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 more advanced functionality whilst simultaneously attempting to be faster and more performant for the end users. Every non-trivial website is in some part of dynamic data, that is data is which not static. Dynamic data could be anything including blog posts, user details, editable content, user-tailored suggestions, product listings, delivery statuses. The report covers 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.

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 all the details of the request, what page are they requesting, what parameters have they specifier, and who is making the request. The server will then construct the page in full, complete with styles and content. This will 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 rendered.

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 navigates to a new page, this process is simply repeated and a new page is built afresh.

This pattern does have some disadvantages, however. on slower networks, the user is forced to wait after many interactions even when the upcoming page is very similar. Every request will see them reload many resources which they have previously seen.

Comment: *there may be two many problems listed, may should pick a couple*

This pattern has a few issues: 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.

A similar issue is that responses always contain the same markup and data independent on the device that is requesting it. For instance, on mobile the site may not display a sidebar which is otherwise present on desktop. This sending of unnecessary data once again causes the most issue 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

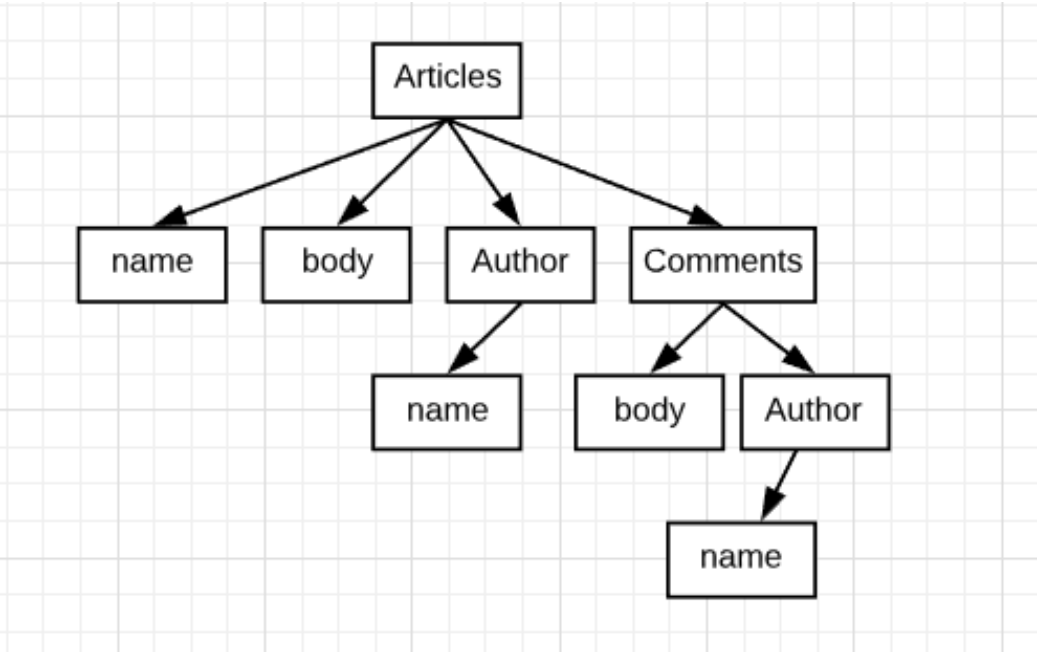
pages request very similar data, only differing in one pages need for the posted date. The options here are to either have two very similar end points which return near identical data, or forcing the end user to download the posted at information when they may never actually need to render it. The former 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.

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.

4.3 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

5 Requirements

5.1 High Level Requirements

- what would need to be achieved to solve all these issues
- Tree Structures
  - Fetch arbitrary tree structures
  - Query interface which is easily sanitizable such that it could be executed from even dangerous clients without risk of returning non-permitted data.
- Simple:
  - Explainable through limited, reasonably sized, help docs
  - Simple GUI interface
  - Requires 0 knowledge of database structures to use associations
  - Includes File management
  - User authentication
- Consistency
  - alert clients to changes in data

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 use by web developers. That is to say they are expected to be somewhat technical people. They 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*

Skeem attempts to replace the database, api, and 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 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.

6.5 Models

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

6.5.1 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.

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 numerically, either a “has one” or a “has many”.

6.5.2 Scopes

Scopes define subsets of data: published articles, popular products, banned users. They are built from many operator functions which will compiled 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

6.5.3 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.

Mutations also allow for changes made to be reversed.

6.6 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 a set of optional configuration to specify 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. In some cases you may also retrieve the total record count see the section Pagination.

