



Transition mellem REST-services og GraphQL-services

Drift og modernisering af Datafordeleren

August 2025



Indholdsfortegnelse

1	Introduction	3
1.1	Intended Audience	3
1.1.1	Prerequisites	3
1.2	Glossary	3
1.3	Limitations	4
1.3.1	Det Centrale Personregister (<i>CPR</i>)	4
2	Data Sources	4
2.1	REST-service documentation	4
2.1.1	Data Overview Documentation	4
2.1.2	Pseudo SQL Code in DLS	5
2.2	GraphQL-service documentation	5
3	Process	6
3.1	Step 0: Evaluate REST-service	6
3.2	Step 1: Determine suitable GraphQL-service	7
3.2.1	FO or Entity-based?	7
3.3	Step 2: Find the root entity	10
3.4	Step 3: Analyse joins	10
3.5	Step 4: Analyse filters	11
3.6	Step 5: Determine fields to fetch	13
3.7	Step 6: Verify the result	13
4	Example: EjerskabMedStamoplysninger	14
4.1	Step 0: Evaluate REST-service	14
4.2	Step 1: Determine suitable GraphQL-service	14
4.3	Step 2: Find the root-entity	15
4.4	Step 3: Analyse joins	15
4.4.1	1 st section of joins: For ejende person-/virksomhedsoplysning	15
4.4.2	2 nd section of joins: For ejende virksomhed	16
4.4.3	3 rd section of joins: For ejende person	18
4.4.4	4 th section of joins: For administrerende person-/virksomhedsoplysning	19
4.4.5	5 th section of joins: For administrerende virksomhed	20
4.4.6	6 th section of joins: For administrerende produktionsenhed	21
4.4.7	7 th section of joins: For administrerende person	22
4.5	Step 4: Analyse filters	23
4.5.1	1 st and 2 nd filter: Ejerskab	25
4.5.2	3 rd filter: PersonVirksomhedsoplysninger for ejende join	25
4.5.3	4 th filter: AlternativAdresse for ejende join	26
4.5.4	5 th filter: PersonVirksomhedsoplysninger for administrerende join	26
4.5.5	6 th filter: AlternativAdresse for administrerende join	27
4.6	Other filters – snippets	27
4.7	Step 5: Determine fields to fetch	27
4.8	Step 6: Verify result	28
4.8.1	REST API response	29
4.8.2	GraphQL query response	30



1 Introduction

This document provides guidance for transitioning from REST-based services to GraphQL-based services. It offers strategic approaches for locating service information and interpreting this data effectively—specifically, how REST functionality maps to corresponding GraphQL implementations.

While this guide does not guarantee perfect one-to-one translations between REST and GraphQL services (as direct equivalents are not always feasible), it equips developers with the knowledge to identify the closest possible matches and understand the inherent differences when they arise.

The following sections will walk through the fundamental differences between REST and GraphQL, mapping strategies, and practical implementation considerations to ensure a smooth transition between these service models. Before proceeding with the technical details, it's important to clarify who this transition guide is designed to serve.

1.1 Intended Audience

This document serves as a guide for developers transitioning from REST-services to GraphQL-services. It should be noted that the recommended approach is to design GraphQL queries that align with specific requirements rather than strictly adhering to current REST-service patterns since GraphQL-services inherently enable the creation of queries that more precisely match individual data needs.

1.1.1 Prerequisites

This document is intended for readers who:

- Possess technical knowledge beyond entry-level development experience
- Have prior familiarity with data distribution platforms and the data they expose
- Understand fundamental principles of GraphQL and SQL, or are willing to learn these technologies

Without this foundational knowledge, readers may require significant additional time to comprehend and implement the guidelines presented.

1.2 Glossary

The terminology presented in Table 1 is essential for understanding the concepts discussed throughout this document

Term	Definition	Abbreviation
Flexible Query Logic	A GraphQL service (flexible/FLEX and flexibleCurrent/FLEX_CURRENT) that enables joins within and between registers, allowing for more complex data relationships and queries.	FO
Entity-based GraphQL	GraphQL services that retrieve discrete entities and do not support join operations between data sets, maintaining strict boundaries between different data entities.	
Dataleverancespecifikation	Comprehensive documentation detailing the data structure for a register on The Data Distribution Platform. This documentation includes XSD and XML schemas which together constitute the	DLS



	complete data model, as well as pseudo SQL code that defines the REST service implementations.	
--	--	--

Table 1: Glossary

1.3 Limitations

This guide has specific boundaries and exceptions that users should be aware of before proceeding with service transitions. The following sections outline systems and scenarios where standard REST to GraphQL mapping approaches may not apply or require special consideration.

1.3.1 Det Centrale Personregister (CPR)

The Central Person Registry (CPR) operates outside the Fleksible Opslagslogik (FO) framework. As a result, CPR does not maintain its own relations, though other registers may establish relations to CPR data.

This architectural distinction, combined with the existence of specialized GraphQL services for CPR (CPRCustom_PrivateSector, CPRCustom_PublicSector) that perform join operations, places CPR outside the scope of this transition guide.

For developers requiring GraphQL functionality that corresponds to CPR REST services, the recommended approach is to use the custom services.

2 Data Sources

Understanding both REST and GraphQL services in depth is essential for successfully translating between these architectural paradigms. This understanding requires access to comprehensive documentation that details the structure, capabilities, and limitations of each service type.

This section outlines the necessary documentation resources for both REST and GraphQL services, where to locate these resources, and effective strategies for interpreting and applying the information they contain.

2.1 REST-service documentation

The documentation for REST services is centralized on Confluence, though it exists in two distinct but complementary sources that together provide a comprehensive understanding of each service.

2.1.1 Data Overview Documentation

The Data Overview section serves as the primary entry point for understanding REST services. This documentation provides an excellent starting point for gaining an initial understanding of what each REST service does and the structure of its response data. Consists of:

1. **Service descriptions:** provided under the "Beskrivelse" heading, offering context and purpose for each service
2. **Input parameters:** lists all request parameters and if they are required or not.
3. **Output structure:** a link to a JSON schema that contains specifications

Can be found at: <https://confluence.sdfi.dk/display/DML/Dataoversigt>. Once there, select the desired register from the available options, then click on "REST" in the side menu. This will lead to a listing of all available REST-services for the chosen register and a data overview for each service.



