



Adding Custom Logic

Contents

- Lesson Overview (#_lesson_overview)
- Codesandbox Setup (#_codesandbox_setup)
- The @cypher GraphQL Schema Directive (#_the_cypher_graphql_schema_directive)
- Custom Resolvers (#_custom_resolvers)
- Exercise: Exploring The @cypher Directive (#_exercise_exploring_the_cypher_directive)
- Check Your Understanding (#_check_your_understanding)
- Summary (#_summary)

Lesson Overview

In the previous lesson we explored building a GraphQL API using the Neo4j GraphQL Library that implemented basic create, read, update, and delete (CRUD) functionality. In this lesson we explore adding custom logic to our GraphQL API using the power of the Cypher query language.

Codesandbox Setup

To begin we'll clear out our Neo4j database, open a new Codesandbox, and run a GraphQL mutation to add some initial data.

First, open Neo4j Browser for your Neo4j Sandbox instance and run the following Cypher query to delete all the data in the database

Make sure you're running this command in the correct database as this will delete all data in the Neo4j database!

Cypher

Copy to Clipboard

```
MATCH (a) DETACH DELETE a
```

Need help? [Ask in the Neo4j Community](#) ↗

Next, open [this Codesandbox](#) and edit the `.env` file, adding the connection credentials for your Neo4j Sandbox. Refer to the setup instructions in the previous lesson if necessary.

In GraphQL Playground running in your Codesandbox, paste and execute the following GraphQL mutation to load the initial data we'll be working with:

GraphQL

Copy to Clipboard

```
mutation {
  createBooks(
    input: [
      {
        isbn: "1492047686"
        title: "Graph Algorithms"
        price: 37.48
        description: "Practical Examples in Apache Spark and Neo4j"
        subjects: { create: [{ name: "Graph theory" }, { name: "Neo4j" }] }
        authors: {
          create: [{ name: "Mark Needham" }, { name: "Amy E. Hodler" }]
        }
      }
      {
        isbn: "1119387507"
        title: "Inspired"
        price: 21.38
        description: "How to Create Tech Products Customers Love"
        subjects: {
          create: [{ name: "Product management" }, { name: "Design" }]
        }
        authors: { create: { name: "Marty Cagan" } }
      }
      {
        isbn: "190962151X"
        title: "Ross Poldark"
        price: 15.52
        description: "Ross Poldark is the first novel in Winston Graham's sweeping saga of
Cornish life in the eighteenth century."
        subjects: {
          create: [{ name: "Historical fiction" }, { name: "Cornwall" }]
        }
        authors: { create: { name: "Winston Graham" } }
      }
    ]
  ) {
    books {
      title
    }
  }

  createCustomers(
    input: [
```

Need help? [Ask in the Neo4j Community](#)

```
{
  username: "EmilEifrem7474"
  reviews: {
    create: {
      rating: 5
      text: "Best overview of graph data science!"
      book: { connect: { where: { isbn: "1492047686" } } }
    }
  }
  orders: {
    create: {
      books: { connect: { where: { title: "Graph Algorithms" } } }
      shipTo: {
        create: {
          address: "111 E 5th Ave, San Mateo, CA 94401"
          location: {
            latitude: 37.5635980790
            longitude: -122.322243272725
          }
        }
      }
    }
  }
}
{
  username: "BookLover123"
  reviews: {
    create: [
      {
        rating: 4
        text: "Beautiful depiction of Cornwall."
        book: { connect: { where: { isbn: "190962151X" } } }
      }
    ]
  }
  orders: {
    create: {
      books: {
        connect: [
          { where: { title: "Ross Poldark" } }
          { where: { isbn: "1119387507" } }
          { where: { isbn: "1492047686" } }
        ]
      }
    }
    shipTo: {
      create: {
        address: "Nordenskiöldsgatan 24, 211 19 Malmö, Sweden"
        location: { latitude: 55.6122270502, longitude: 12.99481772774 }
      }
    }
  }
}
]
```

Need help? [Ask in the Neo4j Community](#) ↗

```
) {  
  customers {  
    username  
  }  
}
```

With our initial data loaded let's dive into adding custom logic to our GraphQL API using Cypher!

