Investigating GraphQL: Evaluating Its Benefits to Web Service Applications

Justin Hendy

July 2025

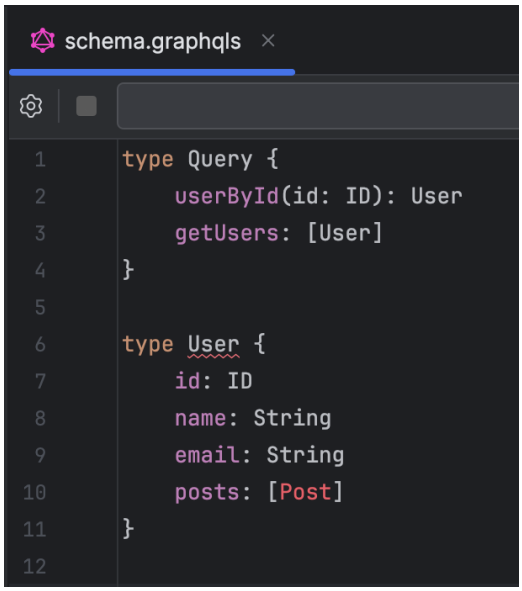
GraphQL is a query language most commonly used with web APIs, providing a flexible and efficient system for requesting and combining data from multiple related objects via a single query. This report provides a brief evaluation of the benefits of GraphQL when used in conjunction with web services, illustrated through code snippets related to the development of a simple GraphQL endpoint.

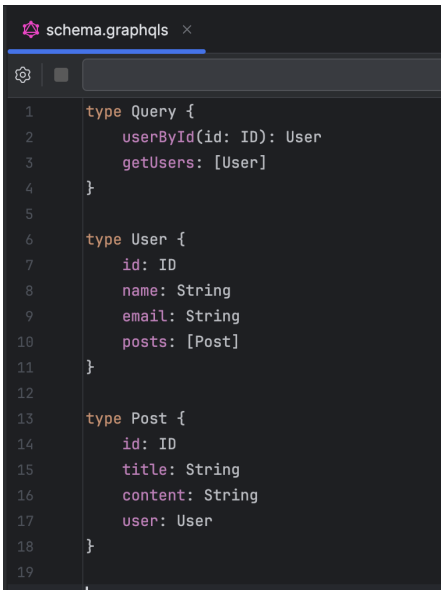
REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) are well established methods for developing web service APIs. Both methods have their strengths; however, it is their challenges or drawbacks that are of particular interest to this report. Common issues associated with REST and/or SOAP include under-fetching/over-fetching, as well as the use of numerous endpoints and restrictive API contracts.

GraphQL, developed by Facebook, centres around the principles of a single endpoint and the use of flexible data queries. GraphQL uses a statically or strongly typed schema to define data types, relationships and operations; the statically typed nature provides compile-time type checking and error detection.

GraphQL aims to alleviate or improve upon several issues associated with traditional web service API development. Under- and over-fetching refer to the delivery of too little or too much data in response to an API request, which occurs with REST and SOAP due to the fixed nature of the response. When issuing a GraphQL query, a client can specify the exact data required and, in return, receive only this data in the response. GraphQL also reduces requests to a single endpoint by consolidating queries from related resources. Often, REST APIs require multiple queries to collect related data, and although SOAP provides a great deal of structure, this can inhibit the ability to provide bespoke data collections. Through the use of strongly typed, flexible schemas, GraphQL ensures that endpoint data can easily evolve as requirements change.

The following scenario demonstrates the use of GraphQL with Spring Boot and is an adaptation of a GraphQL tutorial provided by spring.io [1]. The scenario is relatively simple and involves two types: User and Post, where a single user can make many posts. Here is the GraphQL schema used to implement this scenario:





The two images above demonstrate the creation of a GraphQL schema. All schemas must have a top-level type of query, which is like a REST GET request and provides an entry point to the API. The images also demonstrate type error checking, where in the first image on the left, there is no Post type definition. Each type contains the necessary data fields, including ID, name, email, and other relevant information.

The following code snippets demonstrate how to set up a Spring Boot project to interact with this GraphQL API.

```

@Controller
public class UserController {

    @QueryMapping
    public User userById(@Argument String id) {
        return User.getById(id);
    }

    @QueryMapping
    public List<User> getUsers() {
        return User.getUsers();
    }

    @SchemaMapping(typeName = "User", field = "posts")
    public List<Post> post(User user) {
        return Post.getPostsByUserId(user.id());
    }
}
```

