1. Overview

The DLT API-Service is backed by a Microsoft SQL Service database referred to as ODS/HDS which is maintain as its own project separate to the API-service with its own release cycle.

1.1. References:

1.1.1. Learn Graphql

- https://graphql.org/learn/

1.1.2. Apollo Guideline

- https://www.apollographql.com/docs/apollo-server/schema/schema

1.1.3. Graphql Visually Explained

- https://www.apollographql.com/blog/the-concepts-of-graphql-bc68bd819be3/

1.1.4. Github Graphql Explorer

- https://docs.github.com/en/graphql/overview/about-the-graphql-api
- https://docs.github.com/en/graphql/overview/explorer

1.1.5. Github At Netflix

- https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2
- https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-2-bbe71aaec44a
- https://www.infoq.com/news/2021/02/netflix-graphql-spring-boot/

2. About

2.1. Why use GraphQL

Advantaged	Disadvantages

1. Decouple the relationship between client, server and datastore structures.
2. Allow clients to request data in a nested format that is suitable to their needs using dynamic queries.
3. Allow the GraphQL schema to abstract the database structure
4. Only request and process exactly what is needed no over/under fetching
   a. Request only the native / calculated fields required
   b. Request only the relationships wanted
   c. Access the type graph from any type
5. Natively supports reactive queries via subscriptions
	1. Datasource structure can be difficult to optimize due to dynamic query requirements
	2. In most cases requires moving the joining/relationship logic from the datasource to GraphQL resolvers
	3. N+1 database reads if not careful to use dataloaders and query context
	4. Will generally be less performant that a typical flattened RPC type API backed by explicit tables or views
	5. Error handlining is response content rather than http response code based

2.1.1. When to use GraphQL

When the format and usage requirements of the domain data that will be exposed by the GraphQL API are:

- Are unknown
- Are well known but vary or contradict one structured API
    - client A want all X for Y where X.Z
    - client B want all Y for X where X.Z & Y.T
- You are building a reactive client
- Your data is a natural fit for a graph or tree structure
- You want to avoid the N+1 pitfalls of RESTful APIs and get everything in one transaction.

3. Guidelines

3.1. Naming convention

- **Field names** should use camelCase. Many GraphQL clients are written in JavaScript, Java, Kotlin, or Swift, all of which recommend camelCase for variable names.
- **Type names** should use PascalCase. This matches how classes are defined in the languages mentioned above.
- **Enum names** should use PascalCase.
- **Enum values** should use ALL_CAPS, because they are similar to constants.

3.2. Do's and Don'ts

Some of these concepts will be explored in more detail bellow.

> ⚠ A mixture of SQL view driven resolvers and nested resolvers may be used in the DLT api-service, this is due to the complexity of the underlying database and the ledger it is projecting.

<table>
<tr><th></th><th>Do</th><th>Don't</th></tr>
<tr><td>1</td><td>Get everything you need in one query</td><td>Make multiple specific queries</td></tr>
<tr><td>2</td><td>Build a generalized graph of basic types and nested relationship between them</td><td>Build a graph that is aware of how a client will use it specific to that use case</td></tr>
<tr><td>3</td><td>Use basic type relationships and filter to access complex relationships<pre><code># Get actor roles where type is
"registryRelationship"
Get actors for role where actor id is "1234"
Get securities where actor is issuer of
security by code "ABC"

ActorRoles(roleType: "registryRelationship") {
     Actors(id: "1234") {
         Securities(code: "ABC", isin:
"B00001") {
             isIssuedByActor
             isLatestSecurityCode
             #...
         }
     }
}</code></pre></td><td>Make query resolvers for specific data if you can use a nested query<pre><code>issuerRegistrySecurityBySecurityCode
(securityCode: "ABC")</code></pre>Note: This one may be difficult for us while using views created specifically for each report request.</td></tr>
</table>

<table>
<tr><td>4</td><td>Use InputTypes to group common input arguments into a type

```
input SomeInput {
        a:String
    b:String
    c:String
    d:String
    e:String
}

type Query {
        ATypes(input:SomeInput): [SomeType]
    BTypes(input:SomeInput): [SomeType]
    CTypes(input:SomeInput): [SomeType]
    DTypes(input:SomeInput,f:Int): [SomeType]
}
```

</td><td>Write huge and repeated argument lists

```
type Query {
        ATypes(a:String,b:String,c:String,d:
String,e:String): [AType]
    BTypes(a:String,b:String,c:String,d:String,e:
String): [BType]
    CTypes(a:String,b:String,c:String,d:String,e:
String): [CType]
    DTypes(a:String,b:String,c:String,d:String,e:
String,f:Int): [DType]
}
```

