Table of Contents

Day 1

Python Dependency Management

Understand Python Package Management

  - Objective: Explain the purpose of package management in Python.
  - Outcome: Students will be able to describe what Python packages are and why package management is important.

Learn to Use `pipenv` for Installing Packages

  - Objective: Demonstrate how to install, upgrade, and uninstall Python packages using `pip`.
  - Outcome: Students will be able to manage Python packages using `pip`.

Understand Virtual Environments

  - Objective: Explain the concept and importance of virtual environments.
  - Outcome: Students will be able to create, activate, and manage virtual environments using pipenv or poetry.

Use Dependency Management Tools

  - Objective: Introduce tools like `pipenv` and `Poetry` for managing dependencies and environments.
  - Outcome: Students will be able to use `pipenv` or `Poetry` to manage project dependencies and environments.

Create and Use `requirements.txt`

  - Objective: Demonstrate how to create and use a `requirements.txt` file for dependency management.
  - Outcome: Students will be able to generate and use `requirements.txt` files to specify project dependencies.

Suggested Tutorials

Tutorial 1: Understanding Python Package Management

1. Introduction to Packages:
  - Packages are collections of modules and other resources bundled together.

2. Why Package Management:
   - Ensures that projects have all the necessary dependencies.
   - Facilitates the installation and management of packages.

Exercise:
- Research and list three popular Python packages and their uses.

Tutorial 2: Using `pipenv` for Installing Packages

1. Installing Packages:

```
pipenv install package_name
```

2. Upgrading Packages:

```
pipenv install --upgrade package_name
```

3. Uninstalling Packages:

```
pipenv uninstall package_name
```

Exercise:
- Install the `requests` package, upgrade it, and then uninstall it.

Tutorial 3: Virtual Environments

1. Creating a Virtual Environment:

```
pip install pipenv
```

2. Activating the Virtual Environment:

```
pipenv shell
```

3. Deactivating the Virtual Environment:

```
exit
```

Exercise:
- Create a virtual environment, activate it, install a package, and then deactivate the environment.

Tutorial 4: Using `pipenv` and `Poetry`

1. Installing pipenv:

```
pip install pipenv
```

2. Creating and Managing a Project with `pipenv`:

```
pipenv install package_name
pipenv shell
```

3. Installing `Poetry`:

```
curl -sSL https://install.python-poetry.org | python3 -
```

4. Creating and Managing a Project with Poetry:

```
poetry new myproject
cd myproject
poetry add package_name
poetry install
```

Exercise:
- Create a new project with pipenv and another with Poetry. Install a package in each.

Tutorial 5: Creating and Using requirements.txt

1. Generating requirements.txt:

```
pip freeze > requirements.txt
```

2. Installing from `requirements.txt`:

```
pip install -r requirements.txt
```

Exercise:
- Generate a `requirements.txt` file for a project, create a new virtual environment, and install the dependencies from the `requirements.txt`.

Questions and Answers

What is the command to install a Python package using `pip`?
pip install package_name

How do you create a virtual environment in Python?
python -m venv myenv

What is the purpose of a requirements.txt file?
It lists all the dependencies of a project, allowing them to be installed with pip install -r requirements.txt.

How can you upgrade an installed package using pip?
pip install --upgrade package_name`

What command is used to activate a virtual environment on macOS/Linux?
pipenv shell

How do you install pipenv?
pip install pipenv

What is the command to add a package to a Poetry-managed project?
poetry add package_name

Additional Resources

[pip.pypa.io] https://pip.pypa.io/en/stable/
[virtualenv.pypa.io] https://virtualenv.pypa.io/en/latest/
[pipenv.pypa.io] https://pipenv.pypa.io/en/latest/
[python-poetry.org] https://python-poetry.org/docs/
[pypi.org] https://pypi.org/

HTTP Request/Response, Header/Body, Methods, TLS, CORS

Understand HTTP Request/Response

Explain the structure and components of an HTTP request and response.
Identify and describe the different parts of an HTTP request and response, including methods, headers, status codes, and body.

Learn HTTP Headers and Body

Understand the role of headers and body in HTTP communication.
Explain the purpose of common HTTP headers and differentiate between the header and body of an HTTP message.

Explore HTTP Methods

Describe the various HTTP methods and their use cases.
Students will be able to explain and use different HTTP methods (GET, POST, PUT, DELETE, etc.) appropriately in web communication.

Understand TLS Handshake

Explain the process and importance of the TLS handshake in secure communication.
Describe the steps involved in the TLS handshake and its role in securing HTTP communication.

Learn about CORS (Cross-Origin Resource Sharing)

- Understand the concept of CORS and how it controls resource sharing between different origins.
- Explain CORS, its purpose, and configure CORS policies in web applications.

Suggested Tutorials

Understanding HTTP Request/Response

HTTP Request Structure:
- Start Line: Method, URL, HTTP version.
- Headers: Key-value pairs with metadata.
- Body: Data sent with the request (optional, used in methods like POST).

