

Whatever Chat™ Report

CS 291A - Scalable Internet Services

Fall 2019

Primary Project

Jose Acuna

Swaroop Srinivasa Rao

Dongyu Meng

Karl Wang

Contents

1 Introduction	3
2 Features	4
Welcome Screen	4
Signup and Login Page	5
Home Page	5
Individual Post	6
New Post	7
Profile	8
Private Message	9
Post Ad Modal	9
Ad Matching	10
Comments	10
Retwitte(yes, the misspelling is on purpose)	11
3 Load Testing	12
Fine grained seeding	12
Tsung Test Framework Setup	12
Transactions	12
Sessions	13
Session 1: Registration	13
Session 2: Unauthorized User	14
Session 3: Authorized User (Passive)	14
Session 4: Authorized User (Active)	15
Session 5: Ad Owners	15
Scenarios	16
4 Optimization	18
Pagination	18
Indexing	19
N + 1 Query Optimization	19
Active Storage vs DB	22
Vertical Scaling	23
Horizontal scaling	25
5 Conclusion	26

1 Introduction

No one can deny that the past two decades have been the era of social media and there are no signs that it will slow down soon. There have been a plethora of social media apps which have come up over the years, including Facebook, Twitter, Reddit and Instagram. In particular, Facebook had an estimated 2.45 billion monthly users worldwide in the month of September in 2019. Furthermore, Facebook averages 1.62 billion active users (people who log in to the site everyday). With these huge and unprecedented amount of traffic that these social media sites serve everyday, they face the tricky and complicated tasks of scaling and high availability. Thus, when services fail to keep up with the demand, we see common hashtags such as #InstagramDown and #TwitterBlackOut.

Our application is WhateverChat - which has a subset of features taken from all these major social media apps. In this report, we'll introduce the features we implemented in section 2. However, an objective for this project is not on how many features can we implement, but how can we make the application scale better, thus improving performance. So in section 3 we introduce how we utilize Tsung to perform tests on our application and the significance of the tests. Next, in section 4 we describe the optimizations we have implemented and explore their performance results. Finally, we discuss conclusions about the project and lessons learned in section 5.

2 Features

The application has features which enable users to create new posts, view posts made by other users, comment/like/dislike individual posts, retweet a post, view all posts made by a particular user, send private messages, post ads to the site. In this section, we introduce each feature and provide screenshots of said feature. The services of Whatever Chat utilize web technologies such as Ruby on Rails, PostgreSQL, Bootstrap, React, and AWS Elastic Beanstalk.

Welcome Screen

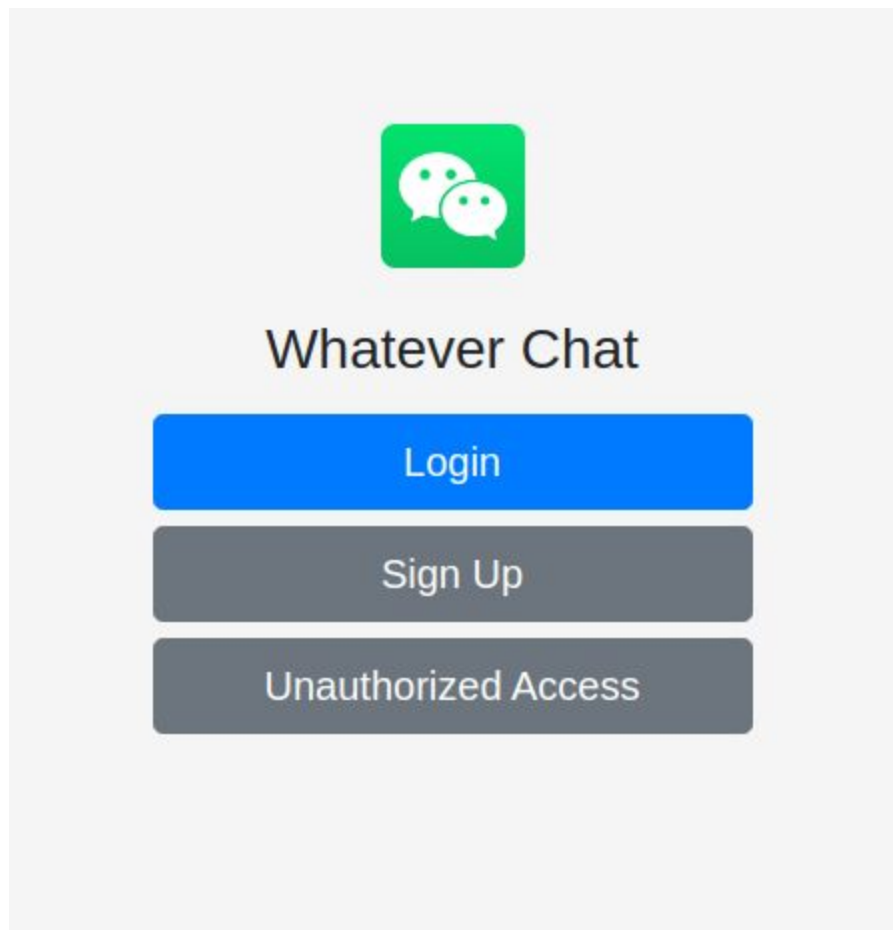


Fig 2.1 Whatever Chat welcome screen

Any user that wishes to use our service will first come to the welcome screen or splash page. Here, existing users can login using their credentials, new users can sign up. Alternatively, a user may enter as unauthenticated user, where the functionality of the site will be restricted to merely viewing and making posts that are visible to all users.

Signup and Login Page

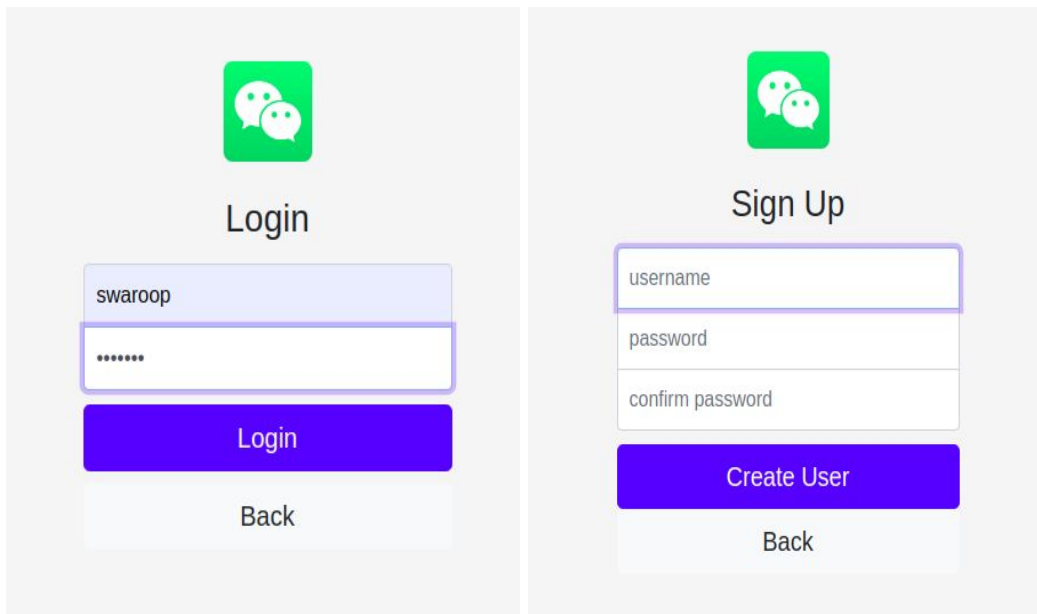


Fig 2.2 Whatever Chat sign up and login page

In order to be an authenticated user, one must either login or sign up through the above screens. Uniqueness of the users are maintained by each user mandatorily having a different username. New users set their passwords which are salted, hashed, and stored in our database. Existing users are authenticated when the provided password's hash matches that in the database. Once authenticated, the user will be provided a token that must be provided on subsequent requests.

Home Page

The homepage is at the heart of our application. All the links to any other action can be obtained from this page. It is split into three sections - header, message board, and ads strip. The header contains the links to other pages that a user might have access to. Authenticated users can see their profile, check out private messages, post an ad or logout. These actions are labeled as “Profile, Private Messages, Post Ad, Logout” respectively in Figure 2.3.. Unauthenticated users will see a similarly styled header, except the only action they may take from the header is to return to the welcome screen via a “Home” button that replaces the “Logout” button.

The next section is the message board and appears in the center of the screen. This is similar to a Facebook wall or timeline except that these posts are made visible to all users. Here, a user may hit the purple button to make a new post. Otherwise, they may view or interact with existing posts shown in reverse chronological order.

