Week-2 Markdown

Table of Contents

* [Week-2 Technical Tasks Overview](#week-2-technical-tasks-overview)
* [Observability](#observability)
* [Observability Security](#observability-security)
* [Honeycomb](#honeycomb)
* [Add react-js to gitpod.yml](#add-react-js-to-gitpod.yml)
* [AWS X-Ray](#aws-x-ray)
* [How To Tag Work By Week]
* [References](#references)
* [

## Week-2 Technical Tasks Overview

* Instrument our backend flask application to use Open Telemetry (OTEL) with Honeycomb.io as the provider
* Run queries to explore traces within Honeycomb.io
* Instrument AWS X-Ray into backend flask application
* Configure and provision X-Ray daemon within docker-compose and send data back to X-Ray AP
* Observe X-Ray traces within the AWS Console
* Integrate Rollbar for Error Logging
* Trigger an error an observe an error with Rollbar
* Install WatchTower and write a custom logger to send application log data to CloudWatch Log group

## Observability

Observability is relied upon by organizations to help monitor applications and looks at the whole life cycle of an application. It works with traditional monitoring to enable better monitoring to make sure an application functions optimally. Observability includes tools and methods of aggregating a continuous flow of measurable data to analyze how applications, and the networks they run on, perform to troubleshoot and debug any issues that may result in a decline in application performance.

&NewLine;

&NewLine;

&nbsp;

Observability in AWS is the way you would break down an application into multiple processes and looking at the individual functions.

&NewLine;

&NewLine;

&nbsp;

### Importance of Observability:

* Visibility of end-to-end logs
* Decreased alert fatigue for SOC teams
* Troubleshoot and debug quickly with minimal cost
* Application Health
* Minimize operational costs
* Increase in customer satisfaction when applications function properly

### 3 Pillars of Observability

* Metrics
* Traces
* Logs

### AWS Observability Services

* AWS Cloudwatch Logs
* AWS Cloudwatch Metrics
* AWS X Ray Traces

### Instrumentation

To enable observability in AWS you need instrumentation, which is what helps you create or produce logs, metrics, traces. In AWS that could be X-Ray Daemon, Cloudwatch Agent or an AWS distro for open-source OpenTelemetry.

### Logs

* Logging helps determine what might have gone wrong or took place during an incident.
* Monolithic logging is everything in one place and very prevalent in Enterprise environments.
* Current state of logging includes on-premises logs and Cloud logs. Both can include log types such as infrastructure, application(s), anti-virus, and firewall.
* Cloud platforms introduce Infrastructure as Code, Platform as a Service, and Software as a Service which add extra levels of complexity and additional logs.

## Observability Security

AWS observability tools are not security tools, but you can use logs, AWS serverless event driven architecture, or AWS services for threat detection (Amazon GuardDuty) to provide application security.

&NewLine;

&NewLine;

&nbsp;

### Building Security Metrics and Logs for Tracing

* Choose an application to analyze for security issues
* Choose type of application (compute, monolith, microservices)
* Conduct threat modelling sessions
* Identify Attack Vectors
* Map Attack Vectors to TTP in Attack MITRE framework
* Identify instrumentation agents to create tracing (Cloudwatch, Firelens, 3rd party agent)
* Use AWS services like ASW Distro for OpenTelementry for metrics and traces
* View dashboard for practical Attack Vectors only for the application being analyzed. Possible dashboards to use include SIEM, AWS Security Hub with Amazon EventBridge, or open source dashboards.
* Repeat this process for the next application to be analyzed.

### Amazon Detective

[Amazon Detective Documentation]([Amazon Detective Documentation](https://docs.aws.amazon.com/detective/))

* AWS tool to investigate, analyze and identify root cause of potential security issues
* Adds security to observability
* Automatically collects log data from AWS resources
* Assists in conducting faster and more efficient investigations into security issues
* Provides visuals representing changes in type and volume of activity over time
* Can access up to one year of historical event data
* Supported by Amazon GuardDuty, Amazon Inspector, AWS Security Hub, and Amazon Macie.