HTTP Response Structure:
- Status Line: HTTP version, status code, reason phrase.
- Headers: Key-value pairs with metadata.
- Body: Data sent back from the server (optional, used in successful responses).

- Use a tool like Postman to send different types of HTTP requests and examine the responses.

HTTP Headers and Body

Common HTTP Headers:
- Request Headers: `Host`, `User-Agent`, `Accept`, `Authorization`.
- Response Headers: `Content-Type`, `Content-Length`, `Set-Cookie`.

HTTP Body:
- JSON, XML, HTML, form data.

- Write an HTTP GET request and a POST request with a JSON body using cURL or Postman.

HTTP Methods

- GET: Retrieve data.
- POST: Send data to create a resource.
- PUT: Update a resource.
- DELETE: Remove a resource.

- Implement a simple REST API using Flask or FastAPI and test each method.

TLS Handshake

TLS Handshake Process:

- ClientHello
- ServerHello
- Server Certificate
- Client Key Exchange
- Finished messages

- Use a tool like Wireshark to capture and analyze a TLS handshake.

Understanding CORS

CORS Mechanism:
- Simple Requests
- Preflight Requests
- Response Headers: `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`.

Configuring CORS:
- In a web server (e.g., Apache, Nginx).
- In a backend framework (e.g., Flask-CORS, FastAPI).

- Set up a simple server with CORS enabled and test cross-origin requests using JavaScript in the browser.

Questions and Answers

What are the main components of an HTTP request?
The main components are the start line (method, URL, HTTP version), headers, and body (optional).

What does the status code 200 indicate in an HTTP response?
It indicates a successful request.

Name three common HTTP request headers and their purposes.
Host (specifies the domain name), User-Agent (provides information about the client), Authorization (contains credentials for authentication).

What HTTP method would you use to update an existing resource?
- PUT.

Explain the purpose of the TLS handshake.
- The TLS handshake establishes a secure connection by negotiating encryption parameters and exchanging keys between client and server.

What is the role of the `Access-Control-Allow-Origin` header in CORS?
- It specifies which origins are permitted to access the resource on the server.

Additional Resources

[HTTP] https://developer.mozilla.org/en-US/docs/Web/HTTP
[CORS] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- Postman: Tool for testing and documenting APIs.
- Wireshark: Network protocol analyzer for capturing and analyzing network traffic.
- Flask: Micro web framework for Python.
- FastAPI: Modern, fast web framework for building APIs with Python 3.6+.

Use of Abstract Classes

Understand the Concept and Purpose of Abstract Classes

- Define what an abstract class is and understand its purpose in object-oriented programming.

Learn How to Define and Use Abstract Classes with `abc` Module

- Learn how to define an abstract class and abstract methods using the `abc` module in Python.

Implement Concrete Subclasses from Abstract Classes

- Demonstrate how to implement concrete subclasses that inherit from an abstract class and provide implementations for abstract methods.

Understand Abstract Classes and Multiple Inheritance

- Explore the use of abstract classes in the context of multiple inheritance and how to resolve potential conflicts.

Best Practices and Design Patterns Involving Abstract Classes

- Learn best practices for using abstract classes and understand common design patterns that involve abstract classes.

Practical Applications and Hands-On Exercises

- Apply knowledge of abstract classes through practical exercises and real-world examples.

Suggested Tutorials

- Provide an overview of object-oriented programming concepts and introduce abstract classes. Discuss scenarios where abstract classes are useful.
- Walk through the steps of creating an abstract class and defining abstract methods using `abc.ABC` and `@abstractmethod`. Show how to create concrete subclasses that implement the abstract methods.

 - Provide examples of abstract classes and their concrete subclasses. Highlight the importance of implementing all abstract methods in subclasses.
 - Discuss multiple inheritance in Python and how abstract classes can be used with it. Provide examples to demonstrate how to resolve method resolution order (MRO) issues and metaclass conflicts.
 - Discuss best practices for designing abstract classes, such as keeping them focused and ensuring they provide a clear contract. Introduce design patterns like Template Method and Strategy that leverage abstract classes.
 - Provide a set of hands-on exercises and projects that require the use of abstract classes. Examples might include creating a plugin architecture, a framework for geometric shapes, or a system for managing different types of accounts.
- Create a concrete subclass from a given abstract class and implement its abstract methods.

[abc — Abstract Base Classes] https://docs.python.org/3/library/abc.html
[Abstract Base Classes (abc) in Python] https://realpython.com/python-interface/
[Abstract classes in Python] https://www.geeksforgeeks.org/abstract-classes-in-python/

Questions and Answers

What error is raised if a subclass does not implement an abstract method?
Explain the role of the `super()` function in the context of abstract classes.
What is multiple inheritance, and how does it relate to abstract classes?
Write an example that demonstrates multiple inheritance with abstract classes.
Explain how Python resolves conflicts in method resolution order (MRO).
What are some best practices to follow when designing an abstract class?
Describe the Template Method pattern and provide a code example using abstract classes.
How does the Strategy pattern benefit from using abstract classes?
Create an abstract class for a plugin system and implement two different plugins.
Design a set of abstract classes and concrete subclasses for a geometric shape hierarchy.
Implement a simple banking system with different types of accounts using abstract classes

What is an abstract class?
 - An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can define abstract methods that must be implemented by its subclasses.

