SERVICE ORIENTED ARCHITECTURES

ACIT3855 – WINTER 2024

AGENDA

- Quick Review
- Quiz 3
- Reflection - Concurrent File Writes
- Topics:
  - Logging
  - Configuration
  - Microservices – Code Organization
- Demo – Logging, Configuration and MySQL
- Lab 4
  - Logging, Configuration and MySQL

QUIZ 3

- Quiz is on the Learning Hub
- Open book, but do your own work
- You have <15 minutes to complete it

COURSE SCHEDULE

Week	Topics	Notes
1	• Services Based Architecture Overview • RESTful APIs Review	Lab 1
2	• Microservices Overview • Edge Service	Lab 2, Quiz 1
3	• Database Per Service • Storage Service (SQLite)	Lab 3, Quiz 2
4	• Logging, Debugging and Configuration • Storage Service (MySQL)	Lab 4, Quiz 3
5	• RESTful API Specification (OpenAPI) • Processing Service	Lab 5, Quiz 4
6	• Synchronous vs Asynchronous Communication • Message Broker Setup, Messaging and Event Sourcing	Lab 6, Quiz 5, Assignment 1 Due
7	• Deployment - Containerization of Services *Note: At home lab for Monday Set*	Lab 7, Quiz 6 (Sets A and B)
8	• Midterm Week	Midterm Review Quiz
9	• Dashboard UI and CORS	Lab 8, Quiz 6 (Set C), Quiz 7
10	• Spring Break	No Class
11	• Issues and Technical Debt	Lab 9, Quiz 8
12	• Deployment – Centralized Configuration and Logging	Lab 10, Quiz 9
13	• Deployment – Load Balancing and Scaling *Note: At home lab for Monday Set*	Lab 11, Quiz 10 (Sets A and B)
14	• Final Exam Preview	Quiz 10 (Set C), Assignment 2 Due
15	• Final Exam	

REFLECTION: CONCURRENT FILE WRITES (DEMO)

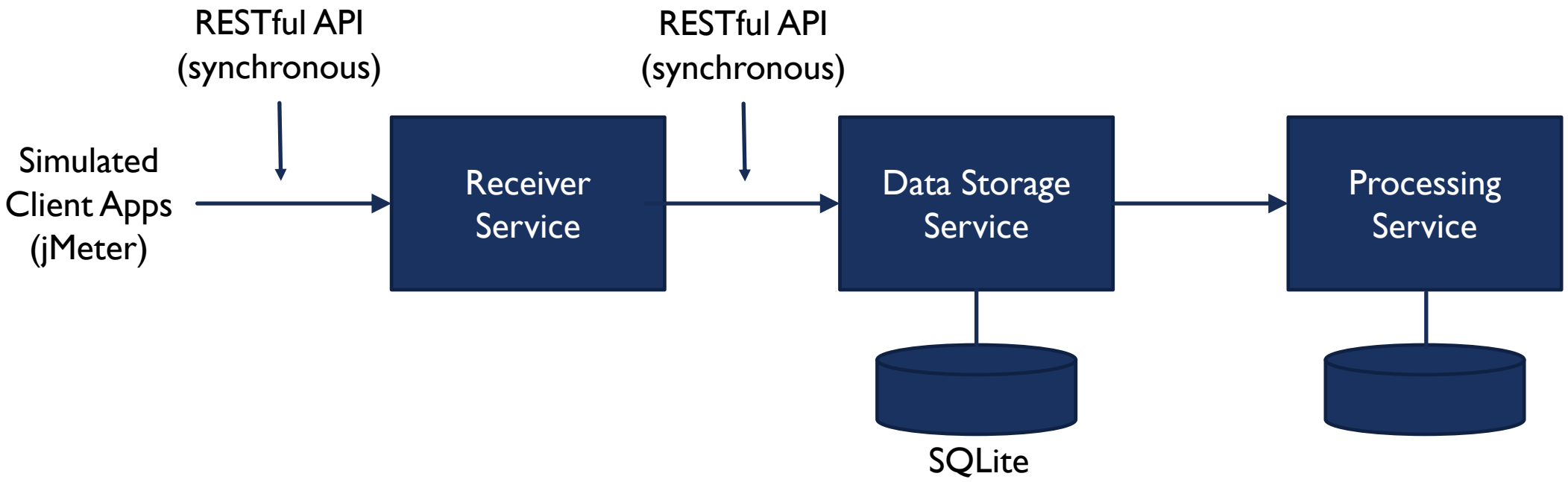
- For Lab 2 (last week's demo), some of us had corruption of our JSON file with concurrent user threads in jMeter
  - Why did this happen?
  - How could we prevent it?

