Gallery

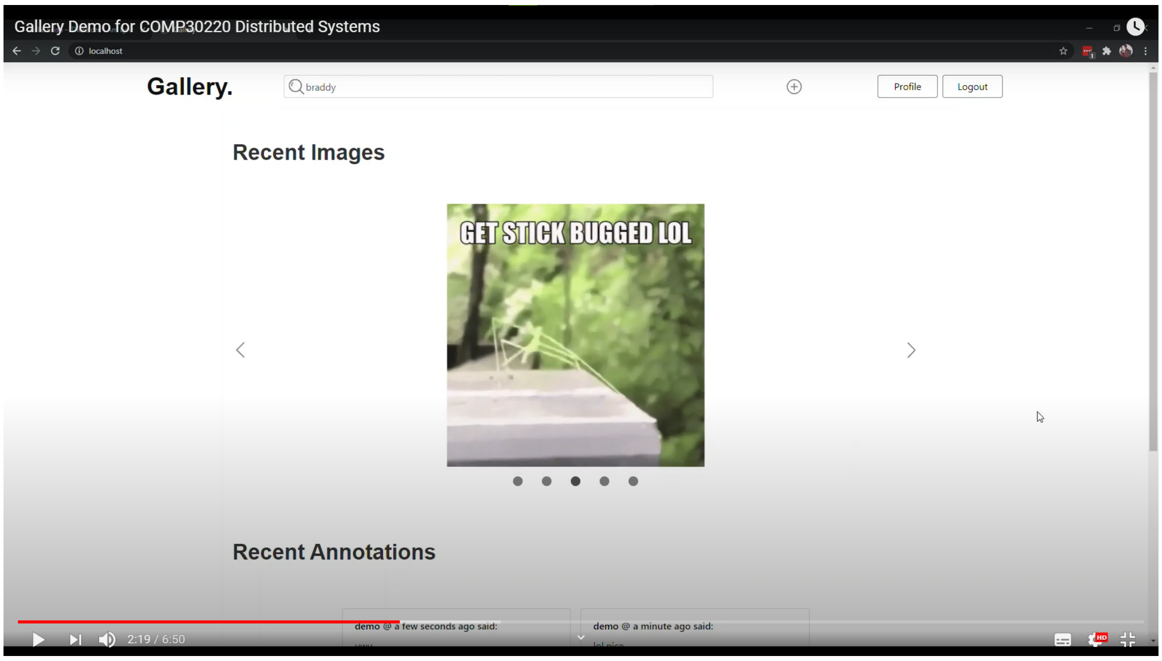
Intro

Social Gallery Image Annotation, think genius.com for images and Reddit up/down voting for the annotations.

Main features

- Signed in users can upload images (with titles and descriptions) and add annotations to uploaded images by drawing boxes over the desired areas.
- Annotations may be up or down voted similar to Reddit's commenting system.
- The homepage shows recent uploads and annotations.
- Users have profiles showing their uploaded images and annotations.
- Search is enabled for users by username and images by their titles and descriptions.

Demo video


Gallery Demo for COMP30220 Distributed Systems
localhost
Gallery.
braddy
Profile
Logout
Recent Images
GET STICK BUGGED LOL
Recent Annotations
demo @ a few seconds ago said:
demo @ a minute ago said:
2:19 / 6:50

