Degree Name
Project Module Code
ID Number

Skeem - a database system to deal with modern needs

by

Henry Morgan

Supervisor: Dr. C. Hatter

Department of Computer Science
Loughborough University

April/June 2019

Abstract

Skeem is a datbase

Contents

1 Introduction

Websites are becoming more complex and offering increasingly advanced functionality whilst at the same time expected to be more performant for end users. Every non-trivial website is in some part of dynamic data, that is data which can change over the course of an applications life time. This could include anything such as blog posts, user details, editable content, user-tailored suggestions, product listings, delivery statuses. The report details the development of a system aimed to make the retrieval and manipulation of this data trivial.

The report will start by laying the foundation of how dynamic data is handled traditionally and more recently as well as some of the associated issues that arises with these approaches. Then how others have attempted to solve the issues. Next I shall discuss how I created a system aimed at solving the problem, including how the system works and how I built it. Finally I shall discuss how I tested the system as well as the opportunities I have had at deploying the system to re-world deployments.

1.1 Motivation

I have been a web developer for a number of years, over this time being involved with the creation of many web-based applications covering a wide variety of scopes. Over this time the method taken to create theses applications has changed significantly always towards the goal of producing better applications with similar techniques in shorter time-spans. I believe the system I have created furthers this goal.

1.1.1 Header

setion 1.1.1

2 Background Information

This chapter will cover how dynamic data is retrieved on a website. To understand it I will briefly cover the architecture of both traditional websites and more recent SPA based websites.

2.1 Traditional Architecture

Traditional architectures treat each page of a site as a separate resource with its own end-point, its own template and its own request. When a user navigates to a page, they will send a request to the server. The server will then look at the details of the request including what page are they requesting and who is making the request before constructing the page in full, complete with styles and content. This process can involve the server making multiple database requests in order to retrieve all the needed information. Finally the server sends the completed HTML document back to the client for it to be displayed.

This approach is very simple, the server knows exactly what data it needs to retrieve and is free to query for this data as its needed. When the user navigates to a new page, this process is simply repeated and a new page is built afresh.

This pattern does have some disadvantages, however. Many pages on a website look very similar, for instance two different articles on a news site likely both consist of the same header, footer, and sidebar. Yet when navigating to a different article the user still has to download all of these assets. This problem is magnified on devices with slow connection speeds such as mobiles.

Another issue issue is that responses is always identical independent to the device sending the request. On mobile, a site may not display a sidebar which is otherwise present on desktop. This sending of unnecessary causes slower responses which once again is most prevalent on mobile devices.

Note: This specific issue can be alleviated by the use of a mobile dedicated website, though this has its own host of issues such as having to duplicate and maintain lots of functionality which is common between platforms.

Comment: *Rigid page strucutres. This approach also does not lend itself to dynamic pages very well.*

Comment: *Large pages are impracticle for development teams, image facebooks home page, and how many developers work on that.*

2.2 SPAs

SPAs (single page applications) solve these issues by moving the page rendering to the client. When a user first goes to the site they download a single javascript bundle which contains the information to render any page of the site. The javascript then, looking at the current url, constructs the page to be rendered and displays it to the user.

When a user navigates to a different page, either by clicking a link or by pressing back in the browser, the javascript will intercept the request and simply construct the new view and render this instead.

This has some obvious benefit's. Since the javascript is fully in charge of rendering it can look at the current device as render specifically what needs to be shown - it does not need to render content which is hidden on small screens when on mobile. This method also means that when a user navigates they can get feedback instantly whether it is the new content or a loading screen, either way the site feels more responsive.

SPAs, however, provide a new challenge - how do you request the dynamic content of the page. Previously the server knew exactly what page the user requested meaning it knew exactly what data was requested. This knowledge combined with direct access to the database resulted in an easy method to retrieve the data. This problem is solved with the introduction of a new sub-system in the websites architecture: the API.

The API is similar to the traditional websites server, it accepts requests performs database queries and returns data. However, instead of returning HTML it returns raw data usually in the form of XML or JSON.

3 Problem Definition

This chapter will provide an overview of the problems associated with building web applications. It will offer a high level overview of existing practices, how they currently function, why they exist and tje problems that exist which require solving.

3.1 API Design

APIs tend to not be broken up into one end-point per page, but rather one end-point per resource. In other words, an API would tend to have an end-point for retrieving blogs and another for retrieving comments as opposed to a single one for handling the "blogs page". This is done to facilitate the notion of not requesting data that is not needed. If for some reason, the javascript wants to render the article without its comments then it shouldn't be forced to receive them anyway. Good API design is a difficult thing and there are a number of issues that present themselves when creating one.

3.1.1 Code Repetition

Consistent and discoverable APIs tend to lead to very repetitive code. If, for example, you need an end point to fetch a list of blogs and you also need one to fetch all the products. What is really different about these routes? The table name in the SQL query and the columns it returns. This duplication of code leads to highly duplicated code, something which is almost always undesirable (**???**)(http://www.informit.com/articles/article.aspx?p=457502&seqNum=5).

3.1.2 Data Duplication

There is also a trade off to be made between receiving unnecessary data and have many very similar endpoints. For instance, imagine having an end point requesting a list of blog posts. On the site you wist to display the title of each blog along with a some preview text and the authors name. On the same website you also have a page for viewing an individual blog. This page contains a blogs title, body, its authors name, and a posted date. Here, both

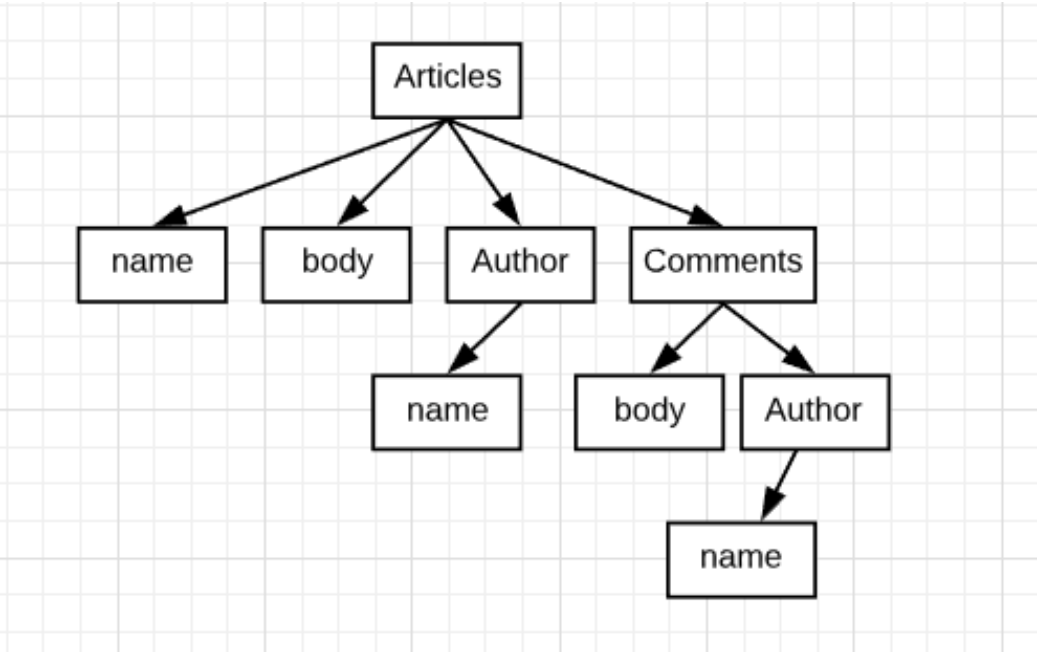
pages request very similar data, only differing in one pages need for the posted date. The options here are to either have two very similar end points which return near identical data, or forcing the end user to download the posted at information when they may never actually need to render it. The former leads to having to maintain two distinct APIs where as the latter uses unnecessary data. Neither option is desirable.

3.2 Storage vs Display

Databases should store normalizes data which, simply speaking, means storing data in flat tables i.e one for articles, one for authors, one for comments and then you storing relational information between different records. Storing data in the fashion is desirable as it greatly removes data duplication which makes updating records easier and removes the risk of desyncing data by only updating it in a single location.

The issue with this, however, is that data is not displayed like this to the end user. The end user is not presented with a page containing an article and is required to navigate to a separate page displaying the authors name then have to navigate to a third place to read a list of comments. Rather the end user will be presented with a single page containing all the data amalgamated in an easily digestible and pleasant format. The data the user sees can be envisioned as a tree of data: the root being the article itself and then containing a connected nodes for each comment each having further nodes containing their authors.

The need to request a tree of data from a database is an extremely common and useful thing, however, despite being conceptually simple it can get incredibly complex even when having to traverse only a few levels deep.




```
SELECT *
FROM "blogs"
LEFT JOIN "users" AS "authors" ON "blogs"."author_id" = "authors"."id"
LEFT JOIN "comments" ON "comments"."blogId" = "blogs"."id"
LEFT JOIN "users" AS "commentAuthors" ON "comment"."authorId" = "commentAuthors"."id"
GROUP BY "blogs"."id"
```

3.3 API and View Separation

The SPA architecture has the harsh divide between the view code and the API code. I.e code which is responsible for making the requests is very separate from that rendering the views, up to the point where they could be written in different languages. This separation presents multiple issues.

3.3.1 Tight Coupling

APIs tend to be closely related to the underlying database storing the data. If you have a properties table, then you will want a properties API to access the data. If you were to change the name of a column within the database then

you would have to remember to update the API to match and this problem grows if you have tables which span multiple APIs.

There is a similar relationship between the client and the API. When an attribute is changed on the API then everywhere using that attribute is required to update simultaneously else risk displaying incorrect responses or worse, completely crash if vital data is changed.

Tight coupling and disparate implementations leads allows for the opportunity for a de-sync which will inevitably lead to bugs.

3.3.2 Cognitive Complexity

Having the data defined in one location and used in another can make if very difficult to comprehend what is going on. It can also make it very easy to make mistakes regarding the data structures the API returns compared to how it is used. Having to understand the data flow across so many levels, through so many systems can make it very difficult to fully understand where something is coming from and why certain effects are happening.

This additional cognitive load can easily make simple tasks look complicated and paves way to slow development. It also creates a high barrier to entry, making it very hard to teach new people how the system works.

3.3.3 Bespoke Knowledge

Similar to the problem of cognitive complexity, having data flows which span many systems leads to the need to understand a lot of processes spanning lots of domains. This is especially true if the API is written in a different language to the front end.

For instance, given a PHP server, when adding a new feature it is quite likely you would have to in some way know: SQL to query the data, PHP to perform the database query and format the response, javascript to perform the api request, and html to render the view.

This need to be knowledgeable about so many domains raises the base level of skill needed to perform many tasks which makes it difficult to introduce new people to the system.

3.4 Boilerplate

Another issue with the existing method is there is a lot of boiler plate to be setup before a project can be started. You have to set up a system to manage database connections, handle database migrations, seeding data, api routing, authentication, etc. Although some of these systems may already be abstracted and so involve minimal amounts of setup and configuration it is still files which are present and visible within a project. This bloat has a few major drawbacks.

Firstly, it can be overwhelming for new entrants to the system to be faced with so many different entwined parts. Secondly, since it is code openly available to be changed within the repository there is the chance that someone unintentionally changes something and inadvertently breaks the system.

Having a large amount of boiler plate also makes it more challenging when starting a new project. This is because when starting something new you either duplicate an existing application and start stripping out unwanted features. Alternatively, you start from scratch and keep adding all the parts until everything is setup. The first can lead to unnecessary code being left in and the latter can take time to implement. Both can requires some knowledge of how the systems work and both delay the start of the task actually wanting to be achieved.

3.5 Authentication

Authentication is the process of ensuring that a request is being made by the person for whom it claims to be. It is not an issue specific to APIs and in fact is present on any application with a user system.

- There are many ways in which to authenticate, however these ways tend to be very defined. e.g not very many

Note: todo

3.6 File Management

Many websites will at some point have the ability for users to upload files in some form, be it a profile picture, a product listing or a blog hero. File uploads usally tend to have a very different request format then that of a normal request - the request usally takes the form of what is called a multipart request. A multipart request combines multiple types of data into a single requests, for instance an image and a JSON request [https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html].