- For Lab 3 (today's demo), our Storage Service will also still start to fail with a larger number of concurrent user threads in jMeter
  - Why might this be happening in SQLite?
  - Why might SQLite not be an appropriate database for our application?
  - Why might MySQL be a better choice?

OUR SAMPLE APPLICATION (SO FAR)

Our sample application will have three initial services:

- Receiver Service (Lab 2)
- Storage Service (Lab 3)
- Processing Service (Lab 5)



CONFIGURATION

External Configuration: Configurable parameters are defined in configuration files rather that in the source code.

These Configuration Parameters may include:

- Environment Specific Settings (i.e., URLs, Database Hostname)
- Options – Turn on/off features or characteristics of the software (i.e., enable/disable 2FA)
- Tuning – Adjust the features or characteristics of the software (i.e., adjust how many retries on failed login)

Reason: Provide a single place where intentionally configurable aspects of the software can be adjusted. You do not want System Admins modifying the code.

CONFIGURATION

Formats Include:

- Python Files
- INI – Traditionally most popular
- JSON – Used more for data transmission and storage, not config
- YAML – Up and coming format
- XML – Used for a while in Enterprise development (i.e., Java)
- And More

CONFIGURATION

- What are some examples of configuration from the applications installed so far in ACIT 4850?

GitLab

gitlab.rb (Ruby File)

- Set our EXTERNAL_URL
- Set our Active Directory settings

Confluence/JIRA

server.xml

- Set our context path
- Set our connector and proxy DNS

Apache2

000-default.conf
default-ssl.conf

- Set our redirect
- Set our Reverse Proxy
- Set our SSL Certificate

CONFIGURATION

Today: One configuration file per service. Each service application has a unique configuration file deployed with the service. System Admin would have to login to the server where the service is installed and modify the file there.

Later: Centralized configuration. Configurations for each service are defined in a central location or repository (i.e., like Git) and services retrieve their configuration from the central repository.

- Allows for more control and visibility over the configurations by centralizing them and (ideally) using a Source Code Management tool
- Possible to dynamically change the configuration of a system

LOGGING

- **Tracing** – Logging for developers. Used to troubleshoot problems during development and sometimes in test or production. Can be very noisy.

- **Event Logging** – Logging for System Admins. Used to monitor and troubleshoot a system in product. Usually has very specific messages. This needs to be designed in during upfront architecture and design.

LOGGING - TRACING

- Most languages have a built-in logging module
- In Python that is called the **logging** module
- It is highly configurable and can write logs to different outputs – files and the console
- Typically the logging configuration is defined in an external configuration file to allow adjustments by the System Admin

LOGGING - TRACING

- Logging modules typically have the concept of logging levels. In Python logging:
  - CRITICAL (highest severity) Corresponds to logging.critical("Critical Message")
  - ERROR Corresponds to logging.error("Error Message")
  - WARNING Corresponds to logging.warning("Warning Message")
  - INFO Corresponds to logging.info("Warning Message")
  - DEBUG (lowest severity) Corresponds to logging.debug("Warning Message")
- It is up to the development team to decide when to use each.
- The logging configuration allows you to specify the lowest logging level output

LOGGING - TRACING

Example rules of usage:

CRITICAL (highest severity)	Crash – i.e., fatal exception
ERROR	Reached an error condition that shouldn't normally happen (i.e., return of non 2XX response code in a RESTful API
WARNING	Reached an unexpected condition that you may expect to happen occasionally
INFO	Informational to show flow through the code. Put at the start and end (or return) of a function/method. It should include the response code if applicable.
DEBUG (lowest severity)	To output data values purely for debugging during development

LOGGING - EVENTS

- Because these are specific events for System Administration purposes, these are normally handled differently:
  - May be output to a different log than the tracing logs
  - May be written to a database table
  - May be published to a messaging system

- Typically there are some well defined rules (or requirements) on what events are logged and the format of the logged data
  - They may include specific text or codes in the log message

LOG MESSAGES FOR DEBUGGING IN MICROSERVICES

Some best practices:

- Add a unique ID to each external API request and carry that through your internal APIs. That way the unique ID and be added to log message to be able to follow the request across the services.
  - This is sometimes called a Trace ID
- Have a consistent logging pattern through each service, even if it is written in a different language.
- Log Aggregation – Send the log messages from your services to a central location (or application) so that:
  - You have a consolidated view of all the log messages to troubleshoot problems (for tracing) or monitoring the system (for events)
  - You don't have to log in to the container of each service to get the logs
  - *We will set this up in the future lab once we've containerized our services*

