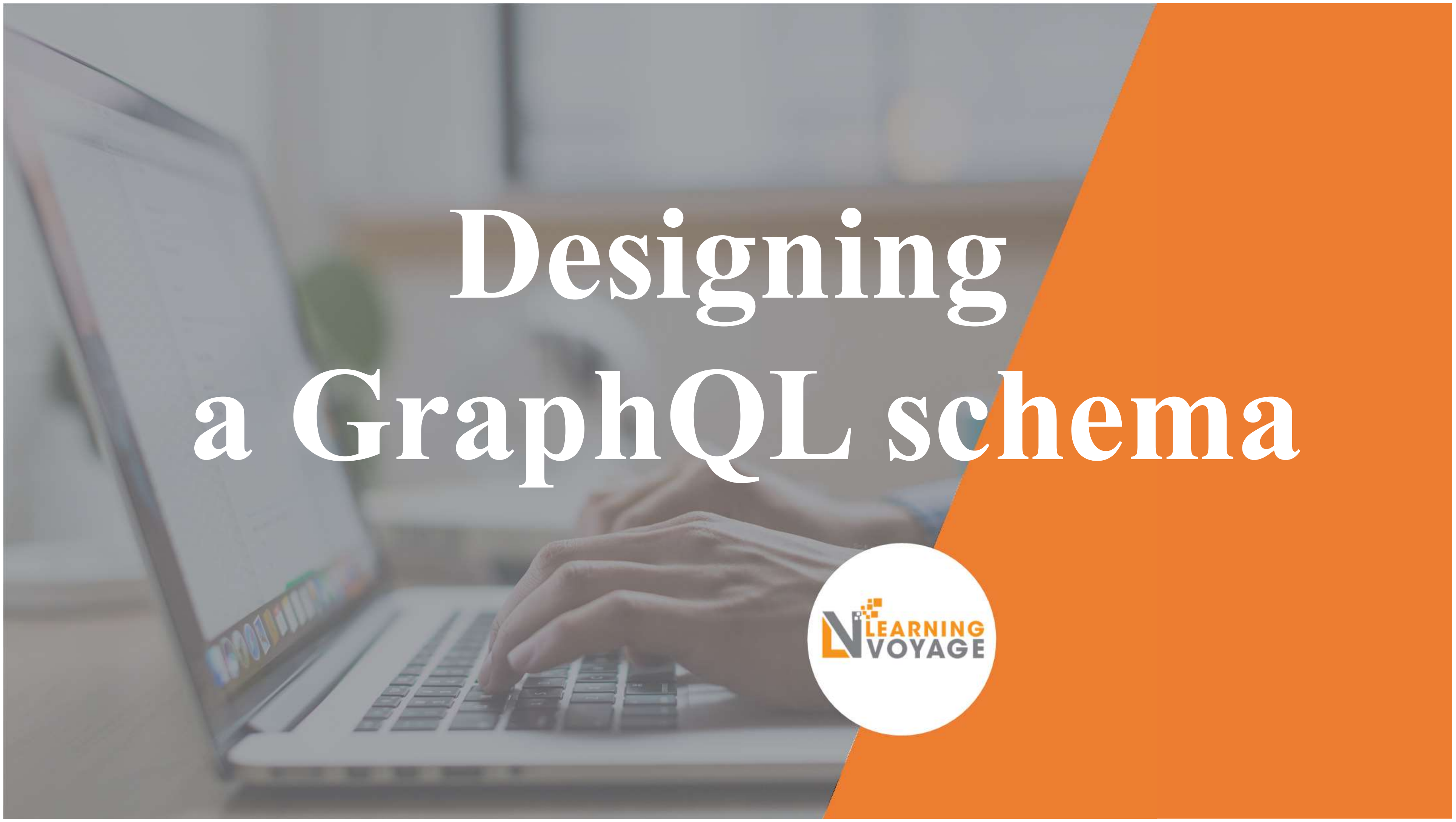


# Designing a GraphQL schema

LEARNING VOYAGE

## Designing a GraphQL schema

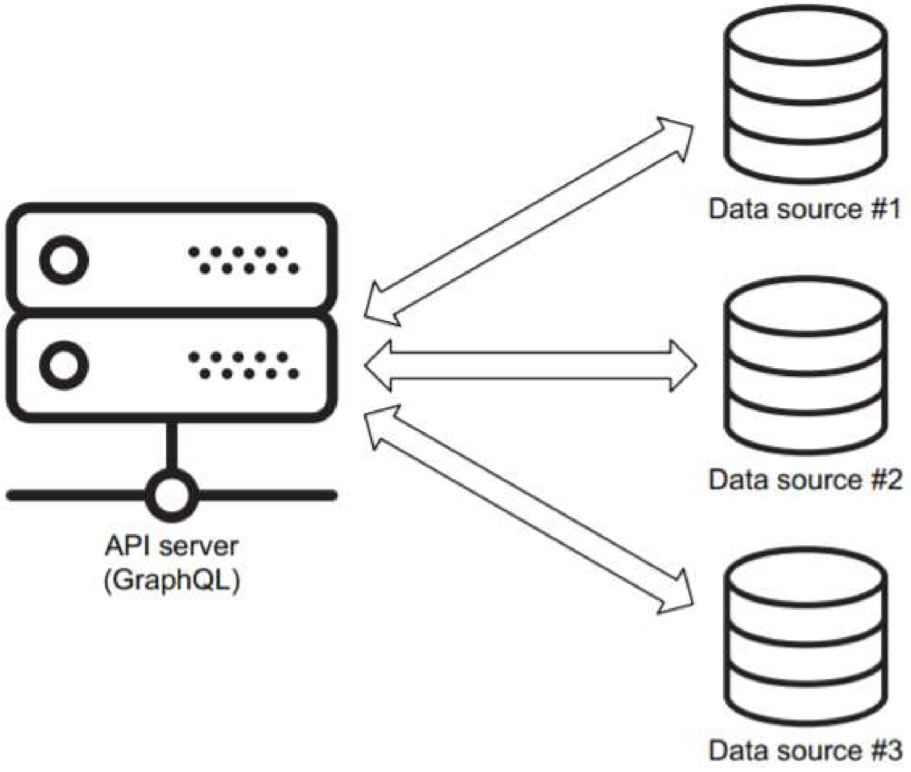
This lesson covers

- 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.
- They are not really interested in scanning pages of documentation to find the simple code example they need.

## The API requirements for AZdev



## The API requirements for AZdev

- It’s not at all unusual to have many sources of data in the same project, They don’t all have to be database services.
- A project can use a key-value cache service, get some data from other APIs, or even read data directly from files in the filesystem.
- A GraphQL schema can interface as many services as needed.

## The API requirements for AZdev

- Extra data elements on Approaches like explanations, warnings, or general notes will be stored in a document database, I picked MongoDB for that.
- A document database like MongoDB is “schemaless,” which makes it a good fit for this type of dynamic data.
- An Approach might have a warning or an explanation associated with it, and it might have other data elements that we’re not aware of at the moment.

## The core types

- The main entities in the API I’m envisioning for AZdev are User, Task, and Approach.
