Designing a GraphQL Schema

LEARNING VOYAGE

Designing a GraphQL schema

This lesson covers

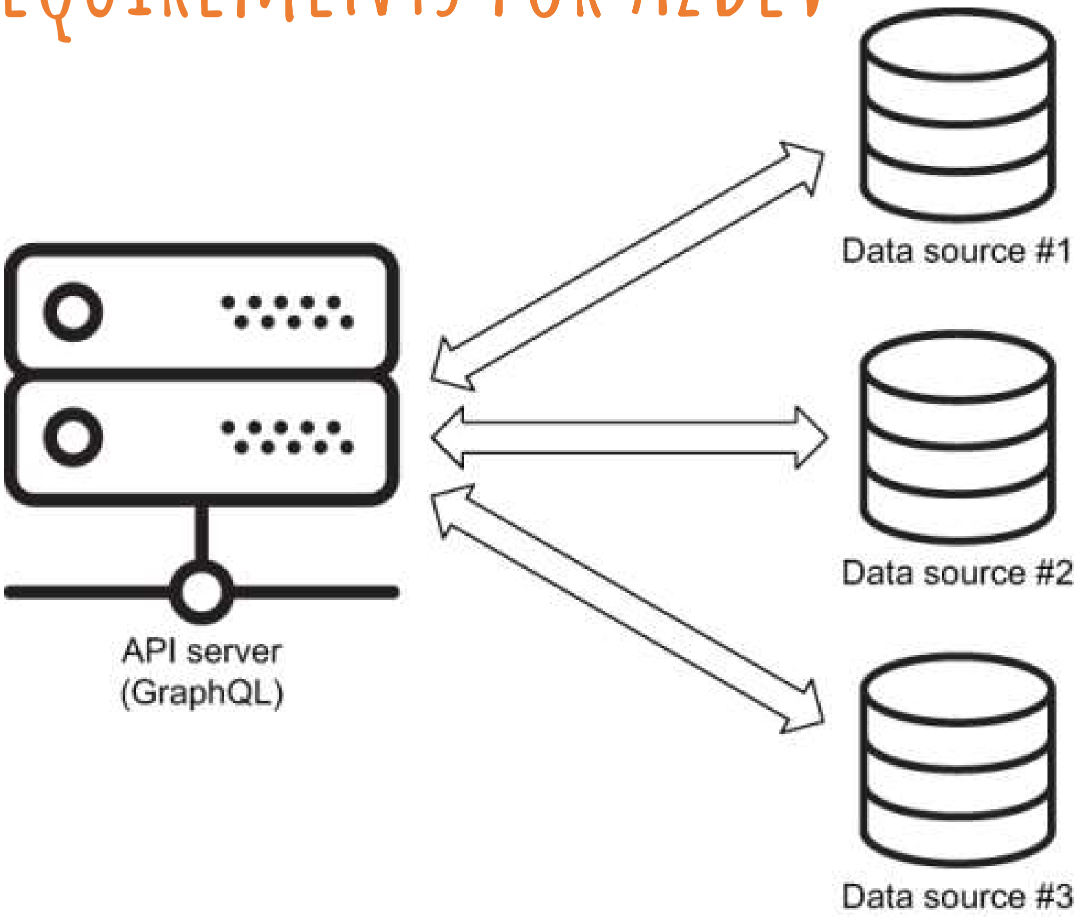
- Planning UI features and mapping them to API operations
- Coming up with schema language text based on planned operations
- Mapping API features to sources of data

Why AZdev?

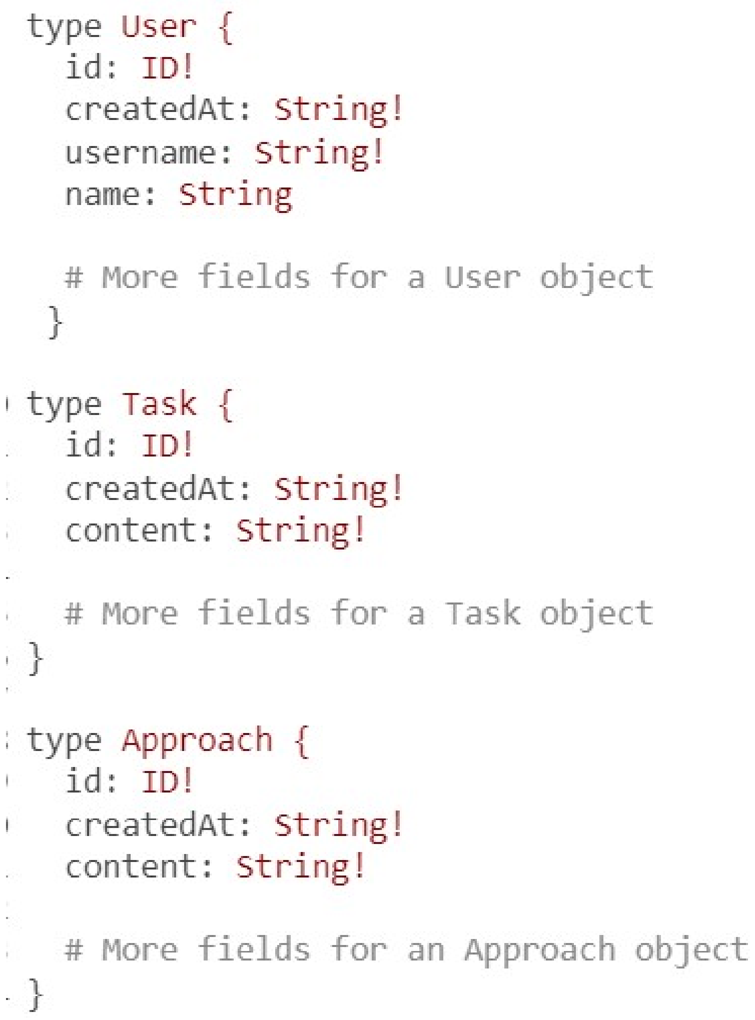
- When software developers are performing their day-to-day tasks, they often need to look up one particular thing, such as how to compute the sum of an array of numbers in JavaScript.

- AZdev is not a question-answer site. It is a library of what developers usually look up. It’s a quick way for them to find concise approaches to handle exactly what they need at the moment.