Trying to perist the APIs consitency through end points which except this style of request can be difficult. There are two real solutions: forgo consistency and discoverablity and accept a bespoke end point. or allow all requests to be multipart requests even when it is purely json being sent. Multipart requests have the downside of being relateivly user hostile to view and browsers tend to be worse at providing developer tools for viewing them as they are expected to contain binary data, this makes the latter solution undesirable.

4 Literature Review

In this chapter I shall discuss existing technologies which attempt solve the problem with pre-existing technologies. I shall giving an overview as to how each system functions as well as their advantages and limitations.

4.1 Graphql

GraphQL is a system which takes a declaration of the data your application stores and produces and API to request said data. The API exposes has a single endpoint which can accept queries generated on the front end.

These queries are designed to allow nested data to be retrieved in a simple, and consistent manner. By having the front end directly send queries for the desired data, GraphQL decouples the API from the view layer. It also eliminates the API duplication problem as the front-end can simply request exactly what data is needed without having to declare a new end point.

A disadvantage of GraphQL is its need for the initial setup the data structure. This adds a lot of boiler plate the an application, with a lot of repetitive declarations. GraphQL also hides a lot of the query logic making it difficult for new developers to comprehend how requests actually work.

4.2 Mongodb

MongoDB is a NoSQL database system which uses a document based data model. Unlike traditional databases which organize data in fixed tables with association records to join related data, MongoDB stores data in arbitrary nested shapes. This means data can be stored in a way that is more closely aligned with how it will be presented to the end user.

One problem with NoSQL databases is there lack of a rigid schema. Being able to store arbitrary fields without prior declaration means you can develop the application, initially, very fast. However, it means that over time you will likely add more fields, but doing so does not update the old previously created records. This means it can be uncertain when requesting a record what attributes it has exactly. Whereas in a relational database, when you create a new column you are forced to retroactively make all other records valid within the new constraints, e.g if you add a column which cant be null, then you must set a default for all prior records.

4.2.1 Client Side Queries

4.3 Auth0

Auth0 is an external authentication system. They move the need to authenticate outside of your application and manage the logic on their own systems. They provide an API which gives access to commands to create new users, and log in to accounts. They also handle common user system tasks such as password resets, offering either email confirmation or text message confirmation.

Auth0 solves the authentication system problem by moving authentication outside of your application. Doing this can ensure that the authentication strategies are well-tested and reliable.

It does however add boiler plate to the application by requiring an interface into auth0 which is likely going to be identical between apps. It also means there is yet another bespoke system to learn.

4.4 Conclusion

Many systems exist which each solve some aspect of the defined problems and through a combination of multiple technologies many of the problems could be eliminated. However, the need to combine multiple technologies, each having their own standards and formats, only serves to exacerbate issues concerning boiler plate and training needs.

5 Requirements

This chapter will outline what a system would need to do in order to effectively solve all of the discussed issues.

5.1 High Level Requirements

These are requirements needed which directly solve the issue.

The approach demonstrated GraphQL+Relay and my mongoDB of allowing queries to be formed and sent directly from the client-side solves many of the outlined issues in an extremely effective manner. It helps to remove one side of thr tight coupling seen with a separated view and api approach. It also completely eliminates the issues associated with API design. It does come with the requirement of needing a query interface which can be properly sanitized from malicious attack.

The ability to query from the client naturally leads to the system requiring a sense of who is making the request, therefore having a built in permission system and authentication system is a logic step.

Many websites have sub-systems which have not been discussed or at least some need to perform custom actions which can only securely be executed on a server. Therefore, having the system capable of being used from both a

client and a server is a necessity as to not create a scenario what having used the system prevents needed functionality.

The handling of files is a very pre-defined task, that is to say there are an obvious, and very limited, set requirements - files must be able to be uploaded and downloaded. Building the ability to handle files into the system would help to reduce the amount of boiler plate surrounding the system and also would allow for a consistent API to be created.

Like with file management, the uses of a database in the context of web applications is fairly limited, and so unlike GraphQL I believe the system could be very opinionated as to the structure and contents of the database. This has the added benefit of reducing coupling between the API and DB structures and removes the need to define the structure in two places. Additionally this would help to limit some specific knowledge needed to manage a database.

A system containing this level of functionality and so opinionated as to be able to remove boilerplate would need documentation to educate people as to what the system does and how exactly each part works.

5.2 Specification Gathering

In order to create a solution which will alleviate these issues I had to ensure that the system achieved everything needed to replace existing systems rather then just add a further system which must be configured, maintained and learnt.

accessed an in-production data base and pulled a list of all interactions with the database

- used by 4'000 unique visitors a day
- 4% are new visitors
- 20'000 registered users

I went through all interactions with the database and records how it was being used:

- attributes
  - has many through
  - has many through where condition

    - has many with condition
    - has many dependent nullify
- Validations
    - presence
    - uniqueness
    - inclusion
    - number greater than
    - uniqueness in scope of attr: value
    - validate uniqueness in scope with condition unless attribute: value
    - Validates on: :create
    - association.attribute must = value
    - validates [if/unless] attribute: value
- Callbacks
    - before_validation
        * default attribute to another attribute if not present
        * default attributes only on create
        * default attributes to parameterized other attribute
        * default attribute to association attribute
    - before_create
        * self.slug = name.parameterize
    - after_create
        * update association
        * send emails
        * update self
- Scopes
    - where(attr: value)
    - where.not(attr: value)
    - order(attr: :desc)
    - where association count >= 1
    - where association count === 0
    - where association attribute
    - where in associations scope e.g where(tag_id: Tag.published)
    - composing scopes (adding limits)
- Permissions
    - through user association
    - through user association | where(attr: value)

Using this information I obtained the minimum viable feature set needed

5.3 Technical Requirements

combining these two sets of requirements produces the following set: ....

Must be able to cope with any future requirements and not pigeonhole functionality.

- Create Models
  - store basic types strings, number
  - store associations between two models
  - store files
- Fetching
  - attributes
    - Request primitive attributes such as strings, numbers
    - request associations
  - provide a filter to a query
    - request a record given the records id
    - request a record based on its attributes i.e requesting published records
  - sort queries
    - by attributes
    - by associations attributes
  - pagination queries
- Mutations
  - create records
  - Update records
  - delete records
  - add/remove association records
  - upload files
  - validate data
- Sessions
  - Authenticate users
  - Specify users permission to access data
- Consistency

  - Use web sockets to be alerted to updates
- Permissions
  - Specify access (read + write + remove) of users on:
    * records
    * attributes
- Provide a way to change production databases safely
- GUI
  - Provide a way to create a database
  - Provide a way to create a model
  - add/update/remove attributes from models
  - seed data
  - view records for a model

6 Solution

This chapter will start with an overview of what Skeem is and why it is useful. Then each part of the system will be discussed in more depth highlighting what the part does and how it is used. This chapter will not cover exactly how the system works or the reasoning behind decisions, this will be done in the chapter on implementation.

6.1 What Is Skeem

Skeem is a tool which allows developers to easily create and manage a database and then write queries to request data which can execute safely from the client side.

Skeem has developers declare a schema of what data they want to store. The system will then automatically create a database tailored to this specific information needed, including optimizations to certain columns such as adding indexes to allow efficient retrieval of associated records.

These queries are specifically designed to fulfill the common needs of web applications, including:

- fetching data in tree structures
- authenticating requests
- handling file uploads

Skeem is agnostic of the exact view technology that is used and provides a client which can run in any javascript environment.

Skeem can be fully setup, configured and maintained without the need to write any code. Instead it is managed through either the command line or through a graphical interface. These interfaces provide instant feedback of any errors that occur when changing things and also provide help information to aid Skeems usage. This helps to solve the issue of training. There are of course still intricacies with using various sub-systems which will require additional help, however, to solve this issue skeem contains a fully set of documentation detailing many aspects.

6.2 Who Is Skeem Built For

Skeem is designed to be used by web developers. They are expected to be somewhat technically minded but do not need to have any knowledge of how databases work or function, nor do they need to know SQL to any level.

6.3 Value Proposition

Comment: *disucsss how skeem solves all the issues in the problem definition*

Comment: *Needs doing*

Skeem attempts to replace the database, api, authentication, and file hosting subsystems with a single unified system.

By moving allowing queries from the front end it helps to reduce the problems of cognitive complexity as it is easy to see exactly what data is being used.

Another advantage of collecting queries on the front end is it allows developers to request exactly the data they need, this then eliminates the problems associated with API design

6.4 Solution TODO

- File management
  - Files are managed by fileProviders. A fileProviders controls how a file gets stored and how it is retreived.
- Server
- Config

6.5 The Schema

The schema is the central part of all of Skeems functionality. It is used to derive the database structure, covering everything from the tables to the columns, indexes and triggers. The schema is also used to validate the queries being sent and transform them into SQL which can the be sent to the database.

6.6 Models

Models define the actual data to be stored. Models also define a set of permissions determining who can access given records.

6.6.1 Attributes

A models attributes are similar to a tables columns in a relational database in that they define a specific data type to store. Each attribute has a unique name in the context of the model and stores a single data type.

There are many built-in attribute types designed to cover all common use cases of data storage. The built-in attributes include:

- string
- number
- boolean
- date
- associations
- passwords
- files

Each attribute type has an set of configuration options to define how it should function, including how it is retrieved and how it should be stored. For instance booleans can set their default value, numbers can be declared to be an integer, or strings may enforce uniqueness.

Many of the attributes have some configuration related to validation. For example, strings allow you to declare that they must be present (not an empty string) before being able to save a record, or numbers can specifiy a minimum and a maximum value.

One key feature of skeem is its ability to request trees of data, this is done via the association attribute. Association attributes are configured with a model they are leading to, this determines where to go when following the association. They also define a type of relation, either a "has one" or a "has many".

6.6.2 Scopes

Scopes define subsets of data like published articles, popular products, banned users. They are built from many operator functions which will ultimately compile down into a single value.

There are many built in operators which provide range from very simple processes such as `eq` for equivalence or `lt` for less than, up to complex ones such as `path` which will traverse through many associations such as `invoice > product > category > name`. If the result after processing the operators is "truthy" for a given record then that records will be part of the subset. By combining different operations you can form a large array of filters covering many use cases.

Listing 1: A filter using the 'eq', 'attr', and 'value' operators to filter only records whose name equals 'some text'

```
{
  "filter": {
    "eq": [
      { "attr": "name" },
      { "value": "some text" }
    ]
```

```
  }
}
```

The following is a list of all operators built into skeem.

- eq
- lt
- lte
- gt
- gte
- in
- path
- empty
- anyIn
- not
- and
- or
- attr
- value
- param
- session
- id
- now
- like
- ids
- query
- scope
- associationEquals

6.6.3 Migrations

Mutations of the schema are done through migrations. Migrations define a specific change that wants to be made such as adding a model or renaming an attribute.

Migrations present a way to easily make changes to the schema in a way which is repeatable, this is essential as when building an application there are usually separate environments for development and production. Migrations, therefore, allow for a record of changes need to be made so after creating a feature in the development environment they can be run in production to mutate that schema.

When a migration is executed it stores some additional data about what was changed. This allows migrations to be reversed. The migrations to add a model, upon reversal, will delete the model thus restoring the schema to its previous state.

6.7 Fetching Data

Fetch requests allow developers to load specific information from the database.

Fetch requests are written in a declarative manner. This means you simply declare that you want all the published articles with their comments and the system will calculate the necessary steps to achieve this goal. This is opposed to traditional SQL where you have to state exactly *how* to retrieve the data e.g join these two tables together using this column.

Requests are made up of a target model which acts as the root for the query and some optional configuration specifying exactly what data is needed. This includes the attributes to retrieve for each record, how to order the records, and a filter determining which records to receive.

A fetch response will always be an array of records, where each record will contain its own id as well as any additional attributes you requested

Listing 2: A request for fetching published articles.

```
{
  // starting from the articles
  "articles": {
    // load the ones which are "published"
    "filter": { "scope": "published" },
    // get their names and bodies
    "attributes": ["name", "body"],
    // sort by their createdAt date newest first
    "sort": { "attr": "createdAt", "dir": "desc" }
  }
}
```

6.7.1 Attributes

The attributes part of the query specify what data you want to receive for each record. In the request, they take the form of an array where each element specifies a separate attribute. If no attributes are specified then each record will just contain their ids.

When requesting an association attribute you can specify another complete

fetch query nested within. This includes its own attributes, filter, sorting, etc. Skeem will then process this configuration in the context of the associations model whilst also limiting the results to only associated records.

