**GraphQL:**

GraphQL is a query language for your API, and a server-side runtime for executing queries using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.

A GraphQL service is created by defining types and fields on those types, then providing functions for each field on each type.

**Why graphQL is called GraphQL?**

**Graph:** The "Graph" part of the name refers to the data graph structure that GraphQL operates on. In GraphQL, your application's data is represented as a graph, where each node in the graph represents an object, and edges represent the relationships between objects. This graph-based approach allows clients to request precisely the data they need, including nested and interconnected data, in a single query.

**QL (Query Language):** The "QL" part stands for "Query Language." GraphQL provides a query language that clients use to specify the data they want to retrieve from the server. This query language allows clients to express their data requirements in a flexible and efficient manner, enabling them to retrieve only the data they need and nothing more. The query language supports features like selecting specific fields, filtering data, specifying relationships, and more.

**Traditional is REST:**

REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. It was introduced by Roy Fielding in his doctoral dissertation in 2000. The term "RESTful" is used to describe web services that adhere to the principles of REST.

RESTful services are based on a few key principles:

1. **Client-Server Architecture:** The client and server are separate entities that communicate over a network. They are independent of each other and can evolve separately.
2. **Statelessness:** Each request from a client to the server must contain all the information necessary to understand and process the request. The server does not store any client state between requests. This improves scalability and simplifies server implementation.
3. **Uniform Interface:** The interface between the client and server is uniform, meaning that the same interface should be used for all interactions. This simplifies the architecture and allows different clients to interact with the server in a consistent manner.
4. **Cacheability:** Responses from the server can be cached by clients to improve performance. This reduces the need for repeated requests to the server for the same data.
5. **Layered System:** The architecture is composed of multiple layers, with each layer responsible for a specific aspect of the system. This allows for scalability and flexibility in the system design.

The term "RESTful" is derived from the phrase "Representational State Transfer," which emphasizes the transfer of representations of resources (such as JSON or XML) between client and server. The term "REST" itself reflects the idea that interactions with a web service should be based on the transfer of representations of resources and the state of those resources between the client and server. Hence, the term "REST APIs" is used to describe APIs that follow the principles of REST.

User

**You**

why called rest

ChatGPT

**ChatGPT**

The term "REST" stands for Representational State Transfer. It was coined by Roy Fielding in his doctoral dissertation in 2000, where he introduced the REST architectural style. The name "Representational State Transfer" emphasizes the idea of transferring representations of resources (such as JSON or XML) between the client and server.

The term "RESTful" is derived from REST and is used to describe web services or APIs that adhere to the principles of REST. These principles include using a client-server architecture, statelessness, a uniform interface, cacheability, and a layered system.

So, "REST" is called "REST" because it emphasizes the transfer of representations of resources between clients and servers, and "RESTful" is used to describe systems or services that follow the REST architectural style.

User

**You**

graphql vs rest

ChatGPT

**ChatGPT**

GraphQL and REST are both approaches for designing and interacting with APIs, but they have different philosophies and characteristics:

1. **Data Fetching:**
   * **REST:** In RESTful APIs, endpoints are predefined, and each endpoint typically corresponds to a specific resource. Clients make HTTP requests to these endpoints to fetch or manipulate data. The server decides what data to return.
   * **GraphQL:** With GraphQL, clients can specify exactly what data they need by sending queries to the server. The server then processes these queries and returns only the requested data in the exact structure specified by the client.
2. **Data Structure:**
   * **REST:** Data is typically structured according to predefined resource endpoints. Clients may need to make multiple requests to different endpoints to assemble the required data.
   * **GraphQL:** Clients can request nested or related data in a single query, allowing for more efficient data retrieval and reducing over-fetching or under-fetching of data.
3. **Flexibility:**
   * **REST:** REST APIs provide a fixed set of endpoints, and any changes to the data structure or relationships may require changes to the API itself.
   * **GraphQL:** GraphQL APIs offer more flexibility because clients can request only the data they need, and the structure of the data is determined by the client's query rather than the server's endpoints.
4. **Caching:**
   * **REST:** RESTful APIs can leverage HTTP caching mechanisms using methods like ETag or Last-Modified headers.
   * **GraphQL:** Because GraphQL queries are typically sent via POST requests, caching can be more complex and often requires client-side caching strategies.
5. **Tooling:**
   * **REST:** There are numerous tools and frameworks available for building and consuming RESTful APIs, given its long-standing presence and familiarity.
   * **GraphQL:** GraphQL has a growing ecosystem of tools and libraries, but it may not be as mature as REST in terms of tooling and community support.