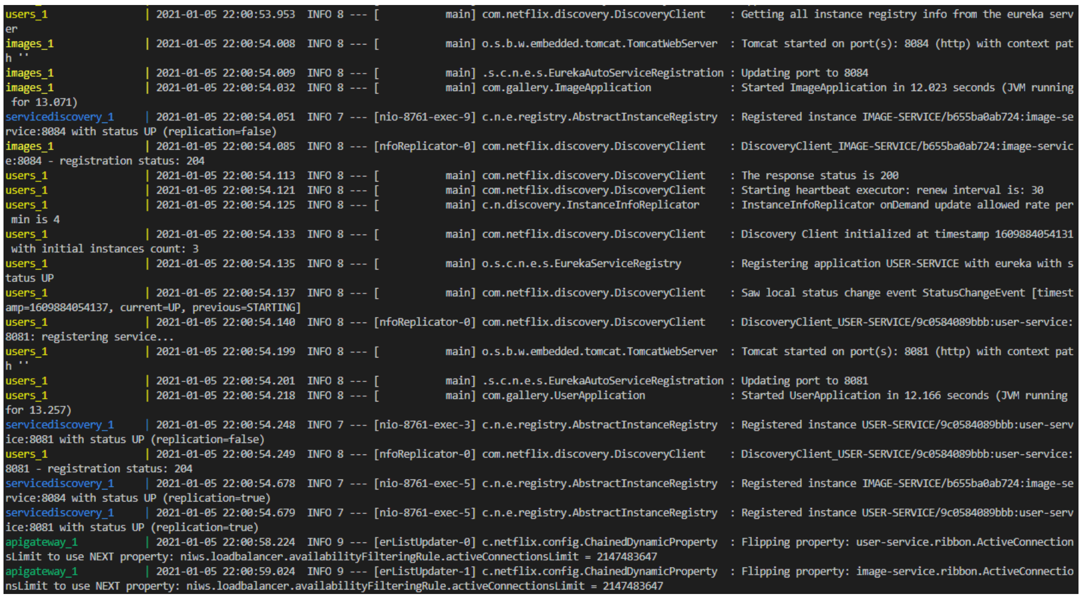
Demo video link

To run

1. At root, run `./run.sh fromscratch`, or in your terminal:

```
docker-compose down -v
mvn clean
mvn install
docker-compose up --build --remove-orphans
```

NOTE: This will take awhile because it mvn installs all services, npm installs the React node modules and runs the services in their containers (some containers wait a specific time due to dependance on other containers). The backend is ready when you reach this point, where user-service and image-service completed initialization:



```
users_1             | 2021-01-05 22:00:53.953  INFO 8 --- [           main] com.netflix.discovery.DiscoveryClient    : Getting all instance registry info from the eureka serv
er
images_1            | 2021-01-05 22:00:54.008  INFO 8 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8084 (http) with context pat
h ''
images_1            | 2021-01-05 22:00:54.009  INFO 8 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8084
images_1            | 2021-01-05 22:00:54.032  INFO 8 --- [           main] com.gallery.ImageApplication             : Started ImageApplication in 12.023 seconds (JVM running
 for 13.071)
servicediscovery_1  | 2021-01-05 22:00:54.051  INFO 7 --- [nio-8761-exec-9] c.n.e.registry.AbstractInstanceRegistry  : Registered instance IMAGE-SERVICE/b655ba0ab724:image-se
rvice:8084 with status UP (replication=false)
images_1            | 2021-01-05 22:00:54.085  INFO 8 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_IMAGE-SERVICE/b655ba0ab724:image-servic
e:8084 - registration status: 204
users_1             | 2021-01-05 22:00:54.113  INFO 8 --- [           main] com.netflix.discovery.DiscoveryClient    : The response status is 200
users_1             | 2021-01-05 22:00:54.121  INFO 8 --- [           main] com.netflix.discovery.DiscoveryClient    : Starting heartbeat executor: renew interval is: 30
users_1             | 2021-01-05 22:00:54.125  INFO 8 --- [           main] c.n.discovery.InstanceInfoReplicator     : InstanceInfoReplicator onDemand update allowed rate per
 min is 4
users_1             | 2021-01-05 22:00:54.133  INFO 8 --- [           main] com.netflix.discovery.DiscoveryClient    : Discovery Client initialized at timestamp 1609884054131
 with initial instances count: 3
users_1             | 2021-01-05 22:00:54.135  INFO 8 --- [           main] o.s.c.n.e.s.EurekaServiceRegistry        : Registering application USER-SERVICE with eureka with s
tatus UP
users_1             | 2021-01-05 22:00:54.137  INFO 8 --- [           main] com.netflix.discovery.DiscoveryClient    : Saw local status change event StatusChangeEvent [timest
amp=1609884054137, current=UP, previous=STARTING]
users_1             | 2021-01-05 22:00:54.140  INFO 8 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_USER-SERVICE/9c0584089bbb:user-service:
8081: registering service...
users_1             | 2021-01-05 22:00:54.199  INFO 8 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8081 (http) with context pat
h ''
users_1             | 2021-01-05 22:00:54.201  INFO 8 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8081
users_1             | 2021-01-05 22:00:54.218  INFO 8 --- [           main] com.gallery.UserApplication              : Started UserApplication in 12.166 seconds (JVM running
for 13.257)
servicediscovery_1  | 2021-01-05 22:00:54.248  INFO 7 --- [nio-8761-exec-3] c.n.e.registry.AbstractInstanceRegistry  : Registered instance USER-SERVICE/9c0584089bbb:user-serv
ice:8081 with status UP (replication=false)
users_1             | 2021-01-05 22:00:54.249  INFO 8 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_USER-SERVICE/9c0584089bbb:user-service:
8081 - registration status: 204
servicediscovery_1  | 2021-01-05 22:00:54.678  INFO 7 --- [nio-8761-exec-5] c.n.e.registry.AbstractInstanceRegistry  : Registered instance IMAGE-SERVICE/b655ba0ab724:image-se
rvice:8084 with status UP (replication=true)
servicediscovery_1  | 2021-01-05 22:00:54.679  INFO 7 --- [nio-8761-exec-5] c.n.e.registry.AbstractInstanceRegistry  : Registered instance USER-SERVICE/9c0584089bbb:user-serv
ice:8081 with status UP (replication=true)
apigateway_1        | 2021-01-05 22:00:58.224  INFO 9 --- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty  : Flipping property: user-service.ribbon.ActiveConnection
sLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
apigateway_1        | 2021-01-05 22:00:59.024  INFO 9 --- [erListUpdater-1] c.netflix.config.ChainedDynamicProperty  : Flipping property: image-service.ribbon.ActiveConnectio
nsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
```

