UNIVERSITY OF MESSINA

DEPARTMENT OF ENGINEERING

Master's Degree in
Engineering and Computer Science

Computer System Security

Project:

Using the sharding technique for PDF file security

Students:
Longo Andrea
Musmeci Edoardo

ACADEMIC YEAR 2024-2025

Contents

Introduction

The sharding technique consists of dividing a file into smaller parts, called shards. This method is often adopted to improve performance and scalability, distributing data across multiple servers or storage systems. Applied to PDF files, this technique allows you to divide a document into smaller fragments, each containing a portion of the original content. For example, a 30-page PDF can be divided into three fragments of 10 pages each. This division is useful for managing large documents, improving accessibility and optimizing file manipulation and distribution. In this case, we use sharding to divide PDF files of various sizes into three distinct fragments. Once divided, the fragments are encrypted using the AES (Advanced Encryption Standard) algorithm, thus ensuring a high level of protection for each individual fragment. The system then sends the encrypted fragments to three different Docker containers. This process is managed by a Flask server that, via HTTP POST requests, distributes the fragments in isolation, with one fragment for each container. Once the distribution phase is completed, the system allows the original PDF file to be recomposed. To do so, the fragments are retrieved from their respective containers and subjected to a decryption process. Subsequently, the decrypted fragments are merged to recreate the original document, which is finally saved in a local output folder. This approach ensures data security and integrity during all phases, from file splitting to file reconstruction. The main goal of this project is to demonstrate how the combination of the sharding technique and AES encryption can improve security in file management. Distribution on separate containers adds an additional layer of protection, while sharding facilitates the management of large files in distributed environments. This system represents a concrete and innovative application of sharding in the context of cybersecurity, offering an effective solution for the protection and secure management of documents.

Chapter 1

Sharding

Sharding is a technique used to improve the scalability and performance of data storage and management systems. Originally introduced in the context of databases, the concept of sharding is now applicable to different types of files and systems, including the sharding of documents such as PDFs. The word "shard" means "fragment" or "sliver," reflecting the idea of dividing a larger entity into smaller parts. In addition, this technique is used to manage large volumes of data by breaking it into smaller, independent parts. This division allows for better distribution and load balancing across multiple systems, improving scalability and overall performance. The fundamental concepts of sharding include logical data division, data distribution, and shard management.

1.0.1 Logical Data Division

The crux of sharding is the logical division of data. This process involves segmenting the original dataset into smaller parts, each of which is called a shard. Each shard is a complete, independent subset of the original dataset, containing a specific portion of the data based on a defined criterion. For example, in a user database, you might split the dataset based on the user's ID, creating shards that contain specific ranges of IDs. This logical division allows for:

- isolate data: each shard contains a subset of data that can be managed independently of other shards. This isolation helps limit the impact of shard-specific issues without impacting the entire system.
- easier access and management: data divided into smaller shards is easier to manage and access than a large dataset or PDF, in fact,

data operations, such as queries and updates, can be performed more quickly on smaller shards.

1.0.2 Data Distribution

Once split, the fragments (shards) are distributed across different nodes within a system. In the context of a database, these nodes can be physical or virtual servers. In the case of large files, such as PDFs, shards can be distributed across different disks or across sections of the same disk. Distributing data has several advantages:

- load balancing: by distributing shards across multiple nodes, a single node is prevented from becoming a bottleneck. This allows for more balanced workload management and improves overall system performance.
- redundancy and resilience: distributing shards across multiple nodes can increase system resilience. If one node fails, other nodes can continue to function, ensuring continuous data availability.

1.0.3 Shard Management

Shard management involves coordinating and accessing distributed data in such a way that the system presents itself as a single, coherent entity to end users. To achieve this, it is necessary to implement addressing and lookup mechanisms that allow quickly identifying the shard that contains the requested data. One of the key aspects is addressing and lookup: each data request must be routed to the correct shard, requiring an efficient lookup system that can quickly determine where the specific data resides. Furthermore, maintaining data consistency across shards is essential, especially in distributed contexts. Operations involving multiple shards must be synchronized to ensure data integrity and consistency across the system. Finally, another crucial element is the continuous monitoring of the performance and status of each shard. This includes not only the detection and management of any errors or anomalies, but also periodic maintenance activities necessary to optimize performance and ensure the overall reliability of the system.

1.0.4 Benefits of sharding

Sharding offers important benefits, including horizontal scalability, which allows you to add new hardware resources, such as nodes or servers, to handle an increase in workload. With this approach, each new node can host additional shards, distributing the overall load and improving system performance. Additionally, by dividing a large dataset into smaller shards, data operations become faster, as each shard is more manageable and allows for optimized queries and processing. An additional benefit of sharding is improved system resilience and reliability. If a node containing one or more shards goes down, the other shards can continue to operate, thus ensuring greater business continuity and reducing the risk of total system failure.

