currently logged into the system. Both Users and Admins are being held in here but can be distinguished.

Attributes:

- *id: number*: The internal unique id of the Person.
- *name: string*: The name of the Person.
- *rank: number*: The rank of a Person (see method documentation for details).

Methods:

- *getId(): number*: Getter for the Person's unique id.
  @return the id as an integer.

- *setId(id: number): void*: Setter for the Person's unique id.
  @param number the unique id.

- *getName(): String*: Getter for the Person's name.
  @return the name represented as a string.

- *setName(name: string): void*: Setter for the Person's name.
  @param name the name as a string.

- *getRank(): number*: Getter for the Person's rank. The rank 0 implicates an Admin, and rank 10 implicates a User. Several more ranks can be added for sophisticated permission modes (e.g. if there should be a new Person which has some rights of the Admin and some of the User).
  @return the rank of the Person.

- *setRank(rank: number): void*: Setter for the Person's rank. The rank 0 implicates an Admin, and rank 10 implicates a User. Several more ranks can be added for sophisticated permission modes (e.g. if there should be a new Person which has some rights of the Admin and some of the User).
  @param rank the new rank of the Person.

4.4.4 AdminGuard

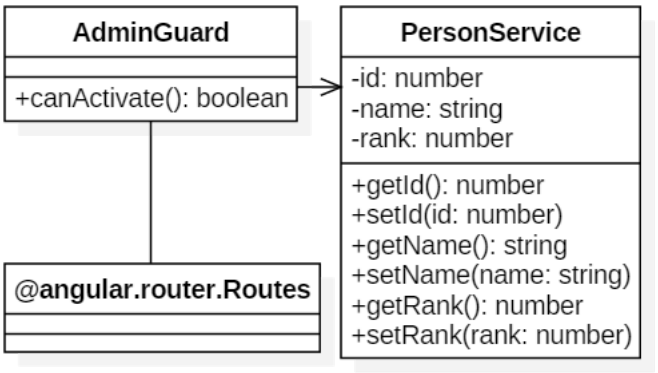


Abbildung 56: UML-Diagramm der AdminGuard-Klasse

Documentation:

The class implements the interface CanActivate. It provides a method that checks if an Admin is authorized to open an url. If this is not the case, the Admin will be redirected.

Methods:

- *canActivate(): boolean*: Determines whether or not an Admin is authorized to view the respective page
  @return true if the Admin is authorized, false if not.

4.4.5 UserGuard

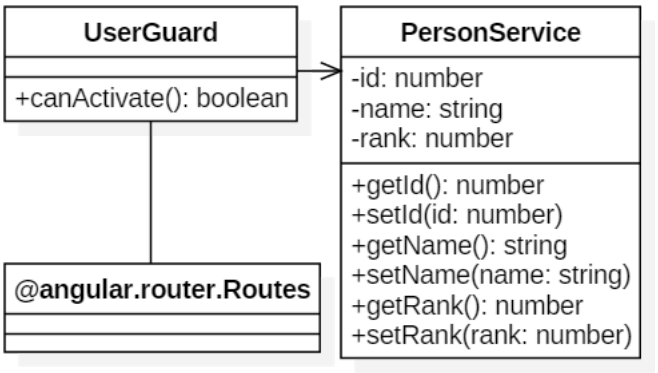


Abbildung 57: UML-Diagramm der UserGuard-Klasse

Documentation:

The class implements the interface CanActivate. It provides a method that checks if an User is authorized to open an url. If this is not the case, the User will be redirected.

Methods:

- *canActivate(): boolean*: Determines whether or not an User is authorized to view the respective page
  @return true if the User is authorized, false if not.

4.4.6 JSONHandlerService

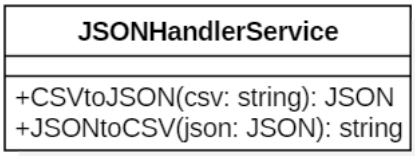


Abbildung 58: UML-Diagramm des JSONHandler-Service

Documentation:

Manages the convertion of file formats.

Methods:

- *CSVtoJSON(csv: string): JSON*: Converts a CSV (Comma-separated values) string to a JSON (JavaScript Object Notation) object.
  @param csv the string in csv pattern.
  @return the converted JSON object.

- *JSONtoCSV(json: JSON): string*: Converts a JSON (JavaScript Object Notation) object to a CSV (Comma-separated values) string.
  @param json the JSON object.
  @return the converted CSV string.

