



EE547-DISCUSSION 2



TOPICS WE ARE COVERING

- Docker
 - Images and containers (recap)
 - Writing Dockerfiles and building images
- Homework 2 Q1 and Q2

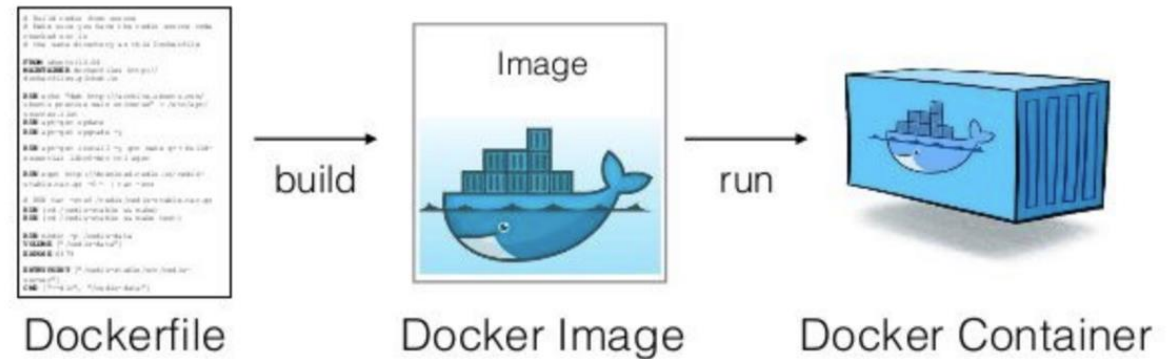


DOCKER



DOCKER IMAGES

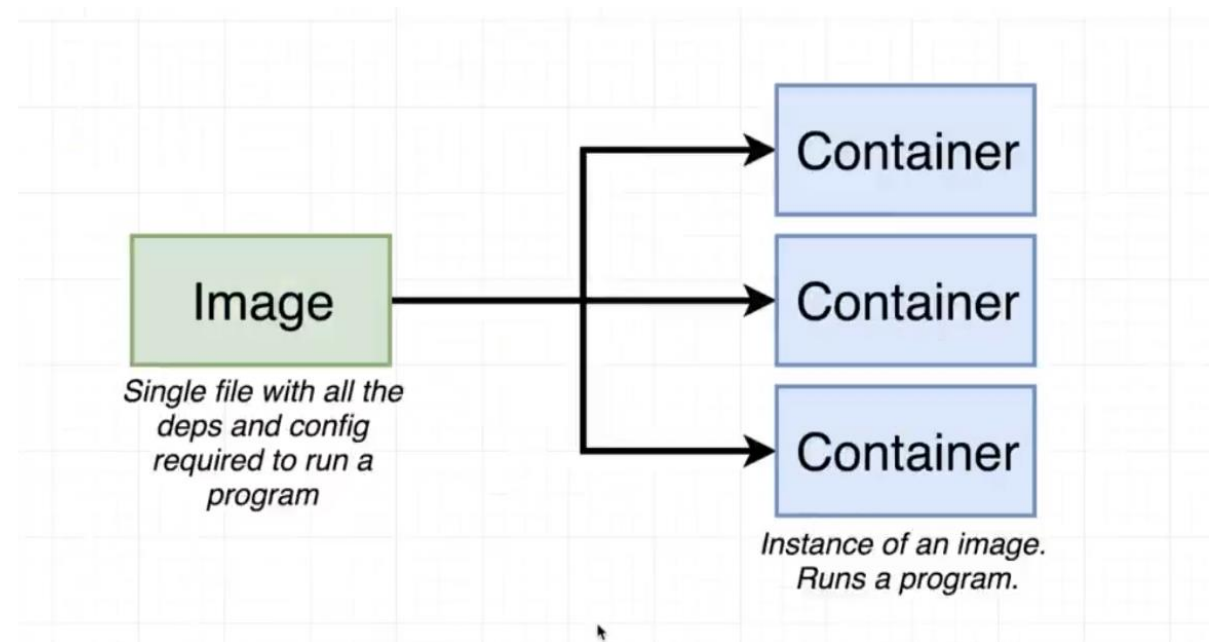
- Read-only template for a container
- Includes:
 - application code
 - dependencies
 - libs & runtimes
 - environment settings



