Golang Entry Task Report

Gerren Seow

May 2021

Table of Contents

1. Design Overview

1.1 Software Architecture

The software architecture comprises several docker container services, all of which are being orchestrated by Docker Compose. This helped to ease the integration of the different services since Docker-Compose launches and manages all containers concurrently.

NextJS a framework built on React, which builds into static pages. These static pages are served by a NodeJS server in the client container. Gin is used for server-side development, and lives in the server container. MySQL is the relational database management system used, and lives in the database container. Nginx serves as a reverse proxy, directing HTTP requests to client service for static assets and server for API endpoints. This architecture is also chosen with scalability in mind, since Nginx can serve as a load balancer if there is a sizable number of users.

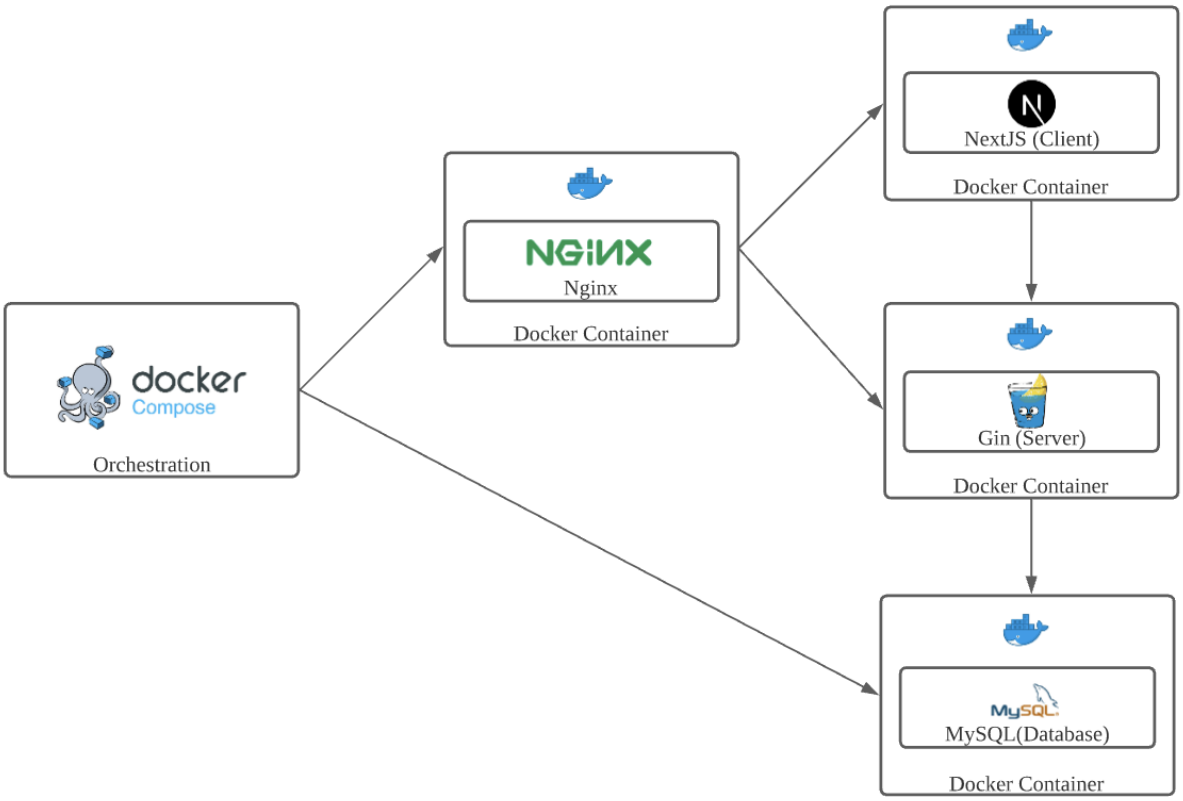


Diagram 1: Software Architecture

1.2 ER Diagram

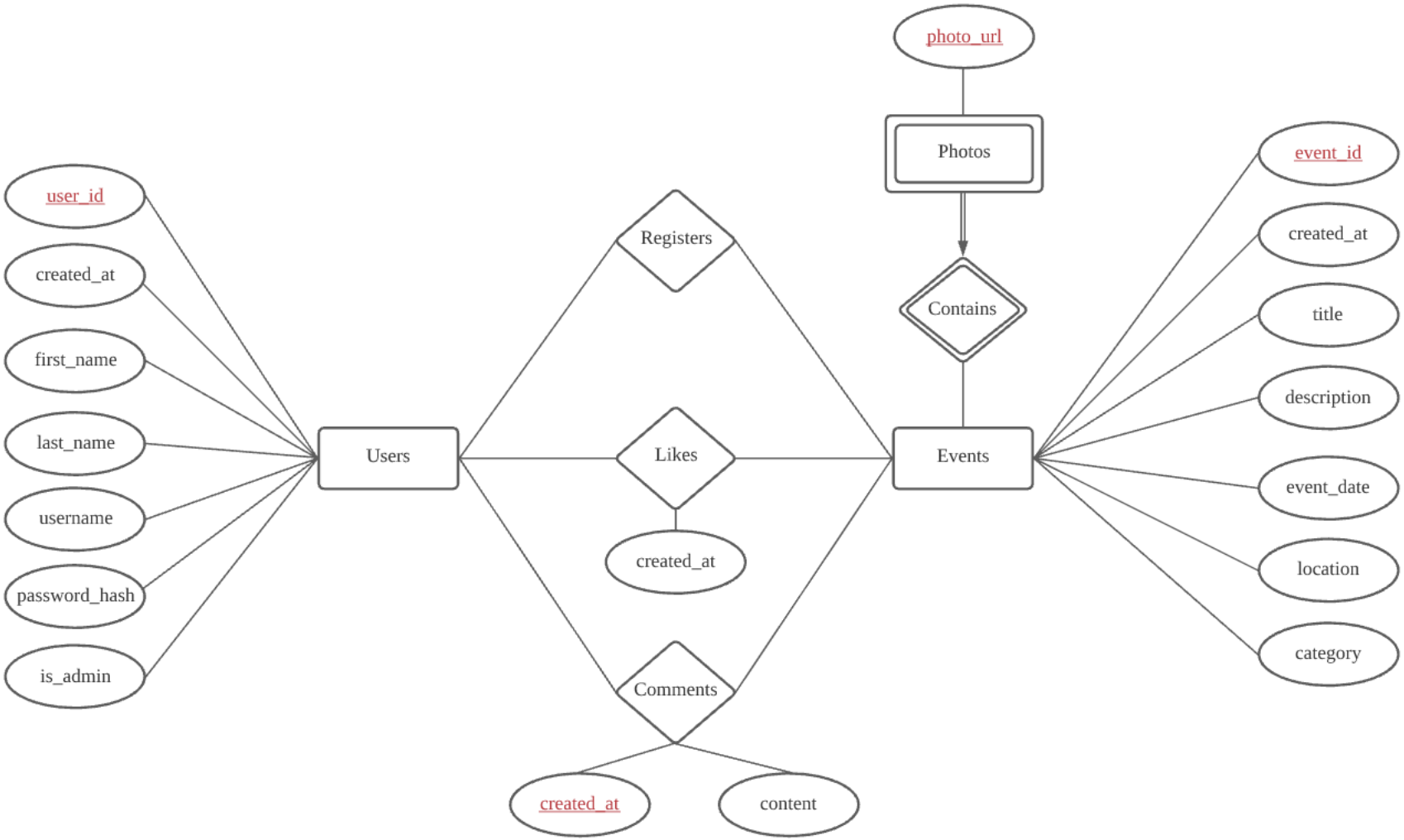


Diagram 2: ER Diagram

1.3 Relational Schema

```sql
CREATE TABLE users (
       user_id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
       created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
       first_name VARCHAR(255) NOT NULL,
       last_name VARCHAR(255) NOT NULL,
       username VARCHAR(255) NOT NULL UNIQUE,
       password_hash VARCHAR(255) NOT NULL,
       is_admin BOOLEAN NOT NULL
);

CREATE TABLE events (
       event_id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
       created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
       title VARCHAR(255) NOT NULL,
       description TEXT NOT NULL,
       event_date DATE NOT NULL,
