Scalars

A GraphQL object type has a name and fields, but at some point those fields have to resolve to some concrete data. That's where the scalar types come in: they represent the leaves of the query (read more here). GraphQL includes the following default types: `Int`, `Float`, `String`, `Boolean` and `ID`. In addition to these built-in types, you may need to support custom atomic data types (e.g., `Date`).

Code first

The code-first approach ships with five scalars in which three of them are simple aliases for the existing GraphQL types.

- `ID` (alias for `GraphQLID`) - represents a unique identifier, often used to refetch an object or as the key for a cache
- `Int` (alias for `GraphQLInt`) - a signed 32-bit integer
- `Float` (alias for `GraphQLFloat`) - a signed double-precision floating-point value
- `GraphQLISODateTime` - a date-time string at UTC (used by default to represent `Date` type)
- `GraphQLTimestamp` - a signed integer which represents date and time as number of milliseconds from start of UNIX epoch

The `GraphQLISODateTime` (e.g. `2019-12-03T09:54:33Z`) is used by default to represent the `Date` type. To use the `GraphQLTimestamp` instead, set the `dateScalarMode` of the `buildSchemaOptions` object to `'timestamp'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    dateScalarMode: 'timestamp',
  }
}),
```

Likewise, the `GraphQLFloat` is used by default to represent the `number` type. To use the `GraphQLInt` instead, set the `numberScalarMode` of the `buildSchemaOptions` object to `'integer'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    numberScalarMode: 'integer',
  }
}),
```

In addition, you can create custom scalars.

Override a default scalar

To create a custom implementation for the `Date` scalar, simply create a new class.

```
import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date', (type) => Date)
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}
```

With this in place, register `DateScalar` as a provider.

```
@Module({
  providers: [DateScalar],
})
export class CommonModule {}
```

Now we can use the `Date` type in our classes.

```
@Field()
creationDate: Date;
```

Import a custom scalar

To use a custom scalar, import and register it as a resolver. We'll use the `graphql-type-json` package for demonstration purposes. This npm package defines a `JSON` GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the `forRoot()` method:

```
import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}
```

Now we can use the `JSON` type in our classes.

```
@Field((type) => GraphQLJSON)
info: JSON;
```

For a suite of useful scalars, take a look at the graphql-scalars package.

Create a custom scalar

To define a custom scalar, create a new `GraphQLScalarType` instance. We'll create a custom `UUID` scalar.

```
const regex = /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]
{12}$/i;

function validate(uuid: unknown): string | never {
  if (typeof uuid !== "string" || !regex.test(uuid)) {
    throw new Error("invalid uuid");
  }
  return uuid;
}

export const CustomUuidScalar = new GraphQLScalarType({
  name: 'UUID',
  description: 'A simple UUID parser',
  serialize: (value) => validate(value),
  parseValue: (value) => validate(value),
  parseLiteral: (ast) => validate(ast.value)
})
```

We pass a custom resolver to the `forRoot()` method:

```
@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { UUID: CustomUuidScalar },
```

```
    }),
  ],
})
export class AppModule {}
```

Now we can use the `UUID` type in our classes.

```
@Field((type) => CustomUuidScalar)
uuid: string;
```

Schema first

To define a custom scalar (read more about scalars here), create a type definition and a dedicated resolver. Here (as in the official documentation), we'll use the `graphql-type-json` package for demonstration purposes. This npm package defines a `JSON` GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the `forRoot()` method:

```
import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      typePaths: ['./**/*.graphql'],
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}
```

Now we can use the `JSON` scalar in our type definitions:

```
scalar JSON

type Foo {
  field: JSON
}
```

Another method to define a scalar type is to create a simple class. Assume we want to enhance our schema with the `Date` type.

```
import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date')
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}
```

With this in place, register `DateScalar` as a provider.

```
@Module({
  providers: [DateScalar],
})
export class CommonModule {}
```

Now we can use the `Date` scalar in type definitions.

```
scalar Date
```

By default, the generated TypeScript definition for all scalars is `any` - which isn't particularly typesafe. But, you can configure how Nest generates typings for your custom scalars when you specify how to generate types:

```
import { GraphQLDefinitionsFactory } from '@nestjs/graphql';
import { join } from 'path';

const definitionsFactory = new GraphQLDefinitionsFactory();
```

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  defaultScalarType: 'unknown',
  customScalarTypeMapping: {
    DateTime: 'Date',
    BigNumber: '_BigNumber',
  },
  additionalHeader: "import _BigNumber from 'bignumber.js'",
});
```

> info **Hint** Alternatively, you can use a type reference instead, for example: `DateTime: Date`. In this case, `GraphQLDefinitionsFactory` will extract the name property of the specified type (`Date.name`) to generate TS definitions. Note: adding an import statement for non-built-in types (custom types) is required.

Now, given the following GraphQL custom scalar types:

```
scalar DateTime
scalar BigNumber
scalar Payload
```

We will now see the following generated TypeScript definitions in `src/graphql.ts`:

```
import _BigNumber from 'bignumber.js';

export type DateTime = Date;
export type BigNumber = _BigNumber;
export type Payload = unknown;
```

Here, we've used the `customScalarTypeMapping` property to supply a map of the types we wish to declare for our custom scalars. We've also provided an `additionalHeader` property so that we can add any imports required for these type definitions. Lastly, we've added a `defaultScalarType` of `'unknown'`, so that any custom scalars not specified in `customScalarTypeMapping` will be aliased to `unknown` instead of `any` (which TypeScript recommends using since 3.0 for added type safety).

> info **Hint** Note that we've imported `_BigNumber` from `bignumber.js`; this is to avoid circular type references.