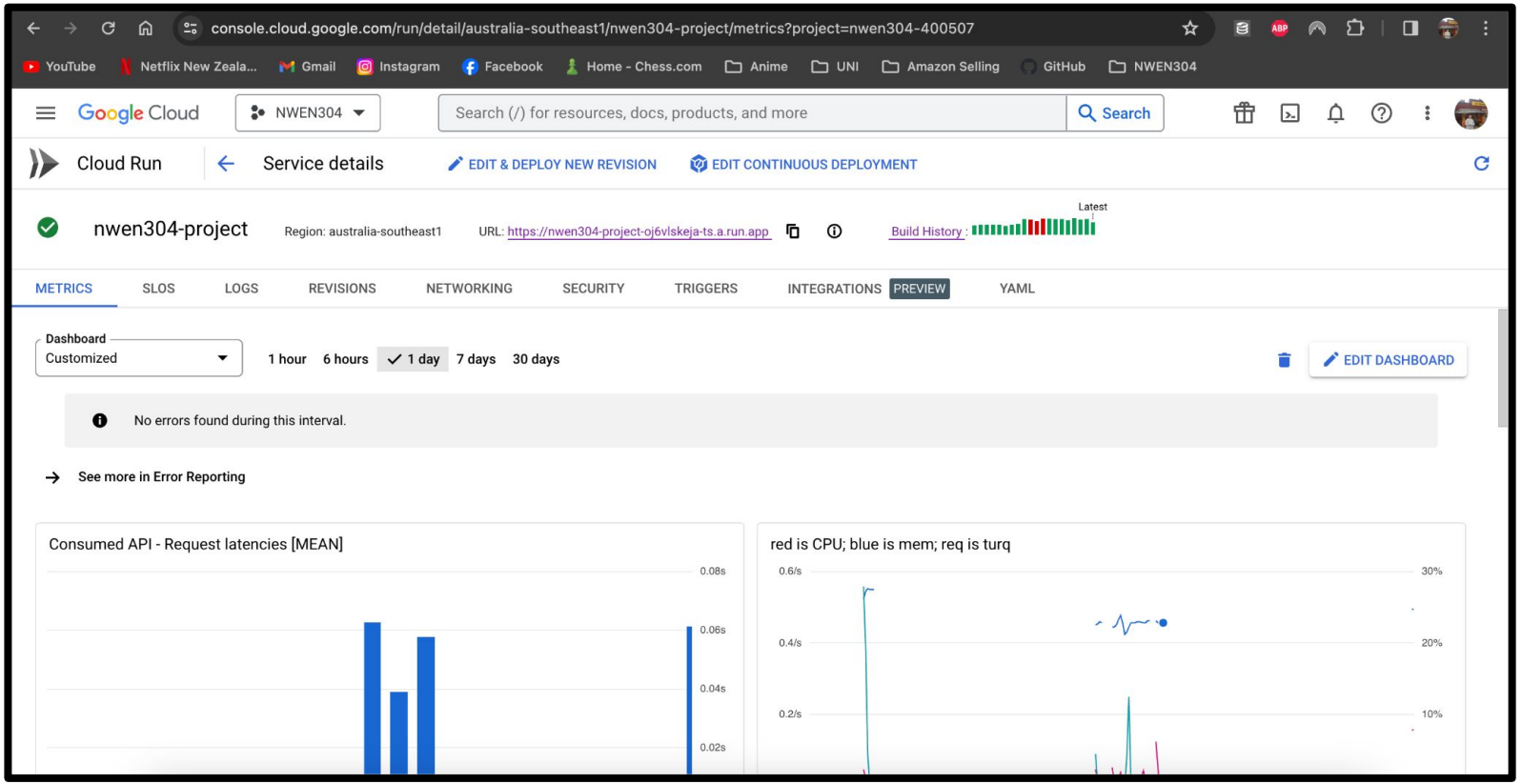
Nwen304 Group Project

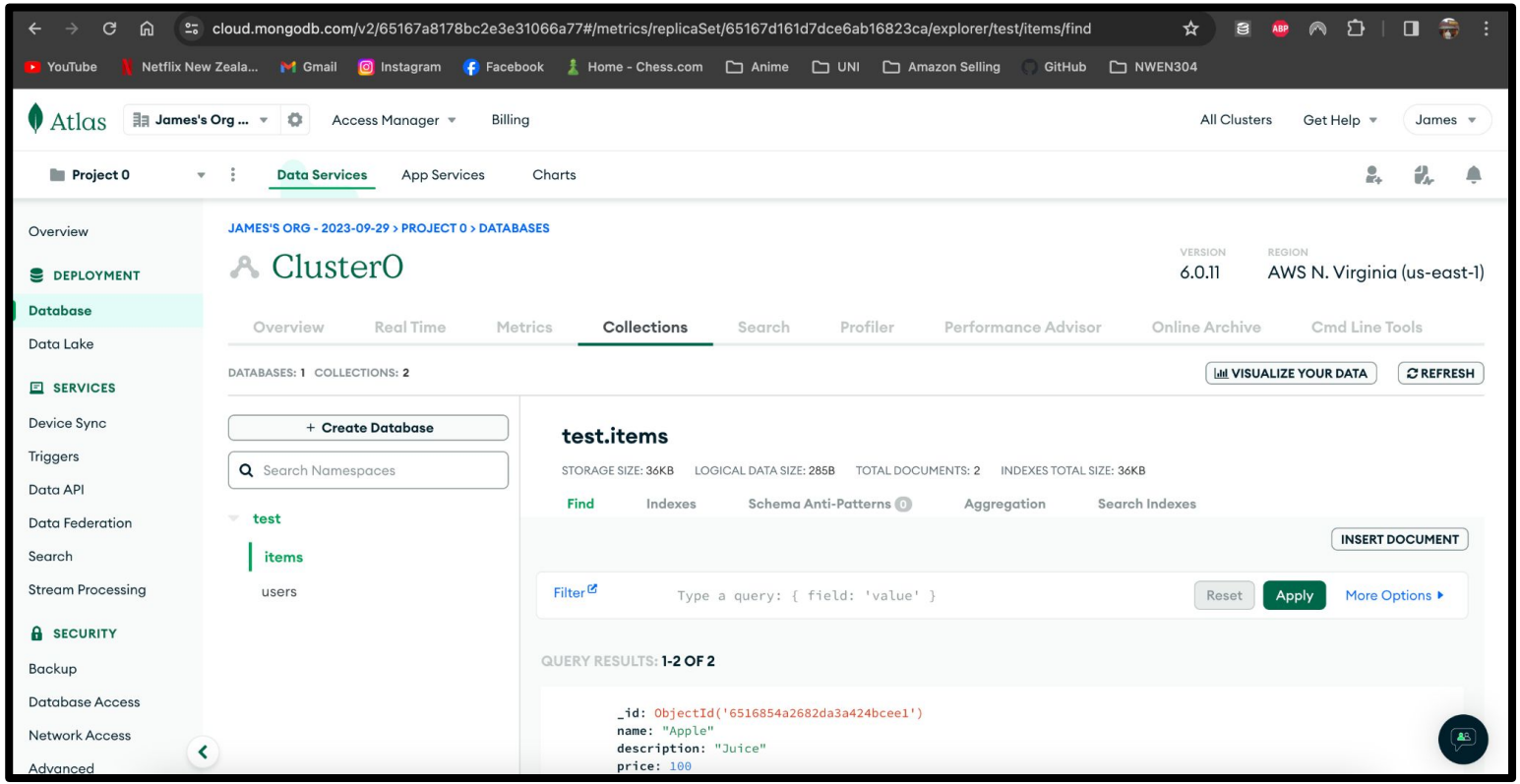
Workload Distribution

Taichi	James La	Together	James Boyle
- Google OAuth - Ensuring Authentication - Dynamic Endpoints - Use Session and Cookies - Privacy - Hashing - Password Reset	- Hosted (Cloud Run) - MongoDB - environments - Structure MVC architecture - Microservices	- Schema - Basic Login/Logout Function - Recommendation Services - Testing - Mocha Chai/Load testing - Timeout Function	- Render Testing - API keys -

