
graphql-constraint-directive

Snyk security

monitored

Snyk security

monitored

Snyk security

monitored

Allows using @constraint as a directive to validate input data. Inspired by Constraints Directives RFC and OpenAPI

Install

```
1 npm install graphql-constraint-directive
```

For GraphQL v15 and below, use v2 of this package

```
1 npm install graphql-constraint-directive@v2
```

Usage

There are multiple ways to make use of the constraint directive in your project. Below outlines the benefits and caveats. Please choose the most appropriate to your use case.

Schema wrapper

Implementation based on schema wrappers - basic scalars are wrapped as custom scalars with validations.

Benefits

- based on [graphql](#) library, works everywhere
- possibility to also validate GraphQL response data

Caveats

- modifies GraphQL schema, basic scalars (Int, Float, String) are replaced by custom scalars

```
1 const { constraintDirective, constraintDirectiveTypeDefs } = require('
  graphql-constraint-directive')
2 const express = require('express')
3 const { ApolloServer } = require('apollo-server-express')
4 const { makeExecutableSchema } = require('@graphql-tools/schema')
5 const typeDefs = `
6   type Query {
```

```
7     books: [Book]
8   }
9   type Book {
10     title: String
11   }
12   type Mutation {
13     createBook(input: BookInput): Book
14   }
15   input BookInput {
16     title: String! @constraint(minLength: 5, format: "email")
17   }
18
19   let schema = makeExecutableSchema({
20     typeDefs: [constraintDirectiveTypeDefs, typeDefs],
21   })
22   schema = constraintDirective()(schema)
23
24   const app = express()
25   const server = new ApolloServer({ schema })
26
27   await server.start()
28
29   server.applyMiddleware({ app })
```

Server plugin

Implementation based on server plugin. Common server plugins are implemented, function `validateQuery(schema, query, variables, operationName)` can be used to implement additional plugins.

Benefits

- schema stays unmodified

Caveats

- runs only in supported servers
- validates only GraphQL query, not response data

Envelop Use as an Envelop plugin in supported frameworks, e.g. GraphQL Yoga. Functionality is plugged in `execute` phase

This plugin requires the following dependencies installed in your project: * `@envelop/core` - ^2.0.0

```
1 const { createEnvelopQueryValidationPlugin, constraintDirectiveTypeDefs
    } = require('graphql-constraint-directive')
2 const express = require('express')
3 const { createServer } = require('@graphql-yoga/node')
4 const { makeExecutableSchema } = require('@graphql-tools/schema')
5
6 const typeDefs = `
7   type Query {
8     books: [Book]
9   }
10  type Book {
11    title: String!
12  }
13  type Mutation {
14    createBook(input: BookInput): Book
15  }
16  input BookInput {
17    title: String! @constraint(minLength: 5, format: "email")
18  }`
19
20 let schema = makeExecutableSchema({
21   typeDefs: [constraintDirectiveTypeDefs, typeDefs],
22 })
23
24 const app = express()
25
26 const yoga = createServer({
27   schema,
28   plugins: [createEnvelopQueryValidationPlugin()],
29   graphql: false
30 })
31
32 app.use('/', yoga)
33
34 app.listen(4000);
```

Apollo 3 Server As an Apollo 3 Server plugin

This plugin requires the following dependencies installed in your project: * dependencies required for your selected Apollo Server 3 variant

```
1 const { createApolloQueryValidationPlugin, constraintDirectiveTypeDefs
    } = require('graphql-constraint-directive')
2 const express = require('express')
3 const { ApolloServer } = require('apollo-server-express')
4 const { makeExecutableSchema } = require('@graphql-tools/schema')
5
6 const typeDefs = `
7   type Query {
```

```

 8     books: [Book]
 9   }
10   type Book {
11     title: String
12   }
13   type Mutation {
14     createBook(input: BookInput): Book
15   }
16   input BookInput {
17     title: String! @constraint(minLength: 5, format: "email")
18   }`
19
20   let schema = makeExecutableSchema({
21     typeDefs: [constraintDirectiveTypeDefs, typeDefs],
22   })
23
24   const plugins = [
25     createApolloQueryValidationPlugin({
26       schema
27     })
28   ]
29
30   const app = express()
31   const server = new ApolloServer({
32     schema,
33     plugins
34   })
35
36   await server.start()
37
38   server.applyMiddleware({ app })

```

Apollo 4 Server As an Apollo 4 Server plugin

This plugin requires the following dependencies installed in your project: `* @apollo/server - ^4.0.0 * graphql-tag - ^2.0.0`

```

1  const { createApollo4QueryValidationPlugin, constraintDirectiveTypeDefs
2    } = require('graphql-constraint-directive/apollo4')
3  const { ApolloServer } = require('@apollo/server')
4  const { startStandaloneServer } = require('@apollo/server/standalone');
5  const { makeExecutableSchema } = require('@graphql-tools/schema')
6
7  const typeDefs = `
8    type Query {
9      books: [Book]
10    }
11    type Book {
12      title: String

