Request Scoped Data Loading in GraphQL Java

Ashu Gairola
Nov 1 2018

Abstract

A core capability that the graphql-java framework provides is to efficiently load data from downstream services. It is implemented in the library java-dataloader. This white paper describes the contributions made to open source frameworks to enable http request scoped data loading and request scoped caching.

Problem Statement

The underlying core frameworks as described in the abstract - graphl-java and java-dataloader provide capabilities for efficient data loading. However the tools available for wiring graphql-java constructs in to a DI / Web framework like Spring do so in a way that makes leveraging dataloding capabilities non trivial. These frameworks are - graphql-spring-boot and graphql-java-servlet.

Background

1. Data Loading. The java-dataloader is a java port of facebook-dataloader. It is a utility that does intelligent evaluation of the requested graphql query to optimize message communication necessary for loading the data requested by the query. In porting over the dataloader to java, each element of the query is independently evaluated as a completableFuture. The completableFutures are dispatched only when the query evaluation has reached a point where it can not be executed further without the data fetch. This is done in a graphql query instrumentation implementation - DataLoaderDispatcherInstrumentation; resulting in efficient batching and caching the data fetches.
2. Caching in a Request Scope. For serving web requests it is important to scope the data caching within the web request scope. Independent execution of a graph of query elements

as described above may result in the same data getting feched multiple times by them independent executions. For this reason it is important to scope the cache of data fetched within a web request scope such that independent query element executions for the same web request do not result in multiple data fetch messages downstream. Moreover, in a concurrent web request environment the data may vary per web request based on the context of the web request. So it is just as important for the the data cache of a web request to not be visible outside the scope of that particular web request. A graphql execution of a web request may in fact need to fetch the same data that is cached in the scope of a different concurrent web request.

3. GraphQL Java Servlet. Is a java module that provides a servlet implementation for serving GraphQL requests over web. It exposes HTTP URIs that can be invoked by clients/callers using standard HTTP methods. The servlet creates a graphql execution based on provided dependencies like the graphql schema, the executionStrategy for query/mutation/subscription, and the provided instrumentation. An ExecutionInput is composed that contains the actual requested query along with any graphql inputs / attributes. A GraphQLContext is also created and passed along to the graphql execution. It holds the HttpServletRequest itself and other supporting metadata for execution of the graphql query.
4. Spring Boot Graphql. Is an open source project for GraphQL-Java that does the boiler plate autowiring of graphql-servlet, query resolvers, data resolvers and instrumentation for spring-boot based web applications. Spring framework provides constructs for web context aware applications to enable Request scoped beans. These are however meant for use in blocking thread execution models as the request context and relevant spring beans are stored as ThreadLocal variables. GraphQL JAVA dataloader is implemented with completableFutures as described above that makes Spring framework RequestScope unusable. Even if a mechanism is devised to enable request scoped spring beans in a non blocking thread execution model, the wiring of the data cache in grapqhl execution context is not spring enabled and would require changes to the wiring implementation itself

Solution

It is recommended to create a new instance for DataLoaderRegistry for each graphql query execution to avoid mixing up of the caches created for the queries. This instance has to be passed to the DataLoaderDispatcherInstrumentation that is set for the graphQL query execution. The most elegant way of doing this requires the following changes:

1. Enhance the GraphQLContext class that is defined in graphql-java-servlet module to include the DataLoaderRegistry. This will make the registry accessesible from the DataLoaders/Resolvers.
2. Update AbstractGraphQLHttpServlet to fetch the correct Instrumentation as needed based on the GraphQLContext provided. if a DataLoaderRegistry is defined and instantiated in the GraphQLContext then use the DataLoaderDispatcherInstrumentation.
3. Any other provided instrumenation such as TracingInstrumentation or FieldValidationInstrumentation may require correctly chaining the instrumentation(s) applicable.
4. Update GraphQLContextBuilder to support the enhancements made in GraphQLContext for DataLoaderRegistry.
5. The parent application is expected to customise GraphQLContextBuilder and instantiate a new DataLoaderRegistry for each GraphQLContext that is built. If it is a spring boot application utilizing spring-boot-graphql module then the application just need annotate it as a spring component.

Conclusion

This solution achieves the request scope necessary for DataLoaderRegistry which also acts as he cache for DataLoaders/Resolvers. Here is a sample of how a parent application utilizing spring-boot-graphql can use request scoped DataLoader:

```java
public class CustomGraphQLContextBuilder implements GraphQLContextBuilder {

    private final DataLoader userDataLoader;

    public CustomGraphQLContextBuilder(DataLoader userDataLoader) {
        this.userDataLoader = userDataLoader;
    }

    @Override
    public GraphQLContext build(HttpServletRequest req) {
        GraphQLContext context = new GraphQLContext(req);
        context.setDataLoaderRegistry(buildDataLoaderRegistry());

        return context;
    }

    @Override
    public GraphQLContext build() {
        GraphQLContext context = new GraphQLContext();
        context.setDataLoaderRegistry(buildDataLoaderRegistry());

        return context;
    }

    @Override
    public GraphQLContext build(HandshakeRequest request) {
        GraphQLContext context = new GraphQLContext(request);
        context.setDataLoaderRegistry(buildDataLoaderRegistry());

        return context;
    }

    private DataLoaderRegistry buildDataLoaderRegistry() {
        DataLoaderRegistry dataLoaderRegistry = new DataLoaderRegistry();
        dataLoaderRegistry.register("userDataLoader", userDataLoader);
        return dataLoaderRegistry;
```

```
    }
}
```

It is then possible to access the DataLoader in the resolvers by accessing the [DataLoaderRegistry] from context. For eg:

```
public CompletableFuture<String> getEmailAddress(User user, DataFetchingEnvironment
dfe) { // User is the graphQL type
        final DataLoader<String, UserDetail> userDataloader =

dfe.getContext().getDataLoaderRegistry().get().getDataLoader("userDataLoader"); //
UserDetail is the data that is loaded

        return userDataloader.load(User.getName())
                .thenApply(userDetail -> userDetail != null ?
userDetail.getEmailAddress() : null);
    }
```

References

GitHub documentation in Readme. Example in github. All the reference tickets

1. https://github.com/graphql-java/java-dataloader