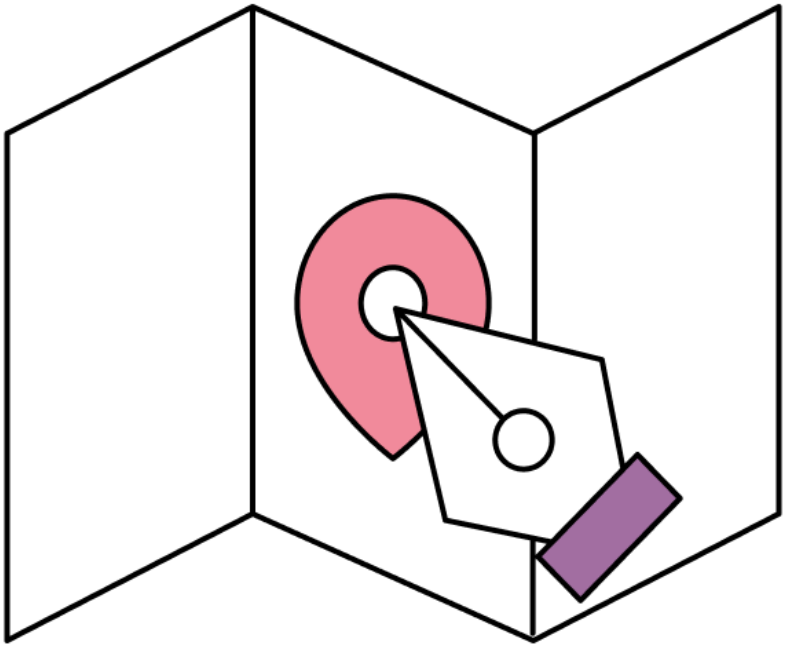
Computer Science 188: Scalable Internet Services
Fall 2020

wanderlust

draw your story.



Connie Chen (904920137)
Robert Geil (104916969)
Megha Ilango (704924099)
Eugene Lo (905108982)
Timothy Rediehs (105009354)

Table of Contents

Introduction

Wanderlust is an application that allows users to share and consume art at real physical locations around the world. Users can upload artwork to the platform that is geographically tagged according to their physical location; this art can be viewed only by other users located in close proximity to the location from which it was originally posted. The goal of Wanderlust is to provide a virtual artistic ecosystem for both artists and art consumers, combining the visual appeal of Instagram with the geographic incentives of Pokemon Go.

Creators can utilize the platform to share their art from any location. They can upload their own art pieces, which are accessible via the platform's map for other users to view when located in close geographic proximity.

Users can each view personal maps that are derived from the platform map, where their location history has been utilized to mark previously viewed creations as "visited." Thus, as they continue to use the app in different locations, more of their map is "unlocked" with viewable art. Allowing users to revisit previously viewed art and track the number and locations of art pieces they have viewed provides an incentive to continue traveling and consuming art through the platform.

Summary

Target User

There are two main groups of target users for this platform: creators and consumers.

Creators are those who primarily use the platform as a means for sharing their creative work; this includes existing artists, photographers, and writers. By providing a geographically tagged medium, we allow users to create installments in a "virtual art gallery," providing a unique opportunity for creators to make and display art inspired by or associated with a particular location.

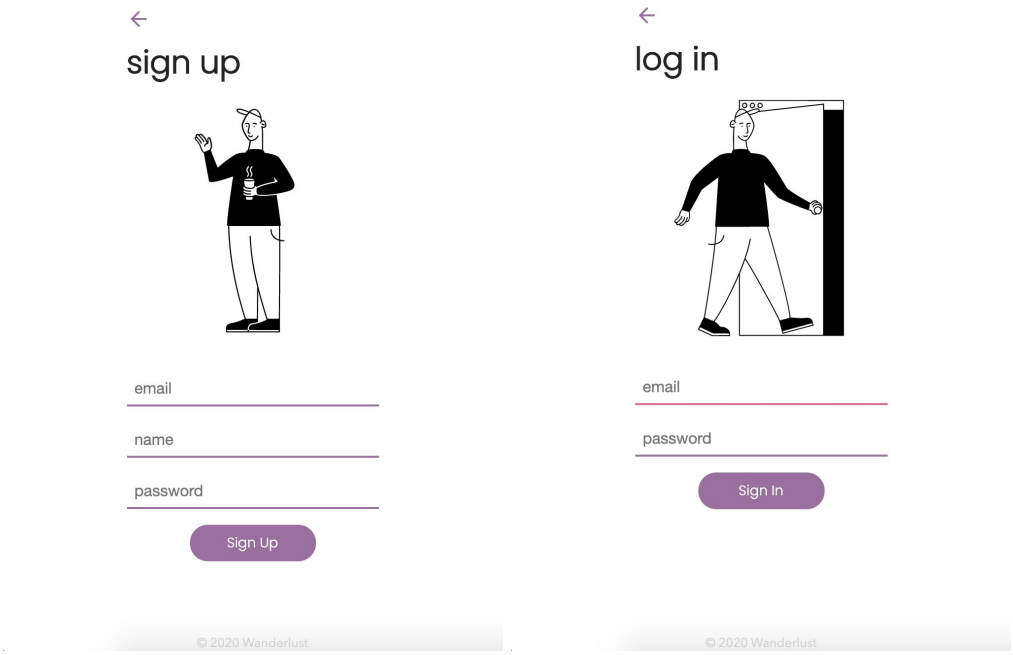
Consumers are those who primarily use the application to view art that has been posted by creators; the target consumer for this app is a traveler, who regularly frequents places outside of their locality. Such consumers can use such an application to enrich their travel experiences with the addition of virtual art pieces representative of a location.

We expect the line between creators and consumers of this platform to be blurred: all users can easily view and share their art, democratizing the process of art sharing via low barriers to entry for publication.

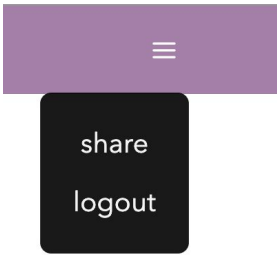
Features

Signup and Login

The application provides users with the ability to create a password-protected account with a unique email address and username.



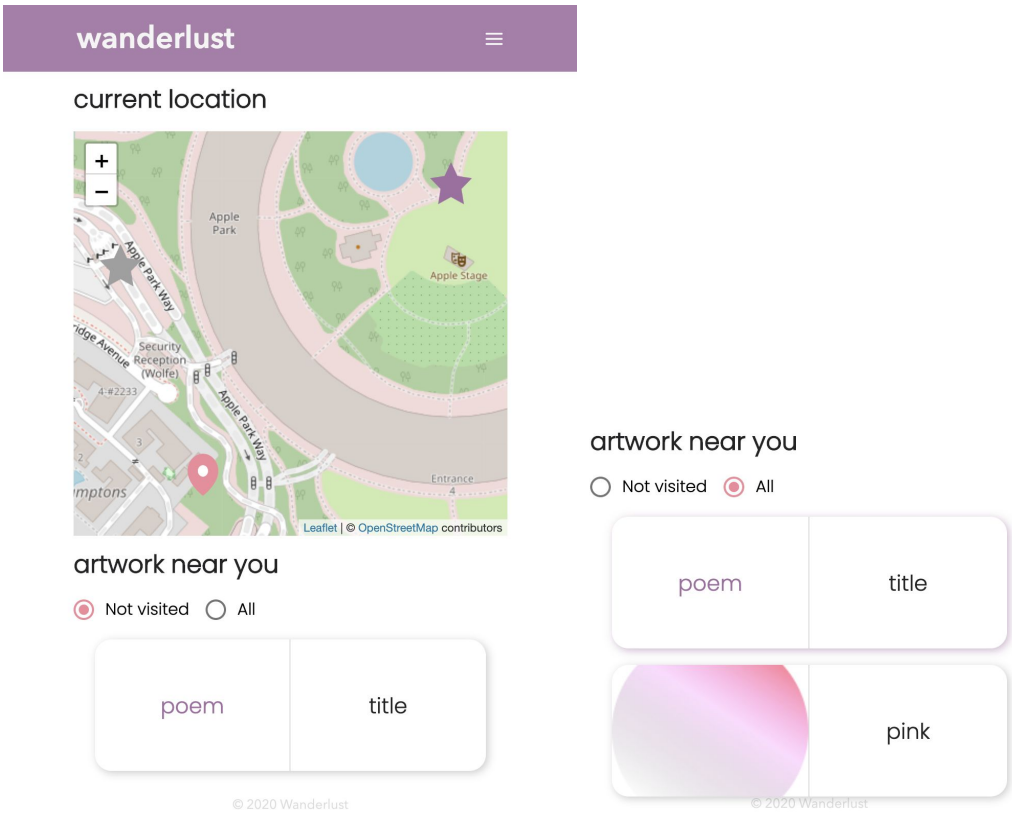
Email addresses are required to be in appropriate format for “Sign Up” and “Sign In” buttons to be enabled. A “logout” option is also available in the navigation bar once users are logged in, which takes them back to the application’s welcome page.



Viewing Art

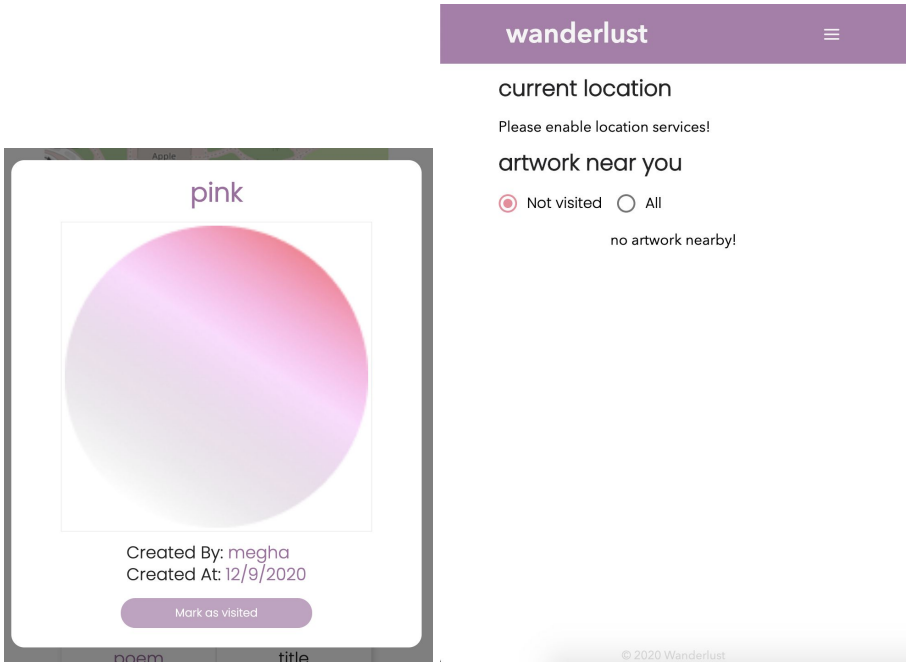
Upon login, the map page of the application is displayed, on which users can view the art around them. Art pieces are displayed on the map as stars, which are colored grey if the user

has not previously visited the piece, and turn purple once the user has previously visited, or “unlocked,” a piece.



Scrolling down on the page also reveals the full list of pieces that are viewable from the user’s current physical location. There is a default option “Not visited” to show only art that they have not previously visited, and another option “All” that shows all available art pieces in the location. This allows users to more easily find new art to view. The list items for previously visited art have a purple shadow behind them, while the items for the remaining art have a grey shadow behind them, providing another visual indicator of whether or not the user has viewed a piece of art even when they have chosen to toggle to the “All” option.

Upon clicking on an art piece in the list, a modal is displayed on which title, author, date of creation, and contents of an artwork are visible. The user also has the option to mark the art as visited before closing the window, which will ensure that it is not displayed as “Not visited” on either the list or the map.

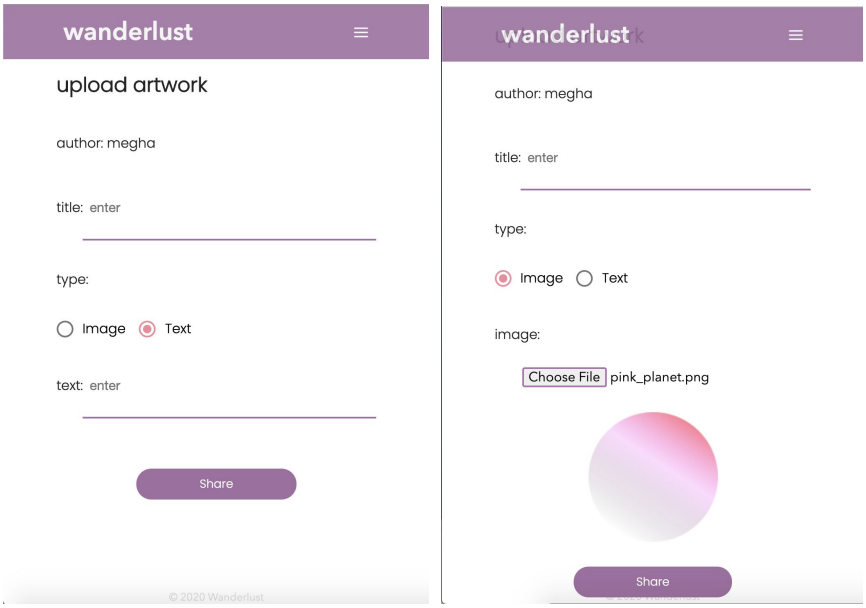


If the location for some reason cannot be retrieved, a “Please enable location services!” message is displayed rather than the map.

Sharing Art

The user can navigate through the Navigation Bar to “share” their own artwork. Users can enter a title for their piece, select whether they want to upload an image or enter text for its contents, and then publish to all users on the platform.

Below are two examples of the page, where 1) the user has clicked on the option to share text-based art, and 2) the user has uploaded an image, prompting a display of its preview before sharing it.



Accepted formats for art pieces are .jpg, .png images and plaintext that the user directly enters into the application. Upon clicking ‘Share,” the data associated with the art piece is stored and the user is navigated back to the ‘Map’ screen, which they can refresh to view their newly posted art. The creator ID associated with the user, email address, title of the art, location, art contents in base 64 binary string format is what constitutes an art piece when stored in the database.

Design

The visual design for this application aims to create an inviting, playful product experience.

We went through various iterations of color selection for the application, choosing a muted purple, pink, and orange as our primary, secondary, and tertiary colors. These colors together create a whimsical feel and evoke imagery of a sunset, a common sight for travelers to visit and artists to be inspired by.

