Chris Bookstore

Edwin Tok Wei Liang
School of Computing and Information System
Singapore Management University
Singapore

Ezekiel Ong Young
School of Computing and Information System
Singapore Management University
Singapore

Jerome Goh Ting Chuan
School of Computing and Information System
Singapore Management University
Singapore

Miguel Alonzo Ortega
School of Computing and Information System
Singapore Management University
Singapore

Tay Wei Jie
School of Computing and Information System
Singapore Management University
Singapore

***Abstract* --- Simple bookstore web application to demonstrate key CI/CD practices for both frontend (AWS Amplify & GitHub) and backend microservices (AWS ECS & GitLab). Further research includes end-to-end testing using docker-compose, JWT authorization managed by AWS Cognito and API Gateway and finally AWS SES for email notifications (*Abstract*)**

Keywords—CI/CD, end-to-end testing, GitLab, Amplify, Cognito

I. Introduction

Books play an important role in everyone's life by introducing them to a world of imagination and providing them with boundless knowledge.

A. Problem

Despite the benefits of reading books, the National Art Council has found that fewer than one in two Singaporeans have read at least one "literary book" in a year [1]. The problem has exacerbated due to the closing of physical bookstores in Singapore as well. This was due to the rise in e-commerce book stores such as Book Depository, where customers can buy books at any time of the day and at a good discount [2].

B. Solution

Our proposed solution is to create an online bookstore for Singaporeans to buy books from the convenience of their home at an affordable price. The solution will be built based on a microservices architecture that consists of atomic microservices and composite microservices. Our group chose this architecture because each microservice can run autonomously which makes it relatively easy to scale. Furthermore, we can reduce the downtime of the microservices by isolating the failure to a single service which prevents cascading failures that would cause the whole architecture to fail. Finally, it supports the mindset of DevOps which allows businesses to rapidly develop and deploy new features. The overall framework is shown in the *Figure 1* below.

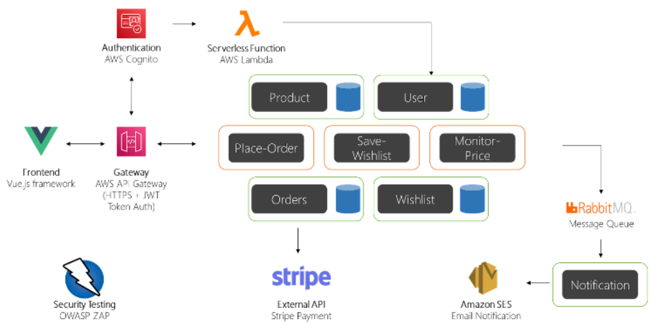


Figure 1: Overall Framework

The overall architecture begins with the frontend, which was built using the Vue.js framework. Any requests made by the frontend will connect to the AWS API Gateway which acts as a reverse proxy to all API calls. Users are required to be authenticated before making any requests on the website and AWS Cognito is responsible for the user registrations and authentications. A JWT token is returned upon a successful login. This can then be used for certain api routes that require authentication such as POST /createorder After the user is authenticated, the user can make API requests from the frontend to the backend microservices.

Atomic Microservices

a) Users Service: User login/logout and registration

b) Products Service: Create new books, read books data, update existing books, delete existing books

c) Orders Service: Create new orders, view existing orders, update existing orders, delete existing orders

d) Notifications Service: Sends Amazon SES email notifications to users when there is a price-drop on their wishlist or when users place an order

e) Wishlist Service: Create new wishlist, view current wishlist and delete existing wishlist

Composite Microservices

a) Place-Order Service: Place orders and process payments through the Stripe API

b) Save-Wishlist Service: Save wishlist into the RabbitMQ message queue

c) Monitor-Price Service: Monitors for price changes based on user's wishlist items

The composite microservices make use of an orchestration pattern to facilitate their communication. The composite services often act as controllers in their specific tasks and send synchronous requests to relevant atomic microservices.