Why would you use an abstract class instead of a regular class?
 - Abstract classes are used to define a common interface for a group of subclasses. They ensure that certain methods are implemented in the subclasses, providing a contract that the subclasses must follow.

Write a simple abstract class with one abstract method.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
```

Explain what happens if a subclass does not implement all abstract methods.
  - If a subclass does not implement all abstract methods, Python will raise a `TypeError` when you try to instantiate the subclass.

What is multiple inheritance, and how does it relate to abstract classes?
  - Multiple inheritance is a feature where a class can inherit from more than one base class. When used with abstract classes, it allows a subclass to inherit and implement abstract methods from multiple abstract base classes.

Day 2

Database Normalization, Constraints, and Indexes

Understanding Database Normalization

- Define database normalization and explain its purpose.
- Describe what normalization is and why it is important in database design.
- Describe the different normal forms (1NF, 2NF, 3NF, BCNF).
- Identify and apply the different levels of normalization to database schemas.
- Normalize a database schema from 1NF to 3NF.
- Transform a denormalized database schema into 1NF, 2NF, and 3NF.

Identifying and Applying Database Constraints

- Explain the various types of database constraints (Primary Key, Foreign Key, Unique, Not Null, Check).
- Define and apply primary key, foreign key, unique, not null, and check constraints to database tables.
- Demonstrate how to enforce data integrity using constraints.
- Use constraints to ensure data integrity and consistency within a database.

Understanding Relationships and Referential Integrity

- Explain the concepts of relationships (one-to-one, one-to-many, many-to-many) and referential integrity.
- Describe and implement different types of relationships and enforce referential integrity using foreign keys.

- Designing a Normalized Database Schema
- Design a database schema based on a given set of requirements.
- Create an Entity-Relationship (ER) diagram and translate it into a normalized database schema.

- Using SQL to Create and Manage Database Constraints
- Write SQL statements to create tables with appropriate constraints.
- Write SQL DDL statements to create tables and define primary keys, foreign keys, unique constraints, and check constraints.
- Write SQL statements to modify and delete constraints.
- Write SQL statements to add, modify, and remove constraints from existing tables.

Understanding and Implementing Indexes

- Explain the purpose and types of indexes (single-column, multi-column, unique, full-text).
- Create and use indexes to optimize query performance.

Suggested Tutorials

Understanding Database Normalization

Introduction to Normalization:
- Explanation of normalization and its purpose.
- Benefits of normalization: reduced data redundancy, improved data integrity.

Normal Forms:
- 1NF: Ensure each table has atomic (indivisible) values and a primary key.
- 2NF: Ensure all non-key attributes are fully functional dependent on the primary key.
- 3NF: Ensure no transitive dependencies, where non-key attributes depend on other non-key attributes.
- BCNF: Address anomalies not covered by 3NF by ensuring every determinant is a candidate key.

- Given a denormalized table, identify issues and normalize it to 3NF.

Identifying and Applying Database Constraints

Types of Constraints:
- Primary Key: Unique identifier for table rows.
- Foreign Key: Ensures referential integrity between tables.
- Unique: Ensures all values in a column are unique.
- Not Null: Ensures a column cannot have null values.
- Check: Ensures column values meet a specified condition.

Applying Constraints:
- Examples and SQL statements to apply constraints.

- Create tables with various constraints and practice modifying them.

Understanding Relationships and Referential Integrity

Types of Relationships:
- One-to-One
- One-to-Many
- Many-to-Many

Referential Integrity:
- Using foreign keys to enforce referential integrity.

- Cascading updates and deletes.

- Design and implement relationships between tables, ensuring referential integrity.

Designing a Normalized Database Schema

ER Diagram:
- Creating an ER diagram based on requirements.

Translating to Schema:
- Converting ER diagram to a normalized database schema.

Example Exercise:
- Given a set of requirements, create an ER diagram and a normalized schema.

Using SQL to Create and Manage Constraints

Creating Constraints:
- SQL statements for creating tables with constraints.

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Modifying Constraints:
- Adding, modifying, and deleting constraints.

```
ALTER TABLE employees ADD CONSTRAINT email_check CHECK (email LIKE
'%@%.%');
ALTER TABLE employees DROP CONSTRAINT email_check;
```

Example Exercise:
- Practice creating and managing constraints using SQL.

Understanding and Implementing Indexes

Purpose of Indexes:
- Explanation of how indexes improve query performance.
- Types of indexes: single-column, multi-column, unique, full-text.

Creating Indexes:
- SQL statements for creating different types of indexes.