Hosted on Google Cloud Run



MongoDB


cloud.mongodb.com/v2/65167a8178bc2e3e31066a77#/metrics/replicaSet/65167d161d7dce6ab16823ca/explorer/test/items/find
YouTube
Netflix New Zeala...
Gmail
Instagram
Facebook
Home - Chess.com
Anime
UNI
Amazon Selling
GitHub
NWEN304
Atlas
James's Org ...
Access Manager
Billing
All Clusters
Get Help
James
Project 0
Data Services
App Services
Charts
Overview
DEPLOYMENT
Database
Data Lake
SERVICES
Device Sync
Triggers
Data API
Data Federation
Search
Stream Processing
SECURITY
Backup
Database Access
Network Access
Advanced
JAMES'S ORG - 2023-09-29 > PROJECT 0 > DATABASES
Cluster0
VERSION
6.0.11
REGION
AWS N. Virginia (us-east-1)
Overview
Real Time
Metrics
Collections
Search
Profiler
Performance Advisor
Online Archive
Cmd Line Tools
DATABASES: 1 COLLECTIONS: 2
VISUALIZE YOUR DATA
REFRESH
+ Create Database
Search Namespaces
test
items
users
test.items
STORAGE SIZE: 36KB
LOGICAL DATA SIZE: 285B
TOTAL DOCUMENTS: 2
INDEXES TOTAL SIZE: 36KB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
INSERT DOCUMENT
Filter
Type a query: { field: 'value' }
Reset
Apply
More Options
QUERY RESULTS: 1-2 OF 2
_id: ObjectId('6516854a2682da3a424bcee1')
name: "Apple"
description: "Juice"
price: 100

Model

Schemas for Database

User Schema:

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
      username: String,
      email: String,
      password: String,
      resetToken: String,
      resetTokenExpiration: Date,
      location: String, //for recommendation
      purchaseHistory: [String]
});
const User = mongoose.model('User', UserSchema);
module.exports = User;
```

Item Schema:

```
const mongoose = require('mongoose');
const itemSchema = new mongoose.Schema({
name: {
type: String,
required: true
},
description: {
type: String,
},
price: {
type: Number,
},
image: {
type: String,
}
});
const Item = mongoose.model('Item', itemSchema);
module.exports = Item;
```

Security and Privacy

Using a combination of sessions, hashing and password validation, we improved security and privacy on our web application.

Session - expiry

Session

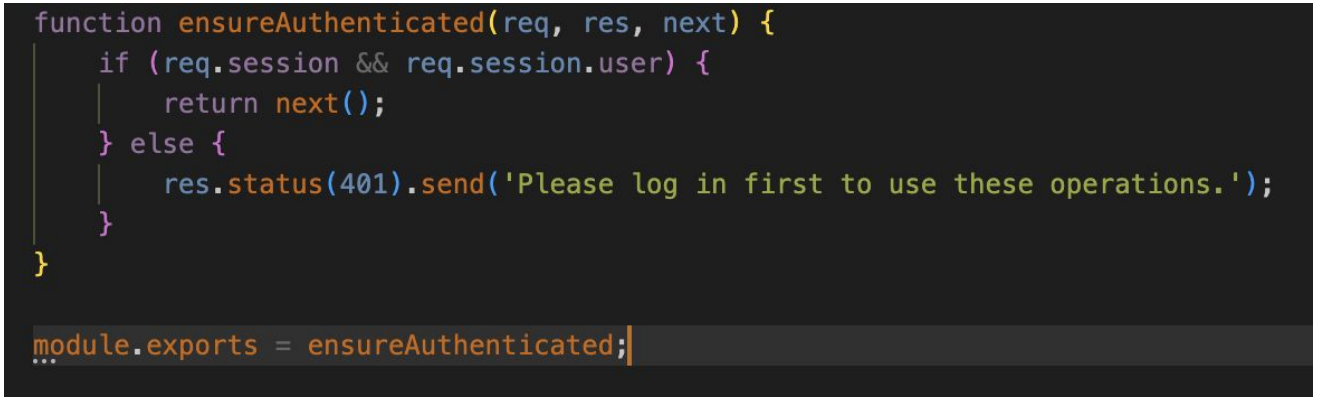
```
app.use(session({
    secret: 'your_secret_key',
    resave: false,
    saveUninitialized: true,
}));
```