In summary, GraphQL offers more flexibility and efficiency in data fetching compared to REST, especially for complex applications where data requirements may vary. However, the choice between GraphQL and REST depends on factors such as the specific use case, development team preferences, and existing infrastructure.

*Problems that GrahpQL solves are*: **Over-Fetching and under-fetching**

User

**You**

pros and cons

ChatGPT

**ChatGPT**

Sure, here are the pros and cons of GraphQL and REST:

**GraphQL:**

Pros:

1. **Efficient Data Fetching:** Clients can request only the data they need, reducing over-fetching and under-fetching.
2. **Flexible Data Structure:** Clients can define the structure of the response, allowing for easier evolution of the API.
3. **Reduced Number of Requests:** Clients can retrieve nested or related data in a single query, reducing the number of requests required.
4. **Strongly Typed Schema:** GraphQL APIs have a strongly typed schema, providing better documentation and type safety.
5. **Tooling:** GraphQL has a growing ecosystem of tools and libraries for development, testing, and documentation.

Cons:

1. **Complexity:** Implementing GraphQL servers can be more complex than REST due to query parsing, validation, and execution.
2. **Caching:** Caching can be more challenging with GraphQL due to the dynamic nature of queries and lack of standardized caching mechanisms.
3. **Learning Curve:** Developers may need time to learn the GraphQL query language and best practices.
4. **Performance:** Poorly constructed queries or inefficient resolvers can lead to performance issues.
5. **Versioning:** GraphQL does not have built-in versioning mechanisms, which can complicate API evolution and backward compatibility.

**REST:**

Pros:

1. **Simplicity:** RESTful APIs have a straightforward architecture with predefined endpoints for resources.
2. **Caching:** REST leverages HTTP caching mechanisms, such as ETag and Last-Modified headers, for efficient caching.
3. **Familiarity:** REST has been around for a long time and is widely understood by developers, with abundant tooling and documentation.
4. **Statelessness:** RESTful APIs are stateless, making them scalable and easier to cache.
5. **HTTP Verbs:** RESTful APIs use standard HTTP verbs (GET, POST, PUT, DELETE), making them easy to understand and work with.

Cons:

1. **Over-fetching/Under-fetching:** Clients may receive more or less data than needed, leading to inefficiencies.
2. **Limited Flexibility:** APIs may need to expose multiple endpoints to support different data requirements, leading to endpoint proliferation.
3. **Versioning:** Changing the data structure or relationships may require versioning of endpoints, leading to maintenance overhead.
4. **Documentation:** REST APIs may require more comprehensive documentation to describe endpoint semantics and data structures.
5. **Understandability:** As APIs grow in complexity, it can become challenging to understand the relationships between resources and endpoints.

Ultimately, the choice between GraphQL and REST depends on factors such as the nature of the application, data requirements, team expertise, and scalability needs. Each approach has its strengths and weaknesses, and the decision should be based on the specific requirements of the project

|  |
| --- |
| User  **You**  underfetching vs overfetching  ChatGPT  **ChatGPT**  Underfetching and overfetching are two common issues in API design:   1. **Underfetching**: This occurs when an API endpoint doesn't provide enough data in a single request, leading to the need for additional requests to fetch the required information. For example, imagine an API endpoint that returns basic information about a user, but doesn't include their associated posts. To get the posts, you'd need to make another request to a different endpoint. 2. **Overfetching**: This happens when an API endpoint provides more data than is actually needed for a particular use case. For instance, if an endpoint returns extensive information about a user along with all their posts, comments, and related data, but your application only needs the user's basic information, fetching all the extra data would be considered overfetching.   GraphQL addresses these issues by allowing clients to request precisely the data they need, structured according to the schema defined by the GraphQL server. Clients can specify their data requirements in the query, and the server responds with exactly that data, eliminating both underfetching and overfetching.  In summary, underfetching occurs when there's not enough data provided, leading to multiple requests, while overfetching happens when there's more data provided than necessary, resulting in wasted bandwidth and resources. GraphQL mitigates both problems by enabling clients to request precisely the data they require in a single query.  Top of Form |