4.4.7 PlotService

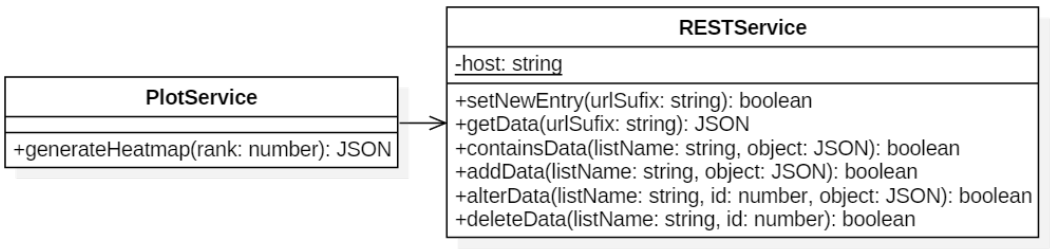


Abbildung 59: UML-Diagramm des Plot-Service

Documentation:

Manages the plotting of graphics in Sessions to visualize feature data.

- *generateHeatmap(rank: number): JSON*: Generates a heatmap using plotly.js. The plotted subspace depends on the given rank.

  @param rank The rank of the subspace which should be plottet. @return an object representing the heatmap

4.4.8 StatisticsService

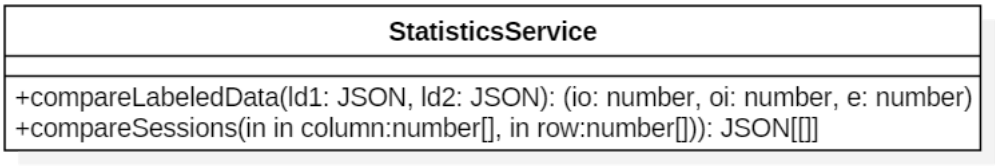


Abbildung 60: UML-Diagramm des Statistics-Service

Documentation:

Manages the computation of statistics. These are used to compare Sessions and to visiualize the Session summary.

Methods:

- *compareLabledDate(id1: JSON, id2: JSON): (io: number, oi: number, e: number)*:
  Compares two Sessions on behalf of their labels.
  @param ld1 the labeled data in Session1.
  @param ld2 the labeled data in Session2.
  @return a result vector.
  io: meaning the number of evaluations inlier to outlier.
  oi: meaning the number of evaluations outlier to inlier.
  e: meaning equal evaluations.

- *compareSessions(column: number[], row: number[]): JSON[[]]*: Compares two Sessions of one Setup to each other. The Sessions are being visualized in a matrix. By selecting a row and column a Session at the respective row and another one at the respective column are being compared.
  @param column the column in the compare matrix.
  @param row the row in the compare matrix.

@return the result of the Session comparison.

4.4.9 CalcService

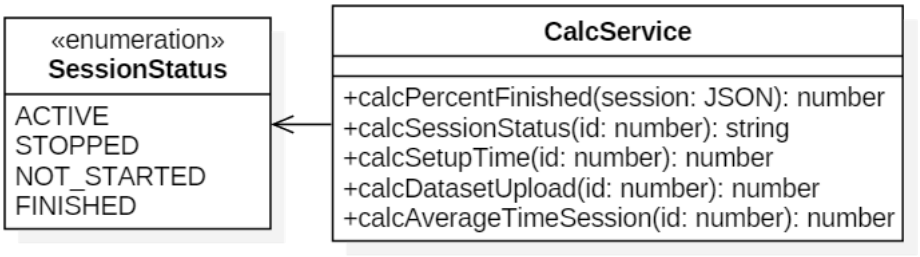


Abbildung 61: UML-Diagramm des Calc-Service

Documentation:

Manages the calculation of Data which is needed to visualize states of completion.

Methods:

- *calPercentFinished(session: JSON): number*: Calculates the finished part of the Session in percent.
  @param session the Session to be calculated.
  @return the percentage, meaning a value between 0 and 100.

- *calcSessionStatus(id: number): String*: Calculates the current Session status, which can be one of the following provided in SessionStatus.
  @param id the unique Session id.
  @return the Session status.

- *calcSetupTime(id: number): number*: Calculates the time since Setup creation.
  @param id the unique Setup id.

@return the time in seconds.

- *calcDatasetUpload(id: number): number*: Calculates the current upload status of the Dataset shown in percent.
  @param id the unique Dataset id.
  @return the percentage, meaning a value between 0 and 100.

- *calcAverageTimeSession(id: number): number*: Calculates the average time needed by a User to complete a Session.
  @param the unique Setup id, to which the Session belongs to.
  @return the average time in seconds.

4.4.10 SessionStatus

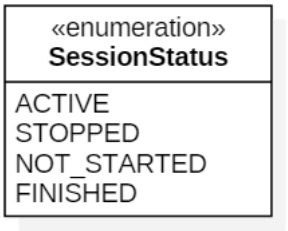


Abbildung 62: UML-Diagramm der SessionStatus Enumeration

Documentation:

Enumeration which provides the possible states of a Session.

