CS Part D
18COD290
B526505

Skeem

A modernised database system

by

Henry Morgan

Supervisor: Dr. A. Trehan

Department of Computer Science
Loughborough University

April/June 2019

Abstract

Dynamic data is a ubiquitous feature amongst all modern websites which comes with its own set of challenges. Currently, many smaller subsystems must be pieced together to achieve even the basic needs of many websites, doing so, however, causes other issues, such as the much bespoke knowledge. Skeem is a system which attempts to drastically simplify the process of dealing with dynamic data while being intuitive for developers to achieve otherwise complex tasks. It achieves this by replacing the combining the various subsystems under a unified API. Skeem has been deployed into a variety of applications spanning a wide range of use cases and has proved to be a powerful and flexible tool.

Contents

1 Introduction

Websites are becoming more sophisticated and offering increasingly advanced functionality while at the same time, expected to be more performant for end-users. Every non-trivial website is, in some part, makes use of dynamic data, that is data which can change throughout an applications lifetime. Dynamic data 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 for how dynamic data is handled traditionally and more recently. It will also cover the associated issues that arise with these approaches and how others have attempted to solve the issues. Next, the report shall discuss how a system that aims to solve these issues was created, including the process of building the system and how it operates. Finally, the report shall discuss how the system was tested, including the opportunities of deploying the system in re-world environments.

1.1 Motivation

I have been a web developer for several years, being directly involved with the creation of many web-based applications covering a wide variety of scopes. Over this time, the method taken to create 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 websites currently handle dynamic data on web applications. 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, each having a dedicated end-point, template, and 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 relatively straightforward; the server is aware of all the assets that make up the page and therefore knows precisely what data it is needed. The server has direct access to the database, which allows it to query for all the data needed. When the user navigates to a new page, this process is repeated, building a new page from scratch.

This pattern has some disadvantages. 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. When navigating to a different article, the user still has to download all of these assets. This problem is most significant on devices with slow connection speeds such as mobiles.

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

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

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, the javascript will intercept the request and construct the new view and render this instead. Since the javascript is in full control of rendering, it has the option to look at the current device specifications and render precisely the needed content - it does not need to render elements which will not be displayed. Additionally, this method means that when a user navigates, they can get instant feedback; whether it is the new content or a loading screen. Either way, the result will be a more responsive interaction.

SPAs, however, provide a new challenge - how does the dynamic content of the page get retrieved. Previously the server knew what page the user requested and could access the database directly to retrieve necessary data. The solution to this problem is with the introduction of a new sub-system into the website's architecture: the API.

The API is similar to the traditional website's server; it accepts requests and performs database queries., However, instead of returning HTML it instead 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 the problems that exist which require solving.

3.1 API Design

APIs tend not to be broken up by resource, not by page. In other words, they would tend to have an endpoint for retrieving blogs and another for retrieving comments as opposed to a single one for handling the "blogs page". This divide is done to allow for requesting only the necessary data. If for some reason, the javascript wants to render an article without its comments, then it should not be forced to receive download the comment data. Good API

design is a tricky task, and many issues present themselves when creating one.

3.1.1 Code Repetition

Consistency across an API means that two endpoints behave in a predictably similar manner. This is an excellent advantage as it means a developer can apply the knowledge they learn from using one endpoint, to all other endpoints on the system. If they can logically predict new functionality, then the API is also very discoverable.

Consistent and discoverable APIs tend to lead to very repetitive code. Given an endpoint fetching blogs and another for products, then what is different about these routes? The table name in the SQL query and the columns it returns. The formatting of the request and the response is likely (or should be for consistency) the same, which can lead to duplicated of code - something that is almost always undesirable (Spinellis 2016).

3.1.2 Data Duplication

Imagine a blog. On the site, there is a list of teasers, each displaying the post title, the author's name and some preview text. There is also a separate page for reading a post, which contains the title, body, author, and the creation date. Here, both pages require very similar data, only differing in the creation date.

When creating an API for this website, there are two options available: either have two very similar endpoints which return near identical data or force the end user to download the additional information when they may never actually view it. The former leads to having to maintain two distinct APIs, whereas the latter is wasteful. Neither option is desirable; a trade-off is required.

3.2 Storage vs Display

Databases should store normalises data which, directly speaking, this means storing data in flat tables, i.e. one for articles, one for authors, one for comments and then storing additional information to relate different records. Storing data in the fashion is desirable as it helps to remove data duplication, which in turn, makes updating records easier and limits the risk of desyncing data by only updating it in a single location.

This structure is fine for storing data; however, the issue is that applications do not display data in this format. The end user is not presented with a page containing merely an article's text and then a separate page holding the author's name and a further page for a list of comments. Instead, there is a single page containing all the data amalgamated in an easily digestible and pleasant format. The data the user sees can be thought of as a tree of data: the root being the article itself and then containing connected nodes for each comment each having further nodes containing their authors.

Requiring a tree of data from a database is a widespread and useful thing; however, despite being conceptually simple, it can get incredibly complex even when having to traverse only a few levels deep.

3.3 API and View Separation

The SPA architecture has a sharp 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 quite possibly exist 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. Having a properties table likely means there is a properties API to access the data. Changing the name of a database column would result in an update the API. If the API is left unmatched, then, interactions become more difficult to reasons about, and more of the system must be thought about at a single time to complete particular tasks. Is it "name" or "title", "body" or "description",

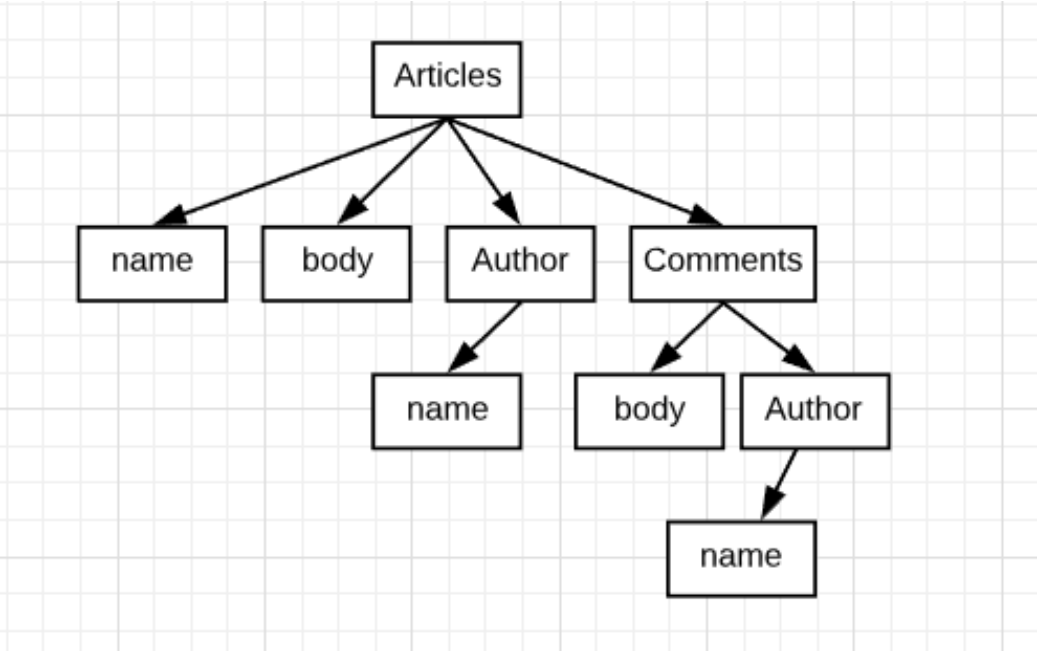

Figure 1: A simple tree structure of data related to an article.

```
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"
```

Figure 2: The SQL required to fetch a tree of data.

"createdAt" of "created_at"? This problem only worsens as the system grows and time goes on.

There is a similar coupling 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.

Tight coupling and removed implementations create the need for constant synchronisation or else will inevitably lead to bugs.

3.3.2 Cognitive Complexity

Defining the data in one location with its use being in another can make it very difficult to discern what is going on and what data is available. It can make it very easy for mistakes regarding the data's structure to occur.

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 specific effects happen.

This additional cognitive load can easily make simple tasks appear complicated, often slowing development. Additionally, It can create 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: understanding data flows which span many domains leads to the requirement of a high level of underlying knowledge. Being is especially true if the domains span languages.

For instance, given a typical PHP server, when adding a new feature it is likely you would need to understand: 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 for so much knowledge about so many domains raises the base level of skill needed to perform many tasks, which makes it challenging to introduce new people to the system. You could have people specialise in particular

fields, but this brings rise to further issues by then requiring attention from multiple team members to perform sometimes conceptually simple tasks.

3.4 Boilerplate

Another issue with the existing methods is the need for a lot of boilerplate to be set up before a project before making progress with the actual application. You have to set up a system to manage database connections, handle database migrations, seeding data, API routing, authentication, file management, code compilation, etc. Although some of these systems may already be abstracted and so involve minimal amounts of setup and configuration, their configuration is still present within a project. This bloat has a few significant drawbacks.

Firstly, it can be overwhelming for new entrants to the system to discover what they can and cannot change. 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 boilerplate also makes it more challenging when starting a new project. Either an existing application is duplicated and unwanted features removed. Or it is started from scratch and parts are added all the parts until functional. The first can lead to the unnecessary code remaining, and the latter can take time to implement. Both, however, require knowledge of how the systems work (leading back to the issue of bespoke knowledge) and delay the start of the actual task.

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 an API and is actually present on all applications with a user system.

This task is abstractly simple but can be very easy to make a mistake when implementing, and the cost of making such a mistake could be extremely costly.

