

MDD NRW Schnittstellenbeschreibung

v1.2

Februar 2025

Contents

1	Autorisierung	2
2	GraphQL Endpunkt	4
2.1	Schema	4
2.2	Komplexe Attribute	5
2.3	Queries	6
2.3.1	Aufbau des Query types	7
2.3.2	Query Argumente	9
2.3.3	Filtern von Queries	10
2.3.4	Abfragen gebündelter Daten	11
2.3.5	Geofilter	13
2.3.6	Datenkatalog Meta-Daten	14
2.3.7	Postman Query Baum	16
2.3.8	GZIP Kompression	18
3	GBFS Endpunkt	19
3.1	Liste der Unterstützten GBFS Versionen	19
4	Datex-II Endpunkt	20
5	Geoserver	21
5.1	Authentifizierung	21
5.2	Endpunkte	21
6	Direkte Schnittstellen	23
6.1	DELFI Landeshintergrundsystem	23
6.2	goFLUX Mobility GmbH	24

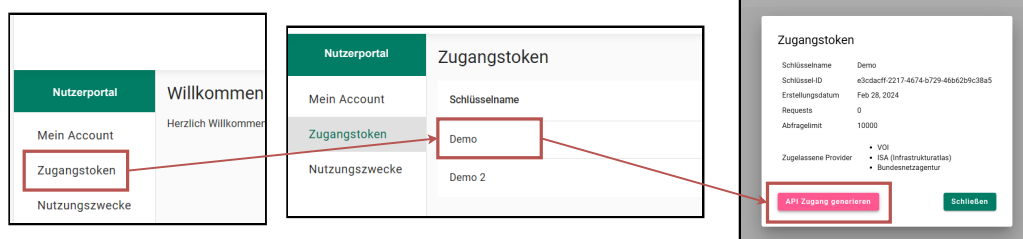
1 Autorisierung

In der MDD NRW sind jedem Nutzer ein oder mehrere Clienten zugeordnet. Clienten stellen voneinander unabhängige Zugänge zu den Daten in der MDD NRW dar. Jeder Client verfügt über einen Zugangstoken (in Form eines API-Keys) mit dem die Daten der MDD NRW abgefragt werden können. An den Clienten sind auch Einschränkungen, wie das Request Limit und Zugriffsrechte, gebunden. Die Anzahl der Clienten und die jeweiligen Zugriffsrechte werden **nicht** vom User selbst verwaltet. Für eine Erweiterung der Zugriffsrechte oder Erhöhung des Request Limits muss der Support kontaktiert werden.

!Wichtig!

Jedem Clienten sind eine oder mehrere Funktionen zugeordnet, aus denen sich die ebenfalls im Nutzerportal einsehbaren Nutzungszwecke ableiten. Es ist die Pflicht jedes Nutzers der MDD NRW diese Nutzungszwecke einzuhalten.

Die Autorisierung selbst erfolgt mittels API-Key. API-Keys können von registrierten Nutzern über das Nutzerportal für jeden Clienten selbst generiert und erneuert werden.



Zur Autorisierung gegenüber einer der Schnittstellen der MDD NRW muss der API-Key im Header mit dem **key x-api-key** mitgegeben werden.

Beispiel:

Abfrage des GBFS Auto-discovery Files der Version 2.3 des Providers mit ID 1 mit dem API-Key <api-key>:

```
curl --location  
'https://mdd.gorheinland.com/gbfs-output/1/2.3/gbfs.json' \  
--header 'x-api-key: <api-key>'
```

Hinweis

API-Keys werden nicht zentral in der MDD NRW gespeichert. Sie werden beim generieren einmalig dem Nutzer angezeigt und müssen vom Nutzer eigenständig gesichert werden. Geht der aktuelle API-Key verloren kann dieser nicht wiederhergestellt werden und der Nutzer muss sich im Nutzerportal der MDD NRW einen neuen API-Key generieren.

2 GraphQL Endpunkt

GraphQL ist eine Abfragesprache, die es erlaubt mittels einer **query** nur diejenigen Daten abzufragen, die man braucht. Das GraphQL-Schema (siehe Kapitel [Schema](#)) definiert welche Abfragen an die Schnittstelle grundlegend möglich sind. Die genaue Erstellung der **query** ist dem Nutzer überlassen.

In GraphQL gibt es zwei Arten von **Attributen**, primitiv und komplex. Ein Attribut ist **primitiv**, wenn es selber keine Attribute hat. Primitive Attribute haben primitive Datentypen wie

- String
- BigInteger
- Boolean
- Float
- DateTime
- ...

Komplexe Attribute werden im Schema beschrieben und bestehen selbst aus primitiven und komplexen Attributen. Durch das verschachteln komplexer Attribute werden Verknüpfungen zwischen Daten hergestellt, welche flexible Abfragen über mehrere Attribute hinweg erlauben. Weitere Informationen zum Abfragen komplexer Attribute finden sich im Kapitel [Queries](#).

2.1 Schema

Das GraphQL-Schema kann über einen **GET**-Request an folgende URL abfragt werden:

```
https://mdd.gorheinland.com/graphql/schema.graphql
```

Die Response ist ein [GraphQL-Schema](#). Das GraphQL-Schema beschreibt welche Abfragen über die MDD NRW GraphQL Schnittstelle möglich sind. Es enthält folgende grundlegenden Keywords:

- **type**: Komplexe Datentypen
- **enum**: Auflistungen
- **input**: Argumente

Ein **type** definiert komplexe Datentypen. Komplexe Datentypen sind in GraphQL vielseitig einsetzbar. Neben abfragbaren Daten werden hierüber auch die Funktionen zum Abfragen der Daten definiert, siehe Kapitel [Queries](#). Mehr zu komplexen Datentypen zum Abfragen von Daten findet sich in Kapitel [Komplexe Attribute](#).