Elements:

- *ACTIVE*

- *STOPPED*
- *NOT_STARTED*
- *FINISHED*

4.4.11 ExportService

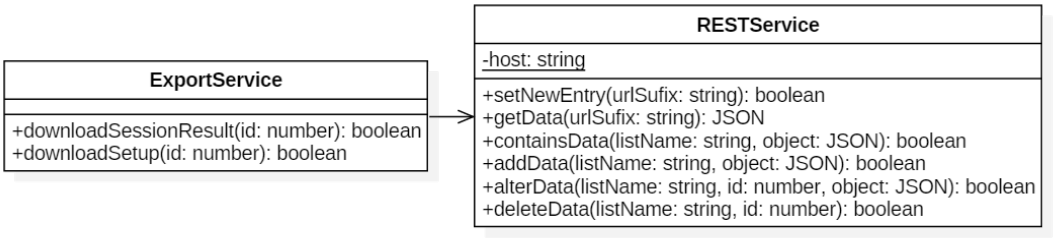


Abbildung 63: UML-Diagramm des Export-Service

Documentation:

Manages the export of the results.

Methods:

- *downloadSessionResult(id: number): boolean*: exports the results of the Session of the given id.
  @param The id of the Session to export.
  @return true, if the export was successfull, else false.

- *downloadSetup(id: number): boolean*: exports the results of the Session with the given id.
  @param id The id of the Setup to export.

@return true, if the export was successfull, else false.

5 Interaktionsabläufe

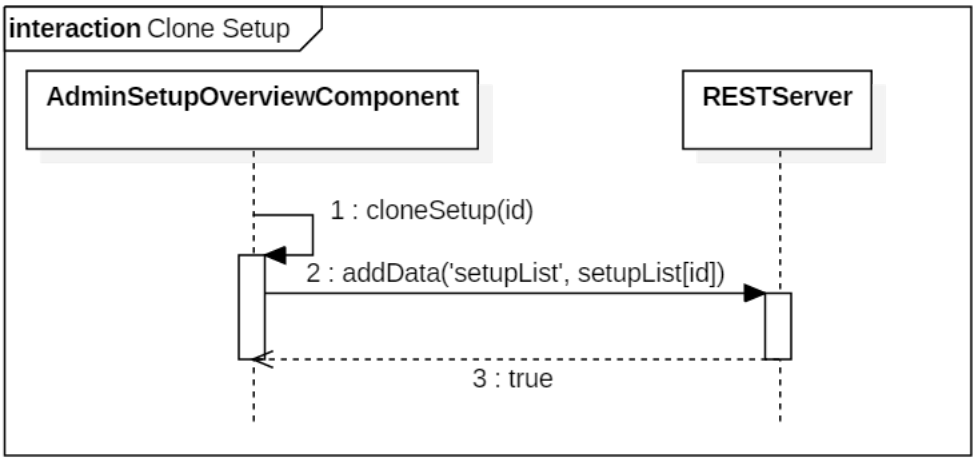


Abbildung 64: Sequenzdiagramm zum Klonen von Setups

Beschreibung: Sequenzdiagramm zur Darstellung eines Klonprozesses von Setups. In der AdminSetupOverviewComponent wird die Methode „cloneSetup(id) “aufgerufen. Die übergebene ID ist die ID des Setups welches geklont werden soll. Daraufhin wird innerhalb der Methode die Methode „add(‚setupList ‘, setupList[id]) vom RESTSerivce aufgerufen. Die übergebene Zeichenkette bezeichnet die Tabelle in welcher das Objekt hinzugefügt wird. Der zweite Parameter ist das JSON-Objekt des Setups mit der ID id, welches neu hinzugefügt werden soll. Nach erfolgreicher Ausführung wird der Wahrheitswert True zurückgegeben.

