Federation

Federation offers a means of splitting your monolithic GraphQL server into independent microservices. It consists of two components: a gateway and one or more federated microservices. Each microservice holds part of the schema and the gateway merges the schemas into a single schema that can be consumed by the client.

To quote the Apollo docs, Federation is designed with these core principles:

- Building a graph should be **declarative.** With federation, you compose a graph declaratively from within your schema instead of writing imperative schema stitching code.
- Code should be separated by **concern**, not by types. Often no single team controls every aspect of an important type like a User or Product, so the definition of these types should be distributed across teams and codebases, rather than centralized.
- The graph should be simple for clients to consume. Together, federated services can form a complete, product-focused graph that accurately reflects how it's being consumed on the client.
- It's just **GraphQL**, using only spec-compliant features of the language. Any language, not just JavaScript, can implement federation.

> warning **Warning** Federation currently does not support subscriptions.

In the following sections, we'll set up a demo application that consists of a gateway and two federated endpoints: Users service and Posts service.

Federation with Apollo

Start by installing the required dependencies:

```
$ npm install --save @apollo/federation @apollo/subgraph
```

Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Apollo query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

Code first

Start by adding some extra decorators to the `User` entity.

```
import { Directive, Field, ID, ObjectType } from '@nestjs/graphql';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field()
  name: string;
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example
```

```
@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

A working example is available here in code first mode and here in schema first mode.

Federated example: Posts

Post service is supposed to serve aggregated posts through the `getPosts` query, but also extend our `User` type with the `user.posts` field.

Schema first

"Posts service" references the `User` type in its schema by marking it with the `extend` keyword. It also declares one additional property on the `User` type (`posts`). Note the `@key` directive used for matching instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```
import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    return this.postsService.findAll();
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    return { __typename: 'User', id: post.userId };
  }
}
```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [PostsResolvers],
})
export class AppModule {}
```

Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';
```

```
@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  @Directive('@external')
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```
import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

We also have to define the `Post` entity class:

```
import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
  @Field((type) => ID)
  id: number;

  @Field()
  title: string;

  @Field((type) => Int)
  authorId: number;

  @Field((type) => User)
  user?: User;
}
```

And its resolver:

```
import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
  constructor(private readonly postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    return this.postsService.findOne(id);
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    return this.postsService.all();
  }

  @ResolveField((of) => User)
  user(@Parent() post: Post): any {
    return { __typename: 'User', id: post.authorId };
  }
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
```

```
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

A working example is available here for the code first mode and here for the schema first mode.

Federated example: Gateway

Start by installing the required dependency:

```
$ npm install --save @apollo/gateway
```

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```
import { IntrospectAndCompose } from '@apollo/gateway';
import { ApolloGatewayDriver, ApolloGatewayDriverConfig } from
'@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
      driver: ApolloGatewayDriver,
      server: {
        // ... Apollo server options
        cors: true,
      },
      gateway: {
        supergraphSdl: new IntrospectAndCompose({
          subgraphs: [
            { name: 'users', url: 'http://user-service/graphql' },
            { name: 'posts', url: 'http://post-service/graphql' },
          ],
        }),
      },
    }),
  ],
})
export class AppModule {}
```

A working example is available here for the code first mode and here for the schema first mode.

Federation with Mercurius

Start by installing the required dependencies:

```
$ npm install --save @apollo/subgraph @nestjs/mercurius
```

> info **Note** The `@apollo/subgraph` package is required to build a subgraph schema (`buildSubgraphSchema`, `printSubgraphSchema` functions).

Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Mercurius query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `MercuriusFederationDriver` driver in the configuration object:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      typePaths: ['**/*.graphql'],
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

Code first

Start by adding some extra decorators to the `User` entity.

```
import { Directive, Field, ID, ObjectType } from '@nestjs/graphql';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field()
  name: string;
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';
```

```
@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `MercuriusFederationDriver` driver in the configuration object:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

Federated example: Posts

Post service is supposed to serve aggregated posts through the `getPosts` query, but also extend our `User` type with the `user.posts` field.

Schema first

"Posts service" references the `User` type in its schema by marking it with the `extend` keyword. It also declares one additional property on the `User` type (`posts`). Note the `@key` directive used for matching

instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```
import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    return this.postsService.findAll();
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    return { __typename: 'User', id: post.userId };
  }
}
```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
```

```
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      federationMetadata: true,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [PostsResolvers],
})
export class AppModule {}
```

Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  @Directive('@external')
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```
import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])
```

```
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

We also have to define the `Post` entity class:

```
import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
  @Field((type) => ID)
  id: number;

  @Field()
  title: string;

  @Field((type) => Int)
  authorId: number;

  @Field((type) => User)
  user?: User;
}
```

And its resolver:

```
import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
  constructor(private readonly postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    return this.postsService.findOne(id);
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    return this.postsService.all();
  }

  @ResolveField((of) => User)
```

```
  user(@Parent() post: Post): any {
    return { __typename: 'User', id: post.authorId };
  }
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

Federated example: Gateway

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```
import {
  MercuriusGatewayDriver,
  MercuriusGatewayDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
```

```
    GraphQLModule.forRoot<MercuriusGatewayDriverConfig>({
      driver: MercuriusGatewayDriver,
      gateway: {
        services: [
          { name: 'users', url: 'http://user-service/graphql' },
          { name: 'posts', url: 'http://post-service/graphql' },
        ],
      },
    }),
  ],
})
export class AppModule {}
```

Federation 2

To quote the Apollo docs, Federation 2 improves developer experience from the original Apollo Federation (called Federation 1 in this doc), which is backward compatible with most original supergraphs.

> warning **Warning** Mercurius doesn't fully support Federation 2. You can see the list of libraries that support Federation 2 here.

In the following sections, we'll upgrade the previous example to Federation 2.

Federated example: Users

One change in Federation 2 is that entities have no originating subgraph, so we don't need to extend `Query` anymore. For more detail please refer to the entities topic in Apollo Federation 2 docs.

Schema first

We can simply remove `extend` keyword from the schema.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

type Query {
  getUser(id: ID!): User
}
```

Code first

To use Federation 2, we need to specify the federation version in `autoSchemaFile` option.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
```

```
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

Federated example: Posts

With the same reason as above, we don't need to extend `User` and `Query` anymore.

Schema first

We can simply remove `extend` and `external` directives from the schema

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

type User @key(fields: "id") {
  id: ID!
  posts: [Post]
}

type Query {
  getPosts: [Post]
}
```

Code first

Since we don't extend `User` entity anymore, we can simply remove `extends` and `external` directives from `User`.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Also, similarly to the User service, we need to specify in the `GraphQLModule` to use Federation 2.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```