The fonts used in our original designs were Poppins for titles and headings, Palanquin for text on push buttons, and Chivo for text entered by the user. All of these are sans-serif fonts, which give our application a modern and casual feel, and also render more cleanly on lower resolution screens.

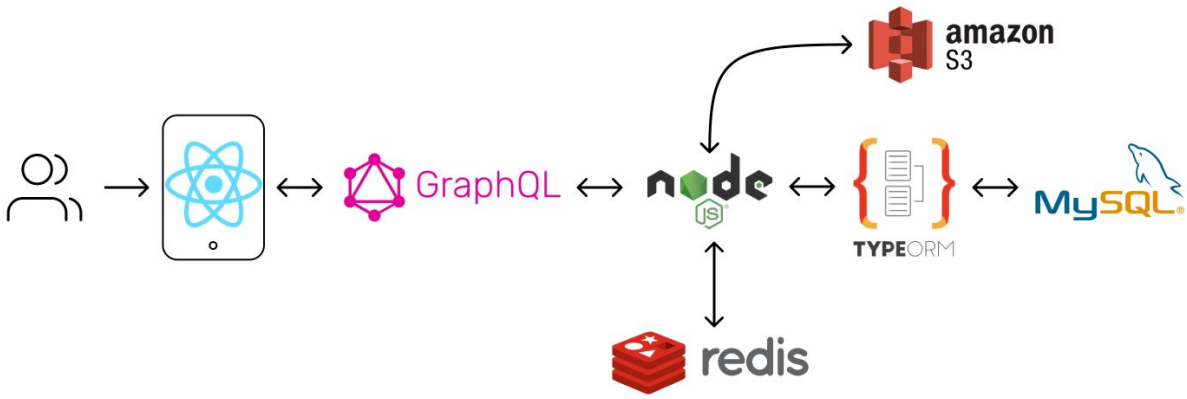
Wanderlust is an application that allows users to share and consume artwork at real physical locations around the world.

Wanderlust is an application that allows users to share and consume artwork at real physical locations around the world.

Wanderlust is an application that allows users to share and consume artwork at real physical locations around the world.

In practice, we did not use Chivo, instead using Palanquin for body text (Arial displayed when unavailable) and Poppins for headings and buttons.

Technology Stack



Wanderlust was developed using React on the frontend, along with the Styletron framework for component-based styling and Material-UI for pre-designed, reusable React components. These React components are first server-rendered, then later hydrated on the client using React's built-in server hydration feature in order to add necessary event handlers and dependencies. This ensures that users view a fast initial page load, and that the most expensive requests can first be done within the server before the final results are sent to the client. The frontend uses the Apollo Client state management library, which provides an implementation of GraphQL that enables in-memory, client-side query caching before remote GraphQL queries are necessary.

The backend, which uses a Node.js 12.9 runtime to run an Express-based Apollo GraphQL application server, receives GraphQL queries from the frontend and runs the necessary application logic to respond to the user's request. The backend uses MySQL for persistent, relational data on users and artwork, which the backend accesses using the TypeORM object

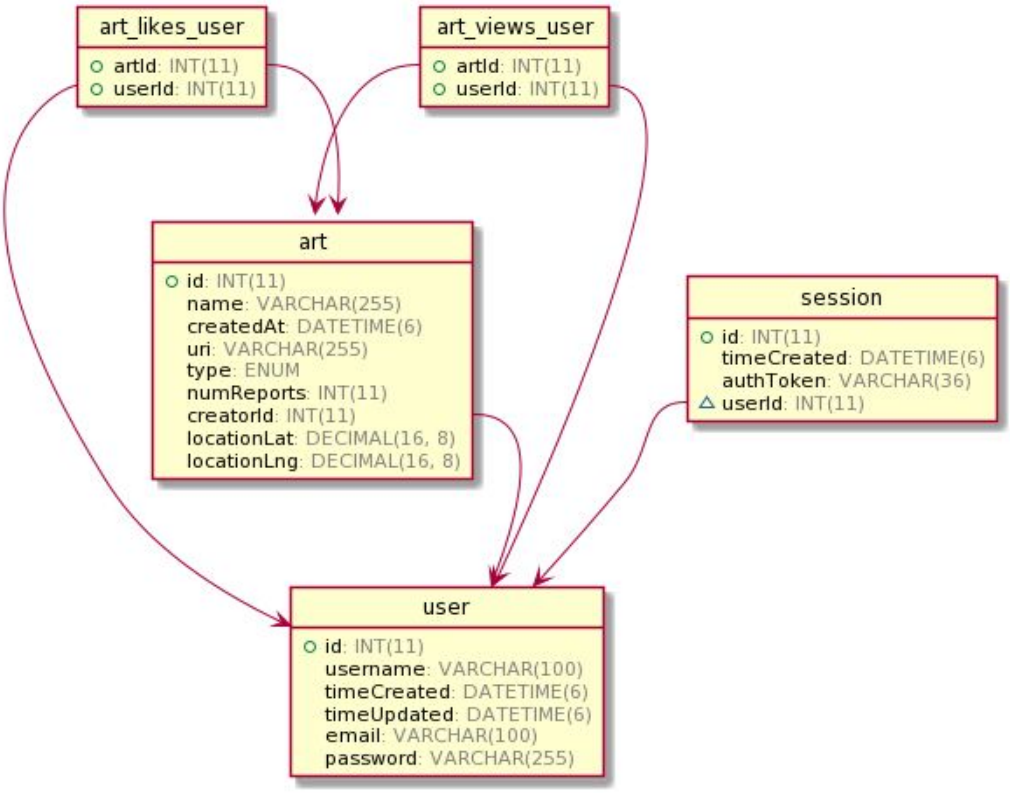
relational model. It also uses the Redis key-value store in order to remotely cache user sessions and user locations so that more expensive queries to the database can be avoided. Finally, the backend stores its images in AWS S3 cloud-based storage buckets.

Wanderlust is housed within an AWS EC2 instance, which communicates with an RDS (Relational Database Service) instance with MySQL. The user interacts with the deployed instance of the app through requests, and receives a response. Depending on the nature of the request, it may be able to be entirely handled within the application server, or it may have to query RDS for data in the database to fulfill the request. Deploying on AWS provides many benefits for scalability and availability. In terms of scalability, EC2 provides auto horizontal scaling, meaning that the number of EC2 virtual instances is able to be dynamically increased or decreased depending on the user traffic. AWS also offers high availability by providing the ability to deploy multiple instances of application resources across numerous regions and availability zones. If one instance of our application crashes in one particular region, it will still be available for usage in other regions. Although our current infrastructure is only deployed to the *us-west-2* region, AWS offers a user interface that allows us to seamlessly add more regions through just a couple of clicks. All AWS resources are declared through and provisioned by Terraform.

One interesting aspect about the app's architecture is the usage of containers via Docker. Before deployment to EC2, Docker first packages our Node.js runtime and server code, along with their dependencies, into a lightweight, standalone piece of software. Through containerization, the previous issue of having differing performance on different computing environments is removed, as containers ensure that the contained software works uniformly. Especially when considering that our app may be deployed to multiple locations, this ensures that all users have the same experience, regardless of where they are.

Data Model

We auto-generated a UML diagram using the @proscom/typeorm-uml package.



Our data is primarily stored as three TypeORM entities: Art, Session, and User, each mapping to a MySQL database table. The Art entity is how our application represents the artwork created by users in the database. It contains the `id`, `name`, `createdAt`, `location`, `uri`, `type`, `numReports`, `creatorId`, `likes`, and `views` columns. We have also constructed a many-to-one relation from Art entities to User, via Art's `creator` column. This is because each Art has one User creator, but each User creator can have multiple pieces of Art. The Art entity also contains two many-to-many relations from Art to User, via the `likes` and `views` columns. Since many-to-many relations require a join table, these two relations are represented by two separate tables in the database. In the figure above, they are represented by the `art_likes_user` and `art_views_user` boxes.

The Session entity contains information about each user's session. It contains the `id`, `timeCreated`, `user`, and `authToken` columns. The `user` column represents a one-to-one relation from the Session entity to the User entity, because we store one Session per User.

The last entity we defined via TypeORM is User. The User entity has the `id`, `username`, `timeCreated`, `timeUpdated`, `email`, `password`, `artworkCreated`, `artworkSeen`, and `artworkLiked` columns. The column `artworkCreated` is a one-to-many relation from the User entity to Art entities, because a user can create many pieces of art. In addition, similarly to the Art table, the columns `artworkSeen` and `artworkLiked` each represent many-to-many relations from User to User. Since these require a join table, they are represented as their own tables in the database.

Therefore, there are a total of 6 tables in the database. In addition to the `migration_schema_history` table, which stores migration schemas, we have the `art`, `session`, and `user` tables, which each map to their respective TypeORM entities. We also have the `user_art_liked_art` and `user_art_seen_art` join tables, each representing a many-to-many relation between Art and User.

Optimizations

Introduction

Our team's feature development ended up running differently than anticipated in the original sprint schedule. We found that parallelization of feature development was limited and continued developing features through the end of the quarter. Because of this, we performed observations in parallel with feature development rather than optimizing after finishing most of our features.

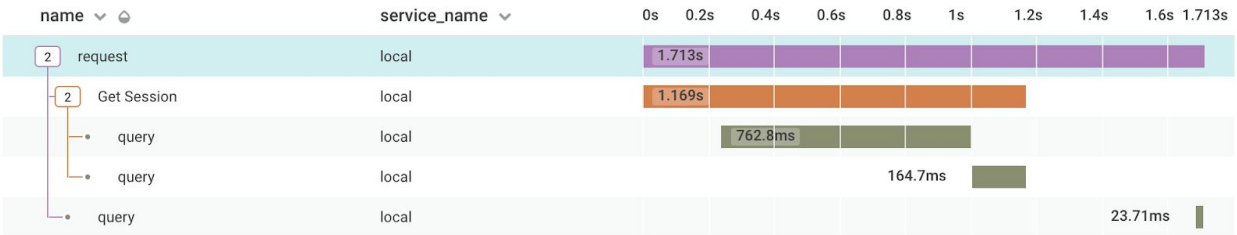
Our development workflow affects how we will present the rest of our paper. Because none of our optimizations share a single baseline development state, the rest of the paper is separated into sections by optimization. Each optimization section will describe the baseline performance before optimizing, give a hypothesis about the result of the optimization, and then describe our experimentation with each optimization and the resulting improvement. For more information on any of the optimizations, please refer to our wiki.

Using Redis for User Sessions

During our optimization, we realized that every GraphQL query to Wanderlust would make a database query to the Session table. We expect a user to make multiple GraphQL queries during one usage of the app, so this database query would be repeated for the same information. One solution we tried was using Redis to cache these queries in memory, which could speed up requests to /graphql.

Baseline

Before this optimization, we saw a P95 request duration of around 1.6 seconds when using the getArt.js load test. Creating a span around where the session is queried demonstrated that that takes up a majority of request duration.



It was clear that increasing the efficiency of this process could greatly improve our request time.

Hypothesis

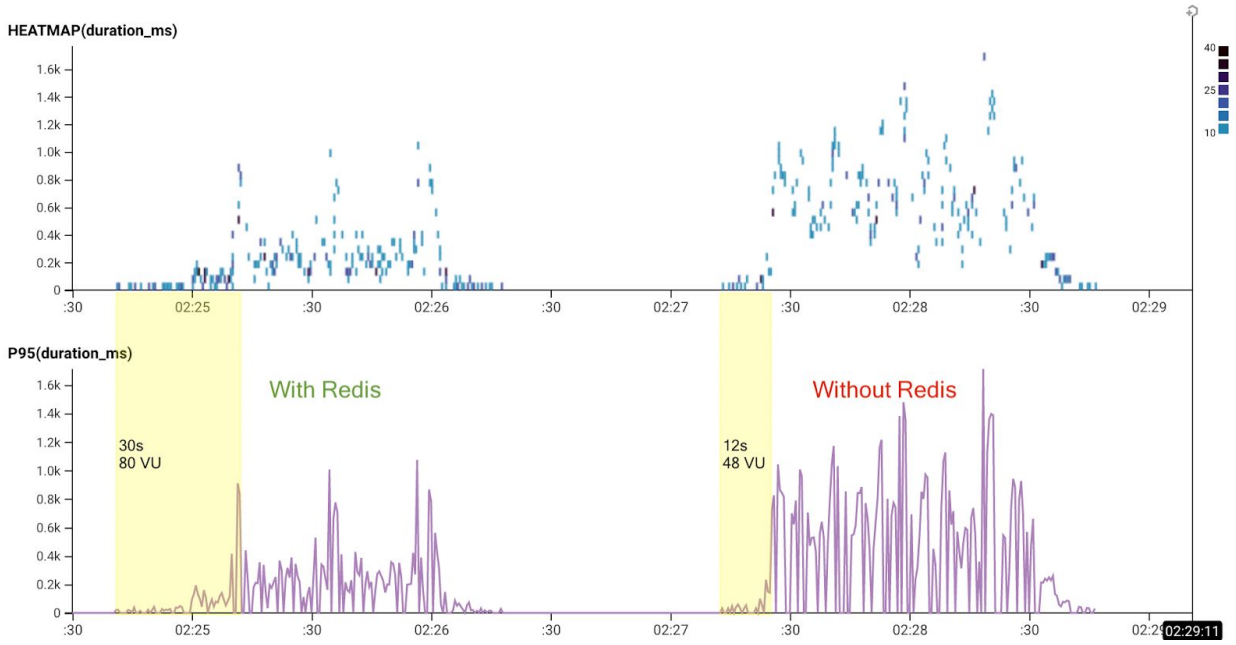
By duplicating the data from the database into faster memory, we anticipate that we can reduce the duration of requests to /graphql. This anticipation of a time-space tradeoff is because Redis operates in memory rather than on disk, which is slower than main memory.

Experimentation/Improvement

The changes required to add Redis were rather minimal since the project already had a Redis instance. Aside from importing the ioredis package, we simply had to choose what data we wanted to store and when we wanted to access it. Also take note of the Get Session span around this block.