1.0.5 Splitting and recomposing shards

In this section, the class called Sharding is defined, which implements two main methods to manage file splitting and recomposition. The split method is responsible for splitting a binary file, such as a PDF, into smaller fragments (shards) and then distributing them across the three containers. Before starting, the code checks that the file exists in the ./input/ directory and, if it does not find it, generates an error message and stops the process. Then, it reads the file content in binary mode, calculates the base size of the fragments based on the number of containers specified and splits them into portions of bytes. The last fragment includes any remaining bytes to ensure that all the content of the file is split correctly. Once split, it returns a list containing the fragments or reports errors, for example if the file is too small compared to the number of fragments requested. The recompose_from_container method reconstructs the original file from the decrypted fragments. It checks that the fragment list is not empty and creates, if necessary, ./output directory to save the recomposed file. It then opens a new output file in binary writing mode and writes the contents of each fragment sequentially to it. Upon completion, it prints a success message with the full path to the recomposed file or reports any errors encountered during the process.

```python
import os
from database import *


class Sharding:
    def split(filename, num_container):

        try:

            if not os.path.exists("./input/" + filename):
                print(f"Errorr: File '{filename}' not found in './input/' directory.")
                return None


            with open("./input/" + filename, 'rb') as f:
                content = f.read()


            total_size = len(content)
            base_chunk_size = total_size // num_container

            if base_chunk_size == 0:
                print("Error: The file size is too small compared to the number of servers.")
                return None

            shard_list = []
            offset = 0
            for i in range(num_container):

                if i == num_container - 1:
                    chunk = content[offset:]
                else:
                    chunk = content[offset:offset + base_chunk_size]
                    offset += base_chunk_size

                shard_list.append(chunk)

            print(f"File '{filename}' divided into {num_container} fragments.")
            return shard_list

        except Exception as e:
            print(f"Error during file splitting: {e}")
            return None


    def recompose_from_container(document_id, decrypted_data_list, output_filename):


        try:

            if not decrypted_data_list:
                print(">> Error: Empty fragment list!")
                return

```

```
55
56              output_dir = "./output"
57              os.makedirs(output_dir, exist_ok=True)
58
59
60              output_path = os.path.join(output_dir, output_filename)
61
62
63              with open(output_path, 'wb') as output_file:
64                  for shard in decrypted_data_list:
65                      output_file.write(shard)
66
67              print(f">> File '{output_filename}' recomposed successfully
    in {output_path}.")
68              return output_path
69
70          except Exception as e:
71              print(f">> Error during recomposition: {e}")
```

Chapter 2

AES Encryption

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm recognized as one of the most secure and widely used encryption standards worldwide. Developed by the National Institute of Standards and Technology (NIST) of the United States, AES was adopted as a standard in 2001, replacing the previous Data Encryption Standard (DES). Key features of AES :

- block cipher: AES is a block cipher, which means that data is encrypted in fixed-length blocks, typically 128 bits. Each block of data is processed independently.
- key length: AES supports three different key lengths: 128 bits (10 rounds), 192 bits (12 rounds), and 256 bits (14 rounds). The security of the algorithm increases with the key length, making AES extremely flexible in terms of balancing security and performance.

AES uses a substitution-permutation (SPN) structure and operates on a 4x4 matrix of bytes, called a state. The data transformation occurs through a series of substitution and permutation operations applied during each encryption round. The main operations of each round are:

- subBytes: each byte of the state is replaced with a corresponding byte from a predetermined substitution table called S-box. This operation introduces nonlinearity and complexity into the encryption process.
- ShiftRows: the rows of the state matrix are cyclically shifted to the left by a different number of positions. This step shuffles the bytes within the rows, helping to spread the data.

- MixColumns: the columns of the state matrix are mixed using a linear transformation. This step mixes the bytes within the columns, further increasing the spread of the data.
- AddRoundKey: each byte of the state is combined with a byte of the round key using the XOR operation. This operation introduces the encryption key into the transformation process.

A crucial aspect of AES is the generation of round keys, which occurs through a process known as key expansion. The master key is expanded into a series of round keys, one for each round of the encryption process. This process involves rotating, substituting, and combining parts of the master key to generate unique round keys.