The `@cypher` GraphQL Schema Directive

Schema directives are GraphQL's built-in extension mechanism and indicate that some custom logic will occur on the server. Schema directives are not exposed through GraphQL introspection and are therefore invisible to the client. The `@cypher` GraphQL schema directive allows for defining custom logic using Cypher in the GraphQL schema. Using the `@cypher` schema directive overrides field resolution and will execute the attached Cypher query to resolve the GraphQL field. Refer to the [@cypher directive documentation for more information.](#)

Computed Scalar Field

Let's look at an example of using the `@cypher` directive to define a computed scalar field in our GraphQL schema. Since each order can contain multiple books we need to compute the order "subtotal" or the sum of the price of each book in the order. To calculate the subtotal for an order with `orderID` "123" in Cypher we would write a query like this:

Cypher

```
MATCH (o:Order {orderID: "123"})-[:CONTAINS]->(b:Book)  
RETURN sum(b.price) AS subTotal
```

With the `@cypher` schema directive in the Neo4j GraphQL Library we can add a field `subTotal` to our `Order` type that includes the logic for traversing to the associated `Book` nodes and summing the price property value of each book. Here we use the `extend type` syntax of GraphQL SDL but we could also add this field directly to the `Order` type definition as well.

Add this extension to the `schema.graphql` file:

GraphQL

[Copy to Clipboard](#)

Need help? [Ask in the Neo4j Community.](#)

```
# schema.graphql

extend type Order {
  subTotal: Float @cypher(statement:"MATCH (this)-[:CONTAINS]->(b:Book) RETURN
sum(b.price)")
}
```

The `@cypher` directive takes a single argument `statement` which is the Cypher statement to be executed to resolve the field. This Cypher statement can reference the `this` variable which is the currently resolved node, in this case the currently resolved `Order` node.

We can now include this `subTotal` field in our GraphQL queries:

GraphQL

[Copy to Clipboard](#)

```
{
  orders {
    books {
      title
      price
    }
    subTotal
  }
}
```

JSON

```
{
  "data": {
    "orders": [
      {
        "books": [
          {
            "title": "Graph Algorithms",
            "price": 37.48
          }
        ],
        "subTotal": 37.48
      },
      {
        "books": [
          {
            "title": "Graph Algorithms",
            "price": 37.48
          },
          {
            "title": "Inspired",
```

Need help? [Ask in the Neo4j Community](#) ↗

```
        "price": 21.38
      },
      {
        "title": "Ross Poldark",
        "price": 15.52
      }
    ],
    "subTotal": 74.38
  }
}
}
```

The `@cypher` directive gives us all the power of Cypher, with the ability to express complex traversals, pattern matching, even leveraging Cypher procedures like APOC. Let's add a slightly more complex `@cypher` directive field to see what is possible. Let's say that the policy for computing the shipping cost of orders is to charge \$0.01 per km from our distribution warehouse. We can define this logic in Cypher, adding a `shippingCost` field to the `Order` type.

Add this extension to the `schema.graphql` file:

GraphQL

Copy to Clipboard

```
# schema.graphql

extend type Order {
  shippingCost: Float @cypher(statement: """
    MATCH (this)-[:SHIPS_TO]->(a:Address)
    RETURN round(0.01 * distance(a.location, Point({latitude: 40.7128, longitude: -74.0060})))
    / 1000, 2)
    """)
}
```

Node And Object Fields

In addition to scalar fields we can also use `@cypher` directive fields on object and object array fields with Cypher queries that return nodes or objects. Let's add a `recommended` field to the `Customer` type, returning books the customer might be interested in purchasing based on their order history and the order history of other customers in the graph.

Add this extension to the `schema.graphql` file:

GraphQL

Copy to Clipboard
Need help? [Ask in the Neo4j Community](#)

```
# schema.graphql

extend type Customer {
  recommended: [Book] @cypher(statement: """
    MATCH (this)-[:PLACED]->(:Order)-[:CONTAINS]->(:Book)<-[:CONTAINS]-(:Order)<-[:PLACED]-
    (c:Customer)
    MATCH (c)-[:PLACED]->(:Order)-[:CONTAINS]->(rec:Book)
    WHERE NOT EXISTS((this)-[:PLACED]->(:Order)-[:CONTAINS]->(rec))
    RETURN rec
  """)
}
```

Now we can use this **recommended** field on the **Customer** type. Since **recommended** is an array of **Book** objects we need to select the nested fields we want to be returned - in this case the **title** field.