</td></tr>
<tr><td>5</td><td>Use ID to expose a globally unique identifier for a type instance for caching and paging</td><td>Have no ID or unique identifier for results</td></tr>
<tr><td>6</td><td>Use fragments in queries to define a common set of fields for a type</td><td>Repeat fields in queries if it can be avoided</td></tr>
<tr><td>7</td><td>Use fragments to up-cast generalized results from a query that returns multiple types:
<table>
<tr><th>Query</th><th>Results</th></tr>
<tr><td>

```
{
  search(text:
"an") {
    __typename
    ... on Human
{
      name
    }
    ... on Droid
{
      name
    }
    ... on
Starship {
      name
    }
  }
}
```

</td><td>

```
{
  "data": {
    "search": [
      {
        "__typename":
"Human",
        "name": "Han Solo"
      },
      {
        "__typename":
"Human",
        "name": "Leia
Organa"
      },
      {
        "__typename":
"Starship",
        "name": "TIE
Advanced x1"
      }
    ]
  }
}
```

</td></tr>
</table>
</td><td>Make multiple query resolvers for returning different types for the same query function.</td></tr>
<tr><td>8</td><td>Use QueryFilter input (see below)

```
Users(fiter: UserFilterInput)
```

</td><td>Use ambiguous arguments to filter a collection

```
Users(name: String, age: Int, createdAt: {
before: LocalDate, after: LocalDate })
```

</td></tr>
<tr><td>9</td><td>Wrap collection results in a Results Wrapper that contain results and paging metadata</td><td>Not expose paging metadata to client</td></tr>
<tr><td>10</td><td>Use default paging on all collection results, and a PageFilter argument to allow the client to control</td><td>Return all results by default this could lead to performance issues or timeouts</td></tr>
<tr><td>11</td><td>Use predictable default ordering on all collection results, and an OrderBy argument to allow the client to control</td><td>Return results in inconsistent or unpredictable order</td></tr>
</table>

12	Use LocalDate and LocalDateTime scalars for date string, let spring do the hard work for you 😎	Use string for dates and time input or ourput

3.3. Query Resolvers

In general there should be at most two query resolvers for a type, avoid making resolvers for specific filters of a collection type, we will use input types for this.

Types can contain their own query resolvers for related tyeps also, this will be shown later.

Querying

```
schema {
    query: Query
}

# An object with a Globally Unique ID
interface Node {
  # The ID of the object.
  id: ID!
}

type User implements Node {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
}

"Query root"
type Query {
        UserById(id: ID!): User
        Users: [User]
}

"Built-in scalar representing a local date"
scalar LocalDate
```

3.4. Filtering a collection resolver

By using a UserFilterInput object the client can describe how they want to filter the results dynamically using the same type resolver.

Filtering Collections

```
schema {
    query: Query
}

enum FilterOperation {
        ISNULL
        NOTULL
        EQ
    NEQ
        LT
        LTE
        GT
        GTE
        IN
        CN # Contains
        SW # Starts With
        EW # Ends With
}

type User {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
}

# For simplicity using <T> there will be a type for each T
input FieldFilter_<T> {
  op: FilterOperation
  value: T
  and: [FieldFilter_<T>!]
  or: [FieldFilter_<T>!]
}

input UserFilterInput {
        id: FieldFilter__String
        username: FieldFilter__String
        age: FieldFilter__Int
        createdAt: FieldFilter__LocalDate
}

"Query root"
type Query {
        UserById(id: ID!): User
        Users(fiter: UserFilterInput): [User]
}

"Built-in scalar representing a local date"
scalar LocalDate
```

3.5. Paging a collection resolver

For large collection, cursor based paging is the obvious choice and is what is recommended.

Due to the limitation of the database structure cursor based paging is not possible for us, so we will have to use offset based paging. This is not a big issue due to the small-ish nature of the collections.

The results should always be limited, if not provided use these rules should be used or a variation of this appropriate to the type:

1. offset
    a. >= 0
    b. default to 0
2. limit
    a. >= 0
    b. default to 10
    c. upper limit 50?