       location VARCHAR(255) NOT NULL,
       category VARCHAR(255) NOT NULL
);

CREATE TABLE registers (
       user_id BIGINT UNSIGNED,
       event_id BIGINT UNSIGNED,
       PRIMARY KEY(user_id, event_id)
);

CREATE TABLE likes (
       user_id BIGINT UNSIGNED,
       event_id BIGINT UNSIGNED,
       created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
       PRIMARY KEY(user_id, event_id)
);

CREATE TABLE comments (
       user_id BIGINT UNSIGNED,
       event_id BIGINT UNSIGNED,
       created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
       content TEXT NOT NULL,
       PRIMARY KEY(user_id, event_id, created_at)
);

CREATE TABLE photos (
       event_id BIGINT UNSIGNED,
       photo_url VARCHAR(255),
       PRIMARY KEY(event_id, photo_url)
);
```

1.4 API Documentation

The APIs used in the application are documented below. This list might differ and appear fewer than the actual number of endpoints implemented in the backend. This is since additional routes were used as utility during the development process.

1.4.1 Authentication API

Endpoint	/api/auth/user	
Description	Check whether or not a user is logged in	
Request Type	GET	
Example Response	200	`{ "data": "User logged in" }`
	401	`{ "error": "Unauthorised access" }`

Endpoint	/api/auth/admin	
Description	Check whether or not an admin is logged in	
Request Type	GET	
Example Response	200	`{ "data": "Admin logged in" }`
	401	`{ "error": "Unauthorised access on admin resource" }`

1.4.2 Users API

Endpoint	/api/users/create	
Description	Create a user / admin profile	
Request Type	POST	
Request Body	`{ "first_name": "James", "last_name": "Tan", "username": "admin1", "password": "adminpw", "is_admin": true }`	