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 tend to require a very different request format than that of a typical request - usually taking the form of, what is known as, a multipart request. A multipart request combines multiple types of data into a single request, for instance, an image and a JSON object (Borenstein and Freed 1992).

Creating a consistent APIs structure through endpoints which except this style of request can be difficult. There are two real solutions: forgo consistency and discoverability and accept a bespoke endpoint or allow all requests to be multipart requests even when only JSON is required. Multipart requests have the downside of being relatively user-hostile to read, and browsers tend to provide less capable developer tools for viewing them as they are expected to contain binary data, this makes the latter solution undesirable. Consistency and discoverability, as mentioned, can be valuable traits of an API, so their sacrifice is unideal. Neither solution is ideal, so a trade-off must be made.

4 Literature Review

This chapter shall discuss existing technologies which attempt to solve the outlined problems, including an overview as to how each system functions along with their advantages and limitations.

4.1 Graphql

GraphQL is a system which takes a declaration of the data within an application and in return, produces an API to make requests for the data (GraphQL 2019).

The API allows for defining and requesting logical clusters of data, even if stored across separate tables. Then queries can be constructed to retrieve multiple chunks of data at the same time through defined relationships helping reduce the “storage vs display”.

Front-end code constructs these queries, allowing the coupling between the APIs definition and the view layer to be lessened. It also eliminates the API design problems as the API structure is the responsibility of the library.

A disadvantage of GraphQL is its need for the definition of the data structure. This definition can add a lot of boilerplate to 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 work.

GraphQL also does not dictate the structure of the database, which has both advantages and disadvantages. On the one hand, it means the system is flexible and can be applied retroactively to projects, but on the other, it means a database system still needs to be set up and maintained independently.

4.2 Mongodb

MongoDB is a NoSQL database system using a document-based data model (MongoDB 2019). Unlike traditional databases which organize data in fixed tables with association records to join related data. MongoDB, on the other hand, stores data in arbitrarily nested shapes. Meaning data is stored in a way that more closely aligns itself with how it is presented to the end user, eliminating the "storage vs display" problem.

The ability for storing arbitrary shapes allows for rapid prototyping; however, having no rigid schema does have some disadvantages. Over the life span of an application, it is inevitable that more fields will be needed. Having no strict schema means new records can store these new fields without change, this means, however, there is no guarantee to the fields a record has and is entirely dependent on on the data available at the time of creation. Whereas in a relational database, when creating a new column, all other records must be made to be valid within the new constraints, e.g. adding a column which cant be null, then a default must be set for all prior records.

4.3 Auth0

Auth0 is an external authentication system, which moves the logic needed to authenticate users outside of the application (Auth0 2019). An API is

available which provides commands to registers and authenticate users. They also handle necessary user system tasks such as password resets, offering either email confirmation or text message confirmation.

Auth0 solves the authentication system by moving authentication outside of the application. Doing this ensures the use of strategies which are well-tested and reliable.

It does, however, further add to the level of knowledge required to understand the system. There is also boilerplate needed for the application to connect to Auth0, which is likely identical between apps.

4.4 Gap Analysis

The approach demonstrated GraphQL of allowing queries to be formed and sent directly from the client-side solves many of the outlined coupling issues in an extremely effective manner. GraphQL, with its opinionated API approach, eliminates the issues associated with API design.

Auth0's approach to prescribing what a user is and how they are authenticated adequately eliminates the authentication concerns.

Many systems exist which each solve some combination of the defined problems, and a combination of multiple technologies could solve many of the issues. However, the need to combine multiple technologies, each having independent standards and practises, only serves to exacerbate issues concerning boilerplate and training needs.

5 Requirements

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

5.1 High-Level Requirements

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 logical step.

Many websites have sub-systems which have not been discussed or at least some need to perform custom actions which can only execute securely on a server. Therefore, having the system capable of being used from both a client and a server is a necessity as not to 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 reasonably limited, and so unlike GraphQL, the system could be very opinionated as to the structure and contents of the database. This viewpoint 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, it was essential to create a system fulfilled all database use-cases in order to replace existing systems rather than adding a further system to maintain.

In order to achieve all database requirements, a specification of an actual use case needed gathering. To do this, I obtained access to the database of an in-production application. The application this database was backing had

20'000 registered users and 4'000 unique daily visitors, with 4% of them being new to the site entirely. I, therefore, deemed this database to be substantial enough to cover many use-cases likely to be mat by a website.

I went through the complete source code of this application and collated a list of all interactions with the database. A minimal viable feature set was obtained using this information. Many interactions have not listed as they are relatively trivial examples, such as the ability to store numbers and strings. The following list includes only notable interactions:

- attributes
  - has many records through another association
  - has many based on a condition
  - set a default for an attribute
  - set default values derived from another column
- Validations
  - presence
  - uniqueness
  - inclusion
  - number greater than
  - uniqueness in scope of attr: value
  - validate uniqueness in scope with condition unless attribute: value
  - Validates only when creating
  - validates only if a condition is met
- Callbacks
  - After creating a record
    - perform an update of an association record
    - send an email
    - update own attributes
- Scopes to find records matching the following criteria
  - where attribute equals a set value
  - where attribute does not equals a set value
  - 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)
- Sort
  - Sort records by given attributes

  - vary the direction of sorting, either ascending or descending
  - sort by the number of records in an association
- Permissions
  - limit access based on an associations value
  - limit access based on an attributes value

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 its usages. 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 compiles the database, API, authentication, and file hosting subsystems into a single system under a unified interface.

Skeem is a very opinionated system; it makes assumptions about how data will want to be stored, how users should authenticate, what operations will want performing. These decisions closely follow current best practices and how, most typically, websites use data.

Developers make a declaration of what data they want to store. The system will then automatically create a database tailored to this specific information needed, including optimizations, such as adding indexes, to selected columns to allow for efficient retrieval of data.

Skeem provides a bespoke query interface specifically designed to fulfil the everyday needs of web applications, including:

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

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

The system can be fully configured and maintained through human-readable interfaces; there is no requirement to write any code. These interfaces can provide instant feedback for any errors that occur and can also provide helpful information aiding in Skeems usage. These interfaces contribute to solving many of the issues concerning training. There are, of course, still intricacies with using the various sub-systems which will require additional help; however, Skeem handles this by providing a full set of documentation detailing many system details.

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 are not required to have any knowledge of how databases work or function, nor do they need to know SQL to any level.

6.3 Value Proposition

Having decisions, concerning database structure, enforced can eliminate much boilerplate from websites as well as reducing the amount of knowledge needed to set up an application. Enforcing how files are stored means no setup is required.

Skeem controls both the structure of the database and the API creating minimal repetition throughout the application. By simply declaring a new attribute will see its availability propagate through all interfaces.

By allowing queries from front-end code, it helps to reduce the problems of cognitive complexity as it is easy to see what data is available at the same time as seeing where it is used. It allows developers to request the exact data that is required; this then eliminates the problems associated with API design.

By including authentication and file management systems within Skeem, they can follow a consistent interface, helping to not only eliminate the need for training and improving discoverability.

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 as well as used to formulate a query interface.

6.5 Models

Models define the actual data to be stored; they are similar in purpose to tables in a database. They also define a set of permissions determining who can access given records.

6.5.1 Attributes

If models are the tables in a database, then attributes are the columns. They each have an identifying name and define a specific data type to store.

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 a set of configuration options to tailor how it functions. For instance, booleans can set their default value, numbers can be declared to be an integer, or strings may enforce uniqueness. This configuration may also be related to validation. For example, strings can declare that they must be present, and numbers can specify a minimum and a maximum value.

One key feature of Skeem is its ability to request trees of data; this is the purpose of the association attribute. Association attributes define relationships between records.

6.5.2 Scopes

Scopes define subsets of data like published articles, popular products, banned users. They are constructed from a series of discrete operations which ultimately produce a single boolean value.

There are many built-in operators which provide range from straightforward processes such as `eq` for equivalence or `lt` for less than, up to complex ones such as `attr` which will query for the value of a specified attribute.

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" }
    ]
  }
}
```

There are many operations provided by Skeem, which can be composed to create a large variety of filters covering many use cases. The following is a list of all built-in operators:

- 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

Migrations perform mutations of the schema, defining specific changes to be carried out, such as creating a model or renaming an attribute.

Migrations present a way to quickly make changes to the schema in a repeatable and reversible fashion. This is essential when building an application, as there are usually separate environments for development and production. Migrations, act as a record of changes, by performing the migrations again it allows for reconstructing a schema.

When executing a migration, it may store some additional data about what was changed. This data allows for the reversal of the execution.

6.6 Fetching Data

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

Fetch requests use a declarative format, meaning the request states what data is needed and the system will calculate the necessary steps to achieve this goal. This approach is opposed to traditional SQL approach has developers write *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 precisely the needed data. The query includes: what records to retrieve, the attributes needed for each record, and how to order the records.

A fetch response will always be an array of records, where each record will contain its id as well as the additional attributes 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" }
  }
}
```

6.6.1 Attributes

The attributes part of the query specifies what data to receive for each record. Attributes, take the form of an array where each element specifies a separate attribute. If no attributes are specified, then each record will contain their id.

When requesting an association attribute, an additional fetch query can be formulated and nested within. This query includes a separate definition of attributes, filters, and sorting. Skeem will then process this configuration in the context of the associations model whilst limiting the results to only associated records.

The attributes of the associated attribute can further contain more associations. By nesting these queries, a tree of data can be generated.

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

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

In certain circumstances, attributes may be requested under a different name. This could be useful, for instance, if the same association attribute with two different filters is required. To alias an attribute, an `as` property alongside the attribute's 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. Filters are built up of a collection of operators in an identical fashion to model scopes.