Paging Collections

```
schema {
    query: Query
}

type User {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
}

enum FilterOperation {
        ISNULL
        NOTULL
        EQ
    NEQ
        LT
        LTE
        GT
        GTE
        IN
        CN # Contains
        SW # Starts With
        EW # Ends With
}

# For simplicity using <T> there will be a type for each T
input FieldFilter_<T> {
  op: FilterOperation
  value: T
  and: [FieldFilter_<T>!]
  or: [FieldFilter_<T>!]
}

input UserFilterInput {
        id: FieldFilter__String
        username: FieldFilter__String
        age: FieldFilter__Int
        createdAt: FieldFilter__LocalDate
}

input PageFilter {
        offset: Int
        limit: Int
}

"Query root"
type Query {
        UserById(id: ID!): User
        Users(fiter: UserFilterInput, page: PageFilter): [User]
}

"Built-in scalar representing a local date"
scalar LocalDate
```

3.6. Ordering a collection resolver

Reliable ordering of paged and filtered result can improve client performance.

Each resolver should have a default order for when the client does not supply one, this is typically the primary key GUID or a date filed of the type

Ording Collections

```
schema {
    query: Query
}

type User {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
}

enum FilterOperation {
        ISNULL
        NOTULL
        EQ
    NEQ
        LT
        LTE
        GT
        GTE
        IN
        CN # Contains
        SW # Starts With
        EW # Ends With
}

# For simplicity using <T> there will be a type for each T
input FieldFilter_<T> {
  op: FilterOperation
  value: T
  and: [FieldFilter_<T>!]
  or: [FieldFilter_<T>!]
}

input UserFilterInput {
        id: FieldFilter__String
        username: FieldFilter__String
        age: FieldFilter__Int
        createdAt: FieldFilter__LocalDate
}

input PageFilter {
        offset: Int
        limit: Int
}

enum OrderDirection {
        ASC
        DESC
}

input OrderBy {
        dir: OrderDirection # default to ASC
        field: String!
}

"Query root"
type Query {
        UserById(id: ID!): User
        Users(fiter: UserFilterInput, page: PageFilter, order: [OrderBy]): [User]
}

"Built-in scalar representing a local date"
scalar LocalDate
```

3.7. Paged Collection Metadata

If possible use cursor based edge node pattern: https://graphql.org/learn/pagination/

Often a client will want to know more about the paged results than just the results.

Like:

- Is there more data?
- total items to page
- total pages
- paging cursor

To make this possible the collection result need to be wrapped in another type, this is typically called, an collection/edge/connection/relationship depending on where it is in the query, and functions much like a bridge table would between two relational tables.

Paged Collection Metadata

```
schema {
    query: Query
}

interface Node {
  id: ID!
}

type User implements Node {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
}

type PageInfo {
    totalItems: Int!
    hasNextPage: Boolean!
          hasPreviousPage: Boolean!
}

type Collection_User {
    pageInfo: PageInfo!
    nodes: [User]!
}

enum FilterOperation {
        ISNULL
        NOTULL
        EQ
    NEQ
        LT
        LTE
        GT
        GTE
        IN
        CN # Contains
        SW # Starts With
        EW # Ends With
}

# For simplicity using <T> there will be a type for each T
input FieldFilter_<T> {
  op: FilterOperation
  value: T
  and: [FieldFilter_<T>!]
  or: [FieldFilter_<T>!]
}

input UserFilterInput {
        id: FieldFilter__String
        username: FieldFilter__String
        age: FieldFilter__Int
        createdAt: FieldFilter__LocalDate
}

input PageFilter {
        offset: Int
        limit: Int
}

enum OrderDirection {
        ASC
```

```
        DESC
}

input OrderBy {
        dir: OrderDirection # default to ASC
        field: String!
}

"Query root"
type Query {
        UserById(id: ID!): User
        Users(fiter: UserFilterInput, page: PageFilter, order: [OrderBy]): Collection_User
}

"Built-in scalar representing a local date"
scalar LocalDate
```

3.8. Edges Connections and Relationships

Here we have modified our schema to model employers and employees relationship. Employers have many Employees and Employees have many Employers.

Particularly in the case of many to many relationships, there is often data that is stored on the relationship that needs to be accessible for filtering and viewing.