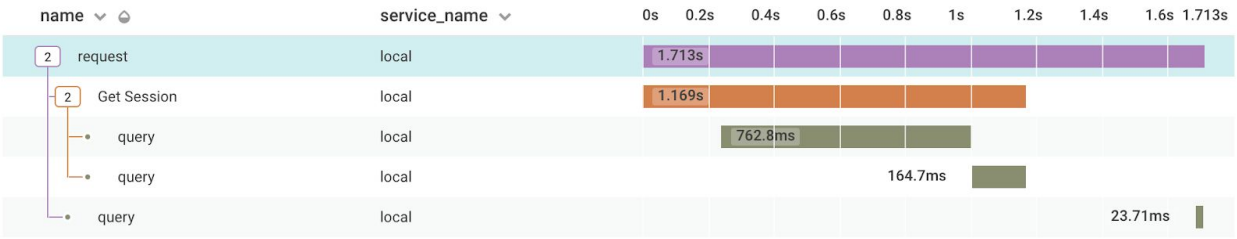
```
   asyncRoute(async (req, res, next) => {
     const authToken = req.cookies.authToken || req.header('x-authtoken')
     if (authToken) {
-      const session = await Session.findOne({ where: { authToken }, relations: ['user'] })
       let span = beeline.startSpan({
         name: 'Get Session',
       })
+      const cachedSession = await redis.get(authToken)
+      let session
+      if (cachedSession) {
+        session = JSON.parse(cachedSession) as Session
+      } else {
+        session = session || await Session.findOne({ where: { authToken }, relations: ['user'] })
+        await redis.set(authToken, JSON.stringify(session), 'EX', 15)
+      }
       if (session) {
         const reqAny = req as any
         reqAny.user = session.user
       }
       beeline.finishSpan(span)
     }
     next()
   })
```

After making this change, we reran our load test and saw a difference in query time. P95 duration has been reduced from a peak of 1.6 seconds to a peak of 1 second. Before Redis, performance degradation at about 48 virtual users was observed. Afterwards, the spike is at the 30s mark, which is well into the maximum of 80 virtual users.

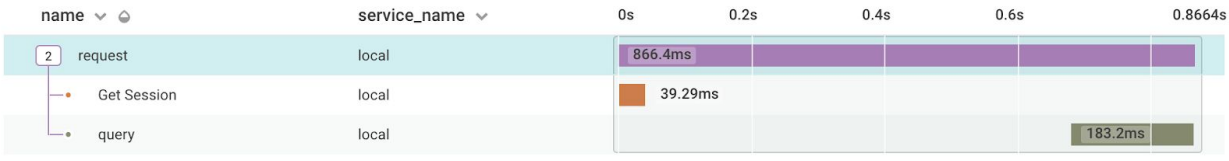


In addition, we can see that the amount of time in duration of the Get Session span has also been reduced and the two queries there have disappeared.

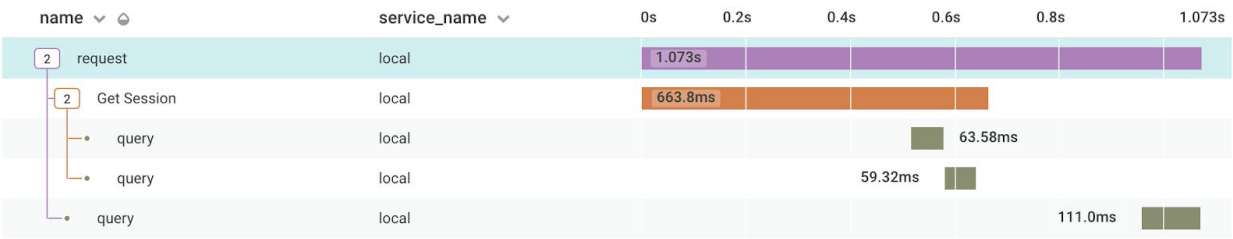
Before Redis



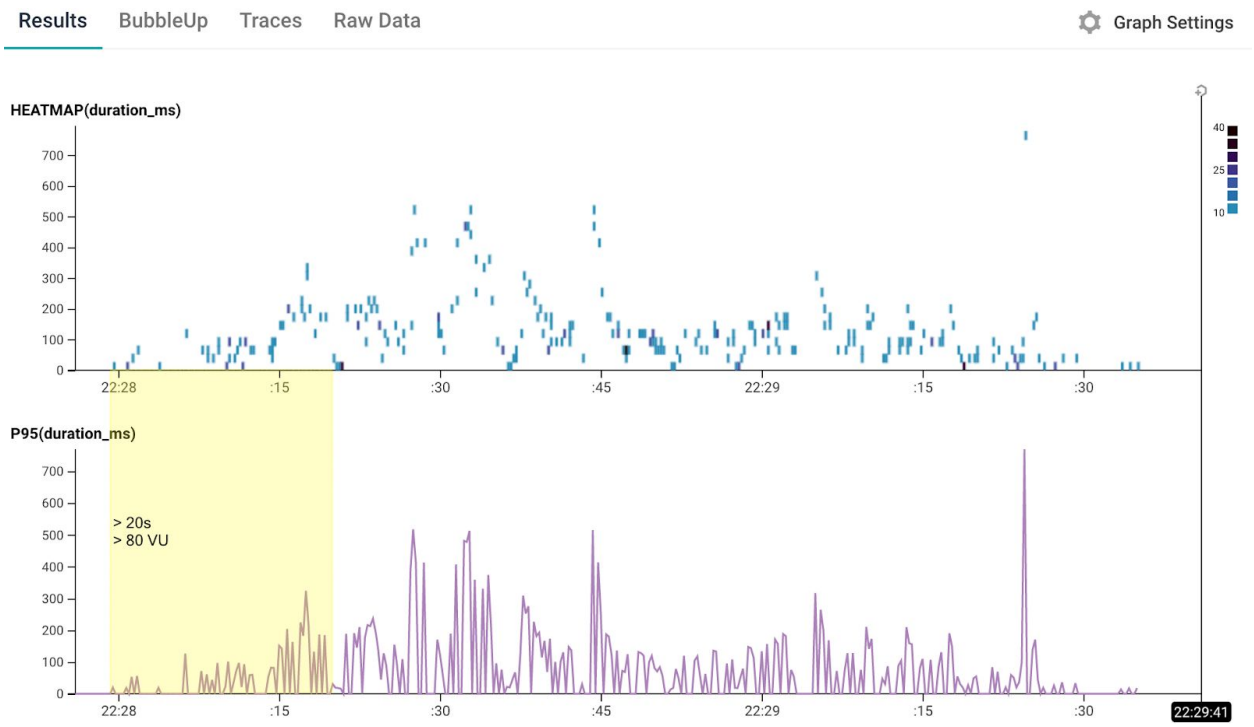
After Redis



However, the requests at one of the periodic spikes in the after Redis graph shows that the Get Session span still has database queries and is long. This is likely related to the Redis entries expiring after a short period of time.



We confirmed this by increasing the amount of time for which the Session entries live in Redis. Below, spikes are reduced, showing more than a 50% improvement. In addition, the beginning for performance degradation is at 80 virtual users. The amount of load we can take increased around 67%.

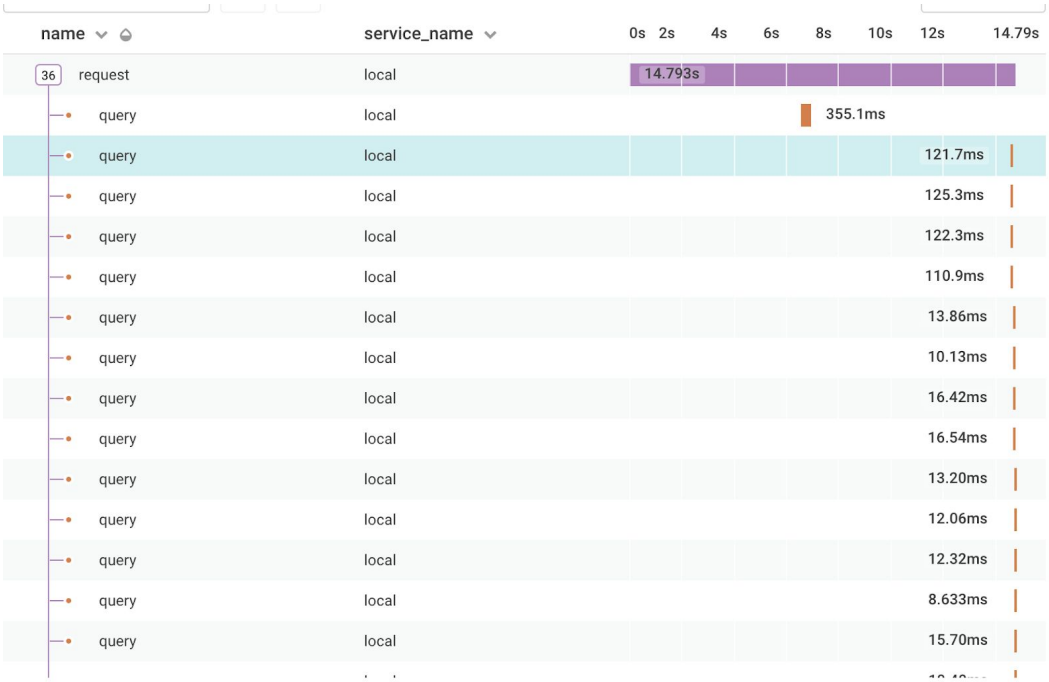


Decouple Types

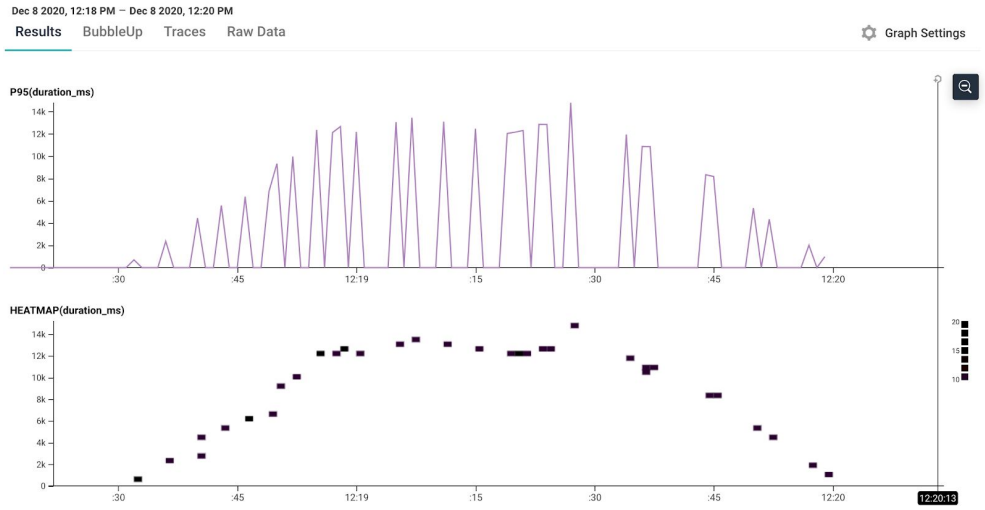
Due to the structure of our database, we have multiple circular relationships between users and artwork. For example, users have a list of artwork they've seen, while each artwork has a list of users that have viewed it. This type of relationship cannot be realized with pure database JOIN queries, thus necessitating multiple independent queries based on the fields requested in a GraphQL query to truly represent the data graph. Once this decoupling was performed, a new scaling issue ensued, the N+1 problem.

Baseline

With the decoupling of GraphQL and the database entities, we set up our OneToMany and ManyToMany fields to be lazily evaluated. This way TypeORM doesn't perform any joins, making the base queries more efficient. However, when joined data must be fetched, such as getting the creator for all pieces of artwork, we must perform N queries where N is the number of pieces of artwork retrieved in the first step. Hence the N+1 problem, where 1 database query spawns N new queries which must be evaluated. This issue can be seen in the below sample trace, the result of load testing using the nearbyGQLQuery.js script, which repeatedly queries for nearby artwork.



In the trace, there is a single request being made to fetch the nearby artwork, followed by N (in this case 35) additional queries to get the creators for each piece of artwork. Naturally, this causes a large amount of latency when many users are simultaneously requesting nearby artwork, as seen in a heatmap of this query, with a latency of roughly 12 seconds for 50 users.



50 VUs

Hypothesis

As we've seen from the trace, the large amount of latency on this query is due to the numerous queries that are spawned to perform the equivalent of a JOIN between the arts and user table. In order to rectify this issue, we turned to another GraphQL tool, DataLoader. This library, also created by Facebook, is a simple batching utility that uses a function which takes an array of keys and returns an array of values. By delaying queries, many of them can be batched together and returned at once, eliminating the N+1 problem.

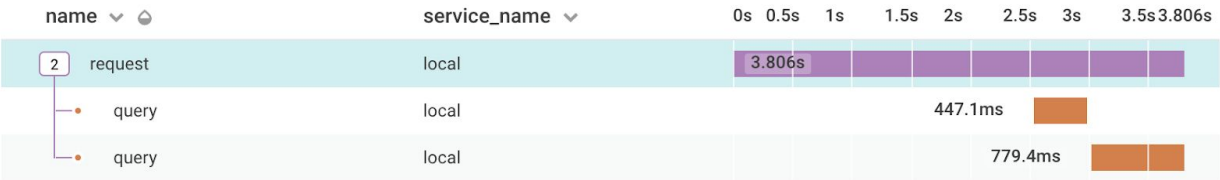
Experimentation/Improvement

For our experiment adding DataLoader to improve latency, we created two batch functions, `batchLoadUsers` (below) and `batchLoadArt`.

```
async function batchLoadUsers(keys: readonly number[]): Promise<(User | null)[]> {
  const users = await User.find({ where: { id: In([...keys]) } })
  const hashmap = new Map<number, User>()
  users.forEach(u => hashmap.set(u.id, u))
  return keys.map(k => hashmap.get(k) || null)
}
```

These functions both took in an array of user or art IDs to fetch, made a single WHERE IN database query, then mapped the results back to the input array's ordering. DataLoader is not a caching tool, so a new instance of our loader was created for every GraphQL query and lasted

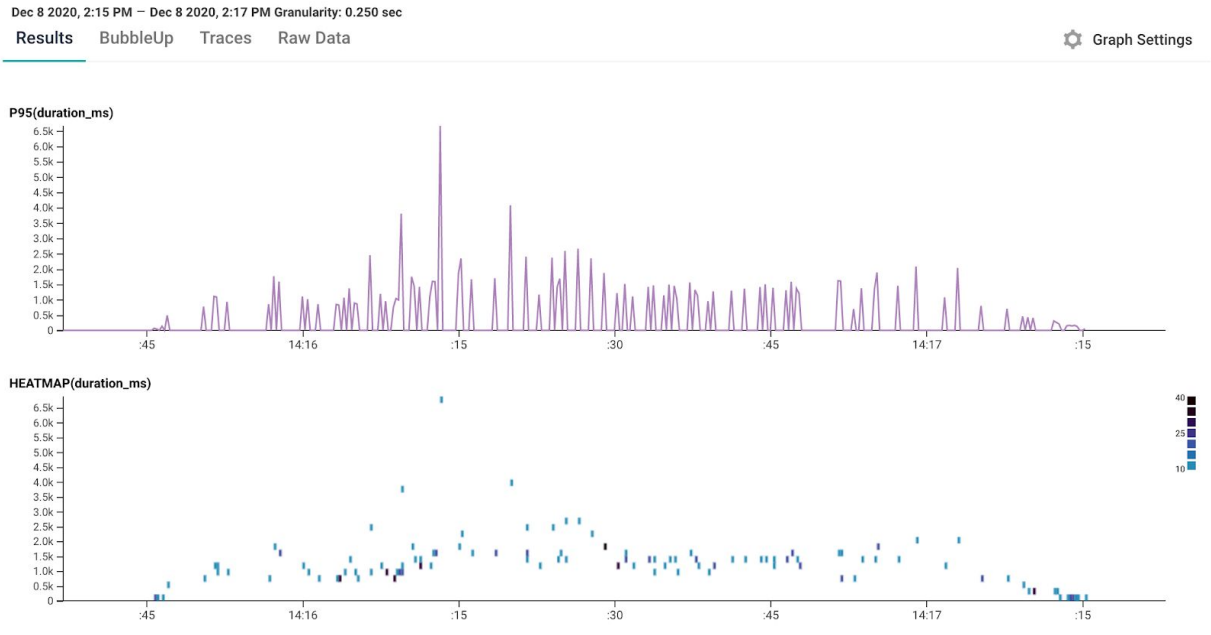
only for the lifetime request. The results of this optimization are quite apparent.



