# Lab 10: Real-Time Subscriptions

The GraphQL API we have built is very advanced, as is the front end. In the previous lab, we introduced server-side rendering to our application. We provided the user with a lot of information through the news feed, chats, and profile pages. The problem we are facing now, however, is that the user currently has to either refresh the browser or we have set a [pollInterval] to all [Query] components to keep the display up to date. A better solution is to implement Apollo Subscriptions through WebSockets. This allows us to refresh the UI of the user with the newest user information in real time without manual user interaction or polling.

This lab covers the following topics:

- Using WebSockets
- Implementing Apollo Subscriptions
- JWT authentication with Subscriptions

### Lab Solution

Complete solution for this lab is available in the following directory:

`cd ~/Desktop/react-graphql-course/labs/Lab10`

# Apollo Subscriptions

When we started implementing Apollo in our stack, I explained how to set it up manually. As an alternative, there is the [apollo-boost] package, which does this for you automatically. We have now reached a point where this package cannot be used anymore since subscriptions are an advanced feature that isn't supported. If you rely on the [apollo-boost] package, take a look at Lab 4 to see how to switch to a manual setup.

#### ProTip

You can find an excellent overview and more details about Apollo Subscriptions in the official documentation at https://www.apollographql.com/docs/react/advanced/subscriptions.html.

The first step is to install all the required packages to get GraphQL subscriptions working. Install them using npm:

```
npm install --save graphql-subscriptions subscriptions-transport-ws apollo-link-ws
```

The following three packages provide necessary modules for a subscription system:

- The [graphql-subscriptions] package provides the ability to connect our GraphQL back end with a PubSub system. It gives the client the option to subscribe to specific channels, and lets the back end publish new data to the client.
- The [subscriptions-transport-ws] package gives our Apollo Server or other GraphQL libraries the option to accept WebSocket connections and accept queries, mutations, and subscriptions over WebSockets.
- The [apollo-link-ws] package is similar to the [HttpLink] or [UploadLink] that we're currently using, but, as the name suggests, it relies on WebSockets, not HTTP, to send requests and allows us to use subscriptions.

Let's take a look at how we can implement subscriptions.

First, we are going to create a new subscription type next to the [RootQuery] and [RootMutation] types inside the GraphQL schema. You can set up events or entities that a client can subscribe to and receive updates inside the new subscription type. It only works by adding the matching [resolver] functions as well. Instead of returning real data for this new subscription type, you return a special object that allows the client to subscribe to events for the specific entity. These entities can be things such as notifications, new chat messages, or comments on a post. Each of them has got their own subscription channel.

The client can subscribe to these channels. It receives updates any time the back end sends a new WebSocket message -- because data has been updated, for example. The back end calls a [publish] method that sends the new data through the subscription to all clients. You should be aware that not every user should receive all WebSocket messages since the content may include private data such as chat messages. There should be a filter before the update is sent to target only specific users. We'll see this feature later in the *Authentication with Apollo Subscriptions* section.

## Subscriptions on the Apollo Server

We have now installed all the essential packages. Let's start with the implementation for the back end. As mentioned previously, we are going to rely on WebSockets, as they allow real-time communication between the front end and the back end. We are first going to set up the new transport protocol for the back end.

Open the `index.js` file of the server. Import a new Node.js interface at the top of the file:

```
import { createServer } from 'http';
```

The [http] interface is included in Node.js by default. It handles the traditional HTTP protocol, making the use of many HTTP features easy for the developer.

We are going to use the interface to create a standardized Node.js HTTP server object because the Apollo [SubscriptionServer] module expects such an object. We'll cover the Apollo [SubscriptionServer] module soon in this section. Add the following line of code beneath the initialization of Express.js inside the [app] variable:

```
const server = createServer(app);
```

The [createServer] function creates a new HTTP [server] object, based on the original Express.js instance. We pass the Express instance, which we saved inside the [app] variable. As you can see in the preceding code, you only pass the [app] object as a parameter to the [createServer] function. We're going to use the new [server] object instead of the [app] variable to let our back end start listening for incoming requests. Remove the old [app.listen] function call from the bottom of the file because we'll be replacing it in a second. To get our server listening again, edit the initialization routine of the services. The [for] loop should now look as follows:

```
for (let i = 0; i < serviceNames.length; i += 1) {
  const name = serviceNames[i];
  switch (name) {
    case 'graphql':
      services[name].applyMiddleware({ app });
      break;
    case 'subscriptions':
      server.listen(8000, () => {
        console.log('Listening on port 8000!');
        services[name](server);
      });
    break;
    default:
      app.use(`/${name}`, services[name]);
      break;
  }
}
```

Here, we have changed the old [if] statement to a [switch]. Furthermore, we have added a second service beyond `graphql`, called [subscriptions]. We are going to create a new [subscriptions] service next to the `graphql` services folder.

The [subscriptions] service requires the [server] object as a parameter to start listening for WebSocket connections. Before initializing [SubscriptionServer], we need to have started listening for incoming requests. That is why we use the [server.listen] method in the preceding code before initializing the new [subscriptions] service that creates the Apollo [SubscriptionServer]. We pass the [server] object to the service after it started listening. The service has to accept this parameter, of course, so keep this in mind.