|  |
| --- |
| 1. **Queries**:    * Queries are used to fetch data from the GraphQL server.    * They are analogous to GET requests in RESTful APIs.    * Queries are read-only operations and do not modify any data on the server.    * They are defined in the GraphQL schema using the **type Query { ... }** syntax.    * Queries are executed using the **query** keyword in GraphQL requests. 2. **Mutations**:    * Mutations are used to modify or update data on the GraphQL server.    * They are analogous to POST, PUT, PATCH, or DELETE requests in RESTful APIs.    * Mutations can create, update, or delete data, and they may have side effects on the server.    * They are defined in the GraphQL schema using the **type Mutation { ... }** syntax.    * Mutations are executed using the **mutation** keyword in GraphQL requests. |

1. **Schema**: The schema defines the types available in the GraphQL API, including queries, mutations, and other types like objects, enums, interfaces, unions, and input types.
2. **Types**: GraphQL types define the shape of the data that can be queried or mutated in the API. They include scalar types (e.g., Int, String, Boolean), object types (representing complex data structures), enum types (enumerated values), interface types (abstract types that other types can implement), union types (representing multiple types), and input types (used as arguments in mutations).
3. **Fields**: Fields are the individual pieces of data that can be requested in a query. Each field corresponds to a property on a GraphQL type.
4. **Arguments**: Arguments are parameters passed to fields or directives in a GraphQL query or mutation. They allow clients to customize the results of their requests.
5. **Directives**: Directives are used to provide instructions to the GraphQL execution engine. They can be used to conditionally include or skip fields, dynamically change the structure of the query, or modify the behavior of schema validation.
6. **Fragments**: Fragments are reusable units of query syntax that allow clients to define sets of fields and reuse them across multiple queries. They improve query readability and reduce duplication.
7. **Subscriptions**: Subscriptions allow clients to subscribe to real-time data updates from the GraphQL server. They are used for scenarios where clients need to receive updates as soon as they occur, such as chat applications or live data feeds.
8. **Resolvers**: Resolvers are functions responsible for fetching the data associated with a particular field in the schema. They are defined for each field in the GraphQL schema and determine how data is retrieved or computed.

|  |  |
| --- | --- |
| **Benefit of GraphQL over Rest (solving overfetching) comments not needed but coming, to solve it we can do as below:**  In your example, if a client only needs the details of a post without its comments, they can specify their query to only request the necessary fields for the post. This reduces the size of the response payload compared to fetching the entire post object, including its comments, in a traditional REST API.  By minimizing the amount of data transferred over the network, GraphQL can improve performance and reduce bandwidth usage, especially for clients with limited resources or slow connections.  **Query:**   |  | | --- | | query GetPostDetails($postId: ID!) {  post(id: $postId) {  id  title  content  author {  id  name  }  createdAt  updatedAt  }  } |   This query will fetch the details of a single post identified by its ID, along with the name of the author, without fetching any unnecessary data like comments or additional nested objects. |

**Solving Under-Fetching: Data is not complete, so fetch from multiple tables in multiple request and round trips which cause performance issues, to solve it do as below:**