Employees Employers

```
schema {
    query: Query
}

"The base of every data type"
interface Node {
  id: ID!
}

"Relationship data relevent to node type"
interface Edge {
        node: Node!
}

"Collection wrapper for a type relationship query"
interface Relationship {
        edges: [Edge]!
        pageInfo: PageInfo!
}

"Collection wrapper for a top level query"
interface Collection {
        nodes: [Node]!
        pageInfo: PageInfo!
}

"Pageing metadata type"
type PageInfo {
    totalItems: Int!
    hasNextPage: Boolean!
          hasPreviousPage: Boolean!
}

# Employee

type Employee implements Node {
        id: ID!
        username: String!
        age: Int
        createdAt: LocalDate
        Employers(
        fiter: EmployeesEmployersRelationshipFilterInput,
        employersFiter: EmployersFilterInput,
        page: PageFilter,
        order: [OrderBy]
    ): EmployeeEmployerRelationship!
}
```

```
type EmployeeCollection  implements Collection {
    nodes: [Employee!]!
    pageInfo: PageInfo!
}

# Employer

type Employer implements Node {
        id: ID!
    name: String!
        Employees(
        fiter: EmployeesEmployersRelationshipFilterInput,
        employeesFiter: EmployeesFilterInput,
        page: PageFilter,
        order: [OrderBy]
    ): EmployerEmployeeRelationship!
}

type EmployerCollection  implements Collection {
    nodes: [Employer!]!
    pageInfo: PageInfo!
}

# Employee -> Employer Relationship

type EmployeeEmployerEdge implements Edge {
    isCurrent: Boolean!
    startDate: LocalDate!
    endDate: LocalDate
    role: String
    wage: Int
    node: Employer!
}

type EmployeeEmployerRelationship implements Relationship {
        edges: [EmployeeEmployerEdge!]!
        pageInfo: PageInfo!
}

# Employee <- Employer Relationship

type EmployerEmployeeEdge implements Edge {
    isCurrent: Boolean!
    startDate: LocalDate!
    endDate: LocalDate
    role: String
    wage: Int
    node: Employee!
}

type EmployerEmployeeRelationship implements Relationship {
        edges: [EmployerEmployeeEdge!]!
        pageInfo: PageInfo!
}

# Quereies

"Query root"
type Query {
        EmployeeById(id: ID!): Employee
        Employees(fiter: EmployeesFilterInput, page: PageFilter, order: [OrderBy]): EmployeeCollection
    EmployerById(id: ID!): Employer
        Employers(fiter: EmployersFilterInput, page: PageFilter, order: [OrderBy]): EmployerCollection
}

# Filtering Input

enum FilterOperation {
        ISNULL
        NOTULL
        EQ
    NEQ
        LT
        LTE
        GT
```

```
        GTE
        IN
        CN # Contains
        SW # Starts With
        EW # Ends With
}

input FieldFilter_String {
  op: FilterOperation
  value: String
  and: [FieldFilter_String!]
  or: [FieldFilter_String!]
}

input FieldFilter_LocalDate {
  op: FilterOperation
  value: LocalDate
  and: [FieldFilter_LocalDate!]
  or: [FieldFilter_LocalDate!]
}

input FieldFilter_Int {
  op: FilterOperation
  value: Int
  and: [FieldFilter_Int!]
  or: [FieldFilter_Int!]
}

input FieldFilter_Boolean {
  op: FilterOperation
  value: Boolean
  and: [FieldFilter_Boolean!]
  or: [FieldFilter_Boolean!]
}

input EmployeesFilterInput {
        id: FieldFilter_String
        username: FieldFilter_String
        age: FieldFilter_Int
        createdAt: FieldFilter_LocalDate
}

input EmployersFilterInput {
        id: FieldFilter_String
        name: FieldFilter_String
}

input EmployeesEmployersRelationshipFilterInput {
    isCurrent: FieldFilter_Boolean
    startDate: FieldFilter_LocalDate
    endDate: FieldFilter_LocalDate
    role: FieldFilter_String
    wage: FieldFilter_Int
}

input PageFilter {
        offset: Int
        limit: Int
}

enum OrderDirection {
        ASC
        DESC
}

input OrderBy {
        dir: OrderDirection # default to ASC
        field: String!
}

"Built-in scalar representing a local date"
scalar LocalDate
```