II. Key Scenarios

Our group mainly used the orchestration framework for managing our microservices. Each key scenario has a composite microservice (place-order, save-wishlist & monitor-price) that acts as the main orchestrator for the atomic services and external api.

A. *Ordering a Product*

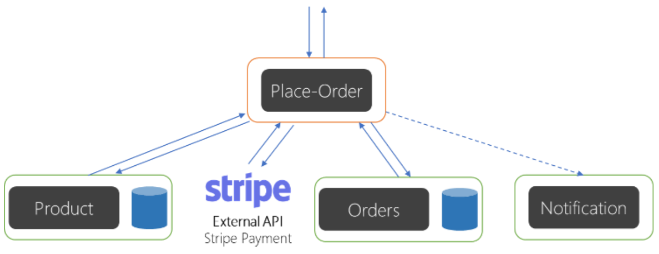
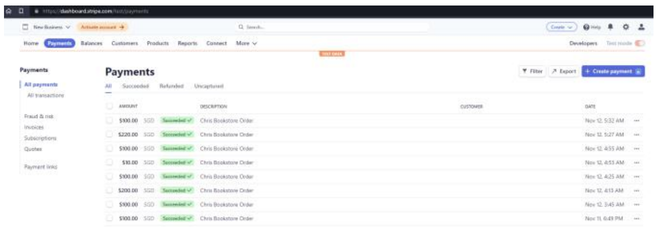


Figure 2: Flow of Ordering a Product

The place-order service acts as a controller to manage the process of ordering a product. It also handles the payment for the product. For every product being ordered, the number of items must be specified in the request. The place-order service verifies that each of the products being ordered is in stock and available. After which, the place-order service verifies the payment using the Stripe API by sending a request with the customer's credit card details. Upon confirmation of the payment, the place-order service then sends a POST request to the order service to create the order. Finally, a notification message is sent out using AMQP, which is then picked up and handled by the notifications service. The notification service handles the sending of the emails to the customers informing them of a successful purchase. In the event of an error in placing the order, whether it is an invalid product, insufficient stock or declined payment, a saga pattern is employed to rollback the placing of the order.



Payments

AMOUNT			DESCRIPTION	CUSTOMER	DATE
$100.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 5:32 AM
$220.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 5:27 AM
$100.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 4:55 AM
$10.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 4:53 AM
$100.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 4:25 AM
$200.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 4:13 AM
$100.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 12, 3:45 AM
$100.00	SGD	Succeeded ✓	Chris Bookstore Order		Nov 11, 6:49 PM

Figure 3: Stripe API recieving payment information

B. *Making a Wishlist*

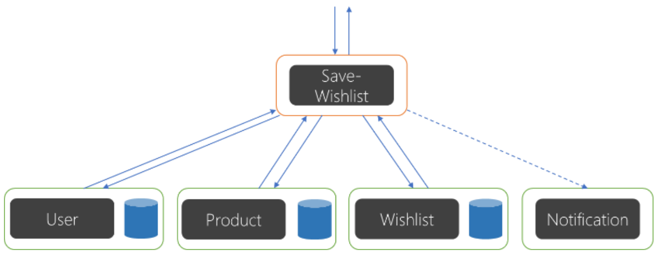


Figure 4: Flow of Making a Wishlist

While the wishlist service provides basic CRUD operations to manipulate wishlist items, the save-wishlist service checks for data integrity by ensuring that the books to be added to the wishlist actually exist. The user invokes the POST method on the save-wishlist service to add books to the wishlist. The format of the body of the method call looks like the following:

```
{
  "user_id": 1,
  "wishlist_books": [
    {
      "book_id": 991
    },
    {
      "book_id": 992
    }
  ]
}
```

Figure 5: Example API Request Body for Creating a Wishlist

Within the save-wishlist service, the user service is called to verify that user_id exists, and for each object in wishlist_books, the product service is called to verify that book_id also exists. After each check, the actual wishlist item is created by calling the wishlist service. As with product ordering, a saga pattern is employed to rollback wishlist item creations whenever an error is encountered within the process. Also, a notification message is sent out for logging purposes using AMQP, which is then picked up and handled by the notifications service.