- These will be represented by database tables in PostgreSQL.
- Let’s make each table have a unique identity column and an automated creation date-time column.
- In a GraphQL schema, tables are usually mapped to object types, and table columns are mapped to fields under these object types.

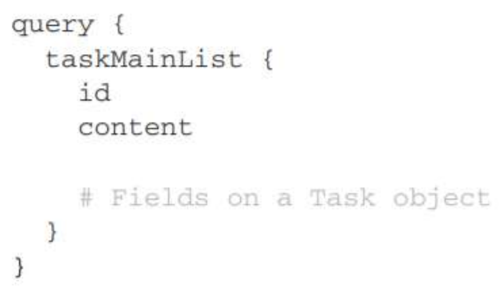
```
type User {
  id: ID!
  createdAt: String!
  username: String!
  name: String

  # 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

- I like to come up with pseudo-code-style operations (queries, mutations, and subscriptions) that are based on the envisioned UI context and then design the schema types and fields to support these operations.

## Listing the latest Task records

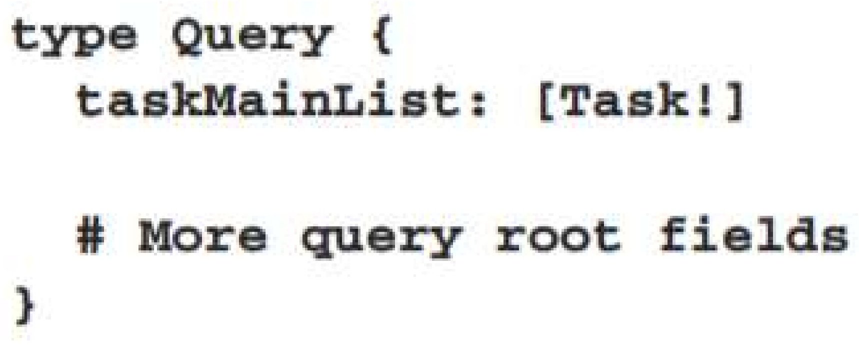
- The GraphQL API has to provide a query root field to do that.
- This list will be limited to just the last 100 records, and it will always be sorted by the creation timestamp (newer first).
- Let’s name this field taskMainList.



```
query {
  taskMainList {
    id
    content

    # Fields on a Task object
  }
}
```

## Listing the latest Task records

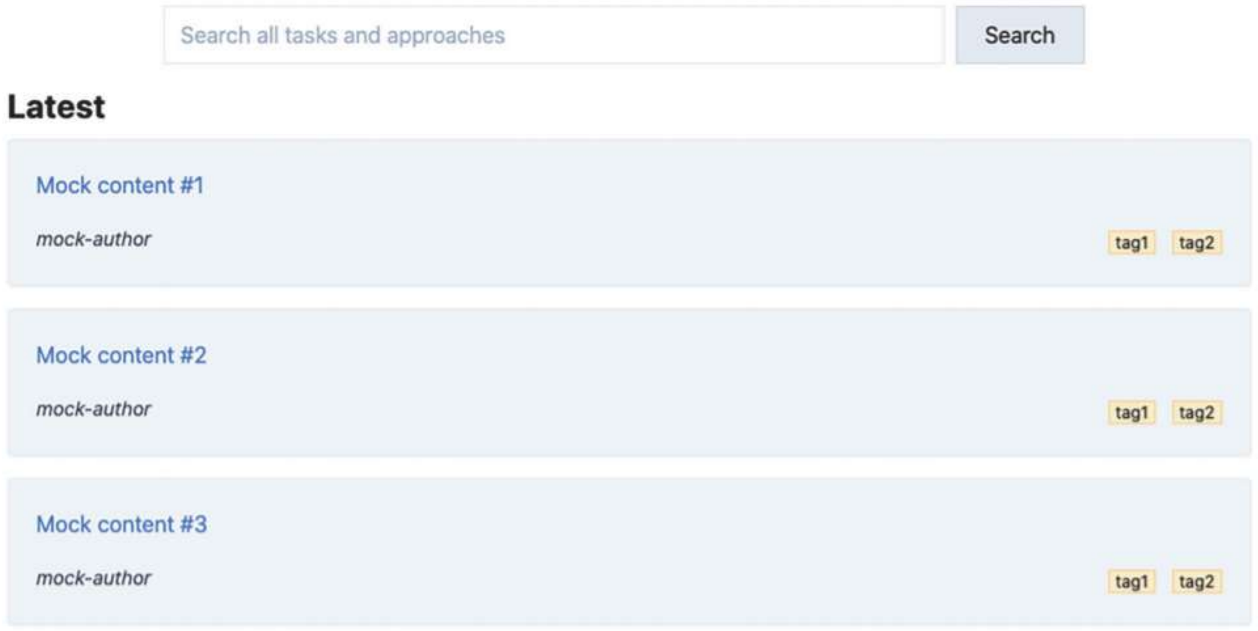
- To support the simple taskMainList query root field, here's a possible schema design.



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

  # More query root fields
}
```