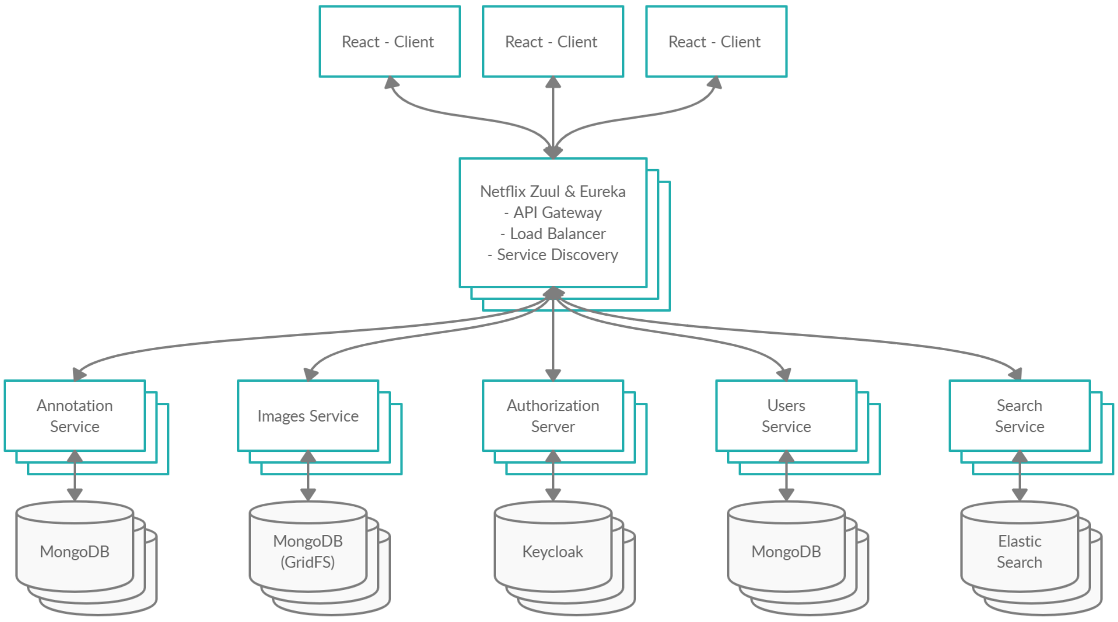
2. Open the browser and visit http://localhost/.