|  |
| --- |
| User  **You**  explain underfetching with example? and how graphql solve it? and can we get response from multiple urls in single query  ChatGPT  **ChatGPT**  Underfetching occurs when a client needs to make multiple requests to the server to gather all the required data for a particular view or operation. This happens when the initial response from the server does not contain all the necessary information, forcing the client to make additional requests to fetch related data.  Let's illustrate underfetching with an example:  Suppose we have a social media application with RESTful endpoints for users and their posts. When a user opens their profile page, the client needs to display the user's information along with their latest posts. With a traditional REST API, the client might need to make multiple requests to different endpoints:   1. Request to fetch user information: **/users/{userId}** 2. Request to fetch the user's latest posts: **/users/{userId}/posts** 3. Request to fetch the user's latest posts: **/users/{userId}/comments**   In this scenario, the client makes three separate requests, resulting in underfetching because the initial request for user information did not include the posts and comments, requiring an additional request.  GraphQL solves the underfetching problem by allowing clients to specify exactly what data they need in a single query. Clients can request nested or related data in the same query, and the GraphQL server responds with exactly the requested data structure.  With GraphQL, you can indeed **get responses from multiple URLs or endpoints in a single query**. Since GraphQL enables clients to define the shape of the response data, you can include fields from different types and related objects in a single query. This eliminates the need to make multiple requests to different URLs or endpoints, making data fetching more efficient and reducing network overhead.  User  **You**  multiple urls request in single query? really how that works  ChatGPT  **ChatGPT**  I apologize for the confusion. In a typical GraphQL setup, there isn't a concept of multiple URLs like in REST APIs. **GraphQL operates through a single endpoint**, often **/graphql**, which accepts queries and mutations.  However, with GraphQL, you can request data from multiple types or entities in a single query. This is achieved by specifying the fields you need from each type in the GraphQL query. The GraphQL server then resolves these fields by fetching data from various data sources (such as databases, external APIs, etc.) and constructs a response containing all the requested data.  So while **GraphQL doesn't perform requests to multiple URLs** in the traditional sense, it allows you to efficiently retrieve data from different parts of your data model in a single query, thus reducing the need for multiple round trips to the server. This helps in addressing underfetching and overfetching issues commonly associated with REST APIs.  User  **You**  okay i understand, requesting data from single end point but data is superbly structured through schema that we can get comments, posts and different tables in single query, unlike REST  ChatGPT  **ChatGPT**  Exactly! GraphQL's ability to request precisely the data you need, structured according to the schema, allows you to efficiently retrieve related data from different parts of your data model in a single query. This contrasts with REST APIs, where you often have to make multiple requests to different endpoints to gather all the necessary data. **With GraphQL, you can reduce overfetching and underfetching,** leading to more efficient data fetching and better performance in your applications.  Top of Form |

**Examples:**

|  |
| --- |
| **//used for nested relations in tables (related Data//association)**  **//we can find review associated with specific game and author associated with specific game**  **//If we need to find game associated with specific review then nested**    **Problem: Related Data and Association between table**  **Query**  **// review(id: $reviewId) {**  **// rating**  **// game {**  **// platform**  **// title**  **// }**  **// }**  **For single review**  **// review(\_, args){**  **// return db.reviews.find((review)=> (review.id===args.id))**  **// },**  **For game associated with single review**  **// Review:{ //parent represent data from reivews array**  **// game(parent){**  **// return db.games.find((r)=>r.id==parent.game\_id)**  **// },**  **// }**    **//use find function to return single element and filter to return array** |

**GraphQL query flow**

|  |
| --- |
|  |

|  |
| --- |
| User  **You**  graphql is server side or client side  ChatGPT  **ChatGPT**  GraphQL can be used both on the server side and the client side.   1. **Server Side**: GraphQL is commonly used on the server side to create a GraphQL API. In this scenario, the server exposes a GraphQL endpoint that clients can query to retrieve data. The server is responsible for resolving the queries and returning the requested data in the specified format. 2. **Client Side**: GraphQL can also be used on the client side to make queries to a GraphQL API. Client-side GraphQL allows applications to fetch data from a server using GraphQL queries. Libraries like Apollo Client and Relay are popular choices for implementing client-side GraphQL functionality in web and mobile applications.   Overall, GraphQL is a query language and runtime that can be implemented on both the server side and the client side, enabling efficient data fetching and manipulation in applications.  Top of Form |