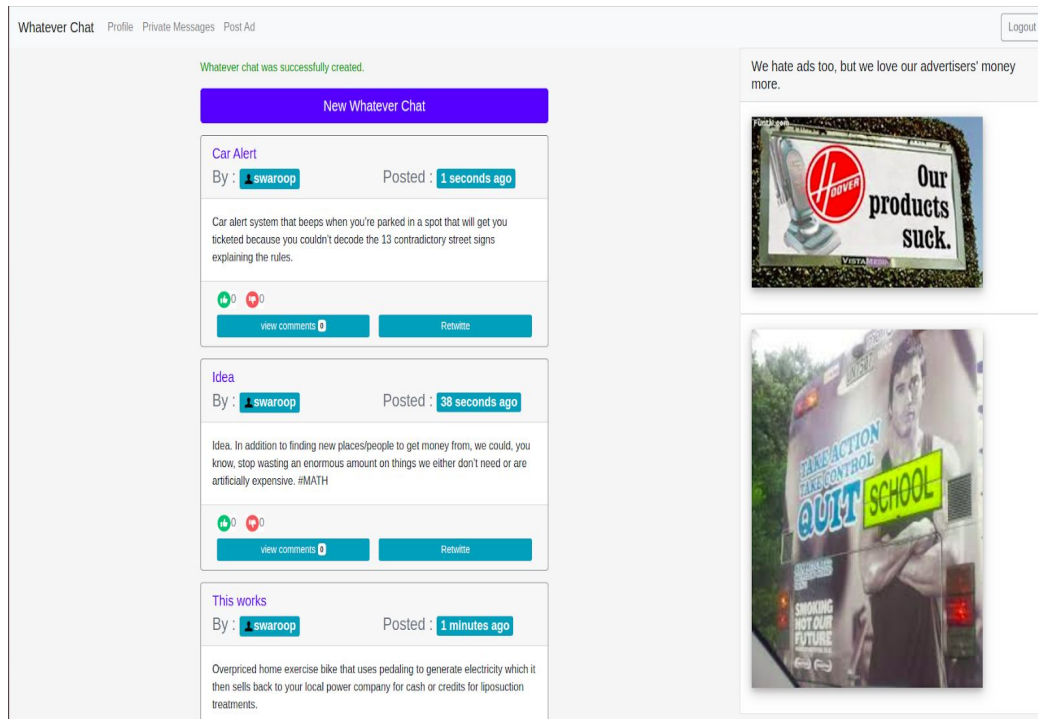


Fig 2.3 Whatever Chat homepage (index page)

Finally, our venture has failed to receive any funding so we instead monetize our site using ads that target the interests of the user. These ads appear in the ad strip on the right side of the page.

Individual Post

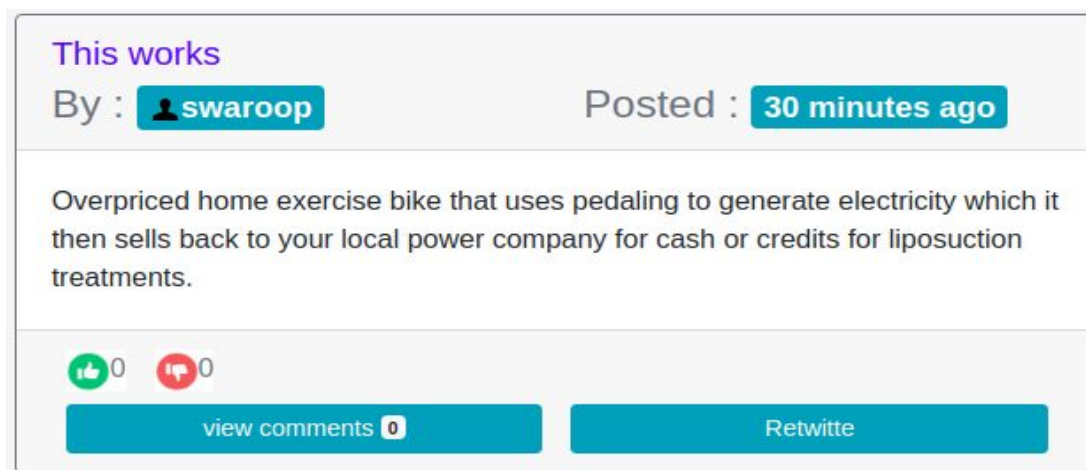



Fig 2.4 Whatever Chat example post

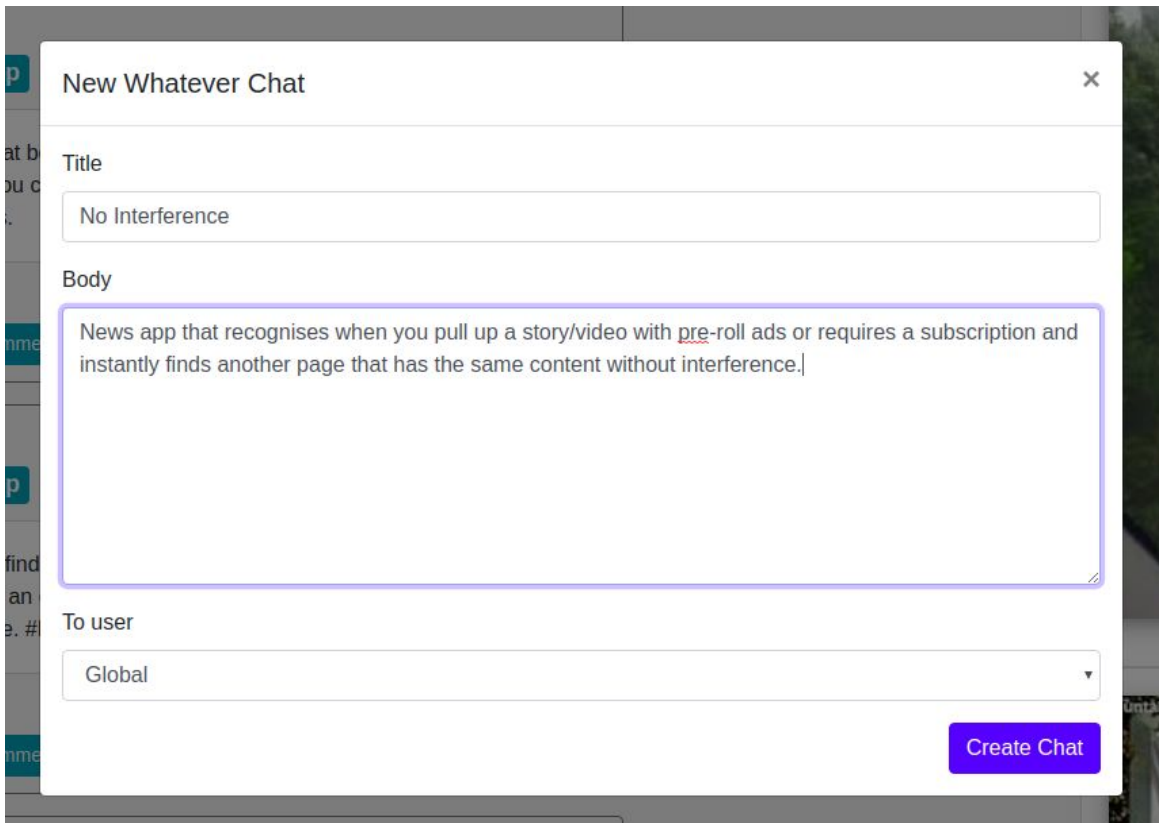
This is an individual post which is embedded in a Card Bootstrap styling. The post contains a list of information:

- The title of the post at the top
- The user who authored the post
- When the post was made
- The actual content of the post
- Like/Dislike Counter
- Comments counter / link to view comments
- Option to Retwitte

It's worth noting that unauthorized users can also make posts. When they do, they choose an alias to associate with the post. However, you can tell a post was made by an authenticated user if you can see an avatar icon  next to their username. When the username is clicked, one is redirected to the profile page corresponding to the username. A user may also click on the title of the post or “view comments” button to see the full post including its comments.

Finally, the timestamp shown on a post is relative to when the post is requested and when it was made. The granularity starts at “seconds ago” and ends at “weeks ago”.

New Post



The image shows a web form titled "New Whatever Chat" with a close button (X) in the top right corner. The form contains three main sections: a "Title" field with the text "No Interference", a "Body" text area containing the text "News app that recognises when you pull up a story/video with pre-roll ads or requires a subscription and instantly finds another page that has the same content without interference.", and a "To user" dropdown menu currently set to "Global". A blue "Create Chat" button is located at the bottom right of the form.