GraphQL

[Copy to Clipboard](#)

```
{
  customers {
    username
    recommended {
      title
    }
  }
}
```

JSON

```
{
  "data": {
    "customers": [
      {
        "username": "EmilEifrem7474",
        "recommended": [
          {
            "title": "Inspired"
          },
          {
            "title": "Ross Poldark"
          }
        ]
      },
      {
        "username": "BookLover123",
        "recommended": []
      }
    ]
  }
}
```

Need help? [Ask in the Neo4j Community](#) ↗

```
}  
}
```

In this case we recommend two books to Emil that he hasn't purchased, however since BookLover123 has already purchased every book in our inventory we don't have any recommendations for them!

Any field arguments declared on a GraphQL field with a Cypher directive are passed through to the Cypher query as Cypher parameters. Let's say we want the client to be able to specify the number of recommendations returned. We'll add a field argument `limit` to the `recommended` field and reference that in our Cypher query as a Cypher parameter.

Modify this extension in the `schema.graphql` file:

GraphQL

[Copy to Clipboard](#)

```
# schema.graphql  
  
extend type Customer {  
  recommended(limit: Int = 3): [Book] @cypher(statement: """  
    MATCH (this)-[:PLACED]->(:Order)-[:CONTAINS]->(:Book)<-[:CONTAINS]-(:Order)<-[:PLACED]-  
(c:Customer)  
    MATCH (c)-[:PLACED]->(:Order)-[:CONTAINS]->(rec:Book)  
    WHERE NOT EXISTS((this)-[:PLACED]->(:Order)-[:CONTAINS]->(rec))  
    RETURN rec LIMIT $limit  
    """)  
}
```

We set a default value of 3 for this `limit` argument so that if the value isn't specified the `limit` Cypher parameter will still be passed to the Cypher query with a value of 3. The client can now specify the number of recommended books to return:

GraphQL

[Copy to Clipboard](#)

```
{  
  customers {  
    username  
    recommended(limit:1) {  
      title  
    }  
  }  
}
```

We can also return a map from our Cypher query when using the `@cypher` directive, an object or

object array GraphQL field. This is useful when we have multiple computed values we want to return or for returning data from an external data layer. Let's add weather data for the order addresses so our delivery drivers know what sort of conditions to expect. We'll query an external API to fetch this data using the `apoc.load.json` [↗](#) procedure.

First, we'll add a type to the GraphQL type definitions to represent this object (`Weather`), then we'll use the `apoc.load.json` procedure to fetch data from an external API and return the current conditions, returning a map from our Cypher query that matches the shape of the `Weather` type.

Add these types and extensions to the `schema.graphql` file:

GraphQL

[Copy to Clipboard](#)

```
# schema.graphql

type Weather {
  temperature: Int
  windSpeed: Int
  windDirection: Int
  precipitation: String
  summary: String
}

extend type Address {
  currentWeather: Weather @cypher(statement:""
    WITH 'https://www.7timer.info/bin/civil.php' AS baseURL, this
    CALL apoc.load.json(
      baseURL + '?lon=' + this.location.longitude + '&lat=' + this.location.latitude +
      '&ac=0&unit=metric&output=json')
    YIELD value WITH value.dataseries[0] as weather
    RETURN {
      temperature: weather.temp2m,
      windSpeed: weather.wind10m.speed,
      windDirection: weather.wind10m.direction,
      precipitation: weather.prec_type,
      summary: weather.weather} AS conditions
    """)
}
```

Now we can include the `currentWeather` field on the `Address` type in our GraphQL queries:

GraphQL

[Copy to Clipboard](#)

```
{
  orders {
    shipTo {
```

Need help? [Ask in the Neo4j Community](#) [↗](#)

```
    address
    currentWeather {
      temperature
      precipitation
      windSpeed
      windDirection
      summary
    }
  }
}
```