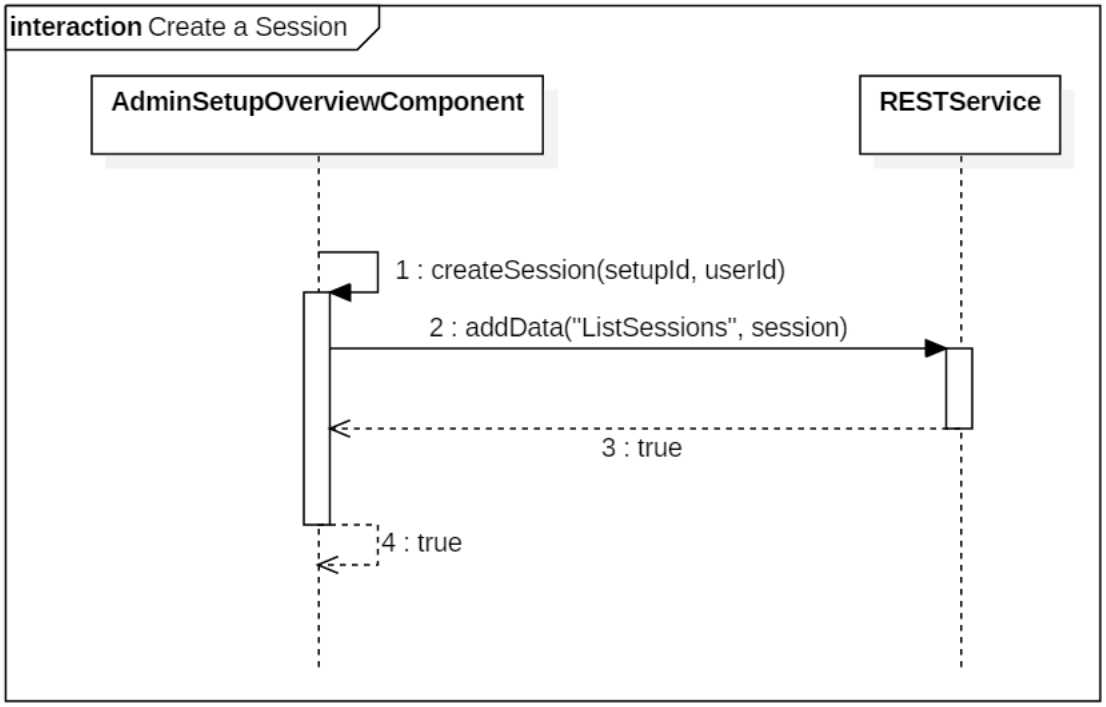


Abbildung 65: Sequenzdiagramm zum Erstellen von Sessions

Beschreibung: Sequenzdiagramm zur Darstellung der Interaktion der Komponenten bei der Erstellung einer neuen Session. Sobald der Admin auf „create session“klickt, wird die Methode createSession mit den Parametern setupId und userId ausgeführt. Diese Parameter stellen dabei die eindeutigen IDs des Setups und des Users dar, welche der Admin ausgewählt hat. Daraufhin wird diese neu erstellte Session mittels des RESTService an den Server geschickt. Dieser quittiert den Erhalt der JSON-Datei. Daraufhin beendet die Methode createSession mit der Rückgabe von true.