Fig 2.5 Whatever Chat example new post

A Bootstrap modal popup is shown when user wants to make a new post. The modal popup option is used so as to give the user a feel of Single Page Application. The form contains the title and the body of the post to be made.

The option 'To user' at the bottom shows whether the post is to be made globally (which can be seen by everyone) or to a particular user (private message). The 'To user' dropdown will be auto-populated with all the users currently in our system, but with the Global option selected by default. If a user is unauthenticated, then their forced to post to Global. Once a post is made, the modal is closed and the home page is refreshed so that new Global posts are made visible.

Profile

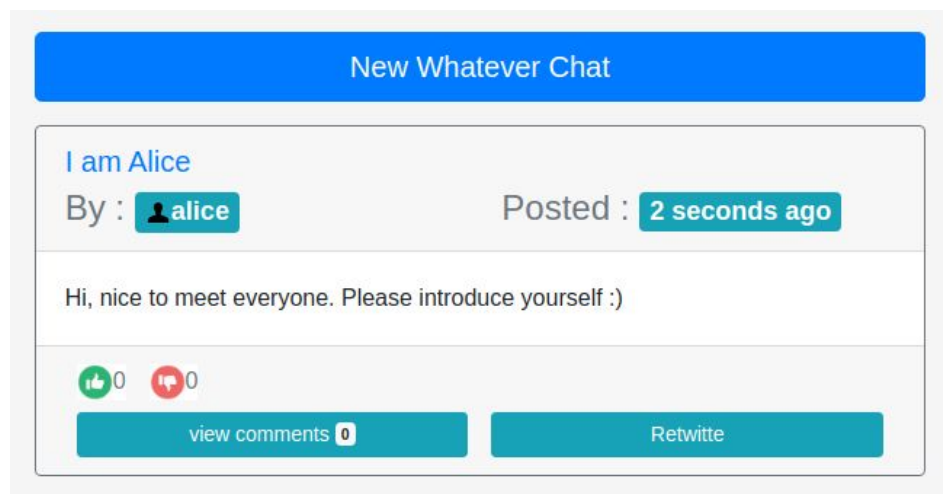


Fig 2.6 Whatever Chat profile demonstration

The profile page shows posts made by yourself. In this case, the user is logged in as Alice so the profile page shows all posts Alice made. The layout of this page is however similar to the homepage to maintain consistency across all the application.

Private Message

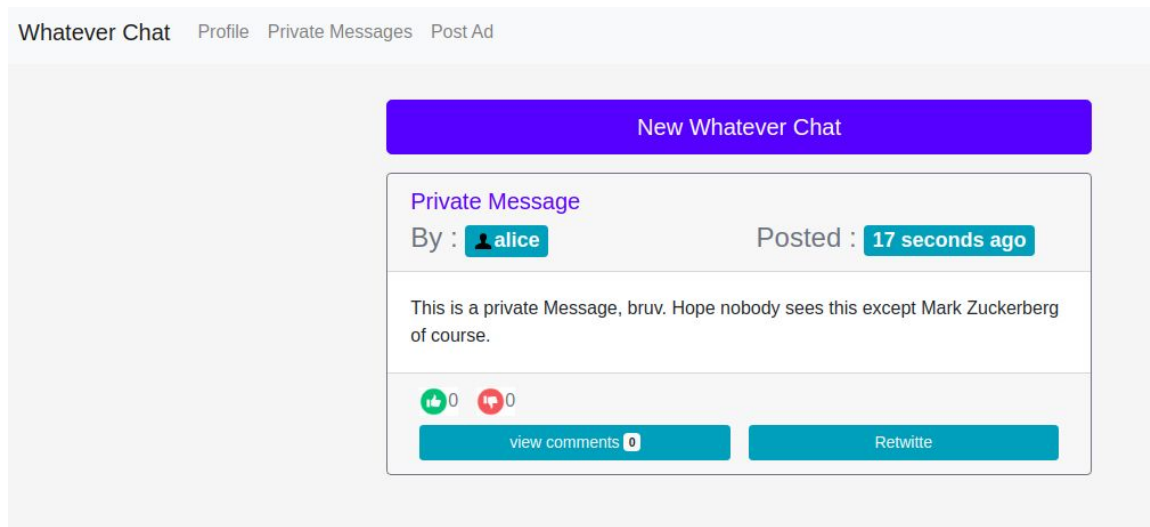


Fig 2.7 Whatever Chat private message demonstration

On navigating to the “Private Messages” tab from the header, you can view all the posts which were sent particularly to you.

Post Ad Modal

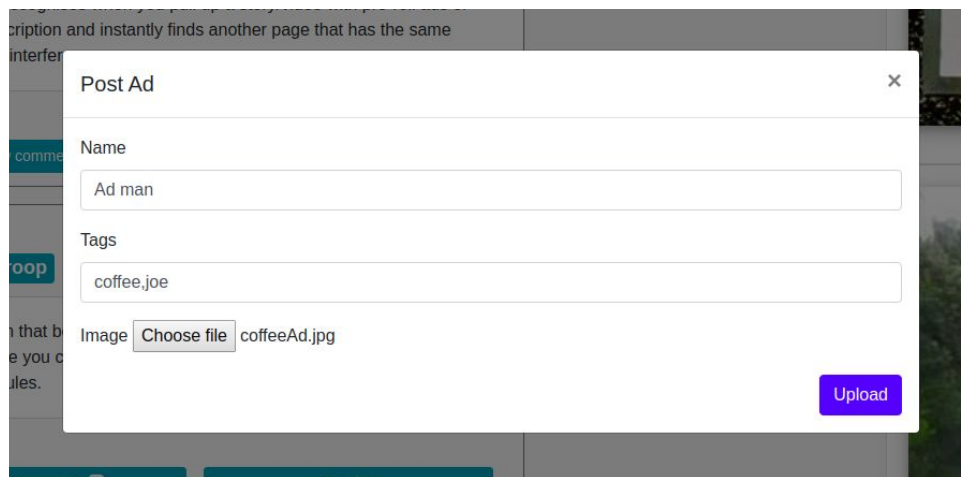


Fig 2.8 Whatever Chat Ad posting (modal)

Users can also post ads in our application to be served to other users based on their interests. An image for the ad is uploaded from their local storage which will then be rescaled before being displayed to other users. The tags are a comma separated list of words which are used to target particular users. It is up to the poster to enter tags that correspond to their image or marketing message.

Ad Matching

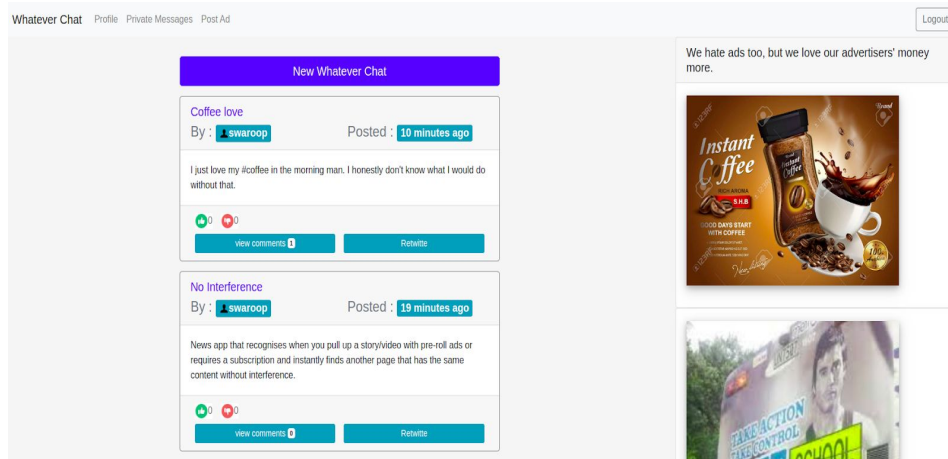


Fig 2.9 Whatever Chat Ads showing demonstration

The first ad on the right is an ad for coffee. In the example, the user made a post “I just love my #coffee in the morning”. At which point, the service keeps track of tags users have generated by using a # in their post’s body. Thus, our ad matching logic will match the users’ tags with an ads tags and then serve the correct ad image.