```
let timeout;

function resetTimer() {
    clearTimeout(timeout);
    timeout = setTimeout(logout, 30000); // 30秒後にlogout関数を実行
}

function logout() {
    //session logout if user is not acdtive for 10 seconds
    fetch('/logout', {
        method: 'POST'
    }).then(() => {
        alert('Session timed out due to inactivity.');
        window.location.href = '/login';
    });
}
```

```
app.post('/purchase/:itemId', ensureAuthenticated, async (req, res) => {
    try {
        const user = await User.findById(req.session.user._id);
        if (!user.purchaseHistory) {
            user.purchaseHistory = [];
        }
        user.purchaseHistory.push(req.params.itemId);
        await user.save();
        res.json({ msg: 'Item purchased' });
    } catch (error) {
        console.error(error);
        res.status(500).send('Server error');
    }
});
```

Ensure Authenticated

Using the below function, we ensured the user was authenticated before allowing them to use certain functions like posting or deleting.



```
function ensureAuthenticated(req, res, next) {
    if (req.session && req.session.user) {
        return next();
    } else {
        res.status(401).send('Please log in first to use these operations.');
    }
}

module.exports = ensureAuthenticated;
```

Hashing

Using the bcrypt module, we hashed the given passwords

```
try {
    const existingUser = await User.findOne({ username: req.body.username });
    if (existingUser) {
        return res.render('register', { errors: ['Username already exists'] });
    }

    const hashedPassword = await bcrypt.hash(req.body.password, 10);
    const newUser = new User({
        username: req.body.username,
        email: req.body.email,
        password: hashedPassword
    });
    await newUser.save();
```

Dynamic Endpoints

To improve performance, scalability, and overall system reliability, we ensure that the resources are accessed or distributed in an optimal and responsive manner using dynamic endpoints.

```
app.get('/register', userController.getRegister);
app.post('/register', userController.postRegister);
app.get('/privacy-policy', userController.getPrivacyPolicy);

app.get('/', (req, res) => {
    res.render('home');  //to home.ejs
});
```

Viewer (Front-end)

Server-Side Rendering

Improved performance on first page load and better SEO. Also, less JavaScript is required on the client side, which reduces the load on the browser.

Example: EJS templates such as item.ejs and login.ejs retrieve data on the server side and use that data to generate HTML.

```
<body>
    <div class="navbar">
        <a href="/"><i class="fas fa-home icon"></i>Home</a>
        <a href="/items"><i class="fas fa-box-open icon"></i>View Your Items</a>
        <a href="/login"><i class="fas fa-sign-in-alt icon"></i>Login</a>
        <a href="/register"><i class="fas fa-user-plus icon"></i>Register</a>
    </div>
```

Routing to pages

```
app.get('/register', userController.getRegister);
app.post('/register', userController.postRegister);
app.get('/privacy-policy', userController.getPrivacyPolicy);

app.get('/', (req, res) => {
    res.render('home');  //to home.ejs
});

