Degree Name
Project Module Code
ID Number

Skeem - a database system to deal with modern needs

by

Henry Morgan

Supervisor: Dr. C. Hatter

Department of Computer Science
Loughborough University

April/June 2019

Abstract

Skeem is a datbase

Contents

1 Introduction

This project focuses on aiding developers building web applications.

1.1 Motivation

I have been a web developer for a number of years have built many applications across a wide range of systems. Over this time I have developed many systems for a wide range of companies and have refined the techniques needed to create blah blah blah...

2 Background Information

creating a website

---

modern websites must be dynamic and interactive - this is achieved by SPAs

SPAs or single-page applications are a growing trend on the web unlike traditional architectures where each page is a separate resource with its own end-point, its own template and its own request, SPAs combine all the pages into a single end point. When a user navigates to a page they download a large javascript bundle which has the ability to construct any page of the website. The javascript is reads the url the user requested and renders the creates the appropriate views. This means that when a user navigates to another page, the javascript can intercept this and simply render new content without a network request creating a much more responsive interaction.

Note: This is a simplified example of how SPAs tend to work. In reality having a user download the entire code bundle would not be ideal, especially on slower networks. Therefore optimizations are performed such as code splitting where the user only downloads the necessary code to build the requested page and any assets needed to display loading screens, then during idle time or upon navigation download any missing code required to render new pages.

Websites currently are made up of many separate but highly dependant parts:

Database a service which allows for the efficient storage and retrieval of large amounts of data.

View a system responsible for which templates can be defined and then populated depending on the specifics of a given request.

API provides an interface between view layer logic and the database

Authentication system provides methods for confirming a users identity and tracking who is making asking for data in between stateless http requests.

This project will attempt to replace the database, api, and authentication subsystems with a single unified system.

3 Problem Definition

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

There are many issues with this method of building websites.

3.1 Api Repetition

Consistent and discoverable APIs tend to lead to very repetitive code. If, for example, you need an end point to fetch a list of blogs and you also need one to fetch all the products. What is really different about these routes? The table name in the sql and the attributes it returns. This duplication of code leads to more code. More code equals more bugs.

- 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

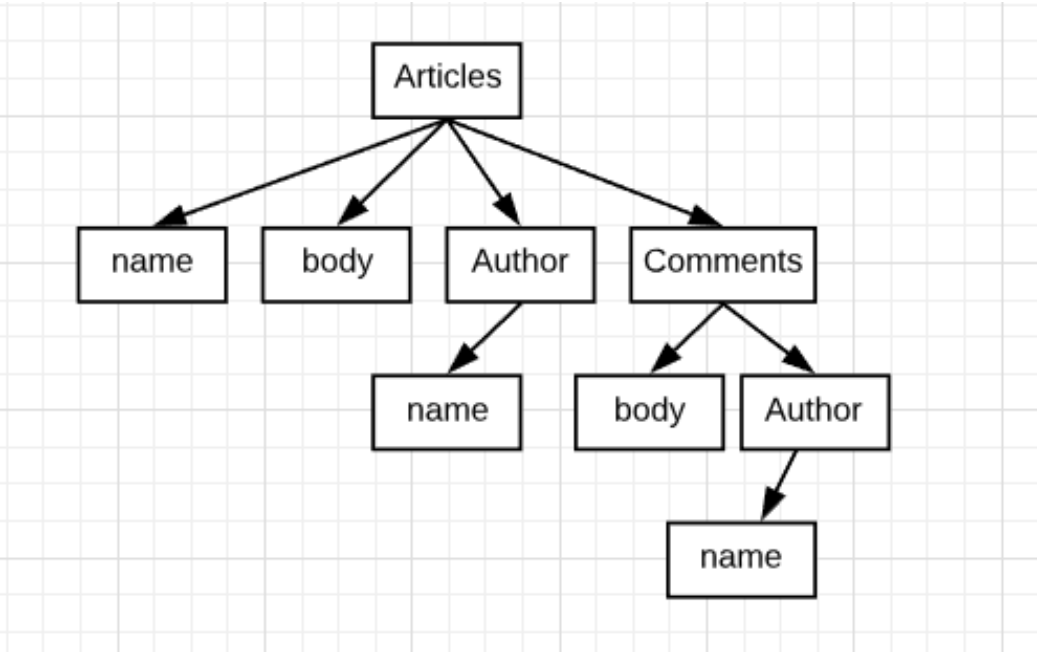
having two end points which return very similar data or reusing the end point but then forcing the end user to download more information then they actually require.

3.2 Storage vs Display

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 numbers of data.

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

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




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

3.3 Tight Coupling

APIs tend to be closely related to the underlying database storing the data. If you have a properties table, then you will want a properties API to access the data. If you were to change the name of a column within the database then you would have to remember to update the API to match and this problem grows if you have tables which span multiple APIs.

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

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

3.4 Authentication

Authentication is a very simple concept in the abstract but is very easy to get wrong.

3.5 Lots Of Boilerplate

Another issue is that there is a lot of boiler plate

- Slow to set all these systems up

3.6 Bespoke Knowledge

- Lots of bespoke knowledge from many domains means it is hard to train people

3.7 Data Consistency

once you have the data it is important to keep it from being stale.

- 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

Although this is not a problem exclusive to SPAs and instead more broadly to websites in general, it is emphasied by SPAs as since they do not need to reload between upon navigation they can potentially cache data more aggressively further improving performance and responsiveness. The risk of stale data, however, greatly limits this potential.

4 Literature Review

In this chapter I shall discuss existing technologies which solve the defined problem. I shall briefly describe how each solution functions as well as their advantages and limitations.

