Subscriptions

In addition to fetching data using queries and modifying data using mutations, the GraphQL spec supports a third operation type, called `subscription`. GraphQL subscriptions are a way to push data from the server to the clients that choose to listen to real time messages from the server. Subscriptions are similar to queries in that they specify a set of fields to be delivered to the client, but instead of immediately returning a single answer, a channel is opened and a result is sent to the client every time a particular event happens on the server.

A common use case for subscriptions is notifying the client side about particular events, for example the creation of a new object, updated fields and so on (read more here).

Enable subscriptions with Apollo driver

To enable subscriptions, set the `installSubscriptionHandlers` property to `true`.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  installSubscriptionHandlers: true,
}),
```

> warning **Warning** The `installSubscriptionHandlers` configuration option has been removed from the latest version of Apollo server and will be soon deprecated in this package as well. By default, `installSubscriptionHandlers` will fallback to use the `subscriptions-transport-ws` (read more) but we strongly recommend using the `graphql-ws`(read more) library instead.

To switch to use the `graphql-ws` package instead, use the following configuration:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': true
  },
}),
```

> info **Hint** You can also use both packages (`subscriptions-transport-ws` and `graphql-ws`) at the same time, for example, for backward compatibility.

Code first

To create a subscription using the code first approach, we use the `@Subscription()` decorator (exported from the `@nestjs/graphql` package) and the `PubSub` class from the `graphql-subscriptions` package, which provides a simple **publish/subscribe API**.

The following subscription handler takes care of **subscribing** to an event by calling `PubSub#asyncIterator`. This method takes a single argument, the `triggerName`, which corresponds to an event topic name.

```
const pubSub = new PubSub();

@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
}
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package, while the `PubSub` class is exported from the `graphql-subscriptions` package.

> warning **Note** `PubSub` is a class that exposes a simple `publish` and `subscribe API`. Read more about it here. Note that the Apollo docs warn that the default implementation is not suitable for production (read more here). Production apps should use a `PubSub` implementation backed by an external store (read more here).

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., `commentAdded` above), or is provided explicitly by passing an option with the key `name` as the second argument to the `@Subscription()` decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```
@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  pubSub.publish('commentAdded', { commentAdded: newComment });
  return newComment;
}
```

The `PubSub#publish` method takes a `triggerName` (again, think of this as an event topic name) as the first parameter, and an event payload as the second parameter. As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```
type Subscription {
  commentAdded(): Comment!
}
```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish('commentAdded', {{ '{' }} commentAdded: newComment {{ '}' }})` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```
@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string) {
  return pubSub.asyncIterator('commentAdded');
}
```

Mutating subscription payloads

To mutate the published event payload, set the `resolve` property to a function. The function receives the event payload (as sent by the event publisher) and returns the appropriate value.

```
@Subscription(returns => Comment, {
  resolve: value => value,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

> warning **Note** If you use the `resolve` option, you should return the unwrapped payload (e.g., with our example, return a `newComment` object directly, not a `{{ '{' }} commentAdded: newComment {{ '}' }}` object).

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  resolve(this: AuthorResolver, value) {
    // "this" refers to an instance of "AuthorResolver"
    return value;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();
```

```
@Resolver('Author')
export class AuthorResolver {
  // ...
  @Subscription()
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

To mutate the published payload, we can use a `resolve` function.

```
@Subscription('commentAdded', {
  resolve: value => value,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
  resolve(this: AuthorResolver, value) {
    // "this" refers to an instance of "AuthorResolver"
    return value;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription. You can find a full sample implementation here.

PubSub

We instantiated a local `PubSub` instance above. The preferred approach is to define `PubSub` as a provider and inject it through the constructor (using the `@Inject()` decorator). This allows us to re-use the instance across the whole application. For example, define a provider as follows, then inject `'PUB_SUB'` where needed.

```
{
  provide: 'PUB_SUB',
  useValue: new PubSub(),
}
```

Customize subscriptions server

To customize the subscriptions server (e.g., change the path), use the `subscriptions` options property.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      path: '/graphql'
    },
  }
}),
```

If you're using the `graphql-ws` package for subscriptions, replace the `subscriptions-transport-ws` key with `graphql-ws`, as follows:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      path: '/graphql'
    },
  }
}),
```

Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `onConnect` callback function that you can specify in the `subscriptions` options.