The attributes of the associated attribute can further contain more associations. By nesting these queries you can obtain a complete tree of data.

Listing 3: Retrieving the comments attribute and specifying additional attributes and a filter

```
...
"attributes": [
  {
    "name": "comments",
    "attributes": ["author"],
    "filter": { "scope": "topRated" }
  }
]
...
```

In certain circumstances you may wish to rename the attribute during the request. This could be, for instance, if you need to request the same association attribute with two different filters. This can be done by specifying an `as` property along side the attribute name.

Listing 4: This query will retrieve the name attribute but will name it "title" in the response."

```
...
"attributes": [{ "name": "name", as: "title" }]
...
```

6.7.2 Filter

Filters restrict what records are retrieved, by default a query will return all the records for a given model. They are identical to scopes and have access to all of the same operations.

It is recommended though that filters should just use the `scope` operation where possible. This is desirable as it allows for sharing logic between parts

of the app - it is unlikely that the notion of published articles will change throughout the app. Using scopes also allow for changing the definition of filters across the app without having to update each use of the filter.

Listing 5: A filter using the "published" scope.

```
{
  "filter": {
    "scope": "published"
  }
}
```

There are times, however, where using a scope would be non-desirable. If, for instance, you were expecting to retrieve a single record, e.g you want to find the blog article with a particular title. Creating a scope for each one of these cases would cause not only result in the creation of lots of scopes, but also creates a separation between what data is being received and what data is needed - one of the main problems Skeem is addressing.

6.7.3 Sort

Sorting data is an extremely common and essential capability of data retrieval: most recent tweets, newest article, product name. When sorting data you specify what you want to sort `by` and the `direction` you want to sort: either *asc*ending or *desc*ending.

Listing 6: This query will return all articles ordered by the articles "name" attribute.

```
{
  articles: {
    sort: {
      by: "name",
      direction: "asc"
    }
  }
}
```

You can also specify an array of sorting criteria. Doing this will sort the data

initially by the first item, then resolve conflicts with the next item in the list.

6.7.4 Pagination

The pagingation options allows for the splitting of a request into multiple discrete pages. This is a common practise throughout the web as having the user download thousands of records would lead to a slow and unpleasent user experience. The functionality of pagination is akin to that of the `LIMIT` and `OFFSET` abilities of SQL. The returned data will be equivalent to a standard array of records.

Listing 7: This query will return the second page of articles where each page holds 30 records.

```
{
  articles: {
    pagination: {
      page: 2,
      perPage: 30,
    }
  }
}
```

Record Count

When pagination is specified, the response will additionally contain a count of the number of the total number of records records you would have received without pagination. This is useful for showing users controls for navigating between pages.

$$totalPages = ceiling(totalRecords/perPage)$$

Retrieving the record count can be disabled by passing the option of `withCount: false`.

6.8 Mutating Data

Comment: *TODO this section needs doing*

Mutation requests allow for records to be created, updated, and removed.

Mutations, like fetches, start by specifying a root model. You then can specify a list of changes wanted to be applied. These changes are either `create`, `update` or `destroy`

6.9 Permissions and Authentication

The client should always be assumed to be dangerous and because queries can be sent directly from them it is imperative to be able to ensure their identity and limit their access accordingly.

6.9.1 Authentication

Before we can control a users access we must first be able to determine who they are. Skeem provides a couple of ways in which to authenticate someone: stored identifier (email, username, etc) with a password or through an oauth2.0 provider such as Facebook or Google.

Skeem allows for multiple session providers to be created, this allows sites to present users with multiple options to log in.

6.9.2 Permissions

With authentication we now have two distinct user states: authenticated and anonymous. Each model allows for a default scope to be applied for each of these roles, meaning that you can declare that only authenticated people are able to view all articles whereas anonymous people may only view public ones.

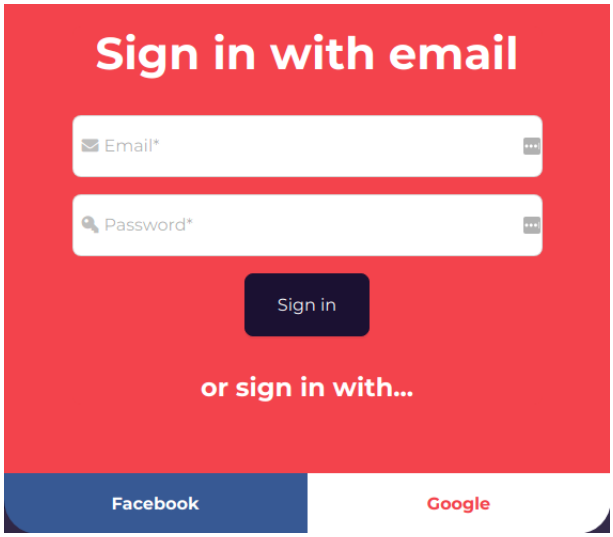


Figure 1: The log in screen for Resooma.com showing options to authenticate with email and password or by google or facebook

6.9.3 Roles

Often sites have a clear set of user groups such as admins, authors, sellers, or guests. Skeem allows for the declaration of additional roles. These roles have a name, and also a scope used to determine if a particular user has the role. Users can have multiple roles simultaneously.

Permissions can be defined on each model for each of the custom defined role. When resolving if a user can access a given record Skeem opts for the most lenient restriction. For instance if authenticated users cannot access the addresses model but admins can, then skeem would allow anyone who is both authenticated and an admin to access the records.

6.10 Management

There are two methods to managing a skeem application: through the use of a command line interface or via a GUI. Both these methods provide ways to create and run migrations, view the current schema, start the server, etc.

6.11 The Client

In order to facilitate the use of Skeem, a library is provided which contain functions for interacting with Skeem from the client. This includes functions to perform fetches and mutations. These functions provide basic type checking on requests to ensure that the format is correct, they also preprocess the response from the server to ensure consistency.

The client also presents functions to facilitate Oauth authentication. The library will perform all the necessary requests to fetch the security tokens before redirecting the user to the relevant 3rd party portal. A response will be listed for automatically and upon receiving it, the server will get notified and the user authenticated. This means adding a full authentication flow to an application involves calling a single function.

- Offers server functionallity which doesnt persist sessions

6.12 Plugins

APIs are large and complex systems which cover an incredibly broad range of use cases. It would be almost impossible to foresee every use case of Skeem and to allow for every possibility. To cope with this, Skeem has the ability to augment functionality by the way of plugins.

Plugins are javascript files located in the project folder. When the server starts, these files are loaded, type checked, and inserted into the system.

6.12.1 Custom Attributes

There are many different types of datum which you may want to use which don't fit within the bounds of the built in types.

The attributes built in to skeem use this very system i.e they contain no special functionality which could not be implemented outside of the core code.

6.12.2 Custom Operation Functions

There are a myriad of different and obscure filters you may want to perform within a database. Whilst skeem contains a lot of built-in operations designed to cover a wide variety of use-cases, it is implausible that they cover every possible desire.

Therefore, like with attributes, skeem provides the ability to create custom operation function outside of the skeem source and have them loaded in dynamically and used seamlessly with the built-in operations.

Creating An Operation

Comment: *this section is too tutorial-like it needs to be more abstract e.g what a plugin does rather then how to create one*

An operation consists of a single function which must return SQL.

Listing 8: The simplist custom operation - it would always return false and so is utterly pointless.

```
function myPointlessOperation() {
  return { value: `'this will' = 'always be false'`
     }
}
```

In order to create a useful operation the function is passed some variables concerning the request. The most useful of which is the `value` argument. The `value` contains the data passed to the operation. Using this we can produce a much more useful operation.

> The SQL returned is inserted into the query as is and so is essential that it is sanitied prior to being returned.

Listing 9: Returns all the records for which the name matches the value supplied. However there are major issues with this and should not be used.

```
function myBadOperation({ value }) {
  return { value: `"name" = '${value}'` }
}
```

The above operation shown in figure (FIGURE XXX) would technically do something which could be deemed as useful. You supply is a value and it will return all the results for which their name attribute is equal to that value. There are a couple of issues with this operation though which means it should not be used.

The biggest and most critical issue is that it does not sanitize the value. This means it is an entry point of an SQL injection. This is relatively easy to solve though through the use of a sister package of skeem named es-qu-el. That is outside the scope of this solution chapter and is discussed in the implementation.

The second issue with the operation is that it makes the assumtion that the model you are currently fetching has a column in its table called "name". This is not always true for obvious reasons. To solve this issue we are passed another useful piece of infomation - the current model. With this we can search through the models attributes and check for the existance of a name attribute and if it does exist then throw an error.

This is an extremely common need for skeem and as such there exists a helper function to achieve this for you. It os exported from the `skeem-common` package and is called `getAttribute`. This function takes a model and the name of the attribute you wish to retrieve. By using this function you guarantee that the attribute exists and if it does not exists then you can be assured that you will get an error message consistent with that of a built in error.

Listing 10: Checks to see if name actually exists on the model being queried.

```
1 const { getAttribute } = require("skeem-common")
2
3 function mySlightlyBetterOperation({ model, value }) {
4   const attribute = getAttribute(model, "name")
5   return { value: `"${attribute.name}" = '${value}'` }
6 }
```

There is one final issue with this operation and that is it misses the opportunity for optimization. As well as returning the SQL value operations can also

return the type of result expected back from the SQL - in this case it would be a "boolean". Supplying this infomation from an operation allows Skeem to optimize the SQL query and possibly not even execute anything. For instance consider the following query:

```
{
  filter: {
    eq: [{ value: "a string" }, { value: 123 }]
  }
}
```

The `value` operation will return the types of string and number (as well as the sanitized SQL value). The `eq` operation then checks these types to see if they are save it is then it will place the values around an equals sign and return it as expected. If, however, they are different then the `eq` operation will return with a type of "false". If the full filter resolves with the type of "false" skeem will skip executing the query as it knows nothing would be returned. Therefore by returning the correct type skeem can potentailly optimize and avoid the database altogether.

Possible types include:

- string
- boolean
- number
- record
- collection
- any - the type could not be determined and so could be anything, this is the default when no type is returned

Listing 11: Checks to see if the attribute actually exists and returns the correct type

```
const { getAttribute } = require("skeem-common")

function myPassableOperation({ model, value }) {
  const attribute = getAttribute(model, "name")
  return { value: `"${attribute.name}" = '${value}'
    `, type: "boolean" }
}
```

It should also be noted that this is a poor use of a custom operation. This operation does not achieve anything that you could not do with the built in operations. Although it does save a few characters, it adds on additional knowledge needed by other people working on a project, the need for testing, and another function which would need maintaining over the life time of the project. Operations should only be added to achieve results which are either not possible with the pre-existing functions or highly impracticle to achieve.

Nested Operations

It is very common for an operation to need to accept an operation object as its value. If you could not compile nested operations then you would not be able to create functions like: `eq`, `lt`, `and`, `not`. This would make things a little tricky. Therefore, along with model and query you also get supplied with a function named `compile`. This function accepts an operation and returns the `{value, type}` object.

Listing 12: A simple implementation of the eq operaion.

```
1 const { getAttribute } = require("skeem-common")
2
3 function simpleEq({ compile, value }) {
4   const left = compile(value[0])
5   const right = compile(value[1])
6   return { value: `${left.value} = ${right.value}`,
      type: "boolean" }
7 }
```

The Request Context

the final argument passed to an operation is `ctx`. This is the current context for the request. With this it is possible to access infomation such as app configuration, the current session, and the database connection.

TODO

Using An Operation Plugin

To use an operation you first must create a javascript file within the `<skeem root>/operations`. The name of this file does not matter and can be anything. This javascript file must have a default export of an object where the keys are the names of the operations and their values are the functions as described above.

Listing 13: A full operation plugin file

```
1
2  module.exports = {
3    isANumber: function({ value }) {
4      if (typeof value === 'number') {
5        return { value: true, type: 'boolean' }
6      } else {
7        return { value: false, type: 'boolean' }
8      }
9    }
10 }
11
12 // Usage:
13
14 fetch:   {
15   articles: {
16     filter: {
17       { isANumber: 123 }
18     }
19   }
20 }
```

6.13 Documentation

Documetnation exists for skeem which contains guides and examples on how to use the system. This is essential to allow other developers to actually use the system. https://cd2.github.io/Skeem/#/

7 Methodology

Comment: *READ THROUGH*

7.1 Development Strategy

When a new feature is to be added it would first have high level tests written aimed to test the final functionality of the feature. For instance when first implementing fetching I wrote tests asserting that given a particular query a specific piece of sql was generated. I would then proceed to implement the feature, using the tests as a guide for when the work was complete.