Example Response	200	`{ "data": { "user_id": 4, "created_at": "2021-05-18T00:49:42.789639+08:00", "first_name": "James", "last_name": "Tan", "username": "admin1", "password_hash": "$2a$10$6..m.pMmgjAX3/soflCdeuiW1jWRU6GF1/MXfKkyrLqj1oglhmB22", "is_admin": true } }`
	400	`{ "error": "Input mismatch" }`
		`{ "error": "Unable to hash password" }`
		`{ "error": "Creation of user failed" }`

Endpoint	/api/users/login
Description	Login a user / admin profile
Request Type	POST
Request Body	`{ "username": "admin1", "password": "adminpw"`

	}	
Example Response	200	{ "data": "Login successful" }
	400	{ "error": "Input mismatch" }
		{ "error": "Record not found!" }
		{ "error": "Invalid password" }
	422	{ "error": "Unable to create JWT token" }

Endpoint	/api/users/signout	
Description	Signout a user / admin profile	
Request Type	GET	
Example Response	200	{ "data": "Signout successful" }
	422	{ "error": "Unable to create JWT token" }

Endpoint	/api/users/current	
Description	Fetches the currently logged in user / admin profile	
Request Type	GET	
Example Response	200	{ "data": { "user_id": 4, "created_at": "2021-05-18T00:49:42.789639+08:00", "first_name": "James", "last_name": "Tan", "username": "admin1",

		"is_admin": true } }

	400	{
		"error": "User not found"
		}

1.4.3 Events API

Endpoint	/api/events/create	
Description	Create a event	
Request Type	POST	

Request Body	<pre>{
	"title": "SunNUS",
	"description": "Lorem Ipsum",
	"event_date": "21/05/2021",
	"location": "Sentosa",
	"category": "Sports"
	}</pre>	

Example Response	200	<pre>{
		"data": {
		"event_id": 1,
		"created_at": "2021-05-18T01:39:35.004575+08:00",
		"title": "SunNUS",
		"description": "Lorem Ipsum",
		"event_date": "21/05/2021",
		"location": "Sentosa",
		"category": "Sports"
		}
		}</pre>