- Example: python:3.11 image contains Python 3.11

CONTAINERS

- Running instance of an image
- For understanding, compare:
 - **Image = class**
 - **Container = object**
- Multiple containers can run from one image



EXAMPLE I – CREATING CONTAINER FROM PULLED IMAGE

- `docker pull busybox`
- `docker run --name hello-busybox busybox echo "Hello"`
- `docker ps -a`
 - (you should see hello-busybox listed as exited)
- `docker rm hello-busybox`

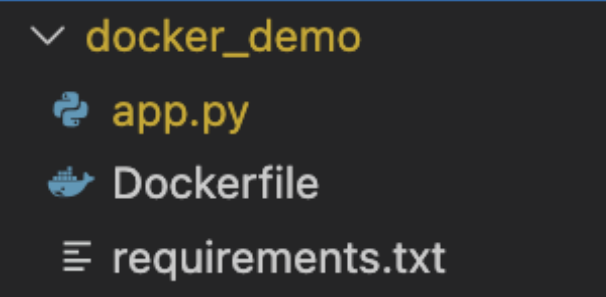


WRITING OUR OWN IMAGE



BUILDING SIMPLE WEBAPP

- We use Flask to build our webapp for simplicity (details see docker_demo folder)



A screenshot of a file explorer window showing the contents of a folder named 'docker_demo'. The folder is expanded, revealing four files: 'app.py' (Python file icon), 'Dockerfile' (Docker icon), and 'requirements.txt' (text file icon). The 'docker_demo' folder name is at the top with a dropdown arrow. The files are listed below it with their respective icons.

```
▼ docker_demo
  🐍 app.py
  🚢 Dockerfile
  ≡ requirements.txt
```

- app.py: Our application (in this case, prints “Hello from EE547!”)
- requirements.txt: In our case, the Python dependencies (Flask)
- Dockerfile: Tells Docker how to create a runnable environment for the application

DOCKERFILE: KEY INSTRUCTIONS

- FROM – base image (Python + minimal OS)
- WORKDIR – like cd inside container
- COPY – copy requirements from our requirements.txt
- RUN pip install – install deps from requirements
- EXPOSE – expose the port the app listens on
- CMD – what to run when the container starts

EXAMPLE 2 – CREATING SIMPLE IMAGE: MESSAGE WEBPAGE

- Give our image a name (flask-webapp):
 - `docker build -t flask-webapp .`
- Verify image existence:
 - `docker images`
- Run Docker with port mapping:
 - `docker run -p 8080:5000 flask-webapp`
 - Format: `-p HOST_PORT:CONTAINER_PORT`
- Visit in browser:
 - `http://localhost:8080`

EXAMPLE 3 – PORT ISSUES

- When we encounter issues, what are some ways to resolve them?
- Try:
 - `docker run -p 5000:5000 flask-webapp`
- This does not work!
- When a port doesn't work:
 1. `docker ps` : Shows all running containers. Is your container up and running?
 2. `docker ps -a` : Shows all existing containers. Did your container exit or crash?
 3. `docker logs <container>` : Shows standard output/error. Troubleshoot your program!
 4. `lsof -i :PORT` : Who owns this port on my machine?
`Get-NetTCPConnection -LocalPort PORT`
 5. Decide: stop process or change port



HOMEWORK 2: QI



PROBLEM OVERVIEW

- **Goal:**
- Build a **command-line Python program**, package it in Docker, and run it **reproducibly**.
- What your program (`fetch_and_process.py`) does:
 - Reads URLs from a file
 - Fetches them over HTTP
 - Collects statistics
 - Writes structured output files
- What Docker does:
 - Ensures the same environment every time
 - Runs your program exactly the same way on our machine

REQUIREMENTS

- For **each URL** in the input file, your program must:
 1. Perform an HTTP GET
 2. Measure response time (milliseconds)
 3. Record HTTP status code
 4. Measure response size (bytes)
 5. Count words *only if the response is text*
 6. Capture timestamp (UTC)
- If one URL fails:
 - Log the error
 - Continue processing the rest

RULES

- **Libraries allowed:**

- Use: urllib.request. **Requests library not allowed!**
- Use only standard library modules: sys, json, time, datetime, os, re, urllib.request