Listing 14: Example of what the high level tests would assert.

```
1 Given: { articles: {} }
2 Expected: `SELECT "id" FROM "articles"`
3
4 Given: { articles: { attributes: ["name"], filter: {
      eq: [ { attr: 'name' }, {value: 'test'}] } } }
5 Expected: `SELECT "id", "name" FROM "articles" WHERE
      "attr" = 'test'`
```

Once all the tests were passing, if there were additional features which either appeared during implementation or that were initially excluded for simplicity, I would add more high level tests asserting the new functionality. I would then proceed to implement these features until the new tests were passing, adding more tests until development was complete.

When the feature was complete, assert by a suite of passing tests I would begin testing the code at a more granular level. I would select functions which were either complicated or hard to test at a high level (maybe code branches for very specific circumstances) and write specific unit tests.

The specifics of how the tests are written are discussed more in the chapter (Testing)[#testing].

8 Implementation

This chapter will cover the implementation details of Skeem including some reasoning behind why certain decisions were made.

8.1 Technologies

8.1.1 Typescript

Some parts of Skeem runs on the client side and thus had to be built in javascript.

Typescript is a superset of javascript which adds typing capabilities to the normally dynamic language (**???**). This typing infomation allows compile time code checking which greatly assists improving relaibility of code as it helps to ensure against common trivial bugs. Typescript is transpiled into standard, es5 compliant, javascript meaning it is able to run on all modern browsers. This compatibility is essential because if skeem only supported the latest version of chrome then it would instantly elliminate the possibility of using the system on any standard website which has compatibility as a requirement.

Typescript also aids other developers using the system as the typing infomation is used in most modern IDEs to supply intellisence infomation allow features such as auto completion, inline errors messages, and hints for expected arguments and returns.

NodeJs is a javascript runtime designed to build scalable network application (**???**). NodeJs was a logical choice as it allowed writing server code also in javascript which allowed consistent interfaces to be constructed. It is also much easier to develop a system when writing in a single language as their is less mental energy exerted to convert from one environment to another.

8.1.2 Postgres

Postgres is a object-relational database system (**???**). Postgres has powerful, inbuilt JSON processing capabilities. It allows for storing JSON objects

natively as well as writing queries which inspect the contents of JSON. More importantly Postgres allows for the construction of JSON objects with queries themselves. This makes Postgres a very logical choice when the goal is to create trees of data as the database can pre-format the response greatly reducing the need for much post query processing.

Another feature which postgres offers is an optimizer which can automatically transform subqueries into join statements. This optimzation is greatly taken advantage of in skeem as throughout the code base there is not a single joining statement. Further reasons for this are discussed in the chapter on fetch query sql generation.

Postgres also provides the ability to write custom database functions relatively easily. These functions help to encapsulate complex pieces of reusable logic and are optimized by postgres in order to maintain performant queries. They are used in a number of place throughtout skeem such as to trigger update messages for live syncing and to format responses in certain circumstances.

8.1.3 Other Libraries and Services

Skeem also takes advantage of a number of prebuilt libraries and services.

NPM is the defacto package manager for node (**???**), it provides easy hosting and distribution of node packages and is the method skeem uses to manage its publications.

Pusher is a web service specialised in providing real-time functionality to applications (**???**). It provides simple wrappers are web sockets as well as fallbacks to ensure compatibility accross, even outdated, browsers. Skeem uses Pusher send messages to clients in order to enable the live updating capabilities.

Amazon Web Services are a cloud computing platform(**???**) which provides relatively affordable file hosting and is integrated into skeems file storage capabilities.

React React

TODO

8.2 Project Structure

Due to the size of the skeem code base (pushing 12'000 lines) and the range of environments it runs on (server and the client). It was essential to split up the project into logical parts. However I did not want to fully isolate each section due to the tight coupling of the interactions e.g if the format of the fetch response changes then you have to update the server code to handle this new change as well as the client code to keep the format consistent from a developers point of view. Due to this I structured the code as a monorepo.

> "A monorepo is a software development strategy where code for many projects are stored in the same repository" ("Monorepo" 2019).

Then using a tool named Lerna I was able to manage the projects simultaneously. Lerna automatically resolves the dependency order so, for instance, when you attempt to build the project it knows that A depends on B which depends on C and therefore wait for A to build before moving on to B then finishing with C.

Skeem is broken up into 7 packages, 5 directly related packages, each prefixed with "skeem-", specific to skeem and 2 auxilary packages which were extracted and can provide useful functionallity independently:

8.2.1 Skeem-Server

This package contains the majority of the logic, it contains the implemention for processing requests, creating a database, session authentication, migration creating and validation, etc...

The `skeem-server` package is split into two, confusingly named, parts: a manager and a server. The manager contains the all the functionallity of the app, it controls loading the schema, producing and executing SQL, updating config, etc. The server simply listens for http requests sent from the client, performs basic format type checking and then calls manager functions in order to create a response.

Server

Manager

8.2.2 Skeem-Cil

This implements a command line interface for interacting with skeem, it implements no fundamental logic and instead acts a wrapper around the server.

8.2.3 Skeem-Gui

Similar to the CLI, the gui acts a wrapper around the server functionality and displays the information in a visual application.

8.2.4 Skeem-Client

Provides functionallity for a client

8.2.5 Skeem-Common

Holds common functionallity needed between packages, such as error messages

8.2.6 Es-Qu-El

provides helper functions to generate and sanitize SQL statements

8.2.7 Typer

Typer standardies type checking accross the app.

8.2.8 Overseer

declarative CLI generator used to power Skeems CLI.

8.3 The Schema

The schema is a single JSON object with three main parts: db, models, and providers.

Listing 15: The empty schema.

```
{
  "db": {
    "functions": [],
    "tables": []
  },
  "models": [],
  "providers": []
}
```

8.3.1 Db

The db section stores the current structure of the database.

Tables

The predominant use of the db array is storing the definition of all tables with names, columns, indexes and constraints.

By default every table has an id field which acts as the primary key. This field is of type uuid and has its value initialized automatically to a random value generated by a cryptographically secure random generator. Having a dedicated, synthetic primary keys was chosen, as opposed to natural keys, as it helps enforce a consistency throughout the system, this has the added benefit of removing some choice from the developer and thus removing the need for some additional knowledge about databases. It also prevents having to deal with database wide updated when a composite key changes.

Listing 16: The schema definition of a table

```
{
  "name": string
  "columns": IColumn[]
```

```
    "indexes": IIndex[]
    "triggers": ITrigger[]
}
```

Functions

Also contained is a list of all custom functions, each having a name, language, and code body. Custom functions allow for complex functionality to be abstracted into the database itself. Skeem uses a custom function in order to properly format its results when paginated this function however is not stored within the schema as it is not application specific.

The functions are an experimental addition and as of the time of writing, no public features exist which utilize these functions.

8.3.2 Models

Models define exactly what data their is and how it can be queried. Each model has a globally unique name used to identify it. They are comprised of many mostly independent parts.

Models also store a tableName field, this references a particular table stored in the schemas `db` object. The tableName is independent of the name of the model, though when created they use the same value, as it allows the model to be renamed without having to alter the database and any references to the given table.

Attributes and permissions are complex in and of themselves, and therefore will be discussed sections 8.10 and 8.12 respectively.

Scopes allow for the definition of subsets of records, such as published articles. Each scope is an object containing a name, to identify the scope; a query, which defines how to realise the subset; and a set of parameter definitions which defines the arguments the scope can accept. The query is an operator tree, which are discussed at length in section 8.11.

When the private field is true, Skeem will prevent querying the model directly. Instead the only way to access the model is through an association. This one done as skeem does not support polymorphic associations (an association

that leads to one of multiple different models). Therefore, it is not possible to create accurate permissions in certain circumstances. For instance, if you had a model for storing addresses, and you wanted to use this same model for many other models (e.g for users and for properties), then you would want the permissions to be: users can only access their own address or those used for properties. Since you cannot create an association from addresses to the other models their is no way to determine if the address is used for properties, therefore by setting the model to private you avoid the issue as you would need access to a joined record to access an address.

The `live` and `callbacks` keys store information pertaining to features which are currently a work in progress.

Callbacks allow for actions to be taken after certain events, such as send an email when a new record is created or perform a mutation before mutating a record. These callbacks will greatly increase Skeems flexibility but as of yet they are in the testing stage before being released.

Live records utilize web sockets to inform the client of when records change. From the client, developers are able to subscribe to a given fetch query and then have a callback triggered when the results of that query update. This allows for views to always display the most accurate information and also enables applications such as real-time messengers. This feature is currently hidden behind a flag, as further testing needs to be done and more thought put into the client API before it could be released publicly. A proof of concept has been created and results have been extremely positive.

Listing 17: The schema definition of a model.

```
{
  "name": string
  "tableName": string
  "attributes": IAttribute[]
  "scopes": IScope[]
  "permissions"?: { [roleName: string]: IPermission
    }
  "private"?: boolean
  "live"?: { enabled: boolean }
  "callbacks"?: IModelCallback[]
}
```

8.3.3 Providers

Comment: *TODO: the providers stores a list of available*

8.4 Database Usage

Skeem utilizes a single postgres database in order to store and query for its data. Skeem fully manages this database and is responsible for create all of its tables and extensions.

8.4.1 Skeem-* Tables

Skeem relies on a handful of predefined database tables in order to function. To differentiate these tables from application specific ones, their name is prefixed with "skeem-", and application tables are prohibited from using this prefix.

skeem-schema Stores the applications schema

skeem-migrations :Stores a list of all migrations

skeem-sessions :stores a list of sessions, including what record the session was for, when the session was created, when the last activity for the user occurred and what session provider was used to authenticate.

skeem-files :stores references and information for uploaded files

skeem-version :stores the current database version

8.4.2 Functions

Skeem also relies on a few custom functions in order to provide certain features. These functions, like the tables, are all prefixed with "skeem-". This is done entirely for future proofing the application as custom functions are not possible to create as of the time of writing.

The `skeem-array_to_object` function accepts an JSON array as input and outputs an object with the indices becoming keys. The object also has a

length property defined. In practise, this function would transform `[1,2,3]` into `{0: 1, 1: 2, 2: 3, length: 3}`. This is used for formatting paginated requests, specifically it allows for the `recordCount` variable to be added to the object. Then in javascript the objects prototype to be that of the built in array. This results in an object with a `recordCount` property which has all the same functionality as a standard array. All constructed within a single database query.

The `skeem-format_results_as_object_with_count` function takes two queries as input, the first returning an array of records and the second returning a count. This function will call the `skeem-array_to_object` function on the results of the first query and add the `recordCount` property being the result of the second query.

8.4.3 Upgrading the Database

In order to accommodate new functionality over time the database will likely have to change, new tables may be required, old tables may need to columns, etc. In order to facility this need skeem provides a database upgrade mechanism.

This upgrade system works by keeping track of a database version number and a list of steps on how to upgrade from one version to another. When the server is started the current version number is loaded from the database and compared to the most highest upgrade number available. If these are equal the database is fully up to data and the system proceeds as normal. If however they are different the database goes through the update process.

The update process involves iterating through all upgrade steps starting from the current version number counting up until all steps have been executed. After each step the version number is updated in order to keep it in sync with the database state allowing for the possibility of an error within the upgrade step.

As of the time of writing there are 6 upgrade stages (starting from 0). These stages perform the following:

0. Check to see if the database exists, if it does not the attempt to create it.

1. Install neccessary extensions, setup tables, initialize empty schema.
2. Rename tables from "cord-″ *to* ″*skeem*-" (The project under went a renaming to prevent conflicts with pre-existing systems) and add columns `exectued` and `timestamp` to the migrations table.
3. updating the schema format to accomadate some new features.
4. Create response formatting functions to remove inconsistencies of certain edge case requests.
5. Add a `loggedOut` column to the sessions table. previously sessions were deleted.

When creating a new application the version number would attempt to be loaded but an error would be thrown as no database would exist. This error is caught and the version number is deemed to be -1. This means that when creating a new application by simply starting the server a database will be created automatically (as step 0 would be the next to be execute). This greatly improves the time required to setup and start using a Skeem application.