Comments

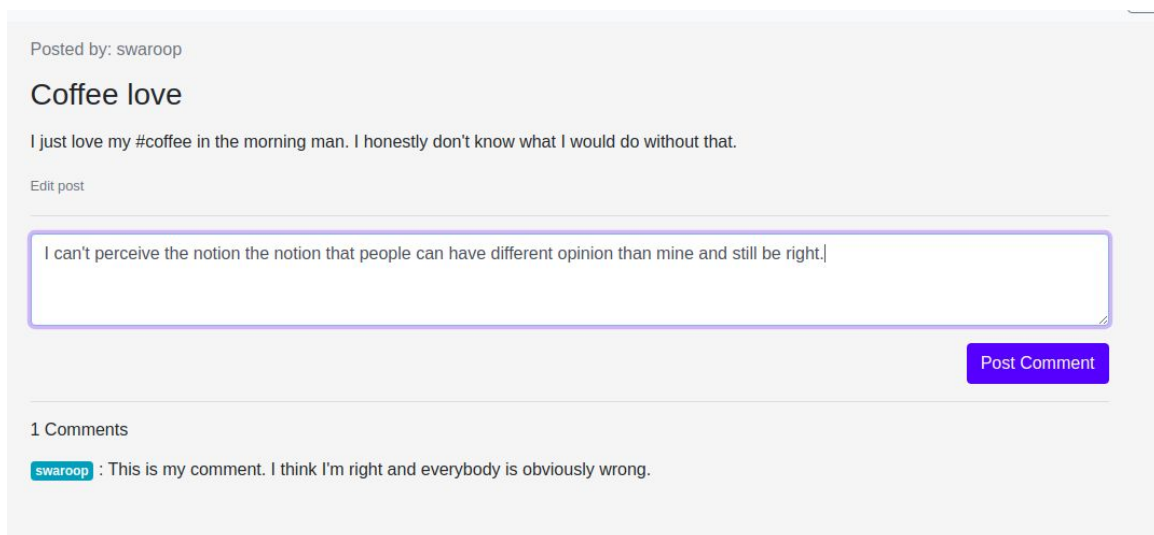


Fig 2.10 Whatever Chat commenting demonstration

Authorized users may post comments to individual posts and also view comments made by other users. Unauthorized users can only view the comments.

Retwitte(yes, the misspelling is on purpose)

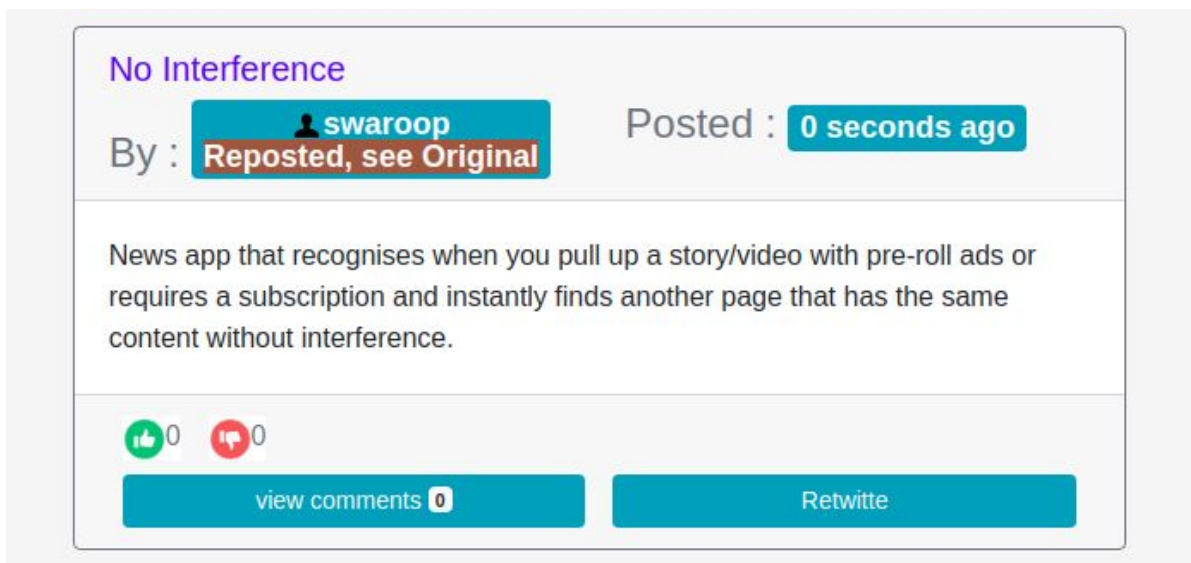


Fig 2.11 Whatever Chat “retwitte”

This is a “retwitted” post. The post is marked as “Reposted” and you can also view the original post by clicking on the ‘see Original’ link. All retwitted posts are treated as new posts made at the time of the retwitte and can only be posted on the global message board.

3 Load Testing

Fine grained seeding

To better test the server under different circumstances, we used different seed sizes (number of initial records). More specifically, we wrote a custom rake (rails) task to facilitate three seeding scripts. The scripts map to small, medium, and large seed size respectively. We use small seed size while developing and switch to large seed(10000 users and posts, 100000 comments and votes) when performing the final test. We leveraged rails' new **bulk insert feature** and managed to finish this process in seconds.

Tsung Test Framework Setup

To stay flexible with our testing infrastructure, we used a three-layer abstraction to describe the Tsung tests we ran. Namely, we had **transactions**, **sessions**, and **scenarios**.

A **transaction** is just a transaction in Tsung world. It serves as a primitive of user behavior, and can be reused across different user patterns. Examples of transactions include *Login*, *Welcome*, and *Registration*. For this project, we defined 17 transactions.

A **session** describes a typical user behavior pattern. It comprises one or more transactions. To this end, we defined 5 sessions: *Registration*, *Unauthorized User*, *Passive Authorized User*, *Active Authorized User*, and *Ads Owner*. A session here maps to a session in Tsung.

A **scenario** formalizes a traffic pattern of the website, which results from different make up of user patterns. Therefore, a scenario naturally has one or more sessions. Each session is assigned with a probability that describes the proportion of the user pattern it represents. We defined 5 scenarios to explore server performance from different perspectives. A scenario corresponds to a Tsung configuration file.

Implementation-wise, we wrote each transaction individually and scribed a bash script to auto generate sessions and scenarios. In this way, we are able to iterate our definition of sessions as well as scenarios basically by modifying a shell array.

Now we introduce the three abstraction layers in more details.

Transactions

We defined the following 17 transactions as primitives: *comment*, *downvote*, *inbox_post*, *index_page*, *login*, *new_post*, *post_ad*, *post_with_tags*, *profile*, *registration*, *retwitte*, *think*, *to_me*,

upvote, *view_chat*, *view_chat_pages*, *welcome*. It covers roughly all the possible interactions a user could have with our application. Thus, sessions and scenarios just cover many permutations of the transactions. For some of the transactions, we bundle several consecutive actions into one transaction (*upvote* actually votes 5 times on different posts).

As most of the transaction names are quite intuitive, we will not go into gory details of what each of the transactions is. Instead, we use one example here to demonstrate some catches and cool features we noticed / used when implementing these transactions in Tsung.

To start with, we noticed that unlike a browser, Tsung does not really do much with the responses. It does not automatically issue requests for resources like images, does not run Ajax, and provides no notion of client caching. Therefore, to mimic what clients would actually experience with a browser, we have to implement some of the mechanisms ourselves.

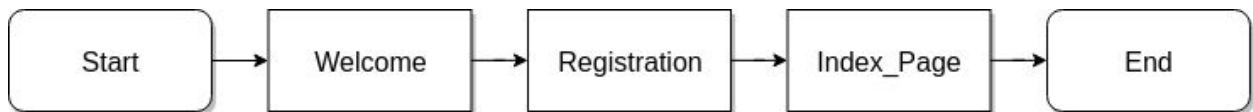
More specifically, we leveraged **tsung-recorder** to inspect which requests should actually be made, and based our transactions on the recording. For example, we show two ads on most of our pages, but Tsung is not able to make these requests itself. We end up using regex matching on server response and issuing separate ad requests explicitly.

Apart from this, we managed to upload images with Tsung as well by storing the bin file of the *tsung-recorder* and resending that file in our transaction.

Sessions

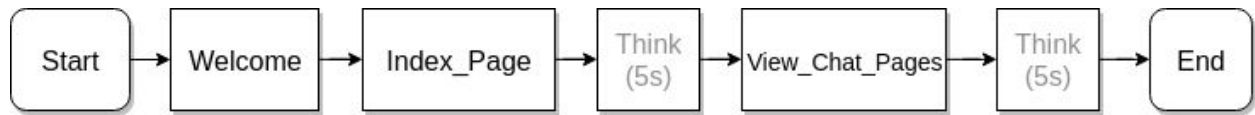
Session captures user behavior patterns. We used the following 5 sessions in our project. Note in the flowchart each operation(square) is a transaction.

Session 1: Registration



The rationale for having registration alone as a session is that registration is independent from normal user activities. Therefore, by having registration as an independent session and tune the probability properly, we are able to mimic new user behavior by combining registration session and normal user behaviour sessions.