2.0.1 Encryption and Decryption Functions

The two functions, encrypt_file_with_user_key and decrypt_file_with_user_key, are used to protect file data through encryption and decryption based on a specific key for each user. The encryption adopted is AES (Advanced Encryption Standard) in GCM mode (Galois/Counter Mode), a method that guarantees both confidentiality and integrity of the data. The encrypt function is used to encrypt the data of a file. It starts by retrieving a key associated with the user, which must be valid for the AES algorithm, i.e. have a length of 16, 24 or 32 bytes. If the key does not respect these dimensions, the function generates an error. A random initialization vector (IV) of 12 bytes is then created, necessary to guarantee the uniqueness of the encryption, even when the input data is identical. The encryption uses a Cipher object, configured with AES, GCM mode. The data is initially adapted through padding that makes it compatible with the format required by AES. Finally, the encryption result includes the IV, the authentication tag and the ciphertext, concatenated into a single block returned by the function. The decrypt function performs the reverse process, decrypting the previously encrypted data. Again, the user's key is retrieved, verifying that it meets the length requirements for AES. The input data is separated into its main components: IV, authentication tag and ciphertext. With these elements, a Cipher object is configured for decryption. The ciphertext is then decrypted and verified via the authen-

tication tag, ensuring that the data has not been tampered with. After decryption, the padding applied during encryption is removed, thus obtaining the original data. This approach ensures that files are protected throughout their lifecycle, ensuring both the confidentiality and integrity of the content.

```
1
2  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
       modes
3  from cryptography.hazmat.backends import default_backend
4  import os
5  from database import connessione
6  from cryptography.hazmat.primitives import padding
7
8  class Encryption:
9      def encrypt_file_with_user_key(file_content, user_id):
10         key = Encryption.get_user_key(user_id)
11         if len(key) not in [16, 24, 32]:
12             print(">> Invalid key length. Must be 16, 24, or 32 bytes."
       )
13             return False
14
15         iv = os.urandom(12)
16         cipher = Cipher(algorithms.AES(key), modes.GCM(iv), backend=
       default_backend())
17         encryptor = cipher.encryptor()
18
19         padder = padding.PKCS7(algorithms.AES.block_size).padder()
20         padded_data = padder.update(file_content) + padder.finalize()
21
22         ciphertext = encryptor.update(padded_data) + encryptor.finalize
       ()
23
24         return iv + encryptor.tag + ciphertext
25
26     def decrypt_file_with_user_key(encrypted_data, user_id):
27         try:
28             key = Encryption.get_user_key(user_id)
29             if len(key) not in [16, 24, 32]:
30                 print(">> Invalid key length. Must be 16, 24, or 32
       bytes.")
31                 return None
32
33             iv = encrypted_data[:12]
34             tag = encrypted_data[12:28]
35             ciphertext = encrypted_data[28:]
36
37             cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag),
       backend=default_backend())
38             decryptor = cipher.decryptor()
39
40             plaintext_padded = decryptor.update(ciphertext) + decryptor
       .finalize()
41
42             unpadder = padding.PKCS7(algorithms.AES.block_size).
       unpadder()
```

```
            plaintext = unpadder.update(plaintext_padded) + unpadder.
    finalize()

            return plaintext
        except Exception as e:
            print(f">> Error during decryption: {e}")
            return None
```

Chapter 3

Database

3.0.1 Login system

The login and registration system described in the code handles user authentication through an interaction with a MySQL database. When the user starts the system, they are asked to enter a username and password. The Login.execute() function handles the login flow, asking for the user's credentials and checking their validity through the Database.login() function. In the latter, a query is performed on the users table of the database to find a record matching the username entered. If the username exists, the password provided is compared to the one stored in the database, which has been previously encrypted using the bcrypt algorithm. The bcrypt.checkpw() function checks whether the password entered matches the one hashed in the database, thus ensuring that the user is securely authenticated. If authentication is successful, the user's ID is returned and the system displays a welcome message, otherwise an error is notified and the user can try again. This process allows the credentials to be kept secure, since the password is never stored in clear text in the database.