## Search and the union/interface types

- The main feature of the AZdev UI is its search form.
- Users will use it to find both Task and Approach objects.



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
      }
    }
  }
}
```

## USING A UNION TYPE

- 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
      }
    }
  }
}
```

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

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

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

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

Replaces the union TaskOrApproach type

## The page for one Task record

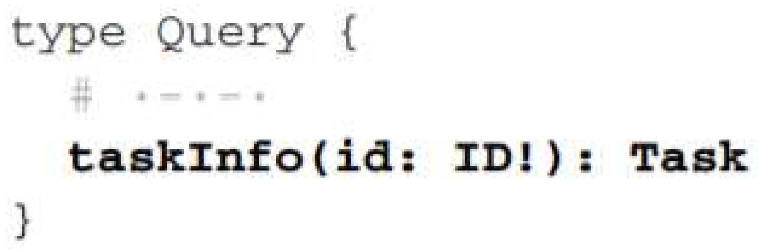
The GraphQL API must provide a query root field to enable consumers to get data about one Task object.

- 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.

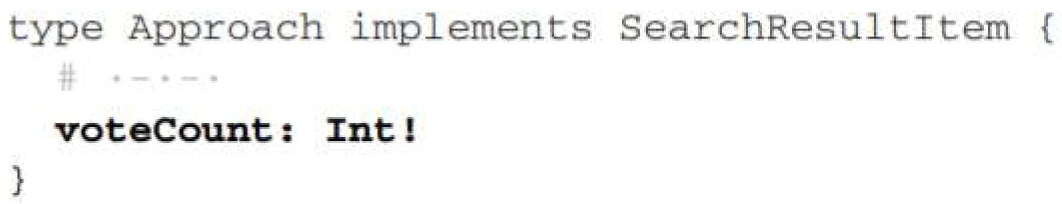


```
type Query {
  # ...
  taskInfo(id: ID!): Task
}
```

## 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

There are a few other relationships that we have to support as well:

- When displaying a Task record in the UI, we should display the name of the user who created it. The same applies to Approach objects.
- For each Approach, the application will display its list of extra detail data elements. It’s probably a good idea to have a new Approach Detail object type to represent that relation

## Entity relationships

We need to represent four relationships in this API:

- A Task has many Approaches.
- A Task belongs to a User.
- An Approach belongs to a User.
- An Approach has many Approach Detail records.

```
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
      }
    }
  }
}
```

- To support these relationships in the schema, we add references to their core types

```
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!]!
}
```

New core type to represent Approach Detail objects

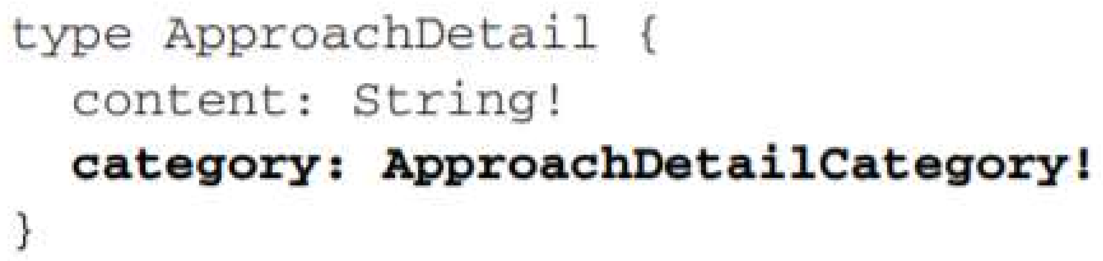
## The ENUM type

- 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.

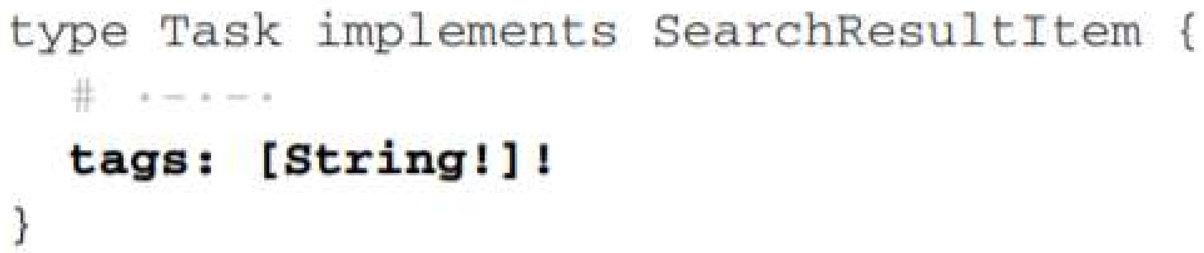


```
type ApproachDetail {
  content: String!
  category: ApproachDetailCategory!
}
```

## List of scalar values

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

It can simply be an array of strings.