C. *Notifying Price Drops*

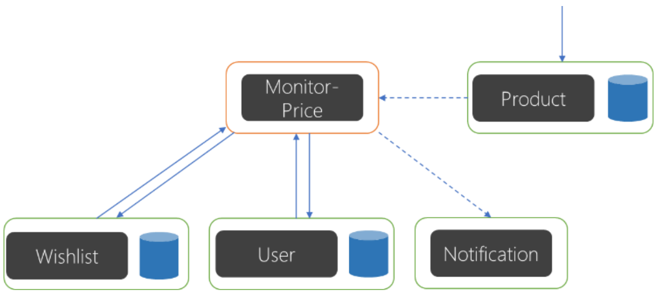


Figure 6: Flow of Price Drop Notification

Whenever the price of a book is caused to drop, for example by invoking the PATCH method on the product service, a message is published using AMQP to a dedicated message queue for price drops. The monitor-price service subscribes to this dedicated queue to process the sending out of notifications to each user who has the particular book in their wishlists. It does this by taking the book ID, which is part of the message payload, and calling the wishlist service to find all wishlist items which have that particular book ID. For each item, it then calls the user service to obtain the email address associated with the user ID, and forms a new message consisting of the email address and the price information. This message is sent out again using AMQP, which is then picked up and handled by the notifications service.

III. DevOps Practices

A. *Utilization of Microservices*

Our group ensured that the microservices were decoupled from the frontend through an API Gateway. This allows our architecture to be more flexible for future developments and changes. Furthermore, we ensured that our atomic services are reusable. An example of this is our notification service where it is reused by the Place-Order, Save-Wishlist and Monitor-

Price service. Different inter-process communication protocols such as HTTP and AMQP were also implemented depending on the nature of each service.

B. *Containerization of Microservices*

To ensure that we can build, test and deploy the pipelines without any complexities, Docker containerisation was implemented to our microservices. With the use of Dockerfiles and docker-compose, it is easy to spin up Docker containers to test and deploy. These containers are also portable in nature which makes it easy to share and deploy. Additionally, the containers provide process isolation which makes it easier to maintain the security of the services.

C. *Agile Development*

In our development process, we attempted to align our development practices in accordance with the manifesto for agile software development. Specifically, we adopted the scrum methodology of development. This included planning our development in incremental and iterative sprints for both our User Interface and microservice development. In addition, we held weekly sprint meetings to update on our progress and plan the next sprint. Agile tools were also utilized to empower our development process. For instance, we used a Kanban board to keep track of the progress of our development process to ensure everyone knew the progress of the development.

D. *Continuous Integraton and Deployment*

To enable our quick and iterative development process, we also created a continuous integration and continuous deployment (CI/CD) pipeline to enable developers to quickly identify problems in our code as well as to allow for quick release. We employed 2 different CI/CD pipeline for the frontend and backend services. The backend CI/CD pipeline utilises Gitlab to build, test and deploy our containerized microservices. We then created Elastic Container Service (ECS) tasks to pull these containerised microservices from our Gitlab container repository. The frontend CI/CD pipeline utilizes AWS Amplify to pull the code base from our Github account where our frontend application is then built and deployed. Overall, this enables us to quickly identify bugs and compatibility issues early to allow for a smoother development process.