- **Timeout:**

- Maximum 10 seconds per request

- **Content-Type handling:**

- If the header contains "text" → count words
- Otherwise → word count = null

OUTPUT FILES #I

- File I: responses.json
- Note:
 - Per-URL results!
 - On transport failure: error is a string; status_code, content_length, word_count are null

```
[
  {
    "url": "[URL string]",
    "status_code": [integer],
    "response_time_ms": [float],
    "content_length": [integer],
    "word_count": [integer or null],
    "timestamp": "[ISO-8601 UTC]",
    "error": [null or error message string]
  },
  ...
]
```

OUTPUT FILES #2

- File 2: summary.json
- Note:
 - Overall report of all URLs
 - Compute this after all URLs are processed!
- successful_requests counts 2xx only;
average_response_time_ms is over successful requests

```
{  
  "total_urls": [integer],  
  "successful_requests": [integer],  
  "failed_requests": [integer],  
  "average_response_time_ms": [float],  
  "total_bytes_downloaded": [integer],  
  "status_code_distribution": {  
    "200": [count],  
    "404": [count],  
    ...  
  },  
  "processing_start": "[ISO-8601 UTC]",  
  "processing_end": "[ISO-8601 UTC]"  
}
```

OUTPUT FILES #3

- File 3: errors.log
- Note:
 - Remember that this is not a json file, but a readable log!
 - One line per transport failure only (HTTP error statuses are data in responses.json)
- Example:

```
[2026-01-28T19:25:10Z] http://bad.url: DNS resolution failed
```

DOCKERFILE (PROVIDED)

```
FROM python:3.11-slim
WORKDIR /app
COPY fetch_and_process.py /app/
RUN mkdir -p /data/input /data/output
ENTRYPOINT ["python", "/app/fetch_and_process.py"]
CMD ["/data/input/urls.txt", "/data/output"]
```

- Key concepts:
 - **ENTRYPOINT** = fixed program
 - **CMD** = default arguments (overrideable)
- Docker does **not** fetch URLs for you. It only runs Python.

RUNNING THE CONTAINER

```
docker run --rm \
  -v "$(pwd)/test_urls.txt":/data/input/urls.txt:ro \
  -v "$(pwd)/output":/data/output \
  http-fetcher:latest
```

- What this means:
 - Input file is mounted read-only
 - Output directory is writable
 - Container runs, exits, disappears
- Your program must work **without modification**.

EXAMPLE 4 –TAKING INPUT + PRODUCE OUTPUT (HW Q1)

- Building docker image (I am naming this image text-analyzer)

```
docker build -t text-analyzer .
```

- Creating an output directory

```
mkdir -p output
```

- Running a container

```
docker run --rm \  
-v "$PWD/input.txt":/data/input.txt:ro \  
-v "$PWD/output:/out" \  
text-analyzer \  
/data/input.txt /out
```

•\$PWD = your current working directory (host)
•input.txt = file on your machine

•/data/input.txt = path **inside the container**
•:ro = Mount is **read-only**

: Start container, remove container after it runs
: Mount host input.txt as read-only
: Mount host output directory
: Image name, if this doesn't exist, check if image exists!
: Arguments passed to the container



HOMEWORK 2: Q2



PROBLEM OVERVIEW

- **ArXiv Paper Metadata Processor**
- **Goal:**
- Build a **containerized data-processing tool** that:
 - Queries the ArXiv API
 - Extracts paper metadata
 - Analyzes paper abstracts
 - Produces structured JSON outputs

THE ARXIV API

- Endpoint:
 - <http://export.arxiv.org/api/query>
- Example request:
 - `?search_query=cat:cs.LG&start=0&max_results=10`
 - You can use `urlencode()` to ensure correct encoding!
- Return XML, not JSON
- Use Atom feed format

REQUIREMENTS

- From each paper entry, extract:
 - ArXiv ID (from <id>)
 - Title
 - Authors
 - Abstract
 - Categories
 - Published date
 - Updated date
- Allowed parser: `xml.etree.ElementTree`
- Expect missing tags, unexpected nesting, and malformed XML
- **Skip invalid papers, Log warnings, Continue processing**