	400	<pre>{
		"error": "Input mismatch"
		}</pre>

		<pre>{
		"error": "Creation of event failed"
		}</pre>

Endpoint	/api/events	
Query Parameters	?page=10&category=Sports&dateRange=2021/06/01:2021/06/05	
Description	Fetch all events paginated by pages of 10 and filtered by category and/or date range	
Request Type	GET	

Example Response	200	<pre>{
		"data": [
		{
		"event_id": 1,
		"created_at": "2021-05-17T16:34:59+08:00",</pre>

		"title": "SunNUS",
		"description": "Lorem Ipsum",
		"event_date": "2021-05-21",
		"location": "Sentosa",
		"category": "Sports"
		},
		{
		"event_id": 2,
		"created_at": "2021-05-17T16:34:59+08:00",
		"title": "SunNUS",
		"description": "Lorem Ipsum",
		"event_date": "2021-05-21",
		"location": "Sentosa",
		"category": "Sports"
		}
		]
		}

Endpoint	/api/events/count	
Description	Fetch count of events	
Request Type	GET	

Example Response	200	{
		"data": 8
		}

Endpoint	/api/events/categories	
Description	Fetch all available event categories	
Request Type	GET	

Example Response	200	{
		"data": [
		{
		"Category": "Social"
		},
		{
		"Category": "Sports"
		},
		{
		"Category": "Running"
		},
		{
		"Category": "Education"
		}

		] }

Endpoint	/api/events/:eventId	
Description	Fetch a specific event record	
Request Type	GET	

Example Response	200	{
		"data": {
		"event_id": 1,
		"created_at": "2021-05-18T01:39:35.004575+08:00",
		"title": "SunNUS",
		"description": "Lorem Ipsum",
		"event_date": "21/05/2021",
		"location": "Sentosa",
		"category": "Sports"
		}
		}

1.4.4 Photos API

Endpoint	/api/photos/:eventId	
Description	Fetch photo of an event	
Request Type	GET	
Example Response	200	`{ "data": [ { "event_id": 1, "photo_url": "uploads/event1-download.jpeg" } ] }`

Endpoint	/api/photos/uploadSingle	
Description	Upload a single photo for an event	
Request Type	POST	
Request Body (Formdata)	`event_id: 1` `photo: download.jpeg`	
Example Response	200	`{ "data": { "event_id": 13, "photo_url": "uploads/event13-download.jpeg" } }`
	400	`{ "error": "Unable to extract photo!" }`
		`{ "error": "Unable to create path!" }`
		`{ "error": "Unable to save file!" }`
		`{ "error": "Unable to parse event ID!" }`

		{
		"error": "Could not save photo information to database!"
		}

1.4.5 Registers API

Endpoint	/api/registers/event/:eventId	
Description	Fetch registrations of an event, and whether the logged in user has registered for it	
Request Type	GET	

Example Response	200	<pre>{
		"data": [
		{
		"User_Id": 1,
		"Event_Id": 1,
		"First_Name": "James",
		"Last_Name": "Tan",
		"Username": "admin1"
		},
		{
		"User_Id": 2,
		"Event_Id": 1,
		"First_Name": "Sarah",
		"Last_Name": "Tan",
		"Username": "admin2"
		}
		],
		"userRegistered": true
		}</pre>

Endpoint	/api/registers/create/:eventId	
Description	Register the logged in user for an event	
Request Type	GET	

Example Response	200	<pre>{
		"data": {
		"user_id": 1,
		"event_id": 1
		}
		}</pre>

	400	<pre>{
		"error": "Creation of registration failed!"
		}</pre>

Endpoint	/api/registers/delete/:eventId
Description	Remove the registration of the logged in user for an event

Request Type	DELETE	

Example Response	200	{
		"data": {
		"user_id": 1,
		"event_id": 1
		}
		}

	400	{
		"error": "Deletion of registration failed!"
		}

1.4.6 Likes API

Endpoint	/api/likes/event/:eventId	
Description	Fetch likes of an event, and whether the logged in user has liked it	
Request Type	GET	