- 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
  - eliminates the tree problem
- Web framework
  - Rails

5 Requirements

5.1 High Level Requirements

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

5.2 Specification Gathering

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

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

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

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

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

        * self.slug = name.parameterize
    - after_create
        * update association
        * send emails
        * update self
- Scopes
    - where(attr: value)
    - where.not(attr: value)
    - order(attr: :desc)
    - where association count >= 1
    - where association count === 0
    - where association attribute
    - where in associations scope e.g where(tag_id: Tag.published)
    - composing scopes (adding limits)
- Permissions
    - through user association
    - through user association | where(attr: value)

Using this information I obtained the minimum viable feature set needed

5.3 Technical Requirements

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

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

- Create Models
    - store basic types strings, number
    - store associations between two models
    - store files
- Fetching
    - attributes
        * Request primitive attributes such as strings, numbers
        * request associations
    - provide a filter to a query
        * request a record given the records id

        * request a record based on its attributes i.e requesting published records
    - sort queries
        * by attributes
        * by associations attributes
    - pagination queries

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

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

- Consistency
    - Use web sockets to be alerted to updates

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

- Provide a way to change production databases safely

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

6 Solution

In order to solve the problem I have created a system named Skeem. This chapter acts as a high level overview to Skeems functionality, and how it is to be used.

Comment:
What is skeem
Send queries from the front end
wraps and maintains a database.
developed a custom query format

Skeem wraps a standard object relational database but augments its functionality by providing a new query interface which is easily sanitizeable and so can be executed directly from the client side. This greatly helps to decouple the API and View layers and the view can directly query for its relevant information. This new query interfaces is designed around tree data structures and so meshes nicely with the needs of the view layer. Skeem fully manages this underlying database and the user is not required to understand how it is layed out or utilized nor do they require any knowledge of SQL to utilize Skeem.

Comment: *auto generate an API based upon the database thus removing some coupling The ability to query from the browser fully eliminates the need for a dedicated API.*

Comment: *built in authentication*

Skeem also provides a full authentication system capable of identifying and tracking users as well as being able limit their access to resources as necessary.

Comment: *Skeem requires no code. no training required.*

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

documentation detailing many aspects.

Comment: *runs a server constantly listening to http requests*

Skeem runs as a service and communication is done via http using a pure json API. This allows browsers to easily communicate with it as all ship with the ability to serialize and parse json data as well as send http requests.

Comment: *websockets for live updates*

Skeem exposes update events through web sockets to enable clients to automatically fetch data when it beomces stale. This allows data to be more aggressivley cached and the interactiveness of sites increased.

6.1 Requests

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

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

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

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

Comment: *TREE OF DATA*

6.1.1 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"

```
{
  articles: {
    attributes: ["name", { name: "body", as: "text"
       }]
  }
}
```

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.

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

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

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.

```
{
  articles: {
```

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

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.

6.1.2 Mutating Data

6.2 Permissions and Authentication

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

6.2.1 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.

6.2.2 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

6.2.3 Permissions

Each model can define a set of permissions based scopes

6.3 Management

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

Through the combination of logically named utilities, help text and descriptive error messages, people can use the system with minimal training. This helps to reduce the need for specialized knowledge

6.3.1 Cli

6.3.2 Gui

6.4 The Client

Skeem is not attempting to replace the view layer of websites. Because of this is has an implementation of a client which can be run on the view layer to wrap the requests into simple function calls as well as adding basic checks.

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.

6.4.1 Caching

6.4.2 Validations

6.4.3 Oauth Flows

6.5 Live Updates

6.6 Plugins

APIs are a large and complex systems which cover an incredibly broad range of use cases. It would be almost impossible to foresee every use case of Skeem

and to allow for every possibility. To cope with this, Skeem has the ability to augment functionality by the way of plugins.

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

6.6.1 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.

6.6.2 Custom Operation Functions

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

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

Creating An Operation

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

Listing 4: 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 5: 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 6: Checks to see if name actually exists on the model being queried.

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

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

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

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

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

Possible types include:

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

Listing 7: Checks to see if the attribute actually exists and returns the correct type

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

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

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

Nested Operations

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

Listing 8: A simple implementation of the eq operaion.

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

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

The Request Context

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

TODO

Using An Operation Plugin

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

Listing 9: 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 }
```

6.7 Documentation

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

7 Methodology

7.1 Development Strategy

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

Listing 10: Example of what the high level tests would assert (not actual tests)

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

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

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

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

8 Usage

9 Implementation

How the system is actually built

9.1 Technologies

10 Testing

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

Tests were written in Jest.

- Tests were written in jest.
- Tests targeted functionality rather then implementation. However tests were written for smaller parts of the system when functionality became too complex or when the underlying functions were critical - such as loading the schema from the database.
- For a complete list of all tests please see the apendix.
- CI
  - Due to skeem being used in production it was essential that it was not only tested but that testing was constantly carried out. By using CircleCI tests are automatically run when a change deployed to the git repository.
- Coverage
  - Code coverage reports show how many lines of the are touched by the tests this is very useful to ensure all the code branches are tested and perform as expected.
  - Code coverage ended up at 46% at the end of the project.

- Code quality
  - Codeclimate is a service which analyzes code and detects "code smells". "A code smell is any characteristic in the source code of a program that possibly indicates a deeper problem." - Wikipedia. This includes problems such as:
    * Cognitive complexity: how complicated is the code to understand.
    * File and function length: does the file or function contain too many lines (only counting actual lines of code, ie. not comments or blank lines)
    * Duplication: are large parts of the code duplicated in multiple places
  - Codeclimate then predicts the amount of time it would take to fix this technical debt. At the project end, skeem contained 147 code smells with a predicted clean up time of 2 months.

11 Deployments

- Throughout the development i has the opportunity to deploy skeem on 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.

12 Conclusion

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

12.1 Future Work

13 References