OUTPUT FILES #I

- File I: papers.json
- Per-paper data! Include:
 - Raw metadata
 - Full abstract
 - Computed abstract statistics
 - Abstract stats include top-20 words and longest/shortest sentence word counts

```
[
  {
    "arxiv_id": "[paper ID]",
    "title": "[paper title]",
    "authors": ["author1", "author2", ...],
    "abstract": "[full abstract text]",
    "categories": ["cat1", "cat2", ...],
    "published": "[ISO-8601 UTC]",
    "updated": "[ISO-8601 UTC]",
    "abstract_stats": {
      "total_words": [integer],
      "unique_words": [integer],
      "total_sentences": [integer],
      "avg_words_per_sentence": [float],
      "avg_word_length": [float]
    }
  },
  ...
]
```

OUTPUT FILES #2

- File 2: corpus_analysis.json
- Note:
 - Overall report of all papers, report after all papers are processed!
 - Include all needed fields

```
{
  "query": "[search query used]",
  "papers_processed": [integer],
  "processing_timestamp": "[ISO-8601 UTC]",
  "corpus_stats": {
    "total_abstracts": [integer],
    "total_words": [integer],
    "unique_words_global": [integer],
    "avg_abstract_length": [float],
    "longest_abstract_words": [integer],
    "shortest_abstract_words": [integer]
  },
  "top_50_words": [
    {"word": "[word1]", "frequency": [count], "documents": [count]},
    ...
  ],
  "technical_terms": {
    "uppercase_terms": ["TERM1", "TERM2", ...],
    "numeric_terms": ["term1", "term2", ...],
    "hyphenated_terms": ["term-1", "term-2", ...]
  },
  "category_distribution": {
    "cs.LG": [count],
    "cs.AI": [count],
    ...
  }
}
```

OUTPUT FILES #3

- File 3: processing.log
- Note: One line per event

[ISO-8601 UTC] Starting ArXiv query: [query]

[ISO-8601 UTC] Fetched [N] results from ArXiv API

[ISO-8601 UTC] Processing paper: [arxiv_id]

[ISO-8601 UTC] Completed processing: [N] papers in [X.XX] seconds

REGARDING XML FILES

```
<book>
  <title>Deep Learning</title>
  <author>Ian Goodfellow</author>
  <year>2016</year>
</book>
```

- **XML (eXtensible Markup Language)** stores structured data using **nested tags**.
- Key ideas:
 - Tags define structure
 - Tags can nest inside each other
 - XML forms a **tree**

PARSING XML IN PYTHON

```
import xml.etree.ElementTree as ET
root = ET.fromstring(xml_text)
```

- Python has a built-in XML parser
- The code above converts XML text → Python objects
 - root represents the **top-level element**
 - You navigate downward using `.find()` and `.findall()`

```
elem.find("tag")
elem.findall("tag")
```

- `find`: Find first matching child (or None)
- `findall`: Find all matching children

Example:

```
title = book.find("title").text
authors = book.findall("author")
```

Real XML files may have missing tags.
Handle them safely! Use:

```
elem = entry.find("summary")
abstract = elem.text if elem is not None else ""
```

Instead of

```
abstract = entry.find("summary").text
```

XML NAMESPACES

- XML Namespaces prevent tag name conflicts
- Arxiv XML starts like this: `<feed xmlns="http://www.w3.org/2005/Atom">`
- Internally, Python sees `{http://www.w3.org/2005/Atom}entry` instead of just "entry"

```
ATOM = "{http://www.w3.org/2005/Atom}"  
entries = root.findall(ATOM + "entry")
```

- You can also use an NS map to do this for a clean code:

```
NS = {  
    "atom": "http://www.w3.org/2005/Atom",  
    "arxiv": "http://arxiv.org/schemas/atom",  
}
```

```
entries = root.findall("atom:entry", NS)  
title = entry.find("atom:title", NS).text
```

EXAMPLE 5: PARSING XML

- I have shown a demo with the above codes in the discussion to demonstrate:
 - Query from Arxiv URL
 - Requesting and simple processing of the XML file
- You can also find a detailed XML guide here:
<https://ee547.usc-ece.com/guides/xml-parsing.html#complete-arxiv-example>