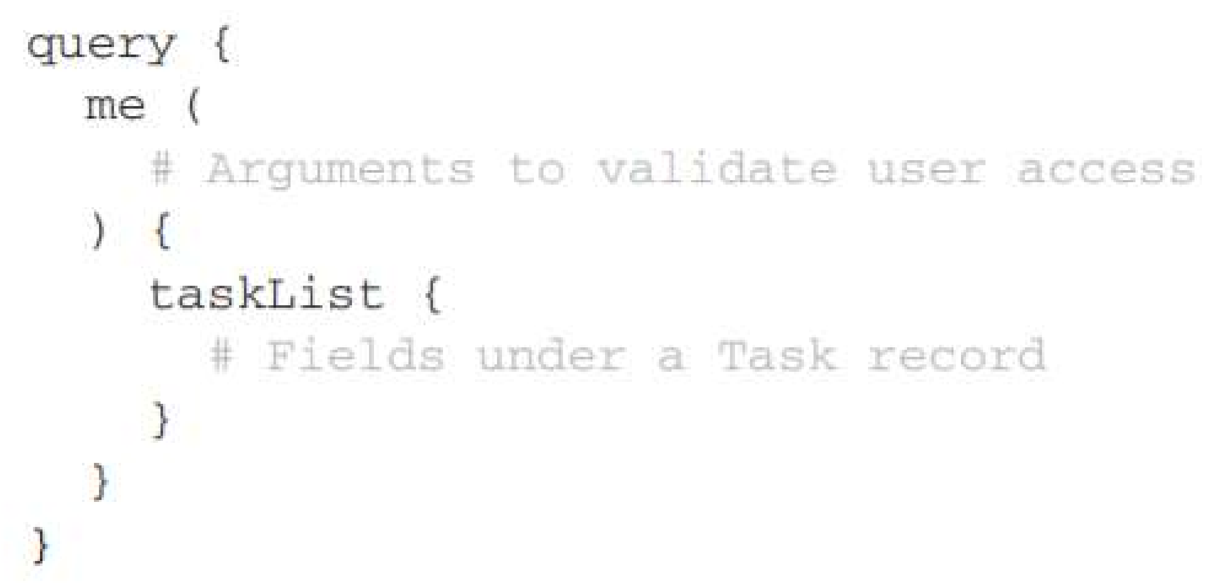


```
type Task implements SearchResultItem {
  # ·-·-·
  tags: [String!]!
}
```

## The page for a user’s Task records

- Let’s give logged-in users the ability to see the list of their Task records, We will name the field to support that taskList.
- However, making taskList a root field might be confusing.
- It could be interpreted as a field to return a list of all Task records in the database! We can name it differently to clear up that confusion, but another useful practice can also solve this issue.

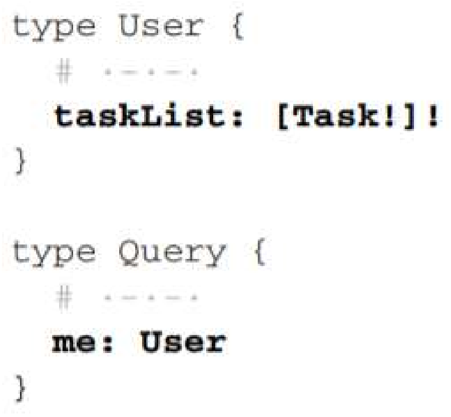
## The page for a user’s Task records



```
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

- The me field will require an access token, In this project, we’re going to use a simple string access token for authentication.

- I’ll refer to that token as authToken from now on, This string value will be stored in the database with a user record, and we can use it for personal query fields like me and search and some mutations as well

## Mutations

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

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

## Mutations

```
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

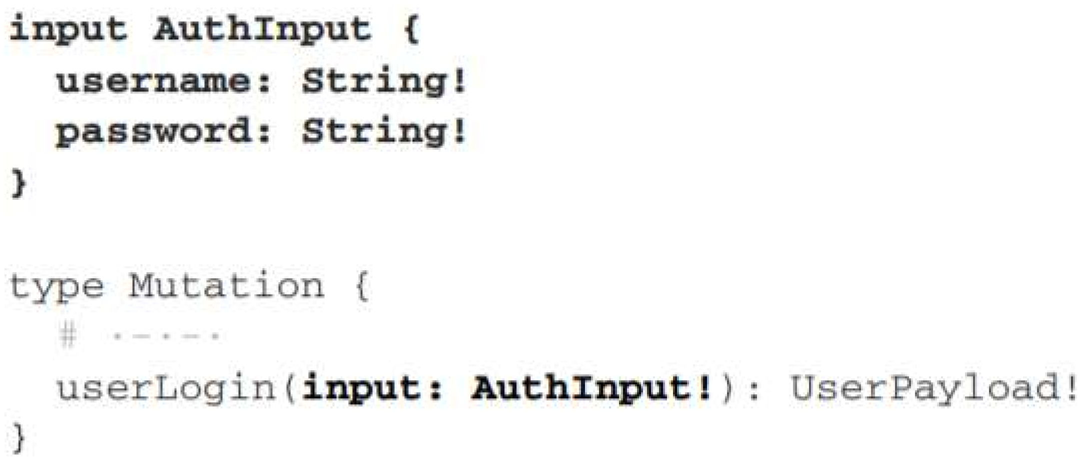
- Mutations always have some kind of input that usually has multiple elements.
- To better represent and validate the structure of a multifield input, GraphQL supports a special input type that can be used to group scalar input values into one object.
- For example, for the userCreate mutation, let’s allow the mutation consumer to specify a first name, last name, username, and password, All of these fields are strings.