IV. DEVOPS TOOLS

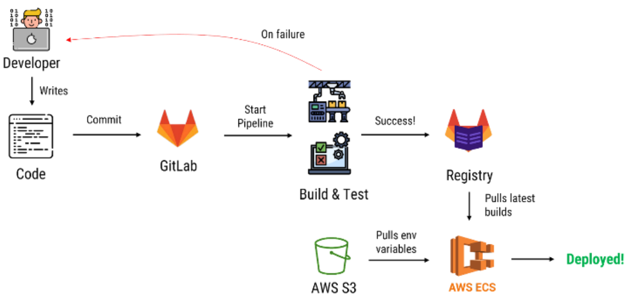
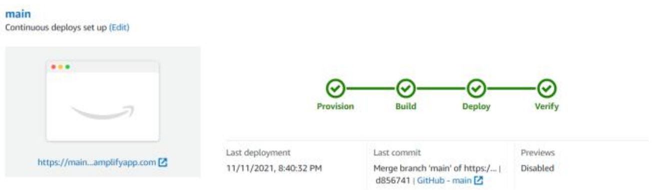


Figure 7: Overview of CI/CD for microservices



main
Continuous deploys set up (Edit)
https://main...amplifyapp.com
Provision
Build
Deploy
Verify

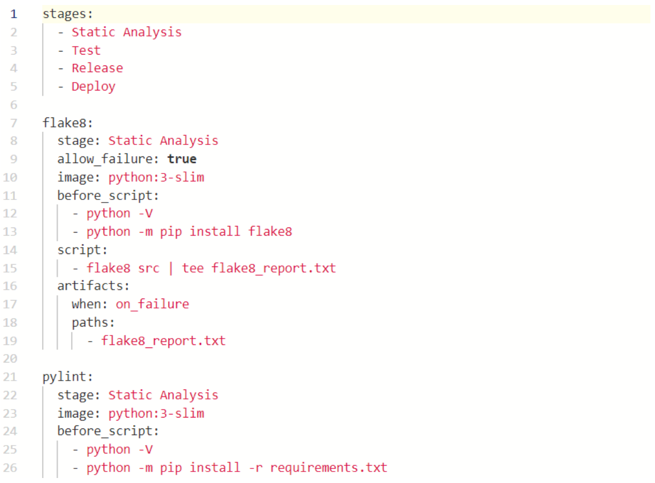
Last deployment
11/11/2021, 8:40:32 PM

Last commit
Merge branch 'main' of https:/... | d856741 | GitHub - main

Previews
Disabled

Figure 8: CI/CD for frontend using AWS Amplify

A. *Microservices with ECS and S3*



```yaml
stages:
  - Static Analysis
  - Test
  - Release
  - Deploy

flake8:
  stage: Static Analysis
  allow_failure: true
  image: python:3-slim
  before_script:
    - python -V
    - python -m pip install flake8
  script:
    - flake8 src | tee flake8_report.txt
  artifacts:
    when: on_failure
    paths:
      - flake8_report.txt

pylint:
  stage: Static Analysis
  image: python:3-slim
  before_script:
    - python -V
    - python -m pip install -r requirements.txt
```

Figure 9: Example GitLab Pipeline for Books Service

Each microservice has a separate repository and ci/cd pipeline managed by Gitlab. Whenever any changes are committed to the repository, Gitlab will build and compile the latest code before testing. The tests include static analysis using pylint and flake8 and integration testing using pytest and Spring Boot tests. When the latest code has successfully passed all the test stages, an image will be built and stored in the GitLab registry. AWS ECS will pull the latest builds from the registry and get the environment variables stored in AWS S3 to deploy the latest versions automatically.

B. *API Gateway with ECS*

To support continuous deployment, we registered each service to a separate target group. This provided the elastic load balancer an interface to connect to the tasks even when they have been taken down and replaced during continuous deployment. Hence, we do not need to redefine the static IP address of the task after deployment as the target group provides a consistent interface. In addition, the target groups conduct periodic health checks on our running tasks to ensure that no instance has unexpectedly crashed. If it detects a failure, it will automatically start up a new instance to replace the failed instance.

The REST API Gateway calls the internal application load balancer via its HTTP address. The load balancer then determines which target group to forward the request to based on the mapping provided in the URL.