Above we see the new trace of a request during the same load test. There are only two queries to the database, since all the users being fetched are coalesced into a single request through DataLoader. Inspecting the second query in the trace, we observe that all the users are being fetched by ID in a single request.

```
SELECT `User`.`id` AS `User_id`,
  `User`.`username` AS `User_username`,
  `User`.`timeCreated` AS `User_timeCreated`,
  `User`.`timeUpdated` AS `User_timeUpdated`,
  `User`.`email` AS `User_email`,
  `User`.`password` AS `User_password`
FROM `user` `User`
WHERE `User`.`id` IN (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

The benefits of this optimization are also shown in the heatmap of P95 times when the same load test is run.



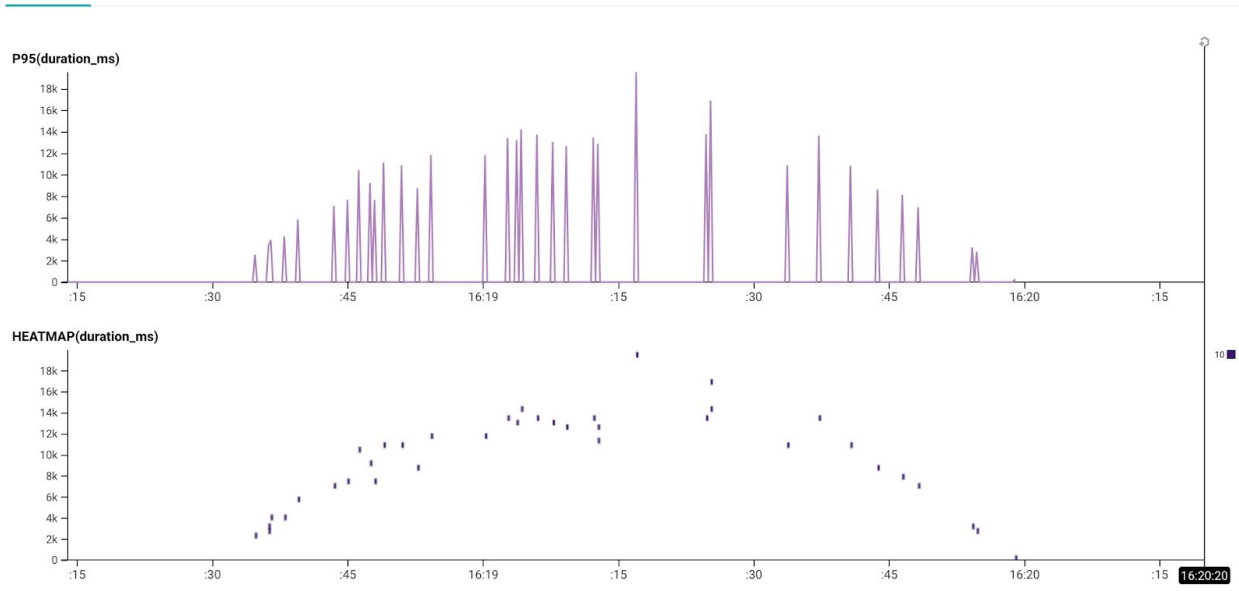
With DataLoader integrated into the server, our P95 is around 1.5 seconds for most requests with 50 virtual users, an almost 10x improvement over the naïve approach. For the few outliers, there was a latency of about 6 seconds, which was still a 2x improvement over the initial performance.

SQL Query Optimization

One dimension of scaling which wasn't discussed as much is scalability in the data dimension. While many load tests checked if the performance of the application degraded under more users, our SQL Query Optimization was intended to test the performance of our database under a large amount of artificial data, simulating the status of the app after the contribution of users over time.

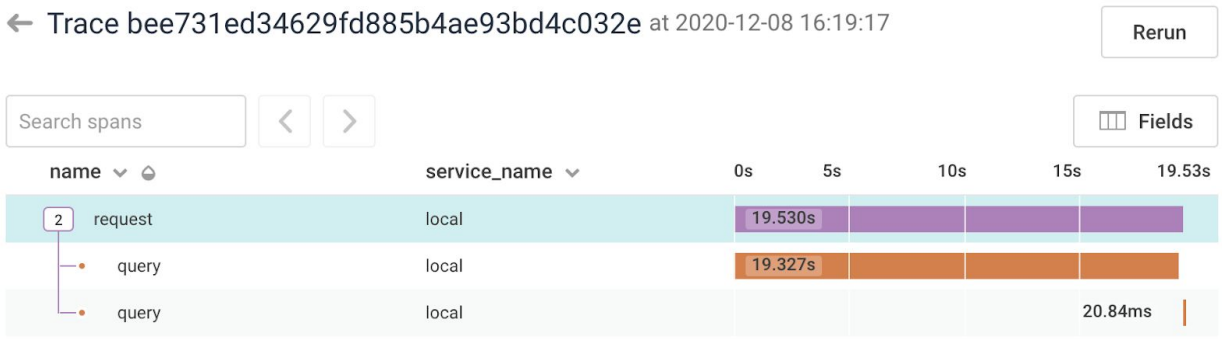
Baseline

In order to create our SQL Load Test, we wrote a Python script to create a migration containing an arbitrary number of users. For this test, we created 100,000 users and almost 300,000 pieces of artwork spread across a 15,000 square mile circle. As seen in the below P95 heatmap of our nearbyGQLQuery.js load test, our latency increased from the roughly 1.5 seconds experienced with the dataloader optimized queries to nearly 20 seconds in some cases, indicating that our database was not scaling well with additional data.



Hypothesis

Looking at a trace generated from the above query, we see that all of our additional latency was from an SQL query to fetch the nearby artwork.



Looking at the offending SQL statement, we see that we are querying:

```
SELECT `art`.`id` AS `art_id`,
  `art`.`name` AS `art_name`,
  `art`.`createdAt` AS `art_createdAt`,
  `art`.`uri` AS `art_uri`,
  `art`.`type` AS `art_type`,
  `art`.`numReports` AS `art_numReports`,
  `art`.`creatorId` AS `art_creatorId`,
  `art`.`locationLat` AS `art_locationLat`,
  `art`.`locationLng` AS `art_locationLng`
FROM `art` `art`
WHERE (abs(`art`.`locationLat` - ?) < 0.02) AND (abs(`art`.`locationLng` -
?) < 0.02)
```

While this looks innocuous enough, by running the query with EXPLAIN ANALYZE in the terminal we got the following output.

```
| -> Filter: ((abs((art.locationLat - 34.0156)) < 0.02) and (abs((art.locationLng -
(-(118.503)))) < 0.02)) (cost=30022.20 rows=292932) (actual time=0.179..7232.685
rows=161 loops=1)
   -> Table scan on art  (cost=30022.20 rows=292932) (actual time=0.131..3679.721
rows=295799 loops=1)|
```

This output, while a bit hard to read, details that we are performing a query with cost 30022.20 and expect to examine 292932 rows, performing a “Table scan”, or looking at every entry in the art table.

Experimentation/Improvement

The reason for this poor performance is likely that there is no computed index on the absolute values of location longitude and the location latitude. Therefore, SQL falls back on a full table

scan and inspecting over a quarter million rows for every request! To rectify this issue, we first added an explicit index on our latitude and longitude columns within TypeORM.

```
export class Location {
  @Index()
  @Column('decimal', { precision: 16, scale: 8 })
  lat: number

  @Index()
  @Column('decimal', { precision: 16, scale: 8 })
  lng: number
}
```

Unfortunately even with the indices, we are performing the same full table scan with the above query, since the absolute value between art and the input location is computed on the fly. We can address this by instead of computing the absolute value, querying a conjunction of less-than and greater-than clauses, such as

```
WHERE art.locationLat > ? - 0.02 AND art.locationLat < ? + 0.02 AND -- same
for lng
```

Modifying our source code to make this more complicated query, we generated the following SQL to execute.

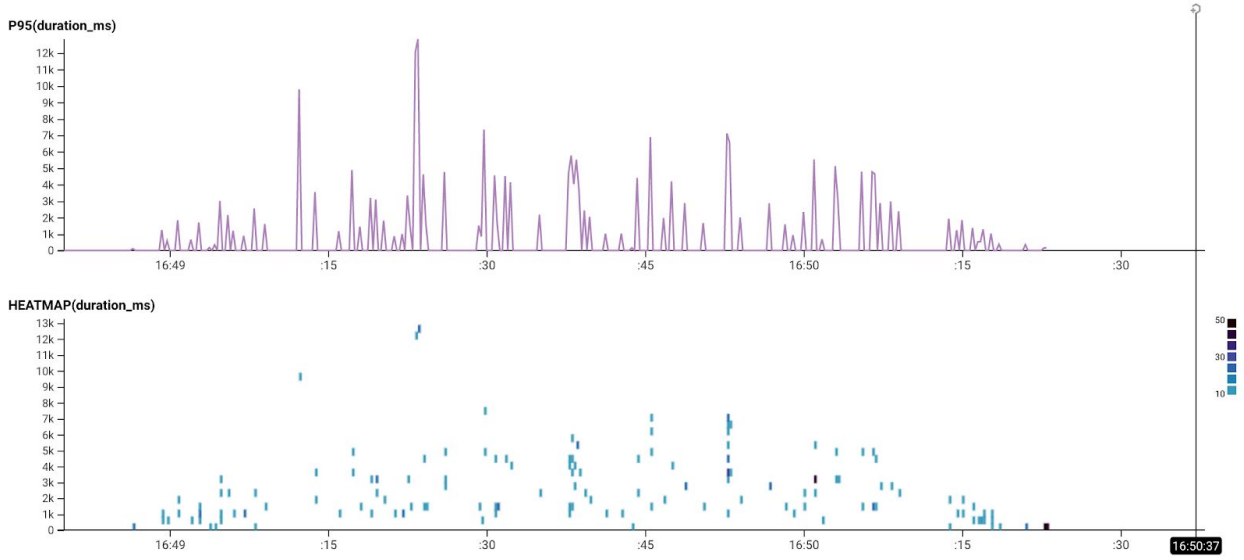
```
SELECT `art`.`id` AS `art_id`,
  `art`.`name` AS `art_name`,
  `art`.`createdAt` AS `art_createdAt`,
  `art`.`uri` AS `art_uri`,
  `art`.`type` AS `art_type`,
  `art`.`numReports` AS `art_numReports`,
  `art`.`creatorId` AS `art_creatorId`,
  `art`.`locationLat` AS `art_locationLat`,
  `art`.`locationLng` AS `art_locationLng`
FROM `art` `art`
WHERE `art`.`locationLat` > ? AND `art`.`locationLat` < ? AND
`art`.`locationLng` > ? AND `art`.`locationLng` < ?
```

Running an EXPLAIN ANALYZE query with our same sample data gives us

```
| -> Filter: ((art.locationLng > -118.52300000) and (art.locationLng < -118.48300000))
(cost=3414.86 rows=201) (actual time=3.837..204.953 rows=161 loops=1)
   -> Index range scan on art using IDX_b89be0b9a5ac0699ef6d841781, with index
condition: ((art.locationLat > 33.99560000) and (art.locationLat < 34.03560000))
(cost=3414.86 rows=7588) (actual time=1.326..124.829 rows=7588 loops=1) |
```

Now rather than performing a full table scan, we are using the generated index to query for artwork within a range. We also see that the cost has come down to just 3414.86 and the rows

to 7588, a 10x and 40x decrease respectively. This improved querying efficiency is reflected when we rerun our load test.



50 VUs

Now, while there are spikes up to 12 seconds, the majority of our queries are below 5 seconds, a 2x improvement over the naïve queries, and only about 3 times slower than the small data case, with roughly 5000x the amount of data to be queried.

Number of Database Connections

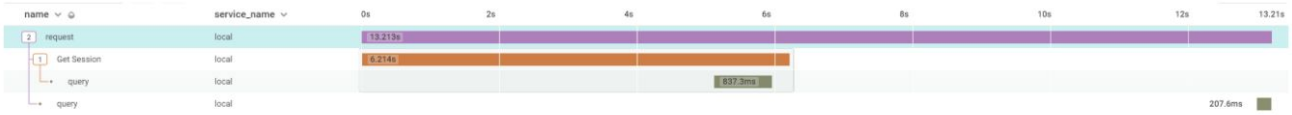
One of the potential optimizations we explored in class was tweaking the number of database connections. As we tried to scale our application, we noticed that the number of queries that our application can process at once is limited by the number of connections our app server has with our MySQL database. Since one database connection can only process at most one query, it is natural to conclude that adjusting the number of database connections could optimize our application performance.

Baseline

Before making this optimization, our application started off with 5 database connections, as denoted by the connectionLimit field in our `initORM()` function, shown below.