Example Response	200	{
		"data": [
		{
		"User_Id": 1,
		"Event_Id": 1,
		"CreatedAt": "2021-05-17T09:43:49Z",
		"First_Name": "Gerren",
		"Last_Name": "Seow",
		"Username": "admin"
		},
		{
		"User_Id": 2,
		"Event_Id": 1,
		"CreatedAt": "2021-05-17T09:50:01Z",
		"First_Name": "Gerren",
		"Last_Name": "Seow",
		"Username": "user"
		},
		{
		"User_Id": 3,
		"Event_Id": 1,
		"CreatedAt": "2021-05-17T09:53:19Z",
		"First_Name": "Bryant",
		"Last_Name": "Khoo",
		"Username": "bryant"
		}
		],
		"userLiked": true
		}

Endpoint	/api/likes/create/:eventId	
Description	Logged in user likes an event	
Request Type	GET	

Example Response	200	{
		"data": {
		"user_id": 1,
		"event_id": 1,
		"created_at": "2021-05-17T18:12:49.76957613Z"
		}

		`}`

	400	`{`
		`"error": "Creation of like failed!"`
		`}`

Endpoint	/api/registers/delete/:eventId	
Description	Logged in user un-likes an event	
Request Type	DELETE	

Example Response	200	`{`
		`"data": {`
		`"user_id": 1,`
		`"event_id": 1,`
		`"created_at": "2021-05-17T18:12:49.76957613Z"`
		`}`
		`}`

	400	`{`
		`"error": "Deletion of like failed!"`
		`}`

1.4.7 Comments API

Endpoint	/api/comments/event/:eventId	
Description	Fetch comments of an event	
Request Type	GET	

Example Response	200	{
		"data": [
		{
		"User_Id": 3,
		"Event_Id": 1,
		"Content": "YO!",
		"CreatedAt": "2021-05-17T09:53:56Z",
		"First_Name": "Bryant",
		"Last_Name": "Khoo",
		"Username": "bryant"
		},
		{
		"User_Id": 2,
		"Event_Id": 1,
		"Content": "HELLO!",
		"CreatedAt": "2021-05-17T09:50:06Z",
		"First_Name": "Gerren",
		"Last_Name": "Seow",
		"Username": "user"
		}
		]
		}

Endpoint	/api/likes/create/:eventId	
Description	Logged in user comments on an event	
Request Type	POST	

Request Body	{
	"event_id": 1,
	"content": "Hello there!"
	}	

Example Response	200	{
		"data": {
		"user_id": 1,
		"event_id": 1,
		"created_at": "2021-05-17T18:16:55.631944212Z",
		"content": "Hello there!"
		}
		}

	400	{
		"error": "Creation of comment failed!"
		}

2. Installation and Maintenance

Installation and maintenance instructions are meant for virtual servers provisioned by TOC. For other servers, the instructions must differ slightly.

Installation:

1. SSH into virtual server
2. Copy the project repository into the folder of your choice
3. Ensure Docker is installed, otherwise install it using these instructions: https://www.simplilearn.com/tutorials/docker-tutorial/how-to-install-docker-on-ubuntu
4. Navigate into the root directory of the repository
5. Run `docker-compose up --build`
6. Application is now ready on port 80

Maintenance:

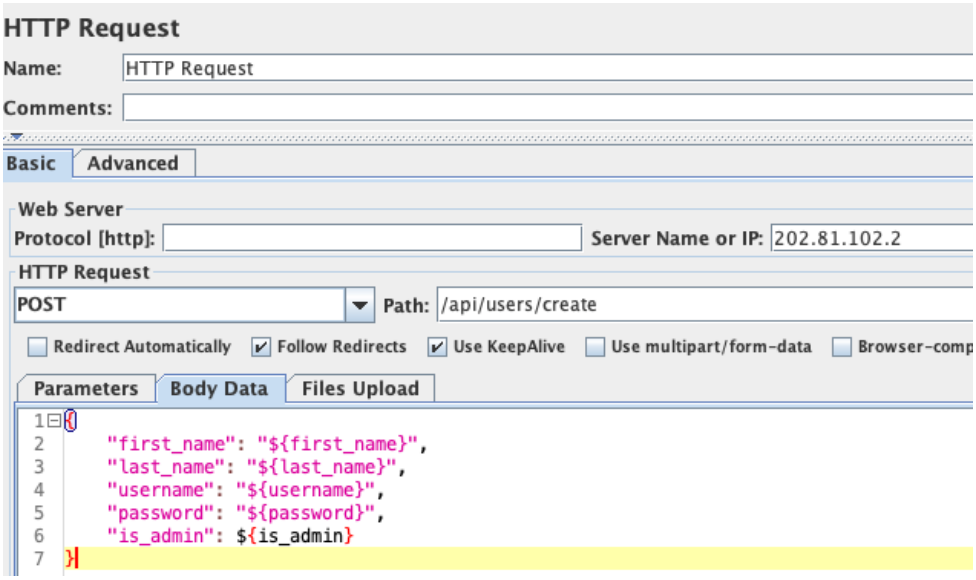
1. After installation, the containers should all be up and running, managed by Docker Compose
2. If there is a need to alter the project source code, run `docker-compose down`, make the changes and run `docker-compose up --build`
3. If there is a need to visualise the database or perform any mission critical queries, connect to it at port 3308 with MySQL Workbench (or your favourite SQL client). Credentials are indicated in the docker-compose.yml located in the root directory of the repository.

3. Performance Test

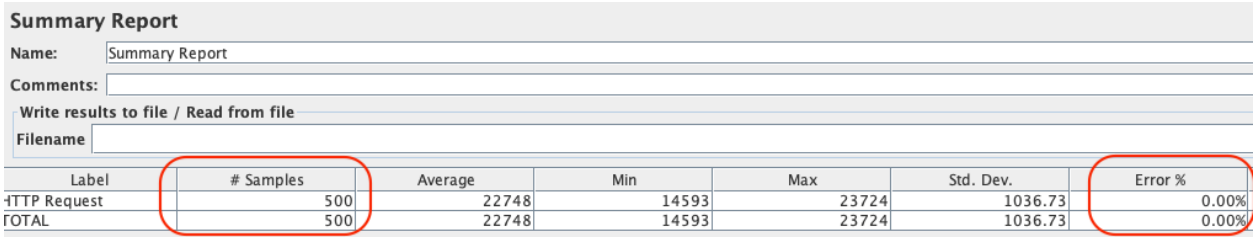
Performance testing included load testing on the API server. The tool used is JMeter, with 500 threads sending HTTP requests to the server in a period of 1 second.

3.1 Concurrent User Creations

Prior to load testing, a python script was written to generate 500 different sets of user information in the form of a CSV. This CSV is then loaded into JMeter, which sends 500 requests to the relevant API endpoint. The server was able to manage the concurrent creation requests well, with no errors. Included below are the screenshots of the API endpoint, summary report from JMeter and database state after the test.

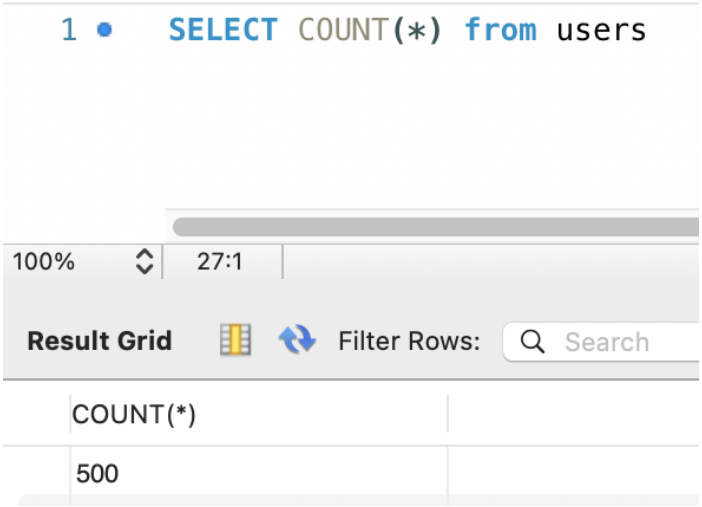


Screenshot 1: HTTP request sent by JMeter to create user



Label	# Samples	Average	Min	Max	Std. Dev.	Error %
HTTP Request	500	22748	14593	23724	1036.73	0.00%
TOTAL	500	22748	14593	23724	1036.73	0.00%