For our API gateway, we had 2 different stages – one for production and another for testing. This allowed us to test new mappings which we had just deployed to the ECS without affecting the main set of mappings which are already running on the main stage. It also serves as a simple rollback feature in case any of our deployments went wrong to minimize the impact of the development of our group members. This happened quite often during development and the multiple saved stages allowed us to compare the areas that went wrong and make the necessary adjustments.

C. *Frontend*

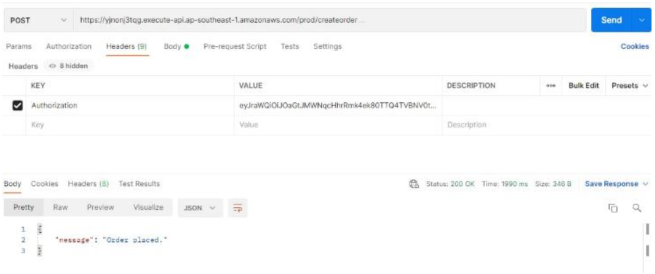
Our front end is maintained with GitHub and CI/CD is done with AWS Amplify. Amplify automatically gets the latest commits from GitHub and rebuilds the new website when changes are committed. They provide a free HTTPS domain with the option of getting a custom domain name in the future either from AWS S3 or other providers such as name.com

D. *Docker Compose*

For local testing, we have also included a docker-compose.yml in the root folder and individual dockerfiles in each of the microservices. This allows the user to easily spin up a local test environment of our entire application with mock databases. External services such as email notifications and the stripe API can still be tested (i.e. the emails will be sent out). This functionality will help to support our end-to-end testing as discussed later.

V. Self-Directed Research

A. *JSON Web Token Authorization with AWS Cognito and REST API Gateway*



https://yjnonj3tqg.execute-api.ap-southeast-1.amazonaws.com/prod/createorder...

KEY	VALUE	DESCRIPTION
✓ Authorization	eyJraWQiOiJOaGtJMWNqcHhrRmk4ek80TTQ4TVBNV0t...	
Key	Value	Description

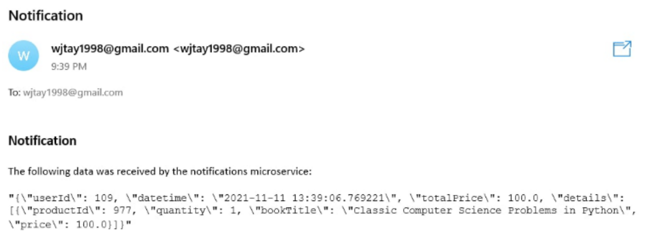
Status: 200 OK Time: 1990 ms Size: 346 B

```
{
    "message": "Order placed."
}
```

Figure 10: Example request requiring JWT Authorization

Our group opted to use AWS Cognito to manage user registration and authorization. Aside from the provision of login and registration pages by AWS, it allows for integration with the AWS API Gateway to provide JWT authorization. When users log in via AWS Cognito, they will be issued a JWT token which is stored in the browser session where subsequent requests can then make use of this token. The API Gateway centralises the management of authorization where it will confirm the validity of the JWT token with Cognito before forwarding the requests to the separate microservices. Hence, our separate microservices do not need to handle authorization.

B. *AWS Simple Email Service (SES)*


Notification

wjtay1998@gmail.com <wjtay1998@gmail.com>
9:39 PM

To: wjtay1998@gmail.com

Notification

The following data was received by the notifications microservice:

```
"{\"userId\": 109, \"datetime\": \"2021-11-11 13:39:06.769221\", \"totalPrice\": 100.0, \"details\":
[{\"productId\": 977, \"quantity\": 1, \"bookTitle\": \"Classic Computer Science Problems in Python\",
\"price\": 100.0}]}"
```

Figure 11: Example email sent by AWS SES

To allow our notification service to notify users of key events, our group made use of AWS SES to send email notifications to the users. AWS SES provides a simple integration with Python where the notification service only has to specify the content and the user email. The email sending process will be fully automated by AWS SES.