There are a wide variety of operators provided which can be combined to achieve complex calculations. There exist operators to compare two values: `eq`, `lt`, `lte`, `gt`, and `gte`. There are also operators which inject values into the query, such as `attr` which places the current value of the named attribute and `value` which injects the supplied value after sanitizing it to prevent SQL injection.

Listing 5: A filter to return only records where the "published" attribute is true.

```
...
"filter": {
  "eq": [{ "attr": "published"}, {"value": true }]
}
...
```

6.6.3 Sort

Sorting data is a ubiquitous and essential capability of data retrieval: most recent tweets, newest article, oldest person. When sorting data, a subject and a direction must be specified. The subject is the name of an attribute and defines what value to look at when sorting. Meanwhile, the direction specifies the order to position the items, 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"
    }
  }
}
```

An array can also be used to specify the sorting criteria. Doing this will sort the data initially by the first item, then resolve conflicts with the next item in the list, and so on.

6.6.4 Pagination

The pagination options allow for the splitting of a request into multiple discrete pages. Paginating is a common practice throughout the web as having the user download thousands of records would lead to slow and undesirable user experiences.

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 that would have been received without pagination. The record count is a useful property as it allows for showing users controls to navigate between pages.

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

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

6.7 Mutating Data

Mutation requests allow for records to be created, updated, and removed. Mutations, like fetches, take the form of an object with a single key specifying a root model. The value then specifies the changes to apply. These changes take the form of an object function, additionally an array of changes can be passed, in which case they will be processed in order. The possible change keys are `create`, `update` or `destroy`.

For `create` and `update` requests, the value is an object where the keys are names of attributes and the values are the values for the attributes. The `update` function must receive an ID passed in with the attributes or else an error thrown.

The `destroy` method accepts a single ID as its value and will delete the record with the corrosponding id.

Listing 8: A mutation request with multiple changes to the articles model.

```
{
  "articles": [
    {
      "create": {
        "name": "A new article"
      }
    },
    {
      "update": {
        "id": "abc123",
```

```
        "name": "A new article"
      }
    }
    {
      "destroy": "abc123"
    }
  ]
}
```

When mutating associations then a similar change object is passed as the value. These changes are applied while respecting the associations i.e when creating a record it will automatically be associated. The available changes are:

- create
- update
- destroy
- add / set
- remove

The `add` function is only available for `has-many` associations and will add a relationship between two existing records. Its value is an ID of a record. The function set is similar `set` but is specific to has-one associations and will remove any previously related records.

The `remove` function will remove the relationship while leaving the records intact and accepts an ID for its value.

Listing 9: A mutation request creating an article and an associated comment

```
{
  "articles": {
    "create": {
      "comments": {
        "create": { "body": "a comment" }
      }
    }
  }
}
```

6.8 Permissions and Authentication

The client should always be assumed to be malicious, and as queries can be sent directly from the client, it is imperative to be able to ensure their identity and limit their access accordingly.

6.8.1 Authentication

Before being able to limit user access, the ability to determine who is making the request is required. Skeem provides a couple of ways in which to authenticate someone. The first is with a stored user identifier such as an email address along with a password. The second method is by using an oauth2.0 provider, such as Facebook or Google.

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

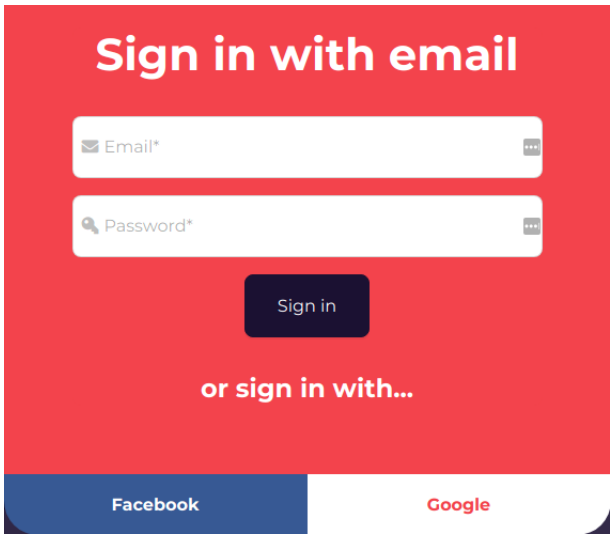

Figure 3: The login screen for Resooma.com showing options to authenticate with email and password or by google or facebook

6.8.2 Permissions

With authentication, there are now two distinct user states: authenticated and anonymous. Every model allows for the definition of a filter for each role. These filters are applied to every request, meaning it is possible to declare that while authenticated people can access all articles, anonymous people may only view public ones.

6.8.3 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.

The roles, like anonymous and authenticated, can have filters defined on models and will apply themselves to requests in the same fashion.

6.9 Management

The primary method of managing a Skeem application is through a command line interface. The CLI provides many functions which can be used to manage all aspects of a Skeem application.

The CLI includes the command `init`. This command gets supplied a name of an application and will generate all the necessary files and folders needed to create a Skeem application.

Migrations can be created with the `migrationsCreate` and executed by running `migrationsRun`.

The CLI also provides method for viewing information about the current application. The `schema` command will print to standard out the entirety of the current applications schema. This command also can be passed a model name, in which case just the schema related to the model will be printed.

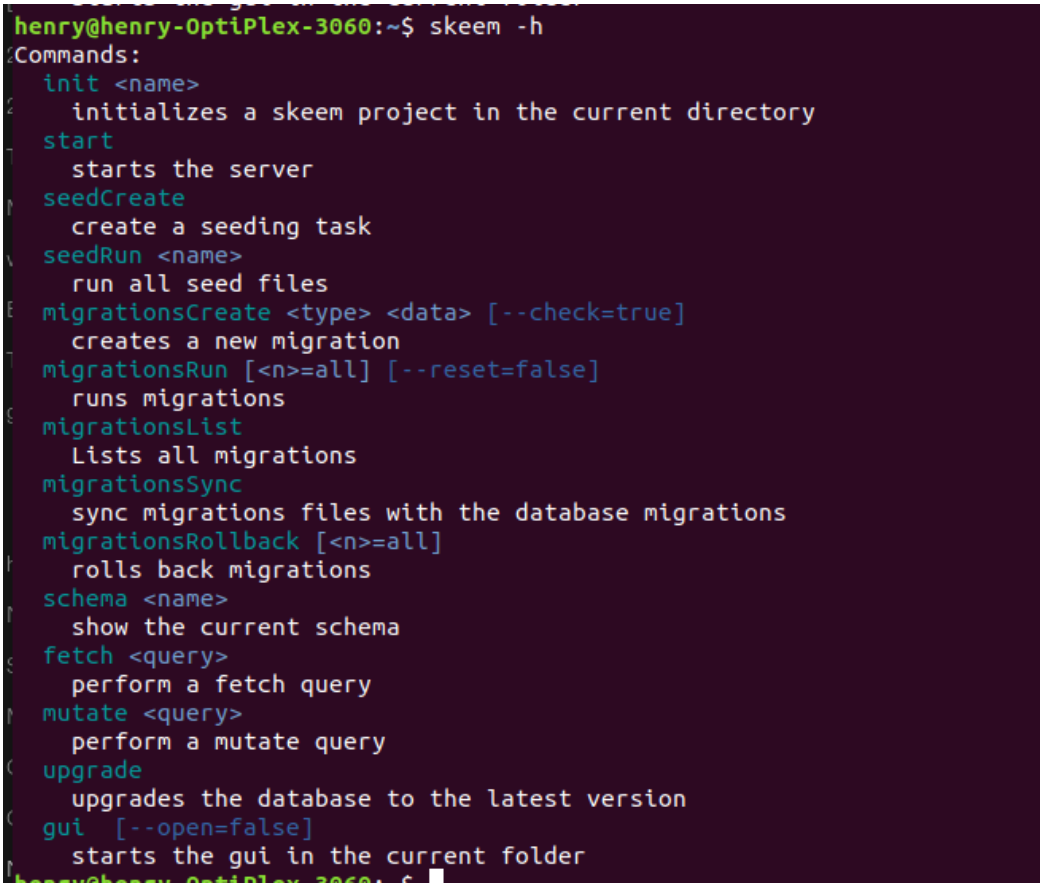

Figure 4: CLI Help screen

6.10 The Client

The client is an interface into a Skeem application designed for use in client-side code. The client contains two distinct, but interrelated sub-systems.

6.10.1 HTTP Server

The first sub-system is an HTTP server which listens for incoming requests, executes them within Skeem and then returns the result. All requests to this server use the POST HTTP method, and all contain a JSON payload.

The JSON payload must be of the format `{ type, payload}` where type is one of: “fetch”, “mutate”, “login”, “logout”, or “me”. The payload contains the query for the given type.

Listing 10: A HTTP request to fetch articles