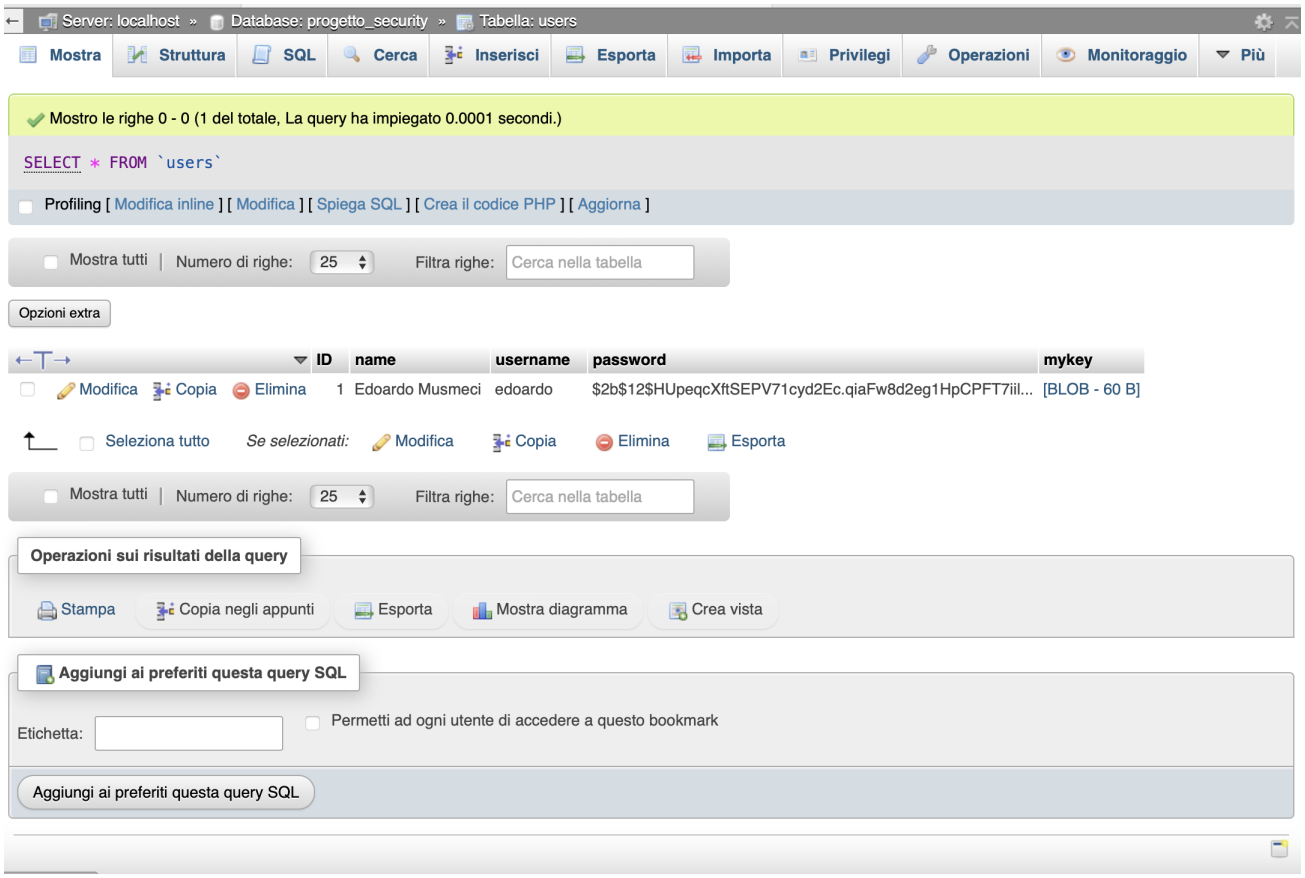

Figure 3.1: Table users

```python
import mysql.connector
from datetime import datetime
import os
import requests
import bcrypt


host = 'localhost'
database = 'progetto_security'
user = 'root'
password = ''


connessione = mysql.connector.connect(
    host=host,
    database=database,
    user=user,
    password=password
)


class Database:

    def hash_existing_passwords():
        if connessione.is_connected():
            try:
                cursor = connessione.cursor()
                cursor.execute("SELECT id, password FROM users")
```

```
29                utenti = cursor.fetchall()
30
31                for user_id, plain_password in utenti:
32                    if not plain_password.startswith('$2b$'):
33                        hashed_password = bcrypt.hashpw(plain_password.
    encode('utf-8'), bcrypt.gensalt())
34                        update_query = "UPDATE users SET password = %s
    WHERE id = %s"
35                        cursor.execute(update_query, (hashed_password.
    decode('utf-8'), user_id))
36
37                connessione.commit()
38                print(">> Password aggiornate con successo.")
39            except Exception as e:
40                print(f">> Errore durante l'aggiornamento delle
    password: {e}")
41            finally:
42                cursor.close()
43
44
45     def login(username, password):
46
47         if connessione.is_connected():
48             try:
49                 cursor = connessione.cursor()
50                 query = "SELECT id, password FROM users WHERE username
    = %s"
51                 cursor.execute(query, (username,))
52                 risultato = cursor.fetchone()
53
54                 if risultato:
55                     user_id, hashed_password = risultato
56                     if bcrypt.checkpw(password.encode('utf-8'),
    hashed_password.encode('utf-8')):
57                         return user_id
58                     else:
59                         return False
60                 else:
61                     return False
62             except Exception as e:
63                 print(f">> Database login error: {e}")
64                 return False
65             finally:
66                 cursor.close()
67         else:
68             print(">> Error: database connection failed")
69             return False
70
71     def register(username, password):
72
73         if connessione.is_connected():
74             try:
75                 cursor = connessione.cursor()
76                 hashed_password = bcrypt.hashpw(password.encode('utf-8'
    ), bcrypt.gensalt())
77                 query = "INSERT INTO users (username, password) VALUES
    (%s, %s)"
```

```
                cursor.execute(query, (username, hashed_password.decode
    ('utf-8')))
                connessione.commit()
                print(">> Registrazione completata.")
            except Exception as e:
                print(f">> Database registration error: {e}")
            finally:
                cursor.close()
        else:
            print(">> Error: database connection failed")
```

```
from database import *
from getpass import getpass

class Login:
    def execute():
        isNotLogged = True
        while isNotLogged:
            username = str(input("Username: "))
            password = str(getpass("Password: "))
            userid = Database.login(username, password)

            if userid:
                print(f"\n>> Benvenuto {username}\n")
                isNotLogged = False
                return userid
            else:
                print("\n>> Login fallito\n")

    def register():
        username = str(input("Nuovo username: "))
        password = str(getpass("Nuova password: "))
        Database.register(username, password)
```

