Skeem - a database system to deal with modern needs

by

Henry Morgan

Supervisor: Dr. C. Hatter

Department of Computer Science
Loughborough University

April/June 2019

Abstract

Skeem is a datbase

Contents

1 Key Terms

Key Definitions JSON (Javascript object notation) is a human readable text format which encodes key-value pairs and arrays of primitive values (strings, numbers, booleans, nulls). Although JSON technically denotes a string the term will be also used when referring to a parsed version of the string i.e JSON may referer to a string denoting an array containing numbers or an actual instance of an array containing the elements 1,2 and 3.

2 Introduction

This is the introduction

3 Problem Definition

3.1 APIs

- APIs are hard to make

  - Consistent
  - Flexible

- They are either very complicated

- API design usally doesnt require a team of people to maintain but everyone needs to interact with at some point, waste of time training people

- Spas are growing in popularity however with an SPA comes the need to an API to serve the correct data

- Pages are no longer requested and built on the server, instead they are built on the client. This allows websites to be more interactive and can massively increase responsiveness. However, with SPAs comes a new issue:- you need a way to request data from the server, this leads to the need of an API.

- Apis tend to require knowledge which is not needed for front end development.

- APIs also tend to be very repetitive i.e you need an end point to fetch a list of blogs and you also need one to fetch all the products. What is really different about these routes? The table name in the sql and the attributes it returns. This duplication of code leads to more code which leads to more potential for bugs.
    - Lots of code = Lots of bugs.

Se- paration of code but a strong connection on functionality.

- SPAs require a server to interact with the database as sql is not safe to execute from a client

-----

- Duplication of requests. Imagine having an end point requesting a list of blog posts. On the site you wist to display the title of each blog with a short extract from the body to act as a teaser. On this teaser you also want to display the authors name. You may also have a author page showing information about the author. This leads to an issue of either having two end points which return very similar data or reusing the end point but then forcing the end user to download more information then they actually require.

Building an API is easy, building a consistent and flexible API is hard.

3.2   Databases

When building any non-trivial database driven application there are certain features which will need to be implemented.

- Repeated Code

- Creating and maintaining a database involves a lot of repeated tasks

- This server has a lot of code dedicated (sometimes entirely) for database interaction.

- Databases are complicated

  - It requires a lot of understanding to setup and maintain a database especially when wanting to perform any non trivial, but abstractly simple, tasks such as:

    * associations
    * file storage
    * user authentication
    * permissions
    * etc

  - All these tasks are very well defined and understood yet when setting up a database you have to reinvent the wheel every time.

  - Many non-technical people could take advantage of a database yet don't understand how.

    * Think how many excel spreadsheets get used and how complex they become
    * People need ordered and relation storage and retrieval systems

---

Databases should store normalizes data which, simply speaking means so structure data in flat tables i.e one for articles, one for authors, one for comments and then you store relational information on the tables. E.g a column on the comments table referencing a specific article and an author field on the article link it to the correct user. This is to remove duplication and allows database systems to cache and index data which is the reason databases can be so performant even over enormous datasets.

The issue with this, however, is that data is not displayed like this to the end user. The end user is not presented with a page containing an article and is required to navigate to a separate page displaying the authors name then have to navigate to a third place to read a list of comments. Rather the end user will be presented with a single page containing all the data

amalgamated in an easily digestible and pleasant format. The data the user sees can be envisioned as a tree of data: the root being the article itself and then containing a connected nodes for each comment each having further nodes containing their authors.