To add the new service into the preceding [serviceNames] object, edit the `index.js` services file with the following content:

```
import graphql from './graphql';
import subscriptions from './subscriptions';

export default utils => ({
  graphql: graphql(utils),
  subscriptions: subscriptions(utils),
});
```

The [subscriptions] service also receives the [utils] object, like the `graphql` service.

Now, create a [subscriptions] folder next to the `graphql` folder. To fulfil the import of the preceding [subscriptions] service, insert the service's `index.js` file into this folder. There, we can implement the [subscriptions] service. As a reminder, we pass the [utils] object and also the [server] object from before. The [subscriptions] service must accept two parameters in separate function calls.

If you have created the new subscriptions `index.js` file, import all the dependencies at the top of the file:

```
import { makeExecutableSchema } from 'graphql-tools';
import Resolvers from'../graphql/resolvers';
import Schema from'../graphql/schema';
import auth from '../graphql/auth';
import jwt from 'jsonwebtoken';
const { JWT_SECRET } = process.env;
import { SubscriptionServer } from 'subscriptions-transport-ws';
import { execute, subscribe } from 'graphql';
```

The preceding dependencies are almost the same as those that we are using for the `graphql` service, but we've added the `subscriptions-transport-ws` package. Furthermore, we've removed the `apollo-server-express` package. `SubscriptionServer` is the equivalent of `ApolloServer`, but used for WebSocket connections rather than HTTP. It usually makes sense to set up the Apollo Server for HTTP and `SubscriptionServer` for WebSockets in the same file, as this saves us from processing `Schema` and `Resolvers` twice. It's easier to explain the implementation of subscriptions without the `ApolloServer` code in the same file, though. The last two things that are new in the preceding code are the `execute` and `subscribe` functions that we import from the `graphql` package. You will see why we need these in the following section.

We begin the implementation of the new service by exporting a function with the [export default] statement and creating the [executableSchema] :

```
export default (utils) => (server) => {
  const executableSchema = makeExecutableSchema({
    typeDefs: Schema,
    resolvers: Resolvers.call(utils),
    schemaDirectives: {
      auth: auth
    },
  });
}
```

As you can see, we use the ES6 arrow notation to return two functions at the same time. The first one accepts the [utils] object and the second one accepts the [server] object that we create with the [createServer] function inside the `index.js` file of the server. This approach fixes the problem of passing two parameters in separate function calls. The schema is only created when both functions are called.

The second step is to start [SubscriptionServer] to accept WebSocket connections and, as a result, be able to use the GraphQL subscriptions. Insert the following code under [executableSchema]:

```
new SubscriptionServer({
  execute,
  subscribe,
  schema: executableSchema,
},
{
  server,
  path: '/subscriptions',
});
```

We initialized a new [SubscriptionServer] instance in the preceding code. The first parameter we pass is a general options object for GraphQL and corresponds to the options of the [ApolloServer] class. The options are as following:

- The [execute] property should receive a function that handles all the processing and execution of incoming GraphQL requests. The standard is to pass the [execute] function that we imported from the `graphql` package previously.
- The [subscribe] property also accepts a function. This function has to take care of resolving a subscription to **AsyncIterator**, which is no more than an asynchronous [for] loop. It allows the client to listen for execution results and reflect them to the user.
- The last option we pass is the GraphQL schema. We do this in the same way as for [ApolloServer].

The second parameter our new instance accepts is the [socketOptions] object. This holds settings to describe the way in which the WebSockets work:

- The [server] field receives our [server] object, which we pass from the `index.js` file of the server as a result of the [createServer] function. [SubscriptionServer] then relies on the existing server.
- The [path] field represents the endpoint under which the subscriptions are accessible. All subscriptions use the [/subscriptions] path.

**ProTip**

The official documentation for the [subscriptions-transport-ws] package offers a more advanced explanation of [SubscriptionServer]. Take a look to get an overview of all its functionalities: https://github.com/apollographql/subscriptions-transport-ws#subscriptionserver.

The client would be able to connect to the WebSocket endpoint at this point. There are currently no subscriptions and the corresponding resolvers are set up in our GraphQL API.

Open the [schema.js] file to define our first subscription. Add a new type called [RootSubscription] next to the [RootQuery] and [RootMutation] types, including the new subscription, called [messageAdded]:

```
type RootSubscription {
  messageAdded: Message
}
```

Currently, if a user sends a new message to another user, this isn't shown to the recipient right away.

The first option I showed you was to set an interval to request new messages. Our back end is now able to cover this scenario with subscriptions. The event or channel that the client can subscribe to is called [messageAdded]. We can also add further parameters, such as a chat ID, to filter the WebSocket messages if necessary. When creating a new chat message, it is publicized through this channel.

We have added [RootSubscription], but we need to extend the schema root tag too. Otherwise, the new [RootSubscription] won't be used. Change the schema as follows:

```
schema {
  query: RootQuery
  mutation: RootMutation
  subscription: RootSubscription
}
```

We have successfully configured the tree GraphQL main types. Next, we have to implement the corresponding resolver functions. Open the [resolvers.js] file and perform the following steps:

1. Import all dependencies that allow us to set up our GraphQL API with a [PubSub] system:

```
import { PubSub, withFilter } from 'graphql-subscriptions';
const pubsub = new PubSub();
```

The [PubSub] system offered by the [graphql-subscriptions] package is a simple implementation based on the standard Node.js [EventEmitter]. When going to production, it's recommended to use an external store, such as Redis, with this package.