```
CREATE INDEX idx_employee_name ON employees(name);
CREATE UNIQUE INDEX idx_unique_email ON employees(email);
```

Example Exercise:
- Create indexes on various columns and analyze query performance improvements.

Questions and Answers

What is database normalization and why is it important?
Database normalization is the process of organizing data to reduce redundancy and improve integrity. It ensures efficient storage and consistency of data.

What are the different types of constraints in a database?
Primary Key, Foreign Key, Unique, Not Null, and Check constraints.

How does a foreign key enforce referential integrity?
A foreign key ensures that values in one table match values in another table, maintaining consistent and valid relationships between data.

Write an SQL statement to create a table with a primary key and a foreign key.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

What is the purpose of an index in a database?
An index improves the speed of data retrieval operations by providing quick access to rows in a table based on the indexed columns.

Additional Resources

[Normalization Tutorial] https://www.studytonight.com/dbms/database-normalization.php
[SQL Constraints] https://www.w3schools.com/sql/sql_constraints.asp
[Referential Integrity] https://www.geeksforgeeks.org/referential-integrity-constraint-in-dbms/
[Indexes in SQL] https://www.tutorialspoint.com/sql/sql-indexes.htm

Context Managers

Understand Synchronous Context Managers

Explain the concept and use of regular context managers in Python.
Define and use regular context managers using the `with` statement, and implement custom context managers using the `__enter__` and `__exit__` methods.

Understand Asynchronous Context Managers

Explain the concept and use of asynchronous context managers in Python.
Define and use asynchronous context managers using the `async with` statement, and implement custom asynchronous context managers using the `__aenter__` and `__aexit__` methods.

Compare and Contrast Synchronous and Asynchronous Context Managers

Highlight the differences and appropriate use cases for regular and asynchronous context managers.
Determine when to use regular versus asynchronous context managers based on the requirements of their code.

Suggested Tutorials

Understanding Regular Context Managers

Using Built-in Context Managers:
- Example with file handling:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Creating Custom Context Managers:
- Implementing `__enter__` and `__exit__` methods:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting the context")

with MyContextManager() as manager:
    print("Inside the context")
```

- Create a custom context manager that logs messages when entering and exiting the context.

Understanding Asynchronous Context Managers

1. Using Built-in Asynchronous Context Managers:
   - Example with `aiohttp`:

```
import aiohttp
import asyncio
```

```
async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

asyncio.run(fetch('https://example.com'))
```

Creating Custom Asynchronous Context Managers:
- Implementing `__aenter__` and `__aexit__` methods:

```
class AsyncContextManager:
    async def __aenter__(self):
        print("Entering the async context")
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print("Exiting the async context")

async def main():
    async with AsyncContextManager() as manager:
        print("Inside the async context")

asyncio.run(main())
```

- Create a custom asynchronous context manager that performs asynchronous operations (e.g., simulates network or I/O operations) and logs messages when entering and exiting the context.

Comparing Regular and Asynchronous Context Managers

Use Cases for Regular Context Managers:
- Synchronous resource management: file handling, database connections, etc.

Use Cases for Asynchronous Context Managers:
- Asynchronous resource management: asynchronous I/O operations, network requests, etc.

Example Comparison:
Synchronous context manager:

```
with open('example.txt', 'r') as file:
    content = file.read()
```

Asynchronous context manager:

```
import aiofiles
import asyncio

async def read_file():
    async with aiofiles.open('example.txt', mode='r') as file:
        content = await file.read()
        print(content)

asyncio.run(read_file())
```

- Identify a scenario where an asynchronous context manager is preferable over a regular context manager and implement both versions.

Questions and Answers

What is a context manager in Python?
A context manager is a construct that allows setup and cleanup actions to be performed around a block of code, typically using the `with` statement.

How do you implement a custom regular context manager in Python?
By defining a class with `__enter__` and `__exit__` methods.

What are the `__enter__` and `__exit__` methods used for?
`__enter__` is used to set up the context and `__exit__` is used to clean up the context.

How do you implement a custom asynchronous context manager in Python?
By defining a class with `__aenter__` and `__aexit__` methods.

When should you use an asynchronous context manager instead of a regular one?
When dealing with asynchronous operations, such as asynchronous I/O or network requests, where the operations can be performed concurrently.

Write a simple custom asynchronous context manager.

```
class SimpleAsyncCM:
       async def __aenter__(self):
           print("Async context setup")
           return self

       async def __aexit__(self, exc_type, exc_val, exc_tb):
           print("Async context cleanup")

   async def main():
       async with SimpleAsyncCM():
           print("Inside async context")

   asyncio.run(main())
```

Additional Resources

[The “with” Statement] https://www.python.org/dev/peps/pep-0343/
[Coroutines with async and await syntax] https://www.python.org/dev/peps/pep-0492/
[contextlib — Utilities for with-statement contexts]
https://docs.python.org/3/library/contextlib.html
[aiohttp - Asynchronous HTTP Client/Server for asyncio]
https://docs.aiohttp.org/en/stable/

[Python's "with" Statement: Context Managers and the `with` Statement] https://realpython.com/python-with-statement/
[asyncio - Asynchronous I/O] https://docs.python.org/3/library/asyncio.html

Day 3

Pydantic vs dataclasses

Unlike libraries like dataclasses, Pydantic goes a step further and defines a schema for your dataclass. This schema is used to validate data, but also to generate documentation and even to generate a JSON schema.