Ein **enum** ist eine vordefinierte Liste von Werten. Hier z.B. FeatureType in gekürzter Darstellung:

```

1 enum FeatureType {
2   ACCESSIBLE_PARTIAL_PAVED_SURFACE
3   ACCESSIBLE_PAVED_SURFACE
4   ACCESSIBLE_UNPAVED_SURFACE
5   ACCESS_AT_GROUND_LEVEL
6   ACCESS_CURB_STEP
7   ACCESS_DOOR
8   ...
9 }
```

Das Keyword **input** weist darauf hin, dass dieser Typ als Input für Queries verwendet wird, wie beispielsweise **Filter**. Der Input-Typ **ParkingFacilityFilter** wird zum Beispiel im **query** `listParkingFacilities` verwendet. Mehr zum Thema Filtern findet sich in Kapitel [Filtern von Queries](#).

Hinweis

Die hier gezeigten Queries dienen nur zur Veranschaulichung und basieren nicht unbedingt auf der aktuellsten Version des GraphQL-Schemas.

Wir empfehlen immer das aktuelle GraphQL-Schema über die Schnittstelle abzufragen und auf dessen Basis Queries zu bauen. Insbesondere während der Beta Phase der MDD NRW sind Änderungen am Schema nicht ausgeschlossen.

2.2 Komplexe Attribute

Ein komplexes Attribut wird mit dem Keyword **type** initialisiert. Alle verfügbaren komplexen Attribute sind im GraphQL-Schema beschrieben und sind wie folgt aufgebaut:

```

1 "DESCRIPTION"
2 type TYPENAME {
3   FIELD_NAME : TYPE
4   OTHER_FIELD_NAME: DATA_TYPE
5   ...
6 }
```

TYPENAME ist der Name des komplexen Attributs. Die Attribute hinter den Feldnamen zeigen an, welche Attribute in einer **query** abgefragt werden können.

Es ist also möglich anhand des Schemas auszulesen welche Informationen über einen im GraphQL-Schema definierten **type** abgefragt werden können und mit welchen anderen Datentypen die Informationen verknüpft sind.

Beispiel: Definition eines komplexen Datentypen

Der **type** `ParkingFacility` ist wie folgt im Schema definiert:

```

1 "parking facility, for example a bikebox"
2 type ParkingFacility {
3   address: String
4   available: Boolean
5   boxes: [Box]
6   capacities: [Capacity]
7   chargingStations: [ChargingStation]
8   description: String
9   "Id given by provider"
10  externalId: String!
11  externalType: String
12  features: [ParkingFacilityFeature]
13  feeDescription: String
14  id: BigInteger
15  "ISO-8601"
16  lastUpdated: DateTime
17  "latitude of current location of the parking facility"
18  lat: Float
19  "longitude of current location of the parking facility"
20  lon: Float
21  lowestPriceInCents: Int
22  name: String!
23  nextPublicTransportLocation: Location
24  openingTimesDescription: String
25  parkingType: ParkingType
26  postCode: String
27  provider: Provider!
28  tariffs: [Tariff]
29  "Get entry/entries for a certain key/s"
30  urls(key: [UrlKey]): [Entry_UrlKey_Url]
31 }

```

Es ist also möglich im komplexen Attribut `ParkingFacility` direkt primitive Attribute wie `lat`, `lon` und `id` abzufragen. Bei komplexen Attributen wie `"Box"` muss in der **query** angegeben werden welche Attribute von `"Box"` enthalten sein sollen.

Hinweis

Wenn ein Datenfeld in der **type** Definition mit eckigen Klammern `"[]"` umschlossen ist, wird bei Abfragen des Datenfeldes eine Liste zurück gegeben. Beispielsweise gibt `"boxes"` eine Liste von komplexen Datentypen der Entität `"Box"` zurück.

2.3 Queries

Um die GraphQL Schnittstelle anzusprechen, sendet man einen **POST Request** an:

`https://mdd.gorheinland.com/graphql`

Eine **query** bestimmt welche Attribute abgefragt werden. Wenn ein Feld selbst ein komplexes Attribut zurück gibt muss festgelegt werden welche Attribute man von diesem Attribut benötigt, solange bis alle Anfrage-Äste auf einem primitiven Attribut enden.

Dieser Baum, den man in einem Query aufspannt muss also **immer auf einem primitiven Attribut enden**. Es können alle Attribute, welche im **type Query** aufgelistet sind, abgefragt werden.

Die **query** selbst wird im **body** des Requests mitgeschickt. Dabei können den Queries Argumente übergeben werden und der Rückgabebetyp ist ebenfalls im Schema angegeben, wie in Kapitel **Komplexe Attribute** beschrieben.