2. We've already added the third [RootSubscription] type to the schema, but not the matching property on the [resolvers] object. The following code includes the [messageAdded] subscription. Add it to the resolvers:

```
RootSubscription: {
  messageAdded: {
    subscribe: () => pubsub.asyncIterator(['messageAdded']),
  }
},
```

The [messageAdded] property isn't a function but just a simple object. It contains a [subscribe] function that returns **AsyncIterable**. It allows our application to subscribe to the [messageAdded] channel by returning a promise that's only resolved when a new message is added. The next item that's returned is a promise, which is also only resolved when a message has been added. This method makes **AsyncIterators** great for implementing subscriptions.

**ProTip**

You can learn more about how AsyncIterators work by reading through the proposal at https://github.com/tc39/proposal-async-iteration.

3. When subscribing to the [messageAdded] subscription, there needs to be another method that publicizes the newly created message to all clients. The best location is the [addMessage] mutation where the new message is created. Replace the [addMessage] resolver function with the following code:

```
addMessage(root, { message }, context) {
  logger.log({
      level: 'info',
      message: 'Message was created',
    });
  return Message.create({
      ...message,
  }).then((newMessage) => {
      return Promise.all([
          newMessage.setUser(context.user.id),
          newMessage.setChat(message.chatId),
      ]).then(() => {
          pubsub.publish('messageAdded', {messageAdded:
          newMessage});
          return newMessage;
      });
   );
},
```

I have edited the [addMessage] mutation so that the correct user from the context is chosen. All of the new messages that you send are now saved with the correct user id. This allows us to filter WebSocket messages for the correct users later in *Authentication with Apollo Subscriptions* section.

We use the [pubsub.publish] function to send a new WebSocket frame to all clients that are connected and that have subscribed to the [messageAdded] channel. The first parameter of the [pubsub.publish] function is the subscription, which in this case is [messageAdded]. The second parameter is the new message that we save to the database. All clients that have subscribed to the [messageAdded] subscription through **AsyncIterator** now receive this message.

We've finished preparing the back end. The part that required the most work was to get the Express.js and WebSocket transport working together. The GraphQL implementation only involves the new schema entities, correctly implementing the resolvers functions for the subscription, and then publishing the data to the client via the [PubSub] system.

We have to implement the subscription feature in the front end to connect to our WebSocket endpoint.

## Subscriptions on the Apollo Client

As with the back end code, we also need to make adjustments to the Apollo Client configuration before using subscriptions. In Lab 4, we set up the Apollo Client with the normal [HttpLink]. Later, we exchanged it with the [createUploadLink] function, which enables the user to upload files through GraphQL.

We are going to extend our Apollo Client by using [WebSocketLink] as well. This allows us to use subscriptions through GraphQL. Both links work side by side. We use the standard HTTP protocol to query data, such as the chat list or the news feed; all of these are real-time updates to keep the UI up to date rely on WebSockets.

To configure the Apollo Client correctly, follow these steps:

1. Open the `index.js` file from the [apollo] folder. Import the following dependencies:

```
import { WebSocketLink } from 'apollo-link-ws';
import { SubscriptionClient } from 'subscriptions-transport-ws';
import { getMainDefinition } from 'apollo-utilities';
import { ApolloLink, split } from 'apollo-link';
```

To get the subscriptions working, we need [SubscriptionClient], which uses [WebSocketLink] to subscribe to our GraphQL API using WebSockets.

We import the [getMainDefinition] function from the [apollo-utilities] package. It's installed by default when using the Apollo Client. The purpose of this function is to give you the operation type, which can be [query], [mutation], or [subscription].

The [split] function from the [apollo-link] package allows us to conditionally control the flow of requests through different Apollo links based on the operation type or other information. It accepts one condition and one (or a pair of) link from which it composes a single valid link that the Apollo Client can use.

2. We are going to create both links for the [split] function. Detect the protocol and port where we send all GraphQL subscriptions and requests. Add the following code beneath the imports:

```
const protocol = (location.protocol != 'https:') ? 'ws://': 'wss://';
const port = location.port ? ':'+location.port: '';
```

The [protocol] variable saves the WebSocket protocol by detecting whether the client uses [http] or [https]. The [port] variable is either an empty string if we use port [80] to server our front end, or any other port, such as [8000], which we currently use. Previously, we had to statically save [http://localhost:8000] in this file. With the new variables, we can dynamically build the URL where all requests should be sent.

3. The [split] function expects two links to combine them to one. The first link is the normal [httpLink], which we must set up before passing the resulting link to the initialization of the Apollo Client. Remove the [createUploadLink] function call from the [ApolloLink.from] function and add it before the [ApolloClient] class:

```
const httpLink = createUploadLink({
  uri: location.protocol + '//' + location.hostname + port +
   '/graphql',
  credentials: 'same-origin',
});
```

We concatenate the [protocol] of the server, which is either [http:] or [https:], with two slashes. The [hostname] is, for example, the domain of your application or, if in development, [localhost]. The result of the concatenation is [http://localhost:8000/graphql].