Understand the Purpose of Pydantic and Data Classes:

Explain the primary use cases of Pydantic and Python data classes.
Highlight the differences in their applications and capabilities.

Learn Basic Usage of Pydantic

- Create Pydantic models.
- Understand validation and parsing of data with Pydantic.

Learn Basic Usage of Data Classes

- Create Python data classes.
- Understand the limitations of data classes compared to Pydantic models.

Differentiate Features of Pydantic and Data Classes:

- Compare type validation, data parsing, and error handling.
- Highlight the differences in defining default values and field types.

Apply Pydantic in Practical Scenarios:

- Use Pydantic models in FastAPI.
- Validate and parse nested data structures.

Suggested Tutorials

Tutorial 1: Introduction to Pydantic and Data Classes

1. Purpose of Pydantic and Data Classes:
  - Pydantic is used for data validation and settings management using Python type annotations.
  - Data classes are used for creating classes that primarily store data, offering a cleaner syntax for defining class properties.

2. Creating a Basic Pydantic Model:

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str
    signup_ts: Optional[datetime] = None
    friends: List[int] = []
```

3. Creating a Basic Data Class:

```
   from dataclasses import dataclass
   from typing import List, Optional
   from datetime import datetime

   @dataclass
   class User:
       id: int
       name: str
       signup_ts: Optional[datetime] = None
       friends: List[int] = None
```

Tutorial 2: Advanced Pydantic Features

1. Validation and Parsing:
  - Pydantic automatically validates and parses input data:

```
   user = User(id='123', name='John Doe', signup_ts='2023-05-18T14:00',
friends=[1, 2, 3])
```

2. Error Handling in Pydantic:

```
   from pydantic import ValidationError

   try:
       user = User(id='not_an_int', name='John Doe')
   except ValidationError as e:
       print(e.json())
```

3. Nested Data Structures:

```
   class Address(BaseModel):
       street: str
       city: str

   class User(BaseModel):
       id: int
       name: str
       address: Address
```

Tutorial 3: Using Data Classes and Highlighting Limitations

1. Creating Nested Data Classes:

```
   @dataclass
   class Address:
       street: str
       city: str

   @dataclass
   class User:
        id: int
        name: str
        address: Address
```

2. Lack of Validation:
  - Data classes do not provide built-in validation:

```
    user = User(id='123', name='John Doe', address=Address(street='Main
St', city='Springfield'))
```

Tutorial 4: Applying Pydantic in FastAPI

1. Using Pydantic Models in FastAPI:

```
    from fastapi import FastAPI
    from pydantic import BaseModel

    app = FastAPI()

    class Item(BaseModel):
        name: str
        price: float

    @app.post("/items/")
    async def create_item(item: Item):
        return item
```

Questions and Answers

What is the primary purpose of Pydantic?
Pydantic is used for data validation and settings management using Python type annotations.

How does Pydantic handle data validation compared to data classes?
Pydantic automatically validates and parses input data based on type annotations, while data classes do not provide built-in validation.

Write a Pydantic model for a `Product` with fields `name`, `price`, and `tags`.

```
    from pydantic import BaseModel
    from typing import List

    class Product(BaseModel):
        name: str
        price: float
        tags: List[str]
```

How can you handle validation errors in Pydantic?
You can handle validation errors by catching the `ValidationError` exception.

Create a FastAPI endpoint that accepts a `Product` model and returns it.

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class Product(BaseModel):
    name: str
    price: float
    tags: List[str]

@app.post("/products/")
async def create_product(product: Product):
    return product
```

Additional Resources

[Pydantic Docs] https://pydantic-docs.helpmanual.io/
[Data Classes] https://docs.python.org/3/library/dataclasses.html
[FastAPI] https://fastapi.tiangolo.com/

Message Orientated Middleware

Understand the Concept of Message-Oriented Middleware (MOM):

- Define MOM and its purpose in distributed systems.
- Explain the benefits of using MOM, such as asynchronous communication, decoupling, and scalability.

Learn the Key Features of MOM:

- Describe key features of MOM, including message queuing, durability, fault tolerance, and message retention.
- Understand the different communication models: point-to-point and publish/subscribe.

Explore AWS Services for MOM:

- Identify the AWS services that provide MOM capabilities: Amazon SQS, Amazon SNS, and Amazon MQ.
- Explain the use cases and features of each AWS service.

Apply AWS MOM Services to Real-World Scenarios:

- Demonstrate how to set up and use Amazon SQS for task queuing and decoupling microservices.
- Demonstrate how to set up and use Amazon SNS for real-time notifications and event-driven architectures.
- Demonstrate how to set up and use Amazon MQ for migrating existing message broker applications to the cloud.

Suggested Tutorials

Tutorial 1: Introduction to Message-Oriented Middleware

1. What is MOM?
   - Define Message-Oriented Middleware (MOM).
   - Discuss the benefits of using MOM, such as decoupling components, enabling asynchronous communication, and improving scalability.

2. Key Features of MOM:
   - Describe message queuing and its importance.
   - Explain durability and fault tolerance in the context of MOM.
   - Discuss the concept of message retention and its benefits.