3.0.2 Document Management

The three functions present operations related to the management of documents within a database, in particular for inserting, searching and updating files associated with a user. All three use a connection to a MySQL database, represented by the connection variable, to execute SQL queries. The getDocumentID function is designed to retrieve the ID of a document associated with a specific user, given his userid and the filename. If the connection to the database is active, the function executes a query to search for the DocumentID in the database in the documents table, using the UserID and the filename as search parameters. If the document is found, the function returns the ID of the document. Otherwise, it prints an error message and returns None. The get_max_id function is instead responsible for determining the maximum value of the DocumentID present

in the documents table. This is useful when you want to assign a new ID to a document that will be inserted into the database. The function executes a query to obtain the maximum value of DocumentID from the table and, if it exists, returns the next value (incremented by 1). If the table is empty, the function signals that this is the first file and returns a value of 1 for the first ID. Finally, the insert_documents_from_folder function handles inserting or updating documents into the database. If a file with the same name and associated with the same user already exists, the function updates the document date in the database to reflect the last modification. If the file does not exist, the function creates a new record in the documents table, assigning a new DocumentID using the get_max_id function. Additionally, for both cases, the function records the current date of the insert or update in the date field. After performing the desired operation, the function commits the transaction to the database.

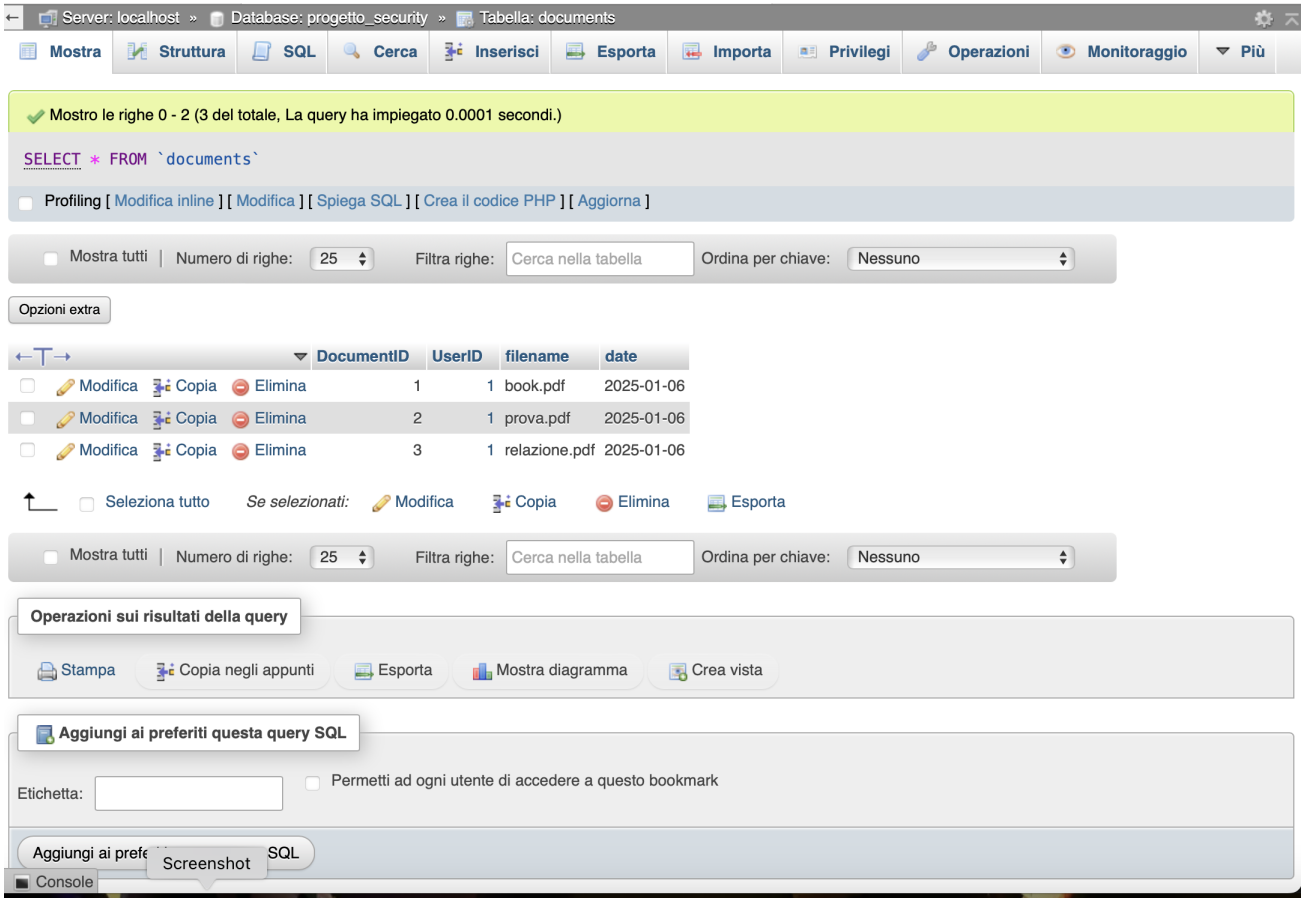

Figure 3.2: Table documents

```python
def getDocumentID(userid, filename):

        if connessione.is_connected():
            try:
                cursor = connessione.cursor()
                query = f"""
                    SELECT DocumentID
                    FROM documents
                    WHERE UserID = '{userid}' AND Filename = '{filename}'
                """
                cursor.execute(query)
                risultato = cursor.fetchone()
                if risultato:
                    return risultato[0]
                else:
                    print(f">> Error: Document '{filename}' not found for user '{userid}'.")
                    return None
            except Exception as e:
                print(f">> Database query error: {e}")