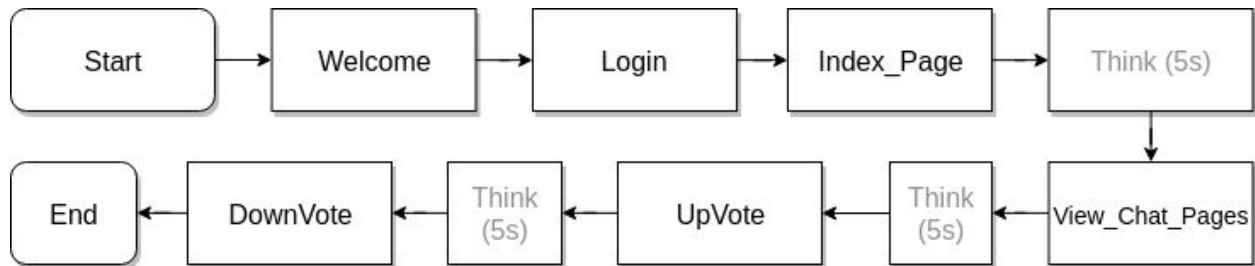
Session 2: Unauthorized User



For many User Generated Content (UGC) info-sharing apps, unregistered users can take up more than half of the traffic. These users do not contribute content to the app, nor do they voice their opinions by voting or commenting. In other words, they are ‘read-only’(wink).

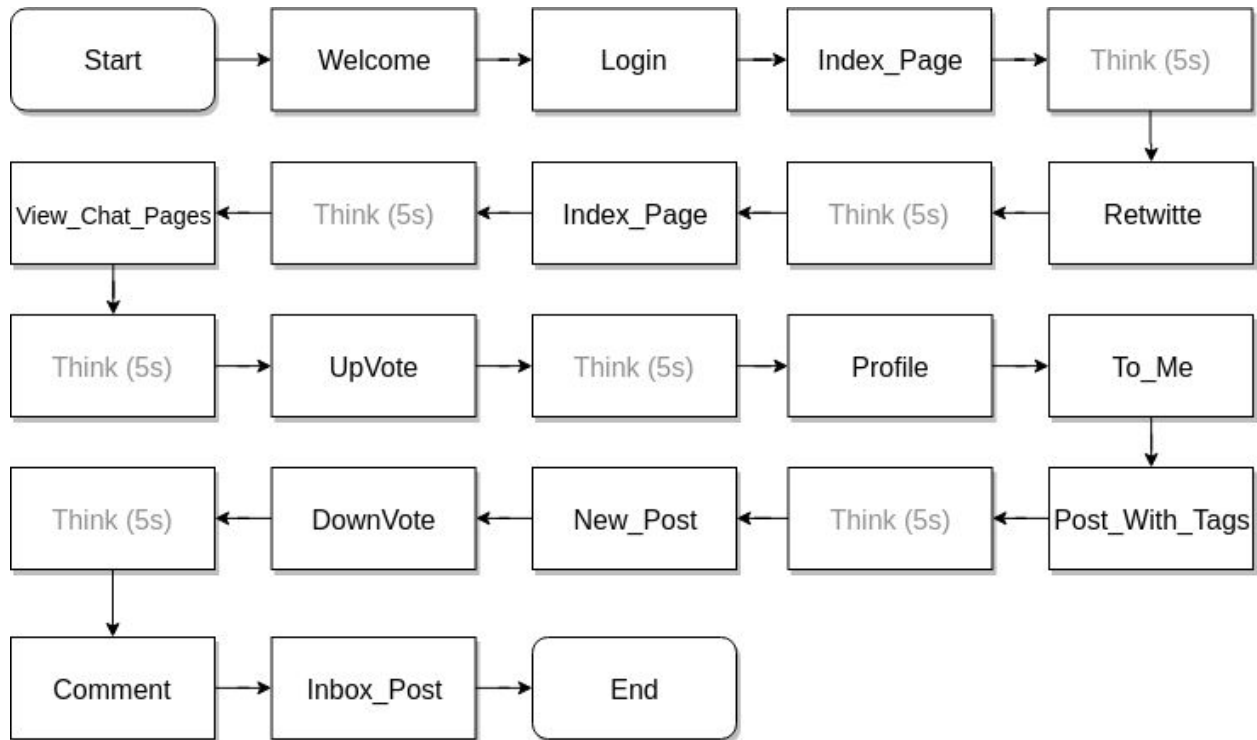
In WhateverChat, these users go through the unauthorized channel.

Session 3: Authorized User (Passive)



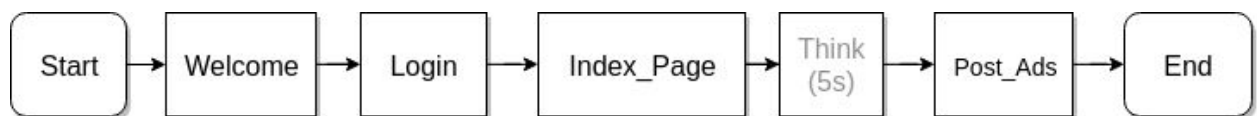
This type of users also take up massive amount of our backend processing and traffic. These passive users usually just view posts, upvote/ downvote individual posts, go to different pages and view more posts.

Session 4: Authorized User (Active)



Active (authorized) users are the backbone of a websites. They contribute content, browse existing content, and connect with each other. As the flowchart above indicates, they spend quite some time on our site and do basically everything possible. We would like to note again here, some of the transactions repeat the action several times (e.g. Comment transaction - comments in 5 posts).

Session 5: Ad Owners



One unique (funny) feature for our site is ads. Ads owners can upload new advertisements (image and corresponding tags) through a designated portal. The pattern for ads owners are different from normal users. They are less active as well.

Scenarios

Scenarios describe site traffic patterns through the lens of user make up. We came up with 5 scenarios to maximize variety . Not all of the scenarios are realistic.

The 5 scenarios are:

Scenarios	Registration	Unauthorized User	Authorized User (Passive)	Authorized User (Active)	Ad Owners
Scenario 1	100%	-	-	-	-
Scenario 2	-	100%	-	-	-
Scenario 3	-	50%	50%	-	-
Scenario 4	5%	40%	35%	20%	-
Scenario 5	5%	40%	30%	20%	5%

Among these, we consider scenario 5 as the most realistic one. We will discuss the traffic pattern of each of the scenarios as experiments unfold.