```
POST / HTTP/1.1
Content-Type: application/json

"{
  "type": "fetch",
  "payload": {
    "articles": {}
  }
}"
```

6.10.2 Client Library

The second client-subsystem is a Javascript library containing functions for interacting with the HTTP server. These functions essentially wrap the possible request types in order to provide a similar interface to make requests. These functions provide all provide simple type checking on requests to ensure that the format is correct. In some cases, the response will also be modified to ensure consistency.

There are three different variations of the library each specialized for different environments.

Dumb Client

The most basic of the variants' is called the "dumb" client. This client provides the most basic wrappers around the different requests. This variant is designed to be the most minimalistic implementation possible and therefore does not make any assumptions about what technologies are available. It includes functions to `fetch`, `mutate`, `login`, `logout` and `me` but at the same time must be told how to perform an HTTP request. This client is not designed to be directly used and is instead used by the other variants to prevent them from having to re-implement basic functionality.

Web Client

The "web" client is designed for use on websites. It provides access to the same basic methods defined in the dumb client with some additions to make using it as straightforward as possible.

The first difference is that since it is presumed to be running on the web, it can use the "fetch" API to make HTTP requests.

Another addition is that it assumes it can store information (by use of localStorage) and therefore, automatically stores and retrieves the session token. When invoking the `login` method, the client will capture the session token and send it on subsequent requests. Then upon reloading, the client will check for the existence of the token and then send it upon all future requests.

This client also presents a method to facilitate OAuth authentication, known as the "OAuth flow". This function will perform all the necessary requests to fetch the security tokens before redirecting the user to the appropriate third party portal. A response will be automatically processed, and the user authenticated. This flow function means adding a full authentication procedure to an application involves invoking a single function.

Server Client

The server client is designed to run in a NodeJS environment. It, therefore, assumes the ability to import a module capable of performing HTTP requests.

This client is cautious not to cache any data as an instance of this client could exist over a very long period, and requests made to it could be from many different sources. Therefore, if it were to cache, for instance, the session token, then some requests could be made with incorrect authentication information.

6.11 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 can augment functionality by way of plugins.

Skeem exposes four of the critical systems to the plugin system: attributes, session providers, operators and file handlers. All the built-in components for each of these categories, use the plugin system itself, i.e. they use no internal access to operate — the built-in's ability to use the plugin system demonstrates the options which are available for external plugins.

6.11.1 Custom Attributes

There are many different types of datum which may want to use which do not fit within the bounds of the built-in types. By writing custom attributes, there is the option to work with custom data types.

6.11.2 Custom Operation Functions

Web applications filter data in a myriad of different and obscure ways. While 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 functions and have them integrate seamlessly with those provided.

6.11.3 Custom Session Providers

There are many other ways in which users may want to authenticate. Skeem, therefore, allows for the creation of additional OAuth providers. By creating new providers, users could log in with services such as Github and Twitter.

6.11.4 Custom File Providers

Files are stored in a magnitude of different ways and although storing on the local file system and AWS may cover most scenarios, files stored on an FTP server, for instance, is not an impossibility. As functions can be written to store and retrieve a file, then Skeem can use it to handle files.

6.12 Documentation

Skeem provides a set of documentation providing guides and examples of how different parts of the system works. This manual is essential to allow other developers to use the system independently, mainly because many of the features use bespoke terminology and practises.

The documentation helps to limit the issue of bespoke knowledge, as described in the problem definition. It provides a single source for developers to find answers and methods for completing particular tasks.

7 Methodology

Skeem's use in several projects throughout development (see section 10) meant that there were many times where features were needed, yet Skeem did not offer. This need for fast development meant releases needed to be frequent and priorities dynamic. Therefore, an agile approach was the only realistic way to deliver a product of use.

Development occurred in week long sprints. Over a week, multiple feature requests were received each having an urgency rating. If a feature were urgent, it would be accelerated to be part of the active sprint (sometimes to the

detriment of other items if velocity did not allow for the extra work). At the end of the week, Skeem would release a new version and an outline for the next sprint created.

Development undertook a test-driven development approach in order to develop the system in a controlled and effective manner. This approach helped facilitate the quick release cycle that was desired by helping to prevent feature regression. Section 9 discusses the specifics of writing tests.

8 Implementation

This chapter will cover the implementation details of Skeem, including some reasoning behind the making of critical decisions.

8.1 Technologies

Skeem takes advantage of many existing technologies in order to provide its functionality. This section acts as an overview of the key technologies used.

8.1.1 Typescript

Due to Skeem having to provide the functionality to a website, using javascript was a must.

Typescript is a superset of javascript which adds typing capabilities to the ordinarily dynamic language (Microsoft 2019). This typing information allows compile-time code checking, which greatly assists in improving the reliability of code; helping to ensure against common trivial bugs. Typescript is transpiled into standard, es5 compliant, javascript meaning it can run on all modern browsers. This compatibility is essential because if Skeem only supported the latest version of Google Chrome, then it would eliminate the possibility of using the system on any website having compatibility requirements.

Typescript also aids other developers using the system as the typing information is used in most modern IDEs to supply IntelliSence information

allow features such as auto-completion, inline errors messages, and hints for expected arguments and returns.

8.1.2 Nodejs

NodeJs is a javascript runtime designed to build scalable network application (NodeJS 2019). NodeJs was a logical choice as it allowed writing server code also in javascript, which allowed for the construction of consistent interfaces. It is also much easier to develop a system when writing in a single language as less mental energy exerted to convert from one environment to another.

8.1.3 Postgres

Postgres is a object-relational database system (Group 2019). 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 ability 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 optimiser which can automatically transform subqueries into join statements. This optimisation is taken advantage of in Skeem as there is not a single joining statement throughout the codebase. The chapter on fetch query SQL generation covers reasons for this further.

Postgres also provides the ability to write custom database functions relatively quickly. These functions help to encapsulate intricate pieces of reusable logic and are optimised by Postgres in order to maintain performant queries. Several places throughout Skeem uses them, such as to format responses of fetch requests.

8.1.4 Jest

Jest is a testing framework which allows for the creating of automated tests (Facebook 2019). It provides a simple interface to perform assertions

concerning the code and is an essential tool used to prevent feature regression on such a large project. Section 9 details the writing of these tests.

8.1.5 Other Libraries and Services

Skeem also takes advantage of many prebuilt libraries and services.

NPM is the defacto package manager for node (NPM 2019), it provides secure 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 (Pusher 2019). It provides simple wrappers are web sockets as well as fallbacks to ensure compatibility across, even outdated, browsers. Skeem uses Pusher to send messages to clients in order to enable the live updating capabilities.

Amazon Web Services are a cloud computing platform(AWS 2019) which provides relatively affordable file hosting. Skeem utilises its capabilities in its file storage functions.

GitHub is a distributed code platform which hosts git repositories. Many features are provided for assisting with project development, one of which is the issue tracker, here, users can raise issues concerning the project. This allows a unified place for developers to track bugs that were discovered and also to make requests for features.

Docsify is a system which allows for a documentation website to be created automatically from markdown files placed in a Git repository on GitHub. Having the documentation hosted alongside the code greatly encouraged and aided with keeping the guides and descriptions up to date.

8.2 Project Structure

Due to the size of the Skeem codebase (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, full isolation is undesirable as each section has a tight coupling of the interactions, e.g. if the format of the fetch response changes then the server must update this new change as well as the client code to keep the format consistent from a developers point of view. Due to this, the structure of the project is that of 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 attempting 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 seven packages, five directly related packages, each prefixed with "skeem", specific to Skeem and two auxiliary packages which were extracted and can provide useful functionality independently:

8.2.1 Skeem Packages

These packages include the core functionality specific to skeem. Each package name is prefixed with "skeem-".

- The **server** package contains the majority of the logic; including the implementation for processing requests, creating a database, session authentication, creating migrations, starting a server, and many other tasks.
- The **CLI** is an interface used to interact with the functions provided by the server.
- The **GUI** is a second interface to execute server functions but takes a graphical approach.
- The **client** holds functionality allowing websites to execute queries in an application. Finally, the **common** package contains utilities shared between the packages, for example, the error messages.

Auxilary Packages

These packages provide key functionality to many parts of Skeem but were extracted from the main core due to the generalized utility they offer.

The **es-qu-el** package provides many helper functions concerning SQL, including methods to sanitize values and build complex queries. **Typer** contains utility functions for type-checking unknown variables. By declaring a schema

of the data structure expects, Typer can analyze objects and find how they differ and offer error messages accordingly. **Overseer** is a declarative CLI generator used to power Skeem's CLI.

8.3 Design Patterns

8.3.1 Object Functions

Object functions define a way to convey function invocations in a serializable format. They take the form a JSON object with a single key. This key is the name of the functions name and the value is the functions argument. This value can take any format and is left to the discretion of the particular function.

Object functions are backed by a selection of handler functions. Each valid key for an object function has a one-to-one correspondence with one of these handler functions.

Listing 11: The format of an object function

```
1 { "name": "value" }
```

This object function pattern is prevalent throughout Skeem including in the values for mutations and the filters for fetches.

8.3.2 Type Checking

Many aspects of skeem are control from external unsafe sources, such as a clients computer. In these cases it is crucial to perform some form of check to ensure that the incoming data is of the correct format. I have found throughout the project that this type-checking should be left as late as possible.

This forces the code that is about to use the data to perform its own checks, this means that if you wish to change the expected format of something then the code performing the checks is situated directly next to the code using it. This helps with understanding what the data is expected to be and how it is used.

8.4 The Schema

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

Listing 12: The empty schema.

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

8.4.1 Db

The db section stores the current structure of the database.

Tables

The primary 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 initialised automatically to a random value generated by a cryptographically secure random generator. Having a dedicated, synthetic primary key was chosen, as opposed to natural keys, as it helps enforce consistency throughout the system, with the additional benefit of removing 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 the natural-key changes.

Listing 13: 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 up into the database itself. Skeem uses a custom function in order to 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 publically accessible features exist which utilise these functions.

8.4.2 Models

Models define precisely what data there is and how it can be queried. Each model has a globally unique name used to identify.

Models also store a tableName field. This name 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 for the renaming of the model 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, taking the form of an operator tree, which defines how to realise the subset; and a set of parameter definitions which defines the arguments the scope can accept. Section 8.11 discusses operator trees at length.

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 ability is required as Skeem does not support polymorphic associations (an association that leads to one of the multiple different models). Therefore, it

is not possible, in certain circumstances, to create accurate permission scopes. For instance, given a model which stores addresses, and the desire to use this same model for many other models (e.g. for users and for properties), then the necessary permissions would likely be: users can only access their address or any used for properties, i.e. not addresses belonging to other users. Since creating an association from addresses to the other models is not possible, there is no way to determine if the address is for properties or users. Setting the model to private avoids the issue as access to a joined record would be required before accessing an address.

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

Callbacks allow for taking actions after certain events, such as send an email when creating a new record or perform a mutation before deleting a record. These callbacks will significantly increase Skeems flexibility, but as of yet, they are in the testing stage before being released.

Live records utilise web sockets to inform the client of when records change. From the client, developers can subscribe to a given fetch query and then have a callback triggered when the results of that query update. These updates allow for views always to 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 is required and more thought put into the client API before it could be released publicly. A proof of concept exists with extremely positive results.

Listing 14: 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[]
}
```

8.4.3 Providers

Providers store a set of configurations defining how users can be authenticated. Providers is stored as an array with each element configuring a single authentication strategy. Section 8.12 discusses the purpose and structure of session providers in detail.