Tutorial 2: Exploring Amazon SQS

1. Setting Up Amazon SQS:
   - Create an SQS queue using the AWS Management Console.
   - Send and receive messages using the AWS SDK for Python (boto3).

```
import boto3

# Create SQS client
sqs = boto3.client('sqs')

# Create a queue
response = sqs.create_queue(QueueName='testQueue')
queue_url = response['QueueUrl']

# Send a message to the queue
sqs.send_message(QueueUrl=queue_url, MessageBody='Hello, World!')

# Receive a message from the queue
messages = sqs.receive_message(QueueUrl=queue_url)
print(messages)
```

2. Understanding SQS Features:
   - Explain standard and FIFO queues.
   - Discuss visibility timeout and dead letter queues.
   - Illustrate message retention settings.

Tutorial 3: Exploring Amazon SNS

1. Setting Up Amazon SNS:

- Create an SNS topic using the AWS Management Console.
- Subscribe an email endpoint to the SNS topic.
- Publish a message to the SNS topic using the AWS SDK for Python (boto3).

```
   import boto3

   # Create SNS client
   sns = boto3.client('sns')

   # Create a topic
   response = sns.create_topic(Name='testTopic')
   topic_arn = response['TopicArn']

   # Subscribe an email endpoint to the topic
   sns.subscribe(TopicArn=topic_arn, Protocol='email',
Endpoint='example@example.com')

   # Publish a message to the topic
   sns.publish(TopicArn=topic_arn, Message='Hello, World!')
```

2. Understanding SNS Features:
  - Explain the pub/sub model.
  - Discuss message filtering and fanout scenarios.
  - Illustrate different delivery protocols supported by SNS.

Tutorial 4: Exploring Amazon MQ

1. Setting Up Amazon MQ:
  - Create an Amazon MQ broker using the AWS Management Console.
  - Connect to the broker using a message client (e.g., Apache ActiveMQ).

# Using the ActiveMQ CLI to connect to the broker
```
   ./activemq admin start
   ./activemq consumer --broker-url tcp://<broker-url>:61616 --destination
QUEUE.FOO
   ./activemq producer --broker-url tcp://<broker-url>:61616 --destination
QUEUE.FOO --message "Hello, World!"
```

2. Understanding Amazon MQ Features:
  - Explain the compatibility with industry-standard APIs and protocols.
  - Discuss the managed nature of Amazon MQ and its benefits.
  - Illustrate the durability and scaling features of Amazon MQ.

Questions and Answers

What is the primary purpose of Message-Oriented Middleware (MOM)?
MOM enables asynchronous communication, decouples components, and improves scalability and resilience in distributed systems.

How does Amazon SQS differ from Amazon SNS in terms of messaging model?

Amazon SQS uses a point-to-point model (message queuing), while Amazon SNS uses a publish/subscribe model.

Write a Python script to send and receive a message using Amazon SQS.

```
import boto3

# Create SQS client
sqs = boto3.client('sqs')

# Create a queue
response = sqs.create_queue(QueueName='testQueue')
queue_url = response['QueueUrl']

# Send a message to the queue
sqs.send_message(QueueUrl=queue_url, MessageBody='Hello, World!')

# Receive a message from the queue
messages = sqs.receive_message(QueueUrl=queue_url)
print(messages)
```

What are dead letter queues in Amazon SQS, and why are they useful?
Dead letter queues (DLQs) handle messages that cannot be processed successfully. They help in isolating problematic messages and preventing them from blocking the main processing flow.

Explain the concept of message filtering in Amazon SNS.
Message filtering in Amazon SNS allows subscribers to receive only the messages that match specific criteria, reducing the noise and processing overhead for each subscriber.

Additional Resources

[Amazon SQS] https://aws.amazon.com/sqs/
[Amazon SNS] https://aws.amazon.com/sns/
[Amazon MQ] https://aws.amazon.com/amazon-mq/

Day 4

Test-Driven Development (TDD) with Pytest and FastAPI

Understand the Principles of Test-Driven Development (TDD)

- Grasp the core concepts of TDD and its benefits in software development.
- Provide an overview of TDD, emphasizing its "Red-Green-Refactor" cycle.

Set Up a FastAPI Project for TDD with Pytest

- Learning Goal: Set up a FastAPI project and configure it for TDD using Pytest.
- Instruction: Guide learners through setting up the necessary project structure and dependencies.

Write Initial Failing Tests (Red Phase)

- Write initial tests that fail to ensure the TDD process starts with a clear goal.
- Provide examples of writing failing tests for a FastAPI endpoint.

Write Minimum Code to Pass the Tests (Green Phase)

- Learning Goal: Write the minimum code required to make the failing tests pass.
- Instruction: Implement the functionality required to pass the initial tests.

Refactor the Code (Refactor Phase)

- Refactor the code to improve its structure and readability without changing its functionality.
- Refactor the code and ensure all tests still pass.

