To implement the functionality where a notification is immediately sent to the requested user upon creating or updating a request, you can follow these steps:

1. **Define the Notification Entity:** Update your `RequestNotification` entity to include the necessary fields and relationships.

```
// request-notification.entity.ts
import { ObjectType, Field } from '@nestjs/graphql';
import { Entity, Column, JoinColumn, ManyToOne } from 'typeorm';
import { Notification } from './notification.entity';
import { Request } from './request.entity';
import { User } from './user.entity';

@Entity()
@ObjectType()
export class RequestNotification extends Notification {
    @Column({ nullable: true })
    @Field({ nullable: true })
    requestId?: string;

    @ManyToOne(() => Request, { nullable: true, onDelete: 'SET NULL' })
    @JoinColumn({ name: 'requestId', referencedColumnName: 'id' })
    @Field(() => Request, { nullable: true })
    request?: Request;
}
```

2. **Update the Request Entity:** Add a relationship to `RequestNotification` in the `Request` entity.

```
// request.entity.ts
import { ObjectType, Field } from '@nestjs/graphql';
import { Entity, Column, ManyToOne, JoinColumn, ManyToMany, AfterInsert,
AfterUpdate } from 'typeorm';
import { StreamLineEntity } from './streamline.entity';
import { File } from './file.entity';
import { RequestItem } from './request-items.entity';
import { RequestStatus } from './enum/requestStatus';
import { User } from './user.entity';
import { RequestType } from './enum/requestType';
import { RequestNotification } from './request-notification.entity';

@Entity()
@ObjectType()
export class Request extends StreamLineEntity {
    // ... (other fields)

    @OneToMany(() => RequestNotification, (notification) =>
notification.request)
    notifications: RequestNotification[];
```

```
    // ... (other methods)

    @AfterInsert()
    @AfterUpdate()
    async sendNotification() {
        // Create or update notification and send it to requested user
        const notification = new RequestNotification();
        notification.requestId = this.id;
        await notification.save();
        // Add logic to send the notification to the requested user (e.g.,
use a message queue, WebSocket, etc.)
        // This can include using GraphQL subscriptions or a message queue
like RabbitMQ or Kafka
        // You might want to consider implementing a separate notification
service for scalability
    }
}
```

Note: The `@AfterInsert` and `@AfterUpdate` decorators are used to trigger the `sendNotification` method after a request is inserted or updated.

3. **GraphQL Subscriptions:** Depending on your setup, you can use GraphQL subscriptions to notify the client about the new notifications. If you are using Apollo Server with subscriptions, you can define a subscription for new notifications.

```
// subscription.resolver.ts
import { Subscription, Resolver } from '@nestjs/graphql';
import { Inject, Logger } from '@nestjs/common';
import { PubSubEngine } from 'graphql-subscriptions';
import { RequestNotification } from './request-notification.entity';

@Resolver(() => RequestNotification)
export class RequestNotificationResolver {
    constructor(
        @Inject('PUB_SUB') private pubSub: PubSubEngine,
    ) {}

    @Subscription(() => RequestNotification)
    async notificationCreated() {
        return this.pubSub.asyncIterator('notificationCreated');
    }

    // Other subscription methods can be added based on your requirements
}
```

When a new notification is created, you can publish an event to the subscription channel (`'notificationCreated'`) to notify clients.

4. **Send Notifications to Requested User:** Implement the logic to send the notification to the requested user. Depending on your requirements, you can use different approaches such as WebSockets,

GraphQL subscriptions, or a message queue system like RabbitMQ or Kafka.

If you are using Apollo Server with WebSocket subscriptions, you can use the `withFilter` function from `graphql-subscriptions` to filter notifications based on the requested user.

```
// request-notification.resolver.ts
import { Resolver, Subscription, Inject } from '@nestjs/graphql';
import { PubSub } from 'graphql-subscriptions';
import { withFilter } from 'graphql-subscriptions';
import { RequestNotification } from './request-notification.entity';

@Resolver(() => RequestNotification)
export class RequestNotificationResolver {
    constructor(
        @Inject('PUB_SUB') private pubSub: PubSub,
    ) {}

    @Subscription(() => RequestNotification, {
        filter: withFilter(
            (payload, variables) =>
payload.notificationCreated.request.requestedUserId ===
variables.requestedUserId,
            (payload, variables) => {
                // Check if the requestedUserId matches the user
```

's ID return payload.notificationCreated.request.requestedUserId === variables.requestedUserId; },), }) async notificationCreated() { return this.pubSub.asyncIterator('notificationCreated'); } }

```
This example assumes that you have set up a PubSub engine for GraphQL
subscriptions (e.g., using Apollo Server with Redis PubSub). Adjust the code based
on your specific setup.

5. **Update AppModule:**
Update your `AppModule` to include the GraphQL subscriptions module and provide a
PubSub instance.