```

```

13     type Mutation {
14         createBook(input: BookInput): Book
15     }
16     input BookInput {
17         title: String! @constraint(minLength: 5, format: "email")
18     }`
19
20     let schema = makeExecutableSchema({
21         typeDefs: [constraintDirectiveTypeDefs, typeDefs],
22     })
23
24     const plugins = [
25         createApollo4QueryValidationPlugin()
26     ]
27
28     const server = new ApolloServer({
29         schema,
30         plugins
31     })
32
33     await startStandaloneServer(server);

```

Apollo 4 Subgraph server There is a small change required to make the Apollo Server quickstart work when trying to build an Apollo Subgraph Server. We must use the `buildSubgraphSchema` function to build a schema that can be passed to an Apollo Gateway/supergraph, instead of `makeExecutableSchema`. This uses `makeExecutableSchema` under the hood.

This plugin requires the following dependencies installed in your project: `* @apollo/server - ^4.0.0 * graphql-tag - ^2.0.0`

```

1  import { ApolloServer } from '@apollo/server';
2  import { startStandaloneServer } from '@apollo/server/standalone';
3  import { buildSubgraphSchema } from '@apollo/subgraph';
4  import { createApollo4QueryValidationPlugin,
5         constraintDirectiveTypeDefsGql } from 'graphql-constraint-directive/
6         apollo4';
7
8  const typeDefs = gql`
9      extend schema @link(url: "https://specs.apollo.dev/federation/v2.0",
10         import: ["@key", "@shareable"])
11
12      type Query {
13          books: [Book]
14      }
15      type Book {
16          title: String
17      }
18      type Mutation {

```

```

16     createBook(input: BookInput): Book
17   }
18   input BookInput {
19     title: String! @constraint(minLength: 5, format: "email")
20   }
21 `;
22
23   const schema = buildSubgraphSchema({
24     typeDefs: [constraintDirectiveTypeDefsGql, typeDefs]
25   });
26
27   const plugins = [
28     createApollo4QueryValidationPlugin()
29   ]
30
31   const server = new ApolloServer({
32     schema,
33     plugins
34   });
35
36   await startStandaloneServer(server);

```

Express This implementation is untested now, as *express-graphql* module is not maintained anymore.

As a Validation rule when query variables are available

```

1   const { createQueryValidationRule, constraintDirectiveTypeDefs } =
2     require('graphql-constraint-directive')
3   const express = require('express')
4   const { graphqlHTTP } = require('express-graphql')
5   const { makeExecutableSchema } = require('@graphql-tools/schema')
6
7   const typeDefs = `
8     type Query {
9       books: [Book]
10    }
11    type Book {
12      title: String
13    }
14    type Mutation {
15      createBook(input: BookInput): Book
16    }
17    input BookInput {
18      title: String! @constraint(minLength: 5, format: "email")
19    }
20  `
21
22  let schema = makeExecutableSchema({
23    typeDefs: [constraintDirectiveTypeDefs, typeDefs],
24  })

```

```

23
24 const app = express()
25
26 app.use(
27   '/api',
28   graphqlHTTP(async (request, response, { variables }) => ({
29     schema,
30     validationRules: [
31       createQueryValidationRule({
32         variables
33       })
34     ]
35   })))
36 )
37 app.listen(4000);

```

Schema documentation

You can use the provided schema transformation to automatically add `@constraint` documentation into fields and arguments descriptions. By default directives are not typically present in the exposed introspected schema

```

1 const { constraintDirectiveTypeDefs, constraintDirectiveDocumentation }
  = require('graphql-constraint-directive')
2 const { makeExecutableSchema } = require('@graphql-tools/schema')
3
4 const typeDefs = ...
5
6 let schema = makeExecutableSchema({
7   typeDefs: [constraintDirectiveTypeDefs, typeDefs]
8 })
9
10 schema = constraintDirectiveDocumentation()(schema);
11
12 // any constraint directive handler implementation

```

This transformation appends `constraint documentation header`, and then a list of `constraint conditions descriptions` to the description of each field and argument where the `@constraint` directive is used.

Original schema:

```

1 """
2 Existing field or argument description.
3 """
4 fieldOrArgument: String @constraint(minLength: 10, maxLength: 50)

```

Transformed schema:

```

1  """
2  Existing field or argument description.
3
4  *Constraints:*
5  * Minimum length: `10`
6  * Maximum length: `50`
7  """
8  fieldOrArgument: String @constraint(minLength: 10, maxLength: 50)

```

CommonMark is used in the description for better readability.