MICROSERVICES – CODE ORGANIZATION

- Each Microservice is a separate, independent application
  - Can be updated independently of other services
  - Can be deployed independently of other services
- Make sure you don't mix the code between the services
  - Use a separate project or, at minimum, folder within a project, for each service
- It is acceptable to copy over identical files (i.e., openapi.yaml, app_conf.yaml, log_conf.yaml)
- Assume that each service is maintained in a separate Git repository
  - Often each service is in a separate repo, or at least a separate folder within a repo

Storage Service Repo

openapi.yaml
app_conf.yaml (today's lab)
log_conf.yaml (today's lab)
dockerfile (future lab)
appy.py
create_tables.py
drop_tables.py
base.py
event1.py
event2.py

Receiver Service Repo

openapi.yaml
app_conf.yaml (today's lab)
log_conf.yaml (today's lab)
dockerfile (future lab)
app.py

LOGGING FOR TODAY'S LAB

- Parts of the log_conf.yml file
  - loggers, handlers and formatters
- Adding log messages to your code
- Filtering on log levels

TODAY'S TOOLS

RESTful API Specification: SwaggerHub and OpenAPI

- Define a RESTful API in a yaml format

RESTful API Implementation: Python connexion

- Built on top of Flask but allows integration with an OpenAPI specification

Configuration and Logging:

- Yaml for configuration
- Python logging module for tracing

Database: SQLAlchemy and MySQL

- Separate Database Server

RESTful API Testing: PostMan and Apache jMeter

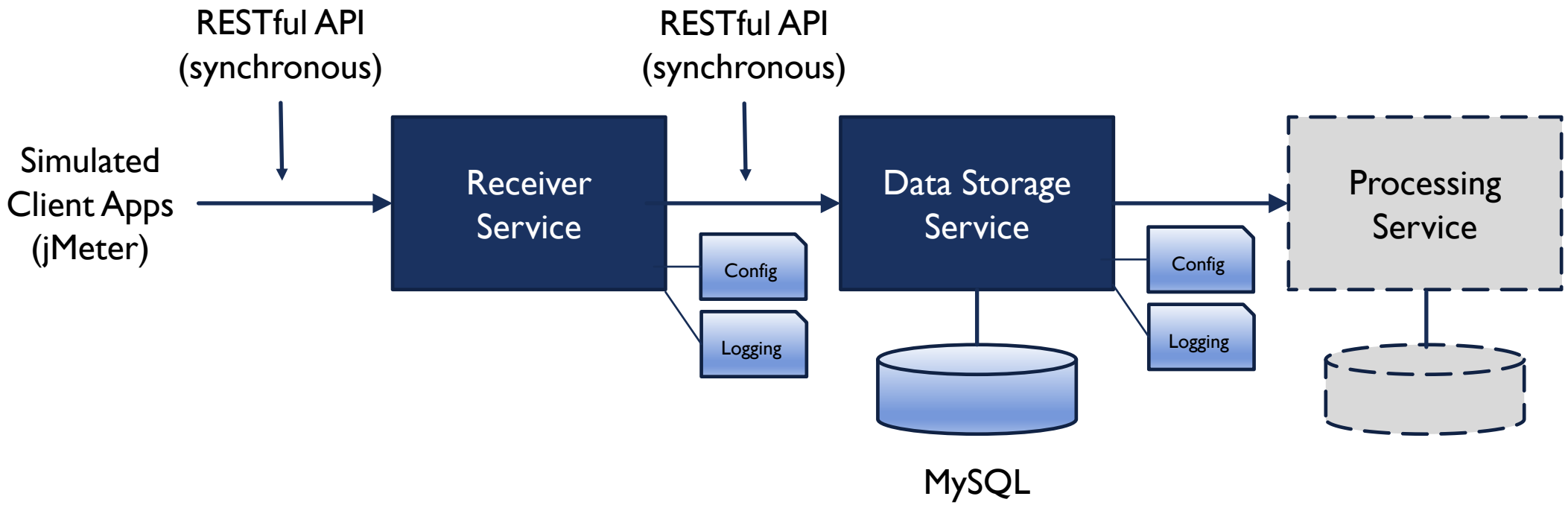
- Postman – same as ACIT 2515
- Apache jMeter – for load testing

You will be using these in your Lab today.

TODAY'S LAB

You will be adding the following to your existing Receiver and Storage Services:

- Configuration
- Logging
- MySQL DB



TODAY'S LAB

The lab is to be submitted individually. Today you will:

- Demo your Lab 3 results
- Add configuration and logging to both of your services
- Switch your SQLite to MySQL for your Storage Service
- Test out your services under load using your jMeter script