JSON

```
{
  "data": {
    "orders": [
      {
        "shipTo": {
          "address": "111 E 5th Ave, San Mateo, CA 94401",
          "currentWeather": {
            "temperature": 9,
            "precipitation": "none",
            "windSpeed": 2,
            "windDirection": "S",
            "summary": "cloudyday"
          }
        }
      },
      {
        "shipTo": {
          "address": "Nordenskiöldsgatan 24, 211 19 Malmö, Sweden",
          "currentWeather": {
            "temperature": 6,
            "precipitation": "none",
            "windSpeed": 4,
            "windDirection": "NW",
            "summary": "clearday"
          }
        }
      }
    ]
  }
}
```

Custom Query Field

We can use the `@cypher` directive on Query fields to compliment the auto-generated Query fields provided by the Neo4j GraphQL Library. Perhaps we want to leverage a [full-text index](#) for fuzzy

Need help? [Ask in the Neo4j Community](#)

matching for book searches?

First, in Neo4j Browser, create the full-text index:

Cypher

Copy to Clipboard

```
CALL db.index.fulltext.createNodeIndex("bookIndex", ["Book"], ["title", "description"])
```

To search this full-text index we use the `db.index.fulltext.queryNodes` procedure:

Cypher

Copy to Clipboard

```
CALL db.index.fulltext.queryNodes("bookIndex", "garph~")
```

Neo4j full-text indexes use Apache Lucene query syntax - the `~` indicates we want to use "fuzzy matching" taking into account slight misspellings.

Next, add a `bookSearch` field to the Query type in our GraphQL type definitions which requires a `searchString` argument that becomes the full-text search term:

GraphQL

Copy to Clipboard

```
# schema.graphql

type Query {
  bookSearch(searchString: String!): [Book] @cypher(statement: """
    CALL db.index.fulltext.queryNodes('bookIndex', $searchString+'~')
    YIELD node RETURN node
    """)
}
```

And we now have a new entry-point to our GraphQL API allowing for full-text search of book titles and descriptions:

GraphQL

Copy to Clipboard

```
{
  bookSearch(searchString: "garph") {
    title
    description
  }
}
```

Need help? [Ask in the Neo4j Community](#) ↗

```
}

```

JSON

```
{
  "data": {
    "bookSearch": [
      {
        "title": "Graph Algorithms",
        "description": "Practical Examples in Apache Spark and Neo4j"
      }
    ]
  }
}
```

Custom Mutation Field

Similar to adding Query fields, we can use `@cypher` schema directives to add new Mutation fields. This is useful in cases where we have specific logic we'd like to take into account when creating or updating data. Here we make use of the `MERGE` Cypher clause [↗](#) to avoid creating duplicate `Subject` nodes and connecting them to books.

GraphQL

Copy to Clipboard

```
# schema.graphql

type Mutation {
  mergeBookSubjects(subject: String!, bookTitles: [String!]!): Subject @cypher(statement:
  """
    MERGE (s:Subject {name: $subject})
    WITH s
    UNWIND $bookTitles AS bookTitle
    MATCH (t:Book {title: bookTitle})
    MERGE (t)-[:ABOUT]->(s)
    RETURN s
  """)
}
```

Now perform the update to the graph:

GraphQL

Copy to Clipboard

```
mutation {
  mergeBookSubjects(
```

Need help? [Ask in the Neo4j Community](#) [↗](#)

```

    subject: "Non-fiction"
    bookTitles: ["Graph Algorithms", "Inspired"]
  ) {
    name
  }
}

```

Custom Resolvers

Combining the power of Cypher and GraphQL is extremely powerful, however there are bound to be some cases where we want to add custom logic using code by implementing resolver functions. This might be where we want to fetch data from another database, API, or system. Let's consider a contrived example where we compute an estimated delivery date using a custom resolver function.

First, we add an `estimatedDelivery` field to the `Order` type, including the `@ignore` [directive](#) which indicates we plan to resolve this field manually and it will not be included in the generated database queries.

GraphQL

[Copy to Clipboard](#)

```

# schema.graphql

extend type Order {
  estimatedDelivery: DateTime @ignore
}

```

Now it's time to implement our `Order.estimatedDelivery` resolver function. Our function simply calculates a random date - but the point is that this can be any custom logic we choose to define.