Advanced Testing with Pytest and FastAPI

- Learning Goal: Use advanced Pytest features such as fixtures, parameterized tests, and mocking in the context of a FastAPI project.
- Instruction: Provide examples and explanations of advanced testing techniques.

Suggested Tutorials

Introduction to Test-Driven Development

Tutorial: Setting Up the Project

1. Create a Project Directory Structure

```
project_root/
├── app/
│   ├── main.py
│    ├── models.py
│    ├── schemas.py
│    ├── views.py
├── tests/
│    ├── __init__.py
│    ├── test_views.py
├── requirements.txt
└── pytest.ini
```

2. Install Dependencies

```
pipenv install fastapi[all] pydantic pytest
```

3. Create `pytest.ini`

```
# pytest.ini
[pytest]
addopts = --tb=short --strict-markers
testpaths = tests
```

Tutorial: Writing Initial Tests

1. Define Pydantic Model and Endpoint

```
# app/schemas.py
from pydantic import BaseModel

class Item(BaseModel):
  name: str
  price: float
```

2. Create FastAPI Endpoint

```
# app/views.py
from fastapi import APIRouter
from app.schemas import Item

router = APIRouter()

@router.post("/items/")
async def create_item(item: Item):
    return item
```

3. Write Failing Test

```
# tests/test_views.py
import pytest
from httpx import AsyncClient
from app.main import app

@pytest.mark.asyncio
async def test_create_item():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.post("/items/", json={"name": "Test Item",
"price": 10.5})
    assert response.status_code == 200
    assert response.json() == {"name": "Test Item", "price": 10.5}
```

4. Run Tests

```
pytest
```

Tutorial: Implementing Functionality

1. Implement the Endpoint

```
# app/views.py
from fastapi import APIRouter
from app.schemas import Item

router = APIRouter()

@router.post("/items/")
async def create_item(item: Item):
    return item
```

2. Run Tests Again

```
pytest
```

Tutorial: Refactoring Code

1. Refactor Code

- Improve code readability, structure, or performance while ensuring functionality remains the same.

2. Run Tests After Refactoring

```
pytest
```

Tutorial: Advanced Testing Techniques

1. Using Fixtures

```
# tests/conftest.py
import pytest
from httpx import AsyncClient
from app.main import app

@pytest.fixture
async def client():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        yield ac
```

2. Parameterized Tests

```
# tests/test_views.py
import pytest
```

```
@pytest.mark.parametrize("item, expected_status", [
    ({"name": "Item 1", "price": 10.5}, 200),
    ({"name": "", "price": 10.5}, 422),
])
@pytest.mark.asyncio
async def test_create_item(client, item, expected_status):
    response = await client.post("/items/", json=item)
    assert response.status_code == expected_status
```

3. Mocking Dependencies

```
# tests/test_views.py
from unittest.mock import AsyncMock, patch

@patch("app.views.some_dependency", new_callable=AsyncMock)
@pytest.mark.asyncio
async def test_mocked_dependency(mock_dependency, client):
    mock_dependency.return_value = "mocked_value"
    response = await client.post("/items/", json={"name": "Test Item",
"price": 10.5})
    assert response.status_code == 200
    assert response.json() == {"name": "Test Item", "price": 10.5}
    mock_dependency.assert_called_once()
```

[Test-Driven Development with FastAPI and Docker] https://testdriven.io/courses/tdd-fastapi/intro/
[FastAPI Testing Tutorial with Pytest] https://fastapi.tiangolo.com/tutorial/testing/
[FastAPI + Pytest: Testing the application] https://testdriven.io/blog/fastapi-crud/
[Refactoring in Test-Driven Development]
https://martinfowler.com/tags/refactoring.html
[Advanced Pytest Techniques] https://docs.pytest.org/en/stable/
[TDD Full Course] https://www.youtube.com/watch?v=eAPmXQ0dC7Q
[An Introduction to Test-Driven Development (TDD) with Python]
https://realpython.com/test-driven-development-of-a-django-restful-api/

Questions and Answers

What is TDD?
  - TDD is a software development process where you write tests before writing the code that needs to be tested.
  - The process follows three main steps: Red (write a failing test), Green (write the minimum code to pass the test), and Refactor (improve the code while ensuring tests still pass).

Benefits of TDD
  - Ensures code quality and correctness.
  - Encourages better design and architecture.
  - Facilitates refactoring and maintenance.

What are the three main steps in the TDD cycle?
 Red, Green, and Refactor.

How does TDD improve code quality?
 - By ensuring that tests are written before code, it guarantees that code meets the specified requirements and facilitates easier refactoring.

What command do you use to install FastAPI, Pydantic, and Pytest?
```
 - pipenv install fastapi[all] pydantic pytest
```

What is the purpose of the `pytest.ini` file?
It configures Pytest settings, such as test paths and additional options.

Why do you write tests that fail first in TDD?
 - Writing failing tests first ensures you start with a clear goal and verifies that the test is correctly identifying the absence of the desired functionality.

What tool do you use to run the tests in Pytest?
 - You use the command `pytest`.

What is the goal of the Green phase in TDD?
 - The goal is to write the minimum amount of code necessary to make the failing tests pass.