```
export async function initORM() {
  return await createConnection({
    ...baseConfig,
    type: 'mysql',
    username: process.env.MYSQL_USER || 'root',
    synchronize: true,
    logging: false,
    entities: [User, Session, Art, Location],
    legacySpatialSupport: false,
    extra: {
      connectionLimit: 5,
    },
  })
}
```

When running the getArt.js load test, we saw that the longest request took approximately 13.21 seconds.



It is clear that there was a lot of room for improvement here.

Hypothesis

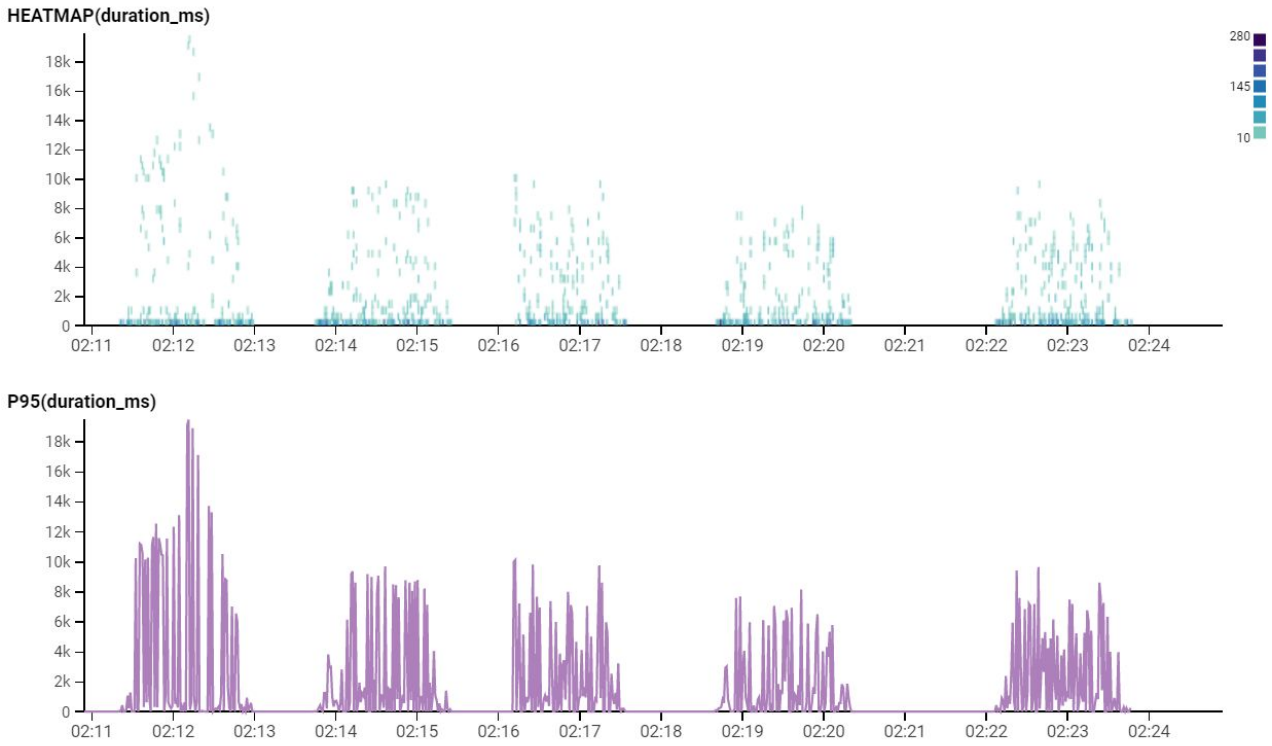
We know that we will have to increase the number of database connections, allowing for more database queries to be processed in parallel. However, the magic number of database connections is something to be determined via experimentation, since the optimal number of database connections is different for each application. There is also an overhead cost associated with creating and maintaining these connections, so there is no guarantee that a greater number of connections leads to faster performance.

Experimentation/Improvement

There was not much work associated with actually modifying the number of database connections. We changed the value of the connectionLimit field in the `initORM()` function. We also made sure that the MySQL database would be able to handle the increased number of connections. To do this, we logged into the SQL server and ran the following commands:

```
-- Show current number of max connections
SHOW VARIABLES LIKE 'max_connections';

-- Increase number of max connections to 1024
SET GLOBAL max_connections = 1024;
```

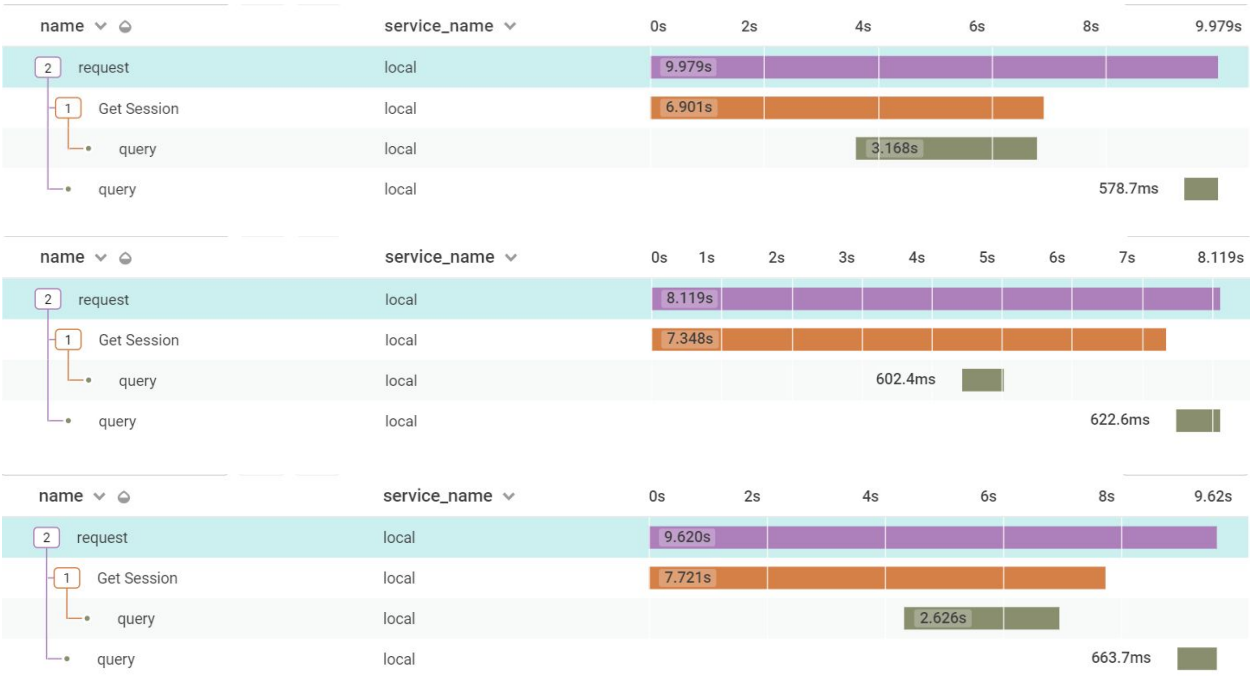
The image below shows the results of the load tests run with different numbers of database connections, from left to right: 5 (baseline), 14, 28, and 56, and 112.



We can see that overall, there is a noticeable reduction in average request duration as we increased the amount of database connections from 5 to 14 – however, the improvement becomes less noticeable as we increase the number from 28 to 56. Similar to the law of diminishing returns in economics, eventually, increasing the number of database connections will no longer improve our application performance due to overhead costs. Performance may even worsen, as demonstrated in the load test run with 112 database connections (rightmost).

Looking more closely at some of the traces for the longest request during each load test, we can observe that the amount of time the server is idle before processing a query is generally reduced as we increase the number of database connections (compared to the baseline). The

following three traces show the longest request during the 28, 56, and 112 database connections load tests (in order).



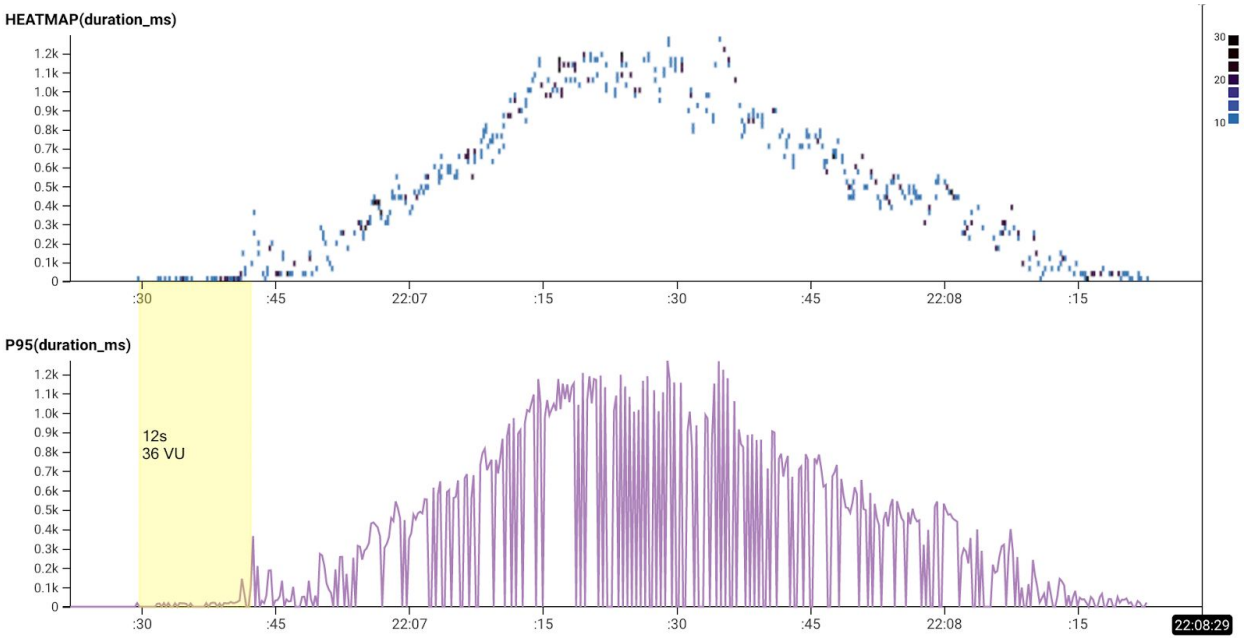
This shows that the amount of time the server is idle and waiting for an open database connection before the query decreases as we increase the number of connections from 28 to 56, but ends up increasing again when we increase the number of connections to 112. This further demonstrates that although there is noticeable improvement in the average duration of our requests from the baseline performance, there are also performance drawbacks from having too many open connections.

SchemaLink

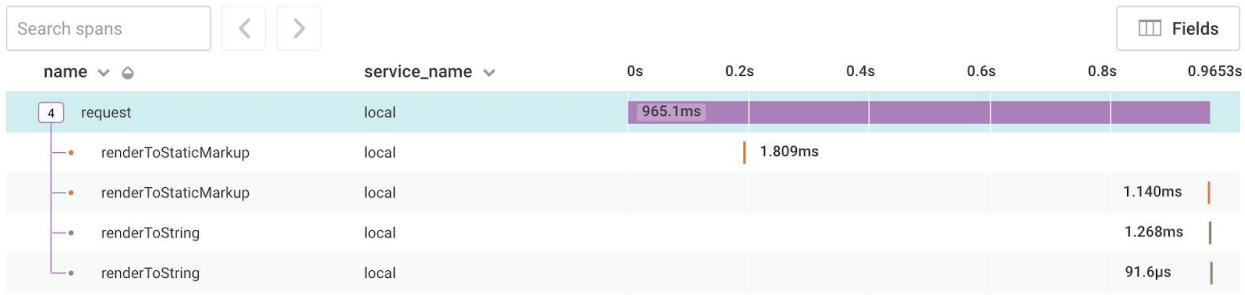
One of the most basic features of Wanderlust is simply accessing the map. We test this functionality using mapScript.js. In class, the professor pointed out that the server made API calls to itself when rendering the page on the server. Because of this, he suggested that we change HttpLink to SchemaLink on the server so the server knows that it can make its GraphQL queries directly instead of making an API call.

Baseline

When we put some load on our application, we saw clear performance degradation:



P95 response times increased above 200ms after 36 virtual users were added to the load test, which takes 50 seconds to ramp up to 140 virtual users. Upon inspection of one of the longer lasting requests, we found this:



Hypothesis

The question here is why these requests take so long and how we can fix it. If Apollo Client is using an HTTP connection to complete GraphQL queries when building the page on the server, we can guess that this causes a lot of latency. We predict that switching the link in the Apollo Client from HttpLink to SchemaLink will significantly reduce the duration of the requests to the map page because we will no longer be waiting on web requests. This hypothesis is prematurely supported by our lecture.

Experimentation/Improvement

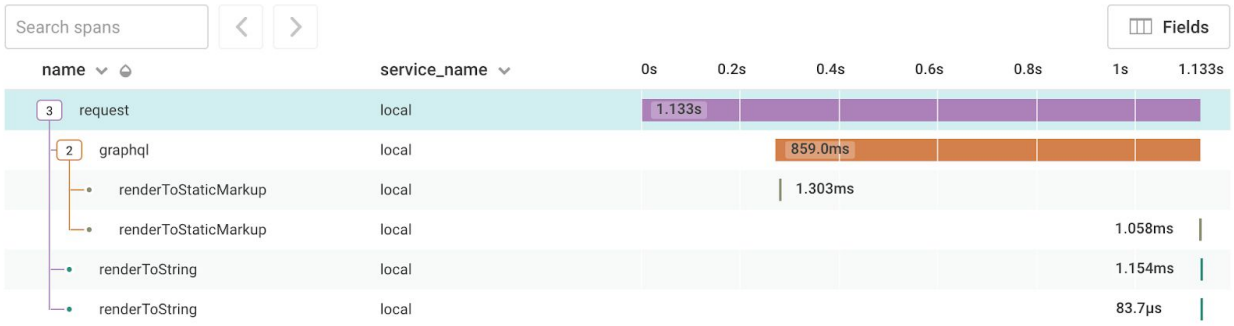
The initial data in our baseline is not very useful, because it does not show what specifically is taking so much time. However, we do find the problem near `renderToString`, which happens after the long gap.