The API requirements for AZdev



The core types



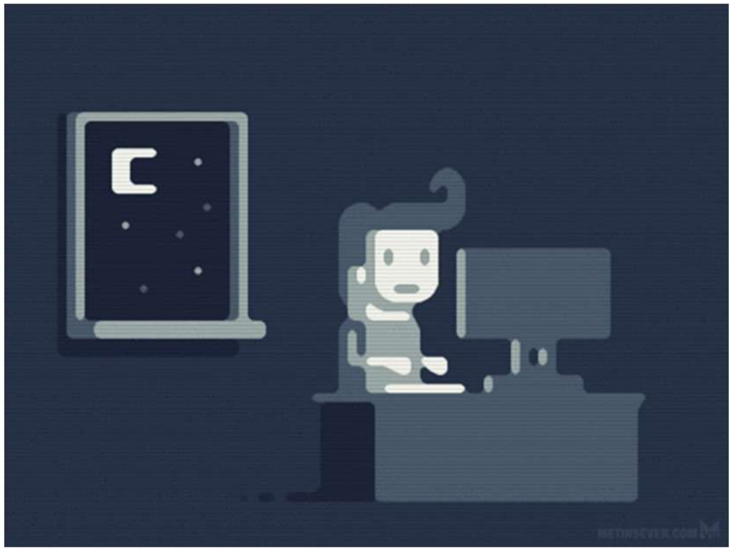
```
type User {
  id: ID!
  createdAt: String!
  username: String!
  name: String

  # More fields for a User object
 }

type Task {
  id: ID!
  createdAt: String!
  content: String!

  # More fields for a Task object
}

type Approach {
  id: ID!
  createdAt: String!
  content: String!

  # More fields for an Approach object
}
```

Queries

- Listing the latest Task records
- Search and the union/interface types
- Using an interface type
- The page for one Task record
- Entity relationships
- The ENUM type
- List of scalar values
- The page for a user’s Task records
- Authentication and authorization

Listing the latest Task records

```
query {
  taskMainList {
    id
    content

    # Fields on a Task object
  }
}
```

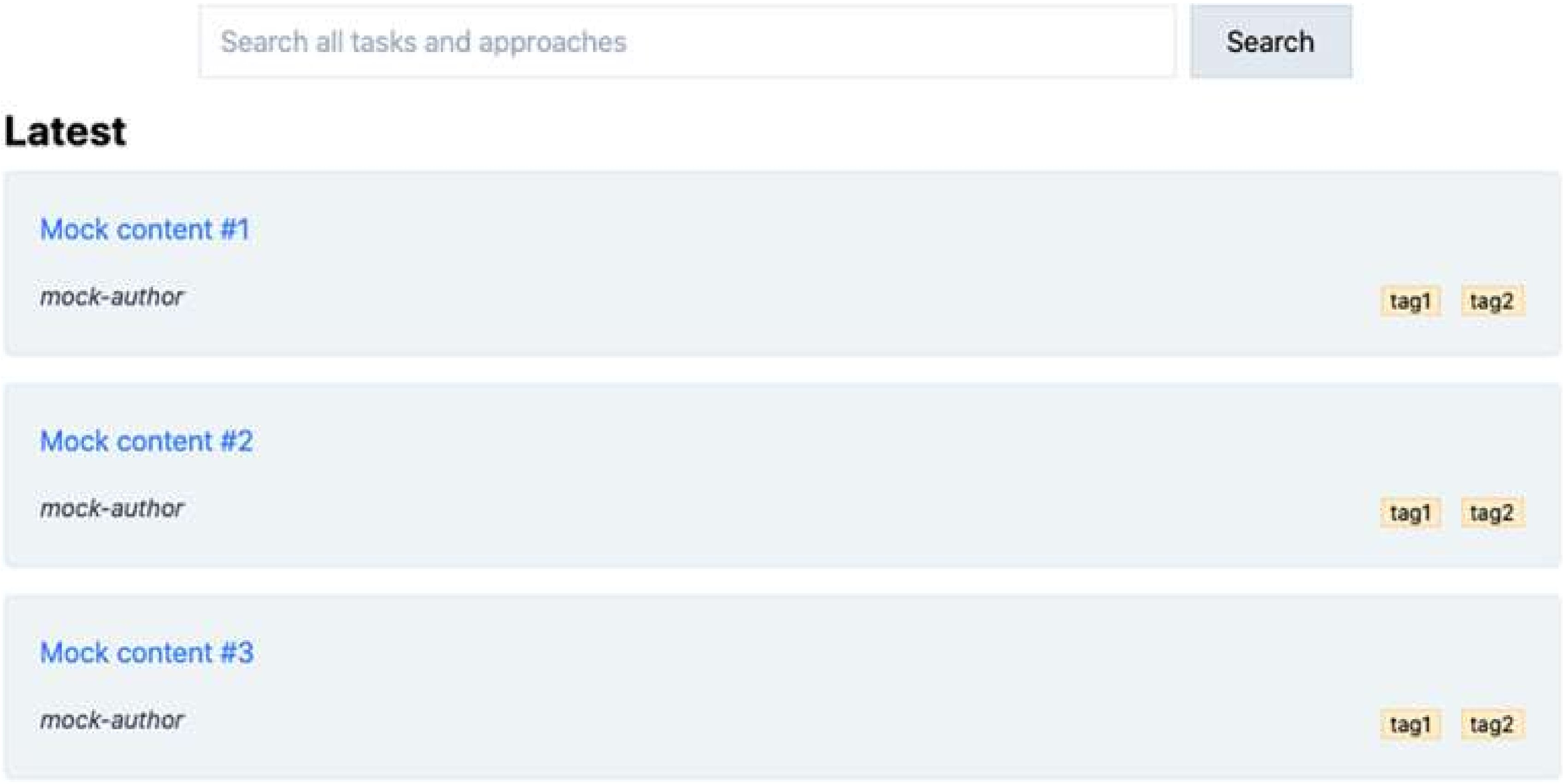


Listing the latest Task records

- To support the simple taskMainList query root field, here's a possible schema design.

```
type Query {
  taskMainList: [Task!]

  # More query root fields
}
```

Search and the union/interface types



Latest

Mock content #1

mock-author

tag1 tag2

Mock content #2

mock-author

tag1 tag2

Mock content #3

mock-author

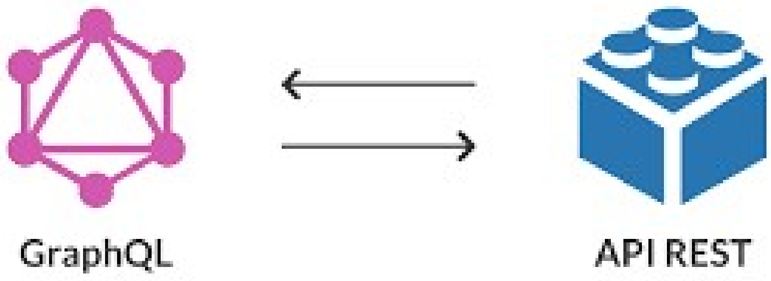
tag1 tag2

Search and the union/interface types

- To support that, we can simply add these new fields to the Task and Approach types.

```
type Task {
  # ·-·-·
  approachCount: Int!
}

type Approach {
  # ·-·-·
  task: Task!
}
```