C. *Docker End to End Testing*

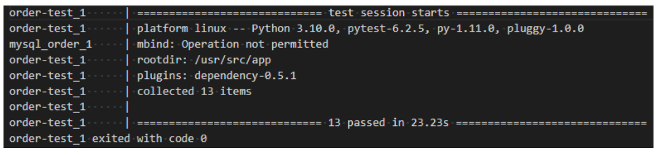

```
order-test_1      | ========================== test session starts ============================
order-test_1      | platform linux -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
mysql_order_1     | mbind: Operation not permitted
order-test_1      | rootdir: /usr/src/app
order-test_1      | plugins: dependency-0.5.1
order-test_1      | collected 13 items
order-test_1      |
order-test_1      | ========================== 13 passed in 23.23s ============================
order-test_1 exited with code 0
```

Figure 12: Logs of end-to-end testing

Aside from static analysis and integration testing for the individual microservices, our group also decided to explore end-to-end testing. We made 3 separate test sets – 1 for each of our key scenarios (placing an order, making a wishlist, price drop notification)

We faced multiple challenges during the setup of the testing environment. Firstly, the configuration of the mock databases and setting up of the schema and tables. We used JPA to automatically create the schema for Java services and manually provided a SQL script via docker compose for the python services. Secondly, the sheer number of containers required to recreate our entire application. This led to issues where certain containers were not starting up in order due to the different build times. The python services were often built much faster than the Java services while the databases and RabbitMQ were often the slowest. As docker-compose does not provide a complete solution for these issues (depends-on seemed buggy and did not ensure that the relevant dependencies were started) we decided to use bash scripts (waitforit.sh) to manage these dependencies. As a further precaution, we introduced a time delay to our in the python script of our python test container to ensure that sufficient time was provided for the test environment to be built before tests were run. Relevant health checks were also done for each test to ensure that the tests were not failing due to incomplete setup of the test environment.

D. *OWASP Zap Security Testing*

As part of our additional research, we wanted to review the security status of our application to review how "safe" our applications are. To do this, we looked into the use of automated vulnerability scanners. While there were many in the market, most of them were commercial product which would require us to pay for the scans. We decided on using OWASP ZAP, the world's most widely used web app scanner provided by OWASP.

1) *UI Security Testing*

Utilizing "Attack Mode" on the deployed web application, we found the following vulnerabilities

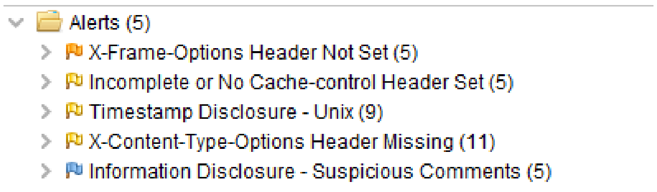

Alerts (5)
- X-Frame-Options Header Not Set (5)
- Incomplete or No Cache-control Header Set (5)
- Timestamp Disclosure - Unix (9)
- X-Content-Type-Options Header Missing (11)
- Information Disclosure - Suspicious Comments (5)

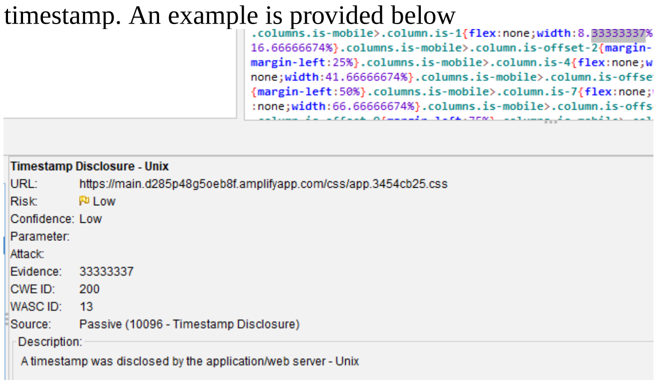
Figure 13: OWASP ZAP UI Testing Report

Upon investigation we found that only the first 2 were true or applicable whereas the later 3 are not true. The risk assessment were classified as low to medium for these vulnerabilities.