                return None
            finally:
                cursor.close()
        else:
            print(">> Error: database connection failed")
            return None


    def get_max_id():
        try:
            if connessione.is_connected():
                cursor = connessione.cursor()

                query = "SELECT MAX(DocumentID) AS MaxDocumentID FROM documents;"
                cursor.execute(query)
                result = cursor.fetchone()


                if result[0] is not None:
                    print(f"Max DocumentID is: {result[0]}")
                    return result[0] + 1
                else:
                    print("This is the first file in the table: DocumentID 1\n")
                    return 1

        except Exception as e:
            print(f"Error in query: {e}")


    def insert_documents_from_folder(folder_path, filename, user_id):
        try:
            if connessione.is_connected():
                cursor = connessione.cursor()


```

```
55                check_query = """
56                SELECT DocumentID FROM documents WHERE UserID = %s AND
   filename = %s
57                """
58                cursor.execute(check_query, (user_id, filename))
59                existing_document = cursor.fetchone()
60
61
62                if existing_document:
63                    document_id = existing_document[0]
64                    print(f"File '{filename}' gi  presente.
   Sovrascrivo con lo stesso DocumentID: {document_id}.")
65
66                    update_query = """
67                    UPDATE documents
68                    SET date = %s
69                    WHERE DocumentID = %s
70                    """
71                    current_date = datetime.now().strftime('%Y-%m-%d')
72                    cursor.execute(update_query, (current_date,
   document_id))
73                else:
74
75                    print(f"File '{filename}' non presente. Inserisco
   un nuovo record.")
76
77                    insert_query = """
78                    INSERT INTO documents (DocumentID, UserID, filename
   , date)
79                    VALUES (%s, %s, %s, %s)
80                    """
81                    document_id = Database.get_max_id()
82                    current_date = datetime.now().strftime('%Y-%m-%d')
83                    cursor.execute(insert_query, (document_id, user_id,
    filename, current_date))
84
85
86                connessione.commit()
87                print(f"File '{filename}' gestito correttamente nel
   database.")
88
89            except Exception as e:
90                print(f"Errore durante l'inserimento o aggiornamento nel
   database: {e}")
91            finally:
92                cursor.close()
```