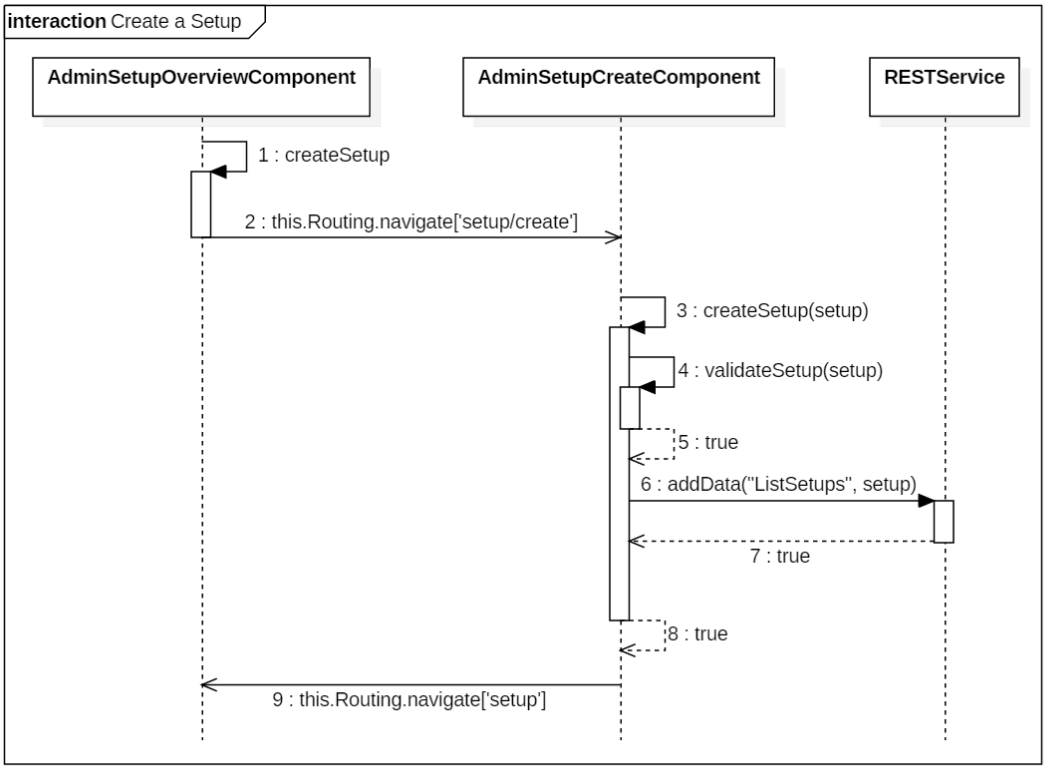


Abbildung 66: Sequenzdiagramm zum Erstellen von Setups

Beschreibung: Sequenzdiagramm zur Darstellung der Interaktion der Komponenten bei der Erstellung eines neuen Setups. Klickt der Admin auf den Knopf „create setup“, so wird die Methode createSetup aufgerufen. Diese löst lediglich das Routing aus, wodurch zur SetupCreate-Ansicht gewechselt wird. Hier hat der Admin die Möglichkeit die Parameter des Setups einzustellen. Klickt er auf „complete Creation“, wird die Methode createSetup mit einer JSON setup, welche aus den Eingaben des Admins generiert wurde, aufgerufen. Diese ruft die Methode validateSetup auf, um die Gültigkeit der Eingaben zu überprüfen. Gibt diese true zurück, so wird die JSON des neuen Setups an den Server via RESTService gesendet. Sobald der Server den Eingang der Daten quittiert, gibt der RESTService true zurück. Daraufhin gibt die Methode createSetup true zurück und es wird das Routing zurück zur Setupübersicht ausgelöst.

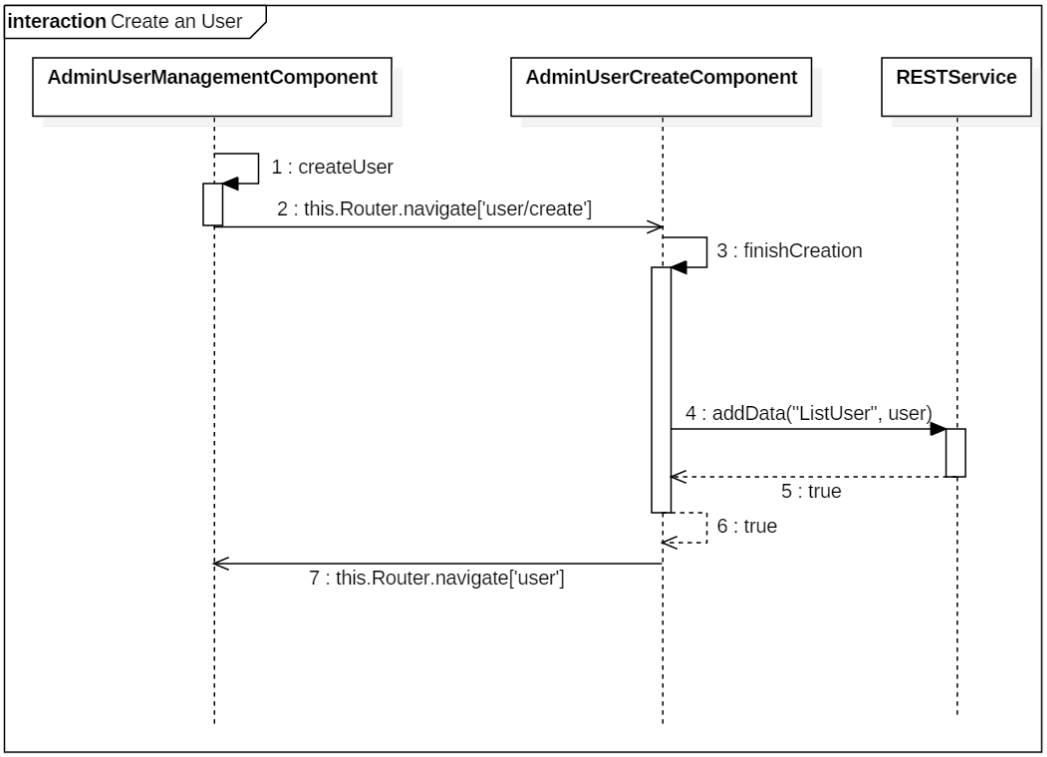


Abbildung 67: Sequenzdiagramm zum Erstellen von Usern

Beschreibung: Sequenzdiagramm zur Darstellung der Interaktion der einzelnen Komponenten, wenn ein Admin einen neuen User erstellt und direkt einen gültigen Namen eingibt. Klickt der Admin auf den Knopf „add user“, so wird die Methode createUser ausgeführt. Diese initiiert lediglich das Routing, sodass zur User-Erstellung gewechselt wird. Hier hat der Admin die Möglichkeit in Textfelder die Daten des Users einzutragen. Klickt er auf „complete creation“wird die Methode finishCreation ausgeführt. Diese sendet den neu erstellten User als JSON mit Hilfe des RESTService an den Server. Dieser überprüft die Attribute des Users auf Richtigkeit und quittiert den Erhalt indem er true zurückgibt. Daraufhin gibt auch finishCreation true zurück und es wird das Routing zurück zur Benutzerverwaltung ausgelöst.