8.5 Database Usage

Skeem utilises a single Postgres database in order to store and query for its data. Skeem fully manages this database and is responsible for creating all of its tables and extensions.

8.5.1 Skeem* Tables

Skeem relies on a handful of predefined database tables in order to function. Their names are prefixed with "skeem", in order to differentiate these tables from application-specific ones. Using this prefix is prohibited for application tables.

skeem-schema stores the applications schema
skeem-migrations stores a list of all migrations
skeem-sessions stores a list of sessions. Each row contains information for the user owning the session, when the session was created, when the last activity occurred and the session provider used for authentication.
skeem-files stores references and information for uploaded files
skeem-version stores the current database version

8.5.2 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".

The `skeem-array_to_object` function accepts a 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}`.

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. This functions primary use is to format paginated requests. Specifically, it allows for the `recordCount` variable to be added to the object. Then in javascript, the object's prototype is set to be that of the built-in array, resulting in an object with a `recordCount` property which has all the same functionality as a standard array — all constructed within a single database query.

8.5.3 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. In order to facilitate 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 system starts the current version number is loaded from the database and compared to the highest upgrade number available. If these are equal, the database is up to date, 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, executing each, starting from the current version number counting up until all steps. After each step, the version number is updated to keep it synchronised with the database state allowing for the possibility of an error within the upgrade step.

As of the time of writing, there are six upgrade stages (starting from zero). These stages perform the following:

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

1. Install necessary extensions, set up tables, initialise empty schema.
2. Rename tables from "cord-" to "skeem-" (The project underwent a renaming to prevent conflicts with pre-existing systems) and add columns `executed` and `timestamp` to the migrations table.
3. updating the schema format to accommodate some new features.
4. Create response formatting functions to remove inconsistencies of particular 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 this would cause the throwing of an error as no database would exist. After catching the error, the version number is deemed to be -1. This process means that when creating a new application, by only starting the system, a database will be created automatically (as step 0 would execute next). This feature dramatically improves the time required to set up and start using a Skeem application.

8.6 Migrations

Migrations act as a way to mutate the schema and thus the database. Migrations act as a record of all changes made to the database and allow for making database changes in a consistent and repeatable manner.

There are multiple migration handlers which handle specific changes. Each type does directly affect the database; instead, they modify the schema, then the differences between the old and the new are calculated, and used to generate a set of transformation steps. This separation allows schemas to be generated without any need for a database, useful when performing automated tests.

Each handler is built in complete isolation from the rest of the system, only exposing essential methods.

8.6.1 Migration Structure

Each migration comprises 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 name of the model, to add the scope to; a unique name, to identify the scope within the model; a set of parameters for the query; and the query itself defining how to identify the subset of data.

The name field is automatically generated and exists purely 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 and is used to ensure the execution order of migrations is consistent.

8.6.2 Storage

Migrations are stored inside the database (using the `Skeem-migrations` table) and also within 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 different developers create multiple migrations. 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 naming scheme is done to help ensure migration files have unique names; there is no guarantee 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 same millisecond. Another advantage of including the timestamp first is that migrations appear correctly ordered 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 executed, at which point they would have a row created - If a migration was in the database, then it had been executed. This pattern, however, proved to be confusing as to get a complete list of migrations the system first had to load those in the database, then read all the files and then eliminate files for which the timestamp and name were present in the database. There was also an issue of migrations loaded from files had different properties then those from the database - the database rows had a downData field and an ID field. Therefore the switch to storing all migrations in the database was made, allowing a full list of migrations to be obtained from a single query and used the file system purely for VCS abilities.

8.6.3 Syncing

Before taking any action involving migrations, including executing or rollbacking back, synchronisation between the database and the filesystem is performed. Firstly, any migration file which does not have a corresponding row is new migrations created by other developers and therefore have a row created.

Next, rows without an associated file are deleted, as the migrations must have been removed. If one of these migrations has been executed, then an error is thrown 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 however, the database row was marked as executed, an error is thrown as once again changing the data could prevent rollbacks from functioning correctly.

8.6.4 Running Migrations

Running migrations first loads the existing schema. Executing migrations get given and mutate this schema.

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. There is no certainty that the passed data will contain the correct information or even be the correct type. Therefore, handler first type checks 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 these checks fail, a `migrationValidation` error is thrown along with the message of precisely 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 specified data or aborting the process if currently executing migrations.

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

After repeating this procedure for all pending migrations, the new schema is saved, marking the migrations as executed.

8.6.5 Database Diffing

Obtaining a list of change steps must occur to mutate the database after running the migrations and producing a new schema.

The first step is to assess the differences by comparing the new schema to the old schema. The models and providers only exist in the abstract, as opposed to the db property, which is backed by database tables. Therefore, it is only the db property of the schema that requires the creation of a difference.

The next step in the diffing process is to separate the new, the removed and which have *potentially* updated tables. 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. Likewise, for each removed table, a `DROP TABLE` query is produced and appended to the list.

For each updated table, every column is compared between the new and the old, and a separation between those that have been updated, created and removed is formed using a similar method as when comparing the tables. An array of column changes is now generated using the following rules:

- If the column is new, then an `ADD COLUMN` command is used.
- Else if the column was removed, then a `DROP COLUMN` command is used.
- Else if there are any changes, then an `CHANGE COLUMN` command is used.

Finally these column alterations are concatinated into a single `ALTER TABLE` query and added to the list.

Listing 15: The format of an ALTER TABLE query

```
1 ALTER TABLE {tableNow.name} {column changes}
```

The list of SQL commands is then executed by first initialising a transaction within the database. This transaction means if an error occurs within the mutate steps, the database can be fully restored to before 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.

8.6.6 Rolling Back

Migrations all can 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.

For example, this data is used 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 before the migration because some migrations such as deleting an attribute are lossy and thus are not purely reversible. Instead, it merely guarantees that the structure of the database will be identical.

8.7 The Server

The "skeem-server" package holds all the core functionality concerned with managing and Skeem application. Many functions are provided to perform everyday actions such as running migrations and performing a fetch query.

Skeem then provides different interfaces which expose these helper functions in various mediums.

CLI allows for the management of the application from the command line (section **??**).
GUI exposes the functions through a graphical interface (section **??**).
Client exposes key functions designed to be used by the final website in combination with the client (section 8.16).

Interfaces such as the CLI are simply wrappers which call these helper functions. Which means neither interface needs to duplicate logic and instead can focus on conveying the data to the user in the most effective manner.

8.7.1 The Context Object

This context object holds all the information pertinent to the current Skeem application, including the applications configuration, the root folder, connections to the database, and the schema. Many functions throughout Skeem require a context object as an argument.

Note: The schema is stored in memory as it is assumed that it will never change during the servers life cycle and therefore caching it can prevent many database requests considering how frequently the schema is loaded.

The server provides a function which creates a context object, after supplying a path. The system then searches recursively upwards from the supplied path looking for a file named `skeem.json`, reaching the root directory without finding the file results in throwing an error, stating the absence of a Skeem application.

Upon finding the file, the location and the contents are used to create a context object with the data indicating the root directory and configuration, respectively.

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

8.7.2 The Request Object

The request object is a specialization of the context object. It holds all the same properties and information with the addition of data concerning the specific request. The additional data includes the user who is making the request and their roles. It also, in some cases, holds a reference to files that were uploaded.

8.8 Fetches

All fetch requests perform a single database query independently of how complex the request is. 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 16: A malformed request and the corrosponding response

```
REQUEST:
{ articles: {},  comments: {} }

RESPONSE:
{ 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 `SelectQuery`. This instance holds all the information about the final query to be executed, such as what columns should be selected, the "where" conditions, and the "limit". Each step in processing the request gets given this object and has to gets to any relevant data needed to formulate the desired response.

The `SelectQuery` object has an execute function which, upon receiving a database connection, returns the result of the query.

8.8.1 Permissions

The first step in processing the query is to find and apply the appropriate permissions for the given model. Section 8.12 covers the process of how these permissions are discovered and processed.

8.8.2 Filter

Next, the filter part of the query is processed. This filter comprises not only the supplied filter but also permission computed from the previous step.

The filters are simply an instance of an operation tree. Section 8.11 details their compilation.

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, returning the empty result set. This optimization allows for processing many requests without the need to touch the database at all.

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

8.8.3 Attributes

Attributes define the data to returned for each record and are compiled immediately after processing the filter. Specifying no attributes results in 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 control what it means to retrieve the data fully.

Listing 17: 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 produce 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 is 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. The `by` value contains the name of an association attribute and an `attribute` property. In this instance, the query will be sorted by the given attribute value for the given association.

Listing 18: Using the sort queries association feature to order results by their author's rating.

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

Sorting can also be done using the sum of an association's attribute. Sorting by this method involves 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 given association. Which 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 precisely to sum of the association.

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 or 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 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 19: 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 executing the query, a new select query is generated 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 20: Wrapping the SQL in another query to format the results

```
SELECT json_agg("results")
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 8.5.2).