8.5 Migrations

Migrations do not directly effect the database instead they modify the schema, then the differences between the old and the new are calculated and a set of transformation steps are generated. This split is present as it allows for running migrations repeat

Migrations are comprised of a type and some data.

- Migrations work to mutate the schema
- They allow multiple developers to work on a single application
- A record of all changes made to the database
- Migrations serve incremental, reversible changes to the schema
- Migrations are stored in files within a folder named migrations located in the root of a skeem project
- This allows for migrations to be transferred between computers

- only stored in files to transfer computers, the ones which get executed are actually stored in the datbase

8.5.1 Migration Structure

Each migration is comprised of a `name`, `timestamp`, `type` and `data`. The type references a specific migration handler such as `models/create` or `roles/destroy`. The data stores information specific to the given provider and can vary wildly, for instance, the `models/create` handler uses just a `name` field, whereas the `models/scopes/create` requires a the name of the model, to add the scope to; a unique name, to identify the scope within the model; a set of parameters to be passed the query; and the query itself which defines how to identify the subset of data.

The name is field is automatically generated and exists purely to for displaying information to developers. Some migration handlers specify a way to generate a name, if the handler does not specify then the name defaults to the name of the handler.

The timestamp field stores when the migration was created. It is used to ensure the order in which migrations are executed remains consistent.

8.5.2 Storage

Migrations are stored both in the database (in the `skeem-migrations` table) and in the file system (in the migrations folder).

Migrations are stored in the file system so that they can be committed to a VCS (version control system) and therefore synced between different machines. Each migration is stored in a separate file to help reduce conflicts when multiple migrations are created by different developers. Each file contains a json object holding the migration type and the data. The downData nor the execution status is stored in the file, as not all developers would have executed the migrations.

Migration file names are in the format of `timestamp.name.json`. This done to help ensure migration files have unique names, there is no guarantee the the migrations auto-generated name is unique but it is unlikely that two

developers would have created migrations which result in the same name at the exact same millisecond. Another advantage of including the timestamp first is that migrations appear in the correct order within the folder - though not a necessity it does assist developers if they have to look at the migrations.

Initially migrations were stored in the filesystem until they were executed, at which point they would have a row created. If a migration was in the database then it had been executed. This, however, proved to be confusing as to get a complete list of migrations you first had to load those in the database, then read all the files and then eliminate files for which the timestamp and name was present in the database. There was also an issue of migrations loaded from files had different properties then those form the database - the database rows had a downData field and an ID field. I therefore switched to storing all migrations in the database allowing a full list of migrations to be obtained from a single query and used the file system purely for VCS abilities.

8.5.3 Syncing

Before any action involving migrations is taken including execution and a roll back, a sync is performed between the migrations in the database and those stored in the filesystem. Firstly, any migration file which does not have a corresponding row are new migrations created by other developers and therefore have a row created.

Next any rows which do not of a file associated are found, the migrations must have been removed and so should be deleted from the database. If one of these migrations has been executed then an error is throw as the system cannot remove the migration as it would break the ability to rollback.

Finally all other migrations (they exist in both the file system and the database) have their data compared to ensure they are identical. If they are not then the data from the filesystem takes presidents. If the database row has been executed however, an error is throw as once again changing the data could prevent rollbacks.

8.5.4 Running Migrations

Before running migrations the existing schema is loaded. Over the course of running the migrations this schema will be mutated.

The first step in running a migration is to load the migrations handler. This handler then gets given the migrations data, the schema, as well as access to a general context object which stores application configuration as well as functions to assist with logging and database access.

Despite each handler being completely free to perform any actions they want to the schema, in general, they all follow a very similar pattern. Each handler is built in complete isolation from the rest of the system, only exposing key methods, the data being passed in has no insurance of containing the correct information or being the correct type. Therefore, handler first type checked the data to ensure it is of the expected format and contains all the relevant information. For example, the `models/create` handle first checks to see if the data contains, exclusively, a name property. Then this name is checked to be a string, only containing letters and underscores and is not prefixed with `skeem-` (this is a reserved prefix). If any of these checks fail a `migrationValidation` error is thrown along with the message of specifically what is wrong. The migration system is then free to act on this error as it chooses, if creating a new migration then the system will prompt the user to amend the data they specified, or if migrations are being executed then the process is aborted.

If the data is correct then the handler will modify the schema, mutating it as needed.

The procedure is repeated for all pending migrations. If all migrations were executed successfully then the new schema is saved to the database and the migrations are marked as executed.

8.5.5 Database Diffing

After a new schema has been produced, a list of change steps must be realized in order to mutate the existing database. This happens are migrations are run or when the application is initialized. In the latter case the empty schema is used as the old schema.

The first step of this process is to compare the new schema and the old one in order to find what is actually different. Because the models and providers only exist in the abstract, as opposed to the db property which is backed by database tables, only the db of the schema is diffed.

The first step in diffing the dbs is to isolate which tables are new, which have been removed and which have been *potentially* updated. The name field of the table is used to link the old and the new schema. If a name exists in the old schema but not in the new, then the table is marked as deleted. Similarly, if a name exists in the new schema but not the old it is marked as created. If the name exists in both then the table is marked as potentially updated and undergoes a further diff.

For each new table the appropriate `CREATE TABLE` SQL query is generated and appended to a list of all pending database queries. Like wise for each removed table a `DROP TABLE` query is produced and appended to the list.

For each updated table each column

The list of sql commands is then executed. To do this, first, a new transaction is created within the database. This means if an error occurs within the mutate steps the database can be fully restored to prior to the mutations. Without this then the schema could become out of sync with the database. The list of sql statements is the run sequentially. After each step has been executed the schema in the database is replaced by the new schema. Finally, a commit message is sent to Postgres informing it to proceed with the mutations. By updating the schema within the same transaction ensures synchronicity between the tables and the schema.

8.5.6 Rolling Back

Migrations all have the ability to be reversed.

When a migration is executed it has the option of returning some data. This data will be passed back to the migration when rolling back. This is used, for example, when creating an association the migration will return: the name of the added attribute and the name of the created joining table. With this information the migration is then able to fully undo any effects it had thus restoring the previous schema.

Rolling back is not guaranteed to revert the database completely to its state prior to the migration, instead it simply guarantees the structure of the database will be identical. This is because some migrations such as deleting an attribute are lossy in nature and thus are not purely reversible.

8.6 The Server

Skeems server is a package which holds all the main functionality concerned with managing and Skeem application.

The server provides many functions to perform common actions in an application, this includes running migrations, starting a http server, performing a fetch query.

Interfaces such as the CLI are simply wrappers which call these helper functions. This means that both the CLI and GUI do not need to duplicate any logic in themselves and instead focus on conveying the data to the user in the most effective manner.

8.6.1 The Context Object

All helper functions require a context object in order to function. This context object holds all the information pertinent to the current skeem application. This includes the applications configuration, the root folder, connections to the database, and the schema.

The schema is stored in memory as it is assumed that the schema will never change without the server being reloaded and therefore caching it can prevent many database requests considering how frequently the schema is loaded.

To create a context object you supply the server with a path. The system then searches recursively upwards from the supplied path looking for a file named `skeem.json`. If this file is not found before the root directory is hit, then an error is thrown stating that no skeem application was found. If the file is found, its location is marked as the root directory and its contents are the configuration object. These two data are then form the basis of the context object.

The context object can also be instantiated by providing the config and root directly. This can be useful when testing the system as it does not require any files to actually exist.

8.7 HTTP Server

The HTTP Server is an interface into skeem just like the CLI. The only difference is this interface exposes a very small selection of the available interface The server handles all incoming requests. It checks

Skeems server handles all incoming requests

- The server hosts
- listens by http
  - so the client can comminucate
- request and reponse formats
  - why pure json
- authentication token

Skeems uses JSON to send requests and receive responses. This allows requests to be much more easily santized against maliscious attempts to manipulate the database in an undesirable way. In other words it helps prevent SQL injection like attacks.

> SQL injection is a code injection technique in which malicious SQL statements are inserted into normally safe SQL by having unsanitized input inserted into the query. SQL injection is a major security vulnerability (**???**).

Due to this sanitization ability queries are safe to execute on the front end and so eliminates the tight coupling between ..... This solves the tight coupling with disperate code issues between the standard client-server model.

All requests are fired to the same end point with a post request and take the form of a JSON object with `type` and `payload`. The type is used to determine what type of request it is: fetch, mutate, etc. and the payload contains specific information depending on the type.

Comment: *TREE OF DATA*

When the server first loads

Authenticating a user

- files in requests
- end point for retrieving files
  - calls to the .retrieve method of the uploaded handle (see section {#file_providers}. The returned stream is piped directly back as the response value.

8.7.1 Files

8.8 Fetches

All fetch requests perform a single database request independent of how simple or complex of a query is made. The query also returns data in the correct format and requires no post-processing.

Fetches start by performing some basic type assertions about the incoming request. The request is checked to be an object with a single key. This key is ensured to be a valid model name and the value is an object with some subset of the keys: filter, attributes, sort, pagination. If there are any additional keys the request is deemed to be erroneous. If any of these checks fail then a malformed request error is thrown containing additional information explaining the exact issue.

Listing 18: A malformed request and the corrosponding response

```
1 REQUEST:
2 { articles: {},  comments: {} }
3
4 RESPONSE:
5 { error: { type: 'malformedRequest', data: 'multiple
      models passed to fetch query, only one allowed.'
      }
```

Parsing a fetch request starts with the instantiation of a class named `SqlQuery`. This instance holds all the information about the final query to be executed, such as the current columns being selected, the where conditions, the limit etc. Each step in processing a the request gets given this object and simply has to adds the relevant data.

The `SqlQuery` object has an execute function which receives a connection to the database and returns the result of the query.

8.8.1 Permissions

The first step in processing the query is to find and apply and relevant permissions for the model. The process of how these permissions are discovered and processed is covered in section 8.12.

8.8.2 Filter

Next the filter of the query is processed, which is now comprised of the filters passed directly in the request and also the result of the permission step.

The filters are simply an instance of an operation tree, details of how this is compiled can be seen in section 8.11.

The result of compiling the filters is an SQL string and a predicted type. If the type is deemed to be `never` i.e there would be no results, then the request is aborted and the empty result set is returned. This optimization allows for many requests to be processed without the need to touch the database at all.

If the type is anything else, however, then the SQL string is simply appended to the `where` attribute of the query object.

8.8.3 Attributes

After compiling a filter the attributes are processed. Attributes define precisely what data is returned. If no attributes are specified within the fetch query then only the id is returned.

Attributes take the form of an array (this is asserted by a type check) this array contains either strings or objects. If an element is a string then it is inflated becoming `{ name: "the string value"}`. This creates an array of object each with a name property. The name property on the attributes relates to directly to an attribute on the model. This is looked up and an error is thrown if the attribute is not found. The attributes `get` method is then called allowing attributes to fully control what it means to retrieve the data.

Comment: *talk about aliases*

Listing 19: The algorithm used to parse the attributes part of a fetch query

```
1  for each attribute query in the array
2    if the attribute query is a string then
3      inflate it to become { name: "attribute query
         string value" }
4    end
5
6    from the selected model, find the attribute with
       the matching name.
7
8    if the attribute exists then
9      call the attributes get method
10   else
11     throw an unknown attribute type error
12   end if
13 end for
```

Details about how each different attribute type works to produces its SQL is described in section 8.10.

8.8.4 Sorting

Sorting controls the order in which the result records appear. Firstly, the sort query is checked to be an array, if it is not then a new array is created and the sort query is set as its only element. Each element of this array is now type checked, ensuring each is an object containing a `direction` (either **asc**ending or **desc**ending) and a `by` value (can be anything and will be

checked momentarily). If either of these are missing or the direction is an invalid value, then a `malformedRequest` error is thrown.

If the `by` value is a string then, the attribute is loaded from the model (throwing an error is missing). Then the attributes `sort` method is called which produces the adds the necessary SQL to the query object. If the attribute type does not define a `sort` method, an error is thrown stating that it is an unsortable attribute, this is the case for file and association attributes.

The `by` value may be an object with a key of association. In this case the `by` value contains a name of an attribute with the type of association as well as an `attribute` property. In this instance, the query will be sorted by the given attribute value for the given association.