Add this function to beginning of `index.js`:

JavaScript

[Copy to Clipboard](#)

```

// index.js

const resolvers = {
  Order: {
    estimatedDelivery: (obj, args, context, info) => {
      const options = [1, 5, 10, 15, 30, 45];
      const estDate = new Date();
      estDate.setDate(
        estDate.getDate() + options[Math.floor(Math.random() * options.length)]
      );
    }
  }
}

```

Need help? [Ask in the Neo4j Community](#) [↗](#)

```
        return estDate;
    }
}
};
```

Next, we include the `resolvers` object when instantiating `Neo4jGraphQL`.

Modify these objects in `index.js`:

JavaScript

[Copy to Clipboard](#)

```
// index.js

const neoSchema = new Neo4jGraphQL({
  typeDefs,
  resolvers,
  debug: true
});

const server = new ApolloServer({
  context: { driver },
  schema: neoSchema.schema,
  introspection: true,
  playground: true
});
```

And now we can reference the `estimatedDelivery` field in our GraphQL queries. When this field is included in the selection instead of trying to fetch this field from the database, our custom resolver will be executed.

GraphQL

[Copy to Clipboard](#)

```
{
  orders {
    shipTo {
      address
    }
    estimatedDelivery
  }
}
```

JSON

```
{
  "data": {
```

Need help? [Ask in the Neo4j Community](#) ↗

```

"orders": [
  {
    "shipTo": {
      "address": "111 E 5th Ave, San Mateo, CA 94401"
    },
    "estimatedDelivery": "2021-05-09T23:43:05.970Z"
  },
  {
    "shipTo": {
      "address": "Nordenskiöldsgatan 24, 211 19 Malmö, Sweden"
    },
    "estimatedDelivery": "2021-04-29T23:43:05.970Z"
  }
]
}
}

```

Exercise: Exploring The `@cypher` Directive

We saw how powerful the Cypher directive can be for adding custom logic to our GraphQL API. For this exercise first be sure to follow along with the steps above to make use of the `@cypher` schema directive in your GraphQL type definitions. If you run into any issues you can refer to [this Codesandbox](#) ↗ with all the code we added in this lesson.

For this exercise you will be adding a `similar` field to the `Book` type which will return similar books. How you determine similarity is up to you. Perhaps consider order co-occurrence (books purchased together) or user reviews? First, think about how to query for similar books using Cypher, testing in Neo4j Browser. Then add the query as a `@cypher` directive field to the `Book` type. Advanced users may wish to explore the [Graph Data Science Library](#) ↗ to leverage graph algorithms. For example, here's how we can use the [Jaccard Similarity function](#) ↗ to find similar books using book subjects:

GraphQL

Copy to Clipboard

```

# schema.graphql

extend type Book {
  similar: [Book] @cypher(statement: """
    MATCH (this)-[:ABOUT]->(s:Subject)
    WITH this, COLLECT(id(s)) AS s1
    MATCH (b:Book)-[:ABOUT]->(s:Subject) WHERE b <> this
    WITH this, b, s1, COLLECT(id(s)) AS s2
    WITH b, gds.alpha.similarity.jaccard(s2, s2) AS jaccard
    ORDER BY jaccard DESC
    RETURN b LIMIT 1
  """)
}

```

Need help? [Ask in the Neo4j Community](#) ↗

```
}
```

Your solution will enable clients of the GraphQL API to include the **similar** field in the selection set and view similar books:

GraphQL

[Copy to Clipboard](#)

```
{
  books(where: { title: "Graph Algorithms" }) {
    title
    similar {
      title
    }
  }
}
```

JSON

```
{
  "data": {
    "books": [
      {
        "title": "Graph Algorithms",
        "similar": [
          {
            "title": "Inspired"
          }
        ]
      }
    ]
  }
}
```

Check Your Understanding

Question 1

Schema directives are used in GraphQL type definitions to indicate custom server-side logic. In the Neo4j GraphQL Library which GraphQL schema directive is used to define custom logic using the Cypher query language?

Select the correct answer.