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


```sql
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"
```
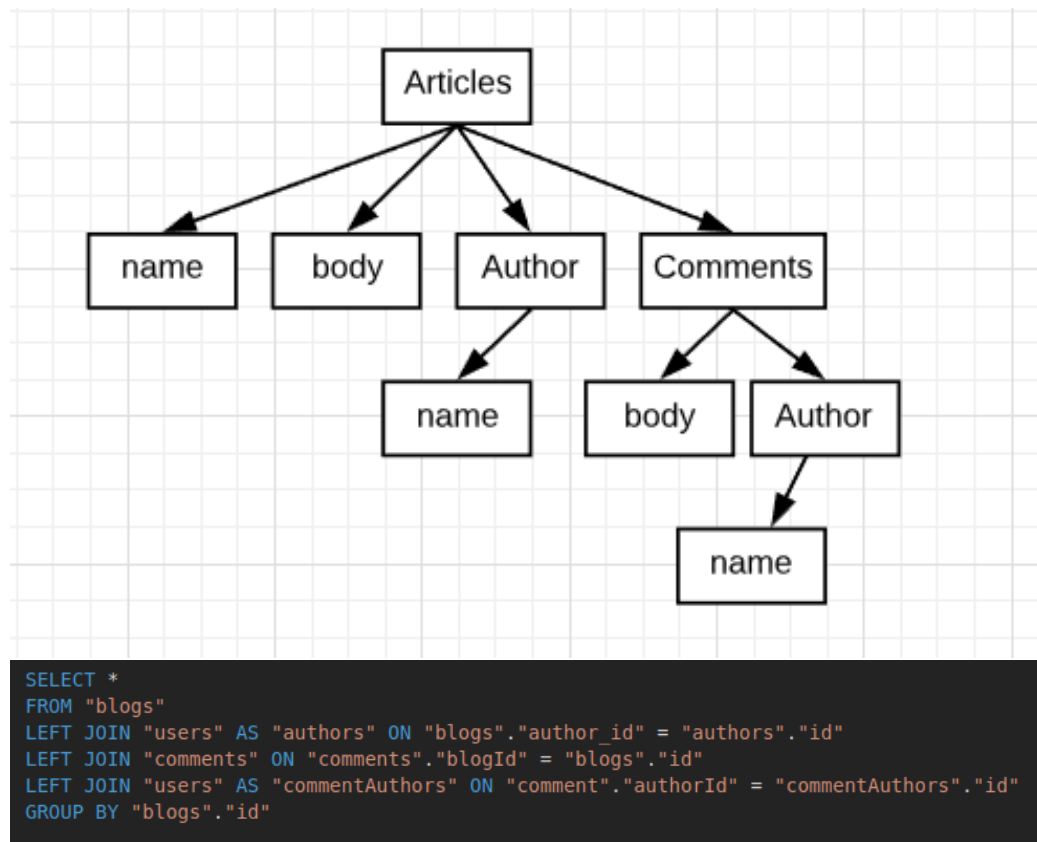
- Data consistency
  - Once you retrieve the data from a data source (server) it is important to keep that up to date
  - if it gets updated on the server it should be reflected on the client

9

4 Literature Review

find some stuff how does it solve the problem where does it fail

- GraphQL + Relay

    - Sits on top of an existing
    - Provides a way to query a tree structure of data
    - complicated - non feasible for non technical people

- NoSQL databases

    - Stores arbitrarily shaped json allowing data to be stored in a fashion similar to its usage

- Excel

    - Very simple
    - Fails at associations, files etc.

- Web framework

    - Rails

5 Requirements

5.1 High level requirements

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

- Consistency
  - alert clients to changes in data

5.2 Specification Gathering

I was given access to a production server

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

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

- attributes
  - has many through
  - has many through where condition
  - has many with condition
  - has many dependent nullify
- Validations
  - presence
  - uniqueness
  - inclusion
  - number greater than
  - uniqueness in scope of attr: value
  - validate uniqueness in scope with condition unless attribute: value
  - Validates on: :create
  - association.attribute must = value
  - validates [if/unless] attribute: value
- Callbacks
  - before_validation
    * default attribute to another attribute if not present
    * default attributes only on create
    * default attributes to parameterized other attribute
    * default attribute to association attribute
  - before_create
    * self.slug = name.parameterize
  - after_create