2.1.2 Pseudo SQL Code in DLS

The DLS contains Pseudo SQL code that offers deeper insights into service implementation:

- These specifications were used to initially define and implement the REST services
- Unlike the overview documentation, DLS reveals not just the inputs and outputs but the actual computational process
- Critical implementation details such as JOIN operations and filtering logic are documented here

The Pseudo SQL code is particularly valuable during the transition process as it exposes the underlying data relationships and logic that must be preserved when moving to GraphQL implementations.

Can be found at: <https://confluence.sdfi.dk/display/DML/DLS-udstilling> under heading "Dataleverancespecifikation (DLS) i produktion". Within the DLS folder, look for SQL-files that share name with the REST-service that is of interest. If such a SQL-file cannot be found, the file may be found in the appendix 2 Excel-file. If neither contain any service logic, it can be that the register has not created one and that this documentation is unfortunately not available for use. If this is the case, a request should be sent in through the portal support. Because it may vary where the registers store the REST-service logic, consider looking through the DLS in its entirety.

2.2 GraphQL-service documentation

The GraphQL-service documentation is presented in GraphQL-schemas. The schemas specify the types of data that can be retrieved, the relationships between these types, and the filtering operations that can be supported for the respective fields. You can use the schemas to see which queries you can send and what data you can retrieve.

Proper documentation on how to get started with the GraphQL services and how to fetch the schemas can be found at: <https://confluence.sdfi.dk/display/DML/GraphQL> (specifically see subpage "[GraphQL på Datafordeleren](#)" and headings "[Hent Schema](#)" and "[Autentifikation og autorisation for GraphQL-tjenester](#)").

In short, the schemas are accessed at the URL presented in Table 2 and require a form of authentication credentials. The GraphQL schema you need to fetch depends entirely on which service will process your query. To identify the most suitable GraphQL service for your requirements and determine the correct schema to retrieve, refer to section 3.2 "[Step 1: Determine suitable GraphQL-service](#)".

Service Type	Service Names	URL Pattern
Entity-based GraphQL	BBR, CPR, CPRCustom_PrivateSector, CPRCustom_PublicSector, CVR, CVRCustom, DAGI, DAR, DHMHoejdekurver, DHMOPridelse, DS, EBR, EJF, FIKSPUNKT, GeoDKV, HISTKORT, MAT, SVR, VUR	<a href="https://graphql.datafordeler.dk/<service_name>/v<version>/schema">https://graphql.datafordeler.dk/<service_name>/v<version>/schema
FO	flexible, flexibleCurrent	

Table 2: URL for GraphQL-services



3 Process

In this section the process of translation is outlined, step by step and the following steps are part of the process:

- 3.1 Step 0: Evaluate REST-service
- 3.2 Step 1: Determine suitable GraphQL-service
- 3.3 Step 2: Find the root entity
- 3.4 Step 3: Analyse joins
- 3.5 Step 4: Analyse filters
- 3.6 Step 5: Determine fields to fetch
- 3.7 Step 6: Verify the result

The process is applied to an example in section 4 “*Example: EjerskabMedStamoplysninger*”.

3.1 Step 0: Evaluate REST-service

Before committing to a translation process, carefully evaluate whether converting a REST service is truly necessary. This evaluation is crucial as the translation requires significant effort and should only be undertaken when absolutely required.

Evaluation Process

1. Review REST-service documentation
 - a. Consult the documentation described in section 2.1.1 “*Data Overview Documentation*”.
 - b. Examine the REST-service name, the description, the input parameters and the output structure (JSON-schema).

2. Assess Your Data Requirements

Consider:

- a. Does the response from the REST-service provide all the data needed?
- b. Does the response from the REST-service provide more data than needed?

If the REST-service fails to provide all required data or delivers substantially more data than needed, it's recommended to explore creating a new custom query rather than following the structure of an existing REST service. This approach often yields more efficient and targeted results.

For assistance with custom query development:

1. Visit: <https://datafordeler.dk/dataoversigt/>
2. Filter on “GraphQL Query” in the side menu.
3. Download the GraphQL schema.
4. Open the GraphQL schema in a GraphQL IDE of choice
5. Utilize the GraphQL IDE with the GraphQL schema to construct tailored queries.

If the REST-service does fit the requirements, then proceed with the translation starting with the next step.



3.2 Step 1: Determine suitable GraphQL-service

The first step in translating a REST service to a GraphQL query is determining the correct GraphQL service to utilize. This is vital not only to later be able to perform the query to the suitable service, but to fetch the relevant GraphQL-documentation (the relevant schema).

To do so we need to consider:

1. Is a FO GraphQL-service required or an entity-based one?
 - a. If it is a FO GraphQL-service, which one?
 - b. If it is a entity-based GraphQL-service, which one?

This section explores these considerations to guide the process of determining the suitable GraphQL-service.

3.2.1 FO or Entity-based?

The fundamental difference between FO and entity-based GraphQL services lies in their join capabilities. FO services support joins both within a register and between different registers, while entity-based services do not offer join functionality.

To determine which type of GraphQL service is appropriate, ask yourself: "Can I limit my search to a single point in time?". "Are joins required for this implementation?".

The first question is answered by considering whether you need both registreringstid and virkningstid at the same time, in which case FO could be the right choice, because these are required parameters, whereas these are not required in entity-based GraphQL-services.

You can easily answer the second question by examining the pseudo SQL code in the REST service documentation. If the word "join" appears in the code, you'll need an FO service. If no joins are present, an entity-based service will suffice. However, you should still consider if querying multiple entity-based services will serve your needs, as you're also able to join the data in your own system.

Important Consideration: It's important to note that the GraphQL services only support left joins. If the REST service uses other join types (inner, right, full outer), you face a decision point. While you could approximate these with left joins in your GraphQL implementation, be aware that this substitution may produce different results compared to the original REST service and require post-processing in your own system.

Such differences could affect both calculations and response structures, potentially creating inconsistencies between the REST service and your GraphQL query. This limitation should be carefully considered when performing the translation.

3.2.1.1 Which FO-service?

When determining which FO-service to use, the key distinction lies in bitemporal filtering capabilities. Let's examine their differences (see Table 3).