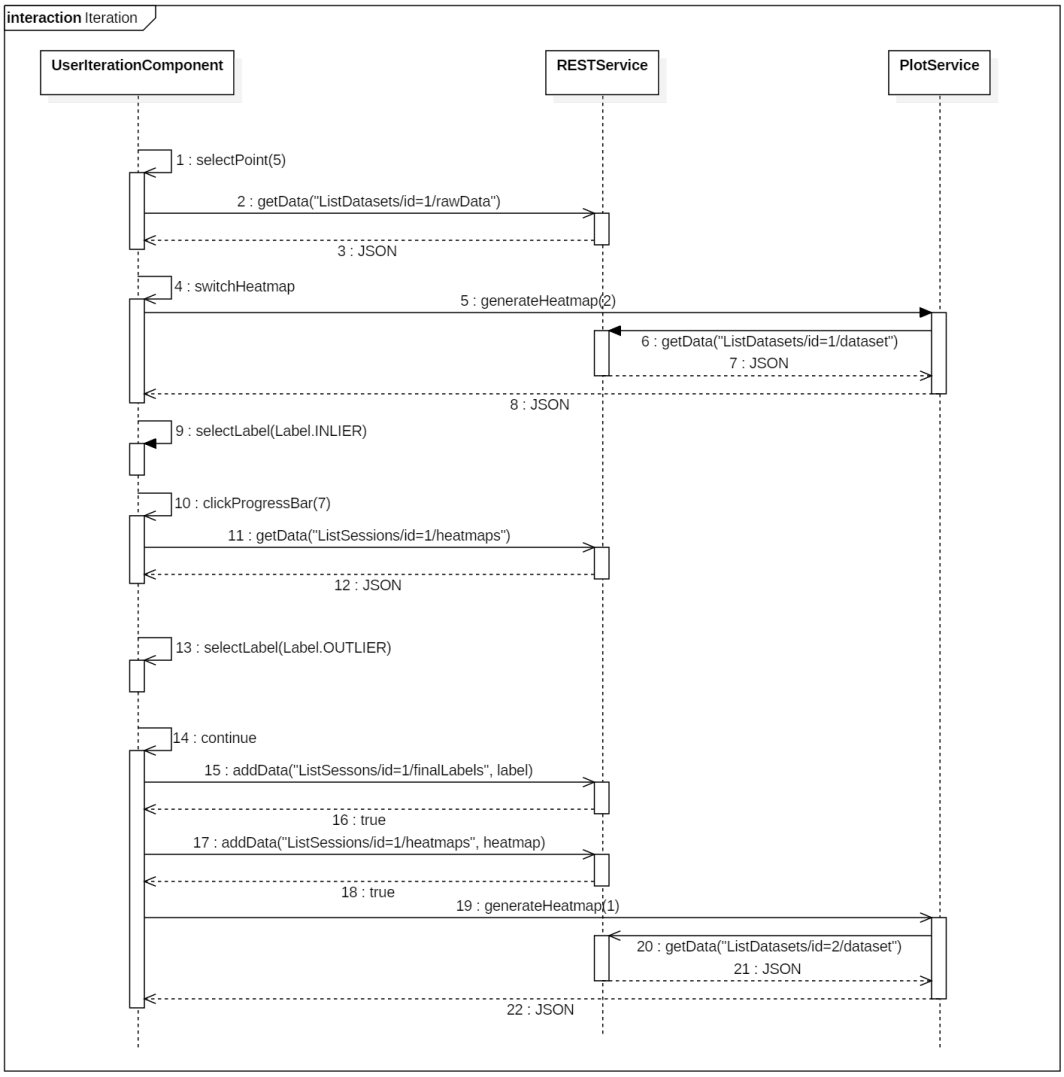


Abbildung 68: Sequenzdiagramm zu einer einzelnen Iteration

Beschreibung: Dieses Sequenzdiagramm modelliert den Ablauf einer Iteration in einer Session. Der User wählt einen Punkt auf der aktuellen Heatmap aus (Feedbackmodus: user). Zu diesem Punkt werden ihm Rohdaten angezeigt. Anschließend entscheidet der User sich, eine zweite Heatmap anzuschauen. Der vorher gewählte Punkt bleibt ausgewählt. Aufgrund dieser Visualiserung wählt der User das Label Inlier. Mit einem

Klick auf den Fortschittsbalken lässt er sich eine Vorschau der Heatmap aus Iteration 7 anzeigen. Anschließend wählt er das Label Outlier. Mit Klick auf continue wechselt er zur nächsten Iteration in der ihm eine neue Heatmap angezeigt wird.