How do you verify that your implementation works?
 - By running the tests and ensuring they pass.

What is the purpose of the Refactor phase in TDD?
 - The Refactor phase aims to improve the code’s structure and readability without changing its external behavior.

How do you ensure that refactoring hasn't broken any functionality?
 - By running the tests to ensure they all still pass after refactoring.

What is a fixture in Pytest?
 - A fixture is a function that provides setup and teardown logic for tests.

How do you use parameterized tests in Pytest?
 - Use the `@pytest.mark.parametrize` decorator to run the same test logic with different sets of inputs and expected outputs.

Why is mocking used in tests?
 - Mocking is used to simulate dependencies and isolate the behavior of the code being tested.

JSON serialization
Header Body Path Parameters Query Parameterrs

@classmethod
Relational vs nonrelational databases (SQL vs NoSQL)
Horizontal vs vertical scaling
Fixed schema
Using sqlite

Day 5

Integration Theory, Dependency Injection, and ACID

Understand the Basic Concepts of Integration Theory

- Define integration theory and its relevance in different fields such as mathematics, computer science, and systems theory.
- Identify the primary goals and principles of integration theory.

Differentiate Between Types of Integration

- Compare and contrast data integration, system integration, and application integration.
- Explain the significance of each type in real-world scenarios.

Explain the Components of Integration Theory

- Describe key components like interfaces, protocols, and middleware.
- Understand the role of standards and frameworks in facilitating integration.

Apply Integration Techniques in Practical Scenarios

- Demonstrate how to integrate two different systems or applications using appropriate tools and techniques.
- Evaluate the effectiveness of different integration approaches in specific contexts.

Suggested Tutorials

[System Integration: Types, Approaches and Implementation Steps] https://www.altexsoft.com/blog/system-integration/
[What is application integration] https://www.ibm.com/topics/application-integration
[What is middleware] https://middlewaredevops.com/what-is-middleware/

Questions and Answers

What is integration theory, and why is it important?
- Integration theory refers to the methodologies and principles used to combine various systems or data sources to work together harmoniously. It is important for ensuring that disparate systems can communicate and operate as a cohesive unit.

What are the differences between data integration and system integration?
- Data integration focuses on combining data from different sources to provide a unified view, while system integration involves connecting different systems to function as a single entity.

Name three tools commonly used for system integration.
- Common tools include middleware like Apache Camel, integration platforms like MuleSoft, and APIs for facilitating communication between systems.

Understand the Concept of Dependency Injection

- Define dependency injection and its purpose in software development.
- Recognize the benefits of using dependency injection in managing dependencies.

Identify Different Types of Dependency Injection

- Describe constructor injection, setter injection, and interface injection.
- Provide examples of scenarios where each type is most applicable.

Implement Dependency Injection in Code

- Demonstrate how to apply dependency injection in a simple project using a programming language like Python or Java.
- Utilize a dependency injection framework (e.g., Spring for Java, FastAPI for Python).

Evaluate the Impact of Dependency Injection

- Analyze how dependency injection improves code maintainability and testability.
- Identify potential pitfalls and best practices for using dependency injection effectively.

Suggested Tutorials

- Tutorial 1: Introduction to Dependency Injection

- Tutorial 2: Types of Dependency Injection

- Tutorial 3: Implementing Dependency Injection

Questions and Answers

What problem does dependency injection solve in software development?
- Dependency injection helps manage and inject dependencies, making the code more modular, easier to maintain, and testable.

Explain the difference between constructor injection and setter injection.
- Constructor injection provides dependencies through a class constructor, while setter injection uses setter methods to provide dependencies after object creation.

How would you implement dependency injection in a Python project using FastAPI?
- In FastAPI, you can use the `Depends` feature to inject dependencies into your routes or services.

Understand the ACID Properties

- Define the ACID properties (Atomicity, Consistency, Isolation, Durability).
- Explain the significance of each property in database transactions.

Identify the Importance of ACID in Databases

- Discuss the role of ACID properties in maintaining database integrity and reliability.
- Compare ACID compliance in different database management systems.

Implement ACID-Compliant Transactions

- Demonstrate how to write ACID-compliant transactions in SQL.
- Implement transaction management in a programming language like Python or Java.

Evaluate Scenarios for ACID Compliance

- Analyze scenarios where ACID properties are critical for application success.
- Discuss the trade-offs between ACID compliance and performance/scalability.

Suggested Tutorials

- Tutorial 1: Introduction to ACID Properties

- Tutorial 2: Importance of ACID Properties

- Tutorial 3: Implementing ACID-Compliant Transactions

Questions and Answers

What does atomicity ensure in a database transaction?
- Atomicity ensures that a transaction is all-or-nothing, meaning that either all operations within the transaction are completed, or none are.

Why is durability important in a database transaction?
- Durability ensures that once a transaction is committed, it will remain so, even in the event of a system failure.

How can you implement a transaction in SQL to ensure ACID compliance?
- By using SQL commands like `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`, you can manage transactions to ensure they meet ACID properties.