Service	Virkningstid	Registreringstid	Use Case
flexible	Customizable - allows filtering by any effective date	Customizable - allows specifying a historical registration time point	When you need to see "what the system knew at time X about what



			was effective at time Y" (full bi-temporal view).
flexibleCurrent	Customizable - allows filtering by any effective date	Fixed to "NOW" (current time)	When you need to see "what the system currently knows about what was/is/will be effective at time Y".

Table 3: FO-services, bitemporal filtering and their use-cases

In short, the difference between the service is the bitemporal filtering that is allowed. FlexibleCurrent does not allow users to filter on registreringstid, but this is allowed on flexible. This is not to be confused by the fields "registsteringFra" and "registreringTil" which can be filtered on afterwards, however the filtering on registreringstid always happens before that on FlexibleCurrent.

To determine which service to use, examine the REST-service input parameters in the service documentation. If parameters related to registration time filtering (registreringFra or registreringTil) are required for your use-case, you must use the flexible service as it's the only one supporting historical registration time points.

In most cases, registration time filtering is optional. If you don't need to filter on registration time, the flexibleCurrent service is recommended. Similarly, if the REST-service doesn't support registration time filtering at all, flexibleCurrent should be your choice.

Important Considerations

1. Bitemporal Filtering Requirements. Each service has specific filtering requirements. The flexible service requires both registreringstid and virkningstid filters to be specified. In contrast, the flexibleCurrent service only requires effective time filtering, as registration time is automatically set to the current moment.
2. Register Availability. Not all data registers are available in both services. For example, the CVR register is only accessible through the flexibleCurrent service and not available in the flexible service due to requirements from the registers.
3. To determine whether a specific register is available in a particular FO-service, consult the comment above the type Query definition in the GraphQL schema (searchable with "type Query"). This comment provides a comprehensive list of all registers available in the service, along with the version of each register service that has been integrated. The version of each register service corresponds to the entity-based service with the same version number.

3.2.1.2 Which Entity-based GraphQL-service?

Determining which entity-based GraphQL-service to use is rather easy. The register that owns the REST-service should also own the GraphQL-service that is queried. For instance, if a BBR REST-service is to be translated a version of the BBR GraphQL-service is to be queried.

Service	Use Case
BBR	When you would usually use the REST-services from the corresponding register.
CPR	
CVR	
DAGI	



DAR	
DHMHøjdekurver	
DHMOprindelse	
DS	
EBR	
EJF	
FIKSPUNKT	
GEODKV	
HISTKORT	
MAT	
SVR	
VUR	

Important consideration: As part of entity-based GraphQL there are also additional custom services (see in Table 4) with specific use cases in mind. When choosing an entity-based GraphQL-service the use cases, described in the table, should be considered.

Custom service/entities	Use Case
CPRCustom_PublicSector	When you would usually use the CPR REST services for public authorities, for example "CprPersonFullComplete", or "CprPersonSmallSimple".
CPRCustom_PrivateSector	When you would usually use the CPR REST services by the name of "CprPrivateAdressName", "CprPrivateDateOfBirthName", or "CprPrivatePNR".
CVRCustom	When you would usually call a CVR REST service that accesses the entity by the name of "CVRPerson", while being a private organization.
EJFCustom	This is not a service like the others, but instead include a set of custom EJF entities in the FO service that may be joined to CPR entities. They are used when you would usually join to CPR from EJF.

Table 4: Custom services and their use-cases. Note that this table only lists custom GraphQL-services and therefore does not contain the regular entity-based GraphQL-services from each of the registers.



3.3 Step 2: Find the root entity

With a GraphQL-service in mind, we can take a look at the constructing a query. This starts with finding the root entity, which represents the primary data object to which the GraphQL query will be directed. To identify this critical element:

1. From the REST-service, determine the entities you are interested in, as well as what filtering possibilities you need. It is **highly recommended** that the entity with the most selective filtering of results is chosen as your root entity.
2. In the GraphQL schema, locate the "type Query" section
 - a. This section lists all available top-level queries for the service
 - b. Find the query that corresponds to your identified root entity
 - c. Note that the query name may include a prefix to the root entity name
 - d. Implementing the Root Entity in Your Query

Once identified, the root entity (referred to as "NameOfRootEntityInQueryType") forms the foundation of your GraphQL query:

```
{  
  <NameOfRootEntityInQueryType>() {  
  }  
}
```

3.4 Step 3: Analyse joins

Once the foundation of the query is constructed, we can move on to filling out more of the details by analyzing and implementing the necessary joins.

If, in previous steps, you've concluded that there are no joins required for your query, then you can proceed directly to the next step. However, most complex queries will involve one or more joins that need to be correctly translated to GraphQL.

In traditional SQL, a join is defined in the Pseudo SQL as follows:

```
LEFT JOIN <TargetEntity> AS <AliasTargetEntity>  
ON <AliasTargetEntity>.<TargetField> = <AliasSourceEntity>.<SourceField>
```

However, GraphQL handles relationships differently.

To translate SQL joins to GraphQL:

1. First, locate the type definition of the source entity in the GraphQL schema.
2. Within this definition, examine all available fields that can be fetched from this entity, including fields that represent relationships (join-fields).
3. Pay special attention to the schema types of the fields. One-to-many relations are denoted as "connections" in the schema.
 - Alternatively, one can pay attention to the comments above these fields in the schema. It is not recommended to build any kind of functionality that relies on the content of the comments. These comments can provide information about:
 1. The type of relationship (one-to-one vs. one-to-many)



2. The source and target entities involved
3. The specific fields that establish the relationship

Using these comments as your guide, identify the field in the GraphQL schema that corresponds to the join defined in your pseudo SQL code. It is not possible to create your own relationships and all relationships are predetermined. If a relationship is supposed to exist but does not exist, a request should be made through the support site of DAF.

Given the two following definitions in pseudo SQL-code,

```
LEFT JOIN <TargetEntityToOne> AS <AliasTargetEntityToOne>  
ON <AliasTargetEntityToOne>.<TargetFieldToOne> = <AliasSourceEntityToOne>.<SourceFieldToOne>  
  
LEFT JOIN <TargetEntityToMany> AS <AliasTargetEntityToMany>  
ON <AliasTargetEntityToMany>.<TargetFieldToMany> = <AliasSourceEntityToMany>.<SourceFieldToMany>
```

where the first is a to-one relationship and the second is a to-many relationship, the GraphQL query in previous steps would be extended like follows.

```
{  
  <NameOfRootEntityInQueryType>() {  
    JoinFieldToOne {  
    }  
    JoinFieldToMany {  
    }  
  }  
}
```

Important consideration:

1. Nested joins. The first join in the pseudo SQL code is usually from the root entity but consecutive joins might be nested. Ensure that you join from the correct entity.
2. Join depth and breadth. There are limitations imposed on the GraphQL-services restricting join depth and breadth. Some REST-services might defy this limit. The translated REST-service, should then be split into two or more queries.