Then, generating an entirely 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 the final function.

Listing 21: 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

Like fetch requests, mutations construct and execute a single database query. Which is always true independently of how many tables are being affected. This singular query is achieved by the use of a data-modifying common table expression (CTE). CTE's acts as auxiliary queries for the use of the main

statement ("WITH Queries" 2019). They can be thought of as sub-queries which execute independently.

8.9.1 Dbchange Object

Throughout the parsing of a mutation request a single object, called `DbChange`, is added to, similar to the `SelectQuery` used by the fetch request. This object holds all the relevant information regarding a database change; this not only includes the actual updates to tables but also validations that must be performed and tasks which must execute.

Listing 22: An empty changes object.

```
{
  "currentKey": 0,
  "waitFor": {
    "before": [],
    "after": []
  },
  "tables": {},
  "returns": [],
}
```

The `currentKey` is used to assign every CTE a unique identifier, which can then be used by others to refer to the results of a query. After every use, it is incremented. The `waitFor` key holds an array of tasks, in the form of asynchronous functions, which must be executed either before or after the query has occurred. The `returns` array contains a list of objects which will formulate the final response. The `tables` object holds data concerning the actual changes to the tables, the keys in this object represent the name of the table being altered and the value is a list of `DbChangeRecords`.

8.9.2 Dbchangerecord

A `DbChangeRecord` holds information pertinent to a single row in the database. Each record has a queryKey which is unique to the `DbChange` object. This

queryKey can be used to refer to the records return value. Following this, the record stores an array of properties that should be returned.

If the record is designed to update or destroy an existing row, then it will contain an `id` property holding the id of the concerned row. If this `id` property is missing, then the record is assumed to be creating a new row.

Records have an object containing the values of the columns to be modified. The value can either be a primitive (string, number, boolean) or an object function. The available options of this object function are:

- `now` which will return the current date.
- `session` which will return the id of the current user.
- `raw` which will allow SQL to be passed as the value.
- `queryKey` this will automatically pull the id from the result of the change with the matching queryKey.

Records also contain an array which can hold `validations`. These validation functions get passed the record and have the option to add errors to an `errors` object held by the record.

8.9.3 Processing a Query

Mutations start by type-checking the general structure of the request. It is ensured to be an object with a single key. This key must the name of a model within the schema. The value in the object must be an object or an array; if it is an object, then it is wrapped to become an array. If these checks fail, then a malformedRequest error is thrown.

The queries values is an array of *change actions* to be made. For each change a handler function is executed which appends the necessary information to the `DbChange` object.

After all the changes have been processed, the "before tasks" in the changes DbChanges object are run. These tasks are asynchronous actions which must occur before any SQL can be executed, usually because the action will add additional data to the changes object.

An example of a before action is seen when uploading an image. The image attributes setter creates a beforeAction which will store the file in the relevant

location and then modify the changes object with the reference string.

Next, the validations are performed. This step involves iterating over every record in the tables object and calling each validation function for each. If, after running the validations, any record has a non-empty errors object, then a validation error is thrown and the mutation aborted.

Finally, the SQL is generated, executed, and its response returned.

8.9.4 Change Actions

Three change actions can be performed: create, update, and destroy. Each action will create a new `DbChangeRecord` and adds it to the `DbChange` for the table specified by the current model.

The **create** action iterates over **all** the attributes for the model and invokes all of their `set` methods. The `set` method is detailed in section 8.10.

The **update** action first ensures an `id` property is present in the data specified. If not, a malformedRequest error is thrown. The action then goes through **only** the attributes which have a new value specified, calling the `set` method on each.

The **destroy** action performs the same check for the `id` property and then sets the `shouldDestroy` flag to true.

8.9.5 SQL Generation

Generating the SQL from a `DbChange` object starts by converting each record to its own, independent, query.

SQL generation starts by transforming the attributes into a list of columns and sanitized SQL values. The attributes are stored as an object where the keys represent the column names. If the value is a primitive, then this is sanitized as a SQL value. If the value, it is in the format `{now: true}` then it is replaced by the SQL of `NOW()`. If it is instead a reference to a queryKey, then it is converted to the SQL of `SELECT id FROM {queryKey}`.

The type of query depends on several different factors for each record.

If the record has its `shouldDestroy` flag marked, then a delete query is generated:

```
DELETE FROM {tableName}
WHERE id={record id}
```

Else if the record does not have an id, then an insert query is formed:

```
INSERT INTO {tableName} (columns)
VALUES (values)
RETURNING id
```

If neither of these conditions is met, then an update query is created:

```
UPDATE {tableName}
SET {column=value for each attribute}
WHERE id={record id}
RETURNING id
```

Next, these individual queries are combined into a single query. The queryKeys for each record become the alias for the sub-query:

```
WITH RECURSIVE
  {queryKey} AS ({record SQL})
SELECT {processed returns}
```

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 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 a 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 implements a specific interface which allows 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 value of the attribute. 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 merely means the attribute cannot be read.

The `get` function is passed:

- the context (see section 8.7.1)
- 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, which, 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 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 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 removes many checks which usually occur throughout front-end code, which have to check if the field is null before its use. Also, when an HTML form field is displayed, its value will always be a string, never null. Therefore removing their ability to store null, eliminates differences between fields once tied to an input field 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 from 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 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 resulting in strings being sorted more logically, e.g. alphabetically 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, which means that this type should cover all use-cases for websites. Numbers accept an `integer` flag; in this case, the column type will be

changed to be of type integer; this offers substantial improvements to certain operations 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, provided 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`, avoiding the need to check if something is `false or null` when performing a query. Additionally, this makes more sense logically as it expected that booleans always represent two-state logic and not three.

Booleans accept a flag which will cause them to default to `true` rather than `false`.

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

When `sorting`, all like-values will be grouped, meaning 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 accept 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 feature allows the date field to default to the current time, useful for storing the date of creation for the record. Enforcing this default is done at a database level.

The `get` function for dates is trivial, merely 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 securely. 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 be tuned to make computing take longer. The resulting hash holds the information about the used algorithm meaning passwords hashed with different algorithms can co-exist (pgcrypto 2019)(https://www.postgresql.org/docs/8.3/pgcrypto.html).

8.10.7 Associations

Association attributes are one of Skeems must valuable 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. Which simplifies query generation as it is always identical, it also allows for associations to change to a "has many" relationship without any changes to the database.

When creating an association, the first step taken is to generate information for the joining table. Which 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.

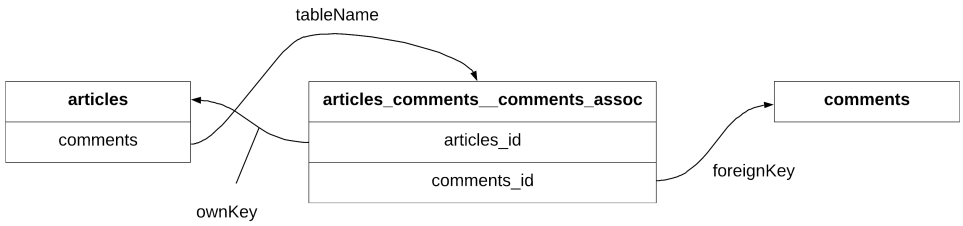

Figure 5: The tables and keys for an articles-to-comments association.

Associations allow for specifying an `inverseOf` property in the configuration indicating 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 are 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, preventing the creation of a relationship of 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. Processing this request is done in the associated model contexts, 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 throughout requests.

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

Listing 23: The where condition responsible for limiting 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 is its argument). There are multiple possible functions which can be called to manipulate the association, though they vary depending on 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 - useful for joining two pre-existing records.
- **Set** is only available for a has-one association 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 indicating whether this attribute should represent multiple images. 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 8.7.2). A callback is added to the databases before tasks (section **??**) 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 centimetres, then a computed attribute could be created returning the value in inches.

This attribute stores a computation in the form of an operator tree. Its `get` function 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 take the form of object functions and are predominantly used to convey calculations for use 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.

Each operator handler function accepts a context object (see section 8.7.1), 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 24: 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 many built-in operators designed to fulfil 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 optimization is performed whereby the returned SQL will 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 complicated function. Databases treat `NULL` values as distinct from all other values, including other `NULL` values. Therefore, the produced SQL takes the form of `a = b OR (a is NULL AND b is NULL)`. Features such as these contribute to making 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 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 is met, then the operator will filter out all true values and combine the remaining values with `AND`.

The `or` operation acts similarly, 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 provides 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 simplest of these operations, only 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 only 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 whose name matches the value, then sanitizes the value then returns it as a value.

The `id` operator returns the ID of the current record, which 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 return a boolean indicating whether the result has any results.

The `anyIn` operator accepts an attribute name (which must be an has-many association) and an 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 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 must be an association attributes name 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 is an exception to this rule and can be of any attribute type.

8.11.5 Miscellaneous Operators

Skeem also provides a few operators who do not 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 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 higher number of operators, very rarely is `ids` used for more than 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 number 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 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 operator is useful in combination with some of the association operators.

8.12 Sessions and Authentication

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

Session providers 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 two distinct user sets which are authenticated with different models, e.g. for a school system there may be one user authentication strategy for 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 provider 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.

8.12.1 Oauth Providers

When authenticating with an OAuth provider, the system will first generate a URL for the given service. This URL is presented to the user, which after following, will present a form allowing them to log in to the service and grant access to necessary information such as their name and email. After the form has been submitted, the Skeem application will receive an access token.