app.get('/login', userController.getLogin);
app.post('/login', userController.postLogin);
app.post('/logout', userController.postLogout);
app.get('/member', userController.getMemberPage);
//Display password reset page
app.get('/reset-password', userController.getResetPassword);
```

Improves user experience by allowing direct access to specific content and functionality via URLs.

When a user accesses a specific URL, the corresponding page or content is displayed.

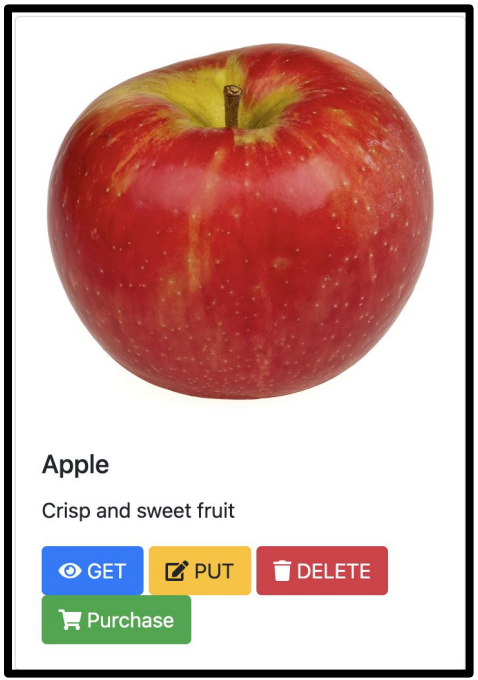
Example: Accessing /login will display the login page, /register will display the registration page, etc.

Used template engine: express.js

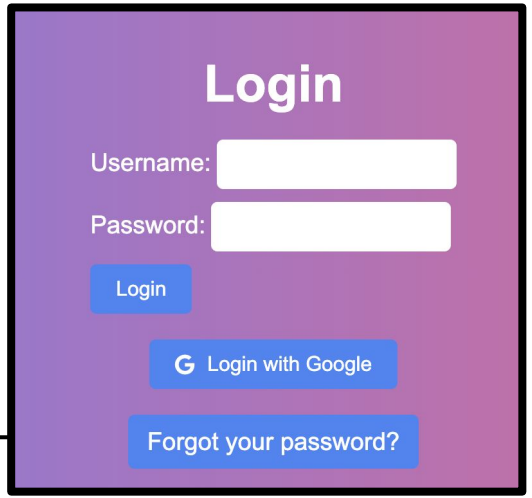
Data from databases and APIs can be dynamically incorporated into HTML, allowing for flexible page generation. It is also easy to reuse code and maintain.

Examples: item.ejs dynamically displays a list of products, login.ejs displays a login form, etc.

Buttons



Examples: item.ejs includes a "GET" button to view product details, a "PUT" button to edit an item, a "DELETE" button to delete an item, etc. login.ejs includes a "Login" button and a "Login with Google" button



Controller

Password Validation

“GetRegister” & “postRegister”: Display the registration form and process registration, respectively.

“getLogin” & “postLogin”: Display login forms and process login.

“PostLogout”: Displays the user logout form and processes the user's logout.

GetMemberPage: Displays the member page after logged in.

Password reset function: It allows a user to request a password reset, validate the reset token, and set a new password.

Relationship between server.js and usercontroller.js

The relationship between server.js and userController is that server.js handles routing (which URL paths trigger which actions) and delegates the actual processing of those actions to methods in userController.

Microservices

Called two external APIs to display the weather and the new in japanese

APIs for news in Japanese

```html
<div class="card" id="news-section">
    <h3 class="section-title">Top News</h3>
    <ul id="news"></ul>
</div>
```