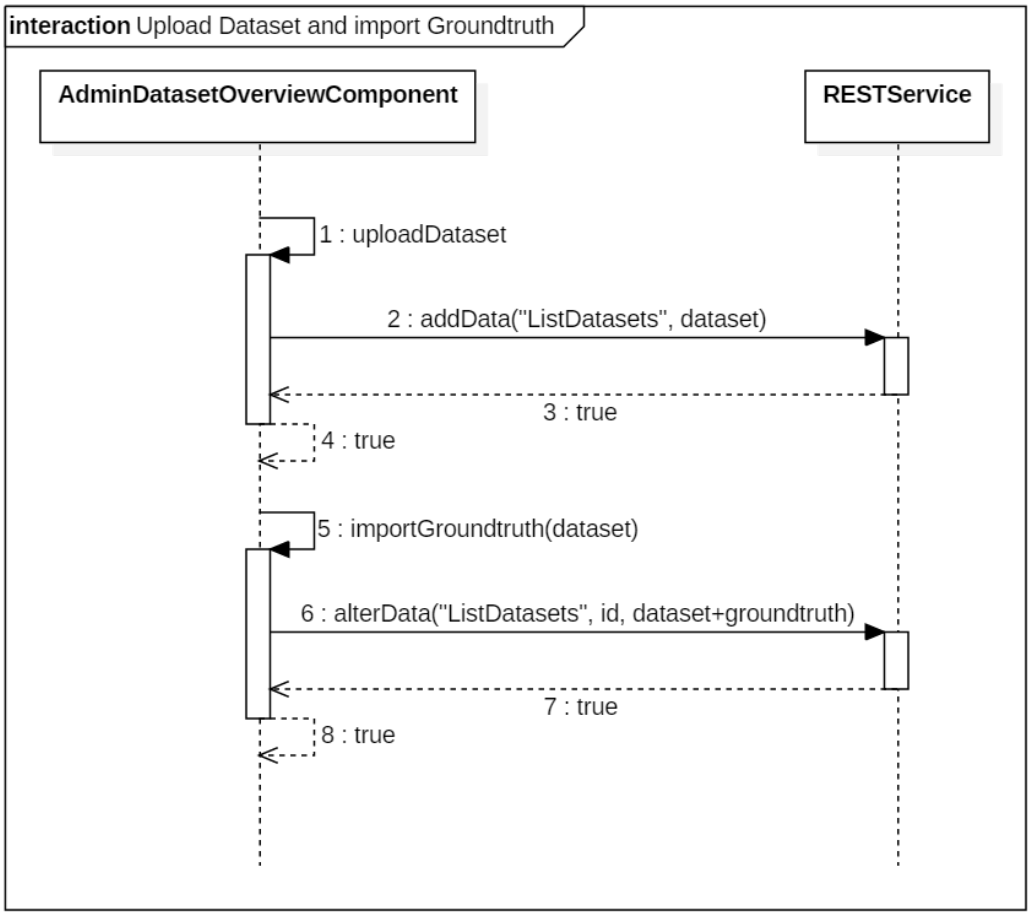


Abbildung 69: Sequenzdiagramm zum Importieren von Datensätzen

Beschreibung: Sequenzdiagramm zur Darstellung der Interaktionen der Komponenten beim hochladen eines Datensatzes und importieren einer Groundtruth. Sobald der Admin den Knopf „Upload Dataset“anklickt wird die Methode uploadDataset ausgeführt. Hierdurch wird ein File-Explorer geöffnet. Sobald der Admin eine Datei ausgewählt hat, wird diese mit Hilfe des RESTService an den Server zur Speicherung geschickt. Dieser

quittiert das Eintreffen des Datensatzes. Daraufhin gibt die Methode uploadDataset true zurück. Klickt der Admin nun auf den Knopf „import Groundtrut“, so wird die Methode importGroundtruth mit dem Datensatz als Parameter, dessen „upload Groundtrut“-Knopf der Admin angeklickt hat, ausgeführt. Hierdurch wird mittels des RESTService die JSON des bereits abgespeicherten Datensatzes verändert. In diese JSON wird die Groundtruth hinzugefügt. Die bereits gespeicherte Datensatz JSON wird nun mit der neuen überschrieben. War dies alles erfolgreich, so geben die Methoden jeweils true zurück.