This token is then used to retrieve the user's personal information, including a unique ID.

Part of the configuration in the schema for session providers is a reference to an attribute on the users model responsible for store the user's ID obtained from the third party service. The system uses this attribute detects if this user has been seen previously, if so then they are authenticated as this record. If, however, no such user exists, then the system will attempt to match them via their email address.

If no user is found, then one must be created. Each provider, in the schema, defines an information map. This map provides a mapping from information obtained from the provider to attributes on the system. A new record is created using this map to populate data such as the user's name and email. Finally, a session is created for this new user.

8.12.2 Authenticating

After a provider has identified a record, a session must be created. A session is merely a row in the `skeem-sessions` table, which holds a token, an ID of a record, the provider that identified the record, and whether the session is still active.

A new row is created in the table for the identified record, and a token is generated securely. This token is then encrypted using the JWT standard and a secret key help within the configuration. The encrypted token can then be sent to the client, and upon returning will be decrypted and used to identify the authentication record.

8.12.3 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 permission lists what role they relate to, what action the permission is for (e.g. fetching, updating, creating) 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 user's roles. If *all* of the permission scope is explicitly false, then the query is aborted, an empty result set is returned. If *any* of the scopes is explicitly true, then the permissions are skipped as the user has access to all the records. Finally, if neither of the conditions is met, then the false values are filtered (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 uploaded file, 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. The configuration object stores the connection information. 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 and handler’s name are stored in the database alongside the other file information. By storing the file handlers name, the default file handler can be changed while 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. Image attributes return this token allowing for file retrieval retrieving.

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 appropriate handler then has its retrieve method called, passing it the reference string and returning a stream of the file data.

8.15 Plugins

Plugins allow Skeem to cover a much more extensive use case than would otherwise be possible given the number of potential edge cases. By forcing system features to use a plugin style system it helps to define the edges of systems better and to creating cleaner more targeted sub-systems. This helps to prevent the core of Skeem from becoming bloated with highly specific features.

> Note: There are examples of 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.

8.15.1 Loading Plugins

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

Each pluggable feature specifies the following:

name this is used when logging debug information.

builtinsFolder specifies 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 25: 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 doesn't 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 pluggable elements, it references it through the manager.

8.16 Client

The client is an interface for the `skeem-server` package and is the system that websites interact with to request and mutate data. Because it is designed to be exposed publicly it does not expose any methods concerning database management

The client is split into two distinct parts. The first is a HTTP server which listens to incoming requests, formats them, and calls through to functions exposed by `skeem-server`. The second is a library which can be used on the front-end of websites which accepts queries and sends requests to the HTTP server.

8.16.1 Http Server

When initializing the server will create a context object. Then will begin to listen on a port defined in the configuration.

Queries

All queries 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.

The endpoint uses a JSON-pure API design. This means, instead of using HTTP status codes to convey errors, all responses are a "200 OK" response and the response data is of a consistent format which holds information pertinent to the error. This means that all requests, no matter their result get returned as a "200 OK" response. Doing this decouples the requests from the HTTP protocol which means Skeem could be transferred to use a different protocol (such as websockets) if desired without much difficulty.

Every request begins with the instantiation of a request object (section 8.7.2).

Request bodies are JSON objects which contain two keys: `type` and `data`.

Responses take the form of `{ data: ANYTHING, error: ANYTHING }`.

Any errors throw during processing are caught and converted to be the correct forma If an error is throw during a request it is caught and its contents

Authentication token is in the header.

This end point can also accept requests with multi-part content types. In this case the request data is expected to contain one or more file objects and a single JSON field under the alias of "body". If this field is not present

File Retrieval

The server provides a dedicated endpoint used to retrieve files and is found at `GET /file/:token`. This end point is uses a GET request as it allows an images to be accessed by going to the a static url. This means an `<img />` tag can be rendered with its source set to that url and have the image display.

The token supplied to the end point is used to retrieve a given file, explained in section 8.14.

8.16.2 Front-End Library

The front-end library provides the user with an object called the `client`. This client holds many functions to assist with the use of Skeem and the HTTP server. The client uses the HTML fetch API to send requests to the server.

The library essentially wraps the each request type in a helper function to provide easy access to the HTTP server. Upon receiving a response, the library will automatically store a response token if one was returned.

The library also performs some minor processing on response objects depending on the request. For instance a paginated fetch request does not return an array, instead it returns an object with a length property and records stored at numeric indexes. This response should be treated as an array though. In order to give it the methods found on the standard javascript array object, the library sets the response objects prototype to `Array.prototype`. This means all the methods are placed upon the response object and it can be treated as a normal array.

8.17 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's secret key. Many of these configuration options could change depending on the current environment and therefore, cannot reside within the schema.

8.17.1 Different Environments

Throughout a projects lifetime, it will be run in multiple environments such as development or production. Each environment will likely want its configuration. For instance, when developing an application, it is likely desirable

to store files locally for speed and cost reasons, whereas, in production, it is probably wanted 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 26: An example of a config with multiple environments.

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

8.17.2 Environment Variables

There are many settings in Skeem for which the value may not want to be hardcoded. These settings 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 being 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, the value in the configuration is set to `{ $env: 'name of env variable'}`. Skeem searches all values of the configuration for objects in this form and substitutes them for the specified variable, throwing an error if this variable is not defined exist.

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 before passing the configuration.

Listing 27: Configuration which uses environment variables to avoid exposing critical information

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

9 Testing

Testing is an essential part of any software project, especially those providing some critical functionality to users - Skeem is no exception.

When writing tests, the philosophy of testing functionality over implementation was taken, i.e. write tests for what something should do rather than how something should be done. Bugs are not usually found by looking at the code, but rather by running the system and getting unexpected behaviour. This means that tests should be written independently of the code.

This methodology, although adequate for the majority of cases, does fall short for some functions. Mostly in cases where there are a high number of branching paths. In these cases, it can be tough to cover every possible outcome that can occur. This issue is solved with smaller, more targeted tests are written, sometimes covering just singular functions. Which, however, tightly couples the tests to the implementation does making them very fragile. Meaning the implementation could not be changed without the need to update many tests at the same time. Therefore, this approach is used only when necessary.

Each test consists of a description detailing what the test is about and a function to be executed. This function will perform some actions within the system and then make assertions about the results. These assertions can be anything from testing that the return is a string, ensuring a specific error was

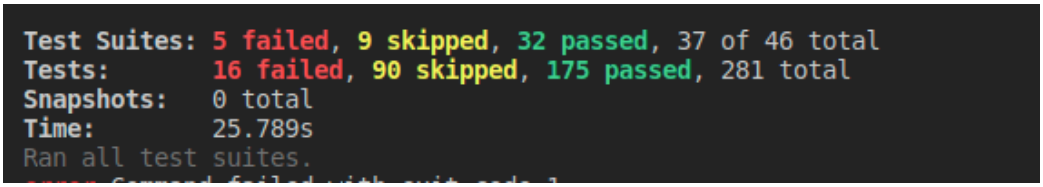

Figure 6: Skeems current test results.

thrown, or making sure a particular function was called in the process. If all of these assertions are successful, then the test passes.

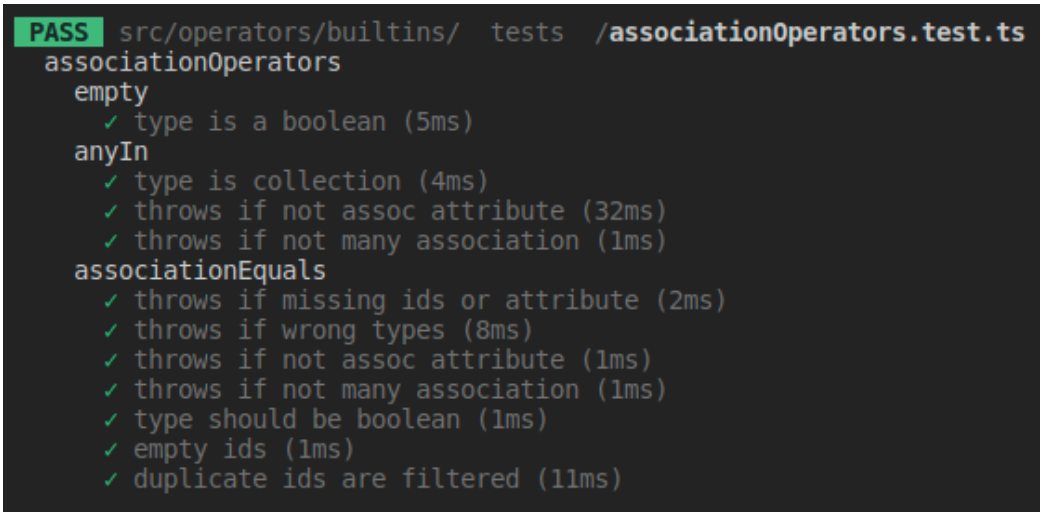

Figure 7: The test results for the association operators.

9.1 Test Driven Development

When a new feature is to be added, I would first plan out what the final interactions with it would be. This usually involved defining the API for the feature and creating use cases demonstrating it. I would then solidify these design choices by writing tests asserting the API was correct, and the use cases passed - these tests would fail as the underlying system did not exist. I would with the actual implementation of the feature, using the tests as a guide for determining when the work was complete.

Once all the tests were passing, I would analyze the code and find paths which did were not tested. I would then write more granular tests focussing on these parts specifically.