3.5 Step 4: Analyse filters

Another key aspect to the REST-services is the filtering. Filtering is defined in the pseudo SQL in two ways. The first being where-statements. They are structured as follows:

```
WHERE <cond>
```

The condition (<cond>) can be translated to a filter in queries. The GraphQL-schema is helpful when understanding how the condition should be translated to a filter in the GraphQL queries. When examining the GraphQL schema, you will notice that filter objects are defined as input types. For example, if you're querying a collection of users, the schema might define a 'UserFilter' input type. These input types contain fields that correspond to the filterable properties of the entity, each with appropriate operators.

For instance, a simple condition like 'WHERE age > 18' from your pseudo SQL would translate to a GraphQL filter structure like:

```
where: {age: { gt: 18 }}
```



The schema will define these filter operators (such as 'gt' for greater than, 'eq' for equals, 'contains' for string inclusion, etc.) as nested fields within each filterable property.

In the statement that follows parameters that are initiated with a '@' signifies input parameters to the REST-services and the statement is an example of how definition of filtering on the field "Id" might look.

```
WHERE (@Id = NULL OR RootEntity.id In @Id)
```

The first part "@Id = NULL OR" signifies that this filtering is optional. The second part signifies that, if the Id-param is given, the REST-service evaluates id_lokalId is part of the Id string.

Which would be translated to something like the following in GraphQL.

```
where:{id: {in: "GivenString"}}
```

If we were to expand the GraphQL-query we have been working on so far it might look something like:

```
{
  <NameOfRootEntityInQueryType>{
    first: 10,
    where: {id: {in: "GivenString"}}
  }
  nodes {
    JoinFieldToOne {
    }
    JoinFieldToMany {
    }
  }
}
```

Important considerations:

1. Registreringstid/Virkningstid parameters. There are filter requirements on the FO-services. In each query to these registreringstid or virkningstid must be defined. Defining registreringstid or virkningstid is done as follows:

```
{
  <NameOfRootEntityInQueryType>{
    registreringstid: "2025-01-01T01:01:01Z",
    virkningstid: "2025-01-01T01:01:01Z",
    first: 10,
    where: {id: {in: "GivenString"}}
  }
  nodes {
    JoinFieldToOne {
    }
    JoinFieldToMany {
    }
  }
}
```

The query will only return those responses which existed in the system at "registreringstid" and were valid in reality at "virkningstid".

2. Besides the where statements, some registers utilize "snippets" of SQL code (often defined in another file in the DLS) to further define bitemporal filtering for joins. Remember to place this filtering



at the correct level, i.e., if it is connected to a join, add the filtering to the join-field and not at the top-level on the root-entity. Consider if you need to implement post-processing of joined results that evaluated to null-values afterwards in your own system. Furthermore, it is important that bitemporal filtering that happens on the top-level automatically cascades to the subentities, using each table's own bitemporal filtering rules.

3.6 Step 5: Determine fields to fetch

The last bit of the query is to understand what fields to fetch. In the *SELECT/FROM* statement, between the SELECT and FROM is a list of fields that are to be fetched via the query. The more common cases is that the REST-services fetch just a singular field or all fields (using the *** syntax).

Below is an example of what it might look like when all fields are to be fetched. Remember that all fields in GraphQL must be explicitly specified, as is required by design in GraphQL.

```
{
  <NameOfRootEntityInQueryType> (
    registreringstid: "2025-01-01T01:01:01Z",
    virkningstid: "2025-01-01T01:01:01Z",
    first: 10,
    where: {id: {in: "GivenString"}}
  )
  nodes {
    JoinFieldToOne {
      SubFieldToOne1,
      SubFieldToOne2
      ...
      SubFieldToOneN
    }
    JoinFieldToMany {
      SubFieldToMany1
      ...
      SubFieldToManyN
    }
    Field1,
    Field2,
    Field3
    ...
    FieldN
  }
}
```

3.7 Step 6: Verify the result

At this point there should be a full GraphQL-query and the very last step of the translation process is to verify the result. This part is done by ensuring that the GraphQL-query that has been created is valid and will yield a response. This is done by trying to execute it.

Next, the response should be compared to the JSON-schema that is defined for the structure of the response. This schema. If the response from the GraphQL-query does not follow the same structure as the JSON-schema defines for the REST-service, it can be worthwhile to consider why. It might just be due to adaptations made to handle differences in between the services (e.g., substituting all types of joins in the REST-services with the only type of joins permitted in the GraphQL-services, left-joins). However, if there does not seem to be any good reason for these differences it might mean that something has been overlooked in the translation process.

Consider:

1. Does the query work, i.e., do I get a response when I run it towards the GraphQL-service?



2. Does the query seem to fill the data needs that I had when I set out to translate the REST-service?

If the answer to both these questions is yes – then differences between the GraphQL-service response and the REST-service response may very well be acceptable.

4 Example: EjerskabMedStamoplysninger

In this section, the translation process (described in section 3. “*Process*”) is run through using an example. Some of the documentation is presented along with descriptions, but it is recommended to have the documentation at hand when reading and following along with the steps. How to fetch the relevant documentation is described in section 2. “*Data Sources*”.

In Table 5, basic information about the REST-service used as an example can be found.

Register	Ejerfortegnelsen (EJF, RC00023)
REST-service	Ejerfortegnelsen
REST-service method	EjerskabMedStamoplysninger

Table 5: Basic information REST-service

4.1 Step 0: Evaluate REST-service

The first step of the process was to evaluate whether the REST-service actually meets ones specific data needs. The questions to consider were:

1. Does the response from the REST-service provide all the data needed?
2. Does the response from the REST-service provide more than the data needed?

To answer the question, we view the documentation of the REST-service on Confluence, the overview one. In the overview we find the description, input parameters and output structure (JSON-schema). If the response to the questions were no, the recommendation was to abort the process. For the sake of the example, we are going to proceed with the process, as if the REST-service does fit our data needs.