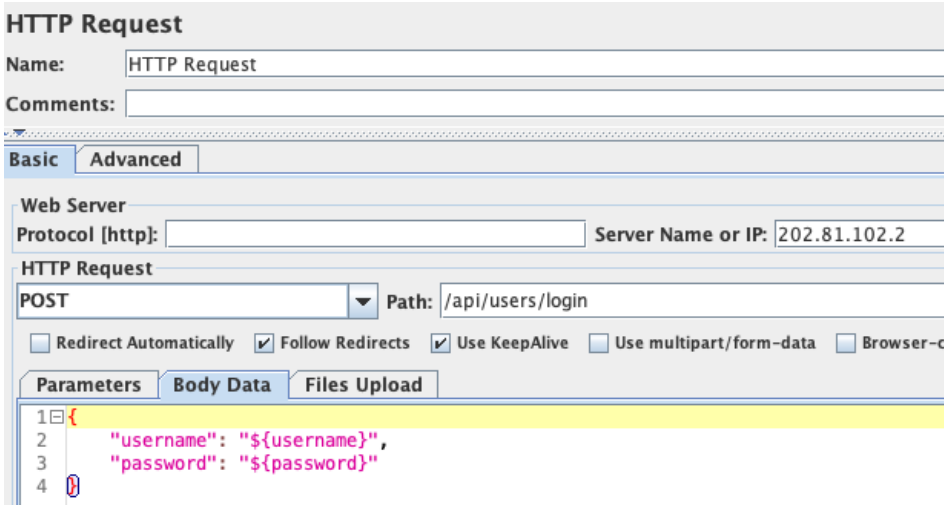
Screenshot 2: Load testing results



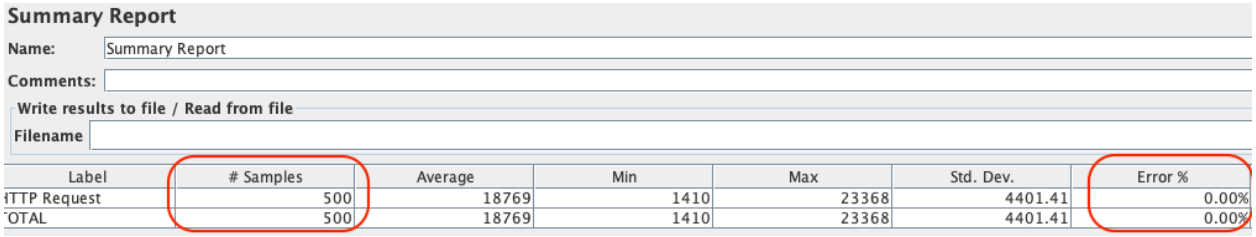
Screenshot 3: Database saving the creation of 500 users

3.2 Concurrent User Logins

The same methodology from above was adopted for load testing logins. A CSV file of 500 different login credentials is generated and loaded into JMeter, which sends 500 requests to the relevant API endpoint. The server was able to manage the concurrent login requests well, with no errors. Included below are the screenshots of the API endpoint as well as the summary report from JMeter.



Screenshot 4: HTTP request sent by JMeter to login user



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %
HTTP Request	500	18769	1410	23368	4401.41	0.00%
TOTAL	500	18769	1410	23368	4401.41	0.00%

Screenshot 5: Load testing results

4. Conclusion

The Golang entry task certainly presented a great challenge, as I had to extensively revise my existing knowledge and learn new technologies on the go. It was a hectic week, trying to glue everything together in a way which entails robustness, security and performance. Go was a language I always had an interest in, and this was a wonderful opportunity to dive in and get my hands dirty. I also had the chance to launch the entire application as an orchestrated set of Docker containers, a widely-used approach which I was keen to explore as well.

Overall, it was a good experience working on this entry task, which was likened to a ramp up for the rest of my internship. The short span of time spent allowed me to gain greater exposure to the tech stack used by the SeaCloud team, which will definitely serve me well as I take on product engineering tasks in the future.