2.3.1 Aufbau des Query types

Welche Anfragen an die Schnittstelle möglich sind ist ebenfalls im **GraphQL-Schema** als **type** definiert.

```

1 "Query root"
2 type Query {
3   vehicleTypeById(id: BigInt!): VehicleType
4   alertById(id: BigInt!): Alert
5   calendarById(id: BigInt!): Calendar
6   geofencingZoneById(id: BigInt!): GeofencingZone
7   "get Rides by goFlux"
8   getRides(arrivalLat: Float!, arrivalLng: Float!, arrivalRadius: Int!, day: Int!, departureLat: Float!
9     !, departureLng: Float!, departureRadius: Int!, hour: Int!, minute: Int!, month: Int!,
10     timeDeltaSeconds: Int!, year: Int!): String
11   listAlerts(limit: Int, offset: Int, providerId: BigInt!): [Alert]
12   listCalendars(limit: Int, offset: Int, providerId: BigInt!): [Calendar]
13   listGeofencingZones(limit: Int, offset: Int, providerId: BigInt!): [GeofencingZone]
14   listLocations(limit: Int, offset: Int, providerId: BigInt!): [Location]
15   listParkingFacilities(filter: ParkingFacilityFilter, limit: Int, offset: Int, order: Order, orderBy:
16     ParkingFacilityOrderBy): [ParkingFacility]
17   listPricingPlans(limit: Int, offset: Int, providerId: BigInt!): [PricingPlan]
18   listProviders(limit: Int, offset: Int): [Provider]
19   listRawFeedData(limit: Int, offset: Int, providerId: BigInt!): [RawFeedData]
20   listRegions(limit: Int, offset: Int, providerId: BigInt!): [Region]
21   listRentalHours(limit: Int, offset: Int, providerId: BigInt!): [RentalHour]
22   listSharingStations(limit: Int, offset: Int, providerId: BigInt!): [SharingStation]
23   listSystemInformation(limit: Int, offset: Int, providerId: BigInt!): [SystemInformation]
24   listVehicleTypes(limit: Int, offset: Int, providerId: BigInt!): [VehicleType]
25   listVehicles(limit: Int, offset: Int, providerId: BigInt!): [Vehicle]
26   locationById(id: BigInt!): Location
27   parkingFacilityById(id: BigInt!): ParkingFacility
28   pricingPlanById(id: BigInt!): PricingPlan
29   providerById(id: BigInt!): Provider
30   rawFeedDataById(id: BigInt!): RawFeedData
31   regionById(id: BigInt!): Region
32   rentalHourById(id: BigInt!): RentalHour
33   sharingStationById(id: BigInt!): SharingStation
34   systemInformationById(id: BigInt!): SystemInformation
35   vehicleById(id: BigInt!): Vehicle
36   "Welcome"
37   welcome(name: String = "World"): String
38 }

```

Alle im **type Query** definierten Funktionen lassen sich für Abfragen der Daten in der MDD NRW nutzen. Die Query

```
1 query ListProviders {  
2   listProviders {  
3     name  
4     id  
5   }  
6 }
```

beispielsweise gibt Namen und ID aller Provider zurück auf die man Zugriffsrechte hat.

Grundlegend lassen sich die so zur Verfügung gestellten Endpunkte in zwei Arten unterteilen.

listEntity,

welche eine Liste der angefragten Entität zurückgeben und

entityById,

welche die der ID zugehörige Entität zurückgeben.

Die **query entityById** benötigt dabei immer den Parameter "id". Zu dieser "id" muss eine Entität in der Datenbank existieren damit die Abfrage funktioniert.

Beispiel: entityById Query

```
1 query {  
2   ParkingFacilityById(id: 1) {  
3     address  
4     available  
5     boxes {  
6       capacity  
7     }  
8   }  
9 }
```

Diese Abfrage gibt alle Entitäten des types **ParkingFacility** zurück welche dem provider mit ID "1" zugeordnet sind.

Analog gibt die folgende Abfrage die gleichen Informationen für Entitäten des types **ParkingFacility** zurück, allerdings anstatt nur die Entitäten eines Providers eine Liste aller Entitäten, auf die man Zugriffsrechte hat.

```
1 query {  
2   listParkingFacility {  
3     address  
4     available  
5     boxes {  
6       capacity  
7     }  
8   }  
9 }
```

Hinweis

Es ist auch möglich bei Abfragen der Art `listEntity` die Ergebnisse zu verfeinern. Mehr dazu findet sich im Kapitel [Filtern von Queries](#).

2.3.2 Query Argumente

Einer Query können **Argumente** übergeben werden um die Response weiter zu verfeinern. Die verfügbaren Argumente können in den im `type Query` definierten Funktionen eingesehen werden. **Beispiel:** "`listParkingFacilities`" ist im Schema definiert als:

```
1 listParkingFacilities(filter: ParkingFacilityFilter,  
2                       limit: Int,  
3                       offset: Int,  
4                       order: Order,  
5                       orderBy: ParkingFacilityOrderBy)  
6                       : [ParkingFacility]
```

Die verfügbaren Argumente sind somit

- `filter`: Komplexe Filter, siehe Kapitel [Filtern von Queries](#)
- `limit` (Int): Maximale Anzahl zurückgegebener Entitäten. Es wird empfohlen bei jeder Abfrage ein Limit zu setzen, da das Abfragen von zu vielen Entitäten auf einmal sehr lange dauern kann. Das ist insbesondere beim Abfragen von GTFS-Daten über GraphQL relevant.
- `offset` (Int): Anzahl von Entitäten die in der Response übersprungen werden. Zusammen mit `limit` für **Pagination** geeignet.
- `order`: **ASC** für aufsteigend, **DESC** für absteigend.
- `orderBy`: Bestimmt, nach welchem Feld sortiert werden soll. Die verfügbaren Felder sind als `enum` der zugehörigen "typeOrderBy" im GraphQL-Schema definiert.