                * update association

                * send emails

                * update self

- Scopes
  - where(attr: value)
  - where.not(attr: value)
  - order(attr: :desc)
  - where association count >= 1
  - where association count === 0
  - where association attribute
  - where in associations scope e.g where(tag_id: Tag.published)
  - composing scopes (adding limits)
- Permissions
  - through user association
  - through user association | where(attr: value)

Using this information I obtained the minimum viable feature set needed

5.3 Technical Requirements

- Create Models

  - store basic types strings, number
  - store associations between two models
  - store files

- Fetching

  - attributes
    - * Request primitive attributes such as strings, numbers
    - * request associations
  - provide a filter to a query
    - * request a record given the records id
    - * request a record based on its attributes i.e requesting published records
  - sort queries
    - * by attributes
    - * by associations attributes
  - pagination queries

- Mutations
  - create records
  - Update records
  - delete records
  - add/remove association records
  - upload files
  - validate data

- Sessions
  - Authenticate users
  - Specify users permission to access data

- Consistency
  - Use web sockets to be alerted to updates

- Permissions
  - Specify access (read + write + remove) of users on:
    * records
    * attributes

- Provide a way to change production databases safely

- GUI
  - Provide a way to create a database
  - Provide a way to create a model
  - add/update/remove attributes from models
  - seed data
  - view records for a model

6   Solution

In order to solve the problem I have created a system named Skeem. Skeem provides an interface for creating Skeem is designed to be used by developers. This chapter acts as a high level overview to skeems functionallity, and how to use it.

- What is skeem
- Skeem requires no code

6.1 Models

Models are one of the most important aspects of Skeem, they define the data. They control what gets stored and how, as well as how it is accessed and who can access it. They are akin to database tables but not neccessarily one to one.

6.1.1 Attributes

At attribute is like a database column, it stores a single piece of data for a model. Attributes are comprised of a name, a type and a set of configuration specific the the type chosen.

There are a number of built in attribute types which aim to accomadate any type of data you may want to store. If, however, you have a need to store data which one of the built in types do not support then you Skeem allows for a custom attribute type by the means of a plugin, see the chapter on [attribute plugins][]

Strings

- presence validations

Numbers

Booleans

Dates

Passwords

Password attributes are designed to store Passwords They are distinct from strings Password attributes can not be retrieved, only updated. They automatically hash the value supplied to them

Associations

Association attributes store links between models.

Files

File attributes store

6.1.2   Scopes

Scopes define methods for fetching subsets of data: published articles. Scopes are built from a tree of comparison functions

6.2   Requests

Communication with the server is achieved through http requests. This means that requests can be made from the client. This solves the tight coupling with disperate code issues between the standard client-server model.

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

TREE OF DATA

6.2.1   Fetching Data

Fetching data involves pulling data from the models in a structured fashion. A fetch query specifies a single root model name as the key to the query

object. The value then specifies exactly what data you want to retrieve, how to filter and sort it and whether you want to split it into pages.

A fetch response will always be an array of records, where each record will contain its own id as well as any additional attributes you requested. In some cases you may also retrieve the total record count see the section Pagination.

Attributes

Attributes specify what data you want to receive for each record. Attributes take the form of an array where each element is a string being the name of the attribute you are requesting, or an object with a name property and a value of the attribute. This object notation is required for specifying additional configuration such as formatting information like an alternative name for results.

Attributes can be aliased by specifying the `as` property

Listing 1: This query will return all articles each containing the article's id, name, and body. The body will be aliased under the name "text"

```
1  {
2    articles: {
3      attributes: ["name", { name: "body", as: "text"
           }]
4    }
5  }
```

     associations: trees of data.....

Filter

Filters all for specifying specific criteria records must meet for them to be returned.

Filters you a tree of "object functions". This means that each object within a filter operation contains a single key. This key specifies what function you which to execute e.g equality, less that, empty check. And the value acts as the arguments for the given function.