```
getDataFromTree(app)
        .then(() => {
                const apolloState = apolloClient.extract()
                const body = ReactDOMServer.renderToString(app)
                const styles = engine.getCss()
                const html = ReactDOMServer.renderToString(
```

Before our two calls to `renderToString`, we have to wait for the promise returned by `getDataFromTree` to resolve. `getDataFromTree` makes the necessary GraphQL queries to render the page. We can prove that this is the problem by adding a honeycomb span.

```
+       const span = beeline.startSpan({
+               name: 'graphql',
+       })
                getDataFromTree(app)
                        .then(() => {
+                               beeline.finishSpan(span)
                                const apolloState = apolloClient.extract()
                                const body = ReactDOMServer.renderToString(app)
```

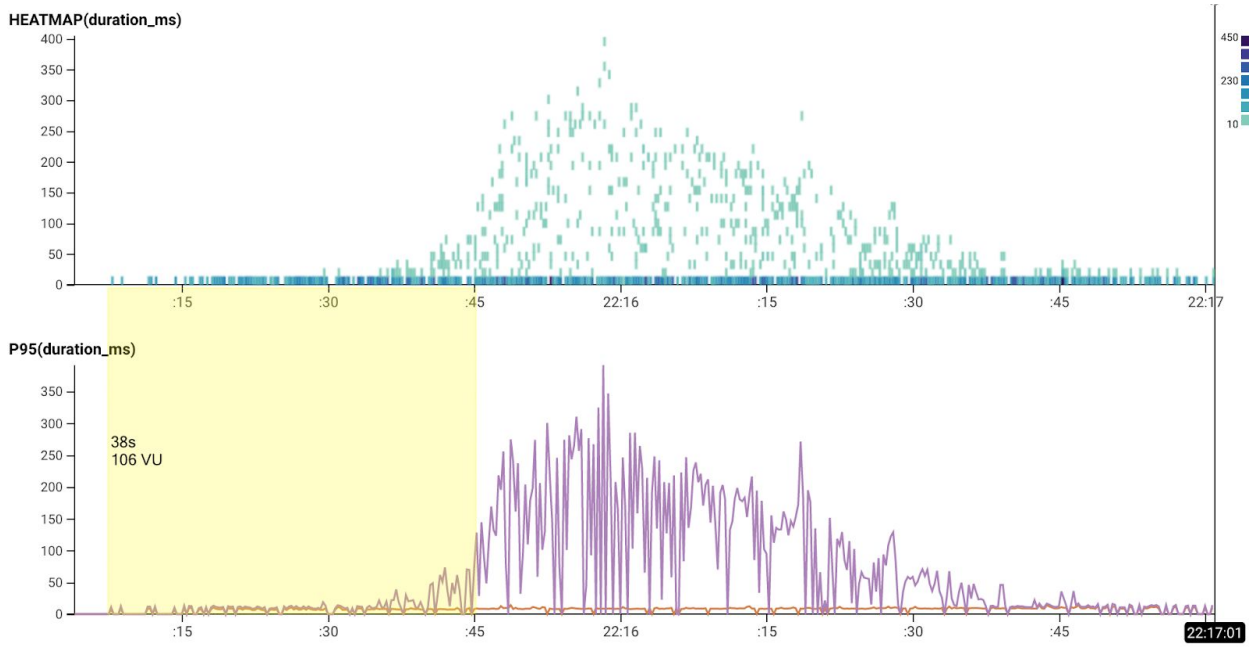
If we do this, we get the following:



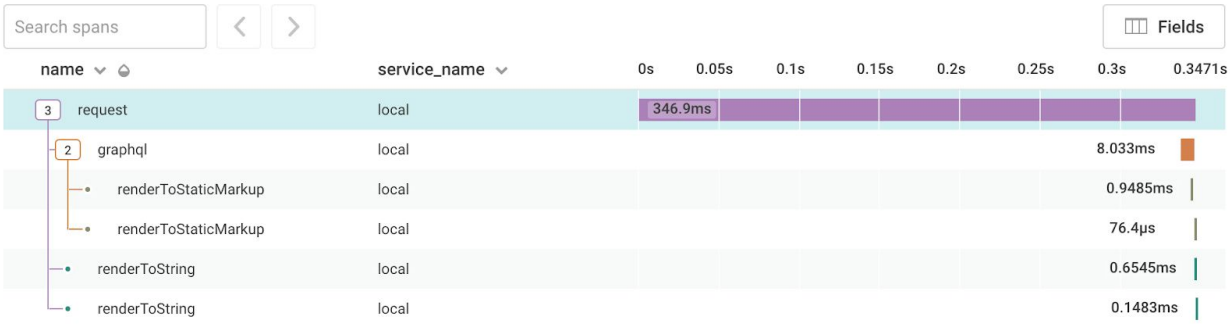
Clearly, this promise takes a long time. Thus, we needed to either reduce the number of queries that getDataFromTree makes or reduce the time that those queries take. These queries are made using the client apolloClient specified in the `<ApolloProvider/>` tag. In the definition of apolloClient, `apolloClient` creates an HttpLink. Because we are server side rendering, this is all happening from the server. Why should the server open an HTTP connection to itself? If only there was some option that…

"assists with mocking and server-side rendering" – Apollo Docs

SchemaLink indeed provides a better alternative. If we use SchemaLink instead, we not only end up with shorter code, but we also get better performance:



It takes about 38 seconds for P95 request times to start spiking, which corresponds to 106 virtual users. This is almost a 3x improvement in capacity. The span's duration has also been reduced:



This reduced the total time from 1.2 seconds to around .35 seconds. While this reduction is less than an order of magnitude, it was practically free, as making an HTTP request to oneself is usually not advisable.

Optimize Art Upload, Deployment and S3

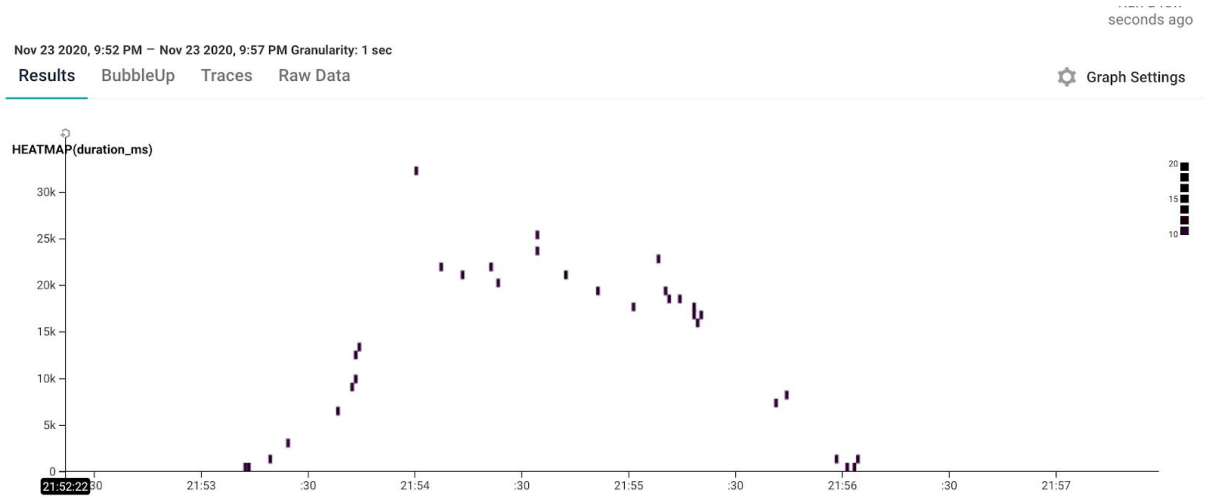
One of our first load tests was testing the mutation to create new artwork. In addition to performing a database operation, this endpoint was doubly complex due to the fact that we store our user generated artwork (image and text) in an AWS S3 bucket. This optimization and feature also required us to use terraform to provision an S3 bucket, along with testing our code in a deployed state.

Baseline

In order to load test this functionality, we wrote a k6 script [textUpload.js] which fetched the upload page, waited a few seconds, then performed the GraphQL mutation to add a new piece of text artwork. We also had to create a terraform module to provision our S3 bucket, using the following code then running `terraform init && terraform apply`

```
resource "aws_s3_bucket" "b" {
  bucket = "wanderlust-images"
  acl    = "public-read"
  cors_rule {
    allowed_headers = ["*"]
    allowed_methods = ["GET"]
    allowed_origins = ["http://localhost:3000",
"https://wanderlust.cloudcity.computer"]
  }
}
```

With the bucket and load test setup, we were able to check our performance. Initially, the results were awful. As seen below, with just 50 VUs, our response times spiked rapidly, reaching over 30 seconds.



50 VUs

Even worse than the response times were the actual responses themselves. Only about 5% of responses were error free, and the rest had an error stating "Column 'creatorId' cannot be null."

Hypothesis

To draw a hypothesis about the poor performance of the load test, we looked at a sample trace to see what was going on under the hood. As we can see from the below image, there are a large number of database queries being performed for every request.

This trace doesn't correspond with our intuition of what should be happening with the code. The mutation should, in theory, simply insert a single row to the database representing the new artwork; however, we have 38 individual queries being run where intuitively only one should exist. Therefore, we performed further investigation to the individual queries and codebase to resolve the issues. Looking at the queries in the trace, we saw two queries to get a user from the database, as well as a query to create new artwork, and numerous queries, wrapped in a transaction to UPDATE the art table, setting the creator id. All of this suggested that there was something wrong with the implementation.

Experimentation/Improvement

The first issue we addressed was the use of multiple queries to get the user from the database. Initially, the server seemed to be querying the database to find a user matching the auth token provided, then attaching that user to the Context that was included in every GraphQL query. However, looking at the code showed the GraphQL endpoint for our mutation also made a lookup into the database for a user with the given ID. Not only was the GraphQL endpoint querying the same data twice from the database, by ignoring the user on the Context, it was

also permitting users to impersonate others! In order to fix that, we applied a relatively small change, replacing the query within the mutation to fetch a user by the provided ID with a check that the logged in user's ID matched the provided one.

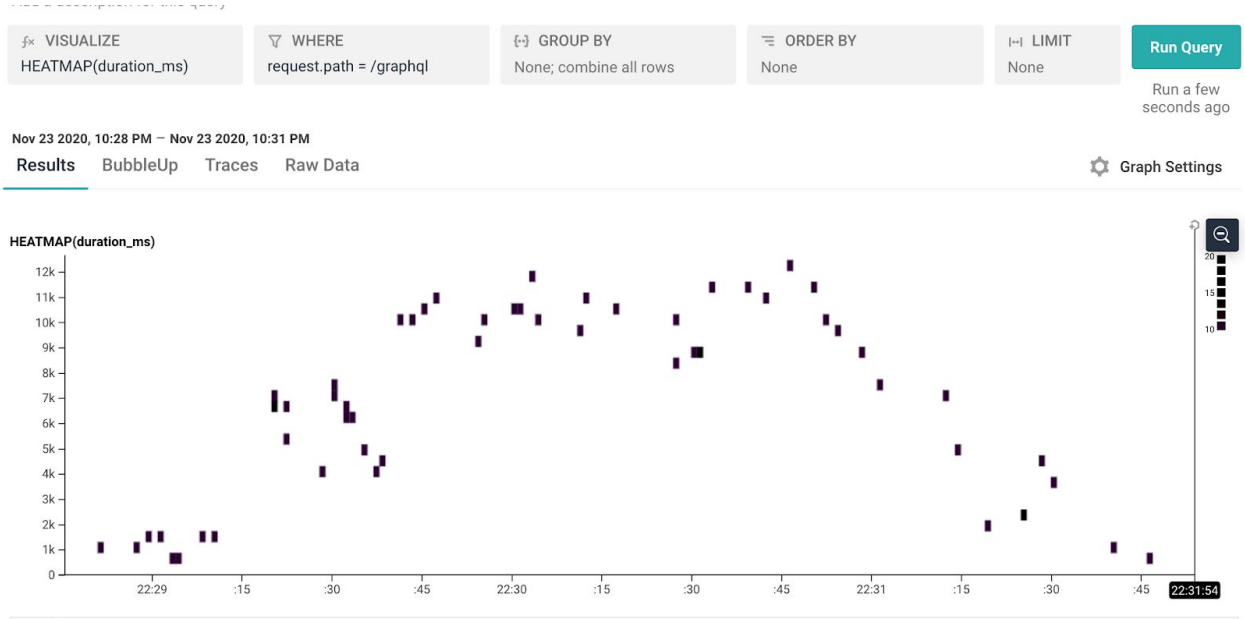
Next we needed to address the many times the below query was being run for every request.

