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.

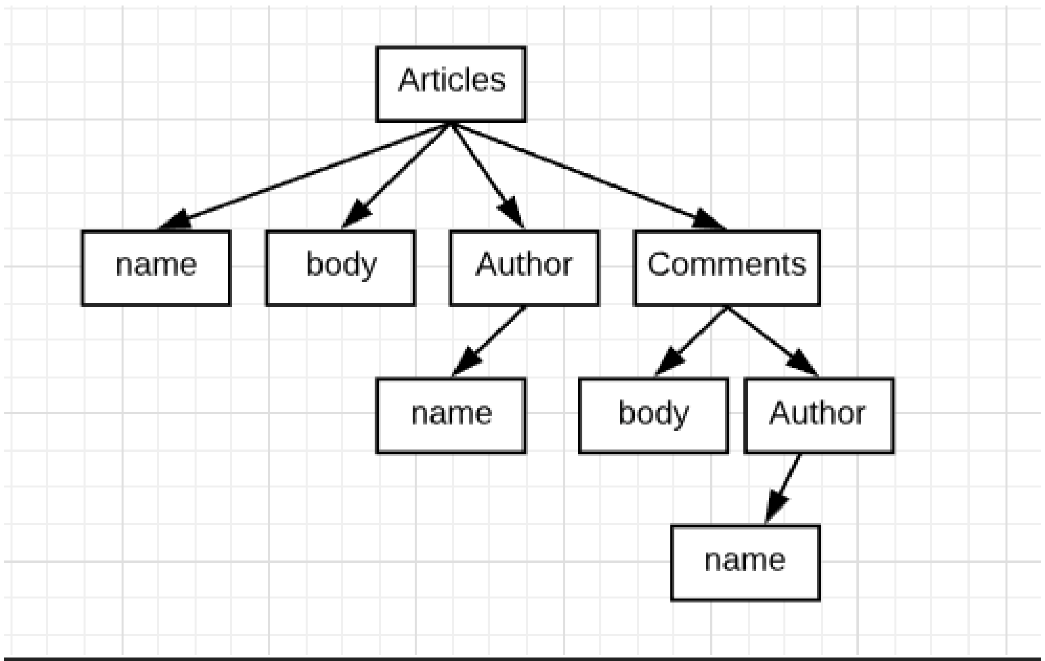
- Duplication of requests. Imagine having an end point requesting a list of blog posts. On the site you wist to display the title of each blog with a short extract from the body to act as a teaser. On this teaser you also want to display the authors name. You may also have a author page showing information 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.

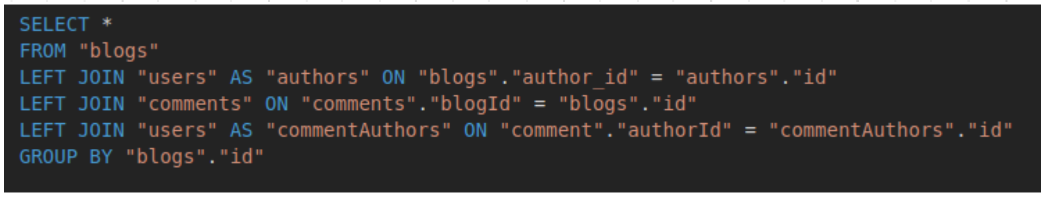
3.2 Storage vs Display

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

To fix these issues I have created a system:

- runs a server constantly listening to http requests
- built in authentication
- wraps and maintains a database.
- auto generate an API based upon the database thus removing some coupling
- Send queries from the front end
- developed a custom query format
- Trees of data to be fetched and mutated
  - Complex filter operations
  - Pagination, sorting, etc...
  - Unified format to improve api consistency
- Plugin system to allow future features

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

```
Given: { articles: {} }
Expected: `SELECT "id" FROM "articles"`

Given: { articles: { attributes: ["name"], filter: {
    eq: [ { attr: 'name' }, {value: 'test'}] } } }
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

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