4. Add the WebSocket link that's used for the subscriptions next to [httpLink]. It's the second one we pass to the [split] function:

```
const SUBSCRIPTIONS_ENDPOINT = protocol + location.hostname + port
 + '/subscriptions';
const subClient = new SubscriptionClient(SUBSCRIPTIONS_ENDPOINT, {
  reconnect: true,
  connectionParams: () => {
    var token = localStorage.getItem('jwt');
    if(token) {
      return { authToken: token };
    }
    return { };
  }
});
const wsLink = new WebSocketLink(subClient);
```

We define the URI that's stored inside the [SUBSCRIPTIONS_ENDPOINT] variable. It's built with the [protocol] and [port] variables, which we detected earlier, and the application's [hostname]. The URI ends with the specified endpoint of the back end with the same port as the GraphQL API. The URI is the first parameter of [SubscriptionsClient]. The second parameter allows us to pass options, such as the [reconnect] property. It tells the client to automatically reconnect to the back end's WebSocket endpoint when it has lost the connection. This usually happens if the client has temporarily lost their internet connection or the server has gone down.

Furthermore, we use the [connectionParams] field to specify the JWT as an authorization token. We define this property as a function so that the JWT is read from `localStorage` whenever the user logs in. It's sent when the WebSocket is created.

We initialize [SubscriptionClient] to the [subClient] variable. We pass it to the [WebSocketLink] constructor under the [wsLink] variable with the given settings.

5. Combine both links into one. This allows us to insert the composed result into our [ApolloClient] at the bottom. To do this, we have imported the [split] function. The syntax to combine the two links should look as follows:

```
const link = split(
  ({ query }) => {
    const { kind, operation } = getMainDefinition(query);
    return kind === 'OperationDefinition' && operation ===
     'subscription';
  },
  wsLink,
  httpLink,
);
```

The [split] function accepts three parameters. The first parameter must be a function with a Boolean return value. If the return value is [true], the request is sent over the first link, which is the second required parameter. If the return value is [false], the operation is sent over the second link, which we pass via the optional third parameter. In our case, the function that's passed as the first parameter determines the operation type. If the operation is a subscription, the function returns [true] and sends the operation over the WebSocket link. All other requests are sent via the HTTP Apollo Link. We save the result of the [split] function in the [link] variable.

6. Insert the preceding [link] variable directly before the [onError] link. The [createUploadLink] function shouldn't be inside the [Apollo.from] function.

We've now got the basic Apollo Client setup to support subscriptions via WebSockets.

In Lab 5, I gave the reader some homework to split the complete chat feature into multiple subcomponents. This way, the chat feature would follow the same pattern as we used for the post feed. We split it into multiple components so that it's a clean code base. We're going to use this and have a look at how to implement subscriptions for the chats.

If you haven't implemented the chat's functionality in multiple subcomponents, you can get the working code from the official GitHub repository. I personally recommend you use the code from the repository if it's unclear what the following examples refer to.

Using the chats as an example makes sense because they are, by nature, real-time: they require the application to handle new messages and display them to the recipient. We take care of this in the following steps.

We begin with the main file of our chats feature, which is the [Chats.js] file in the client folder. I've reworked the [render] method so that all the markup that initially came directly from this file is now entirely rendered by other child components. You can see all the changes in the following code:

```
render() {
  const { user } = this.props;
  const { openChats } = this.state;

  return (
    <div className="wrapper">
      <ChatsQuery><ChatsList openChat={this.openChat} user={user}/>
      </ChatsQuery>
      <div className="openChats">
        {openChats.map((chatId, i) =>
          <ChatQuery key={"chatWindow" + chatId} variables={{ chatId
           }}>
            <ChatWindow closeChat={this.closeChat} user={user}/>
          </ChatQuery>
        )}
      </div>
    </div>
  )
}
```

All the changes are listed here:

- We extract the [user] from the properties of the [Chats] component. Consequently, we have to wrap the [Chats] component with the [UserConsumer] component to let it pass the user. You have to apply this change to the [Main.js] file.
- I have split the GraphQL queries we originally sent inside this file into separate query components. One is called [ChatsQuery] and gives us all the chats that the current user is attached to. The other one is called [ChatQuery] and is executed when a chat is opened to request all the messages inside that chat.
- All the inner markup that was previously the GraphQL queries is now also in separate files to improve reusability. Each component that's exported from these files is wrapped in the corresponding query components. The [ChatsList] class renders a list of chats if [ChatsQuery] is successful. The other one is the [ChatWindow] component, which receives a chat property by [ChatQuery] to render all messages. Both receive a user property to show the correct user.
- The [openChat] and [closeChat] functions are executed either by [ChatsList] or the [ChatWindow] component. All other functions from the [Chats] class have been moved to one or both components: [ChatsList] and [ChatWindow].

The changes I have made here had nothing to do with the subscriptions directly, but it's much easier to understand what I'm trying to explain when the code is readable. If you need help implementing these changes by yourself, I recommend you check out the official GitHub repository. All the following examples are based on these changes, but they should be understandable without having the full source code.

More important, however, is [ChatsQuery], which has a special feature. We want to subscribe to the [messageAdded] subscription to listen for new messages. That's possible by using a new function of the Apollo [Query] component.

To continue, first create a separate [ChatsQuery] component. We send the request for all chats like before. The [render] method of the [ChatsQuery] component should look as follows:

```
render() {
  const { children } = this.props;

  return(
    <Query query={GET_CHATS}>
      {({ loading, error, data, subscribeToMore }) => {
        if (loading) return <Loading/>;
        if (error) return <Error><p>{error.message}</p></Error>;

        const { chats } = data;
        return React.Children.map(children, function(child){
          return React.cloneElement(child, { chats, subscribeToMore });
        })
      }}
    </Query>
  )
}
```

The preceding code looks much like all the render methods of the other query components we've written so far. The [GET_CHATS] query requests all chats the current user is related to. The one thing that's different is that we extract a [subscribeToMore] function and pass it as a property to every child.