Listing 20: Using the sort queries association feature to order results by their authors rating.

```
{
  "by": {
    "association": "author",
    "attribute": "rating"
  },
  "direction": "asc"
}
```

Sorting can also be done by the sum of an associations attribute. This is done by passing an object with the key of `sum` as the `by` value. This sum object contains the association attribute name and the name of an attribute on the supplied association. This then behaves similarly to sorting by an association attribute, however, in this instance the association must be a has many association. An optional `filter` property can be supplied to specify what records of the association should be summed.

8.8.5 Pagination

Finally pagination is applied - the simplest of all the steps. Like the other steps the pagination part of the request is type checked to be an object with the keys of `perPage`, `page`, and an optional key of `withCount`. `page` and `perPage` are numbers and the `withCount` a boolean.

The query limit is then set to be the value of `perPage`, and an offset is calculated from `page` and `perPage` and applied. If the `withCount` option is excluded of set to true then a flag is set on the query to include the count in the final result. This flag is discussed in more depth in the following chapter on SQL Generation.

$$offset = (page - 1) * perPage$$

8.8.6 SQL Generation

After processing is complete the query object must be transformed into SQL. The SQL is simply built up of a combination of the various elements supplied to the query by the previous step. This process is relatively simple as the previous steps provided trusted SQL into the query. This means no type checking nor sanitization needs to be performed.

Listing 21: Example logic within the SqlQuery to create the final SQL

```
1 SELECT {columns specified joined by ","}
2 FROM {table name}
3 WHERE {conditions specified joined by " AND "}
4 ORDER {orders joined by ", "}
5 LIMIT {limit}
6 OFFSET {offset}
```

This SQL string will return all the correct data, however, it would not be in the proper format. So before we execute the query we generate a new select query for which the from value is equal to the previously generated sql. This query then selects a json aggregation of all the results. This means the final result will be a single json object that can be returned directly.

Listing 22: Wrapping the SQL in another query to format the results

```
1 SELECT json_agg("results")
2 FROM ({the generated SQL}) "results"
```

The Withcount Flag

If the `withCount` flag is specified then a slightly different process is conducted for aggregating the results. In this instance we wrap the generated sql in the same aggregation query, but treat this as the input to the `skeem-format_results_as_object_with_count` function (described in section 8.4.2).

Then we generate a completely new query following from the supplied data, this time only processing the `from` and the `where` clauses, the select, order, limit, and offset are ignored. The select of the query is set to `count(*)`. This query will return the total number of records and forms the second argument of our final function.

Listing 23: Generated query if the withCount flag is present.

```
SELECT "skeem-format_results_as_object_with_count"(
  {the generated SQL},
  {the generated counting SQL}
) as "results"
```

8.9 Mutations

- what are they
- structure of the query
- how sql is generated

8.9.1 Asyncronous Tasks

8.10 Attributes

Each attribute on a model is made up of three properties: name, type, and data.

Each attributes' name must be a string, unique within a model, and comprised only of letters, underscores, and hyphens. They are also not permitted to start with the prefix of "skeem-".

There are multiple attribute handlers, each with their own definition of what it means to "set" the attribute or "get" the attribute. The `type` property

refers to which handler the attribute should use.

The `data` property acts as configuration for the attribute handler. It allows the handler to store configuration data. For instance, many attributes store specific validation information.

Skeem comes with a handful of built-in attribute handlers covering all common scenarios. There is also the support to create custom attributes via Skeems plugin system which is described fully in section 8.15.

8.10.1 Attribute Interface

Each attribute handler fulfills a implements a specific interface to allowing other parts of skeem to interact with them generically. This interface is comprised of four functions.

Get

The first is the `get` function. This function is responsible for returning the SQL required to retrieve the attributes value. There is no pre-conceived notion of what a "value" is, only that this function will return it. This function is most commonly called when processing a fetch request. It is this function that is called for every attribute listed in the request. Implementing this function is optional for a handler, excluding it simply means the attribute cannot be read.

The `get` function is passed:

- the context
- the schema
- the model
- the attributes name
- an optional alias indicating what the SQL value should be called
- the attributes data
- an instance of the `SqlQuery` class (see section 8.8 for details)
- configuration for the attributes request

Set

Secondly, is the `set` function. This, as the name suggests, is responsible for setting the value of the attribute. It is called during a mutation request. This like, `get` is not required to be implemented, and will simply throw an error if a mutation is attempted.

The function is passed:

- the context
- schema,
- model,
- database changes object (see section 8.9)
- database changes record
- the attributes data
- the new value for the attribute

Sort

The sort function allows attributes to define what it means to be sorted. It is called when processing the sort term in a fetch query. Like with `get` and `set` this function is optional, and will simply prevent sorting by the attribute.

It is passed:

- The schema
- model
- an instance of the `SqlQuery` class (see section 8.8 for details).
- the data of the attribute
- the direction of the sort, either `asc` or `desc`

Migrate

Finally, the migrate function is responsible for mutating the schema, when being created, updated, or removed. Typically, attributes, will create a single column in the database which stores its value. This function can optionally return data to be stored as part of the migrations downData and will be passed back to the migrate function upon further migrations. This function has to be implemented, else the handler is deemed invalid.

This function is passed:

- schema
- model
- the table for the model
- name of the attribute
- the attributes data
- the down data of its previous migration

8.10.2 Strings

The string attribute handles the storage text. When migrating string attributes will create a single column in the database of which the name will match that of the attribute.

Postgres offers multiple column types designed for handling strings, char, varchar, or text. Skeem elects to store all strings using the text type `text` data type. All the text types are stored using the same underlying data structure and therefore have the exact same storage requirements, the only difference between them is the number of cycles used to calculate their lengths. This loss of speed in this specific use case is an acceptable compromise in favour of simplicity.

All string fields are not nullable and will always default to an empty string. This makes removes many checks which usually occur throughout front-end code which have to check if the field is null before its use. Also when a HTML form field is displayed, its value will always be a string, never null. Therefore by removing their ability to store null you eliminate the difference between fields which were once tied to an input and those which were not.

When creating a string a default value can be declared in its configuration. This default value indicates what the string should be initialized to when a record is created and no value presented. This default does not take effect if the string value is updated to be empty. It is set as a database level constraint for the column and will therefore take effect automatically when a record is created.

Strings accept the `required` validation. This means that the string must contain a non-empty value before a record can be created. It also prevents the value being set to an empty string. This option makes no changes at a database level, instead it is purely enforced within the `set` method.

They also can be declared to be `unique`. Doing this creates a database constraint for the selected attribute, enforcing that no two records share the same value across the table.

Finally, strings have two configuration options regarding their casing. If a string as marked as `caseInsensitive` then the unique index will be instructed to ignore the case (useful for fields such as email). If the `preserveCase` flag is set to false, then all strings will be converted to lower case when being set. This also causes the get method to lower case any retrieved strings to allow this option to apply retroactively.

The `get` method simply returns the attribute name escaped to be a postgres name. If the `preserveCase` flag is set then it wraps this name in the `lower` function, causing the string to be converted to lower case.

The `set` function first applies the default value (either the one define in the migrations or the empty string) if it is a new record. Next it ensures the new value of the field is a string, throwing an error if it is not. Finally a validation is created, ensuring that the is present assuming that the required validation is active.

When `sorting`, values will always be sorted independent of their casing. This means that strings will actually be alphabetical as opposed to `A, C, b`.

8.10.3 Numbers

Number attributes can hold any numeric value. They are always backed by a single database column whose name matches the attributes. Numbers fields are nullable, this was necessary as, unlike strings, no logic default exists,

Numbers can accept a default value which will be used to initialize the column when a record is created. This initialization is done at the database level. They can also be marked as required. Doing this removes the nullable status of the column and will cause an error to be thrown if no value is set during creation (provided there is no default value).

Numbers by default are decimal numbers, stored as a `double precision`. This type was chosen as it is the same data type javascript uses for its number type. This means that all use-cases for websites should be covered by this type. Numbers accept an `integer` flag, in this case the column type will be

changed to be of type integer, this offers large improvements to operation speed.

When setting, numbers apply their default value and ensure their new value is a number (or integer if necessary). They then ensure the value is present if the required validation flag is true.

8.10.4 Booleans

Booleans are one of the simplest built in attribute types. They once again have a single column with a matching name and are stored using the `boolean` data type.

They are not nullable and will instead always default to `false`. This is done as it avoids the need to check if something is `false or null` when trying to perform a query. This makes more logical sense for development as you would expect that if you selected all values for which a boolean field was either true of false then you would receive all the records. With `null`, however, this is not the case.

Booleans accepts a flag which will cause them to default to `true` rather then `false`.

The `get` function simply returns SQL retrieving the column, meanwhile the `set` function simply casts the new value to a boolean and assigns it.

When `sorting` all like-values will be grouped together. this means all the `trues` when the direction is ascending all the trues will be last.

8.10.5 Dates

Date attributes have a single, not nullable, column of using the `timestamp` data type. This data type stores information about both the date and the time, mirroring that of javascript's built in date object.

Dates accepts the required validation, which marks the column as not nullable, and ensures a value is set upon creation.

They also accept a default value to be used as their initialization value. This value can be a string in the format of `2019-05-28T09:05:20.607Z` (this is the

ISO 8601 format and is the default value obtained from converting a javascript date object to json). The default can also take the form of `{ now: true }` - the operator function which returns the current date. This allows the date field to default to the current time, useful for storing a created at date for record. This default is enforced on a database level.

The `get` function for dates is trivial, simply retrieving the column directly. The `set` function accepts the same values as the attribute default - either a string in ISO format or an object with a single key of `now`.

8.10.6 Passwords

Password fields are designed to store a string in a secure manner. They create a single text column to store their data.

They implement the standard required validation and do not implement the `get` or `sort` functions.

When setting a new password the value is automatically salted and hashed. This encryption process uses functions provided by one of postgres' built in extensions, "pgcrypto". This extension includes methods for securely generating random salts, and hashing strings.

The encryption strategy chosen is a blowfish variant as this is the currently recommended strategy. This algorithm is automatically adaptive, meaning that as computing power increases the algorithm can tuned to make computing take longer. The resulting hash holds the information about the used algorithm, this means passwords hashed with different algorithms can co-exist (**???**)(https://www.postgresql.org/docs/8.3/pgcrypto.html).

8.10.7 Associations

Association attributes are one of Skeems must useful assets, their existence allows for extremely complicated operations to be as simple as any other attribute type. They store relationships between two arbitrary models.

Migration

The attributes require two pieces of configuration to function: the model to link to, and whether the connection leads to many records of just one.

Traditionally, if one side of a relationship only has a single record (one-to-one or many-to-one), then you would forgo the joining table and store the foreign keys in one of the tables directly. Skeem opts, however, to always have a joining table. This simplifies query generation as it is always identical, it also allows for associations to change to a many relationship without any changes to the database.

The first step taken, when creating an association, is to generate the information for the joining table. This involves three pieces of information:

- The **tableName** specifies the name of the joining table. This is initialized to be `{MODEL_NAME}_{FOREIGN_MODEL_NAME}__{ATTR_NAME}_assoc`. The length of this name is in attempts to prevent duplicate names being generated. After generating the name, it is checked for uniqueness in the database. If the name already exists then it is appended with the string `_2` and the check is performed again, repeating until unique.
- The **ownKey** column holds the id of the main record. It takes the value of `{MODEL_NAME}_id`.
- Finally, the **foreignKey** column holds the id of the record the association is leading to. Its value is initially `{FOREIGN_MODEL_NAME}_id`. If the association is recursive (e.g users have many users) then both the foreignKey and ownKey will be identical, this is resolved by appending `_2` to the foreignKey.

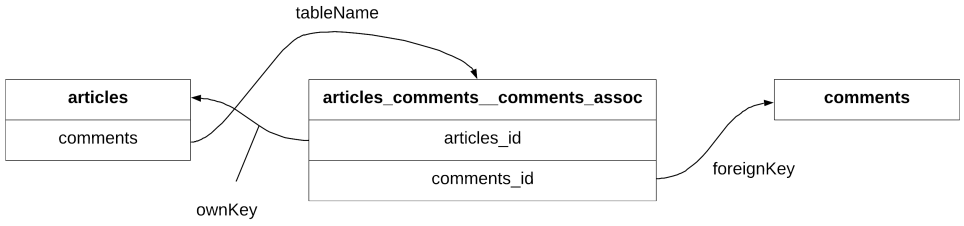


Figure 2: The tables and keys for an articles to comments association.