4.2 Step 1: Determine suitable GraphQL-service

The second step in the process is to find the GraphQL-service that is most suitable to handle the query that the process yields. In order to do so we first have to determine whether we should use a FO or entity-based GraphQL-service. This is done by evaluating whether the REST-service requires joins which in turn can be determined by viewing the pseudo SQL-code. In the pseudo code, we find that there is need for joins which means we should utilize one of the FO-services.

Next, we need to determine which of the FO-services should be utilized. In order to do so, we again refer to the REST-service documentation (overview) and the input parameter list. In the list, we can see that it is not an option to filter on registration time and due to this, we can conclude that it is flexibleCurrent GraphQL-service that is most suitable. Furthermore, the REST-service requires joins to CVR, which is only available in the flexibleCurrent GraphQL-service.



4.3 Step 2: Find the root-entity

Next is to find the root-entity. This is done by viewing the REST-service documentation (pseudo SQL code), specifically the initial “*SELECT/FROM*” statement.

```
SELECT *  
FROM Ejerskab ejerskab
```

Using the statement, we identify “Ejerskab” as the root entity. With the root entity in hand, we view the query type definition in the flexibleCurrent GraphQL-schema and find the query that matches the root entity and find “EJF_Ejerskab”. With the query in hand, we can build the foundation of the GraphQL-query.

```
{  
  EJF_Ejerskab() {  
    nodes {  
    }  
  }  
}
```

4.4 Step 3: Analyse joins

With the foundation in place, we can continue to fill out the rest of the query. Part of this are the joins, which in GraphQL are represented by join-fields.

To find what join-fields to add to our GraphQL-query, we look to the REST-service documentation (pseudo SQL code). Here, we first evaluate the number of joins there are in the file. As the number of joins exceed the limit to the GraphQL-service, we immediately ascertain that there is need for splitting up the resulting GraphQL-query. Since the pseudo SQL already presents the joins in sections we create one query for each section.

4.4.1 1st section of joins: For ejende person-/virksomhedsoplysning

The first section of joins in the pseudo SQL-code is translated as follows.

```
-- Joins for ejende person-/virksomhedsoplysning  
LEFT JOIN PersonVirksomhedsoplys ejende_pv  
ON ejende_pv.id_lokalId = ejerskab.ejeroplysningerLokalId [snippet_bitemp_virkning(ejende_pv)]  
  
LEFT JOIN AlternativAdresse ejende_pv_alt_adresse  
ON ejende_pv_alt_adresse.id_lokalId = ejende_pv.alternativAdresseLokalId  
[snippet_bitemp_virkning(ejende_pv_alt_adresse)]  
  
LEFT JOIN DAR.Adresse ejende_pv_dar_adresse  
ON ejende_pv_dar_adresse.id_lokalId = ejende_pv.adresseLokalId  
[snippet_bitemp_virkning(ejende_pv_dar_adresse)]
```



```
{
  EJF_Ejerskab() {
    nodes {
      oplysningerEjesAfEjerskab {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```

The first join is between the root entity Ejerskab and the entity PersonVirksomhedsoplys, while the second and the third joins are nested in the first, i.e., are between entity PersonVirksomhedsoplys and AlternativAdresse and DAR.Adresse. We can also note, from viewing the documentation, that all of the relations are one-to-one relations.

4.4.2 2nd section of joins: For ejende virksomhed

The second section of joins in the pseudo SQL-code is translated as follows.

```
-- Joins for ejende virksomhed
LEFT JOIN CVR.Virksomhed ejende_virksomhed
ON ejende_virksomhed.CVRNummer = ejerskab.ejendeVirksomhedCVRNr
[snippet_bitemp_virkning_cvr(ejende_virksomhed)]

LEFT JOIN CVR.Reklamebeskyttelse ejende_virksomhed_reklame_beskyttelse
ON ejende_virksomhed_reklame_beskyttelse.CVREnhedsId = ejende_virksomhed.id
[snippet_bitemp_virkning_cvr(ejende_virksomhed_reklame_beskyttelse)]

LEFT JOIN CVR.Navn ejende_virksomhed_navn
ON ejende_virksomhed_navn.CVREnhedsId = ejende_virksomhed.id
[snippet_bitemp_virkning_cvr(ejende_virksomhed_navn)]

LEFT JOIN CVR.Adressering ejende_virksomhed_cvr_adresse ON
ejende_virksomhed_cvr_adresse.CVREnhedsId = ejende_virksomhed.id
[snippet_bitemp_virkning_cvr(ejende_virksomhed_cvr_adresse)]
  AND ejende_virksomhed_cvr_adresse.AdresseringAnvendelse = 'beliggenhedsadresse'
```



```
{
  EJF_Ejerskab() {
    nodes {
      ejendeVirksomhedCVRNr_20_Virksomhed_CVRNummer_ref {
        id_CVR_Reklamebeskyttelse_CVREnhedsId_ref {
          }
        id_CVR_Navn_CVREnhedsId_ref {
          nodes {
            }
          }
        id_CVR_Adressering_CVREnhedsId_ref(
          where: {
            AdresseringAnvendelse: { eq: "beliggenhedsadresse" }
          }
        ) {
          nodes {
            }
          }
        }
      }
    }
  }
}
```



4.4.3 3rd section of joins: For ejende person

```
-- Joins for ejende person
LEFT JOIN CPR.Personnummer ejende_person_personnumre
ON ejende_person_personnumre.personnummer = ejerskab.ejendePersonPersonNR
[snippet_bitemp_virk(ejende_person_personnumre)]

LEFT JOIN CPR.Person ejende_person
ON ejende_person.id = ejende_person_personnumre.personid [snippet_bitemp_virkning_cpr(person)]

LEFT JOIN CPR.Beskyttelse ejende_person_beskyttelse
ON ejende_person_beskyttelse.personid = ejende_person.id
[snippet_bitemp_virk(ejende_person_beskyttelse)]

LEFT JOIN CPR.Navn ejende_person_navn
ON ejende_person_navn.personid = ejende_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, ejendePersonPersonNR)])
[snippet_bitemp_virk(ejende_person_navn)]

LEFT JOIN CPR.CprAdresse ejende_person_cpr_adresse
ON ejende_person_cpr_adresse.personid = ejende_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, ejendePersonPersonNR)])
[snippet_bitemp_virk(ejende_person_cpr_adresse)]

LEFT JOIN CPR.UdrejseIndrejse ejende_person_ud_ind_rejse
ON ejende_person_ud_ind_rejse.personid = ejende_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, ejendePersonPersonNR)])
[snippet_bitemp_virk(ejende_person_ud_ind_rejse)]

LEFT JOIN DAR.Adresse ejende_person_dar_adresse
ON ejende_person_dar_adresse.id_lokalId = ejende_person_cpr_adresse.daradresse
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, ejendePersonPersonNR)])
[snippet_bitemp_virkning(ejende_person_dar_adresse)]
```

Due to CPR rules, the above SQL cannot be translated directly into GraphQL using the usual approach. For this reason, we instead apply a join to the EJFCustom entity that has predefined internal CPR-join logic pre-handled.