```
UPDATE `art` SET `creatorId` = ? WHERE `id` = ?
```

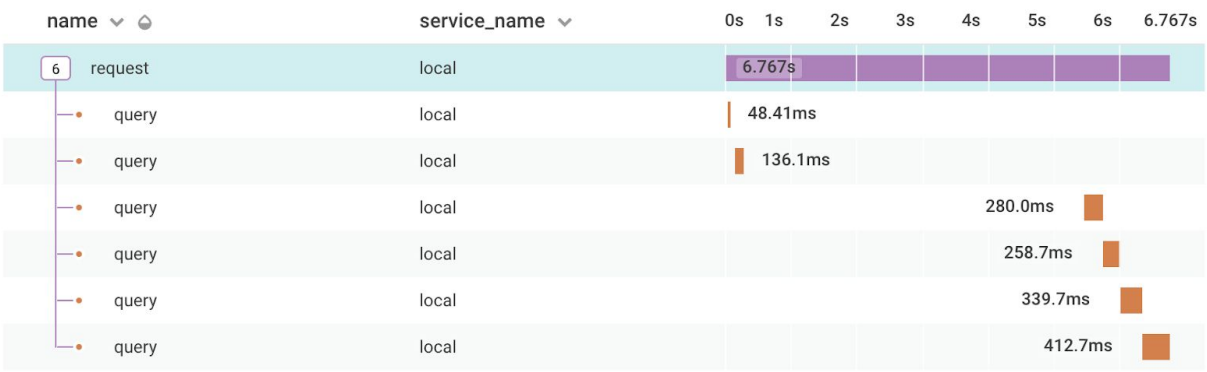
It looked like the database was taking a series of artwork and assigning it to our user who was creating the new artwork. Looking at the code, we saw in order to update the artwork by the given user, we were doing:

```
creator.artworkCreate.push(newArt)
// ...
await creator.save()
```

This was performing an expensive save operation on the user, setting all of their created artwork again in a single transaction after pushing a new piece of art. Eliminating the two lines above and just saving the new artwork in the database should ideally give the same result with a much better performance, since we don't have to resave already existing objects. Running our load test again after this deletion gave a much better result, showing only about 12 seconds of latency. Even better, we no longer received non-200 responses while load testing, indicating that this issue occurred when multiple uploads were being performed due to the redundant synchronization of the user's `artworkCreated` field.



Additionally, the sample trace from one of these requests confirms that we eliminated much of the unneeded querying, as seen below.



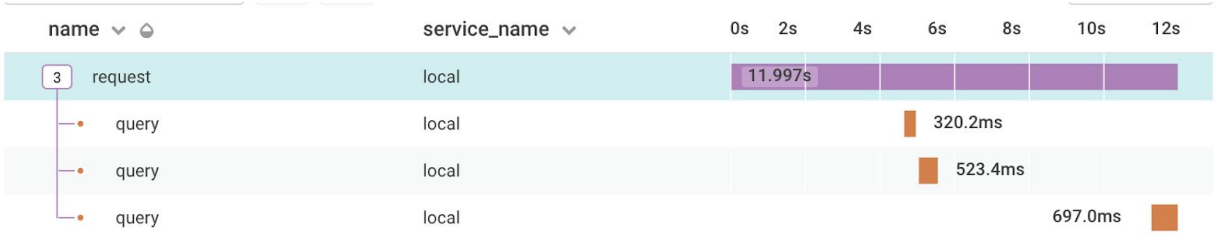
Unfortunately, we still had 6 queries, where we only needed a single INSERT statement. Looking at the code and the contents of these queries, we can piece together what is happening. At the beginning, the two requests are fetching the user context, which cannot be avoided. The 4 queries at the end are a transaction, and when pieced together, are equivalent to the following:

```
START TRANSACTION;
INSERT INTO `art`(`id`, `name`, `createdAt`, `uri`, `type`, `numReports`,
`creatorId`, `locationLat`, `locationLng`) VALUES (DEFAULT, ?, DEFAULT, ?,
?, ?, ?, ?, ?);
SELECT `Art`.`id` AS `Art_id`, `Art`.`createdAt` AS `Art_createdAt`,
`Art`.`numReports` AS `Art_numReports` FROM `art` `Art` WHERE `Art`.`id` =
?;
COMMIT;
```

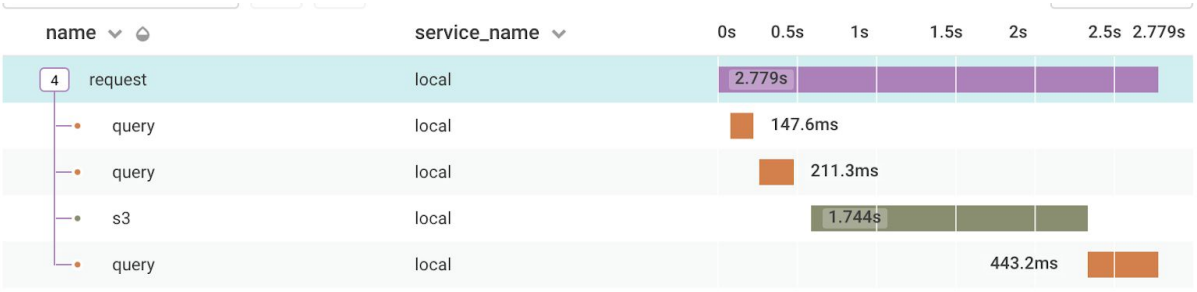
These queries create the new artwork, then get the same artwork out to return to the server, all wrapped in a transaction. However, by the design of our GraphQL schema, the `addArt` mutation only returns success or failure, so getting the result back from the database is superfluous. We can attempt to remove this returning by using a {reload: *false*} flag. With that removed, since MySQL automatically does isolation for single statements, we can also remove transactions. This changes our code:

```
// from
await newArt.save()
// to
await newArt.save({ reload: false, transaction: false })
```

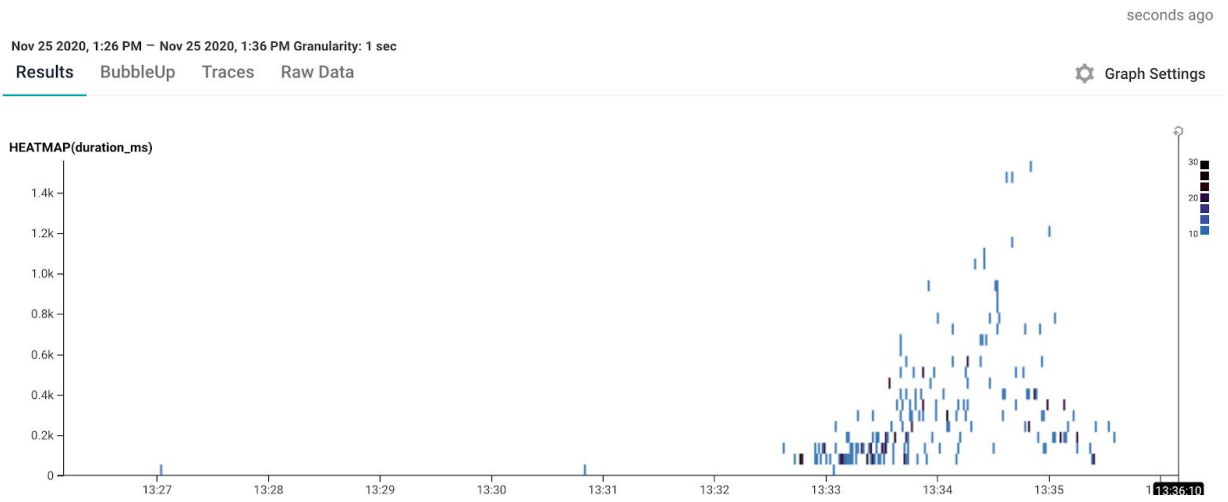
Unfortunately, these changes proved to have little impact on our performance, and after rerunning the load test, we received traces like the following.



While the uploading artwork has been condensed to a single query, we now have a large gap between our user authentication and the art upload. By inserting a new honeycomb span, we were able to trace around what we hypothesized to be the cause of this latency; the upload to AWS S3 bucket. Inserting a span around the uploading function yielded the following result, confirming our suspicion that the critical section was now uploading to S3.

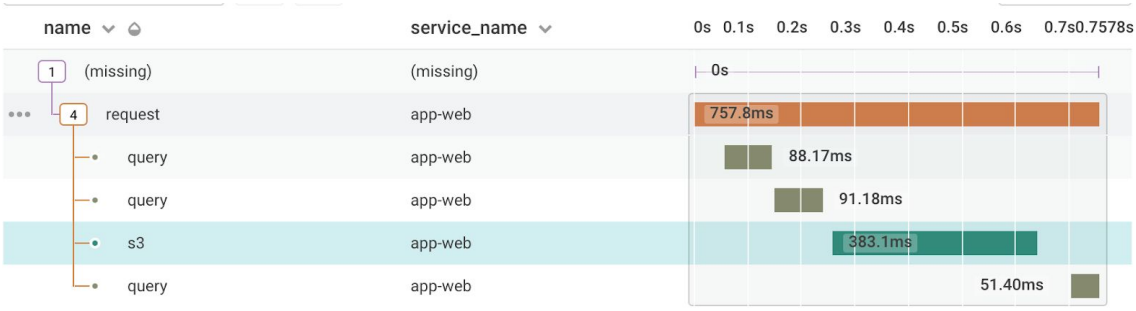


Once this was clearly the issue, we reasoned about the cause. As we can see, now a large percent of the time is spent uploading to the S3 bucket, upwards of 3 seconds in some cases. It shouldn't be the case that a small, less than 1 kilobyte file takes 3 seconds to upload. One of the major slowdowns here is the network bottlenecks. This testing was being done on a laptop, using a wireless connection. Packet loss and the number of hops from the testing location (Los Angeles) to Amazon's S3 servers in the us-west-2 datacenter in Oregon were likely causing delays. The only resolution to this problem was to collocate our application server and the S3 bucket. In order to check this issue, we deployed the code to AWS and ran the same load test. As seen below, our results significantly improved.



50 VUs

Our latency didn't exceed 1.5 seconds, and most were under 1 second in the same 50 VU load test. This improvement came mainly from the greatly reduced latency with the S3 bucket, as seen in a sample trace below.

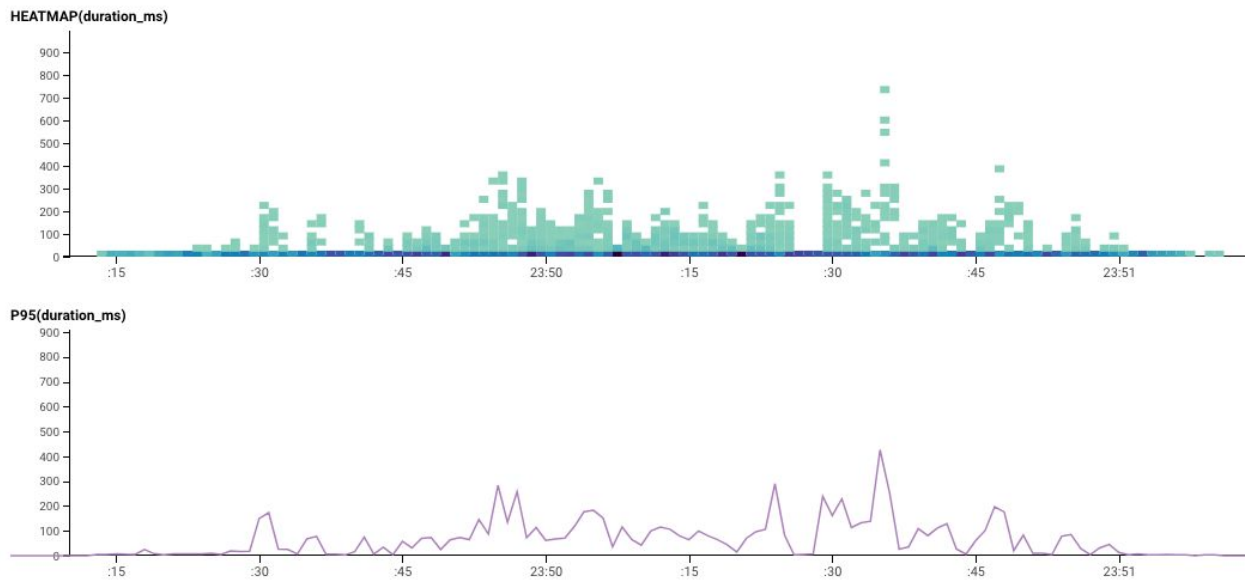


Overall, through the load test we were able to identify bugs in the software, test our deployment process, and achieve a nearly 30x improvement in P95 performance with 50 VUs.

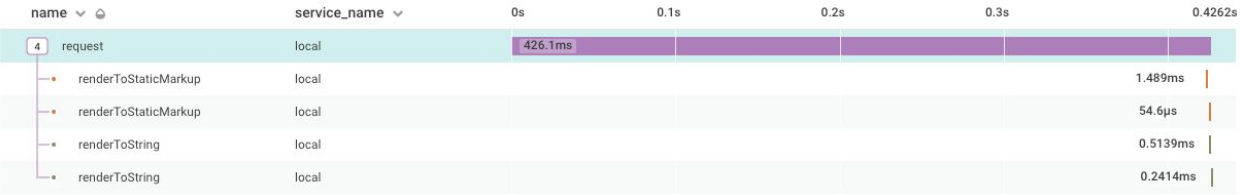
Background Process

Baseline

In its current state, our app does not have a background process. Although the requirement to have a background process that regularly performs a computation was removed as a requirement, we still wanted to observe the performance implications behind having one, and how it impacted the other computations on our app. As a control variable, the following shows the app's performance without any background process run, using the mapScript.js load test, which simply accesses the map page. The load test first ramps up to 300 virtual users in the first 50 seconds, decreases to 200 after another 50, and finally reaches 0 after 20 more seconds.



As the graph demonstrates, the duration for the majority of requests roughly stayed consistent throughout the load test, with the highest spike in the P95 taking 426.1 milliseconds. All in all, performance remains within the acceptable range, and users do not have to wait a long amount of time before they receive a response for their request. We can view a trace of the longest request to see the breakdown of what takes time:



Although the rendering done at the end is performed very quickly, the majority of the time is spent in the beginning, which is mostly attributed to both the waiting time for prior requests to complete and the overhead from the Express server itself.

Hypothesis

In order to represent a background process, we created a function that regularly repeats a cryptographic hashing computation on a long string for a user-specified number of times every two seconds.