Associations allow for an `inverseOf` property to be specified in the configuration. This indicates to Skeem that the association already exists going in one direction and that this new association is the same relationship but backwards,

e.g articles to comments is the inverse of comments to articles. The value of the property will be the name of the attribute on the associated model that holds the information. If this property is given then the tableName, ownKey, and foreignKey is retrieved from the other association and the ownKey and foreignKey and flipped. This means the association will use the same joining table and thus share the same relationships.

These keys are stored along with the association model and the type of association in the attributes data.

The joining table also has three indexes applied to it. Each column gets a foreign key index which ensures their values exist within the other tables. A unique index for a combination of both columns is added to ensure you cannot associate the same two records twice. The joining table is then added to the `schema.db` object.

Getting

The `get` function for associations accepts a full fetch request as its request data. The fetch request is processed in the context of the associated model, almost identically to that of a top level fetch request. By having the association use the same processor as the top level helps ensure consistency through out requests.

The only exception is an additional where condition is added. This where condition only accepts records whose id is listed in the joining table.

Listing 24: The where condition responsible for limitting the records to only those associated.

```
query.where.push(`
  id IN (
    SELECT {attrSchema.data.foreignKey}
    FROM {attrSchema.data.tableName} innerAssoc
    WHERE
      innerAssoc.{(attrSchema.data.ownKey)}
      =
      {model.tableName}."id")
`)
```

Setting

When setting an association attribute, the value passed is an object function (the key being the name of a function, the value being its argument). There are multiple possible functions which can be called to manipulate the association, though they vary depending if the association is a many or a one.

- **Create** is used to create a new record. It accepts the same value as a top level mutate, which will be processed in the context of the associated model. This function also creates a joining row between this new record and the one being modified. In a has many association it will add a new relationship, whereas in a has one it will replace it.
- **Update**, likewise, will update an associated record. If the supplied id is not currently associated with the record then an error is thrown.
- **Destroy** will remove the association and delete the record.
- **Add** is only available for has many associations, it accepts the id of a record and will add it to the list of associated records. This is useful for joining two pre-existing records.
- **Set** is only available for has one associations and is equivalent to the `add` function.
- **Remove** accepts an id of an associated record and will remove the relationship, whilst preserving the record itself - acting as the opposite to `set` or `remove`.

8.10.8 Images

File attributes store large binary objects. This attribute does not create any columns in the database and exists only abstractly on the model.

These attributes have a single piece of configuration where by you can state if you wish to store multiple images for the single attribute. In this case all interactions are the same except for receiving and setting arrays of items.

When getting the images, a sub-query is formed to load the tokens and alt strings from the `skeem-files` table. If the attribute is set to deliver multiple images then the results are aggregated into an array.

Setting images involves passing an object function (same format as operators). If the key of this function is `add` or `set` (depending on the `multiple` flag) then an image is will be created and stored. The value of the function references the name of a file present in the request (see section 8.7.1). A callback is added to the databases before tasks (section 8.9.1) which makes a call to the default file providers store function (section 8.13).

8.10.9 Computed

`Computed` attributes allow for the creation of data which is computed based on other values. For instance, if a user model had a height attribute storing was a number representing centimeters, then you could create a computed attribute which could return the inches value.

This attribute stores a computation in the form of an operator tree. Its `get` function simply consists of parsing this tree. There is also no way, as of yet, to define what it means to set the value.

8.11 Compiling Operators

Operators define a way to convey calculations in a serializable JSON format. They are predominantly used in the scopes and permissions of models as well as the filter of fetch queries. Though they have started to spread to other aspects of the system, such as a fetches sort or a mutations setters.

An operator takes the form of a json object with a single key. This key is the name of the operator and the value is the operators value. This value can take any format and is left to the discretion of the particular function.

Listing 25: The format of an operator

```
1 { "operator": "value" }
```

Operators have a one-to-one correspondence with an operator function. These functions all accept a context object, a model to act on, and the value supplied in the object. They must return a SQL string and a data type that the SQL will produce, such as number or boolean. The SQL will be placed in a query where the from clause relates to the model passed to the operator.

Listing 26: The return of an operator function

```
1 { value: 'SQL STRING', type: 'boolean' }
```

Many operator functions accept other operator objects as their value. For this reason, the functions are also supplied with a function which can be used to compile an operator tree. By composing multiple operators complex trees of calculations can be created to perform a wide variety of actions. Skeem provides a number of built in operators designed to fulfill many use cases.

8.11.1 Comparison Operators

Comparison operators are a collection of functions which compare two values. For a value they accept an array which contains two elements, both of which must be operator objects. Both arguments are compiled to produce their sql strings and then concatenated with an appropriate boolean operation. The available comparisons are:

lt arg1 < arg2
lte arg1 <= arg2
gt arg1 > arg2
gte arg1 >= arg2
eq arg1 == arg2

If the compiled arguments are of the same type then the comparison operators will return a type of boolean. If, however, the types are different then an optimization is performed, where by the returned SQL will simply be the value of `false` with a return type to match. This can be done because if the types are different then it cannot be true; a string is not less than a number therefore `{lt: [string, number]}` is always false. Doing this can simplify the final SQL query as the database will not need to perform these comparisons.

The `eq` operator performs a slightly more complex function. Databases treat `NULL` values as distinct from all other values including other `NULL` values. Because of this the SQL produced by the `eq` operator actually takes the form of `a = b OR (a is NULL AND b is NULL)`. It is features such as this which help make skeem more user friendly as it removes the need to know this fact about `NULL` values for SQL.

8.11.2 Control Operators

In order to combine different operations skeem provides three control operators.

The `and` operations take an array of operation objects, compiles each of them, and then joins their SQL statements with the keywords `AND`. The function also analyzes the return types of their elements in order to perform an optimization. The `and` operator checks to see if *any* of the returns are explicitly false, if so then the operator will simply return the SQL of `false`. if, however, *all* of the returns are `true`, then the operator will return `true` (this is because `X and true = X`). If neither of these conditions are met then the operator will filter out all true values and combine the remaining values with `AND`.

The `or` operation acts in a similar manner, except joins its statements with the `OR` keyword. This operator also performs similar optimizations: if *all* are `false` then `false` is returned, if *any* are `true` then `true` is returned, else all `false` values are filtered out and the remaining joined together (`X OR FALSE = X`).

The `not` operator accepts a single operator object as its value, compiles it and then returns prefaces the SQL with the `NOT` keyword. This operator has the return type of `boolean` unless the compiled value has a type of `true` or `false`. In these cases the return type is set to be `false` or `true` respectively.

8.11.3 Leaf Operators

Skeem providers operators which can inject values into the computation. The operator objects can be thought of as a tree of operations, in this case these value injecting operations would be the leaves.

The `value` operator is the simplist of these operations, simply accepting any primitive as its value (string, number, or boolean) and injects it into the SQL. This value is also sanitized to prevent SQL injection.

The `attr` operator is probably the most useful operator skeem offers. It accepts the name of an attribute, and returns the SQL necessary for reading

the value. For most attributes it simply returns the name of the attribute as this matches the column name. In the case of, associations, however it returns a fully formed sub-query which returns all the associated record ids.

The `param` operator is used exclusively by operator objects defined within scopes. Params can act as arguments for scopes, allowing for scopes such as `newThanDate` where it accepts a date as an argument. The `param` operator looks at the request for a param who's name matches the value, then sanitizes the value then returns it as a value.

The `id` operator returns the id of the current record. This is needed as because `id` cannot be passed to the `attr` operator, as it is technically not an attribute.

The `now` operator returns the sql of `NOW()` allowing queries to be written which use the current date.

8.11.4 Association Operators

There are several operators dedicated to working with the association attribute type.

The `empty` operator accepts an operator object and then will returns a boolean indicating whether the result has any results.

The `anyIn` operator accepts an attribute name (which must be an has many association) and a operator object named `query`. This operator compiles the `query` in the context of the given attributes model. Then checks if there are any records which exist in both the query results and the association.

The `associationEquals` operator accepts an `attribute` property denoting a has many association attribute and an array of `ids`. The returned SQL checks if the associated record ids equals the array of supplied ids.

The `path` operator accepts an array of strings. This array denotes a path traversing a set of associations such as `article > author > address > city > name`. Each element therefore must be the name of an attribute which is a has one association for the model that the previous element lead to (the first must be in the context of the main model for the query). The last element, however, can be any attribute for the model.

8.11.5 Miscellaneous Operators

Skeem also provides a few operators which do not really fit within the other categories.

`like` is an operator which accepts two operator objects. Each one is compiled and then used as the left and right hand sides of the `ILIKE` keyword (case-insensitive like). Each side must be of type string when compiled.

The `ids` operator is accepts an array of strings, the SQL returned checks if the current records id is in the supplied array. This operator was one of the first to be implemented and is now marked for depreciation. Instead it is preferred to use an `or` operator containing multiple `eq` comparison of an `id` and a `value` operator. Although this may, potentially, be a greater number of operators, very rarely is `ids` used for more then a single id. This operator was implemented before the creation of the `id` operator so passing `{ ids: [1] }` was the only way to find a specific record. Removing this operator serves to remove the amount of things required to learn to understand Skeem and also eliminates redundant choice from the developer.

The `scope` operator lets you call a scope defined on the model. It accepts the name of the scope and an object containing the parameters to be passed to said scope. This operator simply loads the scope from the model and calls the compile function on the scopes query.

Finally the `query` operator accepts an entire fetch query as its value, complete with a new model. This is useful in combination with some of the association operators.

Comment: *TODO*

8.12 Sessions and Authentication

These methods of authentication are referred to generally as session providers (they provide methods for authentication sessions).

Session provides define a name, model, a type and some configuration dependant on the type selected.

The **name** is used purely to distinguish between different providers and allows for multiple authentication strategies of the same type. You may have to distinct user sets which are authenticated with different models e.g for a school system you may have one user set named teachers and another for pupils. The model defines where the session provider should look to find the necessary data to check against any credentials provided.

The **type** defines which session provider to use. Skeem comes with three built in providers: local, facebook, and google.

The local provide authenticates users by storing some identifying attribute and a password in the database itself. Then when an authentication request is made the database is queried for a record with the specified credentials. If a record is found then skeem authenticates the user as that record. The user attribute is most commonly an email address or a username. The password is stored securely using a secure hashing algorithm.

The Facebook and Google providers allow users to authenticate using these services via the familiar "login with XXX" buttons. These providers can specify a list of attributes to extract from the service such as name, email, image.

- Providers
  - What are they
- Local Provider
  - what it is
  - how it works
- Oauth Providers
  - built in ones
- once authenticated a session token is generated which is used to identify. this is encrypted with a secret key using the JSON web token standatd
- Permissions and their structure in the schema

8.12.1 Roles and Permissions

Roles consist of a name and a query. The name is a globally unique value identifying what the role is called, such as "admin". The query takes the form of an operator tree which is executed in the context of the model holding the user information. If the operator is truthy then the current user has that role.

Each model can define a set of permissions. Each permissions lists what role they relate to, what action the permission is for (fetching, updating, creating, etc.) and also an operator tree defining what records can be accessed by the role.

Processing permissions for a model starts by selecting all the relevant operator trees which pertain the current users roles. If *all* of the permission scope is explicitly false, then the query is aborted an an empty result set is returned. If *any* of the scopes are explicitly true then the permissions are skipped as the user has access to all the records. Finally if neither of these conditions are met then the false values are filtered out of the scopes (false or X = X) and the remaining values are combined with the `or` operator and appended to the queries filters.

8.13 File Management

Skeem defines a file as being an object with the following properties:

name a human supplied name, ideally descriptive such as "Home Banner"
filename the actual filename being uploaded e.g "homeBanner.png"
mime is the format of the file, this is a standard defined by IANA.
data is a buffer containing the binary data of the file.

Every file that gets stored by skeem consists of a row in the database and the file data.

8.13.1 File Data

Storing the file data is the job of the File handlers. A file handler is comprised of two functions, `store` and `retrieve`. In Skeems configuration object there is an option to specify which handler should be when storing new files.

The `store` function gets supplied a file in the form of a buffer object. It must store this data somewhere and return a reference string. The `retrieve` function well later get given the reference string, which is then used to locate the file and then return a stream of the data.