Listing 2: A request for fetching published articles.

```
1 {
2   // starting from the articles
3   "articles": {
4     // 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" }
  }
}
```

6.6.1 Attributes

Attributes 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" }]
...
```

6.6.2 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.

6.6.3 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.

6.6.4 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 paginating the response will also 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`.

6.7 Mutating Data

Comment: *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`

6.8 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.

6.8.1 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.

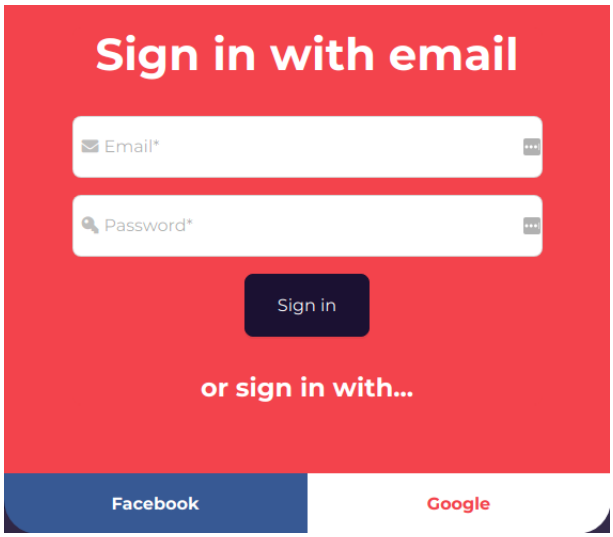


Figure 1: The log in screen for Resooma.com showing options to authenticate with email and password or by google or facebook

6.8.2 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.

By using the `session` operator it is possible to define permissions relating to the current user, for instance users may only access their own addresses.

Listing 8: A scope that only passes when the user record is equal to the current session.

```
1 { eq: [ { session: true }, { attr: 'user' } ] }
```

6.8.3 Roles

Often sites have a clear set of user groups such as admins, authors, sellers, or guests. Skeem allows to declare different roles along with how to determine if a particular user is part of that 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.

6.9 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.

6.10 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 the server to fetch the needed authentication urls and then redirect users to the relevant. It will then listen for a response and process the returned token. This means adding a full authentication flow to an application involves calling a single function.

6.11 Live Updates

6.12 Plugins

APIs are a 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 alot of built in operations which can achieve a large variety of results 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

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

Listing 9: 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 10: 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 11: 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:

```
1 {
2   filter: {
3     eq: [{ value: "a string" }, { value: 123 }]
4   }
5 }
```

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 12: 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 13: A simple implementation of the eq operaion.

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

```

```
function simpleEq({ compile, value }) {
  const left = compile(value[0])
  const right = compile(value[1])
  return { value: `${left.value} = ${right.value}`,
     type: "boolean" }
}
```

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 14: A full operation plugin file

```

module.exports = {
  isANumber: function({ value }) {
    if (typeof value === 'number') {
      return { value: true, type: 'boolean' }
    } else {
      return { value: false, type: 'boolean' }
    }
  }
}

// Usage:

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

7.1 Development Strategy

When a new feature would 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 15: Example of what the high level tests would assert (not actual tests)

```
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 Usage

9 Implementation

In this chapter I shall be explaining some of the intricacies of skeem, how it works, and why certain key decesicions have been made. This chapter will not cover the implementation of everything skeem offers but instead covers a the key aspects for which their were challanges and interesting solutions.

9.1 Technologies

9.1.1 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.

9.1.2 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.

9.1.3 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

9.2 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:

9.2.1 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

9.2.2 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.

9.2.3 Skeem-Gui

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

9.2.4 Skeem-Client

Provides functionallity for a client

9.2.5 Skeem-Common

Holds common functionallity needed between packages, such as error messages

9.2.6 Es-Qu-El

provides helper functions to generate and sanitize SQL statements

9.2.7 Typer

Typer standardies type checking accross the app.

9.2.8 Overseer

declarative CLI generator used to power Skeems CLI.

9.3 Database Usage

- Skeem wraps a Database
  - why? performance
- Built in tables
- Upgrading

9.4 The Schema

- what is it
  - models

- attributes
  - an attribute in the abstract simply comprises of a set of simple methods get, set, migrate.
    - these are then
  - simple types such as string, date, number, boolean
  - skeem offers more complex types
    - These types are where skeem really shines
    - association
    - file
      - see the chapter on file manager for more specifics as to how the files get processed and stored

9.5 Migrations

- Migrations
  - what do they do?
    - the manipulate the Schema
  - they dont manipulate the database directly because then they can be run in order to assert there are no errors

Migrations are the way the schema is mutated -they provide a simple and repeatable method for manipulation.

Each migration is stored in a separate file. This is done predominant to helps reduce conflicts if multiple developers create migrations at the same time. Each files name begins with a timestamp of when it was created this goes further to help reduce the possibility of conflicts but also ensures that migrations can be run in the order they were created.