3. If you require running docker-compose a few more times:

    - you could skip `mvn clean` and `mvn install` step by running `./run.sh tearbuild`, or in your terminal:

      ```
      docker-compose down -v
      docker-compose up --build --remove-orphans
      ```

Tech stack



Eureka and *Zuul* act as the **service discovery** and **API gateway** of the application respectively, aka the "front door". Eureka keeps track of where the services are while Zuul routes every request (e.g. POSTing and GETing users/images/annotations) to the appropriate service. Zuul (complemented by Eureka) also provides load balancing between instances of the same service round-robin style, e.g. between multiple annotation services.

Users, images, and annotations are created and stored via their respective REST APIs and *MongoRepositories* (MongoDB).

The **authorization server** authenticates users via *Keycloak* (embedded in Spring Boot), an open-source identity and access management solution. In this project, Keycloak manages users and their access tokens, and Spring Boot handles the Keycloak Authorization Server.

Search is provided via *Elasticsearch* which indexes images (ids, titles, and descriptions) and users (ids and usernames) internally with Lucene. Our search service provides the APIs to talk to ES and determines how images and users are indexed in ES.

The **client** is a single-page application built with *React*, react-bootstrap, styled-components, Tachyons the CSS utility framework, and more (which can be found in `client/src/package.json`).

Example retrieval

User creation: user registers via client -> client POSTs user data to API gateway -> API gateway routes that request to user service -> user service tells Keycloak to create a new user and token -> Keycloak provides the token and success response to user service -> user service informs search service to index the new user -> search service sends success response to user service -> user service tells client via API gateway the user token and that user creation is successful and logged in -> the user token is stored in window.localStorage.

User is signed in but refreshed web page: user token is retrieved from window.localStorage -> client POSTs user token to API gateway -> API gateway routes request to user service -> user service verifies token with Keycloak -> Keycloak confirms with user service -> user service returns the user data the route it came from.

Dockerizing

We used docker-compose to dockerize all services similarly to the tech stack outlined. Each of the `images`, `users`, and `annotations` services are a container and have their respective MongoDB containers. They depend on the `authserver`. The `search` service has its own container but depends on the Elasticsearch container. All services (and their related MongoDB) depend on `servicediscovery` and `apigateway` services. Finally, the React client is installed and deployed in a NGINX container.

Reflection

Comparisons of our stack vs other tools

1. At the inception, NGINX was considered as our service discovery and API gateway, however we found that load balancing, key-value store and service discovery features are only provided in NGINX Plus (their paid version). Furthermore, we found that NGINX do not provide complex routing logic, whereas Zuul provides this feature with the power of the Java ecosystem/programming language.
2. MongoDB is an open source NoSQL database which makes it easy for us to change how user, image, or annotation data may be structured during development. The data is saved in JSON format, the most common for REST body/responses.
3. Embedding Keycloak in a Spring Boot application means that we don't need to download and setup Keycloak as a standalone server, simplifying the configuration. An adapter is provided for Spring Boot so we only need to add `keycloak-spring-boot-starter` as a pom.xml dependency.
4. Elasticsearch is built with Java on top of Apache Lucene, based on JSON, and suitable for NoSQL data. It is also open source, distributed, highly scalable, has overall good performance, and one of the us has some experience with it.
5. React vs Thymeleaf: Both can implement our client, but React (client side rendering) would be quicker at updating the UI than Thymeleaf (server side rendering). React can update parts of the webpage as users interact with it (e.g. up/down voting, adding annotations) while Thymeleaf regenerates the whole page each time.

Pros & cons of our stack

Pros	**Cons**
Loosely bound system where modifications or improvements of a part of the system have limited impact on other parts of the system.	There are additional points of failure in user or image creation due to the need of informing the service itself, the search service, and the authserver.