The [subscribeToMore] function is provided by default with every result of an Apollo Query component. It lets you run an update function whenever a message is created. It works in the same way as the [fetchMore] function. It's best to use the [subscribeToMore] function inside the [ChatsList] component. It already receives the chats property from the preceding [ChatsQuery] component to render the chats panel.

Let's have a look how we can use this function to implement subscriptions on the front end.

Because we pass the [subscribeToMore] function to the [ChatsList] class, we're going to implement this class now. Just follow these steps:

1. We have three necessary dependencies that you should import at the top of the [list.js] file where the [ChatsList] class is saved:

```
import React, { Component } from 'react';
import gql from 'graphql-tag';
import { withApollo } from 'react-apollo';
```

The [withApollo] HoC gives you access to the Apollo Client directly in your component's properties. We only have to export the [ChatsList] class through this HoC. We need access to the client to read and write to the Apollo Client's cache.

2. Parse the GraphQL subscription string. The chats query was executed previously, and [ChatsList] receives the response. The new [messageAdded] subscription has to look as follows:

```
const MESSAGES_SUBSCRIPTION = gql`
  subscription onMessageAdded {
    messageAdded {
      id
      text
      chat {
        id
      }
      user {
        id
        __typename
      }
      __typename
    }
  }
`;
```

The subscription looks exactly like all the other queries or mutations we are using. The only difference is that we request the [__typename] field, as it isn't included in the response of our GraphQL API when using subscriptions. From my point of view, this seems like a bug in the current version of [SubscriptionServer]. You should check whether you still need to do this at the time of reading this course.

We specify the operation type, [subscription], of the request, as you can see in the preceding code. Otherwise, it attempts to execute the default query operation, which leads to an error because there's no [messageAdded] query, only a subscription. The subscription events the client receives when a new message is added holds all fields, as stated in the preceding code.

3. Create the new [ChatsList] class like a standard React component. You can copy the [shorten] function from [Chats.js] and remove it from there. It should look like this:

```
class ChatsList extends Component {
  shorten(text) {
    if(!text.length) {
      return "";
    }
    if (text.length > 12) {
      return text.substring(0, text.length - 9) + '...';
    }
    return text;
  }
}
```

We have to move all the standard functions we were already using for the chats list to this new class.

4. The [usernamesToString] function changes a bit, and I have also added a new [getAvatar] function. When we first created the chats functionality, there was no authentication system. We are now going to rewrite this and use the information we have at our disposal to display the correct data. Copy these functions into our new class:

```
usernamesToString = (userList) => {
  const { user } = this.props;
  var usernamesString = '';
  for(var i = 0; i < userList.length; i++) {
    if(userList[i].username !== user.username) {
      usernamesString += userList[i].username;
    }
    if(i - 1 === userList.length) {
      usernamesString += ', ';
    }
  }
  return usernamesString;
}
getAvatar = (userList) => {
  const { user } = this.props;
  if(userList.length > 2 ) {
    return '/public/group.png';
  } else {
    if(userList[0].id !== user.id) {
      return userList[0].avatar;
    } else {
      return userList[1].avatar;
    }
  }
}
```

The [usernamesToString] function can access the [user] property that the [ChatsList] component receives from its parent through [UserConsumer]. It concatenates the usernames of all users except the logged-in user to display the names in the chats panel. The [getAvatar] function returns the correct image for a chat. It either shows the group image, if there are more than two people involved in a chat, or the avatar image of the second user if exactly two users are involved. This is possible because the getAvatar function can filter by the logged in user

5. The [render] method returns the same markup we had in the [Chats] component. It's now way more readable as it's in a separate file. The code should look as follows:

```
render() {
  const { chats } = this.props;
  return (
    <div className="chats">
      {chats.map((chat, i) =>
        <div key={"chat" + chat.id} className="chat" onClick={() =>
         this.props.openChat(chat.id)}>
          <div className="header">
            <img src={this.getAvatar(chat.users)} />
            <div>
              <h2>{this.shorten(this.usernamesToString(chat.users))
                }
              </h2>
              <span>{chat.lastMessage &&
               this.shorten(chat.lastMessage.text)}</span>
            </div>
          </div>
        </div>
      )}
    </div>
  )
}
```

6. To export your [ChatsList] class correctly, use the [withApollo] HoC:

```
export default withApollo(ChatsList)
```

7. Here's the crucial part. At the moment, the [ChatsList] component is mounted, so we subscribe to the [messageAdded] channel. Only then is the [messageAdded] subscription used to receive new data or, to be

exact, new chat messages. To start subscribing to the GraphQL subscription, we have to add a new method to the [componentDidMount] method:

```
componentDidMount() {
  this.subscribeToNewMessages();
}
```

In the preceding code, we execute a new [subscribeToNewMessages] method inside the [componentDidMount] function of our React component.

It's common to start async operations, such as fetching or listening for a subscription, in the [componentDidMount] method of a React component. You can also use the [componentWillMount] function, but this isn't recommended, as the [componentWillMount] method is executed twice if you support SSR. Furthermore, the [componentWillMount] function first returns after an initial render. There's no way to let the rendering wait until the data has been fetched.