```typescript
// app.module.ts
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { RequestModule } from './request/request.module';
import { RequestNotificationModule } from './request-notification/request-
notification.module';
import { PubSub } from 'graphql-subscriptions';

@Module({
    imports: [
        GraphQLModule.forRoot({
```
```

```
            autoSchemaFile: 'schema.gql',
            installSubscriptionHandlers: true,
        }),
        RequestModule,
        RequestNotificationModule,
    ],
    providers: [
        {
            provide: 'PUB_SUB',
            useValue: new PubSub(),
        },
    ],
})
export class AppModule {}
```

6. **Update RequestNotificationModule:** If you have a separate module for `RequestNotification`, ensure it is properly configured.

```
// request-notification.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { RequestNotification } from './request-notification.entity';
import { RequestNotificationResolver } from './request-
notification.resolver';

@Module({
    imports: [TypeOrmModule.forFeature([RequestNotification])],
    providers: [RequestNotificationResolver],
})
export class RequestNotificationModule {}
```

   Make sure that the `RequestNotificationModule` is also included in the `imports` array of the `AppModule`.

With these steps, you've set up a basic notification system using GraphQL subscriptions. Notifications are created and sent to the requested user whenever a new request is inserted or updated. The subscription mechanism ensures that clients are notified in real-time about new notifications.

Certainly! If you're using Apollo GraphQL Gateway and want to implement subscriptions across multiple services, you'll need to make adjustments to ensure that your subscription events are communicated across services. Here's how you can modify the setup accordingly:

1. **PubSub Setup in Gateway:**

   In your Apollo Gateway (probably in a service that orchestrates multiple schemas), you need to set up a common PubSub instance that is shared among your services. You can use an external PubSub server or use an in-memory PubSub for simplicity.

```
// apollo-gateway/src/pubsub.ts (example)
import { PubSub } from 'apollo-server-express';

export const commonPubSub = new PubSub();
```

Then, in your `AppModule` or main entry point for the gateway:

```
// apollo-gateway/src/main.ts
import { ApolloGateway } from '@apollo/gateway';
import { commonPubSub } from './pubsub';

const gateway = new ApolloGateway({
    serviceList: [
        // Your service configurations here
    ],
    buildService({ url }) {
        return new RemoteGraphQLDataSource({
            url,
            willSendRequest({ request, context }) {
                // Attach the common PubSub instance to the context
                context.pubsub = commonPubSub;
            },
        });
    },
});

// ... rest of the gateway setup
```

2. **Service-Specific Subscriptions:**

   In each service, when a new notification is created or updated, you'll publish the notification event to the common PubSub instance. Services should also subscribe to the common PubSub instance to listen for events.

```
// request-service/src/request.service.ts
import { PubSub, withFilter } from 'graphql-subscriptions';
import { commonPubSub } from '../apollo-gateway/pubsub';

const pubSub = new PubSub();

// ...

@Resolver(() => RequestNotification)
export class RequestNotificationResolver {
    @Mutation(() => RequestNotification)
    async createNotification() {
        // ... create notification logic ...
```

```
        // Publish the notification event to the common PubSub
        commonPubSub.publish('notificationCreated', {
            notificationCreated: { /* notification data */ },
        });

        return createdNotification;
    }

    @Subscription(() => RequestNotification, {
        filter: withFilter(
            (_, __, { pubsub }) =>
pubsub.asyncIterator('notificationCreated'),
            (payload, variables) => {
                // Your filtering logic based on variables
                return true;
            },
        ),
    })
    notificationCreated() {
        // Subscribe to the common PubSub instance
        return commonPubSub.asyncIterator('notificationCreated');
    }
}
```

3. **GraphQL Subscription in Apollo Client:**

   In your Apollo Client (frontend), ensure that your subscription setup points to the GraphQL Gateway's WebSocket endpoint.

```
// apollo-client/src/apolloClient.ts
import { ApolloClient, InMemoryCache, split } from '@apollo/client';
import { WebSocketLink } from '@apollo/client/link/ws';
import { getMainDefinition } from '@apollo/client/utilities';

const httpLink = createHttpLink({
    uri: 'http://your-gateway-url/graphql',  // Adjust this to your GraphQL
Gateway URL
});

const wsLink = new WebSocketLink({
    uri: `ws://your-gateway-url/graphql`,  // Adjust this to your GraphQL
Gateway WebSocket URL
    options: {
        reconnect: true,
    },
});

const splitLink = split(
    ({ query }) => {
        const definition = getMainDefinition(query);
        return (
```

```
            definition.kind === 'OperationDefinition' &&
            definition.operation === 'subscription'
        );
    },
    wsLink,
    httpLink,
);

const client = new ApolloClient({
    link: splitLink,
    cache: new InMemoryCache(),
});

export default client;
```

Ensure that `ws://your-gateway-url/graphql` is pointing to the WebSocket URL of your GraphQL Gateway.

With these adjustments, you should have a working GraphQL subscription setup that communicates across services through the common PubSub instance. The key is to ensure that all services are using the same PubSub instance and that your GraphQL Gateway orchestrates the communication effectively.