## Honeycomb

Honeycomb is an observability tool that provides information about code and customer application experience through very quick analysis of data collected about software code performance and behavior. It is a tool that allows for software debugging and distributive tracing to solve problems with distributive services.

### Create Honeycomb Account

* Created a free tier Honeycomb account at <https://ui.honeycomb.io>
* Clicked on `Environment` > `Manage Environments` > `Create Environment`
* Named new data set environment `bootcamp’ and made it purple for Cruddur.
* Clicked `Create Environment’ to complete data set environment set-up.

### Set Honeycomb API Key Environment Variable

The honeycomb API keys for data set bootcamp will

All parts of the Cruddur project should use the same Honeycomb API key. However, the different parts of the project, or services, should have different names.

* Copied the `bootcamp` Honeycomb data set API key.
* In Gitpod, created a text file as a notepad and pasted API key.
* Ran the export command to mark `HONEYCOMB\_API\_KEY` and `HONEYCOMB\_SERVICE\_NAME` as environment variables. Note: service name is associated with the docker.compose.yml and the backend container.

```sh

export HONEYCOMB\_API\_KEY="40M9NgWQivhNArSpvxgYsB"

export HONEYCOMB\_SERVICE\_NAME="Cruddur"

```

* Made HONEYCOMB\_API\_KEY` and `HONEYCOMB\_SERVICE\_NAME` global variables.

```sh

gp env HONEYCOMB\_API\_KEY="40M9NgWQivhNArSpvxgYsB"

gp env HONEYCOMB\_SERVICE\_NAME="Cruddur"

```

* Checked that `HONEYCOMB\_API\_KEY` and `HONEYCOMB\_SERVICE\_NAME` variables are set using the `grep` command..

```sh

Env | grep HONEY

```

![both env vars honeycomb](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/both-env-vars-honeycomb.png)

### Add Open Telemetry (OTEL) Env Vars to backend-flask in docker compose

OTEL, or Open Telemetry, is an open-source/open-standard observability framework for the backend. Adopted by AWS, Azure and Google Cloud platforms. It is a collection of tools, APIs and SDKs used to manage telemetry data such as traces, metrics and logs to help analyze software performance through distributive tracing. Frontend OTEL is not as effective in gathering metrics as for the backend. We’re configuring OTEL to send to Honeycomb. The librairies we’rr installing inside Cruddur app are open-sourec from the Open Telemetry project. The Open Telemetry project is part of the CNCF (Cloud Native Compute Foundation) which also runs Kubernetes, so OTEL is really well-governed open-source code.Honeycom is not in your cloud environment. Your cloud environment is sending standardized messages out to Honeycomb. Honeycomb stores the messages in its database and gives you a user interface to look at them. But, you could also configure OTEl to send messages to open-source Telemetry backend observability platforms other than Honeycomb like X-Ray.

* Add the following OTEL commands to the `docker-compose.yml` file under backend-flask right after BACKEND\_URL.

```sh

OTEL\_SERVICE\_NAME: "backend-flask”

OTEL\_EXPORTER\_OTLP\_ENDPOINT: "https://api.honeycomb.io"

OTEL\_EXPORTER\_OTLP\_HEADERS: "x-honeycomb-team=${HONEYCOMB\_API\_KEY}"

```

![otel env vars](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/otel-env-vars.png)

### Set Up Honeycomb Instrumentation

* Went into Honeycomb account under `bootcamp` environment and clicked on `Home ‘in the left-hand taskbar.
* Clicked on `Python` to get the Honeycomb OTEL python istall package to put into `requirements.txt` located in the `backend-flask` directory.
* Copy and paste following into `requirements.txt`

```sh

opentelemetry-api

opentelemetry-sdk

opentelemetry-exporter-otlp-proto-http

opentelemetry-instrumentation-flask

opentelemetry-instrumentation-requests

```

![python requirements txt](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/python-requirements-txt.png)

* Change directory to backend-flask

```sh

cd backend-flask

```

* Install python opentelemetry-api

```sh

pip install opentelemetry-api

```

Proof opentelemetry-api installed successfully:

![otel python install](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/otel-python-install.png)

* Next, `installed all dependencies` listed in the backend-flask `requirements.txt` file, including ones for sdk, exporter-otlp-proto-http, instrumentation-flask, and instrumentation-requests..

```sh

pip install -r requirements.txt

```

![success requirements dependencies]( https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/success-requirements-dependencies.png)

Refer to <https://ui.honeycomb.io>

* Add app.py updates to the `app.py` file located in backend-flask directory.

```sh

# Honeycomb

from opentelemetry import trace

from opentelemetry.instrumentation.flask import FlaskInstrumentor

from opentelemetry.instrumentation.requests import RequestsInstrumentor

from opentelemetry.exporter.otlp.proto.http.trace\_exporter import OTLPSpanExporter

from opentelemetry.sdk.trace import TracerProvider

from opentelemetry.sdk.trace.export import BatchSpanProcessor

```

![add to app py](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/add-to-app-py.png)

* Initialize tracing and an exporter that can send data to Honeycomb

Add the following code to the `app.py` file below the Honeycomb app.py section.

```sh

#Honeycomb

# Initialize tracing and an exporter that can send data to Honeycomb

provider = TracerProvider()

processor = BatchSpanProcessor(OTLPSpanExporter())

provider.add\_span\_processor(processor)

trace.set\_tracer\_provider(provider)

tracer = trace.get\_tracer(\_\_name\_\_)

```

![initialize tracing](https://github.com/SBecraft/aws-bootcamp-cruddur-2023/blob/main/\_docs/assets/week-2-assets/intialize%20tracing.png)

* Intialize automatic instrumentation with Flask

Add the following code to the `app.py` file below the “initialize tracing….” Code.

```sh

#Honeycomb

# Initialize automatic instrumentation with Flask

FlaskInstrumentor().instrument\_app(app)

RequestsInstrumentor().instrument()

```

NOTE: I excluded the line ` app = Flask(\_\_name\_\_)` listed in the Honeycomb/python instructions because I already had that line of code in my app.py file. If listed twice, ` app = Flask(\_\_name\_\_)` would throw an error.

### Run Distributive Tracing with Honeycomb

* Go back to main directory `aws-bootcamp-cruddur-2023`. Then change directory to `frontend-react-js/`

```sh

cd ..

cd frontend-react-js/

```

* Run an NPM install to install javascript dependencies that will be listed in a file called `package.json`.

```sh

npm i

```

NOTE: There is a line of code in the `Dockerfile” that runs an npm install, so it would seem unnecessary to have to run npm -I again. But, in the aws bootcamp we used the same Dockerfile for development and production, which requires running `npm -i` each time we open the container.

Industry standard is to separate development containers from production containers. The containers for development and production come from different images. Production containers are based on a much more slimmed-down image than that used for development which includes many tools not needed for production.

* Changed into main directory with `cd ..`
* Run container by right-clicking on ‘docker.-compose.yml`and then clicking on `Compose Up`

Docker Containers created:

![docker containers created]( )

![docker containers]( )

### Set Open Ports for Frontend and Backend

Updated `gitpod.yml` file with code to set frontend and backend port numbers to unlock/become public when Containers are run, so that it doesn’t have to be done manually.

* Place the following code in `gitpod.yml` below vscode extension block.

```sh

ports:

- name: frontend

port: 3000

onOpen: open-browser

visibility: public

- name: backend

port: 4567

visibility: public

- name: xray-daemon

port: 2000

visibility: public

```

![ports set]( )

* Had issue with getting to Crudder URL from port 3000. The browser tab shows `Cruddur` but I received a `failed to compile` error`, so I can not see the contents of the webpage.
* Stopped and removed any any and all docker containers that may be open in background competing for open ports with the commands:

```sh

docker stop $(docker ps -a -q)

docker rm $(docker ps -a -q)

```

* gp forward-port any\_port to detect state of ports

A black screen with white text

Description automatically generated

A black screen with blue and white text

Description automatically generated

docker ps

docker container ls -a

docker logs “container id or name”

npm view react version

A screen shot of a computer

Description automatically generated

-SOLUTION TO NODE VERSION

Content for here already in markdown file

![Screenshot updated node]( )