Beispiel: Query mit Argumenten

```
1 query {  
2   listParkingFacilities(limit: 3, offset: 2, orderBy: parkingType, order: ASC) {  
3     id  
4     name  
5     parkingType  
6     lastUpdated  
7     provider {  
8       id  
9     }  
10  }  
11 }
```

2.3.3 Filtern von Queries

Filter arbeiten auf den in `type Query` definierten Funktionen für die Datenabfrage. Im Beispiel von `"listParkingFacilities"` ist als Filter der `ParkingFacilityFilter` definiert. Filter sind, wie in Kapitel `Schema` beschrieben, im GraphQL-Schema als `input` definiert. Der `ParkingFacilityFilter` sieht wie folgt aus:

```

1 input ParkingFacilityFilter {
2   address: [StringFieldFilter]
3   available: EqualityFieldFilter_Boolean
4   coordinate: LocationFieldFilter
5   description: [StringFieldFilter]
6   externalId: EqualityFieldFilter_BigInteger
7   externalType: [StringFieldFilter]
8   lastUpdated: [DateFieldFilter]
9   lowestPriceInCents: [NumberFieldFilter_Int]
10  name: [StringFieldFilter]
11  nextPublicTransportLocationId: EqualityFieldFilter_BigInteger
12  nextPublicTransportLocationName: [StringFieldFilter]
13  parkingType: EqualityFieldFilter_ParkingType
14  postCode: [StringFieldFilter]
15  providerId: EqualityFieldFilter
16  providerName: [StringFieldFilter]
17 }

```

Im `ParkingFacilityFilter` sind für die meisten primitiven Attribute vom `type ParkingFacility` je nach Datentyp **Feld-Filter** definiert. Darüber hinaus gibt es einen Filter für ID und Name des Providers, also für primitive Attribute des komplexen Attributs `Provider`. Die Feld-Filter sind ebenfalls im GraphQL-Schema aufgelistet. Um nach `providerId` zu filtern wird hier der Feld-Filter `ProviderIdFilter` genutzt. Dieser ist definiert als:

```

1 input ProviderIdFilter {
2   operator: ContainedOperator
3   values: [BigInteger]
4 }

```

Feld-Filter bestehen immer aus einem `value` und je nach Filter über einen `operator`. Wenn es einen Operator gibt ist der ebenfalls im GraphQL-Schema als `enum` definiert. Operatoren geben an, wie die Einträge in "value" zum Filtern genutzt werden sollen. Hier wird der `ContainedOperator` verwendet, dieser ist definiert als:

```

1 enum ContainedOperator {
2   containedIn
3   notContainedIn
4 }

```

Um hier also nur ParkingFacilities abzufragen die den Providern mit ID "1", "2" und "3" zugeordnet sind muss der Filter wie folgt gesetzt werden:

```
1 query {  
2   listParkingFacilities(  
3     filter: { providerId: { values: [1, 2, 3], operator: containedIn}  
4   } {  
5     id  
6     name  
7     parkingType  
8     lastUpdated  
9     provider {  
10      id  
11    }  
12  }  
13 }
```

Es können beliebig viele Filter unter dem **filter** Keyword gesetzt und kombiniert werden. Multiple Filter werden mit Kommas getrennt und sind logisch als "und" verknüpft.

```
1 query {  
2   listParkingFacilities(  
3     filter: {  
4       providerId: { values: [1, 2, 3], operator: containedIn},  
5       lastUpdated: { value: "2024-01-17T18:00:00.00Z", operator: after}  
6     }  
7   } {  
8     id  
9     name  
10    parkingType  
11    lastUpdated  
12    provider {  
13      id  
14    }  
15  }  
16 }
```

Diese Abfrage gibt also nur Entitäten zurück, welche providerId "1", "2" oder "3" haben **und** nach dem 17.01.2024 00:00 Uhr zuletzt aktualisiert wurden.

Geofilter sind besondere Filter, die in Kapitel **Geofilter** beschrieben werden.

2.3.4 Abfragen gebündelter Daten

GraphQL erlaubt durch das setzen der entsprechenden Filter das Abfragen von gebündelten Daten. Mit gebündelten Daten sind in diesem Kontext die Daten mehrerer Provider gemeint, welche im Datenmodell der MDD NRW harmonisiert wurden.

Beispiel: Abfrage aller Fahrzeuge die zu den Providern mit den IDs 1,2 und 3 gehören:

```
1 query ListVehicles {
2   listVehicles(
3     filter: {
4       providerId: { values: [1, 2, 3] }, operator: containedIn}
5   ) {
6     id
7     lastReported
8     lastUpdated
9     lat
10    lon
11    status
12  }
13 }
```

Der Provider Filter erlaubt es außerdem Nutzern der MDD NRW die gebündelten Daten aller Zonen eines Providers abzufragen. Wenn zum Beispiel der Provider mit ID 1 Zonen mit den IDs 11, 12 und 13 hat sind folgende Abfragen identisch:

```
1 query ListVehicles {
2   listVehicles(
3     filter: {
4       providerId: { values: [1] }, operator: containedIn}
5   ) {
6     id
7     lastReported
8     lastUpdated
9     lat
10    lon
11    status
12  }
13 }
```

und

```
1 query ListVehicles {
2   listVehicles(
3     filter: {
4       providerId: { values: [11, 12, 13] }, operator: containedIn}
5   ) {
6     id
7     lastReported
8     lastUpdated
9     lat
10    lon
11    status
12  }
13 }
```

Im Datenmodell der MDD NRW wird nicht zwischen Providern und Provider-Zonen unterschieden. Des weiteren verfügt nicht jeder Provider über Zonen. Die Zugehörigkeit von Provider-Zonen zu Providern hat kann im Datenkatalog der MDD NRW oder dem Feld [parentProvider](#) in der [listProviders](#) und [providerbyID](#) Queries entnommen werden.