**Directives**

|  |  |  |  |
| --- | --- | --- | --- |
| We can use directives in GraphQL to dynamically change the structure and shape of our queries based on whether the user is authenticated or not. Here's how we can construct a query for such a component:   |  | | --- | | **query UserProfile($userId: ID!, $isAuthenticated: Boolean!) {**  **user(id: $userId) {**  **name**  **email @include(if: $isAuthenticated)**  **phone @include(if: $isAuthenticated)**  **voice @skip(if: $isAuthenticated)**  **address {**  **street**  **city**  **country**  **}**  **}**  **}** | | **Explanation:**  In this example:   * We have a **user** field that retrieves information about a user based on their **userId**. * We include the **name** field in all cases because it's public information. * We use the **@include** directive to conditionally include the **email** and **phone** fields based on whether the user is authenticated (**$isAuthenticated**). * We use the **@skip** directive to conditionally exclude the ‘**voice’** field based on whether the user is authenticated (**$isAuthenticated**). * The **address** field is always included and contains public information about the user's address.   Now, in the client-side code, when the user logs in or out, we can pass the appropriate value for the **$isAuthenticated** variable to the GraphQL query. If the user is authenticated, the query will include the **email** and **phone** fields; otherwise, those fields will be omitted from the response.  This way, we can dynamically change the structure of our query based on runtime conditions without the need for manual string manipulation or separate query endpoints. Directives like **@include** allow us to write cleaner and more flexible queries in GraphQL. | | * **@include(if: Boolean)** Only include this field in the result if the argument is true. * **@skip(if: Boolean)** Skip this field if the argument is true.   Directives can be useful to get out of situations where you otherwise would need to do string manipulation to add and remove fields in your query. Server implementations may also add experimental features by defining completely new directives. | |

|  |
| --- |
|  |

**Scalar vs Object variables**

|  |
| --- |
| In GraphQL, a scalar type represents a single atomic value. Scalars are the fundamental building blocks for defining the shape and structure of GraphQL schemas. They represent simple data types like strings, numbers, booleans, and dates.  Here are some common scalar types in GraphQL:   1. **String**: Represents a sequence of characters, such as **"hello"**. 2. **Int**: Represents a signed 32-bit integer, such as **42**. 3. **Float**: Represents a signed double-precision floating-point value, such as **3.14**. 4. **Boolean**: Represents a true or false value, such as **true**. 5. **ID**: Represents a unique identifier, often used to fetch or identify objects, such as **"123"**.   Scalar types are atomic and cannot be further divided into smaller components. They are used to define the types of individual fields in GraphQL schemas and serve as the leaf nodes in the query execution process.  In GraphQL, an input object type is a complex type used to represent a structured set of input fields. Unlike scalar types, which represent single atomic values, input object types allow you to bundle multiple related input fields together into a single object.  **input ReviewInput {**  **stars: Int!**  **commentary: String**  **}**  Input object types are useful for defining clear and structured input data formats, especially when dealing with mutations that require multiple input parameters. |

|  |
| --- |
| **Multiple fields in mutations**  A mutation can contain multiple fields, just like a query. There’s one important distinction between queries and mutations, other than the name:  **While query fields are executed in parallel, mutation fields run in series, one after the other.**  This means that if we send two incrementCredits mutations in one request, the first is guaranteed to finish before the second begins, ensuring that we don’t end up with a race condition with ourselves. |

**Inline fragments**

|  |
| --- |
| Inline fragments are particularly useful when dealing with interfaces or unions, where the actual type of the object may vary. They allow you to conditionally include fields based on the concrete type of the object, without needing to duplicate the entire query for each possible type. This helps keep the query concise and maintainable.  **Dynamic Selection**: By using inline fragments, you're dynamically selecting fields based on the type of the object returned by **hero**. If **hero** returns a **Droid**, you get **primaryFunction**; if it returns a **Human**, you get **height**.  //like how can we know what to get in character=> name, age or primary function, if we mention all three, then one value will be null, like primary function inside some results will be null, since it’s not present incase of human, similarly. Age will be null incase of Droid. Just for understanding I said, the field is null, in fact expereiemntally, whole object even if it contains some fields {“id”:1, “name”:”atisam”,”age”:10} if we try to fetch **primaryFunction**; then this object will be returned as null and other objects with **primaryFunction**; returned with expected values  See example below    We tried to get field but whole object is returned null even ID was there, but whole object returned. Some times we don’t know what to do with query  **Query**  query GetCharacters {    characters {      ... on Human {        name        age      }      ... on Droid {        name        primaryFunction      }    }  }  // now we are using inline fragments, conditionally return data based upon concrete type of the object  **Db.js**  let characters =[      {        "name": "Luke Skywalker",        "age": 23      },      {        "name": "R2-D2",        "primaryFunction": "Astromech"      }    ]  export default {characters }  **Schema.js**  export const typeDefs = `#graphql  type Query {      characters: [Character!]!  }  interface Character {    name: String!  }  type Human implements Character {    name: String!    age: Int!  }  type Droid implements Character {    name: String!    primaryFunction: String!  }  `  **Index.js**  import { ApolloServer } from '@apollo/server';  import { startStandaloneServer } from '@apollo/server/standalone';  import { typeDefs } from './schema.js';  import db from './db.js';  const resolvers = {        Query: {        characters() {          return db.characters        },      },      Character: {        \_\_resolveType(obj) {          if (obj.age !== undefined) {            return 'Human';          }          if (obj.primaryFunction !== undefined) {            return 'Droid';          }          return null; // Return null if the object doesn't match any concrete type        },      }    };  const server = new ApolloServer({      typeDefs,      resolvers    });  const { url } = await startStandaloneServer(server, {      listen: { port: 4000 },    });    console.log(`🚀  Server ready at: ${url}`); |