```
{
  EJF_Ejerskab() {
    nodes {
      ejendePerson {
        beskyttelser {
        }
        navn {
        }
        cprAdresse {
        }
        udrejseIndrejse {
        }
      }
    }
  }
}
```

4.4.4 4th section of joins: For administrerende person-/virksomhedsoplysning

```
-- Joins for administrerende person-/virksomhedsoplysning
LEFT JOIN PersonVirksomhedsoplys admin_pv
ON admin_pv.id_lokalId = ejerskab.administratoroplysLokalId [snippet_bitemp_virkning(admin_pv)]

LEFT JOIN AlternativAdresse admin_pv_alt_adresse
ON admin_pv_alt_adresse.id_lokalId = admin_pv.alternativAdresseLokalId
[snippet_bitemp_virkning(admin_pv_alt_adresse)]

LEFT JOIN DAR.Adresse admin_pv_dar_adresse
ON admin_pv_dar_adresse.id_lokalId = admin_pv.adresseLokalId
[snippet_bitemp_virkning(admin_pv_dar_adresse)]
```

```
{
  EJF_Ejerskab() {
    nodes {
      ejerskabAdministreresAfPersonEllerVirksomhedsoplysninger {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```



The first join is from the root entity, while the two consecutive joins are nested in the entity fetched from the first join, PersonVirksomhedsoplysninger.

4.4.5 5th section of joins: For administrerende virksomhed

```
-- Joins for administrerende virksomhed
LEFT JOIN CVR.Virksomhed admin_virksomhed
ON admin_virksomhed.CVRNummer = ejerskab.administrerendeVirksomhedCVRNr
[snippet_bitemp_virkning_cvr(admin_virksomhed)]

LEFT JOIN CVR.Reklamebeskyttelse admin_virksomhed_reklame_beskyttelse
ON admin_virksomhed_reklame_beskyttelse.CVREnhedsId = admin_virksomhed.id
[snippet_bitemp_virkning_cvr(admin_virksomhed_reklame_beskyttelse)]

LEFT JOIN CVR.Navn admin_virksomhed_navn
ON admin_virksomhed_navn.CVREnhedsId = admin_virksomhed.id
[snippet_bitemp_virkning_cvr(admin_virksomhed_navn)]

LEFT JOIN CVR.Adressering admin_virksomhed_cvr_adresse
ON admin_virksomhed_cvr_adresse.CVREnhedsId = admin_virksomhed.id
   AND admin_virksomhed_cvr_adresse.AdresseringAnvendelse = 'beliggenhedsadresse'
[snippet_bitemp_virkning_cvr(admin_virksomhed_cvr_adresse)]
```

```
{
  EJF_Ejerskab() {
    nodes {
      administrerendeVirksomhedCVRNr_20_Virksomhed_CVRNummer_ref {
        id_CVR_Reklamebeskyttelse_CVREnhedsId_ref {
        }

        id_CVR_Navn_CVREnhedsId_ref {
          nodes {
          }
        }

        id_CVR_Adressering_CVREnhedsId_ref(
          where: {
            AdresseringAnvendelse: { eq: "beliggenhedsadresse" }
          }
        ) {
          nodes {
          }
        }
      }
    }
  }
}
```



4.4.6 6th section of joins: For administrerende produktionsenhed

```
-- Joins for administrerende produktionsenhed
LEFT JOIN CVR.Produktionsenhed admin_penhed
ON admin_penhed.pNummer = ejerskab.produktionsenhedPNr
[snippet_bitemp_virkning_cvr(admin_penhed)]

LEFT JOIN CVR.Reklamebeskyttelse admin_penhed_reklame_beskyttelse
ON admin_penhed_reklame_beskyttelse.CVREnhedsId = admin_penhed.id
[snippet_bitemp_virkning_cvr(admin_penhed_reklame_beskyttelse)]

LEFT JOIN CVR.Navn admin_penhed_navn
ON admin_penhed_navn.CVREnhedsId = admin_penhed.id
[snippet_bitemp_virkning_cvr(admin_penhed_navn)]

LEFT JOIN CVR.Adressering admin_penhed_cvr_adresse
ON admin_penhed_cvr_adresse.CVREnhedsId = admin_penhed.id
   AND admin_penhed_cvr_adresse.AdresseringAnvendelse = 'beliggenhedsadresse'
[snippet_bitemp_virkning_cvr(admin_penhed_cvr_adresse)]
```

```
{
  EJF_Ejerskab() {
    nodes {
      produktionsenhedPNr_20_Produktionsenhed_pNummer_ref {
        id_CVR_Reklamebeskyttelse_CVREnhedsId_ref {

        }

        id_CVR_Navn_CVREnhedsId_ref {
          nodes {

          }
        }

        id_CVR_Adressering_CVREnhedsId_ref(
          where: {
            AdresseringAnvendelse: { eq: "beliggenhedsadresse" }
          }
        ) {
          nodes {

          }
        }
      }
    }
  }
}
```