There are many built in filter functions which cover a broad range of use cases

Sort

Sorting data is an extremely common and essential ability for data retrieval: Most recent tweets, video length, article title. When sorting data you specify what you want to sort `by` and the `direction` you want to sort: either *asc*ending or *desc*ending.

Listing 2: This query will return all articles ordered by the articles "name" attribute.

```
1  {
2    articles: {
3      sort: {
4        by: "name",
5        direction: "asc"
6      }
7    }
8  }
```

You can also specify an array of sorting criteria. Doing this will sort the data initially by the first item, then resolve conflicts with the next item in the list.

Pagination

Pagination chunks the data into pages. You don't want the end user to download 100'000 records, this would be very slow and wasteful. The returned data will be equivalent to a standard array of records.

Listing 3: This query will return the second page of articles where each page holds 30 records.

```
1  {
2    articles: {
3      pagination: {
4        page: 2,
5        perPage: 30,
6      }
7    }
```

```
8    }
```

Record count

As well as the actual records, you also get given a count of the number of records you would have gotten if you did not paginate the data. This is useful when wanting to show end users a list of page numbers and allow them to jump to them arbitrarily.

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

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

6.2.2 Mutating Data

6.3 Permissions and Security

Controlling what users can access, who can create and edit data is an essential part of all application which make use of a database.

6.3.1 Authentication

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

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

The name is used purely to distinguish between different providers and allows for multiple authentication strategies of the same type. You may have to distinct user sets which are authenticated with different models e.g for a

school system you may have one user set named teachers and another for pupils. The model defines where the session provider should look to find the necessary data to check against any credentials provided.

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

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

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

6.3.2   Roles

With authentication we now have two distinct user states: authenticated and anonymous. These roles can be used to define permissions on fetches and mutates

6.3.3   Permissions

Each model can define a set of permissions based scopes

6.4   Config

Application wide configuration is done by means of a json file named `skeem.` `json`

- When running a command the system search recursively upwards from the current directory looking for a file named skeem.json

TODO

6.4.1 Different Environments

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

The configuration for different enviroments 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` enviroment variable in order to guage which configuration block should be loaded. `NODE_ENV` is a standard variable throughout the node ecosystem to distinguish between environments.

Listing 4: An example of a config with multiple environments.

```
1 {
2   developments: {
3     /* development config */
4   },
5   production: {
6     /* development config */
7   }
8 }
```

6.4.2 Environment Variables

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

To use an environment variable you must set the value of the configuration item to be { `$env`: `'name of env variable'`}. Skeem searches all values of

the configuration for objects in this form and substitutes them for the specified variable. If this variable does not exist an error is thrown.

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

Listing 5: Configuration which uses enviroment variables to avoid exposing critical infomation

```
1  {
2    database: {
3      host: { $env: "DATABASE_HOST" },
4      username: { $env: "DATABASE_USERNAME" },
5      password: { $env: "DATABASE_PASSWORD" },
6    },
7  }
```

6.5   Management

There are two predominant methods to managing a skeem application: through the use of a command line interface or via a GUI.

The first step in using skeem is to initialize a project. The easiest way to do this is to use Skeems command line interface (CLI). The CLI has a lot of functionality to help you manage your database and configure your application, see (#CLI). To initialize the project simply run the command `skeem init myProject`. This will create all the necessary files and folders in the current directory for an application named "myProject". The next step would be to create a model.

6.5.1 CLI

6.5.2 GUI

6.6 The Client

Skeem has a client, written in Javascript, designed to be used with SPAs. The client provides functions which will process queries and send them to the server.

6.6.1 Caching

6.6.2 Validations

6.6.3 Oauth flows

6.7 Live Updates

6.8 Plugins

APIs are a large and complex systems which cover an increadly broad range of use cases. It would be almost impossible to foresee every use case of skeem and to allow for every possibility. As such, skeem has the ability to augment functionality by the way of plugins.

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

6.8.1 Custom attributes

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

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

6.8.2 Custom session providers

There are many different ways you may want a user to authenticate.