With the [componentDidMount] method, it's clear that the component has rendered at least once without data. The method only executes on the client-side code as the SSR implementation doesn't throw this event because there's no DOM.

We have to add the corresponding [subscribeToNewMessages] method as well. We're going to explain every bit of this function in a moment. Insert the following code into the [ChatsList] class:

```
subscribeToNewMessages = () => {
  const self = this;
  const { user } = this.props;
  this.props.subscribeToMore({
    document: MESSAGES_SUBSCRIPTION,
    updateQuery: (prev, { subscriptionData }) => {
      if (!subscriptionData.data || !prev.chats.length) return prev;

      var index = -1;
      for(var i = 0; i < prev.chats.length; i++) {
        if(prev.chats[i].id ==
        subscriptionData.data.messageAdded.chat.id) {
          index = i;
          break;
        }
      }

      if (index === -1) return prev;

      const newValue = Object.assign({},prev.chats[i], {
        lastMessage: {
          text: subscriptionData.data.messageAdded.text,
          __typename: subscriptionData.data.messageAdded.__typename
        }
      });
      var newList = {chats:[...prev.chats]};
      newList.chats[i] = newValue;
      return newList;
    }
  });
}
```

The preceding [subscribeToNewMessages] method looks very complex, but once we understand its purpose, it's straightforward. We primarily rely on the [subscribeToMore] function here, which was passed from [ChatsQuery] to [ChatsList]. The purpose of this function is to start subscribing to our [addedMessage] channel, and to accept the new data from the subscription and merge it with the current state and cache so that it's reflected directly to the user.

The [document] parameter accepts the parsed GraphQL subscription.

The second parameter is called [updateQuery]. It allows us to insert a function that implements the logic to update the Apollo Client cache with the new data. This function needs to accept a new parameter, which is the previous data from where the [subscribeToMore] function has been passed. In our case, this object contains an array of chats that already exist in the client's cache.

The second parameter holds the new message inside the [subscriptionData] index. The [subscriptionData] object has a [data] property that has a further [messageAdded] field under which the real message that's been created is saved.

We'll quickly go through the logic of the [updateQuery] function so that you can understand how we merge data from a subscription to the application state.

If [subscriptionData.data] is empty or there are no previous chats in the [prev] object, there's nothing to update. In this case, we return the previous data because a message was sent in a chat that the client doesn't have in their cache. Otherwise, we loop through all the previous chats of the [prev] object and find the index of the chat for which the subscription has returned a new message by comparing the chat ids. The found chat's index inside the [prev.chats] array is saved in the [index] variable. If the chat cannot be found, we can check this with the index variable and return the previous data. If we find the chat, we need to update it with the new message. To do this, we compose the chat from the previous data and set [lastMessage] to the new message's text. We do this by using the [Object.assign] function, where the chat and the new message are merged. We save the result in the [newValue] variable. It's important that we also set the returned [__typename] property, because otherwise the Apollo Client throws an error.

Now that we have an object that contains the updated chat in the [newValue] variable, we write it to the client's cache. To write the updated chat to the cache, we return an array of all chats at the end of the [updateQuery] function. Because the [prev] variable is read-only, we can't save the updated chat inside it. We have to create a new array to write it to the cache. We set the [newValue] chat object to the [newList] array at the index where we found the original chat. At the end, we return the [newList] variable. We update the cache that's given to us inside the [prev] object with the new array. Importantly, the new cache has to have the same fields as before. The schema of the return value of the [updateQuery] function must match the initial [ChatsQuery] schema.

You can now test the subscription directly in your browser by starting the application with `npm run server`. If you send a new chat message, it's shown directly in the chat panel on the right-hand side.

We have, however, got one major problem. If you test this with a second user, you'll notice that the [lastMessage] field is updated for both users. That is correct, but the new message isn't visible inside the chat window for the recipient. We've updated the client store for the [ChatsQuery] request, but we haven't added the message to the single [ChatQuery] that's executed when we open a chat window.

We're going to solve this problem by making use of the [withApollo] HoC. The [ChatsList] component has no access to the [ChatQuery] cache directly like with the [ChatsQuery] above. The [withApollo] HoC gives the exported component a [client] property, which allows us to interact directly with the Apollo Client. We can use it to read and write to the whole Apollo Client cache and it isn't limited to only one GraphQL request. Before returning the updated chats array from the [updateQuery] function, we have to read the state of the [ChatQuery] and insert the new data if possible. Insert the following code right before the final return statement inside the [updateQuery] function:

```
try {
  const data = self.props.client.store.cache.readQuery({ query:
```

```
  GET_CHAT, variables: { chatId:
  subscriptionData.data.messageAdded.chat.id } });
  if(user.id !== subscriptionData.data.messageAdded.user.id) {
    data.chat.messages.push(subscriptionData.data.messageAdded);
    self.props.client.store.cache.writeQuery({ query: GET_CHAT,
    variables: { chatId: subscriptionData.data.messageAdded.chat.id },
    data });
  }
} catch(e) {}
```

In the preceding code, we use the [client.store.cache.readQuery] method to read the cache. This accepts the [GET_CHAT] query as one parameter and the chat id of the newly sent message to get a single chat in return. The [GET_CHAT] query is the same request we sent in the [Chats.js] file before, and which the [ChatQuery] component is sending when opening a chat window. We wrap the [readQuery] function in a [try-catch] block because it throws an unhandled error if nothing is found for the specified [query] and [variables]. This can happen if the user hasn't opened a chat window yet and so no data has been requested with the [GET_CHAT] query for this specific chat.