4.4.7 7th section of joins: For administrerende person

```
-- Joins for administrerende person
LEFT JOIN CPR.Personnummer admin_person_personnumre
ON admin_person_personnumre.personnummer = ejerskab.administrerendePersonPersonNr
[snippet_bitemp_virk(admin_person_personnumre)]

LEFT JOIN CPR.Person admin_person
ON admin_person.id = admin_person_personnumre.personid [snippet_bitemp_virkning_cpr(person)]

LEFT JOIN CPR.Beskyttelse admin_person_beskyttelse
ON admin_person_beskyttelse.personid = admin_person.id
[snippet_bitemp_virk(admin_person_beskyttelse)]

LEFT JOIN CPR.Navn admin_person_navn
ON admin_person_navn.personid = admin_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, administrerendePersonPersonNr)])
[snippet_bitemp_virk(admin_person_navn)]

LEFT JOIN CPR.CprAdresse admin_person_cpr_adresse
ON admin_person_cpr_adresse.personid = admin_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, administrerendePersonPersonNr)])
[snippet_bitemp_virk(admin_person_cpr_adresse)]

LEFT JOIN CPR.UdrejseIndrejse admin_person_ud_ind_rejse
ON admin_person_ud_ind_rejse.personid = admin_person.id
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, administrerendePersonPersonNr)])
[snippet_bitemp_virk(admin_person_ud_ind_rejse)]

LEFT JOIN DAR.Adresse admin_person_dar_adresse
ON admin_person_dar_adresse.id_lokalId = admin_person_cpr_adresse.daradresse
    AND NOT EXISTS ([snippet_cpr_beskyttet(ejerskab, administrerendePersonPersonNr)])
[snippet_bitemp_virkning(admin_person_dar_adresse)]
```



```
{
  EJF_Ejerskab() {
    nodes {
      administrerendePerson { # Uses the EJFCustom Person-entity
        beskyttelser {
        }
        navn {
        }
        cprAdresse {
        }
        udrejseIndrejse {
        }
      }
    }
  }
}
```

NOTE: The LEFT JOIN DAR.Adresse admin_person_dar_adresse has the same limitation as 4.4.3 - it would require a separate query using the daradresse field from the full CPR address data.

4.5 Step 4: Analyse filters

The first filtering that we should consider is the bitemporal filtering that is required on the GraphQL-services.

It has been determined that the most suitable GraphQL-service for this REST-service is the flexibleCurrent service. For this GraphQL-service it is required to have a filter on effective time (virkningstid) using the virkningstid parameter. This should be added to all the queries. As an example, we will look at the query from the first join section. The filter should be applied as follows to all queries (the date can be substituted with another, but should follow the same format):

```
{
  EJF_Ejerskab(
    virkningstid: "2025-01-01T01:01:01Z"
  ) {
    nodes {
      oplysningerEjesAfEjerskab {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```



Besides this mandatory filtering, we need to consider any filtering that is done in the REST-service. In the REST-service documentation overview (under input parameters heading) we can along read the information presented in Table 6.

Name	Type	Description (Danish, from Confluence)	Default value	Mandatory parameters (Danish, from Confluence)
Status	String	Status jf. grunddatamodellens udfaldsrum for EjerfortegnelsestatusVærdi*	NULL	Enten BFE-nummer (BFEnr) eller ejerskabsid (Ejerskabsid).
Ejerskabsid	String	Lokalid for et ejerskab	NULL	
Virkningstid	DateTime	Dato som falder indenfor en virkningstidsperiode. Anvendes fx 2015-09-02 vil de instanser med en gældende virkningstid på denne dato udvælges	NOW()	
BFEnr	String	Bestemt Fast Ejendom nummer	NULL	

*Information about specific types (such as the enumeration EjerfortegnelsestatusVærdi) is found at: <https://grunddatamodel.datafordeler.dk/objekttypekatalog/>

Table 6: Input parameters

It leaves us with a choice to make. Note that the REST-service requires the user to include one of the parameters “BFE-nummer” or “Ejerskabsid”, and utilize it in the “where”-statement below. For the sake of the example, let us include BFEnr and status as parameters. This, however, is not required by the GraphQL-service.

```
-- Conditions for input parametre
WHERE (@BFEnr IS NULL OR ejerskab.bestemtFastEjendomBFEnr = @BFEnr)
AND (@Ejerskabsid IS NULL OR ejerskab.id_lokalId = @Ejerskabsid)
AND (@Status IS NULL OR (
    ejerskab.status = @Status
    AND (ejende_pv.status IS NULL OR ejende_pv.status = @Status)
    AND (ejende_pv_alt_adresse.status IS NULL OR ejende_pv_alt_adresse.status = @Status)
    AND (admin_pv.status IS NULL OR admin_pv.status = @Status)
    AND (admin_pv_alt_adresse.status IS NULL OR admin_pv_alt_adresse.status = @Status)))
```

Given our choices of input parameters and the input statement we can deduct the following:

1. There are two filters on entity ‘Ejerskab’. One on the bestemtFastEjendomBFEnr field, and one on status field.
2. There is a filter on entity ‘PersonVirksomhedsoplysninger’ on the status field.
 - a. It is applied on both joins where ‘PersonViksomhedsoplysninger’ is included
 - b. Can be deduced from the use of aliases.
3. There is one filter on the entity ‘AlternativAdresse’, on the status field.
 - a. It is applied on both joins where ‘AlternativAdresse’ is included.
 - b. Can be deduced from the use of aliases.



We will go through the filters, one by one in the coming subsections:

- 4.5.1 1st and 2nd filter: Ejerskab
- 4.5.2 3rd filter: PersonVirksomhedsoplysninger for ejende join
- 4.5.3 4th filter: AlternativAdresse for ejende join
- 4.5.4 5th filter: PersonVirksomhedsoplysninger for administrerende join
- 4.5.5 6th filter: AlternativAdresse for administrerende join

4.5.1 1st and 2nd filter: Ejerskab

The first two filters are placed on Ejerskab, which is the root-entity. The filter therefore needs to be applied to all queries, since the root entity is the foundation for all of them. As an example, we will look at the query from the first join section. The filter should be applied as follows to all queries (where, 'SomeBestemtFastEjendomBfeNr' is the chosen input for bestemtFastEjendomBfeNr parameter and 'gældende' is the chosen input for status parameter).