## 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
}
```

## Mutation input

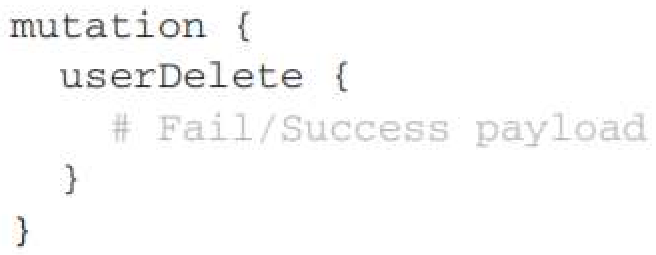
- For the userLogin mutation, we need the consumer to send over their username and password.
- Let's create an AuthInput type for that



```
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.

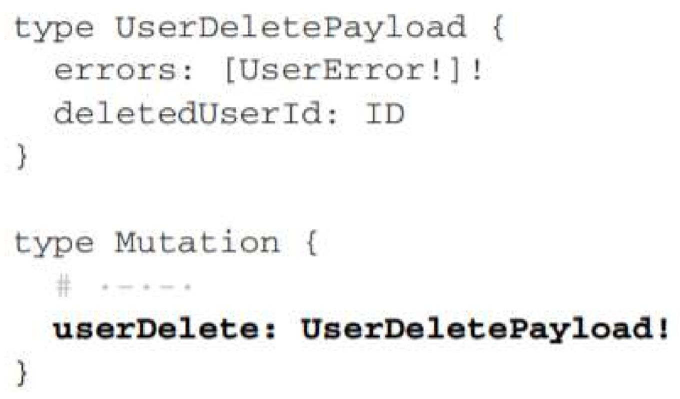


```
mutation {
  userDelete {
    # Fail/Success payload
  }
}
```

## 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

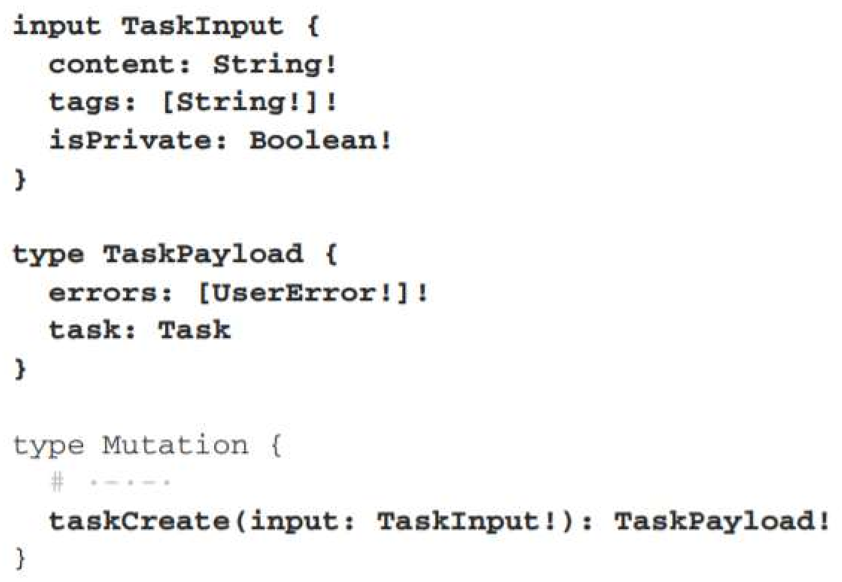
- To create a new Task record in the AZdev application, let’s make the API support a taskCreate mutation.

- Here’s what that mutation operation will look like.

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

## Creating a Task object

- Here’s the SDL text that represents what we planned for the Task entity mutations.



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

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

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

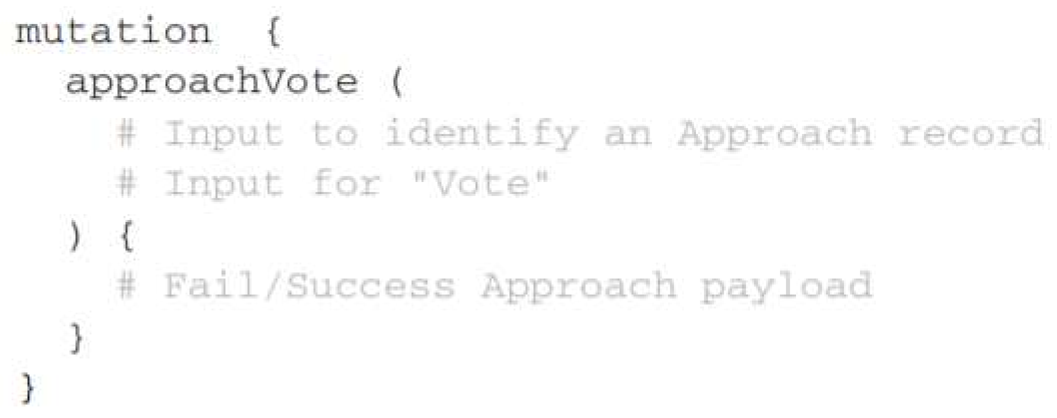
## Creating and voting on Approach entries

- 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

- We’ll make that part of the Approach payload.