You can test these new changes by viewing the chat window and sending a message from another user account. The new message should appear almost directly for you without the need to refresh the browser.

## Authentication with Apollo Subscriptions

We need to take a look at the back end code that we have written and compare the initialization of `ApolloServer` and `SubscriptionServer`. We have a `context` function for `ApolloServer` that extracts the user from the JWT. It can then be used inside the resolver functions to filter the results by the currently logged in user. For `SubscriptionServer`, there's no such `context` function at the moment. We have to know the currently logged in user to filter the subscription messages for the correct users. We can use the standard WebSockets events, such as `onConnect` or `onOperation`, to implement the authorization of the user.

The `onOperation` function is executed for every WebSocket frame that is sent. The best approach is to implement the authorization in the `onConnect` event in the same way as the `context` function that's taken from `ApolloServer` so that the WebSocket connection is authenticated only once when it's established and not for every frame that's sent.

In `index.js`, from the `subscriptions` folder of the server, add the following code to the first parameter of the `SubscriptionServer` initialization. It accepts an `onConnect` parameter as a function, which is executed whenever a client tries to connect to the `subscriptions` endpoint. Add the code just before the `schema` parameter:

```
onConnect: async (params,socket) => {
  const authorization = params.authToken;
  if(typeof authorization !== typeof undefined) {
    var search = "Bearer";
    var regEx = new RegExp(search, "ig");
    const token = authorization.replace(regEx, '').trim();
    return jwt.verify(token, JWT_SECRET, function(err, result) {
      if(err) {
        throw new Error('Missing auth token!');
      } else {
        return utils.db.models.User.findById(result.id).then((user) =>
        {
          return Object.assign({}, socket.upgradeReq, { user });
        });
      }
    });
  } else {
    throw new Error('Missing auth token!');
  }
},
```

This code is very similar to the [context] function. We rely on the normal JWT authentication but via the connection parameters of the WebSocket. We implement the WebSocket authentication inside the [onConnect] event. In the original [context] function of [ApolloServer], we extract the JWT from the HTTP headers of the request, but here we are using the [params] variable, which is passed in the first parameter.

Before the client finally connects to the WebSocket endpoint, an [onConnect] event is triggered where you can implement special logic for the initial connection. With the first request, we send the JWT because we have configured the Apollo Client to read the JWT to the [authToken] parameter of the [connectionParams] object when [SubscriptionClient] is initialized. That's why we can access the JWT-not from a [request] object, directly but from [params.authToken] in the preceding code. [socket] is also given to us inside the [onConnect] function; there, you can access the initial upgrade request inside the [socket] object. After extracting the JWT from the connection parameters, we can verify it and authenticate the user by that.

At the end of this [onConnect] function, we return the [upgradeReq] variable and the user, just like we do with a normal [context] function for the Apollo Server. Instead of returning the [req] object to the [context] if the user isn't logged in, we are now throwing an error. This is because we only implement subscriptions for entities that require you to be logged in, such as chats or posts. It lets the client try to reconnect until it's authenticated. You can change this behavior to match your needs and let the user connect to the WebSocket. Don't forget, however, that every open connection costs you performance and a user who isn't logged in doesn't need an open connection at least for the use case of **Graphbook**.

We have now identified the user that has connected to our back end with the preceding code, but we're still sending every frame to all users. This is a problem with the resolver functions because they don't use the context yet. Replace the [messageAdded] subscription with the following code in the [resolvers.js] file:

```
messageAdded: {
  subscribe: withFilter(() => pubsub.asyncIterator('messageAdded'),
  (payload, variables, context) => {
    if (payload.messageAdded.UserId !== context.user.id) {
      return Chat.findOne({
        where: {
          id: payload.messageAdded.ChatId
        },
        include: [{
          model: User,
          required: true,
          through: { where: { userId: context.user.id } },
        }],
      }).then((chat) => {
        if(chat !== null) {
          return true;
        }
        return false;
      })
    }
    return false;
  }),
}
```

Earlier in this lab, we imported the [withFilter] function from the [graphql-subscriptions] package. It allows us to wrap [AsyncIterator] with a filter. The purpose of this filter is to conditionally send publications through connections to users who should see the new information. If one user shouldn't receive a publication, the return value of the condition for the [withFilter] function should be false. For all users who should receive a new message, the return value should be true.

[withFilter] accepts the [AsyncIterator] in its first parameter. The second parameter is the function that decides whether a user receives a subscription update.

The filter function is executed for every user that has subscribed to the [messageAdded] channel.

## Notifications with Apollo Subscriptions

In this section, I'll quickly guide you through the second use case for subscriptions. Showing notifications to a user are traditional events that a user should see as you know from Facebook. Instead of relying on the [subscribeToMore] function, we use the [Subscription] component that's provided by Apollo. This component works like the [Query] and [Mutation] components, but for subscriptions.