Skeem provides two built in file handlers:

- The **local** handler stores files in the local filesystem - ideal for development. It automatically generates a filename based on the given files name and the current timestamp. This filename is then checked for uniqueness in the filesystem and then used as the reference string.
- The **AWS** handler places files on in an AWS bucket for which the connection information is held in the configuration object. After uploading an image, Amazon returns a unique identifier for the given file which is used as the reference.

8.13.2 File Information

All the file information, including the name, filename, and mime type is stored within the `skeem-files` table in the database.

After a file handler has stored an image, the returned reference string is stored in the database along side the other file information as well as the name of the file handler responsible for the storage. By storing the file handlers name you are able to change what the default file handler whilst still maintaining the ability to retrieve previously uploaded images.

This row also contains a token field which is automatically securely generated to a random string. This token is used for retrieving a file and its information and is what is returned by the image attribute.

8.14 Retrieving a File

When retrieving a file, the generated token string must be supplied. This token is then used to find the relevant row within the `skeem-files` table. After this row is found, the reference string and the handlers name are extracted. The relevant handler then has its retrieve method called, passing it the reference string. This then returns a stream of the file data.

8.15 Plugins

Plugins allow skeem to cover a much wider use case then would likely be possible if I had to implement all edge cases manually. By forcing system features to use a plugin style system it helps to better define the edges of systems and to creating cleaner more targeted sub-systems. This helps to prevent the core of skeem to become bloated with extremely specific features (how).

> Note: There are examples of very specific features within the code base such as the operator `associationEquals` operation. It was additions such as this which prompted the need for the plugin system.

Every pluggable feature (attributes, session providers, operators and file handlers) all work off the same abstracted code and each one simply specifies a set of configuration. The abstracted code handles the file loading and parsing and exposes key functions to the specific implementations.

Each pluggable feature specifies the following:

name this is used when logging debug information.

builtinsFolder this specify the name of the folder where the built in features can be located.

folderName the folder name of where the external plugins will be located relative to the application root.

validateExports a function which will get passed a file when it has been imported and parsed. This function should return a boolean indicating whether the exported contents of the file is valid. For instance checking whether a file handler contains a store and retrieve method.

merge a function responsible for merging all the individual files together into one final object.

Abstracting the plugin system like this helped to enforce a consistent plugin style (which can aid developers when writing plugins), reduce duplicated complexity, and to isolate a critical system feature to be tested separately.

Listing 27: The code required to define the attribute plugin system

```
1 export const loadAttributes = createPluginLoader<
    IAttribute>({
```

```
  name: "attributes",
  builtinsFolder: "./builtins",
  folderName: CUSTOM_ATTRIBUTES_PATH,
  validateExports(_filename, exps) {
    // attributes should be an object with three
       keys: "migrate", "get", "set"

    if (!isObject(exps)) {
      return false
    }

    const expectedKeys = ["migrate", "get", "set"]
    const isValid = Object.keys(exps).every(key =>
       expectedKeys.includes(key))
    return isValid
  },
  merge(acc, filename, exps) {
    acc[filename] = exps
    return acc
  }
})
```

Loading a plugin executes the following algorithm:

1. combine the `builtinsFolder` variable and the directory the function was defined in, in order to find the full path for the built ins.
2. if this path doesnt exist, then skip steps 2-6.
3. load a list of all files in this directory.
4. for each javascript file require it
5. pass the contents to the validateExports method along with the file name. If this function returns false, throw an error
6. otherwise, store the contents along with the file name in an array.
7. combine the `folderName` variable and skeems root directory to gain a full path for where external elements are stored.
8. repeat steps 2-6 using this new path.
9. for each element in the loaded array, pass it to the merge function.
10. return the result of the merge.

When skeem is started it calls all the loaded functions created by the plugin system. These functions will then load all the built elements and any external

ones, merge them, and then store them with the manager. Then when a component wants to use one of the plugagble elements it references it through the manager.

8.16 CLI

- How its structured
- what commands are available
- The cli interacts with skeem by means of a text based command line.The was split into an actual definition of the interface and the parsing and processing of the command line text.
- Command Definition:
  - Commands are defined declaratively as an object containing: a name, some help text, a list of accepted options and arguments (each with a name, a type and an optional default) and a function to call when the command is executed. Skeem currently contains 14 commands.

8.16.1 Overseer

8.17 File Management

- How its built
  - react

8.18 Configuration

Many aspects of skeem work off of application wide configuration, items such as the database connection information, the preferred file provider, or the session secret key. This information does not make sense to exist (and some parts can't be) in the schema as they configure skeem as a whole and are not application specific. Many of these configuration options also change

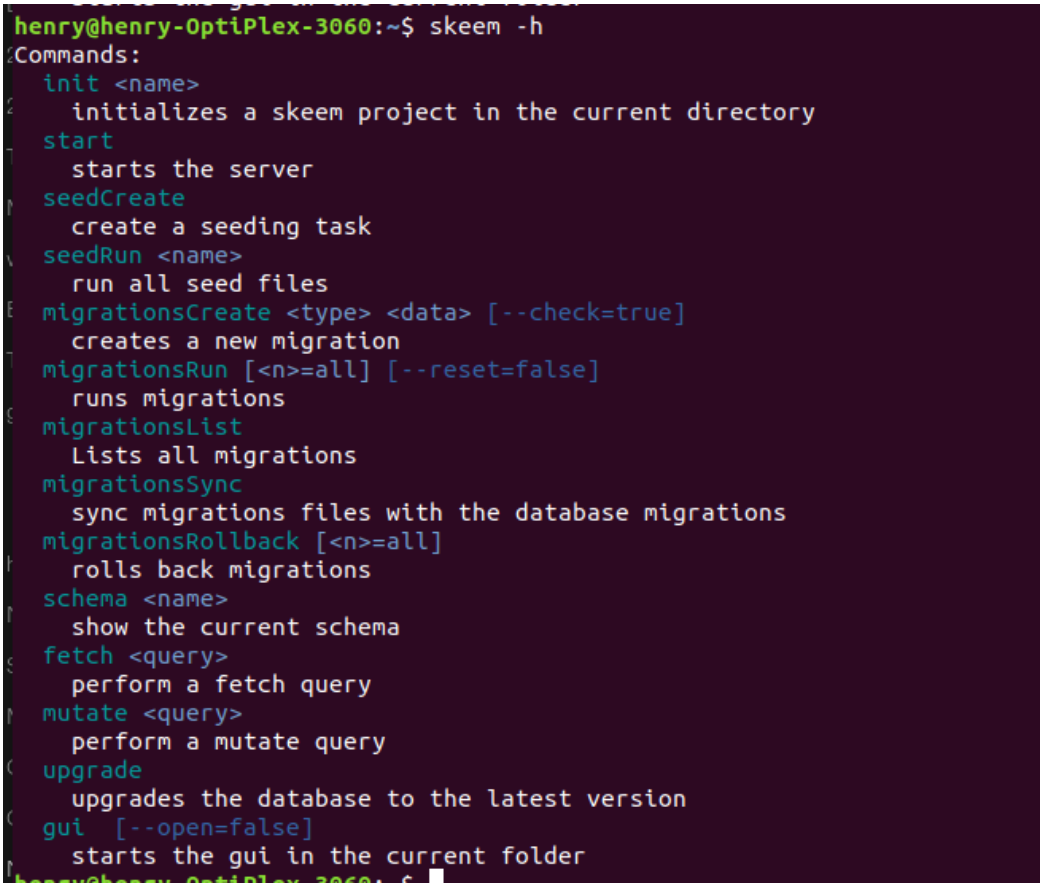


Figure 3: CLI Help screen

depending on the current environment. This configuration is stored in a JSON file named `skeem.json`

When running a command in the CLI, the system searches recursively upwards from the current working directory looking for this `skeem.json` file. If this file is not found before the root directory is hit, then an error is thrown stating that you are attempting to run a command when not inside a skeem application. When the file is found, its location is marked as the root directory for all relative paths.

8.18.1 Different Environments

Throughout a projects life time it will be run in multiple environments such as development or production. Each environment will likely want its own configuration. For instance when developing an application you would want to store files locally for speed and cost, whereas, in production you probable want to use a cloud storage solution such as AWS.

The configuration for different environments can be declared within the configuration file by nesting all the options within an object and making the key equal to the name of the environment. Skeem looks at the `NODE_ENV` environment variable in order to determine which configuration block should be loaded. `NODE_ENV` is a standard variable throughout the node ecosystem used to determine the current environment.

Listing 28: An example of a config with multiple environments.

```
{
  developments: {
    /* development config */
  },
  production: {
    /* production config */
  }
}
```

8.18.2 Environment Variables

There are many settings in skeem for which the value may not want to be hardcoded. This may be because the value is likely to change often, may be different for each developer, or because the value should be kept secret and so would want to avoid be committed into the applications repository. The solution to that is the use of environment variables. Environment variables provide run time configuration options to many programs and are used to solve this issue within skeem.

To use an environment variable you must set the value of the configuration item to be `{ $env: 'name of env variable'}`. Skeem searches all values of the configuration for objects in this form and substitutes them for the specified variable. If this variable does not exist an error is thrown.

To further facilitate the use of these variables skeem will automatically search for a file in the application root named ".env". This file should contain key value pairs. Upon starting, skeem will automatically load this file and merge the contained variables into the environment prior to passing the configuration.

Listing 29: Configuration which uses enviroment variables to avoid exposing critical infomation

```
{
  database: {
    host: { $env: "DATABASE_HOST" },
    username: { $env: "DATABASE_USERNAME" },
    password: { $env: "DATABASE_PASSWORD" },
  },
}
```

8.19 Documentation

- why it is important
  - many parts of skeem seem very complicated
  - lots of stuff can be hard to know exactly what to do
  - contains examples
- docsify

- built into the repo and automatically deploys

9 Testing

Tests are an essential part of any software project especially those providing some critical functionallity to users - Skeem is no exception.

Tests were written in Jest.

- Tests were written in jest.
- Tests targeted functionality rather then implementation. However tests were written for smaller parts of the system when functionality became too complex or when the underlying functions were critical - such as loading the schema from the database.
- For a complete list of all tests please see the apendix.
- CI
  - Due to skeem being used in production it was essential that it was not only tested but that testing was constantly carried out. By using CircleCI tests are automatically run when a change deployed to the git repository.
- Coverage
  - Code coverage reports show how many lines of the are touched by the tests this is very useful to ensure all the code branches are tested and perform as expected.
  - Code coverage ended up at 46% at the end of the project.
- Code quality
  - Codeclimate is a service which analyzes code and detects "code smells". "A code smell is any characteristic in the source code of a program that possibly indicates a deeper problem." - Wikipedia. This includes problems such as:
    * Cognitive complexity: how complicated is the code to understand.
    * File and function length: does the file or function contain too many lines (only counting actual lines of code, ie. not comments or blank lines)
    * Duplication: are large parts of the code duplicated in multiple places

  - Codeclimate then predicts the amount of time it would take to fix this technical debt. At the project end, skeem contained 147 code smells with a predicted clean up time of 2 months.

10 Deployments

Throughout the development i has the opportunity to deploy skeem on many real world applications. This chapter I shall discuss those deployments, how effective Skeem has been for each and any changes that were made to skeem because of them.

10.1 Resooma

- Resooma is a bills consolidation company focussing primarily on university students.
  - 88 distinct models, 700 attributes, 50'000'000 users

Resooma started development prior to the launch of skeem and so not all parts of the site utilize skeem.

10.2 Resooma Native

Resooma has multiple mobile applications. All of them use skeem as their primary means to collect data.

10.3 Enterprise Security Distributions Norwich

Quote generator and Stock management tool for Enterprise Security Distributions Norwich

- Tracks more than XXX quotes for customers concerning more than 20'000 products. Used heavily

10.4 Inbox Integration

- Invoice fraud detection using machine learning

10.5 Other Applications

- Voluble
  - Messages API as a service
- Rolecall
  - Job tracking and communication platform aimed at contract workers
- Fileboy

10.6 Conclusion

There are a wide range of projects using skeem. Including some mobile applications. Very flexible.

Comment: *need to include what ive had to learn for the project*

11 Conclusion

Skeem has turned out to be a very successful project already helping out a wide range of projects

11.1 Future Work

References

"Monorepo." 2019. *Wikipedia*. Wikimedia Foundation. https://en.wikipedia.org/wiki/Monorepo.