Need help? [Ask in the Neo4j Community](#)

- ☐ @gql
- ☐ @cypher
- ☐ @gds
- ☐ @ignore
- ☐ @relationship

Question 2

Which of the following GraphQL SDL snippets show the proper usage of the `@cypher` schema directive to compute the subtotal for an order?

Select the correct answer.

- ☐ `subTotal: Float @cypher(statement:"MATCH (this)-[:CONTAINS]→(b:Book) RETURN sum(b.price)")`
- ☐ `MATCH (o:Order)-[:CONTAINS]→(b:Book) RETURN b ORDER BY p.price DESC`
- ☐ `{ orders { subTotal } }`

Question 3

The Cypher query used in a `@cypher` schema directive cannot use Cypher procedures such as APOC, the Graph Data Science Library, or other built-in procedures.

Is this statement true or false?

- ☐ True
- ☐ False

Summary

In this lesson, we explored two methods for adding custom logic to our GraphQL API: the `@cypher` schema directive and custom resolvers. In the next lesson we address adding authorization rules to our API using the `@auth` directive and JSON Web Tokens (JWTs).

✓ [Check Answers](#)

Need help? [Ask in the Neo4j Community](#)

[Prev](#)[Next](#)[← The Neo4j GraphQL Library](#)[Authorization →](#)

Contents

[Lesson Overview \(#_lesson_overview\)](#)
[Codesandbox Setup \(#_codesandbox_setup\)](#)
[The @cypher GraphQL Schema Directive \(#_the_cypher_graphql_schema_directive\)](#)
[Custom Resolvers \(#_custom_resolvers\)](#)
[Exercise: Exploring The @cypher Directive \(#_exercise_exploring_the_cypher_directive\)](#)
[Check Your Understanding \(#_check_your_understanding\)](#)
[Summary \(#_summary\)](#)

© 2021 Neo4j, Inc.
[Terms \(https://neo4j.com/terms/\)](https://neo4j.com/terms/) | [Privacy \(https://neo4j.com/privacy-policy/\)](https://neo4j.com/privacy-policy/) | [Sitemap \(https://neo4j.com/sitemap/\)](https://neo4j.com/sitemap/)

Neo4j[®], Neo Technology[®], Cypher[®], Neo4j[®] Bloom[™] and Neo4j[®] Aura[™] are registered trademarks of Neo4j, Inc. All other marks are owned by their respective companies.





[Contact Us → \(https://neo4j.com/contact-us/?ref=footer\)](https://neo4j.com/contact-us/?ref=footer)

US: 1-855-636-4532
Sweden +46 171 480 113
UK: +44 20 3868 3223
France: +33 (0) 8 05 08 03 44


Learn


 [Sandbox \(https://neo4j.com/sandbox/?ref=developer-footer\)](https://neo4j.com/sandbox/?ref=developer-footer)
 [Neo4j Community Site \(https://community.neo4j.com?ref=developer-footer\)](https://community.neo4j.com?ref=developer-footer)
 [Neo4j Developer Blog \(https://medium.com/neo4j\)](https://medium.com/neo4j)

Social

 [Twitter \(https://twitter.com/neo4j\)](https://twitter.com/neo4j)
 [Meetups \(https://www.meetup.com/Neo4j-Online-Meetup/\)](https://www.meetup.com/Neo4j-Online-Meetup/)
 [Github \(https://github.com/neo4j/neo4j\)](https://github.com/neo4j/neo4j)
 [Stack Overflow \(https://stackoverflow.com/questions/tagged/neo4j\)](https://stackoverflow.com/questions/tagged/neo4j)

Need help? [Ask in the Neo4j Community](#) ↗

 [Neo4j Videos \(https://www.youtube.com/neo4j\)](https://www.youtube.com/neo4j)

 [GraphAcademy \(https://neo4j.com/graphacademy/?ref=developer-footer\)](https://neo4j.com/graphacademy/?ref=developer-footer)

 [Neo4j Labs \(https://neo4j.com/labs/?ref=developer-footer\)](https://neo4j.com/labs/?ref=developer-footer)

Want to Speak? [Get \\$ back. \(https://neo4j.com/speaker-program/\)](https://neo4j.com/speaker-program/)

Need help? [Ask in the Neo4j Community](#) ↗