Resolvers Everywhere

Rethinking Client and Server Boundaries in GraphQL

Janette Cheng
Software Engineer at Meta

Overview

01 **What *is* a resolver?**

02 **Relay Resolvers**
Resolvers defined on your client!

03 **To client-resolve or not to client-resolve?**
Pros and cons of client resolvers

01 What *is* a resolver?

What is a resolver? (from the spec)

6.4.2 Value Resolution

While nearly all of GraphQL execution can be described generically, ultimately the internal system exposing the GraphQL interface must provide values. This is exposed via *ResolveFieldValue*, which produces a value for a given field on a type for a real value.

As an example, this might accept the *objectType* Person, the *field* "soulMate", and the *objectValue* representing John Lennon. It would be expected to yield the value representing Yoko Ono.

ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*)

1. Let *resolver* be the internal function provided by *objectType* for determining the resolved value of a field named *fieldName*.
2. Return the result of calling *resolver*, providing *objectValue* and *argumentValues*.

What is a resolver? (from the spec)

6.4.2 Value Resolution

While nearly all of GraphQL execution can be described generically, ultimately the internal system exposing the GraphQL interface must provide values. This is exposed via *ResolveFieldValue*, which produces a value for a given field on a type for a real value.

As an example, this might accept the *objectType* Person, the *field* "soulMate", and the *objectValue* representing John Lennon. It would be expected to yield the value representing Yoko Ono.

ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*)

1. **Let *resolver* be the internal function provided by *objectType* for determining the resolved value of a field named *fieldName*.**
2. Return the result of calling *resolver*, providing *objectValue* and *argumentValues*.

What does the spec say about client-server boundaries?

What does the spec say about client-server boundaries?

Basically nothing!

What does the spec say about client-server boundaries?

Basically nothing!

We often think of the GraphQL spec as a server spec, but the word “server” only shows up 3 times in the spec.

What does the spec say about client-server boundaries?

Basically nothing!

We often think of the GraphQL spec as a server spec, but the word “server” only shows up 3 times in the spec.

The word “client” shows up 35 times…interesting!

Claim: GraphQL is agnostic when it comes to the server-client boundary

GraphQL is:

- An API for requesting exactly the data you need
- An execution model with a specific algorithm

Claim: GraphQL is agnostic when it comes to the server-client boundary

GraphQL is:

- An API for requesting exactly the data you need ← **the heart and soul of GraphQL**
- An execution model with a specific algorithm

Claim: GraphQL is agnostic when it comes to the server-client boundary

GraphQL is:

- An API for requesting exactly the data you need ← **the heart and soul of GraphQL**
- An execution model with a specific algorithm ← **a "recommendation" 😉 for how you can achieve the API (can live on the server and/or the client!)**

Why would you need client resolvers?

- Centralizing the business logic that transforms server-side data models into view models
- Client-only data (UI state; encrypted/private data; persistent data stores)
- Unified data access API through a GraphQL schema