2.3.5 Geofilter

Ortsbezogene Daten der MDD NRW können mittels **Umkreisfilter** und **Polygonfilter** gefiltert werden. Umkreisfilter erlauben es Entitäten innerhalb eines nutzerdefinierten Kreises abzufragen. Wenn ein Attribut als Feld-Filter "[LocationFieldFilter](#)"

```
1 input LocationFieldFilter {
2   lat: Float
3   lon: Float
4   rangeInMeter: Float
5 }
```

hat ist es möglich durch übergeben eines Mittelpunkts, in der Form von longitude und latitude Koordinaten und eines Radius in Metern nur jene Entitäten zu erhalten, die innerhalb des dadurch entstehenden Kreises liegen.

Beispiel: Umkreisfilter mit Radius 15km

```
1 query {
2   listParkingFacilities(
3     filter: {
4       coordinateInRadius: { lat: 50.786075564, lon: 6.101432940, rangeInMeter: 15000 }
5     }
6   ) {
7     id
8     name
9     lat
10    lon
11    provider {
12      id
13    }
14  }
15 }
```

Der Polygonfilter erlaubt es eine Liste von Geokoordinaten zu übergeben. Es werden nur Entitäten zurückgegeben die sich innerhalb des so definierten Polygons befinden. Das gegebene Polygon muss dabei zusammenhängend und geschlossen sein, das heißt der erste Punkt entspricht dem letzten. Um mehrere, nicht überlappende Polygone abzufragen müssen multiple Abfragen gestellt werden.

Beispiel: Polygonfilter

```
1 query {
2   listParkingFacilities(
3     filter: {
4       coordinateInPolygon: {coordinates : [{lat: 50.781547, lon: 6.087198},
5                                           {lat: 50.790207, lon: 6.057198},
6                                           {lat: 50.774207, lon: 6.057198},
7                                           {lat: 50.781547, lon: 6.087198}]}
8     }
9   ) {
10    id
11    name
12    lat
13    lon
14    provider {
15      id
16    }
17  }
18 }
```

2.3.6 Datenkatalog Meta-Daten

Die Synchronisation zwischen Datenkatalog und MDD NRW erlaubt es die in dort gesetzten Meta-Daten als Filter für GraphQL zu verwenden.

Die verfügbaren Meta-Daten für alle verfügbaren Provider sind über die folgende Abfrage verfügbar:

```
1 query {  
2   listMetakeywords {  
3     keyword  
4     type  
5     value  
6     id  
7   }  
8 }
```

Die Datenkatalog Meta-Daten setzen sich aus drei Feldern zusammen, [Typ](#), [Schlüsselwort](#) und [Wert](#).

Typ:

Der Typ des Meta-Datums entspricht den verschiedenen Schlüsselwort Typen des Datenkatalogs. Die möglichen Werte sind als ENUMS definiert und müssen ohne Anführungszeichen (") übergeben werden. Die Möglichen Typen sind:

- TAG
- GROUP
- FORMAT
- LICENSE
- CUSTOM_FIELD

Schlüsselwort:

Ist nur notwendig bei Abfrage von Meta-Daten des Typs "CUSTOM_FIELD". Benutzerdefinierte Felder bieten mit dem Schlüsselwort eine weitere Ebene in der Hierarchie bei der Abfrage um verschiedene Benutzerdefinierte Felder voneinander abtrennen zu können. Das Schlüsselwort entspricht bei allen Typen außer "CUSTOM_FIELD" dem [Wert](#) des Meta-Datums.

Wert:

Dieses Feld beinhaltet die Ausprägung des Meta-Datums. **Beispiel:** „Bikesharing“.

Die Meta-Daten können beim Filtern der Provider angewendet werden, in dem eine liste von von [Typ](#), [Schlüsselwort](#) und [Wert](#) Kombinationen in den [metaKeyword](#) Filter gesetzt werden. Das Filtern einer Kombination dieser Felder ist mit einem logischen „**UND**“ verknüpft und liefert alle Datenlieferanten, auf die diese Kombination zutrifft.

Beispiel:

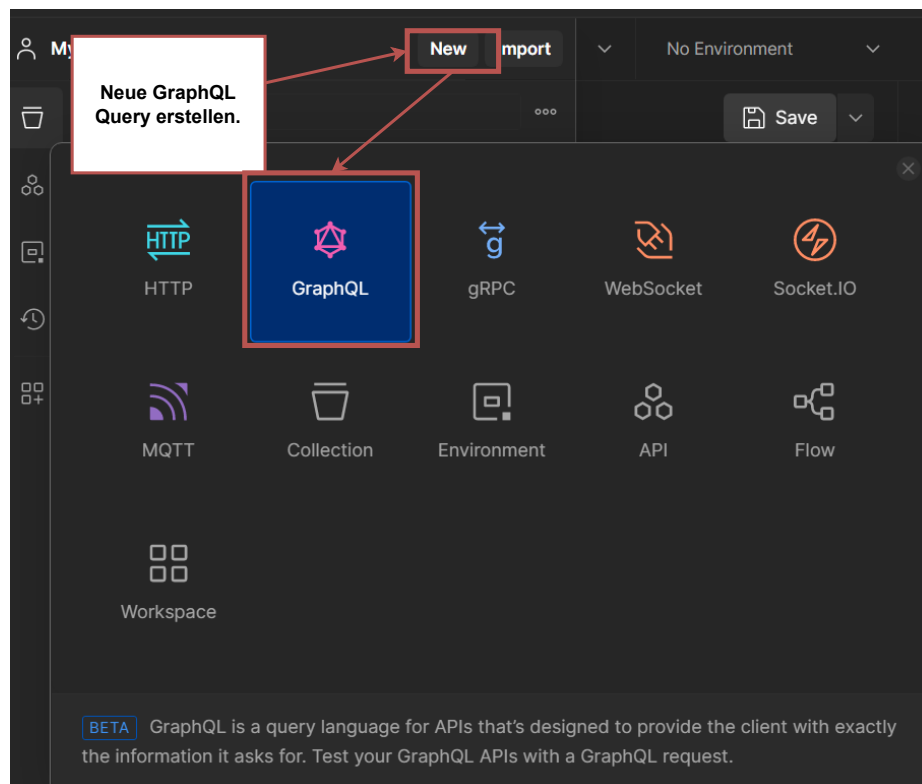
Schema für eine Abfrage von providern mit Meta-Daten Filter.

```
1 query {  
2   listProviders(  
3     filter: {  
4       metaKeyword: [  
5         { type: <Typ> }  
6         { keyword: "<Schluessselwort>" }  
7         { value: "<Wert>" }  
8       ]  
9     }  
10  ) {  
11    id  
12    name  
13    metakeywords {  
14      keyword  
15      type  
16      value  
17    }  
18  }  
19 }
```

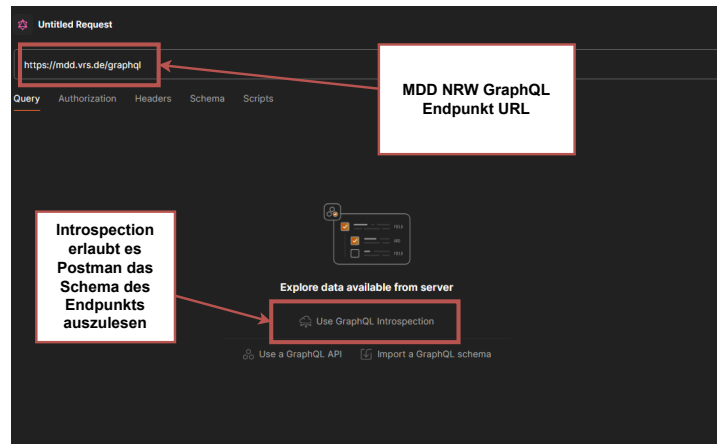
2.3.7 Postman Query Baum

Zum einfachen erstellen von Queries empfehlen wir die Nutzung von [Postman](#). Über die GraphQL Introspection ist es möglich zum Schema passende Queries direkt in Postman zu erstellen.

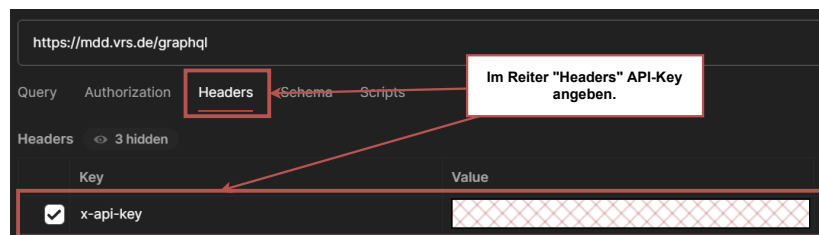
Dazu wird zunächst eine neuer GraphQL Request in einem Postman Workspace erstellt.



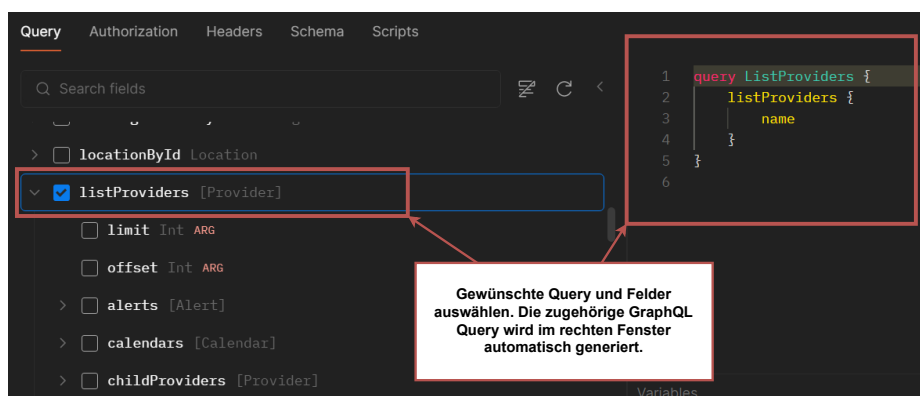
Als URL wird der GraphQL Endpunkt der MDD NRW angegeben und die Introspection gestartet.