Migrations do not directly effect the database instead they modify the schema, and then the differences between the old and the new are calculated and a set of transformation steps are generated. The reason for this split is so that migrations can be executed without affecting the database which allows any errors to be detected without having to....

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

9.5.1 Running Migrations

9.5.2 Database Diffing

After a new schema has been produced for an application, a list of change steps must be realised 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.

9.5.3 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.

9.5.4 Storing and Syncing

- migrations are stored in the database along with additional infomation such as whether they have been executed and what data they returned after being executed.
- They are stored in files to allow them to commited to be transfered between system to be run on a different machine
- also gets committed to version control software allowing multiple developers to be working on the same project

9.6 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*

9.7 Fetches

- they use the Schema
- build sql
- execute a single query
- what does the sql look like

9.7.1 Operators

9.8 Mutations

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

9.9 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

9.10 File Management

- What are file providers
- How do files get stored
- How are files set

9.11 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 `associationEquals` operation. It was, in fact, the addition of features 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 16: The code required to define the attribute plugin system

```
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.

9.12 Attributes

9.13 Operator

9.14 Session Provider

- just talk about them in the context of wanting to implement twitter

9.15 File Handlers

9.16 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.

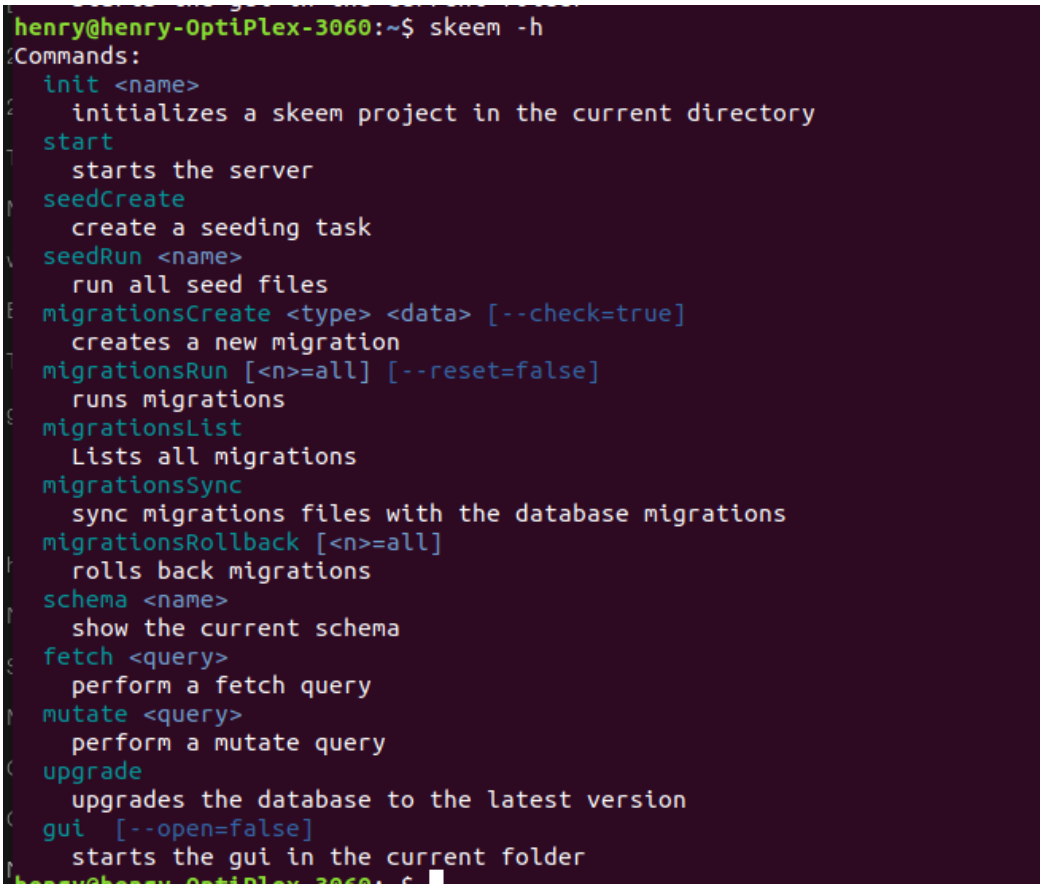


Figure 2: CLI Help screen

9.16.1 Overseer

9.17 File Management

- How its built
  - react

9.18 Configuration

- this is how skeem identifys the project root, and allows you to execute commands even when within subfolders

9.19 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

10 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.

11 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.

11.1 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.

11.2 Resooma Native

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

11.3 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

11.4 Inbox Integration

- Invoice fraud detection using machine learning

11.5 Other Applications

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

11.6 Conclusion

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

12 Conclusion

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

12.1 Future Work

References

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