Listing 28: Example of what the high level tests would assert during the development of fetch queries.

```
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'`
```

9.2 Continuous Integration

Continuous integration (CI) is the process of automatically running the build and test scripts when changes are made to version controlled code (Guckenheimer, n.d.). Due to Skeem use in production, it was imperative that the testing was continuously run. By using CI methods, I was able to ensure that Skeems code base could always be relied upon and that progress was always made without breaking pre-existing features.

9.3 Code Climate

Code climate is a tool which performs static analysis on codebases and provides feedback about any potential issues found. These issues include:

- Cognitive complexity: how complicated is the code to understand.
- File and function length: does the file or function contain too many lines
- Duplication: are large parts of the code duplicated in multiple places

At the time of writing, Skeems codebase contains 94 code issues, mostly concerning the length of certain functions, and 28 issues about duplication. Some of these duplication concerns, however, are not overly relevant as Skeem

is broken up into multiple sub-packages and Code Climate does not take this into consideration.

Additionally, providing metrics like code coverage (i.e. the percentage of code executed when running tests). Code coverage is a useful metric for determining what parts of the system have not been tested. It allows testing efforts to be focused on code that actually requires attention. At the time of writing Skeem testing covers 49% of all the lines in the codebase.

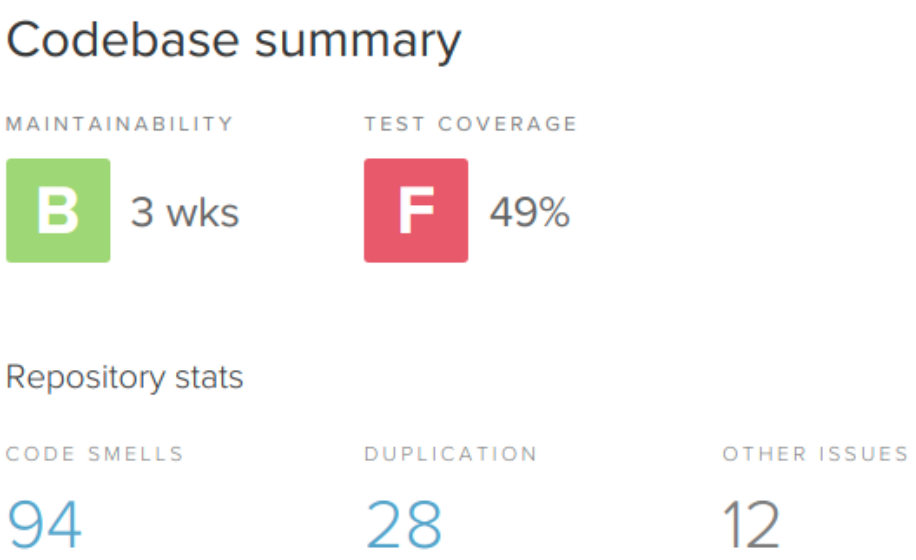

Figure 8: The results of Code Climates analysis at the current time of writing.

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. Their website was rebuilt from scratch late last year, which perfectly coincided with Skeems development. They were the first platform that used Skeem and helped to influence many of the decisions and features Skeem now presents.

> At the time of writing, their website consists of 38 models with 397 attributes.

Having a real world application to develop Skeem for meant that Skeem is known to contain features which are practical and required for web development. Features such as pagination returning the record count and being able to associate pre-existing records, were not included until Resooma demonstrated a need for them. Resooma also prompted the need to have third party authentication methods such as Facebook and Google.

10.1.1 Resooma Native

Later in the project, Resooma launched a mobile application backed by the same Skeem system as their website. Their ability to use Skeem in mobile applications demonstrates the cross platform compatibility of the client.

10.2 Enterprise Security Distributions Norwich

Enterprise Security Distributions is a relater for security products. Over the past year, their Norwich branch has commissioned a piece of software to generate quotes for their customs and keep stock of their inventory. Skeem has been deployed as their main database system.

Their system was for internal use and consisted of many tables displaying information products, quotes, customers. Each table required a set of filters to allow them to filter their collection of more than 20'000 products. Skeem handled this filtering requirement with no issues.

Prior to this system fetch requests had a much more limited sort functionality. It was this system that prompted for additional capabilities to be added such

as being able to sort by multiple attributes as well as sorting by a query path.

Their system consists of 16 models, 22 scopes, and 257 attributes.

10.3 Other Applications

There many other applications which are in development which have chosen Skeem as their primary method for data handling. These projects are in their early days and will no doubt help to shape Skeem as they grow and their needs change. For now, though, each one has been able to make rapid progress by utilizing Skeems ability to create and manager databases extremely quickly in a way that is intuitive for web development.

Voluble is a chat system as a service. It provides the ability to easily integrate a messaging functionality to a website. There needs consists of having to store, potentially thousands of records covering messages, conversations, and users.

Rolecall is a application that helps facilitate the management of, contract based, jobs requiring on-site work. The application includes mapping features to track workers, and messaging capabilities to enable live communication. There is also plans for a mobile application to be created in the near future. Skeem has been deemed a viable tool to use for this project.

Inbox Integration is a company who is using machine learning to analyze and detect fraudulent invoices. There high dependence on machine learning means the system contains many bespoke features that are not common across other websites. With Skeems ability to create a client which on the server, they are able to make full use its abilities.

11 Evaluation

This chapter outlines the whole project, covering the challenges faced during development and the future for Skeem.

11.1 Requirements Review

Skeem has been hugely successful in solving the presented issues. Its built-in database management and declarative schema have removed any issues concerning API design. This design, in conjunction with the client library, has dramatically reduced many of the coupling issues.

The built-in authentication and file management systems not only limit the bespoke knowledge problem but also greatly removes the boilerplate required to utilize such features.

Skeem's documentation and helper library functions goes a long way in helping developers discover how to complete tasks. This, in combination with descriptive and precise error messages, greatly helps to reduce the training time and required knowledge.

The specification gathered from the in-production database has been mostly fulfilled by Skeems capabilities with a few exceptions. There is no ability to trigger actions, such as sending an email, after creating a record. The lack of functionality means that Skeem may not be so universal as to replace database in all scenarios. This functionality was never implemented due to its lack of need on the projects I was working alongside. Another missing feature was conditional validations. Again being sacrificed for the same reason as callbacks.

11.2 Challenges and Limitations

Before this project, I had minimal experience SQL, having only really performed relatively trivial tasks. Learning the nuances of constructing queries to complete complex tasks took significant time. The format of the fetch queries was greatly expermiented with to gain the best performance through the similist query. The mutations, on the other hand, require further expermintation as, although functional, are quite complicated for the limited functionallity supplied.

11.3 Future Work

One of my biggest priorities for Skeem is increasing test coverage, which currently stands at just under 50%. This is an essential task as it the longer code goes untested the high chance of feature regression and the harder it is to bring new developers on board to the project.

Another big priority is adding more features which were listed in the gathered specification. These features indicate real-world use-cases and therefore fulfilling these would further Skeems viability to be used.

The client is currently a thin wrapper around requests; however, so much more is feasible. For example, one area of interest is implementing the caching of results. As Skeem controls all aspects of storing and retrieving the data, it is feasible for the client to be able to parse fetch requests and perform queries against a local cache and then only send to the server a request for the missing data. Doing this could significantly increase performance for many skeem queries.

Another avenue of development is improving the method in which migrations are created. Currently having to create attributes one at a time can be a slow and monotonous process. The CLI could implement a higher level wrapper around migrations to allow for the editing of models more directly and then derive the necessary migrations after the fact.

12 Conclusion

There is a range of projects using Skeem, including some mobile applications. I believe Skeem to be a very flexible and beneficial system to these companies, helping to simplify their development process. In general, it has been very well received by the developers of these systems.

Now that this project is complete, from a university perspective, development of Skeem will be opened to a handful of other developers who are eager to start work. I am hopeful for Skeem's future as a fundamental tool in aiding the development of exciting systems.

References

Auth0. 2019. “Never Compromise on Identity.” *Auth0*. Auth0. https://auth0.com/.

AWS. 2019. “Amazon Web Services (Aws) - Cloud Computing Services.” *Amazon*. Amazon. https://aws.amazon.com/.

Borenstein, Nathaniel S., and Ned Freed. 1992. “The Multipart Content-Type.” W3C Recommendation. W3C.

Facebook. 2019. “Jest · Delightful Javascript Testing.” https://jestjs.io/.

GraphQL. 2019. “GraphQL: A Query Language for Apis.” *A Query Language for Your API*. https://graphql.org/.

Group, The PostgreSQL Global Development. 2019. “PostgreSQL: The World’s Most Advanced Open Source Database.” https://www.postgresql.org/.

Guckenheimer, Sam. n.d. “What Is Continuous Integration? - Azure Devops.” *Azure DevOps | Microsoft Docs*. Microsoft. https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration.

Microsoft. 2019. “TypeScript - Javascript That Scales.” *Typescript*. https://www.typescriptlang.org/.

MongoDB. 2019. “The Most Popular Database for Modern Apps.” *MongoDB*. https://www.mongodb.com/.

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

NodeJS. 2019. “Node.js.” *Node.js Foundation*. https://nodejs.org/en/.

NPM. 2019. “Npm | Build Amazing Things.” https://www.npmjs.com/.

pgcrypto. 2019. “F.20. Pgcrypto.” *PostgreSQL*. https://www.postgresql.org/docs/8.3/pgcrypto.html.

Pusher. 2019. “Leader in Realtime Technologies.” *Pusher*. https://pusher.com/.

Spinellis, Diomidis. 2016. "The Bad Code Spotter's Guide." *InformIT*. http://www.informit.com/articles/article.aspx?p=457502&seqNum=5.

"WITH Queries." 2019. *PostgreSQL*. The PostgreSQL Global Development Group. https://www.postgresql.org/docs/9.1/queries-with.html.