```
{
  EJP_Ejerskab(
    virkningstid: "2025-01-01T01:01:01Z"
    where: {
      bestemtFastEjendomBfeNr: { eq: "SomeBestemtFastEjendomBfeNr" }
      status: { eq: "gældende" }
    }
  ) {
    nodes {
      oplysningerEjesAfEjerskab {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```

4.5.2 3rd filter: PersonVirksomhedsoplysninger for ejende join

The third filter is placed on the entity ' PersonVirksomhedsoplysninger' in the join found in section 4.4.1 "1st section of joins: For ejende person-/virksomhedsoplysning". It is applied as follows



```
{
  EJF_Ejerskab(
    virkningstid: "2025-01-01T01:01:01Z"
    where: {
      bestemtFastEjendomBFENr: { eq: "SomeBestemtFastEjendomBfeNr" }
      status: { eq: "gældende" }
    }
  ) {
    nodes {
      oplysningerEjesAfEjerskab (
        where: { status: { eq: "gældende" } }
      ) {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```

4.5.3 4th filter: AlternativAdresse for ejende join

The fourth filter is placed on the entity 'AlternativAdresse' in the join found in section 4.4.1 "1st section of joins: For ejende person-/virksomhedsoplysning". It is applied as follows:

```
{
  EJF_Ejerskab(
    virkningstid: "2025-01-01T01:01:01Z"
    where: {
      bestemtFastEjendomBFENr: { eq: "SomeBestemtFastEjendomBfeNr" }
      status: { eq: "gældende" }
    }
  ) {
    nodes {
      oplysningerEjesAfEjerskab (
        where: { status: { eq: "gældende" } }
      ) {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref (
          where: { status: { eq: "gældende" } }
        ) {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```

4.5.4 5th filter: PersonVirksomhedsoplysninger for administrerende join

The fifth filter is placed on the entity 'PersonVirksomhedsoplysninger' in the join found in section 4.4.4 "4th section of joins: For administrerende person-/virksomhedsoplysning". It is applied as follows



```
{
  EJF_Ejerskab() {
    nodes {
      ejerskabAdministreresAfPersonEllerVirksomhedsoplysninger {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
        }
        personEllerVirksomhedsoplysningerharAdresse {
          where: { status: { eq: "gældende" } }
        } {
        }
      }
    }
  }
}
```

4.5.5 6th filter: AlternativAdresse for administrerende join

The sixth filter is placed on the entity 'AlternativAdresse' in the join in the join found in section 4.4.4 "4th section of joins: For administrerende person-/virksomhedsoplysning". It is applied as follows:

```
{
  EJF_Ejerskab() {
    nodes {
      ejerskabAdministreresAfPersonEllerVirksomhedsoplysninger {
        alternativAdresseLokalId_23_AlternativAdresse_id_lokalId_ref {
          where: { status: { eq: "gældende" } }
        } {
        }
        personEllerVirksomhedsoplysningerharAdresse {
        }
      }
    }
  }
}
```

4.6 Other filters – snippets

The REST-services use filtering snippets to handle complex logic for data protection, temporal constraints, and relationship validation. The GraphQL-services simplify this approach by replacing these snippets with query parameters and built-in schema functionality. The SQL snippets are found in the DLS, as also described in section 2.1.2.

4.7 Step 5: Determine fields to fetch

The very last thing to fill out in our queries is the fields that we wish to fetch. We look, once again, to the REST-service documentation and at the initial "SELECT/FROM" statement.

```
SELECT *
FROM Ejerskab ejerskab
```

Here, we can see that the *-syntax is utilized. This means that all fields should be fetched in our queries. This is both for the root entity and all the joined entites. However, to find the fields to fetch, one should find the JSON-schema for the output of the REST-service to find the corresponding fields. To find what fields are



available to fetch, we refer to the GraphQL-service documentation (schema). For each entity in our query we find the type definition in the schema and add all fields available (with exception to join field). This results in the following queries.

4.8 Step 6: Verify result

This section demonstrates how the REST API and GraphQL responses can be compared for verification. The verification uses the GraphQL query and pseudo SQL from section 4.4.7 as an example to demonstrate that the complex SQL joins and filtering logic have been successfully translated to GraphQL relationship traversals. When filling out the required "virkningstid"-field as well as other fields, the query will look like the following:

```
{
  EJF_Ejerskab(virkningstid: "2024-01-01T00:00:00Z") {
    nodes {
      administrerendePerson {
        beskyttelser {
          beskyttelsestype
        }

        navn {
          navn
        }

        cprAdresse {
          vejadresseringsnavn
          husnummer
          etage
          sidedoer
          postnummer
          postdistrikt
        }

        udrejseIndrejse {
          udlandsadresselinje1
          udlandsadresselinje2
          udlandsadresselinje3
        }
      }
    }
  }
}
```

While the response structures differ due to REST's GeoJSON format versus GraphQL's hierarchical approach, the underlying business data should be identical, demonstrating that GraphQL can fully replace the SQL-based implementation without data loss or functional regression.

This example has the following verification points:

- "virkningstid"-parameter applies same temporal logic as SQL snippets
- All 6 accessible SQL LEFT JOINS successfully replaced by nested GraphQL relationships
- The rest is handled automatically by the implementation.



4.8.1 REST API response

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "id_lokalId": "ejerskab-12345",
        "administrerendePerson": {
          "navn": "Jens Jensen",
          "personnummer": "2345678901",
          "beskyttelser": ["reklamebeskyttelse"],
          "cprAdresse": {
            "vejadresseringsnavn": "ABC Gade",
            "husnummer": "15",
            "etage": "2",
            "sidedoer": "th",
            "postnummer": "1337",
            "postdistrikt": "ABC By"
          },
          "udrejseIndrejse": {
            "udlandsadresselinje1": "123 Main Street",
            "udlandsadresselinje2": "Apartment 4B",
            "udlandsadresselinje3": "New York, NY 10001"
          }
        }
      }
    }
  ]
}
```



4.8.2 GraphQL query response

```
{
  "data": {
    "EJF_Ejerskab": {
      "nodes": [
        {
          "administrerendePerson": {
            "beskyttelser": [
              {
                "beskyttelsestype": "reklamebeskyttelse"
              }
            ],
            "navn": {
              "navn": "Jens Jensen"
            },
            "cprAdresse": {
              "vejadresseringsnavn": "ABC Gade",
              "husnummer": "15",
              "etage": "2",
              "sidedoer": "th",
              "postnummer": "1337",
              "postdistrikt": "ABC By"
            },
            "udrejseIndrejse": {
              "udlandsadresselinje1": "123 Main Street",
              "udlandsadresselinje2": "Apartment 4B",
              "udlandsadresselinje3": "New York, NY 10001"
            }
          }
        }
      ]
    }
  }
}
```