6.8.3 Custom operation functions

There are a myriad of different and obscure filters you may want to perform within a database. Whilst skeem contains alot of built in operations which can achieve a large variety of results it is implausible that they cover every possible desire.

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

Creating an operation

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

Listing 6: The simplist custom operation - it would always return false and so is utterly pointless.

```
1  function myPointlessOperation() {
2    return { value: `'this will' = 'always be false'`
         }
3  }
```

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

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

Listing 7: Returns all the records for which the name matches the value supplied. However there are major issues with this and should not be used.

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

```
3  }
```

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

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

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

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

Listing 8: Checks to see if name actually exists on the model being queried.

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

There is one final issue with this operation and that is it misses the opportunity

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

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

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

Possible types include:

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

Listing 9: Checks to see if the attribute actually exists and returns the correct type

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

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

Nested operations

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

Listing 10: A simple implementation of the eq operaion.

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

The request context

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

TODO

Using an operation plugin

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

Listing 11: A full operation plugin file

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

6.9   Documentation

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

7 Methodology

7.1 Development Strategy

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

Listing 12: Example of what the high level tests would assert (not actual tests)

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

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

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

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

8 Implementation

In this chapter I shall be explaining some of the intricacies of skeem, how it works, and why certain key decesicions have been made.

8.1 Technologies

8.1.1 Typescript

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

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

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

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

8.1.2 Postgres

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

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

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

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

8.1.3   Other Libraries and Services

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

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

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

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

React React

TODO

8.2 Project Structure

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

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

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

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

8.2.1 skeem-server

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

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

8.2.2 Server

8.2.3 Manager

8.2.4 skeem-cil

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

8.2.5 skeem-gui

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

8.2.6 skeem-client

Provides functionallity for a client

8.2.7 skeem-common

Holds common functionallity needed between packages, such as error messages

8.2.8 es-qu-el

provides helper functions to generate and sanitize SQL statements

8.2.9 Typer

Typer standardies type checking accross the app.

8.2.10 overseer

declarative CLI generator used to power Skeems CLI.

8.3 Database Management

8.3.1 skeem-* Tables

Skeem relies on a handful of datebase tables in order to function. Each of these tables are prefixed with "skeem-" in order to differentiate them from application specific tables.

skeem-schema Stores the applications schema

skeem-migrations :Stores a list of all migrations

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

skeem-images :stores references for files uploaded to Skeem

skeem-version :stores the current database version

8.3.2 Functions

Skeem also relies on a few custom functions in order to properly handle certain requests.

8.3.3 Upgrading the Database

In order to accomadate new functionalitty it is likely the features of the database may have to change, new tables may be required, old tables may need to columns, etc. In order to facility this need skeem provides a database upgrade mechanism.

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

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

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

0. Check to see if the database exists, if it does not the attempt to create it.
1. Install neccessary extensions, setup tables, initialize empty schema.
2. Rename tables from "cord-" to "skeem-" (The project under went a renaming to prevent conflicts with previous systems) and add columns `exectued` and `timestamp` to the migrations table.
3. updating the schema format to accomadate some new features.
4. Create response formatting functions to remove inconsistencies of certain edge case requests.
5. Add a `loggedOut` column to the sessions table. previously sessions were deleted.

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

8.4   Schema

- Every application has a single schema
- The schema is responsible for defining all aspects of the site from how and what data is stored to how people can authenticate and what information different people can access

The schema is a single large JSON object which holds infomation concering all aspects of the application. When an application is first created its schema is initialized to an empty skeleton.

The schema is stored in the database in the table "skeem-schema". This table only ever holds a single row. When the schema is updated the table is cleared and the new schema is inserted.

The schema is serializable.

Listing 13: The empty schema

```
1  {
2    db: {
3      functions: [],
4      tables: []
5    },
6    models: [],
7    providers: []
8  }
```

The schema has three top level keys, db, models, and providers.