SUCCESSFUL RUN OF DISTRIBUTIVE TRACING:

* Went back to main directory `aws-bootcamp-cruddur-2023` to make another attempt to connect to frontend and backend servers. C
* Changed directory to `frontend-react-js/`

```sh

cd ..

cd frontend-react-js/

```

* Run an NPM install to install javascript dependencies that will be listed in a file called `package.json`.

```sh

npm i

```

* Changed into main directory with `cd ..`
* Run container by right-clicking on ‘docker.-compose.yml`and then clicking on `Compose Up`
* The frontend container(s) is on port 3000
* The backend container(s) is on port 4567
* Make sure both are unlocked.

![Open ports]( )

* I got a 404-ERROR after running the backend container.
* Then, I added /api/activities/home to the end of the URL for the backend server in order to access cruddur data on server on port 4567.

![backend data]( )

* Here is proof I connected to the frontend server on port 3000 to access the Cruddur web application.

![cruddur web app week 2]( )

#### Honecomb Query in Backend-Flask

* Now that I’ve run Docker Compose-Up and can connect to the backend-flask server, there should now be a dataset in the Honecomb bootcamp query results.

![honeycomb data set]( )

![honeycomb data set-2]( )

![honeycomb data set-3]( )

Note: Can check that you have the correct Honeycomb api key with the command

```sh

E | grenv | grep Honey

```

Honeycomb-whoami.glitch.me : how to find out who a Honeycomb api belongs to

![honeycomb whoami]( )

### Create Span Around Hard-Coded Data With a Tracer

* Go to [Honeycomb/OpenTelemetry for Python Documentation}(https://docs.honeycomb.io/getting-data-in/opentelemetry/python-distro/) to get code to create a tracer.

OpenTelemetry requires that a Tracer is given a name as a string. Honeycomb takes the name of a tracer associated with a data trace and turns it into the `library.name` field.

Honecomb.io recommends “pick a name that matches the appropriate scope for your traces. If you have one tracer for each service, then use the service name. If you have multiple tracers that live in different “layers” of your application, then use the name that corresponds to that “layer”.

* Add the following code to the `home\_activities-py` file that is found in the `backend-flask/services/ ` directory.

```sh

from opentelemetry import trace

tracer = trace.get\_tracer("tracer.name.here")

```

![add tracer]( )

* Replaced “tracer.name.here” with `home.activities`
* Add the following code to `home.activities.py` right under `def run():`

```sh

with tracer.start\_as\_current\_span("home-activities-mock-data"):

```

-Run docker `Compose Down` and then `Compose Up` to open new frontend and backend containers.

- Go to the frontend and backend URLS

- Go to Honeycomb.io to see the most recent traces and any spans. More than one color in a trace means that we successfully created a span.

![trace with two colors]( )

![span showing mock data]( )

#### Add Attributes to Spans

* Add the following attribute code to` home.activities.py`

```sh

span = trace.get\_current\_span()

```

```sh

span.set\_attribute(“app.now”, now.isoformat())

```

![span attribute 1]( )

* Add the following attribute code to line above ~return results~

```sh

span.set\_attribute(“app.result\_length”, len(results))

```

* Refresh Cruddur backend and frontend URLs to create new data set in Honeycomb.
* In Honeycomb.io run a new query with visualize as `count` and group by as `trace.trace\_id`. Here are some results:

![app now]( )

## Update Gitpod.yml with react-js

* Make it so we don’t have do a ` npm i` install every time we boot up a container.
* Add the following code to `gitpod.yml`:

```sh

- name: react-js

command: |

cd frontend-react-js

npm i

```

## AWS X-Ray

AWS’s built in distributive tracing and observability tool.

In order for X-Ray to work it needs an X-Ray daemon, another container/application, that runs alongside your application that your application sends data to and the x-ray daemon collects, batches it, and sends it over to the X-Ray API so that you can visualize data in x-Ray.