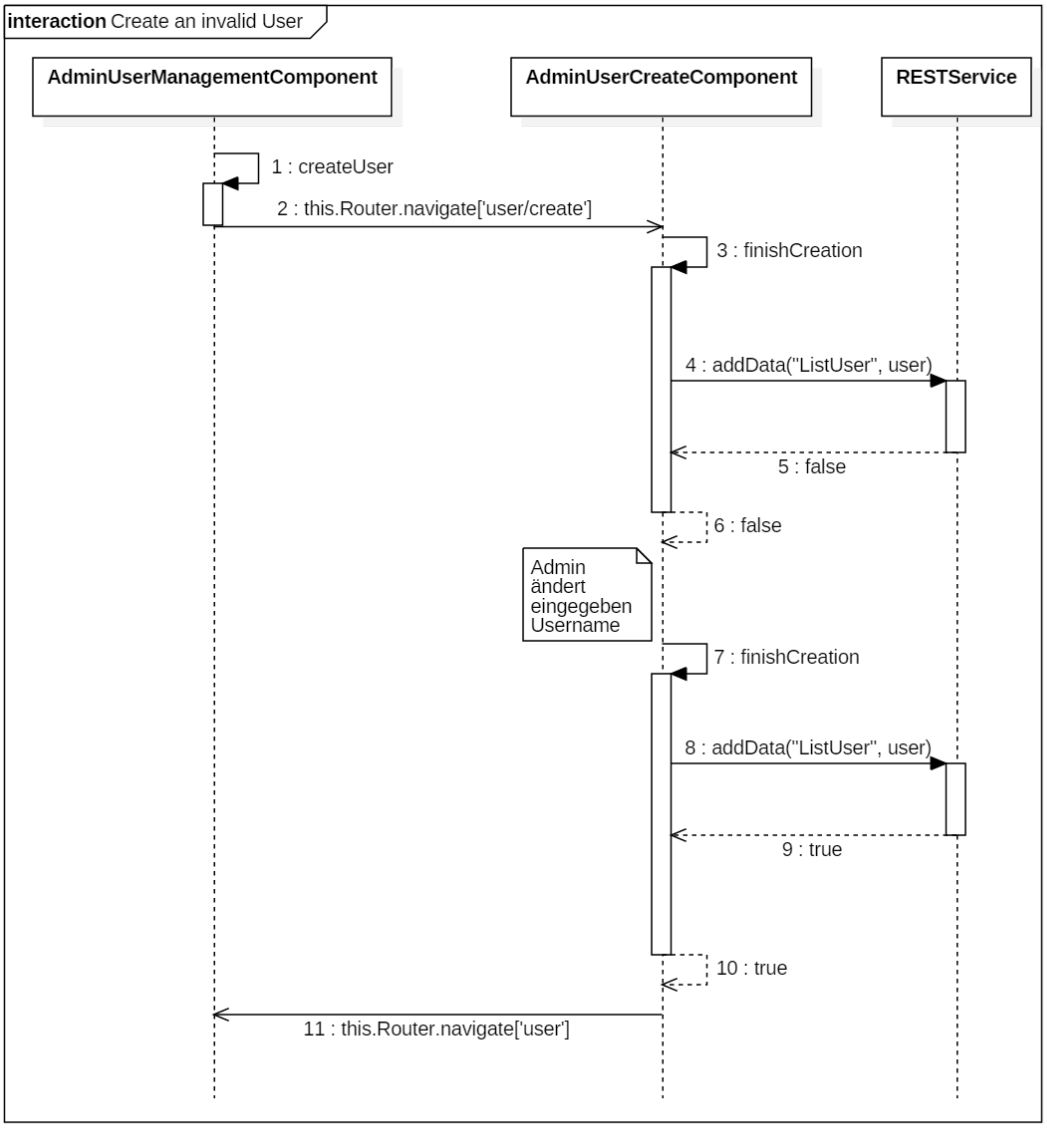


Abbildung 70: Sequenzdiagramm zu fehlschlagender Nutzererstellung

Beschreibung: Sequenzdiagramm zur Darstellung der Interaktion der einzelnen Komponenten, wenn ein Admin einen neuen User erstellt und zuerst einen ungültigen Namen, gefolgt von einem gültigen, eingibt. Klickt der Admin auf den Knopf „add user“, so wird die Methode createUser ausgeführt. Diese initiiert lediglich das Routing, sodass

zur User-Erstellung gewechselt wird. Hier hat der Admin die Möglichkeit in Textfelder die Daten des Users einzutragen. Klickt er auf „complete creation“wird die Methode finishCreation ausgeführt. Diese sendet die JSON mit den Informationen über den neuen User an den Server. Dieser überprüft die Daten und erkennt einen Fehler. Daraufhin gibt der RESTService false zurück und dem Admin wird eine entsprechende Meldung durch die Methode finishCreation angezeigt. Die Methode finishCreation gibt false zurück. Nun ändert der Admin den eingegebenen Namen und klickt erneut auf „complete creation“. Es wird erneut die finishCreation-Methode aufgerufen, diesmal verläuft die Namensüberprüfung des Servers positiv, wodurch die Methoden addData true zurückgibt. Die Daten über den neu angelegten User liegen nun auf dem Server. Daraufhin gibt auch finishCreation true zurück und es wird das Routing zurück zur Benutzerverwaltung ausgelöst.