### Meta fields

Given that there are some situations where you don’t know what type you’ll get back from the GraphQL service, you need some way to determine how to handle that data on the client. GraphQL allows you to request **\_\_typename**, a meta field, at any point in a query to get the name of the object type at that point.

**Query:**

|  |
| --- |
| **search(text: "an") {**  **\_\_typename**  **... on Human {**  **name**  **}**  **... on Droid {**  **name**  **}**  **... on Starship {**  **name**  **}**  **}**  **}** |

**Output**

|  |
| --- |
| {  "data": {  "search": [  {  "\_\_typename": "Human",  "name": "Han Solo"  },  {  "\_\_typename": "Human",  "name": "Leia Organa"  },  {  "\_\_typename": "Starship",  "name": "TIE Advanced x1"  }  ]  }  } |

In the above query, search returns a union type that can be one of three options. It would be impossible to tell apart the different types from the client without the **\_\_typename** field.

**\_\_typename field**

Regarding changing the name of the meta field "\_\_typename", **it's not recommended to do so because "\_\_typename" is a *reserved meta* field in GraphQL** and is used by clients to determine the object type. Changing its name could lead to compatibility issues with GraphQL clients.

|  |
| --- |
| //db.js  **Let data= [**  **{ id: '1', \_\_typename: 'Human', name: 'Han Solo' },**  **{ id: '2', \_\_typename: 'Human', name: 'Leia Organa' },**  **{ id: '3', \_\_typename: 'Starship', name: 'TIE Advanced x1' }**  **];**  //Schema.js  type Query {  search(text: String!): [SearchResult!]!  data: [data!]!  }  interface SearchResult {  id: ID!  \_\_typename: String!  }  type Human implements SearchResult {  id: ID!  name: String!  }  type Droid implements SearchResult {  id: ID!  name: String!  }  type Starship implements SearchResult {  id: ID!  name: String!  }  type data {  id: ID!  name: String!  }  //index.js  **import db from './db.js';**  **const resolvers = {**  **Query: {**  **search: (\_, { text }) => {**  **// Perform search logic here and return an array of SearchResult objects**  **// For demonstration purposes, let's return some mock data**  **db.data**  **}**  **},**  **SearchResult: {**  **\_\_resolveType: (obj) => obj.\_\_typename**  **}**  **};** |

|  |
| --- |
|  |

**Output based questions**

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

**Union types VS Inline fragments**

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Asynchronous Resolver:**  In this example, the **getUser** resolver is asynchronous because it uses **async/await** to fetch user data asynchronously from a database.    **Trivial Resolver**  No, a trivial resolver is not necessarily asynchronous.  In GraphQL, a resolver is considered trivial if it merely returns a value without requiring any additional processing or data fetching. Trivial resolvers are typically synchronous because they directly return the resolved value without any asynchronous operations such as fetching data from a database or external API.    **Example query:**   |  | | --- | | query GamesQuery($gameId: ID!){    game(id: $gameId) {      id      title      platform      reviews {        id        content        rating        author {          name        }      }  }  } | |

**Scalar Coercion:**

|  |
| --- |
|  |

|  |
| --- |
| List resolver: A list resolver in GraphQL is a resolver function responsible for resolving a field that returns a list of values. It operates similarly to other resolver functions but specifically handles fields that represent collections or arrays of data. |

**Introspection**

|  |
| --- |
|  |

|  |
| --- |
|  |