The above controller maps the two query types set at the start of the schema, using the @QueryMapping annotation, to two methods set in two Java records representing a User and a Post (see following code snippet). The userById method requires a string ID argument and returns a user with a specific ID, whereas the getUsers method does not require any arguments and returns all users. The @SchemaMapping annotation and the attached method provide a link between users and their posts, returning all posts associated with the user passed as an argument.

```java

public record User(String id, String name, String email) {

    private static List<User> users = Arrays.asList(
        new User("user-1", "Justin", "justin@example.com"),
        new User("user-2", "Brad", "brad@email.com"),
        new User("user-3", "Iestyn", "iestyn@example-email.com")
    );

    public static User getById(String id) {
        return users.stream()
            .filter(user -> user.id().equals(id))
            .findFirst()
            .orElse(null);
    }

    public static List<User> getUsers() {
        return users;
    }
}

public record Post(String id, String title, String content, String userId) {

    private static List<Post> posts = Arrays.asList(
        new Post("post-1", "Using GraphQL", "Some content about GraphQL", "user-1"),
        new Post("post-2", "Web Services", "Some content about Web Services", "user-2"),
        new Post("post-3", "GraphQL vs REST vs SOAP", "Some content about comparing GraphQL, REST and SOAP", "user-1")
    );

    public static List<Post> getPostsByUserId(String userId) {
        return posts.stream()
            .filter(post -> post.userId().equals(userId))
            .toList();
    }
}
```

One of the key benefits of GraphQL is its ability to work with any data source, including databases, static in-memory lists, and external sources [1]. The Java record type offers a convenient solution for creating a static, in-memory list of users and posts (see code above). The Java records also contain the methods used by the UserController, specifically getById and getUsers for the User record and getPostsByUserId for the Post record. With the GraphQL schema, the UserController and the two Java records, it is now very straightforward to write a range of queries to extract data in a multitude of shapes explicitly set by the client.

The following pages present a series of queries sent to this GraphQL API, along with their corresponding responses, and a description of each request and response.

The two queries and their subsequent responses below utilise the userById query type as defined in the GraphQL schema. In both Query One and Query Two, the userId passed as an argument is "user-1". The rest of the query defines the data required for this request. Not only is it possible to nest related posts for a user within the query, but it is also relatively straightforward to amend the data requirements and retrieve the exact data needed. Specifically, Query two only requests the user's name and the title of their posts. The ability to define the exact data required as part of a request eliminates the issues of under-fetching and over-fetching associated with REST and SOAP. The ability to nest the user's post data in GraphQL is very efficient when compared to the multiple requests this same outcome would require when carried out by a REST API.

```
# Query One request
query userPostInfo {
  userById(id: "user-1") {
    id
    name
    email
    posts {
      id
      title
      content
    }
  }
}
# Query One response
{
  "data": {
    "userById": {
      "id": "user-1",
      "name": "Justin",
      "email": "justin@example.com",
      "posts": [
        {
          "id": "post-1",
          "title": "Using GraphQL",
          "content": "Some content about GraphQL"
        },
        {
          "id": "post-3",
          "title": "GraphQL vs REST vs SOAP",
          "content": "Some content about comparing GraphQL, REST and SOAP"
        }
      ]
    }
  }
}
```

```
# Query Two request
query userPostInfo {
  userById(id: "user-1") {
    name
    posts {
      title
    }
  }
}
# Query Two response
{
  "data": {
    "userById": {
      "name": "Justin",
      "posts": [
        {
          "title": "Using GraphQL"
        },
        {
          "title": "GraphQL vs REST vs SOAP"
        }
      ]
    }
  }
}
```

Query three below demonstrates how a single GraphQL endpoint can provide a client with multiple query types. In this instance, the getUsers query returns all users, along with the requested user data and their post data. Within a REST API, facilitating a request of this nature requires multiple endpoints.

```
# Query Three request
query users {
  getUsers {
    id
    name
    email
  }
}
# Query Three response
{
  "data": {
    "getUsers": [
      {
        "id": "user-1",
        "name": "Justin",
        "email": "justin@example.com"
      },
      {
        "id": "user-2",
        "name": "Brad",
        "email": "brad@email.com"
      },
      {
        "id": "user-3",
        "name": "Iestyn",
        "email": "iestyn@example-email.com"
      }
    ]
  }
}
```

Finally, query four further demonstrates the flexibility of GraphQL by including post data along with the getUsers query.

```
# Query Four request
query usersAndPosts {
  getUsers {
    name
    posts {
      title
    }
  }
}
# Query Four response
{
  "data": {
    "getUsers": [
      {
        "name": "Justin",
        "posts": [
          {
            "title": "Using GraphQL"
          },
          {
            "title": "GraphQL vs REST vs SOAP"
          }
        ]
      },
      {
        "name": "Brad",
        "posts": [
          {
            "title": "Web Services"
          }
        ]
      },
      {
        "name": "Iestyn",
```

```
        "posts": []
      }
    ]
  }
}
```

To summarise, the key benefits of GraphQL, as demonstrated here, are its ability to fetch data precisely required by a client, provide nested related data within a single request, implement a strongly typed schema, expose a single endpoint that supports multiple query types, and offer frontend flexibility, which in turn results in fewer backend amendments in order to meet changing client demands.

References

[1] Spring.io. (2025) GraphQL with Spring Boot. Accessed: 2025-07-15. [Online]. Available: https://spring.io/guides/tutorials/graphql/