02 Relay Resolvers

Resolvers defined on your client!

```ts
/**
* @relayType DecryptedNote
* @weak
*/
export type DecryptedNote = {
 plaintext_title: string,
 plaintext_body: string,
};
```

```ts
/**
* @relayField Viewer.decrypted_note: DecryptedNote
* @rootFragment ViewerDecryptedNoteFragment
*/
export function decryptedNote(
 key: ViewerDecryptedNoteFragment$key,
): DecryptedNote {
 const data = readFragment(
   graphql`
     fragment ViewerDecryptedNoteFragment on Viewer {
       encrypted_note_blob
     }
   `,
   key,
 );
 const decrypted = ClientEncryption.decrypt(data.encrypted_note_blob);
 return {
   plaintext_title: decrypted.title,
   plaintext_body: decrypted.body,
 };
}
```

```
/**
* @relayType DecryptedNote
* @weak
*/
export type DecryptedNote = {
 plaintext_title: string,
 plaintext_body: string,
};
```

```
/**
* @relayField Viewer.decrypted_note: DecryptedNote
* @rootFragment ViewerDecryptedNoteFragment
*/
export function decryptedNote(
 key: ViewerDecryptedNoteFragment$key,
): DecryptedNote {
 const data = readFragment(
   graphql`
     fragment ViewerDecryptedNoteFragment on Viewer {
       encrypted_note_blob
     }
   `,
   key,
 );
 const decrypted = ClientEncryption.decrypt(data.encrypted_note_blob);
 return {
   plaintext_title: decrypted.title,
   plaintext_body: decrypted.body,
 };
}
```

```
/**
* @relayType DecryptedNote
* @weak
*/
export type DecryptedNote = {
 plaintext_title: string,
 plaintext_body: string,
};
```

```
/**
* @relayField Viewer.decrypted_note: DecryptedNote
* @rootFragment ViewerDecryptedNoteFragment
*/
export function decryptedNote(
 key: ViewerDecryptedNoteFragment$key,
): DecryptedNote {
 const data = readFragment(
   graphql`
     fragment ViewerDecryptedNoteFragment on Viewer {
       encrypted_note_blob
     }
   `,
   key,
 );
 const decrypted = ClientEncryption.decrypt(data.encrypted_note_blob);
 return {
   plaintext_title: decrypted.title,
   plaintext_body: decrypted.body,
 };
}
```

```ts
/**
* @relayType DecryptedNote
* @weak
*/
export type DecryptedNote = {
 plaintext_title: string,
 plaintext_body: string,
};
```

```ts
/**
* @relayField Viewer.decrypted_note: DecryptedNote
* @rootFragment ViewerDecryptedNoteFragment
*/
export function decryptedNote(
 key: ViewerDecryptedNoteFragment$key,
): DecryptedNote {
 const data = readFragment(
   graphql`
    fragment ViewerDecryptedNoteFragment on Viewer {
      encrypted_note_blob
    }
   `,
   key,
 );
 const decrypted = ClientEncryption.decrypt(data.encrypted_note_blob);
 return {
   plaintext_title: decrypted.title,
   plaintext_body: decrypted.body,
 };
}
```

```ts
/**
* @relayField DecryptedNote.body_preview: String
* @rootFragment BodyPreviewFragment
*/
export function bodyPreview(
 key: BodyPreviewFragment$key,
): string {
 const note = readFragment(
   graphql`
     fragment BodyPreviewFragment on DecryptedNote {
       plaintext_body
     }
   `,
   key,
 );

 if (note.plaintext_body.length <= 100) {
   return note.plaintext_body;
 }
 return note.plaintext_body.slice(0, 100) + '…';
}
```

```ts
/**
* @relayField DecryptedNote.should_show_body_preview: Boolean
* @rootFragment ShouldShowBodyPreviewFragment
*/
export function shouldShowBodyPreview(
 key: ShouldShowBodyPreviewFragment$key,
): boolean {
 const note = readFragment(
   graphql`
     fragment ShouldShowBodyPreviewFragment on DecryptedNote {
       plaintext_title
       plaintext_body
     }
   `,
   key,
 );

 return note.plaintext_body.length > 20
   && note.plaintext_body !== note.plaintext_title;
}
```

```ts
/**
* @relayField DecryptedNote.body_preview: String
* @rootFragment BodyPreviewFragment
*/
export function bodyPreview(
 key: BodyPreviewFragment$key,
): string {
 const note = readFragment(
   graphql`
    fragment BodyPreviewFragment on DecryptedNote {
      plaintext_body
    }
   `,
   key,
 );

 if (note.plaintext_body.length <= 100) {
   return note.plaintext_body;
 }
 return note.plaintext_body.slice(0, 100) + '…';
}
```

```ts
/**
* @relayField DecryptedNote.should_show_body_preview: Boolean
* @rootFragment ShouldShowBodyPreviewFragment
*/
export function shouldShowBodyPreview(
 key: ShouldShowBodyPreviewFragment$key,
): boolean {
 const note = readFragment(
   graphql`
    fragment ShouldShowBodyPreviewFragment on DecryptedNote {
      plaintext_title
      plaintext_body
    }
   `,
   key,
 );

 return note.plaintext_body.length > 20
   && note.plaintext_body !== note.plaintext_title;
}
```

```
/**
* @relayField DecryptedNote.body_preview: String
* @rootFragment BodyPreviewFragment
*/
export function bodyPreview(
 key: BodyPreviewFragment$key,
): string {
 const note = readFragment(
   graphql`
     fragment BodyPreviewFragment on DecryptedNote {
       plaintext_body
     }
   `,
   key,
 );

 if (note.plaintext_body.length <= 100) {
   return note.plaintext_body;
 }
 return note.plaintext_body.slice(0, 100) + '…';
}
```

```
/**
* @relayField DecryptedNote.should_show_body_preview: Boolean
* @rootFragment ShouldShowBodyPreviewFragment
*/
export function shouldShowBodyPreview(
 key: ShouldShowBodyPreviewFragment$key,
): boolean {
 const note = readFragment(
   graphql`
     fragment ShouldShowBodyPreviewFragment on DecryptedNote {
       plaintext_title
       plaintext_body
     }
   `,
   key,
 );

 return note.plaintext_body.length > 20
   && note.plaintext_body !== note.plaintext_title;
}
```

```
function NoteRow({viewerRef}) {
 const data = useFragment(
   graphql`
    fragment NoteRow_viewer on Viewer {
      decrypted_note {
        plaintext_title
        body_preview
        should_show_body_preview
      }
    }
   `,
   viewerRef,
 );
 const note = data.decrypted_note;
 return (
   <Row>
     <Title>{note.plaintext_title}</Title>
     {note.should_show_body_preview && (
       <Subtitle>{note.body_preview}</Subtitle>
     )}
   </Row>
 );
}
```

```
function NoteRow({viewerRef}) {
 const data = useFragment(
   graphql`
     fragment NoteRow_viewer on Viewer {
       decrypted_note {
         plaintext_title
         body_preview
         should_show_body_preview
       }
     }
   `,
   viewerRef,
 );
 const note = data.decrypted_note;
 return (
   <Row>
     <Title>{note.plaintext_title}</Title>
     {note.should_show_body_preview && (
       <Subtitle>{note.body_preview}</Subtitle>
     )}
   </Row>
 );
}
```