The `onConnect` will receive as a first argument the `connectionParams` passed to the `SubscriptionClient` (read more).

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      onConnect: (connectionParams) => {
        const authToken = connectionParams.authToken;
        if (!isValid(authToken)) {
          throw new Error('Token is not valid');
        }
        // extract user information from token
        const user = parseToken(authToken);
        // return user info to add them to the context later
        return { user };
      },
    }
  },
  context: ({ connection }) => {
    // connection.context will be equal to what was returned by the
"onConnect" callback
  },
}),
```

The `authToken` in this example is only sent once by the client, when the connection is first established. All subscriptions made with this connection will have the same `authToken`, and thus the same user info.

> warning **Note** There is a bug in `subscriptions-transport-ws` that allows connections to skip the `onConnect` phase (read more). You should not assume that `onConnect` was called when the user starts a subscription, and always check that the `context` is populated.

If you're using the `graphql-ws` package, the signature of the `onConnect` callback will be slightly different:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      onConnect: (context: Context<any>) => {
        const { connectionParams, extra } = context;
        // user validation will remain the same as in the example above
        // when using with graphql-ws, additional context value should be
stored in the extra field
        extra.user = { user: {} };
      },
    },
  },
  context: ({ extra }) => {
    // you can now access your additional context value through the extra
field
  },
});
```

Enable subscriptions with Mercurius driver

To enable subscriptions, set the `subscription` property to `true`.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: true,
}),
```

> info **Hint** You can also pass the options object to set up a custom emitter, validate incoming connections, etc. Read more here (see `subscription`).

Code first

To create a subscription using the code first approach, we use the `@Subscription()` decorator (exported from the `@nestjs/graphql` package) and the `PubSub` class from the `mercurius` package, which provides a simple **publish/subscribe API**.

The following subscription handler takes care of **subscribing** to an event by calling `PubSub#asyncIterator`. This method takes a single argument, the `triggerName`, which corresponds to an event topic name.

```
@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

> info **Hint** All decorators used in the example above are exported from the `@nestjs/graphql` package, while the `PubSub` class is exported from the `mercurius` package.

> warning **Note** `PubSub` is a class that exposes a simple `publish` and `subscribe` API. Check out this section on how to register a custom `PubSub` class.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., `commentAdded` above), or is provided explicitly by passing an option with the key `name` as the second argument to the `@Subscription()` decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```
@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
  @Context('pubsub') pubSub: PubSub,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  await pubSub.publish({
    topic: 'commentAdded',
    payload: {
      commentAdded: newComment
    }
  });
  return newComment;
}
```

As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```
type Subscription {
  commentAdded(): Comment!
}
```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish({{ '{' }} topic: 'commentAdded', payload: {{ '{' }} commentAdded: newComment {{ '}' }} {{ '}' }})` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```
@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub:
PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub:
PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();

@Resolver('Author')
export class AuthorResolver {
  // ...
  @Subscription()
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription.

PubSub

In the examples above, we used the default `PubSub` emitter (mqemitter) The preferred approach (for production) is to use `mqemitter-redis`. Alternatively, a custom `PubSub` implementation can be provided (read more here)

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: {
    emitter: require('mqemitter-redis')({
      port: 6579,
      host: '127.0.0.1',
    }),
  },
});
```

Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `verifyClient` callback function that you can specify in the `subscription` options.

The `verifyClient` will receive the `info` object as a first argument which you can use to retrieve the request's headers.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: {
    verifyClient: (info, next) => {
      const authorization = info.req.headers?.authorization as string;
      if (!authorization?.startsWith('Bearer ')) {
        return next(false);
      }
      next(true);
    },
  }
}),
```