If `constraint documentation header` already exists in the field or argument description, then constraint documentation is not appended. This allows you to override constraint description when necessary, or use this in a chain of subgraph/supergraph schemes.

Both `constraint documentation header` and `constraint conditions descriptions` can be customized during the transformation creation, eg. to localize them.

```

1  schema = constraintDirectiveDocumentation(
2    {
3      header: '*Changed header:*',
4      descriptionsMap: {
5        minLength: 'Changed Minimum length',
6        maxLength: 'Changed Maximum length',
7        startsWith: 'Changed Starts with',
8        endsWith: 'Changed Ends with',
9        contains: 'Changed Contains',
10       notContains: 'Changed Doesn\'t contain',
11       pattern: 'Changed Must match RegEx pattern',
12       format: 'Changed Must match format',
13       min: 'Changed Minimum value',
14       max: 'Changed Maximum value',
15       exclusiveMin: 'Changed Grater than',
16       exclusiveMax: 'Changed Less than',
17       multipleOf: 'Changed Must be a multiple of',
18       minItems: 'Changed Minimum number of items',
19       maxItems: 'Changed Maximum number of items'
20     }
21   }
22 )(schema);

```

API

String

minLength `@constraint(minLength: 5)` Restrict to a minimum length

maxLength `@constraint(maxLength: 5)` Restrict to a maximum length

startsWith `@constraint(startsWith: "foo")` Ensure value starts with foo

endsWith `@constraint(endsWith: "foo")` Ensure value ends with foo

contains `@constraint(contains: "foo")` Ensure value contains foo

notContains `@constraint(notContains: "foo")` Ensure value does not contain foo

pattern `@constraint(pattern: "[0-9a-zA-Z]*$")` Ensure value matches regex, e.g. alphanumeric

format `@constraint(format: "email")` Ensure value is in a particular format

Supported formats: - byte: Base64 - date-time: RFC 3339 - date: ISO 8601 - email - ipv4 - ipv6 - uri - uuid

Custom Format You can add your own custom formats by passing a `formats` object to the plugin options. See example below.

`@constraint(format: "my-custom-format")`

```
1  const formats = {
2    'my-custom-format': (value) => {
3      if (value === 'foo') {
4        return true
5      }
6
7      throw new GraphQLError('Value must be foo')
8    }
9  };
10
11 // Envelop
12 createEnvelopQueryValidationPlugin({ formats })
13
14 // Apollo 3 Server
15 createApolloQueryValidationPlugin({ formats })
16
17 // Apollo 4 Server
18 createApollo4QueryValidationPlugin({ formats })
```

Int/Float

min `@constraint(min: 3)` Ensure value is greater than or equal to

max `@constraint(max: 3)` Ensure value is less than or equal to

exclusiveMin `@constraint(exclusiveMin: 3)` Ensure value is greater than

exclusiveMax `@constraint(exclusiveMax: 3)` Ensure value is less than

multipleOf `@constraint(multipleOf: 10)` Ensure value is a multiple

Array/List

minItems `@constraint(minItems: 3)` Restrict array/List to a minimum length

maxItems `@constraint(maxItems: 3)` Restrict array/List to a maximum length

ConstraintDirectiveError

Each validation error throws a `ConstraintDirectiveError`. Combined with a `formatError` function, this can be used to customise error messages.

```
1 {
2   code: 'ERR_GRAPHQL_CONSTRAINT_VALIDATION',
3   fieldName: 'theFieldName',
4   context: [ { arg: 'argument name which failed', value: 'value of
               argument' } ]
5 }
```

```
1 const formatError = function (error) {
2   const code = error?.originalError?.originalError?.code || error?.
    originalError?.code || error?.code
3   if (code === 'ERR_GRAPHQL_CONSTRAINT_VALIDATION') {
4     // return a custom object
5   }
6
7   return error
8 }
9
```

```
10 app.use('/graphql', bodyParser.json(), graphqlExpress({ schema,
    formatError })))
```

Apollo Server 3 Throws a `UserInputError` for each validation error.

Apollo Server 4 Throws a prefilled `GraphQLError` with `extensions.code` set to `BAD_USER_INPUT` and http status code 400. In case of more validation errors, top level error is generic with `Query is invalid`, **for details see** `extensions.validationErrors` message, detailed errors are stored in `extensions.validationErrors` of this error.

Envelop The Envelop plugin throws a prefilled `GraphQLError` for each validation error.

uniqueTypeName

`@constraint(uniqueTypeName: "Unique_Type_Name")` Override the unique type name generate by the library to the one passed as an argument. Has meaning only for `Schema wrapper` implementation.