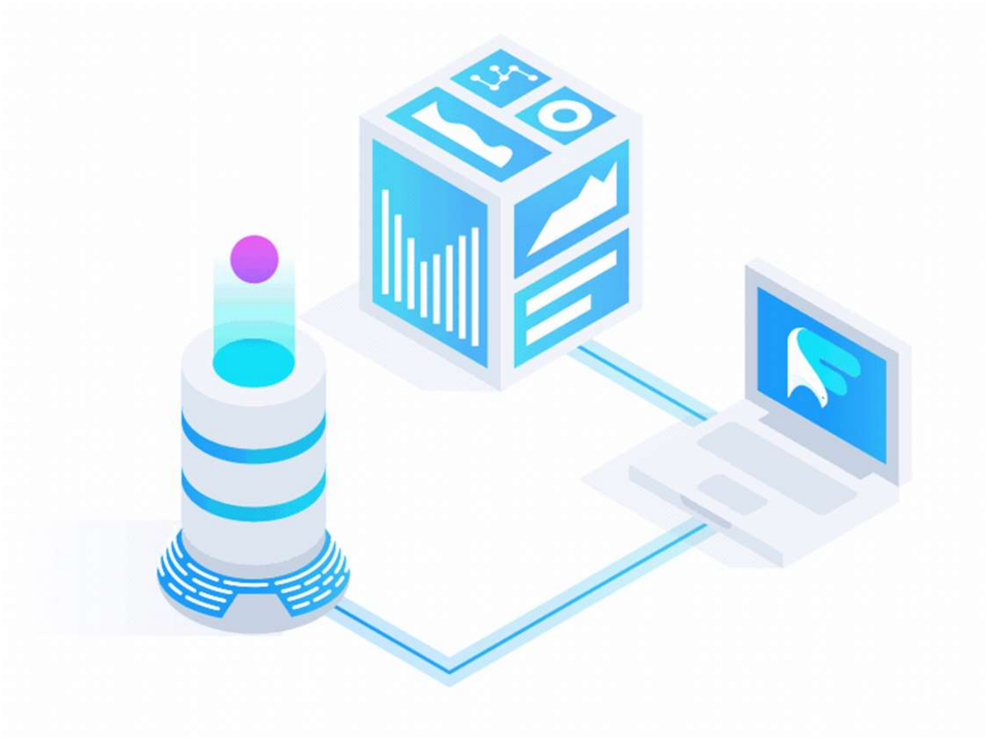
```
query {
  search(term: "something") {
    taskList {
      id
      content
      approachCount
    }
    approachList {
      id
      content
      task {
        id
        content
      }
    }
  }
}
```



Search and the union/interface types

```
search(term: "something") {
  id
  content

  approachCount // when result is a Task

  task {        // when result is an Approach
    id
    content
  }
}
```

```
query {
  search(term: "something") {
    type: __typename
    ... on Task {
      id
      content
      approachCount
    }
    ... on Approach {
      id
      content
      task {
        id
        content
      }
    }
  }
}
```



Search and the union/interface types

- In the GraphQL schema language, to implement this union type for the search root field, we use the union keyword with the pipe character (|) to form a new object type.

```
union TaskOrApproach = Task | Approach

type Query {
  # ·-·-·
  search(term: String!): [TaskOrApproach!]
}
```

Using an interface type

```
query {
  search(term: "something") {
    type: __typename
    id
    content
    ... on Task {
      approachCount
    }
    ... on Approach {
      task {
        id
        content
      }
    }
  }
}
```

Using an interface type

```
interface SearchResultItem {
  id: ID!
  content: String!
}

type Task implements SearchResultItem {
  # ·-·-·
  approachCount: Int!
}

type Approach implements SearchResultItem {
  # ·-·-·
  task: Task!
}

type Query {
  # ·-·-·
  search(term: String!): [SearchResultItem!]
}
```

The page for one Task record

- The GraphQL API must provide a query root field to enable consumers to get data about one Task object.
- Let’s name this root field taskInfo.

```
query {
  taskInfo (
    # Arguments to identify a Task record
  ) {
    # Fields under a Task record
  }
}
```

The page for one Task record

- To identify a single Task record, we can make this field accept an id argument.

- Here is what we need to add in the schema text to support this new root field.

```
type Query {
  # ·-·-·
  taskInfo(id: ID!): Task
}
```