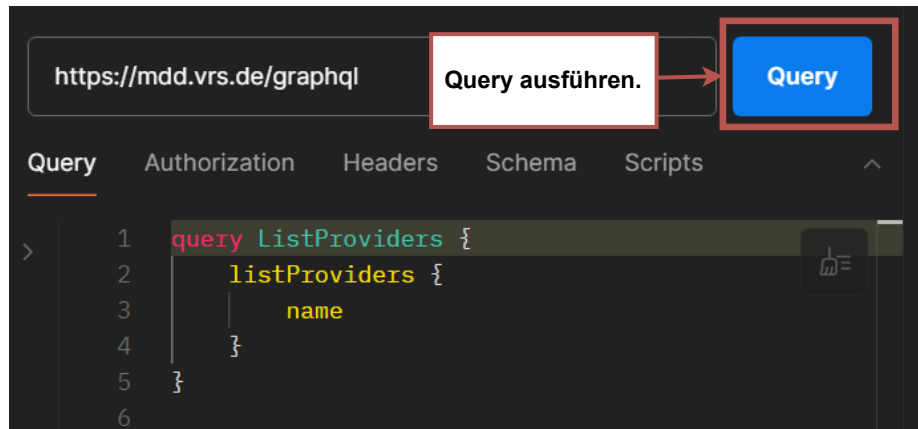
Für die Autorisierung muss im Header über den Key `x-api-key` ein gültiger API-Key übergeben werden, vgl. Kapitel [Autorisierung](#).



Postman erkennt nun automatisch die in Abschnitt [Queries](#) beschriebenen Endpunkte und erlaubt durch eine klickbare Oberfläche gültige Queries zu generieren.



Die Query kann nun direkt in Postman ausgeführt werden oder an anderer Stelle verwendet werden.



Postman unterstützt auch Pagination und Filter über diese Oberfläche, allerdings werden Listenfilter aktuell noch nicht unterstützt und müssen vom Nutzer manuell zur Query hinzugefügt werden.

2.3.8 GZIP Kompression

Es ist möglich die Größe des Outputs einer GraphQL Abfrage zu reduzieren, indem bei der Abfrage GZIP-Kompression verwendet wird.

Dafür müssen im Header der Anfrage folgende Parameter mitgegeben werden

Content-Type: application/graphql+json

und

Accept-Encoding: gzip

3 GBFS Endpunkt

Der GBFS Endpunkt kann per **GET**-Request nach folgendem Schema abgerufen werden:

```
https://mdd.gorheinland.com/gbfs-output/<provider_id>/<version>/<endpunkt>
```

Als [<provider>](#) wird der Name eines zulässigen Providers eingesetzt. In diesem Beispiel arbeiten wir mit dem erfunden Provider "ScooterSharer" mit ID "1". Das Keyword [<version>](#) bestimmt die Version des GBFS Feeds der abgefragt werden soll und als [<endpunkt>](#) stehen alle der gewählten Version üblichen Endpunkte nach [GBFS-Spezifikation](#) zur Verfügung. Über den [gbfs.json](#) Endpunkt erhält man somit auch eine Übersicht über die Verfügbaren GBFS-Endpunkte des jeweiligen Datenlieferanten in der MDD.

Beispiel: Abfrage des Auto-discovery Files des GBFS-Feeds der Version 2.3 für den Provider mit ID 1:

```
https://mdd.gorheinland.com/gbfs-output/1/2.3/gbfs.json
```

3.1 Liste der Unterstützten GBFS Versionen

Zum aktuellen Zeitpunkt unterstützt die MDD GBFS Feeds folgender Versionen:

- [Version 2.3](#)
- [Version 3.0](#)

4 Datex-II Endpunkt

Datex-II ist der europäische Standard für den Austausch von Verkehrsdaten.

Die MDD stellt für geeignete Anbieter dynamische und statische Datex-II Feeds zur Verfügung.

Das URL Schema für die Abfrage der statischen Feeds ist

```
https://mdd.gorheinland.com//datex2-output/<version>/<provider_id>/
```

Die URL für dynamische Feeds ist

```
https://mdd.gorheinland.com//datex2-output/<version>/<provider_id>/  
status
```

Welche Anbieter welche Feeds unterstützt kann den Factsheets entnommen werden.

Zum aktuellen Zeitpunkt unterstützt die MDD Datex-II in der Version 3.1.

[Link zur offiziellen Dokumentation.](#)

Das Archiv für das verwendete Datex-II Schema kann [hier](#) heruntergeladen werden.

Beispiel: Abfrage des statischen Datex-II Feeds der Version 3.1 für den Provider mit ID 1:

```
https://mdd.gorheinland.com/datex2-output/3.1/1/
```

5 Geoserver

5.1 Authentifizierung

Der Zugang zum Geoserver erfolgt anders als bei anderen Endpunkten über Basic Auth mit Nutzernamen und Passwort. Nutzernamen und Passwort setzen sich aus der Schlüssel-ID und dem API-Token eines Clienten zusammen.

Zugangstoken

Schlüsselname	Geoserver
Schlüssel-ID	4fe5c217-c69e-4046-be9e-8694a35c573b Nutzername
Erstellungsdatum	10.10.2024 08:55
Monatliches Abfragelimit	3000
Zugelassene Provider	Car&RideSharing Community eG Connected Mobility Düsseldorf VRR Fahrplan Echtzeitdaten goFLUX GTFS Velocity Free2move Lime wupsiCar

API Zugangsschlüssel

Passwort

YzFn [] IA==

Stellen Sie sicher, dass Sie den API-Zugang kopieren. Aus Sicherheitsgründen wird dieser nicht noch einmal angezeigt.