The first vulnerability X-Frame-Options Header not set is a vulnerability that could be exploited during clickjacking attacks. This is done by impersonating our site elsewhere by loading our actual site in a full scaled iframe element. To prevent such an attack, we could set to deny all in our x-frame option HTTP header

The second vulnerability was Incomplete or no-cache control header set which implies that browsers and proxies are able to cache our content. In the context of our web application, we could apply some form of cache-control such as restricting caching of sessions using headers such as Cache-Control: no-cache="Set-Cookie, Set-Cookie2".

All other vulnerabilities were found to be untrue. For instance, the timestamp disclosure result raised several numeric values as possible timestamp which are actually not timestamp. An example is provided below

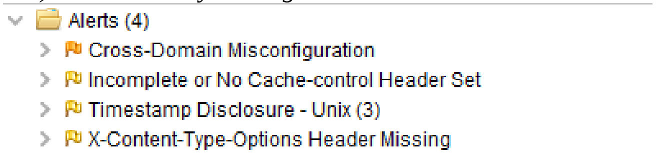


```
.columns.is-mobile>.column.is-1{flex:none;width:8.33333337%
16.66666674%}.columns.is-mobile>.column.is-offset-2{margin-
margin-left:25%}.columns.is-mobile>.column.is-4{flex:none;w
none;width:41.66666674%}.columns.is-mobile>.column.is-offse
{margin-left:50%}.columns.is-mobile>.column.is-7{flex:none;
:none;width:66.66666674%}.columns.is-mobile>.column.is-offs
```

```
Timestamp Disclosure - Unix
URL:        https://main.d285p48g5oeb8f.amplifyapp.com/css/app.3454cb25.css
Risk:       Low
Confidence: Low
Parameter:
Attack:
Evidence:   33333337
CWE ID:     200
WASC ID:    13
Source:     Passive (10096 - Timestamp Disclosure)
Description:
 A timestamp was disclosed by the application/web server - Unix
```

Figure 14: Timestamp Disclosure Vulnerability

2) *API Security Testing*



Alerts (4)
- Cross-Domain Misconfiguration
- Incomplete or No Cache-control Header Set
- Timestamp Disclosure - Unix (3)
- X-Content-Type-Options Header Missing

Figure 15: OWASP ZAP API Testing Report

During our API security testing, we found 4 vulnerabilities amongst which only 2 are true or applicable.

The first vulnerability was due to "Cross-Domain Misconfiguration". This is caused by an overly permissive Cross Origin Resource sharing. While this is a valid concern, we were unable to resolve this without having the web and microservices within the same domain. We could resolve this by having all our front and backend servi*ces within the same domain.*

The second vulnerability is similar to the one we found in our UI testing as described in the previous section.

VI. Reflections

By utilising a microservice architecture, the team was able to further delegate the development of the microservices to its members who were solely in charge of developing an entire service. This enabled significantly higher productivity by avoiding the high communication overhead of having more than one core developer. The use of pre-established endpoints also helped the members abstract away the complexity of having to understand each service entirely when communicating with the service.

However, our group faced some difficulties when it came to making the frontend UI as none of us had any frontend programming experience. We overcame this obstacle by learning frontend and Vue.js tutorials on YouTube from Professor Chris Poskitt and FreeCodeCamp. If we had more time, we would definitely have attempted to make the UI look better and cleaner.