Chapter 4

Docker Containers

In this project, we use Docker, an open-source platform for building, deploying, and running containerized applications, to develop a distributed system for securely managing fragments of a PDF file. But what are containers? Containers are lightweight, portable software units that include everything an application needs to run: code, runtime, libraries, dependencies, and settings. This technology isolates the application from the underlying operating system environment, ensuring that it runs identically on any system that supports Docker. Docker makes it easy to create and manage containers through intuitive commands and tools, including configuration files like docker-compose.yml, that allow you to declaratively define an infrastructure composed of multiple containers. In our project, the focus is on secure and scalable distribution of fragmented data. A PDF file is split into multiple pieces, each encrypted to preserve the confidentiality of the information. These fragments are then sent to three separate containers using HTTP POST requests, managed by a Flask server. Flask's role is to act as a central point to receive fragments and forward them to dedicated containers. Each container is configured via a docker-compose.yml file, which defines the specifications needed to create and connect them. This file allows you to quickly build a distributed environment where containers can work independently but in a coordinated way. This approach demonstrates the effectiveness of Docker in supporting innovative solutions for distributed processing of sensitive data, taking full advantage of the modularity and autonomy that containers offer.

4.0.1 docker-compose.yml code

```
version: '3.9'

services:
  container1:
    image: python:3.9-slim
    container_name: container1
    volumes:
      - ./server.py:/app/server.py
      - ./data/container1:/app/data
    ports:
      - "5001:5000"
    working_dir: /app
    environment:
      - FLASK_ENV=development
    command: >
      sh -c "pip install flask && python server.py"

  container2:
    image: python:3.9-slim
    container_name: container2
    volumes:
      - ./server.py:/app/server.py
      - ./data/container2:/app/data
    ports:
      - "5002:5000"
    working_dir: /app
    environment:
      - FLASK_ENV=development
    command: >
      sh -c "pip install flask && python server.py"

  container3:
    image: python:3.9-slim
    container_name: container3
    volumes:
      - ./server.py:/app/server.py
      - ./data/container3:/app/data
    ports:
      - "5003:5000"
    working_dir: /app
    environment:
      - FLASK_ENV=development
    command: >
      sh -c "pip install flask && python server.py"
```

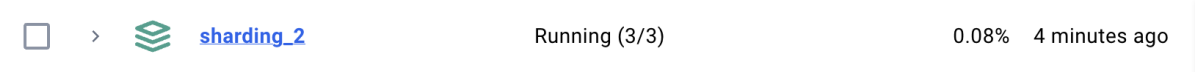

Figure 4.1: Container Sharding

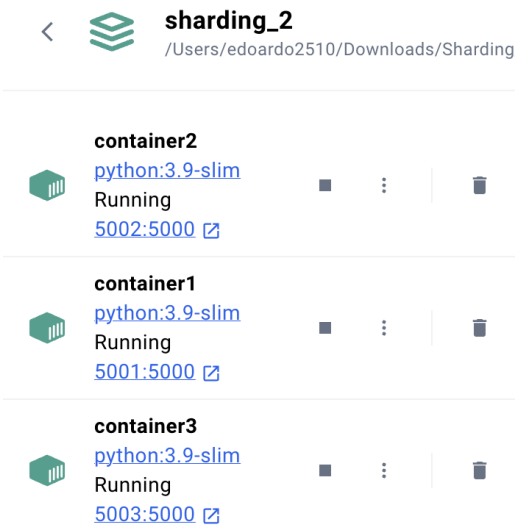

Figure 4.2: Containers

4.0.2 Flask

Flask is a lightweight framework for building web applications in Python. It is widely regarded for its simplicity and flexibility, allowing developers to quickly build servers and web services without imposing too many rules or complex configurations. Flask provides the basic tools to handle HTTP requests, create APIs, and integrate custom functionality, making it an ideal choice for both small projects and more advanced applications. The server.py file allows you to upload, download, and delete files on a server.

- upload a file: When a user uploads a file with the specified name (called a "shard"), the server saves it in a folder called /app/data. If the upload is successful, it returns a success message; otherwise, it reports an error.

- download a file: the user can request a file by specifying its name. The server searches for the file in the folder and, if it exists, sends it back as a download. If the file is not there, it notifies the user that it was not found.