`db` holds infomation about the database. This includes a record of all tables including their name, columns indexes, and constraints. It also stores a list of custom defined functions.

`models` stores all the models created within the application.

`providers` holds infomation about the available session providers. Each provider contains a name, its type as well as specific configuration for the given provider type.

8.4.1   Models

Each model has an assocated table

- Always has an id

8.5   Migrations

Migrations are the way the schema is mutated. They provide a simple and reversible method for manipulation.

- Migrations work to mutate the schema

- They allow multiple developers to work on a single application

- A record of all changes made to the database

- Migrations serve incremental, reversible changes to the schema

- Migrations are stored in files within a folder named migrations located in the root of a skeem project

- This allows for migrations to be transferred between computers

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

8.5.1 Database Diffing

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

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

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

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

For each updated table each column

The list of sql commands is then executed. To do this, first, a new transaction is created within the database. This means if an error occurrs within the mutate steps the database can be fully restored to prior to the mutations. Without this then the schema could become out of sync with the database. The list of sql statements is the run sequentially. After each step has been executed the schema in the databse 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 syncricity between the tables and the schema.

8.6   Requests

All requests to the skeem server are a post request sent to the same end point. Request bodies are JSON objects which contain two keys: `type` and `data`.

There is a second format request bodies can take - multipart data. Multipart requests are designed to effectiently transfer large data objects such as files and are designated by the content type of `multipart`/`form`-`data`. Skeem expects multipart bodies to contain a single data field named `body`. This body should be a JSON string which contains all the data normally present in a standard JSON request. The rest of the body contains file data which is processed and exposed to skeem via the requests context. The body field is extracted and treated as the main body and processing the request continues as normal.

> Note: if more then one data field is detected, or the body field is missing, or the body field is not valid JSON then the a malformed request error is thrown and the request aborted.

The `type` part of the body is used to route the request to an apprioriate handler.

If an error is detected with a requests body at any point, either during the inital routing of the request or later after delegating by the type, a malformed request error will be thrown.

Responses always take the form of a JSON object with two keys: error and data - this is similar to stdout and stderror.

If an error is throw during a request it is first caught then it is checked against a list of known skeem errors (skeem errors consitently have a type and a data

where the type is one of a limited number of errors). If it is a skeem error then the response error property is set this error and the request is ended. If it is not then a new Skeem error is created of type server (the list specific type of error).

> If the system is in the development environment then the error will have the stack trace appended to it to aid debugging.

8.7   Fetches

All fetch requests perform a single database request independent of how simple or complex of a query is made. This is a major reason as to why skeem is so performant.

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 that this modelName actually exists. The value of the object is ensured to be an object with some subset of the keys: filter, attributes, sort, pagination. If there are any additional keys the request is deemed to be errorneous. If any of the checks fail then a malformed request error is thrown containing additional infomation explaining the exact issue.

Listing 14: A malformed request and the corrosponding response

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

The fetch request constructs an object name `SqlQuery`. This object holds all the infomation about the final query to be executed, such as the current columns being selected, the where conditions, the limit etc. . . Each step in processing a fetch request simply adds additional infomation to this object.

After the type assertions are complete and the SQL object is instantiated the

next stage is to process the filter and permissions. This is dont first as there is a chance the filter may conclude that no results could be returned - in this case the request can be prematurely aborted and there was no time wasted in processing features like the attributes.

A filter is simply an operation function

After compiling a filter the attributes are processed.

Sorting is applied next. . . . .

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

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

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

8.7.1 SQL Generation

After processing is done the query object must be transformed into SQL. This object is specific to the fetch query and formats its response such that it can be returned without any post processing.

Listing 15: Example logic within the SqlQuery to create the final query

```
1  SELECT ${columns specified joined by ","}
2  FROM ${table name}
3  WHERE ${conditions specified joined by " AND "}
```

TOOD:

The withCount flag

If the `withCount` flag is specified then skeem . . . .. TODO

8.8   Mutations

8.9   Sessions

8.9.1   Authentication

1. type check request
2.