Pros	**Cons**
Easier to isolate, discern and troubleshoot causes of failure to a part of the system (e.g. to a single service).	
Easier management due to microservices mimicking most organization's management style (i.e. teams that manage different aspects of the system), therefore it was easier for us to divide the development task between each other.	

Stretch goals (not achieved due to time constraints, hardware constraints and configuration issues)

- We ran into issues configuring Redis to cache user data or aggregated responses at the API gateway. This would have shortened the hops between the client and Keycloak to verify the user token and prevent us from re-requesting data from user-service repeatedly. However, we would need to manually keep track if the cached data is outdated due to any recent edits and evict any old cached data following an LRU policy.
- Enabling editing and deleting features for the client. These are available via the API but not via the client.
- Refactoring the code further as there are some duplications, particularly in client.
- We ran into issues getting the authserver to register with service discovery automatically. This would have meant we did not need to tell the api gateway specifically where authserver is, and we could then have multiple instances of the authserver.
- Including an extensive test suite for both the application and the API enabling CI (continuous integration) to automate testing for an overall faster development process (e.g. trigger the tests each commit).
- Kubernetes, itself highly available and self-healing, would allow us to change the scale of our services with zero downtime so we can progress towards a more scalable fault tolerant application.
- Deploying to a web server, and implementing CD (continuous deployment).

Individual reflections

Bryan

I learnt that daily stand-ups allowed us to keep track of each others progress and allocate manpower to help accordingly. Furthermore, I learnt how to implement api-gateway and service discovery and understand how these technologies are integral to the overall application's architecture. I also learnt how to implement authentication in a distributed context, this requires setting up token-based authentication following Spring's new OAuth stack which includes a third-party authorization server (via Keycloak embedded into a Spring Boot instance for easier deployment) and configuring the resource servers (e.g. image-service, user-service, etc) to check if requests received for a particular endpoint has strict authentication rules (i.e. a POST to /api/users/auth/signin **does not require** user to be signed in, but a POST to /api/images **requires** users to be signed in). Other responsibilities include containerizing our architecture, implementing image-service with MongoDB's GridFS to store images into distributed capable BSON binary chunks and implementing the authentication, single-page routing, upload image and add annotation functionality of the React frontend client.

Emily

Responsibilities: Initial idea and wireframes, annotations service, working with Bryan for the client UI, testing the services with Braddy, report and video.

What I learnt:

- Daily stand-ups helps track progress.
- Merge requests make tracking features implemented easy.
- More understanding of and experience with REST and React, including getting and fetching from endpoints via Postman and Reach Hooks.
- React Hooks are hard.

Braddy

I implemented the search feature and learned that Elasticsearch is a very powerful tool, and is especially useful in terms of logging. On the topic of logging, I learned logs are very useful and help in debugging issues. In addition, I learned about containerising microservices, provisioning resources, and configuring the services. Automation is very important and useful, in particular, the healthchecks allowed be to determine the services are running okay. Furthermore, I learned a little bit about Kubernetes. In general, I learned how to design a distributed, fault-tolerant system, and how to architect neat and scalable code. Other responsibilities include writing Swagger API, profile and search React frontend and functionality, a little API testing, API design, implementing users service, and creating a single source-of-truth dotenv file location.

Summary of reflections

We used Spring Boot for the REST API in addition to Eureka (service discovery), Zuul (API gateway), MongoDB (database for each service), Keycloak (user authentication), Elasticsearch (search as a service great for NoSQL), React (client-side rendering), which are not covered in the module.

We structured the services such that we can change the scale of our application (e.g. having multiple instances of one service and having Zuul balance the load between them), and that one service going down does not mean other services go down like dominos, we just need to restart that service. Our stretch goals would have further improved fault tolerance for the application.

Acknowledgements

- Professor Rem Collier
- TA Eoin O'Neill
- Baeldung's articles.

Built by @bryansng, @lxemily, and @yeohbraddy in Nov 2020 - Jan 2021 for COMP30220 Distributed Systems.

▶ Instructions for developing locally without Docker