- delete files: the user can ask the server to delete all files that start with a certain prefix (a word or initial string). The server checks the files in the folder and deletes those that match the given prefix. Finally, it returns the list of deleted files or warns if there were none.

```python
from flask import Flask, request, Response
import os

app = Flask(__name__)

@app.route('/upload', methods=['POST'])
def upload_fragment():
    fragment = request.files['file']
    shard_name = request.form.get('shard_name', 'shard')
    save_path = os.path.join('/app/data', shard_name)

    try:
        fragment.save(save_path)
        return f"Shard {shard_name} saved successfully at {save_path}", 200
    except Exception as e:
        return f"Error saving shard: {e}", 500


@app.route('/download', methods=['GET'])
def download_fragment():
    shard_name = request.args.get('shard_name')
    file_path = os.path.join('/app/data', shard_name)

    if os.path.exists(file_path):
        try:
            with open(file_path, 'rb') as file:
                file_content = file.read()
            return Response(
                file_content,
                mimetype='application/octet-stream',
                headers={
                    'Content-Disposition': f'attachment; filename={shard_name}'
                }
            )
        except Exception as e:
            return f"Error reading shard: {e}", 500
    else:
        return f"Shard {shard_name} not found.", 404



@app.route('/delete', methods=['POST'])
def delete_shards():
    shard_prefix = request.form.get('shard_prefix')
    data_path = '/app/data'

    try:
        deleted_files = []
        for filename in os.listdir(data_path):
            if filename.startswith(shard_prefix):
                file_path = os.path.join(data_path, filename)
                os.remove(file_path)
                deleted_files.append(filename)

        if deleted_files:
            return f"Deleted files: {', '.join(deleted_files)}", 200
```

```
        else:
            return "No files matched the prefix.", 404
    except Exception as e:
        return f"Error deleting files: {e}", 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

4.0.3 Sending shards to containers

The uploadShardToContainers function handles uploading a list of data shards (data_shards_list) to a specified set of containers. For each shard, it calculates the destination using rotation logic based on the total number of available container. Each shard is uniquely identified with a name based on a document ID and the shard index. Each shard is then sent to the designated server via a POST request that includes the shard as a file and some additional information in the request data. The function verifies the success of the upload by examining the HTTP response code from the server and provides log messages to indicate the outcome of each upload.

```
def uploadShardToContainers(data_shards_list, document_id):

        server_urls = [
            "http://localhost:5001/upload",
            "http://localhost:5002/upload",
            "http://localhost:5003/upload"
        ]

        try:

            num_container = len(server_urls)

            for index, data_shard in enumerate(data_shards_list):

                server_url = server_urls[index % num_container]

                shard_name = f"shard_{document_id}_{index}"

                files = {
                    'file': (shard_name, data_shard)
                }
                data = {
                    'shard_name': shard_name
                }
                response = requests.post(server_url, files=files, data=
    data)

                if response.status_code == 200:
                    print(f"Shard {shard_name} uploaded successfully to
     {server_url}")
                else:
```

```
30                     print(f"Error uploading shard {shard_name} to {
   server_url}: {response.text}")
31
32         except Exception as e:
33             print(f"Error during shard upload: {e}")
```

```
Tutti i file sono stati inseriti con successo!
File 'book.pdf' divided into 3 fragments.
Shard shard_1_0 uploaded successfully to http://localhost:5001/upload
Shard shard_1_1 uploaded successfully to http://localhost:5002/upload
Shard shard_1_2 uploaded successfully to http://localhost:5003/upload

>> File split and uploaded to containers successfully!
```

Figure 4.3: Result of the decomposition operation

The above photo shows the result of the split and upload operation of a PDF file called "book.pdf". The file was successfully split into 3 fragments (shards), each encrypted and sent to three different Docker containers via HTTP POST requests. The fragments were uploaded to the containers with addresses on ports 5001, 5002 and 5003 respectively. The message confirms the completion of the operation without errors.

```
2025-01-05 16:17:47 192.168.65.1 - - [05/Jan/2025 15:17:47] "POST /upload HTTP/1.1" 200 -
```

Figure 4.4: Log Container

Chapter 5

Conclusions

The developed project represents an effective solution to guarantee the security of PDF files during the storage and transfer process. Through the application of advanced encryption techniques and the adoption of a distributed architecture based on Docker containers, the main objectives were achieved: protecting sensitive data and ensuring a robust and scalable system. The decomposition of PDF files into three encrypted fragments using the AES algorithm added an additional level of security, reducing the risk of compromise even in the event of unauthorized access to one of the fragments. The distribution of each fragment on separate Docker containers via HTTP POST requests represents a distributed and modular solution, easily adaptable to different application scenarios. The reverse process, which includes the recovery of fragments via HTTP GET requests, decryption and recomposition of the original PDF file, demonstrated the consistency and effectiveness of the system. Another key element is the automatic insertion of files into the MySQL database managed with phpMyAdmin. This feature not only facilitates the management and organization of documents, but also allows centralized and secure storage. The inclusion of the login system with secure authentication, based on saving hashed passwords in the database, is a key point to ensure controlled access to users. This feature adds an additional layer of protection, ensuring that only authorized users can access the system. This project represents an excellent basis for further innovations and applications in the field of information security and distributed management of sensitive data.