8.10   CLI

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

8.10.1   Overseer

8.11   Plugins

Plugins allow skeem to cover a much wider use case then would likely be
possible if I had to implement all edge cases manually. They also provide a
way to prevent the core of skeem to become bloated with extremely specific
features.

Note: There are examples of very specific features within the code
base such as the `associationEquals` operation. It was, in fact,

Figure 1: CLI Help screen

the addition of features such as this which prompted the need for the plugin system.

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

Each pluggable feature specifies the following:

name this is used when logging debug information.
builtinsFolder this specify the name of the folder where the built in features can be located.
folderName the folder name of where the external plugins will be located relative to the application root.
validateExports a function which will get passed a file when it has been imported and parsed. This function should return a boolean indicating whether the exported contents of the file is valid. For instance checking whether a file handler contains a store and retrieve method.
merge a function responsible for merging all the individual files together into one final object.

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

Listing 16: The code required to define the attribute plugin system

```
1  export const loadAttributes = createPluginLoader<
     IAttribute>({
2    name: "attributes",
3    builtinsFolder: "./builtins",
4    folderName: CUSTOM_ATTRIBUTES_PATH,
5    validateExports(_filename, exps) {
6      // attributes should be an object with three
         keys: "migrate", "get", "set"
7
8      if (!isObject(exps)) {
9        return false
10     }
11
```

```
12        const expectedKeys = ["migrate", "get", "set"]
13        const isValid = Object.keys(exps).every(key =>
             expectedKeys.includes(key))
14        return isValid
15     },
16     merge(acc, filename, exps) {
17        acc[filename] = exps
18        return acc
19     }
20  })
```

Loading a plugin executes the following algorithm:

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

When skeem is started it calls all the loaded functions created by the plugin system. These functions will then load all the built elements and any external ones, merge them, and then store them with the manager. Then when a component wants to use one of the plugagble elements it references it through the manager.

9   Testing

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

Tests were written in Jest.

- Tests were written in jest.
- Tests targeted functionality rather then implementation. However tests were written for smaller parts of the system when functionality became too complex or when the underlying functions were critical - such as loading the schema from the database.
- For a complete list of all tests please see the apendix.
- CI
  - Due to skeem being used in production it was essential that it was not only tested but that testing was constantly carried out. By using CircleCI tests are automatically run when a change deployed to the git repository.
- Coverage
  - Code coverage reports show how many lines of the are touched by the tests this is very useful to ensure all the code branches are tested and perform as expected.
  - Code coverage ended up at 46% at the end of the project.
- Code quality
  - Codeclimate is a service which analyzes code and detects "code smells". "A code smell is any characteristic in the source code of a program that possibly indicates a deeper problem." - Wikipedia. This includes problems such as:
    * Cognitive complexity: how complicated is the code to understand.
    * File and function length: does the file or function contain too many lines (only counting actual lines of code, ie. not comments or blank lines)
    * Duplication: are large parts of the code duplicated in multiple places
  - Codeclimate then predicts the amount of time it would take to fix this technical debt. At the project end, skeem contained 147 code smells with a predicted clean up time of 2 months.

10 Deployments

- Throughout the development i has the opportunity to deploy skeem on real world applications. At this point skeem is in production use in two apps and in employed on 3 further, currently in development, projects.
- Resooma
  - Resooma is a bills consolidation company focussing primarily on university students.
  - 88 distinct models, 700 attributes, 50'000'000 users
- Quote generator and Stock management tool for Enterprise Security Distributions Norwich
  - Tracks more than XXX quotes for customers concerning more than 20'000 products. Used heavily
- II
  - Invoice fraud detection using machine learning
- Voluble
  - Messages API
- Rolecall
  - Job tracking and communication platform aimed at contract workers

There are a wide range of projects using skeem. Very flexible.

11 Conclusion

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

11.1 Future Work

References

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