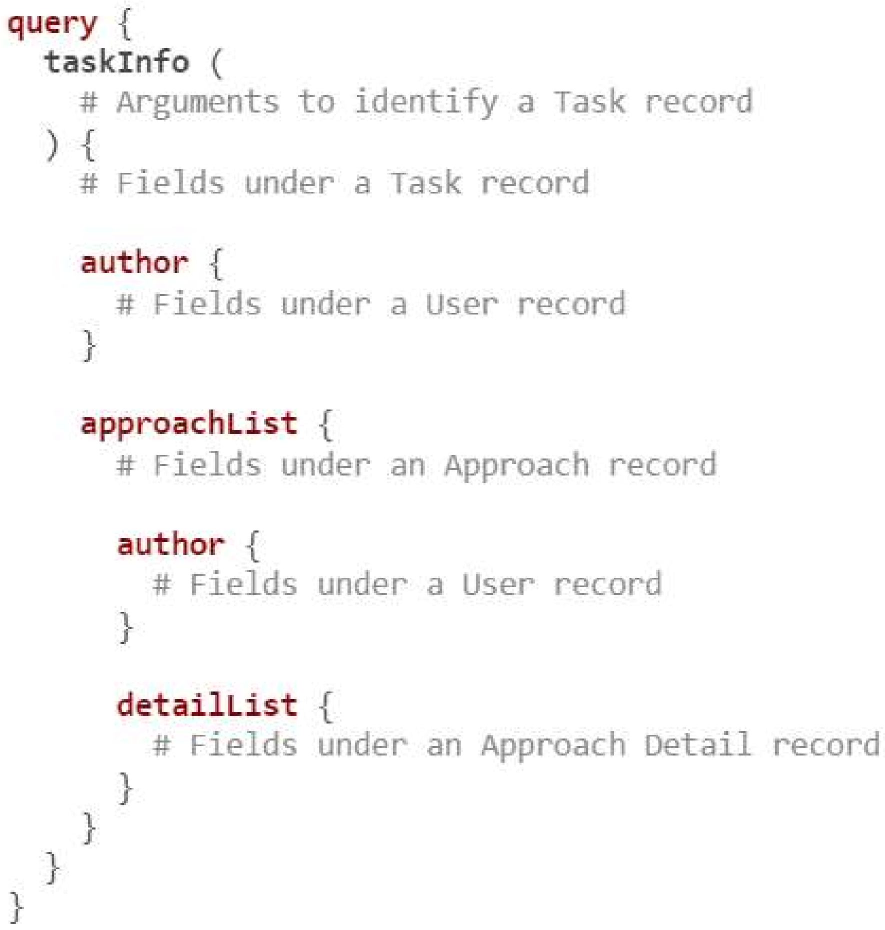
The page for one Task record

- The simplest way to account for the number of votes on Approaches is to add a field to track how many current votes each Approach object has.

- Let's do that.

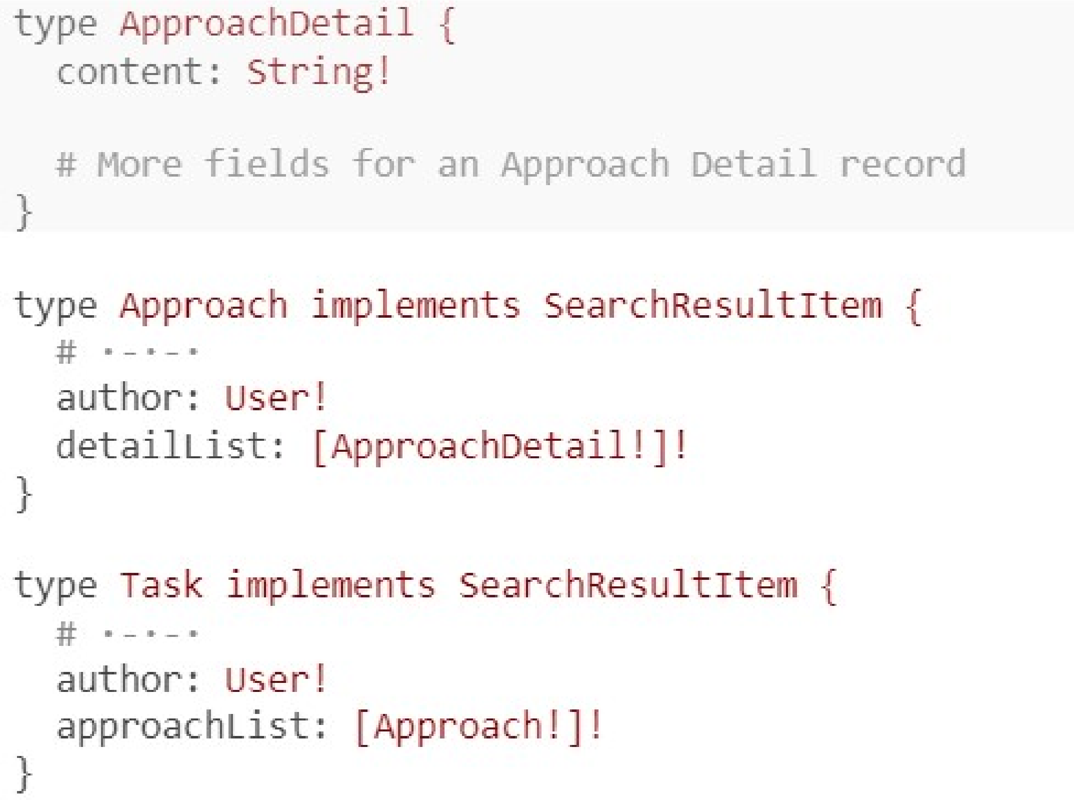
```
type Approach implements SearchResultItem {
  # ·-·-·
  voteCount: Int!
}
```

Entity relationships



```
query {
  taskInfo (
    # Arguments to identify a Task record
  ) {
    # Fields under a Task record

    author {
      # Fields under a User record
    }

    approachList {
      # Fields under an Approach record

      author {
        # Fields under a User record
      }

      detailList {
        # Fields under an Approach Detail record
      }
    }
  }
}
```

Entity relationships



```
type ApproachDetail {
  content: String!

  # More fields for an Approach Detail record
}

type Approach implements SearchResultItem {
  # ·-·-·
  author: User!
  detailList: [ApproachDetail!]!
}

type Task implements SearchResultItem {
  # ·-·-·
  author: User!
  approachList: [Approach!]!
}
```

The ENUM type

- we can use GraphQL’s special ENUM type to represent them. Here is how to do that (in SDL).

```
enum ApproachDetailCategory {
  NOTE
  EXPLANATION
  WARNING
}
```

The ENUM type

- Now we can modify the ApproachDetail GraphQL type to use this new ENUM type.

```
type ApproachDetail {
  content: String!
  category: ApproachDetailCategory!
}
```

List of scalar values

- Let’s make these tags part of the data response for each field that returns Task objects.

- It can simply be an array of strings.

```
type Task implements SearchResultItem {
  # ·-·-·
  tags: [String!]!
}
```

The page for a user's Task records

```
query {
  me (
    # Arguments to validate user access
  ) {
    taskList {
      # Fields under a Task record
    }
  }
}
```

The page for a user's Task records

- To support the me { taskList } feature, we will have to introduce two fields in the schema: a root me field that returns a User type and a taskList field on the User type.

```
type User {
  # ·-·-·
  taskList: [Task!]!
}

type Query {
  # ·-·-·
  me: User
}
```

Authentication and authorization

- When an authToken is included with a request, the API server will use it to identify the user who is making that request, This token is similar in concept to a session cookie.
- It will be remembered per user session and sent with GraphQL requests made by that session.
- It should be renewed when users log in to the AZdev application.
- Authorization is the business logic that determines whether a user has permission to read a piece of data or perform an action.