In Zwischenablage kopieren Schließen

Hinweis

Die Schlüssel-ID eines Clienten ist fest und ändert sich **nicht** wenn ein neues API-Token generiert wird. In diesem Fall muss für den Zugang zum Geoserver also nur das Passwort ausgetauscht werden.

5.2 Endpunkte

Die MDD NRW bietet Karten-Endpunkte über einen Geoserver an. Die Unterstützten Formate sind:

Jedes Format verfügt dabei über einen eigenen Endpunkt:

WMS

Endpunkt	Versionen
WMS	1.3.0 1.1.1
WMTS	1.1.1
TMS	1.0.0
WMS-C	1.1.1
WFS	2.0.0 1.1.0 1.0.0
WCS	2.0.1 1.1.1 1.1.0 1.1 1.0.0

```
https://mdd.gorheinland.com/geoserver/ows?service=WMS&version=<version>&request=GetCapabilities
```

WMTS

```
https://mdd.gorheinland.com/geoserver/gwc/service/wmts?service=WMTS&version=<version>&request=GetCapabilities
```

TMS

```
https://mdd.gorheinland.com/geoserver/gwc/service/tms/<version>
```

WMS-C

```
https://mdd.gorheinland.com/geoserver/gwc/service/wms?service=WMS&version=<version>&request=GetCapabilities&tiled=true
```

WFS

```
https://mdd.gorheinland.com/geoserver/ows?service=WFS&version=<version>&request=GetCapabilities
```

WCS

```
https://mdd.gorheinland.com/geoserver/ows?service=WCS&version=<version>&request=GetCapabilities
```

Beispiel: Abfrage des WCS Services mit Version 2.0.0

```
curl --location 'https://mdd.gorheinland.com/geoserver/ows?service=WCS&version=2.0.0&request=GetCapabilities' \
--header 'Authorization: <Base64Encode{Nutzername:Passwort}>'
```

6 Direkte Schnittstellen

Datenlieferanten, deren Daten nicht in der MDD gespeichert werden aber über die MDD Abfragbar sind werden als direkte Schnittstellen zur Verfügung gestellt.

Diese Schnittstellen sind über die MDD Abfragbar indem der Request über die MDD direkt an den betreffenden Service weiter geleitet wird.

Zum jetzigen Zeitpunkt ist die direkte Abfrage von Daten aus [DELFI Landeshintergrundsystem](#) und [goFLUX Mobility GmbH](#) möglich.

6.1 DELFI Landeshintergrundsystem

Das DELFI Landeshintergrundsystem verwendet als Schnittstelle die vom [VDV](#) entwickelte VDV 431 TRIAS Schnittstelle. Die Dokumentation kann auf dieser Website eingesehen werden: [Informationen zu TRIAS](#).

Abfragen werden über den Endpunkt

```
POST https://mdd.gorheinland.com/delfi
```

unter Angabe des MDD API-Key gestellt, siehe Kapitel [Autorisierung](#). Der Content-Type des Requests ist "application/xml".

Der Inhalt des Requests wird über den Request-Body abgefragt, dieser muss dem TRIAS-Format entsprechen.

Beispiel:

Schematischer Request-Body für eine DELFI Abfrage:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Trias version="1.2" xmlns="http://www.vdv.de/trias" xmlns:siri="http://www.siri.org.uk/siri">
3   <ServiceRequest>
4     <siri:RequestTimestamp>YYYY-MM-DDTHH:mm:SS</siri:RequestTimestamp>
5     <siri:RequestorRef>xxx</siri:RequestorRef>
6     <RequestPayload>
7       <StopEventRequest>
8         <Location>
9           <LocationRef>
10            <StopPointRef>STOP_POINT_REF</StopPointRef>
11            </LocationRef>
12            <DepArrTime>YYYY-MM-DDTHH:mm:SS</DepArrTime>
13          </Location>
14          <Params>
15            <NumberOfResults>10</NumberOfResults>
16            <StopEventType>arrival</StopEventType>
17            <IncludePreviousCalls>true</IncludePreviousCalls>
18            <IncludeOnwardCalls>true</IncludeOnwardCalls>
19            <IncludeRealtimeData>true</IncludeRealtimeData>
20          </Params>
21        </StopEventRequest>
22      </RequestPayload>
23    </ServiceRequest>
24  </Trias>

```

6.2 goFLUX Mobility GmbH

Abfragen werden über den Endpunkt

POST <https://mdd.gorheinland.com/goflux>

unter Angabe des MDD API-Key gestellt, siehe Kapitel [Autorisierung](#). Der Content-Type des Requests ist "application/json".

Der Inhalt des Requests wird über den Request-Body abgefragt, dieser muss dem goFLUX-Format entsprechen. Die Schnittstellenbeschreibung kann auf dieser Website eingesehen werden: [Dokumentation goFLUX](#).

Beispiel

Schematischer Request-Body für eine goFLUX Abfrage:

```
1 {  
2   "date": {unix_timestamp[s]}, # Beispiel: 1721137180  
3   "timeDelta": 7200,  
4   "departureLat": 48.800534,  
5   "departureLng": 2.295088,  
6   "arrivalLat": 48.765463,  
7   "arrivalLng": 2.072499,  
8   "departureRadius": 10,  
9   "arrivalRadius": 10,  
10  "expectedDepartureDateTime": "{{request_timestamp}}"  
11 }
```