Follow these steps to get your first [Subscription] component running:

1. Create a [subscriptions] folder inside the client's [components] folder. You can save all subscriptions that you implement using Apollo's [Subscription] component inside this folder.
2. Insert a [messageAdded.js] file into the folder and paste in the following code:

```
import React, { Component } from 'react';
import { Subscription } from 'react-apollo';
import gql from 'graphql-tag';
const MESSAGES_SUBSCRIPTION = gql`
  subscription onMessageAdded {
    messageAdded {
        id
        text
        chat {
          id
        }
        user {
            id
            __typename
        }
        __typename
    }
  }
`;
export default class MessageAddedSubscription extends Component {
  render() {
    const { children } = this.props;
    return(
        <Subscription subscription={MESSAGES_SUBSCRIPTION}>
            {({ data }) => {
                return React.Children.map(children,
                  function(child){
                    return React.cloneElement(child, { data });
                })
            }}
        </Subscription>
    )
  }
}
```

The general workflow for the [Subscription] component is the same as for the [Mutation] and [Query] components. First, we parse the subscription with the [graphql-tag] package. The [render] method of the [MessageAddedSubscription] class returns the [Subscription] component. The only difference is that we don't use a loading or error state. You could get access to both, but as we're using WebSockets, the loading state only becomes true when a new message arrives, which isn't useful. Furthermore, the error property could be used to display alerts to the user, but this isn't required. In the [render] method, we pass the data field to all underlying children. By default, it's an empty object. It's filled with [data] when a new message arrives through the subscription.

3. Because we want to show notifications to the user when a new message is received, we install a package that takes care of showing pop-up notifications. Install it using npm:

```
npm install --save react-toastify
```

4. To set up [react-toastify], add a [ToastContainer] component to a global point of the application where all notifications are rendered. This container isn't only used for the notifications for new messages but for all notifications, so choose wisely. I decided to attach [ToastContainer] to the [Chats.js] file. Import the dependency at the top of it:

```
import { ToastContainer } from 'react-toastify';
```

Inside the [render] method, the first thing to render should be [ToastContainer]. Add it like in the following code:

```
<div className="wrapper">
  <ToastContainer/>
```

5. To handle the subscription data, we need a child component that gets the data as a property. To do this, create a [notification.js] file inside the [chats component] folder. The file should look as follows:

```
import React, { Component } from 'react';
import { toast } from 'react-toastify';

export default class ChatNotification extends Component {
  componentWillReceiveProps(props) {
      if(typeof props.data !== typeof undefined && typeof
      props.data.messageAdded !== typeof undefined && props.data
      && props.data.messageAdded)
      toast(props.data.messageAdded.text, { position:
      toast.POSITION.TOP_LEFT });
  }
  render() {
      return (null);
  }
}
```

We only import React and the [react-toastify] package. The [render] method of the [ChatNotification] class returns [null] because we don't render anything directly through this component. Instead, we listen for the [componentWillReceiveProps] method for new data from the subscription. If the properties passed to this class were a filled with [data] property, we can use the [react-toastify] package.

To display a new notification, we execute the [toast] function from the [react-toastify] package with the text to show as the first parameter. The second parameter takes optional settings to indicate how the notification should display. I've given the top-left corner as the position for all notifications because the right part of the screen is already pretty full.

6. Import [ChatNotification] and the [MessageAddedSubscription] component inside the [Chats.js] file:

```
import MessageAddedSubscription from './components/subscriptions
/messageAdded';
import ChatNotification from './components/chat/notification';
```

7. Include both components in the [render] method of the [Chats] class from the [Chats.js] file. The final method looks like this:

```
return (
  <div className="wrapper">
      <ToastContainer/>
      <MessageAddedSubscription><ChatNotification/>
      </MessageAddedSubscription>
      <ChatsQuery><ChatsList openChat={this.openChat} user=
      {user}/></ChatsQuery>
      <div className="openChats">
          {openChats.map((chatId, i) =>
              <ChatQuery key={"chatWindow" + chatId} variables={{
               chatId }}>
                  <ChatWindow closeChat={this.closeChat} user=
                  {user}/>
              </ChatQuery>
          )}
      </div>
  </div>
)
```

I've wrapped the [ChatNotification] component inside the [MessageAddedSubscription] component. The subscription component triggers a new notification every time it receives new data over the WebSocket and updates the properties of the [ChatNotification] component.

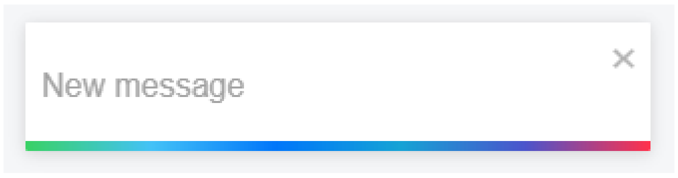
8. Add a small CSS rule and import the CSS rules of the [react-toastify] package. Import the CSS in the [App.js] file:

```
import 'react-toastify/dist/ReactToastify.css';
```

Then, add these few lines to the custom [style.css] file:

```
.Toastify__toast-container--top-left {
  top: 4em !important;
}
```

You can see an example of a notification in the following screenshot:


New message

The entire subscriptions topic is complex, but we managed to implement it for two use cases and thus provided the user with significant improvements to our application.

## Summary

This lab aimed to offer the user a real-time user interface that allows them to chat comfortably with other users. We also looked at how to make this UI extendable. You learned how to set up subscriptions with any Apollo or GraphQL back end for all entities. We also implemented WebSocket-specific authentication to filter publications so that they only arrive to the correct user.

In the next lab, you'll learn how to verify and test the correct functionality of your application by implementing automated testing for your code.