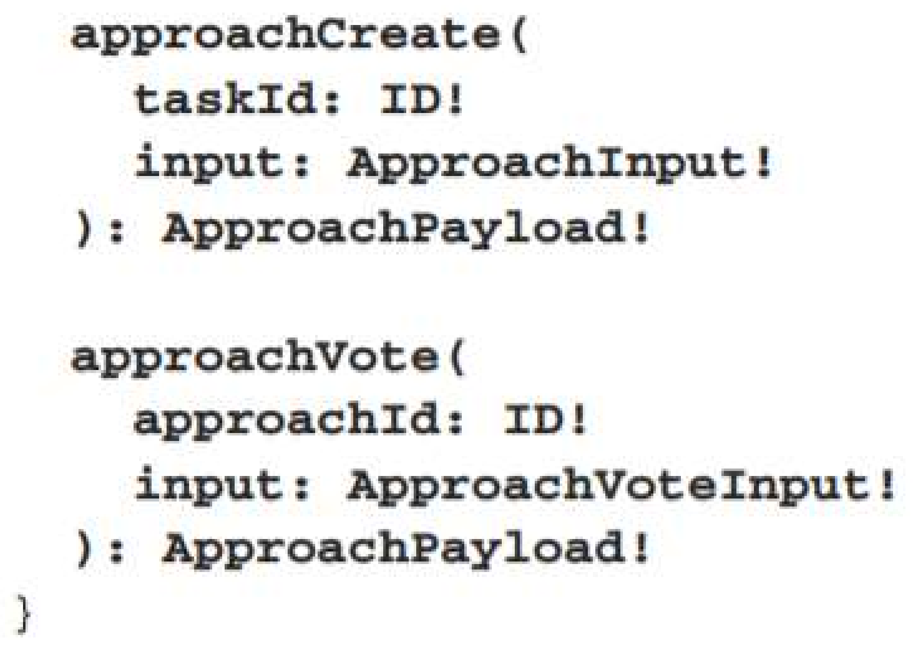


```
mutation  {
  approachVote (
    # Input to identify an Approach record
    # Input for "Vote"
  ) {
    # Fail/Success Approach payload
  }
}
```

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

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

input ApproachVoteInput {
  up: Boolean!
}

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

type Mutation {
  # ·-·-·
```

The ENUM type here will validate the accepted categories.

## Creating and voting on Approach entries



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

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

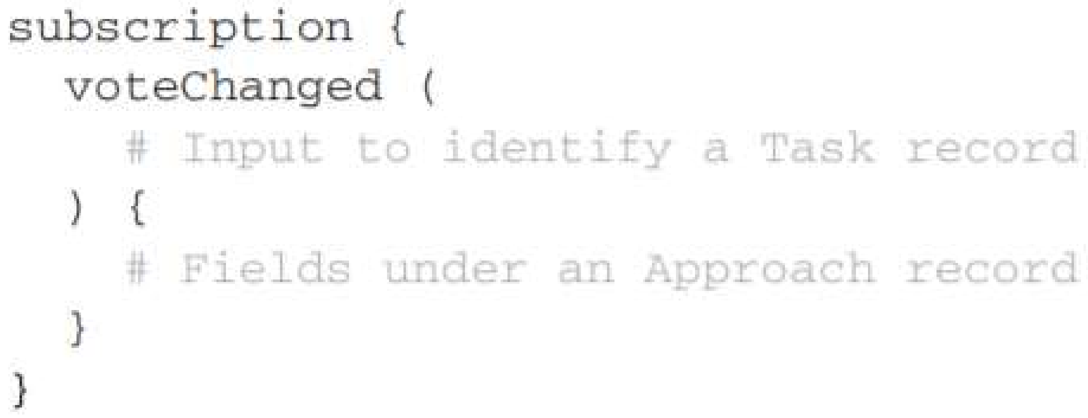
## Creating and voting on Approach entries

- 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

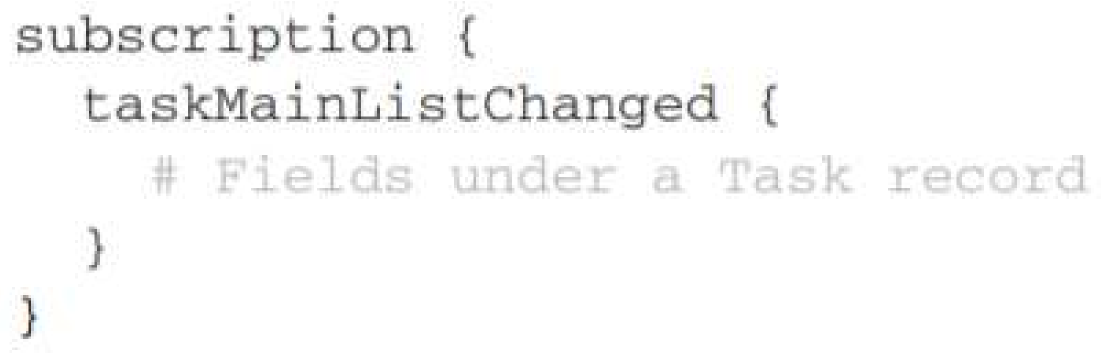
- Let’s name this subscription operation voteChanged.



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

## Subscriptions

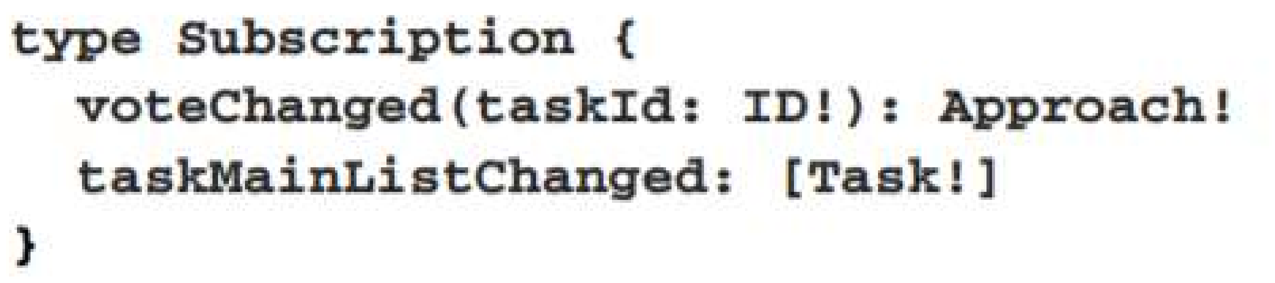
- Let’s name this subscription operation taskMainListChanged.