build time

run time

run time

run time

```
function NoteRow({viewerRef}) {
 const data = useFragment(
   graphql`
     fragment NoteRow_viewer on Viewer {
       ...ViewerDecryptedNoteFragment
     }
   `,
   viewerRef,
 );

 const note = decryptedNote(data);
 return (
   <Row>
     <Title>{note.plaintext_title}</Title>
     {shouldShowBodyPreview(data) &&
<Subtitle>{bodyPreview(data)}</Subtitle>}
   </Row>
 );
}
```

```ts
/**
* @relayType NoteVault
* @weak
*/
export type NoteVault = {
 note_count: number,
 last_decrypted_at_ms: ?number,
};
```

```ts
/**
* @RelayResolver Viewer.note_vault: NoteVault
* @live
*/
export function note_vault(): LiveState<NoteVault> {
 return {
   read(): NoteVault {
     const state = VaultStore.getState();
     return {
       note_count: state.notes.length,
       last_decrypted_at_ms: state.lastDecryptedAtMs,
     };
   },

   subscribe(callback: () => void): () => void {
     // VaultStore fires listeners when notes are added/decrypted.
     return VaultStore.subscribe(callback);
   },
 };
}
```

```ts
/**
* @relayType NoteVault
* @weak
*/
export type NoteVault = {
 note_count: number,
 last_decrypted_at_ms: ?number,
};
```

```ts
/**
* @RelayResolver Viewer.note_vault: NoteVault
* @live
*/
export function note_vault(): LiveState<NoteVault> {
 return {
   read(): NoteVault {
     const state = VaultStore.getState();
     return {
       note_count: state.notes.length,
       last_decrypted_at_ms: state.lastDecryptedAtMs,
     };
   },

   subscribe(callback: () => void): () => void {
     // VaultStore fires listeners when notes are added/decrypted.
     return VaultStore.subscribe(callback);
   },
 };
}
```

```ts
/**
* @relayType NoteVault
* @weak
*/
export type NoteVault = {
 note_count: number,
 last_decrypted_at_ms: ?number,
};
```

```ts
/**
* @RelayResolver Viewer.note_vault: NoteVault
* @live
*/
export function note_vault(): LiveState<NoteVault> {
 return {
   read(): NoteVault {
     const state = VaultStore.getState();
     return {
       note_count: state.notes.length,
       last_decrypted_at_ms: state.lastDecryptedAtMs,
     };
   },

   subscribe(callback: () => void): () => void {
     // VaultStore fires listeners when notes are added/decrypted.
     return VaultStore.subscribe(callback);
   },
 };
}
```

https://github.com/captbaritone/redux-to-relay-with-live-resolvers-example/tree/new-syntax

README

Example: Migrating a Redux App to Relay using Live Resolvers

Relay Resolvers are a feature in Relay which allows you to model derived state in your Relay GraphQL schema. Live Resolvers (currently undocumented) are an experimental variant of Relay Resolvers which allow you to expose non-Relay dynamic client state in your Relay GraphQL schema.

While useful on thier own, they can be combined to provide an incremental migration path from a legacy client data provider -- like Redux -- onto Relay.

In this repository we aim to migrate an example Redux app to Relay incrementally.

03 To client-resolve or not to client-resolve?

Pros and cons of client resolvers

What about logic that could go on either the server or the client?

Even if your logic doesn't strictly *need* to belong on the client, it can be tempting to use client resolvers BUT...

Pros

- Fast to add! No need to coordinate with the server
- Lives and thus *dies* on the client

Cons

- Must be defined once per client (bad for logic that should be shared across different clients)
- Bad for expensive computation
- Relay Resolvers are not a full-blown GraphQL executor…

Could your client become both a GraphQL executor AND federated gateway??

Perhaps! Someone should build it and let me know how it goes 👀

The End