Mutations

- The GraphQL API will need to provide mutations to create a user and allow them to obtain an authorization token.

```
mutation {
  userCreate (
    # Input for a new User record
  ) {
    # Fail/Success response
  }
}
```

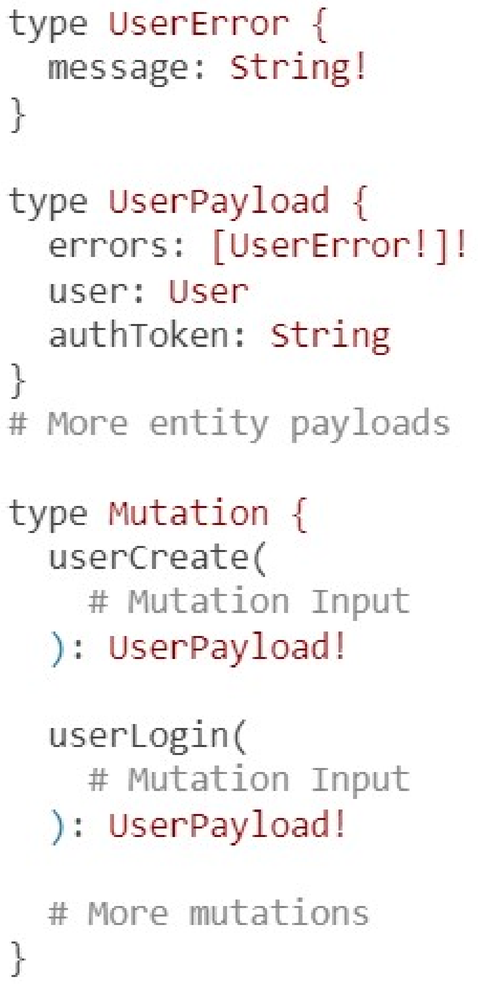
Mutations

```
mutation {
  userLogin (
    # Input to identify a User record
  ) {
    # Fail/Success response
  }
}
```

Mutations

- For example, the userLogin mutation can include the generated authToken value as part of its output payload.

- Here’s an example of how that can be done.



```
type UserError {
  message: String!
}

type UserPayload {
  errors: [UserError!]!
  user: User
  authToken: String
}
# More entity payloads

type Mutation {
  userCreate(
    # Mutation Input
  ): UserPayload!

  userLogin(
    # Mutation Input
  ): UserPayload!

  # More mutations
}
```

Mutation input

```
# Define an input type:
input UserInput {
  username: String!
  password: String!
  firstName: String
  lastName: String
}

# Then use it as the only argument to the mutation:
type Mutation {
  userCreate(input: UserInput!): UserPayload!

  # More mutations
}
```

Mutations

- For the userLogin mutation, we need the consumer to send over their username and password.
- Let’s create an AuthInput type for that.

```
input AuthInput {
  username: String!
  password: String!
}

type Mutation {
  # •–•–•
  userLogin(input: AuthInput!): UserPayload!
}
```

Deleting a user record

- Let's also offer AZdev API consumers a way to delete their user profile.

- We will plan for a userDelete mutation to do that.

```
mutation {
  userDelete {
    # Fail/Success payload
  }
}
```

Deleting a user record

- For a payload, we can just return the ID of the deleted user if the operation was a success.
- Here’s the SDL text that represents this plan:

```
type UserDeletePayload {
  errors: [UserError!]!
  deletedUserId: ID
}

type Mutation {
  # ·-·-·
  userDelete: UserDeletePayload!
}
```

Creating a Task object

- To create a new Task record in the AZdev application, let’s make the API support a taskCreate mutation.
- Here’s what that mutation operation will look like.

```
mutation {
  taskCreate (
    # Input for a new Task record
  ) {
    # Fail/Success Task payload
  }
}
```

```
input TaskInput {
  content: String!
  tags: [String!]!
  isPrivate: Boolean!
}

type TaskPayload {
  errors: [UserError!]!
  task: Task
}

type Mutation {
  # ·-·-·
  taskCreate(input: TaskInput!): TaskPayload!
}
```

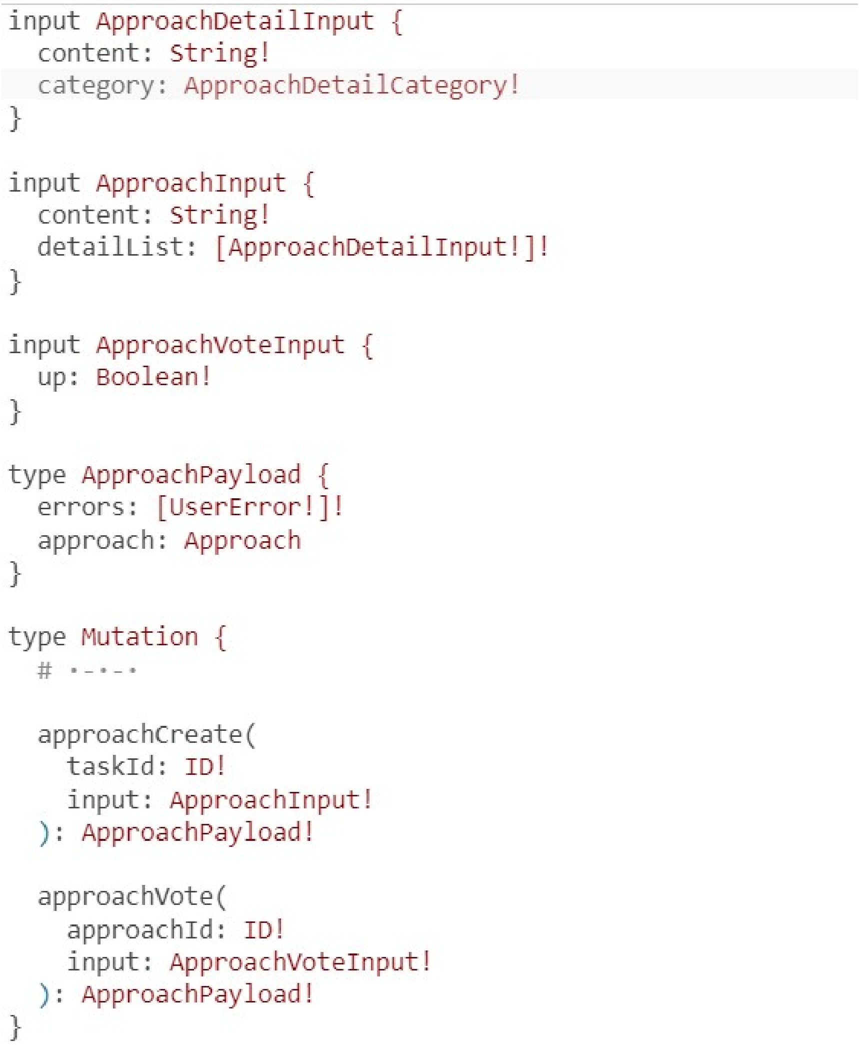
Creating and voting on Approach entries

- To create a new Approach record on an existing Task record, let's make the API support an approachCreate mutation.

```
mutation {
  approachCreate (
    # Input to identify a Task record
    # Input for a new Approach record (with ApproachDetail)
  ) {
    # Fail/Success Approach payload
  }
}
```