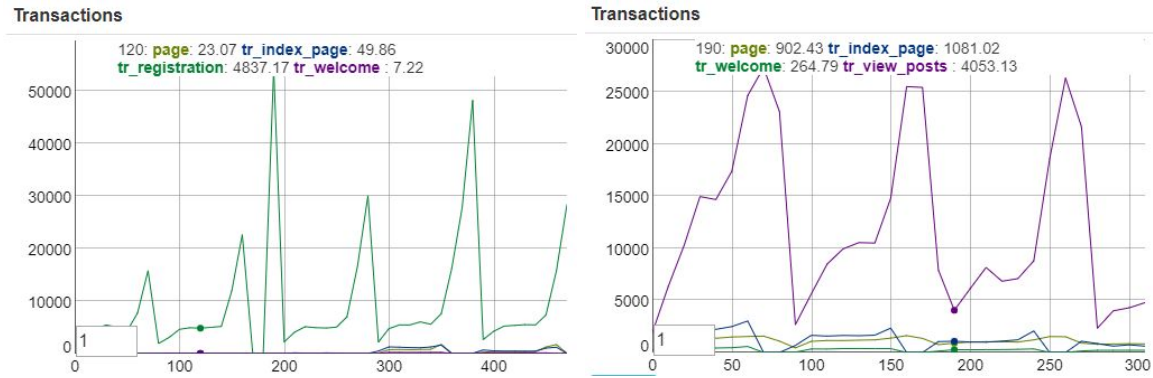


Fig 3.1 Scenario 1 (left) and scenario 2 (right) - transaction times

Transactions

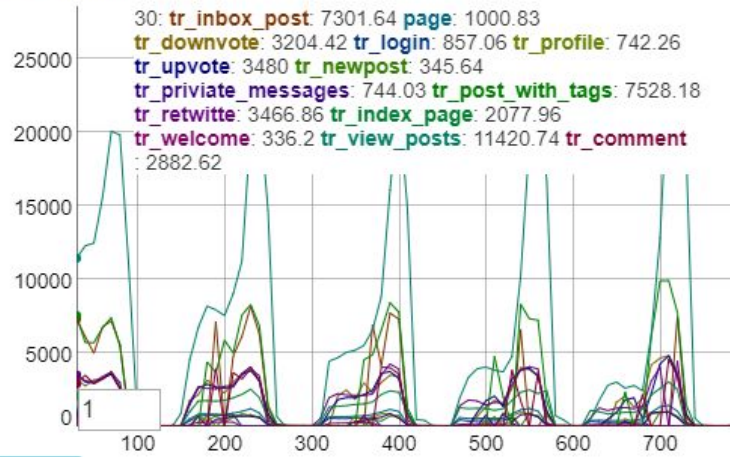


Fig 3.2 Scenario 3 transaction times

Transactions

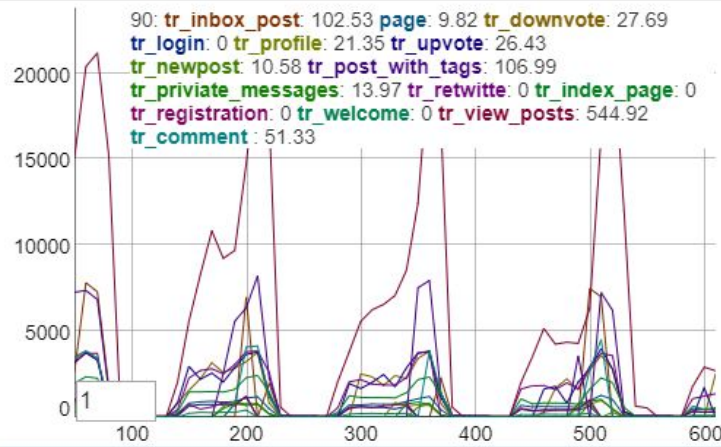


Fig 3.3 Scenario 4 transaction times

Transactions

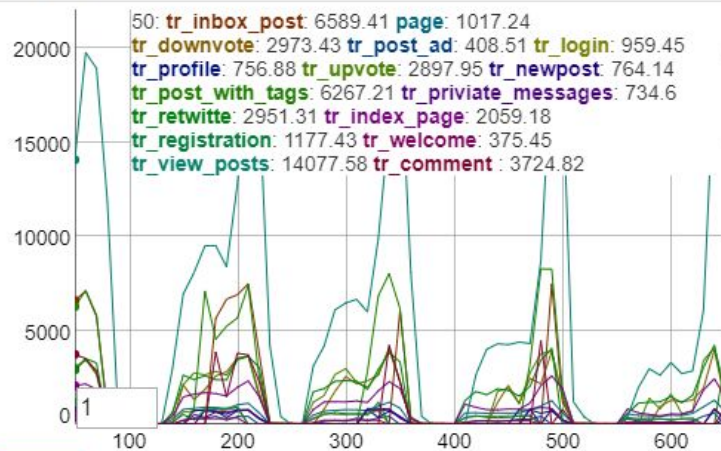


Fig 3.4 Scenario 5 transaction times

4 Optimization

In this section, we describe the improvements we made to get better performance of our application. We also include corresponding graphs of our application running with and without the optimization to compare the performance. This data comes from experiments done using Tsung as described in the previous section.

Pagination

Over the course of Tsung load testing, the index page of our application will end up fetching many posts of which a user might view only a handful of posts at a time. The overhead of getting unnecessary data may render the system slow or even unusable. To solve the problem, we used Pagination which gets only 10 posts and its related content per page, drastically reducing the DB Query time and render time per page wherein it now just needs to get the top 10 results from the database. Additionally, this will also reduce the initial page load time and size of data sent over the network. The following graph shows the dramatic increase in requests served due to pagination.

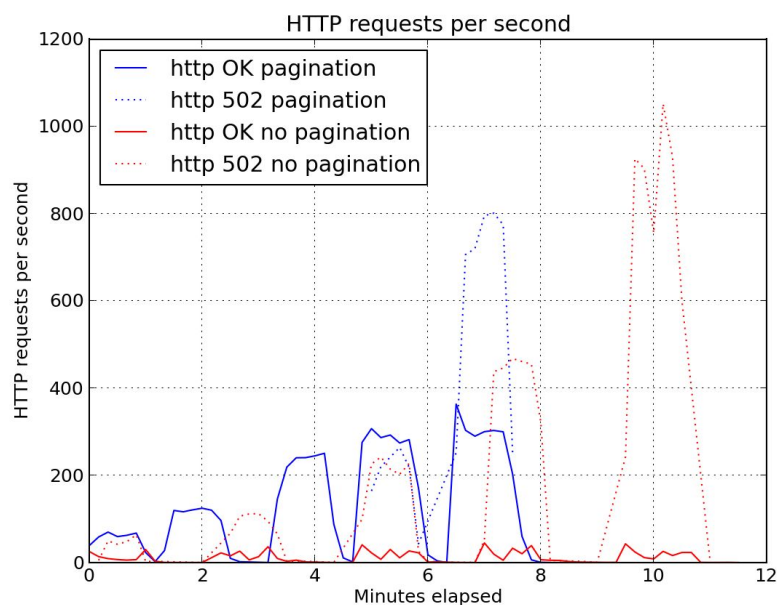


Fig 4.1 HTTP response status with and without pagination

As we can see from Figure 4.1, the number of HTTP 200 responses received from the server are considerably higher with pagination than without pagination.

Indexing

By default, Rails just indexes the default id column of each of the tables created. This index individually simply does not suffice. In fact, we need to create an index on columns which are used for foreign key references, as these columns are used in joins, and adding indexes on these foreign keys will improve performance.

We ran scenarios 4 on both indexed and non-indexed version of our app. The following graph shows how the transaction duration of transaction `index_page` improve with the introduction of indexing. For all five phases we considered, the indexed version outperformed the non-indexed version by at least 4x.

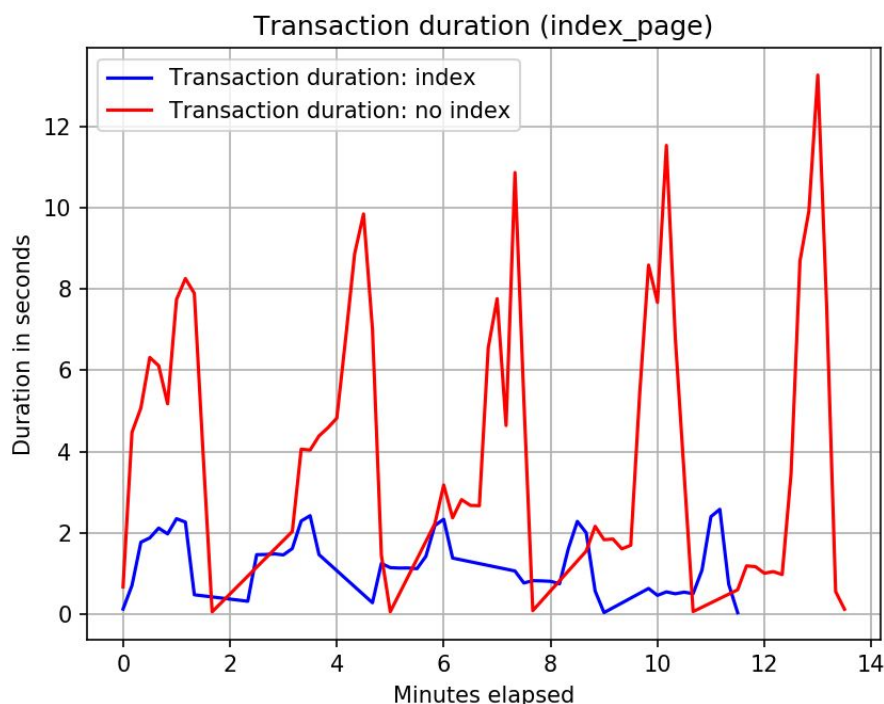


Fig 4.2 Transaction duration of transaction `index_page` with and without indexing

N + 1 Query Optimization

This is a common issue with Rails application, where there'll be independent queries to the same table even though all of them could be grouped together to fetch all at once.