Created using https://app.graphqleditor.com/

Example query

Query	Results

```
{
  EmployerById(id: "") {
              id
    name
    Employees(
      fiter: {role: {op:EQ, value:"
developer"}},
      employeesFiter: {age: {op:GT, value:18}}
      page: {limit: 3},
      order: [{field:"startDate"}]
    ){
      pageInfo {
        totalItems
      }
      edges {
        startDate
        endDate
        role
        wage
        node {
          id
          username
          age
          Employers {
            pageInfo {
              totalItems
            }
          }
        }
      }
    }
  }
}
```


```
{
  "data": {
    "EmployerById": {
      "Employees": {
        "edges": [
          {
            "endDate": "2014-08-20T08:31:12.555Z",
            "node": {
              "Employers": {
                "pageInfo": {
                  "totalItems": 14797
                }
              },
              "age": 72021,
              "id": "31704ab3-3d31-4f4c-b6a7-
71ad7db8d743",
              "username": "Antonia.Roob59"
            },
            "role": "developer",
            "startDate": "2014-08-20T08:31:12.555Z",
            "wage": 82000
          },
          {
            "endDate": "2018-05-21T07:06:37.354Z",
            "node": {
              "Employers": {
                "pageInfo": {
                  "totalItems": 52103
                }
              },
              "age": 29163,
              "id": "0087d068-adaf-4797-a8b0-
9a6134d4bad1",
              "username": "Haven_Weimann"
            },
            "role": "developer",
            "startDate": "2014-08-20T08:31:12.555Z",
            "wage": 39542
          },
          {
            "endDate": "2020-12-27T16:03:04.510Z",
            "node": {
              "Employers": {
                "pageInfo": {
                  "totalItems": 78805
                }
              },
              "age": 29974,
              "id": "7fba31a2-dda5-4bb8-9052-
bdc2329d00be",
              "username": "Noelia41"
            },
            "role": "developer",
            "startDate": "2014-08-20T08:31:12.555Z",
            "wage": 67075
          }
        ],
        "pageInfo": {
          "totalItems": 65703
        }
      },
      "id": "fb4b49ca-753f-41b1-8f46-88bb75660276",
      "name": "Mrs. Dena Dibbert"
    }
  }
}
```

4. Query Examples

Filter Order, and Page Input example

```
fragment cemvFields on CEMV {
    id
    issuerActorId
    accountId
    securityCode
    isin
    corporateActionEventId
    exDate
    apirCode
    bomType
    businessDate
    closingBalance
    netOff
    netOn
    regOff
    regOn
}

query {
  cemv(
        order: [
      { field:"businessDate", dir:DESC },
      { field:"accountId" }
    ],
        page: { offset:0, limit:5 },
    filter: {
      actorId: { op: EQ, value: "55003" }
      businessDate: {
        and: [
          { op: GTE, value: "2021-01-01" }
          { op: LTE, value: "2021-01-02" }
        ]
      },
      accountId: {
        or: [
          { op: ISNULL }
          { op: NEQ, value: "143" }
        ]
      }
      securityCode: { op: IN, value: "AAA,BBB"] }
    }
  ) {
        ...cemvFields
  }
}
```

4.1. Holding Balance Sub Positions

POC: https://stash.asx.com.au/projects/DLT/repos/reporting_apis/pull-requests/123/overview

Here we can see how a nested query might be used to grab a much relevant data as possible given attributes from a 601 workitem request. Each nested type can be filtered, ordered and paged

- Get actor by Id
    - get actor's roles
    - get actor's accounts
        - get account holdings
            - get holding security
        - get holding's subpositions

Holding should have enough context to not include its related security if not needed. eg finding a holding of a particular security code.

In another query in the same request you might query securities by aissuer, code and isin to find out if the actor is the current issuer. if the account holdings security match the query you know there is an error in the workitem request.

```
fragment actorFields on Actor {
  actorId
  actorStatus
  actorParty
  archivedAt
  bic
  entityId
  eventId
}

fragment roleFields on ActorRole {
  roleId
  roleType
  roleStatus
  archivedAt
}

fragment accountFields on Account {
  accountId
  accountName
  accountStatus
  archivedAt
}

fragment holdingBalanceFields on HoldingBalance {
  securityId
  totalBalance
  availableBalance
  archivedAt
}

fragment subpositionFields on HSBL {
  subpositionType
  unitQuantity
  corporateActionEventId
  optionType
  optionNumber
}

fragment securityFields on Security {
  apirCode
  securityCode
  isin
}

query {
  actorById(actorId: "20004"){
    ...actorFields
    roles(filter: { roleType: { op: EQ, value: "AccountCreator" }}) {...roleFields}
    accounts(filter: { accountId: { op: EQ, value: "0000000086" }})  {
      ...accountFields
      holdings(filter: { securityId: { op: EQ, value: "000008" }}) {
        ...holdingBalanceFields
        security{...securityFields}
        subpositions(filter: { lockType: { op: EQ, value: "SETL" }}) {
          ...subpositionFields
        }
      }
    }
  }
}
```