```
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!]
}
```

## Full schema text

- Did you notice that I came up with the entire schema description so far just by thinking in terms of the UI?
- How cool is that? You can give this simple schema language text to the frontend developers on your team, and they can start building the frontend app right away!
- They don't need to wait for your server implementation.

## Designing database models

We have four database models in this project so far:

- User, Task, and Approach in PostgreSQL
- ApproachDetail in MongoDB

## Designing database models

- To create a PostgreSQL schema, you can use this command:

```
CREATE SCHEMA azdev;
```

## The User model

- The users database table will have a record for each registered user.

- Besides the unique ID and creation-time fields we're adding under each model, a user record will have a unique username field and a hashed password field.

- These fields are required, We've designed the GraphQL User type to have a name field.

## The User model

- Here’s 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

- The tasks table will have a record for each Task object that’s submitted to the AZdev application.
- We designed a Task object to have a content text field, a list of tags, and an approachCount integer field.
- We’ll also need to add a field to support the isPrivate property, which we planned for in the mutation to create a new Task object, A Task object can have many tags.

## The Task/Approach models

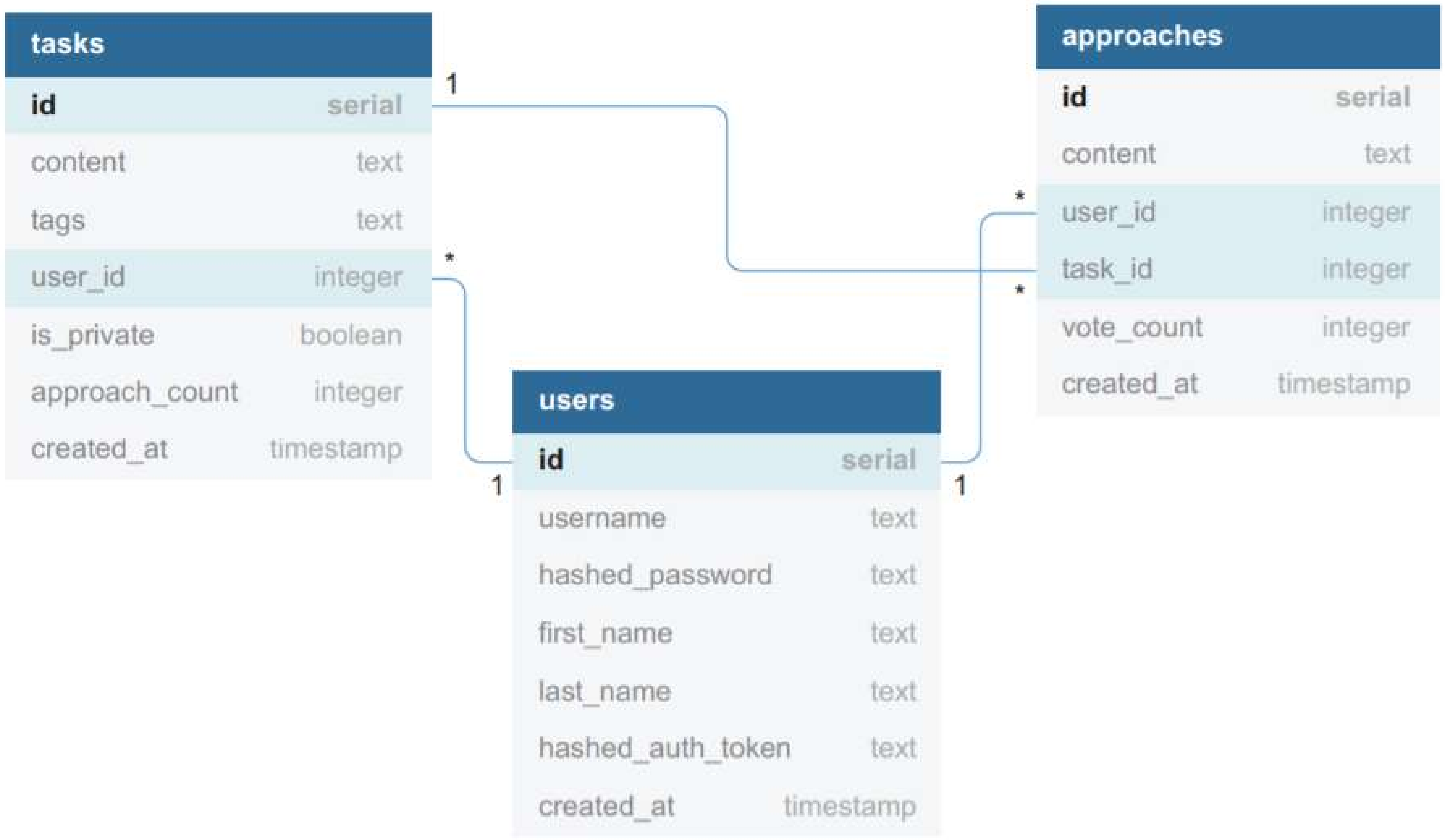
- Here's a SQL statement to create a table for the Task model