Moreover, the short timeline provided to get our entire solution working which included our frontend and backend services posed a serious challenge for the group. As a result, we had to run both frontend and backend development in parallel which forced us to make some assumption regarding the development of both teams which did not necessarily hold. For example, our development suffered significant setbacks halfway through when we realised that the different microservices did not conform to the same standards of data for their API endpoints. In hindsight, it would have been better to start by discussing at a high level the expected inputs and outputs for the microservices and the frontend to avoid the scenario of having to constantly update the services as the project moved forward. The developers for each service should also create their API documentation upon the creation of services to simplify the workflow process for the creation of the end-to-end tests.

Another challenge is the use of microservices. Our project scale and development component is rather small and suited for a monolithic application architecture. However, due to the requirement we had to split up our services into multiple atomic and composite services. This made our development, testing and CI/CD pipeline much more complex than it needed to be. Nevertheless, the project proved to be an invaluable learning experience for understanding the real-life applications for continuous integration and development.

VII. Future Work

A. *DevSecOps*

Security in CI/CD involves multiple aspects including securing the CI/CD pipeline, allowing for security in the CI/CD pipeline as well as automating security testing. To ensure security of the CI/CD pipeline, we employed best practices in securing our github code base as well as ensuring best practices in our Gitlab and AWS configuration. To ensure security in the CI/CD pipeline, we could leverage static and dynamic analysis tools within our pipeline to check each push for known vulnerabilities. Unfortunately this was not available in the free version of Gitlab. A less than ideal but possible alternative would be to run our OWASP ZAP security test on a regular basis such as every night and having the report reviewed each day.

B. *End-to-end Testing*

End-to-end testing will get increasingly more complicated when more services are added to the application. There would also be issues with versioning of outdated tests in subsequent updates. We could explore methods to break down the application into smaller independent sub-components to reduce the number of containers having to be created at once. For the versioning, API versioning could be implemented directly into the code rather than at the gateway level to allow for different test cases to be written for different versions of the API, so that it will be easier to ensure that test cases are still relevant after the changes to the code.

C. Automatic Generation of Test Cases

Given the increasing number of services, it could be interesting to explore libraries for automatic test generation such as pynguin for Python. This could be very useful for unit testing where we are mainly focused on the methods found within the service and there are fewer dependencies to manage.

References

[1] Sin, Y. (2016, March 17). *Less than half of singaporeans read literary books, National Arts Council Survey finds*. The Straits Times. Retrieved November 15, 2021, from https://www.straitstimes.com/singapore/less-than-half-of-singaporeans-read-literary-books-national-arts-council-survey-finds.

[2] Ping, Y. B., Tan, Y. L., Ng, T., & Jie, P. (2019, September 15). *Who's really killing Singapore's bookstores?* RICE. Retrieved November 15, 2021, from http://www.ricemedia.co/culture-life-whos-really-killing-singapores-bookstores/.

[3] *The zap homepage*. OWASP ZAP. (2021, October 25). Retrieved November 15, 2021, from https://www.zaproxy.org/.

[4] *Session management cheat sheet*¶. Session Management - OWASP Cheat Sheet Series. (n.d.). Retrieved November 15, 2021, from https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#web-content-caching.

[5] *Zap*. X. (n.d.). Retrieved November 15, 2021, from https://www.zaproxy.org/docs/alerts/10020-1/.

[6] *Cross-origin resource sharing (CORS) - http: MDN*. HTTP | MDN. (n.d.). Retrieved November 15, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[7] Matt Boegner. (2019, March 16). *Integrating security into the CI/CD pipeline: Step-by-step recommendations*. Matt Boegner. Retrieved November 15, 2021, from https://mattboegner.com/secure_cicd_pipeline_2/.

[8] *Integrating security into your DevOps lifecycle*. GitLab. (n.d.). Retrieved November 15, 2021, from https://about.gitlab.com/solutions/dev-sec-ops/.

[9] *Audit events*. GitLab. (n.d.). Retrieved November 15, 2021, from https://docs.gitlab.com/ee/administration/audit_events.html.