```javascript
async function fetchNews() {
    const response = await fetch('/api/news');
    const data = await response.json();
    const headlines = data.map(article => `<li>${article.title}</li>`).join('');
    document.getElementById('news').innerHTML = headlines;
}
```

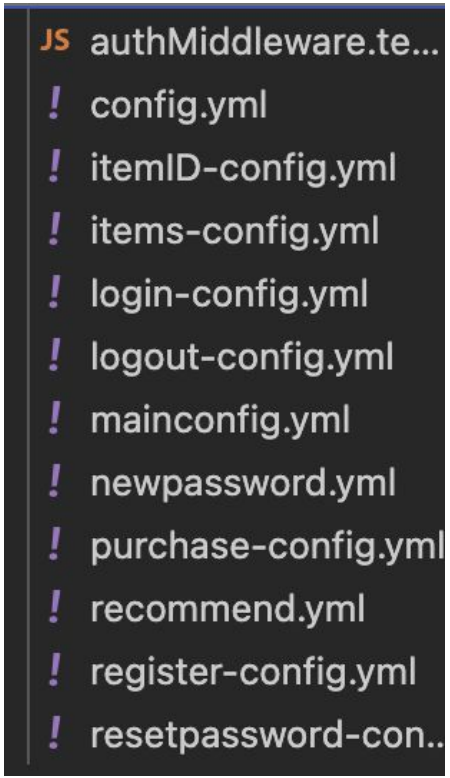
```javascript
get('/api/weather', async (req, res) => {
const apiKey = process.env.WEATHER_API_KEY;
const response = await fetch(`https://api.openweathermap.org/data/2.5/weather?q=Tokyo&appid=${apiKey}`)
const data = await response.json();
res.json({ temp: data.main.temp });
```

API for weather

```html
<div class="card" id="weather-section">
    <h3 class="section-title">Weather</h3>
    <div id="weather"></div>
</div>
```

```javascript
async function fetchWeather() {
    const response = await fetch('/api/weather');
    const data = await response.json();
    document.getElementById('weather').innerText = `Tokyo: ${data.temp}°C`;
}
```

```javascript
app.get('/api/news', async (req, res) => {
    const apiKey = process.env.NEWS_API_KEY;
    const response = await fetch(`https://newsapi.org/v2/top-headlines?country=jp&apiKey=${apiKey}`
    const data = await response.json();
    res.json(data.articles);
});
```



Testing

We tested the middleware using chai and performed load testing using YAML files.

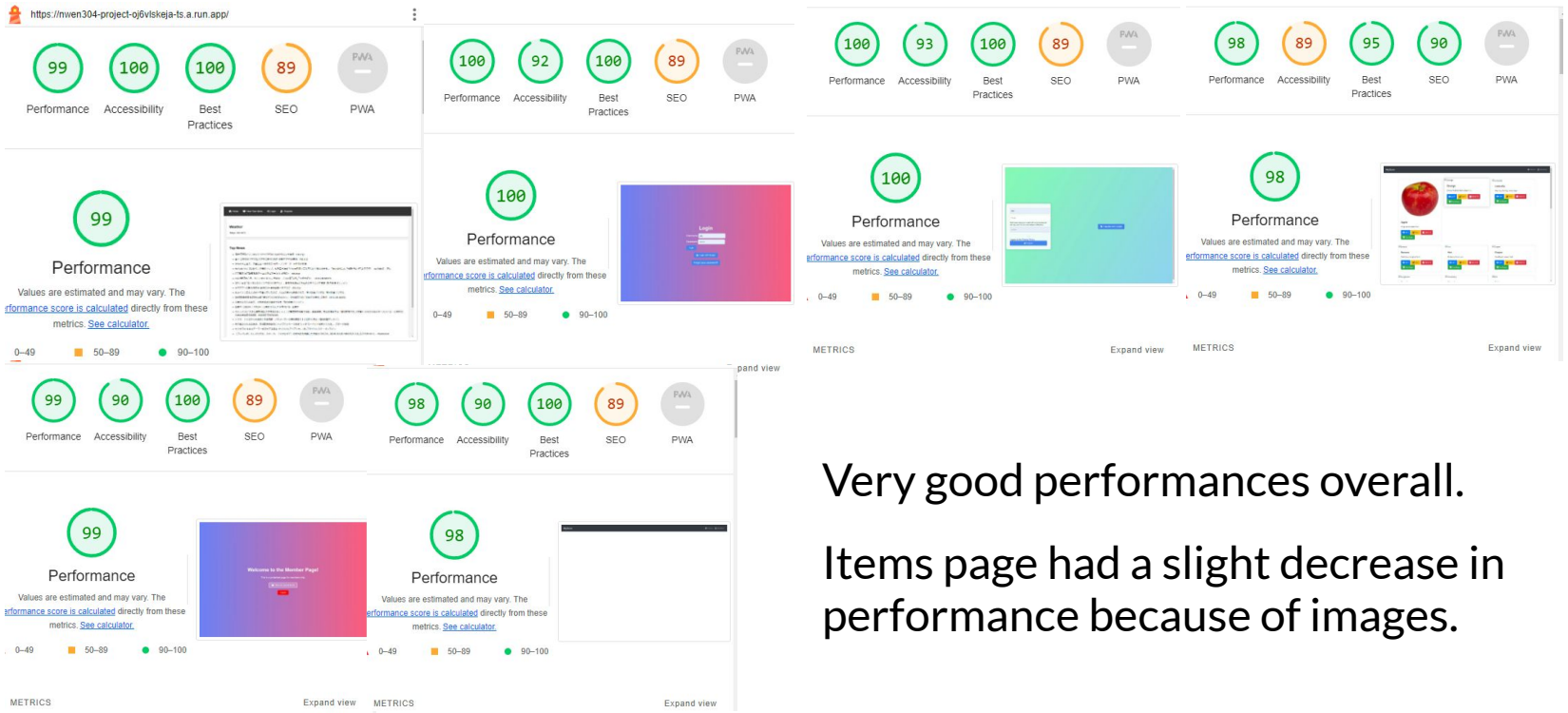
Example file | Command to run `artillery run test/config.yml`