Creating and voting on Approach entries

```
mutation  {
  approachVote (
    # Input to identify an Approach record
    # Input for "Vote"
  ) {
    # Fail/Success Approach payload
  }
}
```



```
input ApproachDetailInput {
  content: String!
  category: ApproachDetailCategory!
}

input ApproachInput {
  content: String!
  detailList: [ApproachDetailInput!]!
}

input ApproachVoteInput {
  up: Boolean!
}

type ApproachPayload {
  errors: [UserError!]!
  approach: Approach
}

type Mutation {
  # ·-·-·

  approachCreate(
    taskId: ID!
    input: ApproachInput!
  ): ApproachPayload!

  approachVote(
    approachId: ID!
    input: ApproachVoteInput!
  ): ApproachPayload!
}
```

Here are the schema text changes needed to support these two new mutations.

Creating and voting on Approach entries

- We just put the comment text on the line before the field that needs it and surround that text with triple quotes (""").

```
input ApproachVoteInput {
  """true for up-vote and false for down-vote"""
  up: Boolean!
}
```

Subscriptions

```
subscription {
  voteChanged (
    # Input to identify a Task record
  ) {
    # Fields under an Approach record
  }
}
```

Subscriptions

- Let’s name this subscription operation taskMainListChanged.

```
subscription {
  taskMainListChanged {
    # Fields under a Task record
  }
}
```

Subscriptions

- To support these subscriptions, we define a new Subscription type with the new fields under it, like this:

```
type Subscription {
  voteChanged(taskId: ID!): Approach!
  taskMainListChanged: [Task!]
}
```

Designing database models

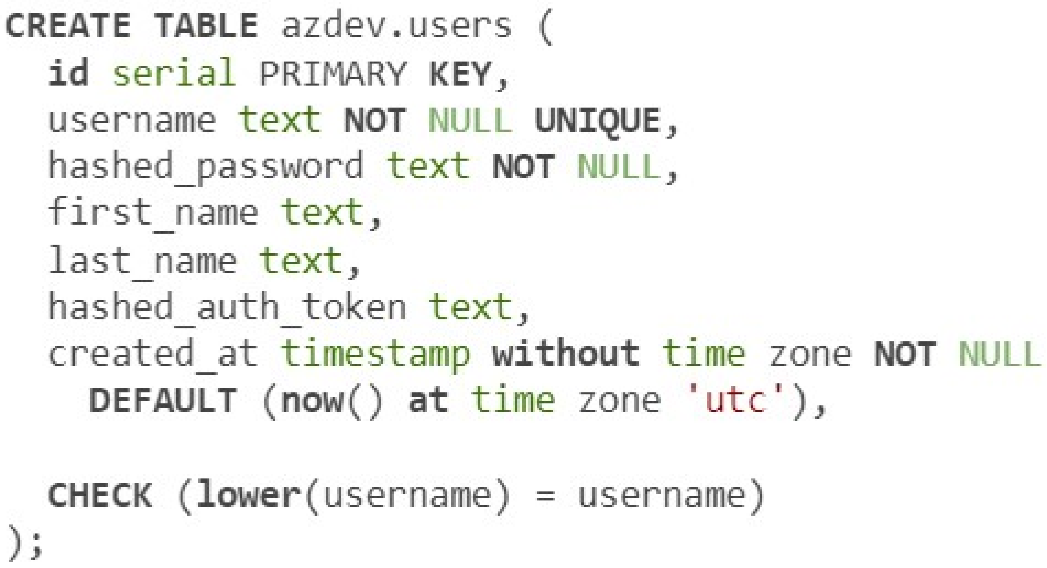
We have four database models in this project so far:

- User, Task, and Approach in PostgreSQL
- ApproachDetail in MongoDB
- To create a PostgreSQL schema, you can use this command:

```
CREATE SCHEMA azdev;
```

The User model

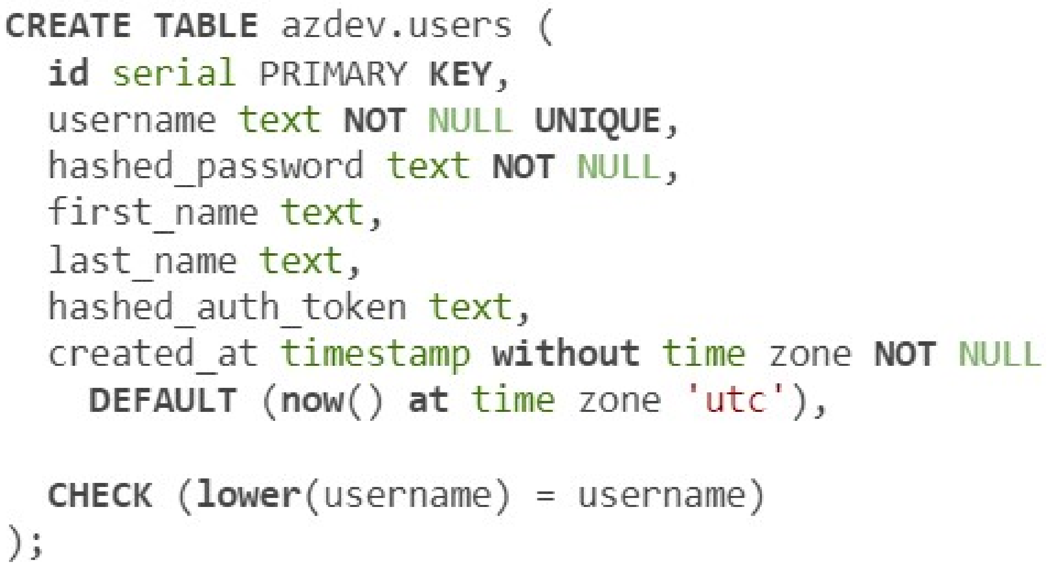
- A SQL statement to create a table for the User model.



```
CREATE TABLE azdev.users (
  id serial PRIMARY KEY,
  username text NOT NULL UNIQUE,
  hashed_password text NOT NULL,
  first_name text,
  last_name text,
  hashed_auth_token text,
  created_at timestamp without time zone NOT NULL
    DEFAULT (now() at time zone 'utc'),

  CHECK (lower(username) = username)
);
```