```
function runBackgroundProcess() {
  setInterval(() => {
    for (let i = 0; i < 10; i++) {
      const data =
        'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTU
      require('crypto').createHash('sha256').update(data).digest('base64')
    }
  }, 2000)
}
```

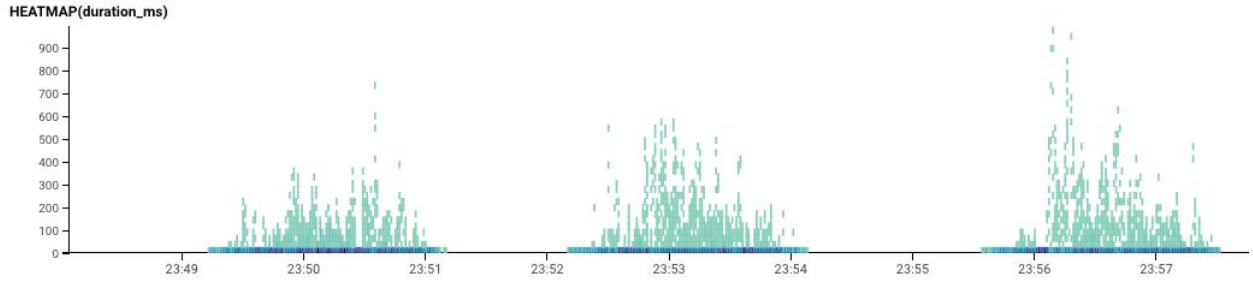
An instance of this background process is set up when the server is set up, like so:

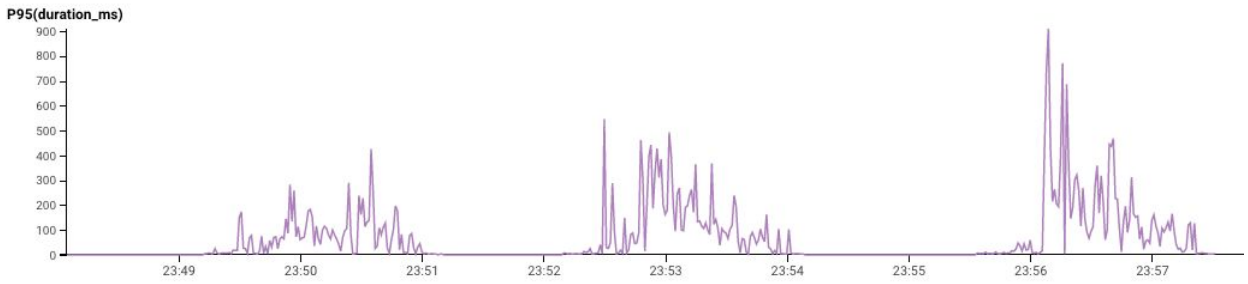
```
server.express.use(cookieParser())
server.express.use(json())
server.express.use(raw())
server.express.use('/app', cors(), expressStatic(path.join(__dirname, '../../public')))

// run background process
runBackgroundProcess()

const asyncRoute = (fn: RequestHandler) => (...args: Parameters<RequestHandler>) =>
  fn(args[0], args[1], args[2]).catch(args[2])

/* ... server routing .... */
```

The general hypothesis was: the more computationally expensive this background process is (in our case, the more times that the hashing algorithm repeats), the higher the duration of each virtual user's request will be, especially at a high number of concurrent users.

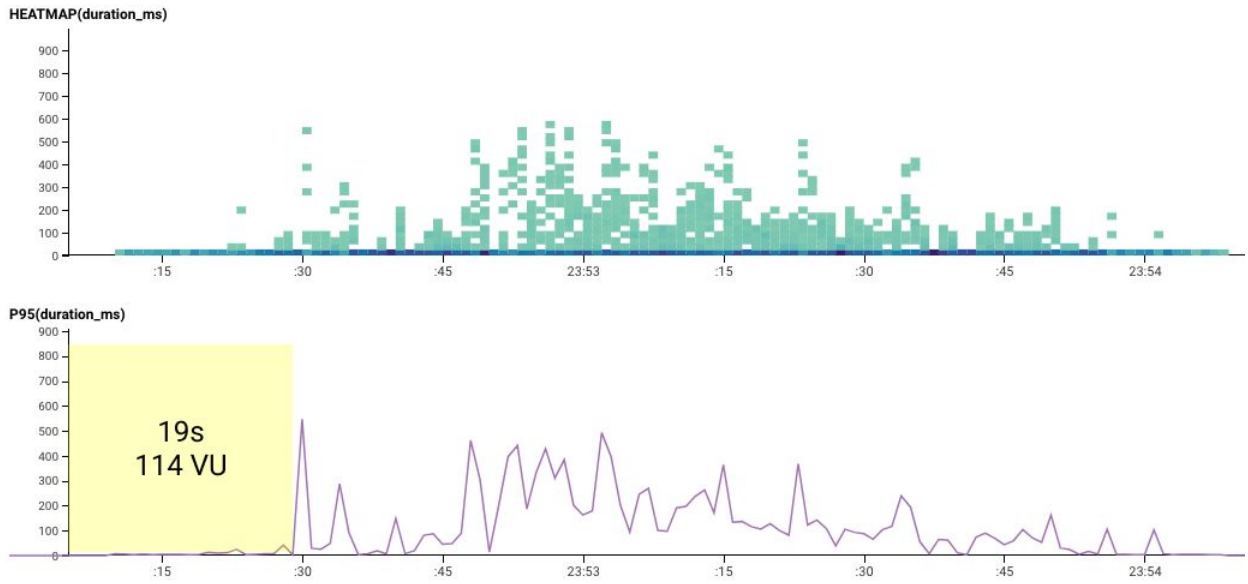
Experimentation/Improvements

In order to test the relationship between background processes and performance, we viewed the app's performance with the background process repeating the cryptographic hashing algorithm 10 times, and again with 20 times, in addition to the control test. The following graphs (from left to right) show the heatmap and P95 query results of mapScript.js on the app running with no background process, a background process that repeats the hash computation 10 times, and a process that repeats the hash computation 20 times.

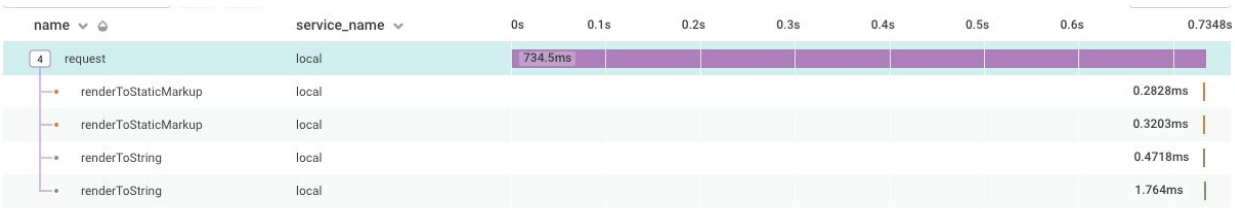




We have already viewed the baseline performance of the app, which is the leftmost section of this query. We can now take a closer look at the P95 results of the load test on the app running a background process repeating its hash computation 10 times:

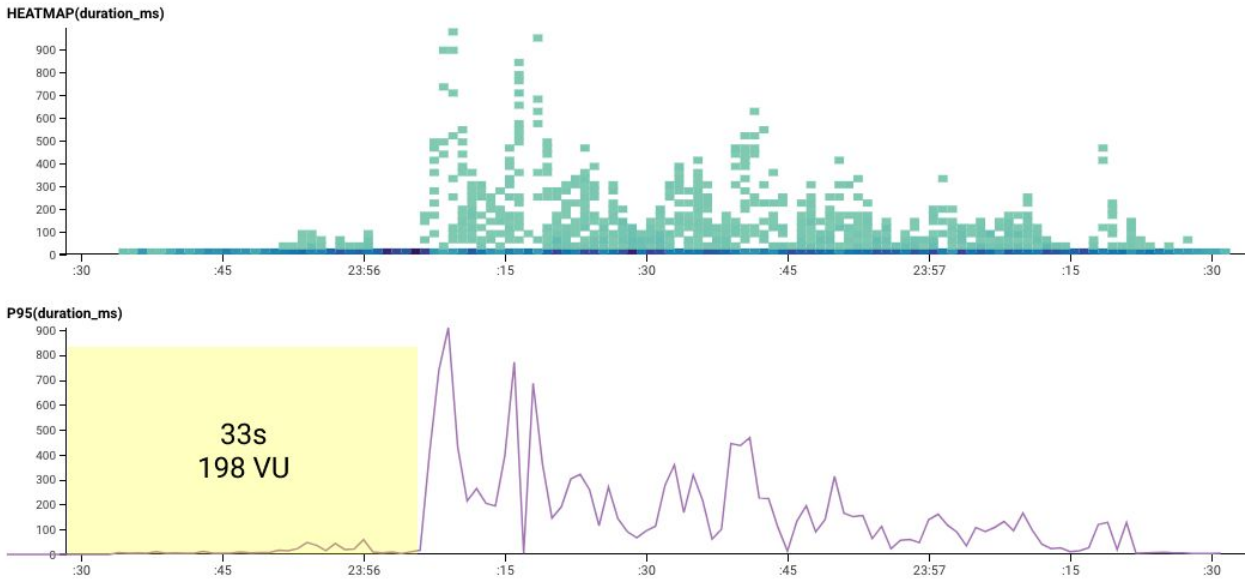


At the beginning of the run, the addition of the background process does not seem to interfere with the response time of requests at all. However, there is a spike in request duration that occurs at 19 seconds, at which point there are 114 virtual users. There continue to be duration spikes approximately every 2 seconds, which directly correlates to every time the background process computations are run. The following shows a trace of the longest request, which takes 734.5 milliseconds:

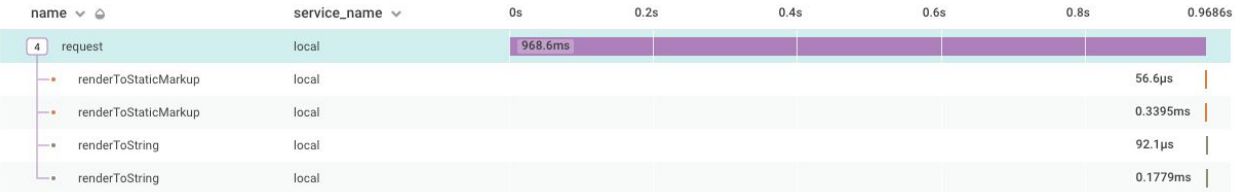


It is evident that there is little to no change to the rendering done at the end of the request. The section of the request that increases is the initial waiting time due to the addition of the background process.

Once we dialed the number of repeated hashing computations to 20 for our background process, the spikes became even more egregious:



With an increase in computation, duration also further increases. The initial spike in duration shown in the graph occurs at approximately 33 seconds when there are 198 virtual users. Afterwards, duration continues to be higher for each request. The trace of the initial spike, which is also the maximum duration, is shown below:



Again, the section of the request that worsens with the increased number of computations is the beginning, where this request has to wait for the background process and other previous requests to finish before running. If a user has to wait nearly a second for their request to complete, it is likely that they have already clicked off the site.

Every 2 seconds, `setInterval()` adds the repeating cryptographic hashing function to the Node.js event queue. Because Node.js is a single-threaded environment, the computations *must* be run to completion before anything else can be run, and there is no chance to parallelize across multiple cores through the usage of multi-threading. As the amount of computational overhead increases within the background process, each request is shown to take an increased

amount of time before it completes, eventually becoming a bottleneck for users, who are forced to wait for the background process to finish before their request can be fulfilled.

Although this background process was added for testing purposes, and we removed it from the final product afterwards, one way we could potentially improve the performance of a permanent background process is by repeating it as sparsely as possible and increasing the time interval on `setInterval()`. If a background process does not need to be repeated so frequently, it is best to limit the number of times it is run to avoid severely degrading performance. Another solution if the background process is particularly computation-heavy is to run it asynchronously in a separate process, outside the application server in a different machine. This way, hardware resources such as CPU and RAM do not need to be shared, and the single thread of Node.js can dedicate itself to pure application logic while the background process can regularly run elsewhere, potentially concurrently, without directly blocking app progress.

Conclusion

Feature Improvements

Due to time constraints, we were unable to implement all the features from our initial design, and instead opted to focus on the features that would best test scalability improvements. If time permitted, however, we would have still implemented them to further enrich our app.

One feature we want to expand, for example, is the media types users are able to upload. Currently, the application supports entering text and uploading .jpg and .png images, but we also want to support the upload of text, audio, and video files in the future to allow for different mediums of art to be displayed on the platform.

Adding functionality to actively keep track of how many “views” or “likes” each art piece was also originally planned for, as demonstrated by the inclusion of `views` and `likes` columns in our Art entity. This would allow art that many people appreciate to gain traction and visibility among the community of users on our application.

We also designed a few pages that weren’t implemented. One such page that was originally designed for was a “Create art” option, where users could choose to draw their own art given a whiteboard-type feature on the “Share” page of the application, allowing for digitally created art or editing of image files. This is not a high-priority feature; however, it could be an interesting addition to the application.

Having a profile and settings page for each account would have also provided a more complete product experience. This would allow a user to change the username, email address, or password associated with their account after initial account creation. It would also allow them to include a picture and bio that others on the platform could potentially view upon discovering their art.

We also ideated additional features that we didn’t originally design for in the process of building the application.

Upon building the map page with visual differences between visited and unvisited artwork, we realized that another feature that could be helpful is a visual indicator to differentiate the user's own art from that of other artists on the platform. This could be accomplished by changing the color of the star and shadow of the list item on the map page to our tertiary color, orange, when the creator ID of the artwork matches that of the users. This could allow users to more clearly distinguish whether their art has been published on the map, and prevents confusion between art pieces by other creators that the user may wish to view.

Another feature that could help the user navigate the platform is to allow users to hover over stars on the map to see titles of artwork but click on the stars to see the modal, in the same way that they currently can access the modal while clicking on the cards. This would allow them to view a piece with a location and title that intrigues them, rather than having to scroll through the list of cards to view an art piece that they find on the map.

For true engagement on the application, improvement to the content quality and social component of the platform are important. For example, on Instagram, users can see content according to the number of likes it already has or filtered to be from just accounts that they follow. Likewise, Wanderlust could use such features to serve as an incentive for users to come back to the platform. In the future, we could implement a "ranking algorithm" for the art a user sees in their list that's based on the amount of views or likes on each piece and whether it comes from artists whose artwork they have previously liked. Viewing art that is more attuned to popularity or their preferences could improve content quality for each user, improving their engagement with the app. We could also create a feature where users could follow other users via their profile pages, and be notified when someone they follow has uploaded new work. This could potentially incentivize users to travel to a new place to view the art through the app if the artist has posted it from a unique location.

Increasing Availability

By deploying our app on AWS, we provided the infrastructure necessary to make it highly available. However, we did not take full advantage of the AWS toolkit, as our app was only deployed to the *us-west-2* region. If the number of users drastically increases, and this *us-west-2* region were to experience some unfortunate event that would bring down the entire server, such as a natural disaster, the service would just be down with no recovery.

To avoid indefinite outages, and to always provide users with a running application, we can increase the availability of our application by creating app copies that are deployed across multiple regions and availability zones. This way, if an app running in one particular region goes down, another region can temporarily be used to service requests. Although there may be increased network latency due to the longer distance between the user and this temporary server, this is a preferred alternative to not being able to use the app at all.

Horizontal Scalability

Currently, our app is housed in one EC2 instance, which means that if the user traffic becomes extremely high, there would be no way to dynamically increase computing resources. AWS provides the toolkit necessary to automatically scale; ECS (EC2 Container Service), for example, provides an auto-scaling container service that manages EC2 Auto Scaling groups without any further intervention from the developer after initial setup. ECS ensures that EC2 instances are able to be spun up automatically when user traffic is high; likewise, during downtimes, the amount of EC2 instances can be decreased, saving on resources. With the addition of auto scaling, the computational requirements of our app will always be dynamically met without us having to worry about manually increasing infrastructure. ECS also runs a load balancer (Elastic Load Balancer), which runs periodic health checks to ensure that services and tasks are running correctly. These features, which are provided out of the box by AWS, will undoubtedly help our app scale at large.