For example, In our application, the index page makes a request to get all the posts, but separate queries are executed to load each of the votes, comments per each post. This is highly unnecessary as all the comments and likes/dislikes can be loaded with a single query using preloading.

```

web_1 | Rendering whatever_chats/index.html.erb within layouts/application
web_1 | User Load (0.7ms) SELECT "users".* FROM "users"
web_1 | ↳ app/views/whatever_chats/_form.html.erb:27
web_1 | Rendered whatever_chats/_form.html.erb (Duration: 9.0ms | Allocations: 1638)
web_1 | Rendered ads/_form.html.erb (Duration: 3.6ms | Allocations: 785)
web_1 | WhateverChat Load (0.6ms) SELECT "whatever_chats".* FROM "whatever_chats" WHERE "whatever_chats"."from_user_id" = $1 ORDER BY created_at DESC LIMIT $2 OFFSET $3 [{"from_user_id", "swaroop"}], [{"LIMIT", 10}, {"OFFSET", 0}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:98
web_1 | Vote Load (0.5ms) SELECT "votes".* FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."username" = $2 LIMIT $3 [{"whatever_chat_id", 22}, {"username", "swaroop"}, {"LIMIT", 1}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:149
web_1 | (1.3ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 22}, {"att", "up"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:155
web_1 | (0.8ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 22}, {"att", "down"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:156
web_1 | (2.1ms) SELECT COUNT(*) FROM "comments" WHERE "comments"."whatever_chat_id" = $1 [{"whatever_chat_id", 22}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:163
web_1 | Vote Load (0.4ms) SELECT "votes".* FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."username" = $2 LIMIT $3 [{"whatever_chat_id", 21}, {"username", "swaroop"}, {"LIMIT", 1}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:149
web_1 | (0.6ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 21}, {"att", "up"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:155
web_1 | (0.5ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 21}, {"att", "down"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:156
web_1 | (1.1ms) SELECT COUNT(*) FROM "comments" WHERE "comments"."whatever_chat_id" = $1 [{"whatever_chat_id", 21}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:163
web_1 | Vote Load (0.3ms) SELECT "votes".* FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."username" = $2 LIMIT $3 [{"whatever_chat_id", 20}, {"username", "swaroop"}, {"LIMIT", 1}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:149
web_1 | (0.6ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 20}, {"att", "up"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:155
web_1 | (0.9ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 20}, {"att", "down"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:156
web_1 | (0.9ms) SELECT COUNT(*) FROM "comments" WHERE "comments"."whatever_chat_id" = $1 [{"whatever_chat_id", 20}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:163
web_1 | Vote Load (0.6ms) SELECT "votes".* FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."username" = $2 LIMIT $3 [{"whatever_chat_id", 18}, {"username", "swaroop"}, {"LIMIT", 1}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:149
web_1 | (0.5ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 18}, {"att", "up"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:155
web_1 | (0.4ms) SELECT COUNT(*) FROM "votes" WHERE "votes"."whatever_chat_id" = $1 AND "votes"."att" = $2 [{"whatever_chat_id", 18}, {"att", "down"}]
web_1 | ↳ app/views/whatever_chats/index.html.erb:156

```

Fig 4.3 Index page db queries without query optimization

You can see votes and comments table being referenced multiple times. This can quickly exhaust the DB Connection pool.


```

web_1 | WhateverChat Load (4.2ms) SELECT "whatever_chats".* FROM "whatever_chats" WHERE "whatever_chats"."to_user_id" = $1 ORDER BY whatever_chats.created_at DESC LIMIT $2 OFFSET $3 [{"to_user_id", "0"}], [{"LIMIT", 10}], [{"OFFSET", 0}]
web_1 | ↳ app/controllers/whatever_chats_controller.rb:29:in `index'
web_1 | Comment Load (1.8ms) SELECT "comments".* FROM "comments" WHERE "comments"."whatever_chat_id" IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10) [{"whatever_chat_id", 22}, {"whatever_chat_id", 21}, {"whatever_chat_id", 20}, {"whatever_chat_id", 19}, {"whatever_chat_id", 18}, {"whatever_chat_id", 17}, {"whatever_chat_id", 16}, {"whatever_chat_id", 15}, {"whatever_chat_id", 14}, {"whatever_chat_id", 13}]
web_1 | ↳ app/controllers/whatever_chats_controller.rb:29:in `index'
web_1 | Vote Load (2.1ms) SELECT "votes".* FROM "votes" WHERE "votes"."whatever_chat_id" IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10) [{"whatever_chat_id", 22}, {"whatever_chat_id", 21}, {"whatever_chat_id", 20}, {"whatever_chat_id", 19}, {"whatever_chat_id", 18}, {"whatever_chat_id", 17}, {"whatever_chat_id", 16}, {"whatever_chat_id", 15}, {"whatever_chat_id", 14}, {"whatever_chat_id", 13}]
web_1 | ↳ app/controllers/whatever_chats_controller.rb:29:in `index'

```

Fig 4.4 Index page db queries with query optimization

You can see that all the votes and comments retrieved by the database in one go.

We also tested the performance change under Tsung’s load. With our working deployment, one page only has 10 posts, making it hard to demonstrate the performance boost from query optimization. Therefore, here we change the number of posts to 100 just for demonstration. As shown in Figure 4.5, the response time for index page for non-optimized version is at least 3 times longer than the optimized version.

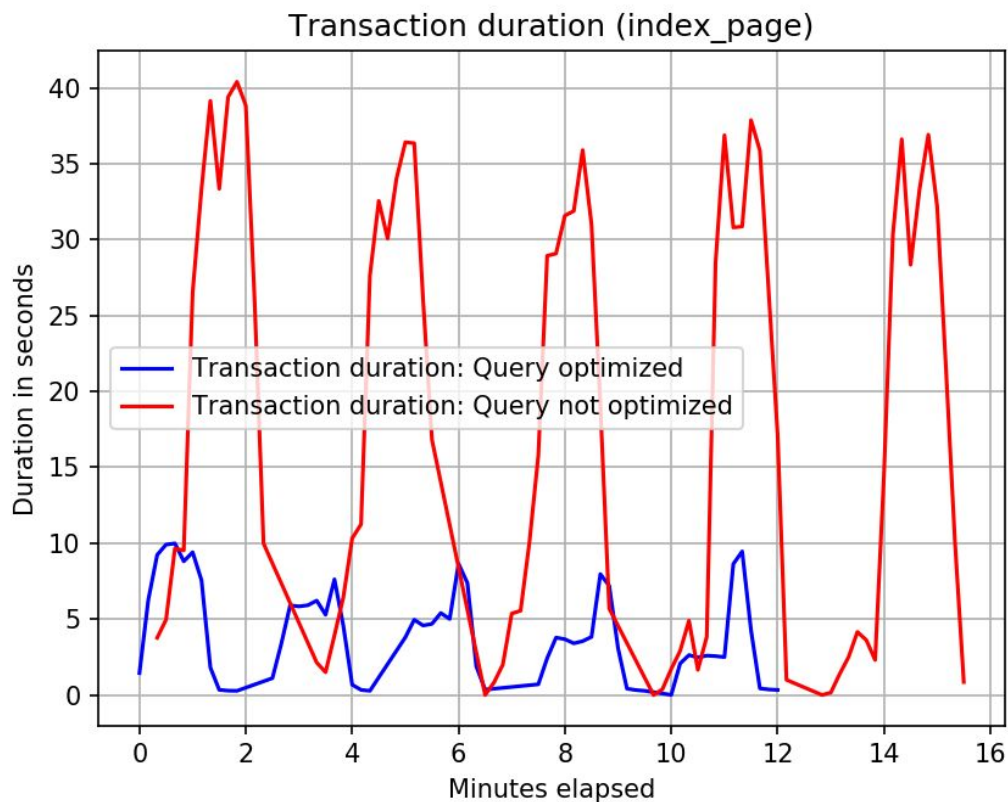


Fig 4.5 Transaction duration of transaction *index_page* with and without query optimization

Active Storage vs DB

When storing ads images, we came across two possible solutions. One is to store images in database and manage them with normal object management logic, the other is to use Active Storage by rails. Active Storage supports most major file storage platforms and facilitates many basic image processing functionalities. In addition, Active Storage provides a static url for each image, which helps client-side caching. We were curious about the performance difference between the two and compared them on scenario 4.

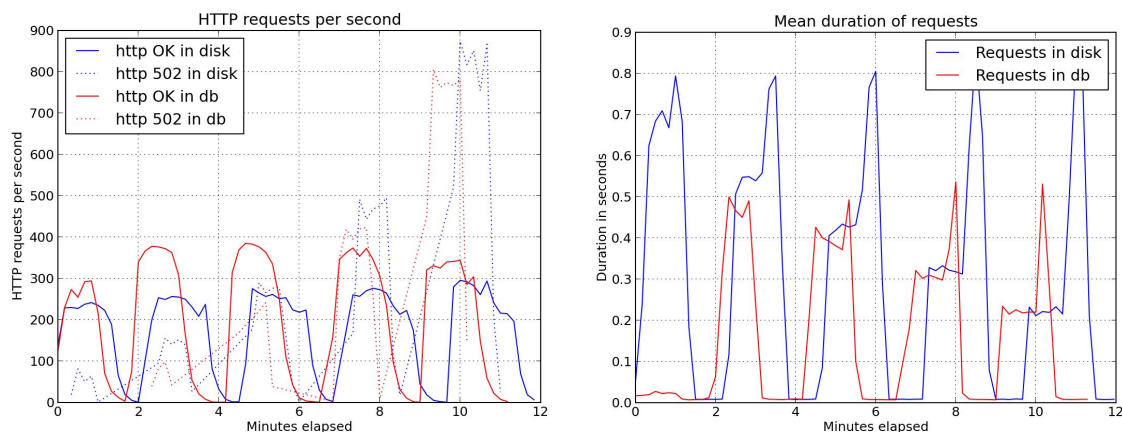


Fig 4.6 HTTP response status code and mean duration of requests with images stored in Active Storage and images stored in db

From the above graph we see although Active Storage is somewhat disk-based, it was not outperforming the db solution. On the contrary, the db solution was faster in most cases. We do note here that Tsung does not provide any notation of client-side caching so this result might be biased.