The Task/Approach models

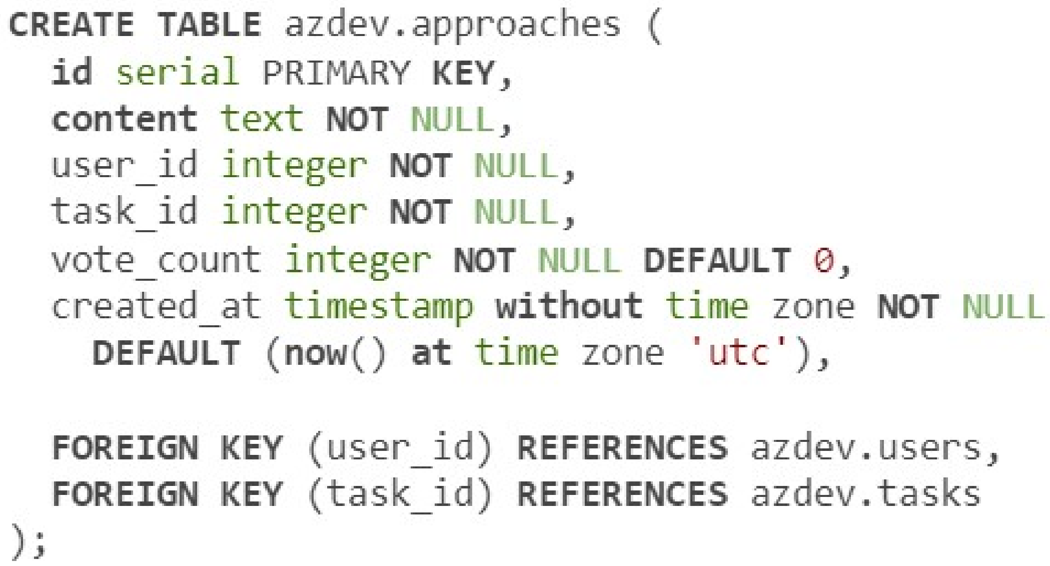
- A SQL statement to create a table for the Task model.



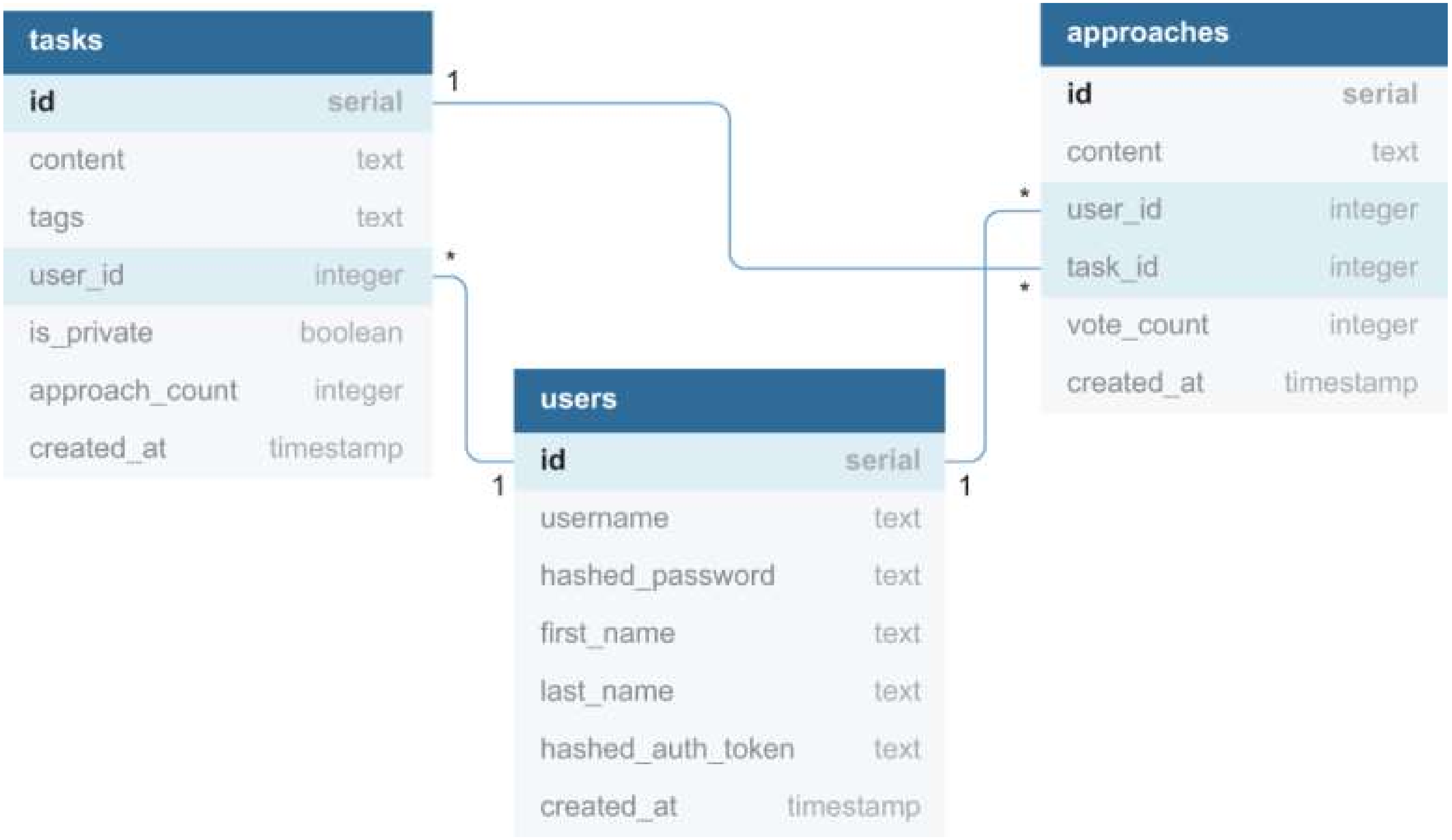
```
CREATE TABLE azdev.users (
  id serial PRIMARY KEY,
  username text NOT NULL UNIQUE,
  hashed_password text NOT NULL,
  first_name text,
  last_name text,
  hashed_auth_token text,
  created_at timestamp without time zone NOT NULL
    DEFAULT (now() at time zone 'utc'),

  CHECK (lower(username) = username)
);
```