```
CREATE TABLE azdev.tasks (
  id serial PRIMARY KEY,
  content text NOT NULL,
  tags text,
  user_id integer NOT NULL,
  is_private boolean NOT NULL DEFAULT FALSE,
  approach_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
);
```

## The Task/Approach models

- Here’s 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 |

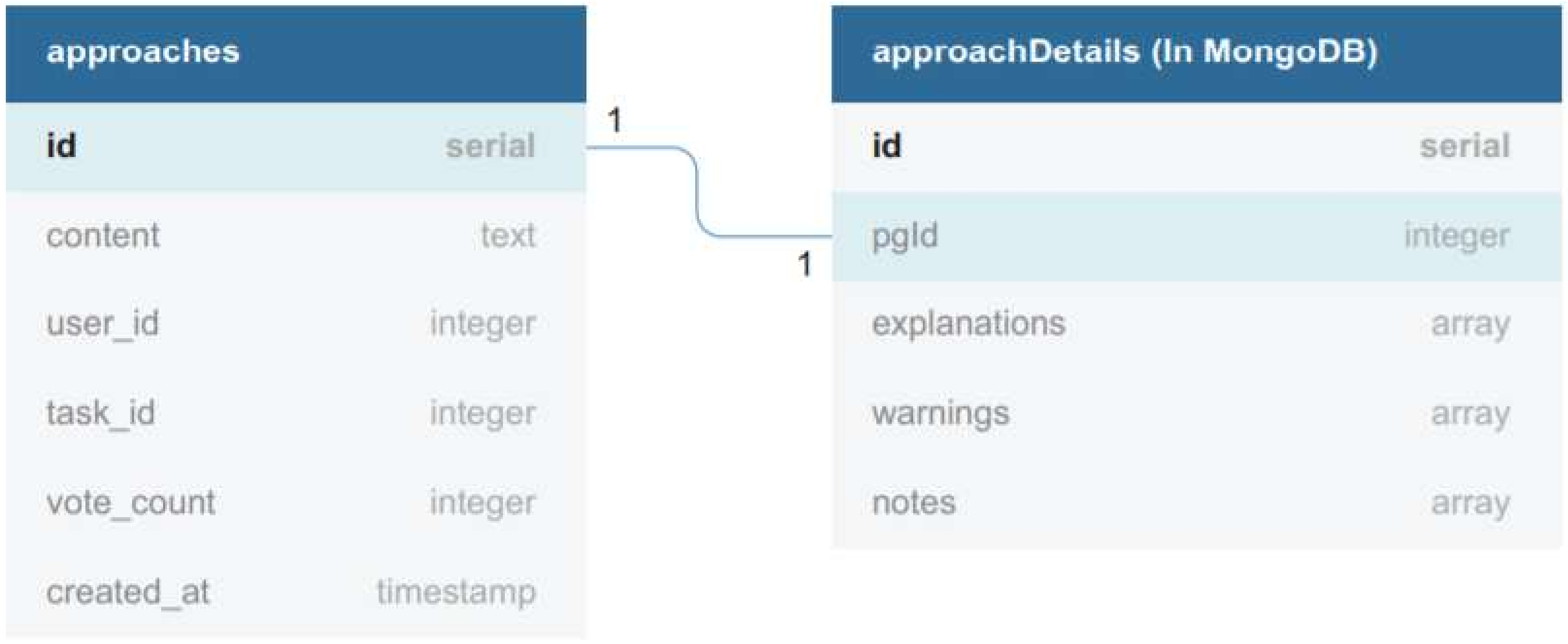
| users | |
|---|---|
| **id** | serial |
| username | text |
| hashed_password | text |
| first_name | text |
| last_name | text |
| hashed_auth_token | text |
| created_at | timestamp |

## The Approach Details model

- You actually don’t need to “create a database”;
- you just use it, and MongoDB automatically creates the currently used database the first time you insert any data into it.
- 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 |

## The Approach Details model

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

## The Approach Details model

- Each Approach record will have a single record in the approachDetails collection.
- The Approach Detail record will have fields like explanations, warnings, notes, and other categories in the future.
- Each of these fields will have an array of text items.
- We'll have to transform this special storage schema when resolving a GraphQL API request that asks for Approach Details.

## 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.
- A good first step when designing a GraphQL API is to draft a list of operations that will theoretically satisfy the needs of the application you’re designing.

## "Complete Lab"