Vertical Scaling

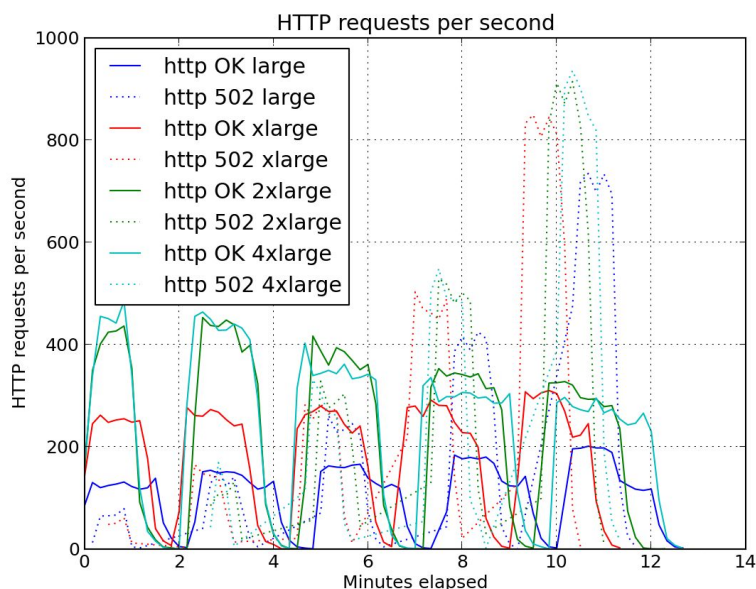


Fig 4.7 HTTP response status with c5 instance types

The easiest way to scale an application is often to increase the resources available on the application server. However, as shown in our test, it is not the most efficient way to scale our app. We tested scenario 4 on three different instance types: c5.large (\$0.085/hr), c5.xlarge (\$0.17/hr), c5.2xlarge (\$0.34/hr), and c5.4xlarge (\$0.68/hr).

We observed significant improvement moving from large to xlarge and large to 2xlarge. This is expected, as each upgrade doubles CPU core count and memory size. However there is almost no difference between 2xlarge and 4xlarge. We suspect this is due to the max thread limit of 5 set on puma, which prevents rails from fully utilizing the 16 cores offered by c5.4xlarge.

We also observed that for bigger instance types, 200 response decreases as the load increases. In contrast, 200 response from smaller instance types increases as the load increases. Given the extremely long test cases in scenario 4, we hypothesize that as the load increases, large instance types have more opportunity to get stuck in CPU intensive or write intensive tasks later in the test sequence, neglecting smaller tasks in the process. This results in a lower amount of 200 responses overall.

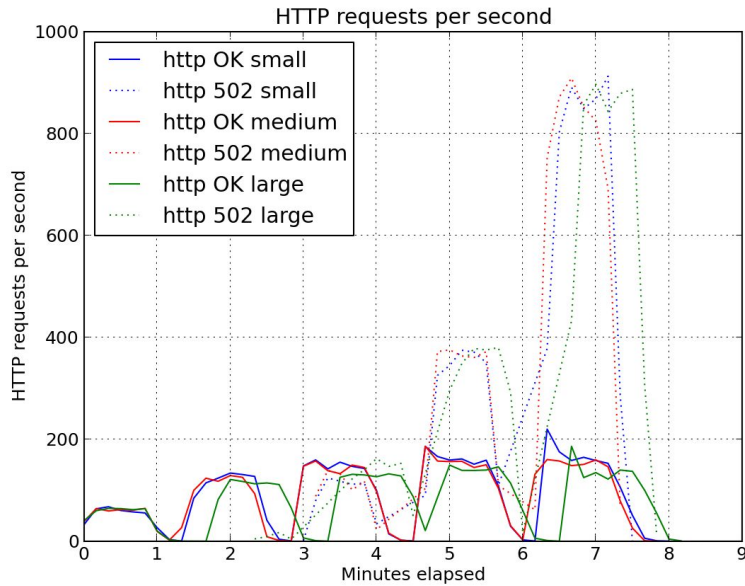


Fig 4.8 HTTP response status with t3 instance types

We also tested three different burstable instance types: t3.small (\$0.02/hr), t3.medium (\$0.04/hr), and t3.large (\$0.08/hr). Regardless of the instance type, the rate of 200 response is capped at around 180 per second.

We suspect the similarity in performance is partly due to the nature of t3 machines - t3 is a burstable instance type, meaning it can temporarily boost to 100% cpu utilization by using CPU credits. While each tier accumulate CPU credit at different rates, they are all able to boost to 100% given about 15 minutes of idle time.

We initially used t3.micro to run our application but we had to move to t3.small due to running out of memory. It seems that the 2 GiB of memory offered by t3.small is sufficient for our app. However, additional memory did not improve performance so we can conclude that memory is not a limited resource after 2 GiB.

Horizontal scaling

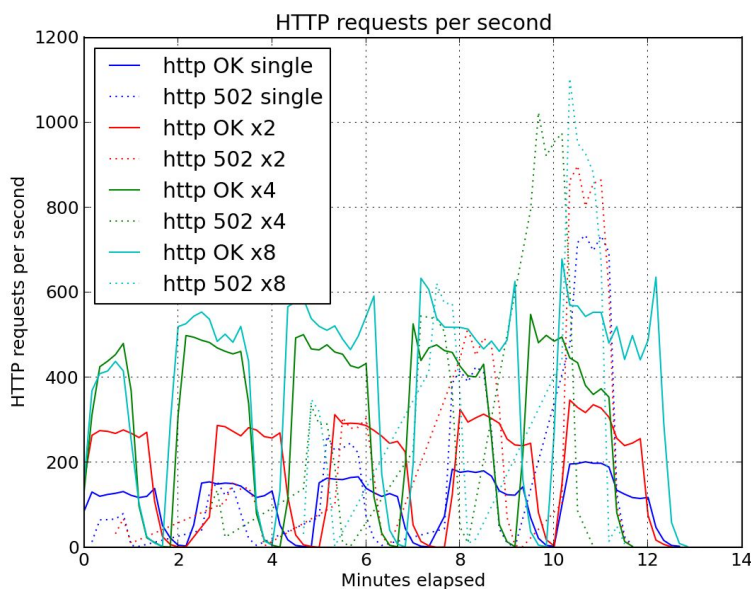


Fig 4.9 Response status code of requests with different amount of instances

Horizontal scaling achieved better performance compared to vertical scaling. We compared our baseline environment of 1 c5.large instance ("single") against environment of 2 ("x2"), 4 ("x4"), and 8 ("x8") c5.large instances behind a load balancer. Compared to vertical scaling, horizontal scaling provides almost identical increase in performance between the bottom two tiers. However, x4 and x8 performed much more consistently on high loads as compared to c5.2xlarge and c5.4xlarge.

It is also worth noting that x8 did not achieve a significant increase over x4, despite it having twice as many instances.

5 Conclusion

In this project, we learned how to build a social network that borrows ideas from many existing platforms. Next, we learned how to deploy this onto Amazon Web Services (AWS) Elastic Beanstalk (EB) which allows for dynamic creation of configurable virtual machine instances. Afterwards, we learned how to use the Tsung to simulate high traffic of users visiting our site. Using both Tsung and EB, we were able to run many tests of various optimizations to the application as well as traditional vertical and horizontal scaling of the application servers. Many times, we saw results that were unexpected which forces us to think about how our assumptions were misguided. As a learning exercise, this project was very successful across all fronts.

Performance-wise, the application shows clear growth for some optimizations, the largest being pagination. We also observed that there are diminishing returns on some optimizations such as vertical scaling, where t3.small has sufficient memory and anything larger is negligible, but almost linear scaling for horizontal scaling.

In the future, we could try different configurations of the database server similar to how to experimented horizontal and vertical scaling of our application servers. Furthermore, we could store images on Amazon Simple Storage Service (S3) to alleviate image requests from the server since S3 is well suited for sending and receiving files. To alleviate the processing time required to complete a database request, we can serve less data to the user initially and then provide the data to the user via subsequent Ajax requests.