The Task/Approach models

- A SQL statement to create a table for the Approach model.



```
CREATE TABLE azdev.approaches (
  id serial PRIMARY KEY,
  content text NOT NULL,
  user_id integer NOT NULL,
  task_id integer NOT NULL,
  vote_count integer NOT NULL DEFAULT 0,
  created_at timestamp without time zone NOT NULL
    DEFAULT (now() at time zone 'utc'),

  FOREIGN KEY (user_id) REFERENCES azdev.users,
  FOREIGN KEY (task_id) REFERENCES azdev.tasks
);
```



tasks	
id	serial
content	text
tags	text
user_id	integer
is_private	boolean
approach_count	integer
created_at	timestamp

approaches	
id	serial
content	text
user_id	integer
task_id	integer
vote_count	integer
created_at	timestamp

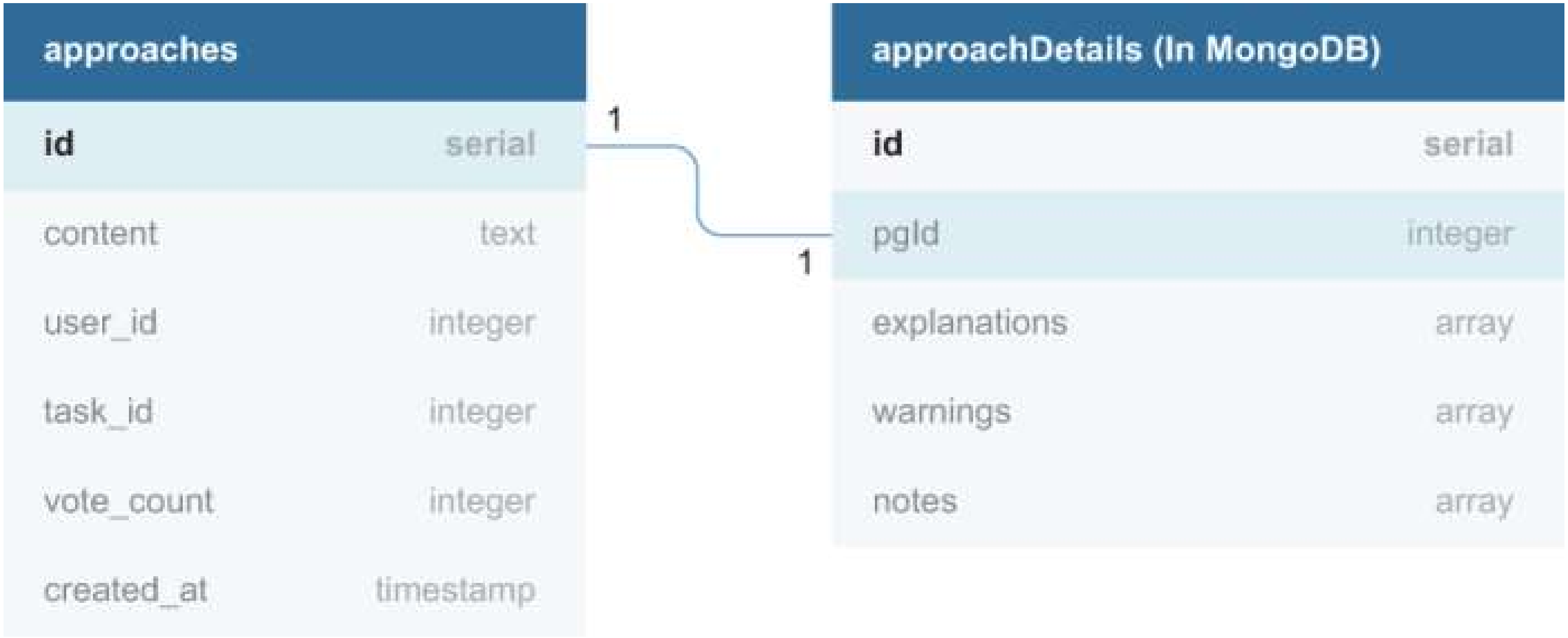
users	
id	serial
username	text
hashed_password	text
first_name	text
last_name	text
hashed_auth_token	text
created_at	timestamp

The Approach Details model

- You can run the following command to use a new database in a MongoDB client:

```
use azdev
```

The Approach Details model



approaches	
id	serial
content	text
user_id	integer
task_id	integer
vote_count	integer
created_at	timestamp

approachDetails (In MongoDB)	
id	serial
pgId	integer
explanations	array
warnings	array
notes	array

```
db.createCollection("approachDetails", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["pgId"],
      properties: {
        pgId: {
          bsonType: "int",
          description: "must be an integer and is 
required"
        },
      }
    }
  }
});
```

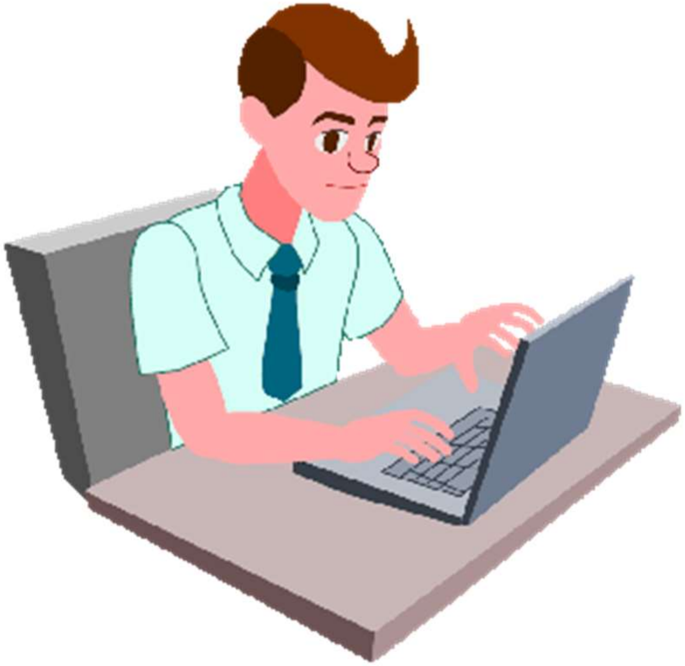
Summary

- An API server is an interface to one or many data sources.

- GraphQL is not a storage engine; it’s just a runtime that can power an API server.

- An API server can talk to many types of data services.

- Data can be queried from databases, cache services, other APIs, files, and so on.s



"Complete Lab"