```yaml
config:
  target: 'https://nwen304-project-oj6vlskeja-ts.a.run.app/'
  phases:
    - duration: 60
      arrivalRate: 5

scenarios:
  - flow:
      - get:
          url: '/member'
#member page
```

Chai Testing for Middleware

```
describe('ensureAuthenticated Middleware', () => {
    it('should return 401 if user is not authenticated', (done) => {
        chai.request(app)
            .get('/recommended-items')
            .end((err, res) => {
                expect(res).to.have.status(401);
                done();
            });
    });
});
```

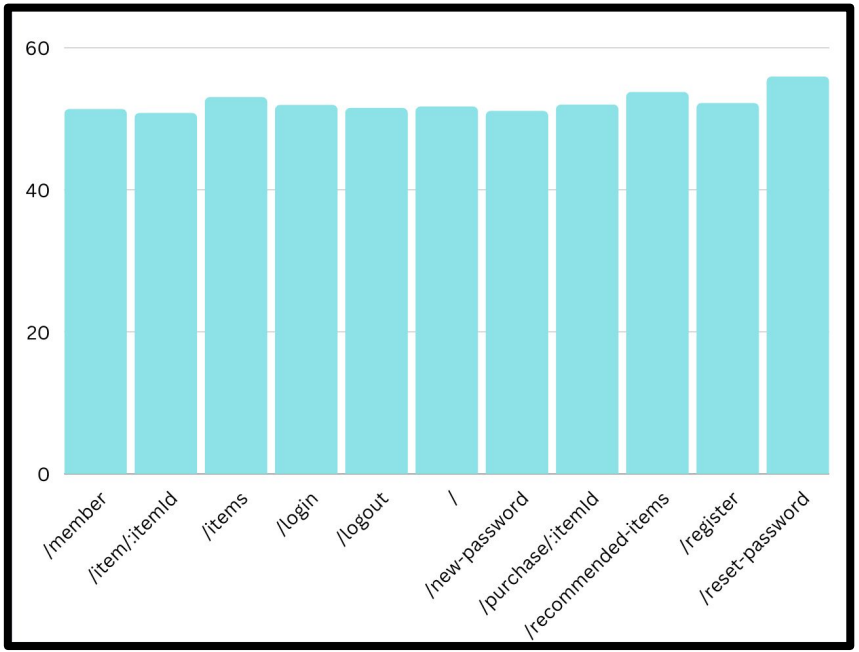
Lighthouse Performances for Rendering



Very good performances overall.

Items page had a slight decrease in performance because of images.

Bottlenecks | Request Response Times



The use of async improved our performance times and our schemas didn't contain much data.

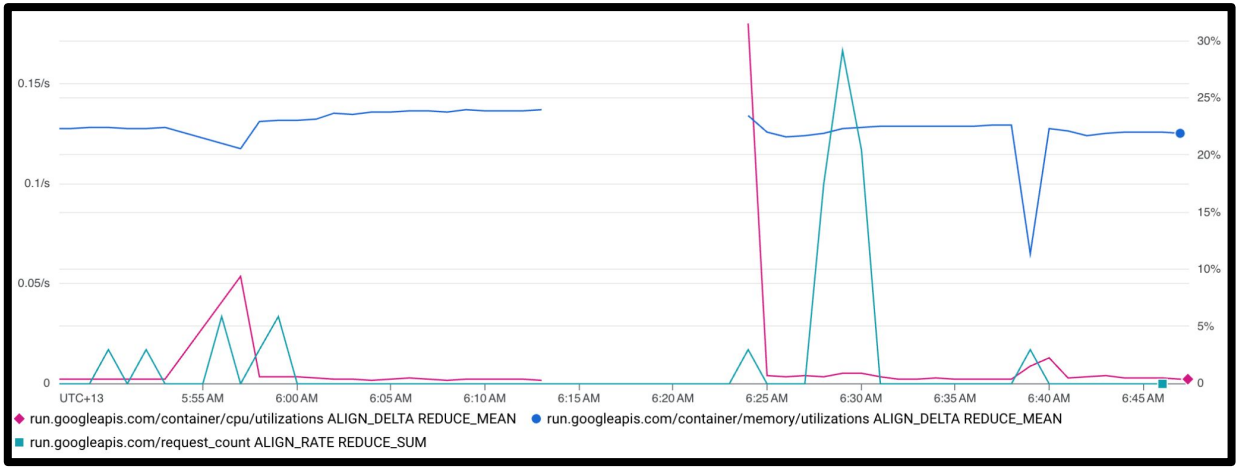
Very balanced throughout all requests.

Bottlenecks | CPU and RAM Utilization

(–) CPU Utilization

(–) RAM Utilization

(–) Mean Request Count



No apparent spikes in CPU and RAM utilization when requests spiked

Middleware

We made “ensureAuthenticated” function that checks if the user is authenticated or not.

“ensureAuthenticated” functionality:

1. Check for req.session and req.session.user:

   It checks if there's a session and a user associated with that session

2. Proceed if Authenticated:

   If req.session.user exists, the middleware calls next(), which means it passes control to the next middleware function or the route handler.

3. Return Unauthorized if Not Authenticated:

   If there's no req.session.user, it means the user is not authenticated. Thus, the middleware responds with a 401 status (Unauthorized) and a message prompting the user to log in.

Relationship with server.js

In server.js, the “ensureAuthenticated” middleware is imported and used as a guard for specific routes. This means that for the routes where ensureAuthenticated is placed, a user must be authenticated to access or manipulate the data.

For example:

```
app.put('/item/:itemId', ensureAuthenticated, (req, res) => {

});
```

This code above from server.js, before the logic inside the PUT route for updating an item (/item/:itemId) is executed, the ensureAuthenticated middleware is run. If the user is authenticated, the route logic will be executed. If not, the user will receive a 401 Unauthorized response.

And also other routes like…

```
app.delete('/item/:itemId', ensureAuthenticated, ...);

app.post('/item', ensureAuthenticated, ...);

app.post('/purchase/:itemId', ensureAuthenticated, ...);
```

use this middleware. This means these operations (adding, deleting, purchasing an item) are protected, and only authenticated users can perform them.

Simple recommendation service

Firstly we made countPurchases function.

This function tallies the number of items purchased for each genre based on the user's purchase history.

Next we made express route: app.get('/recommended-items', ensureAuthenticated, async (req, res) => { ... }):

This route retrieves recommended items for an authenticated user based on their purchase history in database.

```
purchaseHistory:  [String] //for recommendation
```

Overall, based on the user's purchase history, it identifies the genre the user purchases most frequently and recommends random items from that genre.