![xray]( )

### Install AWS SDK

[AWS SDK for Python Documentation]( )

* Add ` aws-xray-sdk` to the `requirements.txt`

```sh

aws-xray-sdk

```

![ aws-xray-sdk]( )

* Install Python dependencies by CLI in the backend-flask directory

```sh

cd backend-flask/

```

```sh

pip install -r requirements.txt

```

* Add code to the `app.py` file to add middleware to flask
* Place the following code into ~app.py~ right below Honeycomb OpenTelemetry code block:

```sh

from aws\_xray\_sdk.core import xray\_recorder

from aws\_xray\_sdk.ext.flask.middleware import XRayMiddleware

```

* Place the following code into `app.py` right below ~#Honeycomb Intiialize tracing and an exporter that can send date to Honeycomb`

```sh

xray\_url = os.getenv("AWS\_XRAY\_URL")

xray\_recorder.configure(service='backend-flask', dynamic\_naming=xray\_url)

XRayMiddleware(app, xray\_recorder)

```

![add xray]( )

* Next, setup AWS X-Ray Resources by adding the file `aws/json/xray.json’ with the following code:

```sh

{

"SamplingRule": {

"RuleName": "Cruddur",

"ResourceARN": "\*",

"Priority": 9000,

"FixedRate": 0.1,

"ReservoirSize": 5,

"ServiceName": "backend-flask",

"ServiceType": "\*",

"Host": "\*",

"HTTPMethod": "\*",

"URLPath": "\*",

"Version": 1

}

}

```

![xray json]( )

* While still in backend-flask directory, create X-Ray group called `Cruddur` with the following code in the AWS CLI:

```sh

aws xray create-group \

```

```sh

--group-name "Cruddur" \

```

```sh

--filter-expression "service(\“backend-flask\")”

```

![create xray group]( )

* View new `backend-flask` group in AWS X-Ray Console:
* Create sampling rule, which determines what information you what to see. Collecting every data point would be too expensive due to the amount of data. Put the following code in a new bash terminal in the main directory.

```sh

aws xray create-sampling-rule --cli-input-json <file://aws/json/xray.json>

```

![sampling rule]( )

* View new `backend-flask` group and sampling rule in AWS X-Ray Console at:

`<https://us-east-1.console.aws.amazon.com/cloudwatch/home?region=us-east-1#xray:settings>`

![xray group in console]( )

* Add X-RAY daemon service to `docker-compose.yml`

```sh

xray-daemon:

image: "amazon/aws-xray-daemon"

environment:

AWS\_ACCESS\_KEY\_ID: "${AWS\_ACCESS\_KEY\_ID}"

AWS\_SECRET\_ACCESS\_KEY: "${AWS\_SECRET\_ACCESS\_KEY}"

AWS\_REGION: "us-east-1"

command:

- "xray -o -b xray-daemon:2000"

ports:

- 2000:2000/udp

```

* Add two environment variables to the backend-flask in the `docker-compose.yml`

```sh

AWS\_XRAY\_URL: "\*4567-${GITPOD\_WORKSPACE\_ID}.${GITPOD\_WORKSPACE\_CLUSTER\_HOST}\*"

AWS\_XRAY\_DAEMON\_ADDRESS: "xray-daemon:2000"

```

![daemon and env vars]( )

#### Determine if X-RAY Daemon sent data to AWS X-RAY

* After running the `amazon/aws-xray-daemon` logs I could see that the X-RAY Daemon successfully sent a batch data to AWS X-Ray

![success xray daemon]( )

* After refreshing the Cruddur backend URL on port 4567 there were successful data traces with the X-Ray Daemon.
* Proof our X-Ray Daemon Service can collect data from the backend and send data traces to AWS X-RAY.

![aws xray query success]( )

## How to Tag Work by Weeks

```sh

git tag week-2

git push --tags

```

## References

[Honeycomb.io] <https://ui.honeycomb.io>

[Honeycomb Documentation for Python] <https://docs.honeycomb.io/getting-data-in/opentelemetry/python-distro/>