פרויקט סיום: Keylogger "תוכנת ריגול"

הקדמה

בחלק הראשון של הקורס בנינו KeyLogger שמאזין להקשות מקשים, מציג אותן לקונסול, ומאפשר למשתמש להוציא פלט על פי דרישה.

כעת, נרחיב את תרגיל מקורי זה ל**תרגיל הסיום של הקורס!**

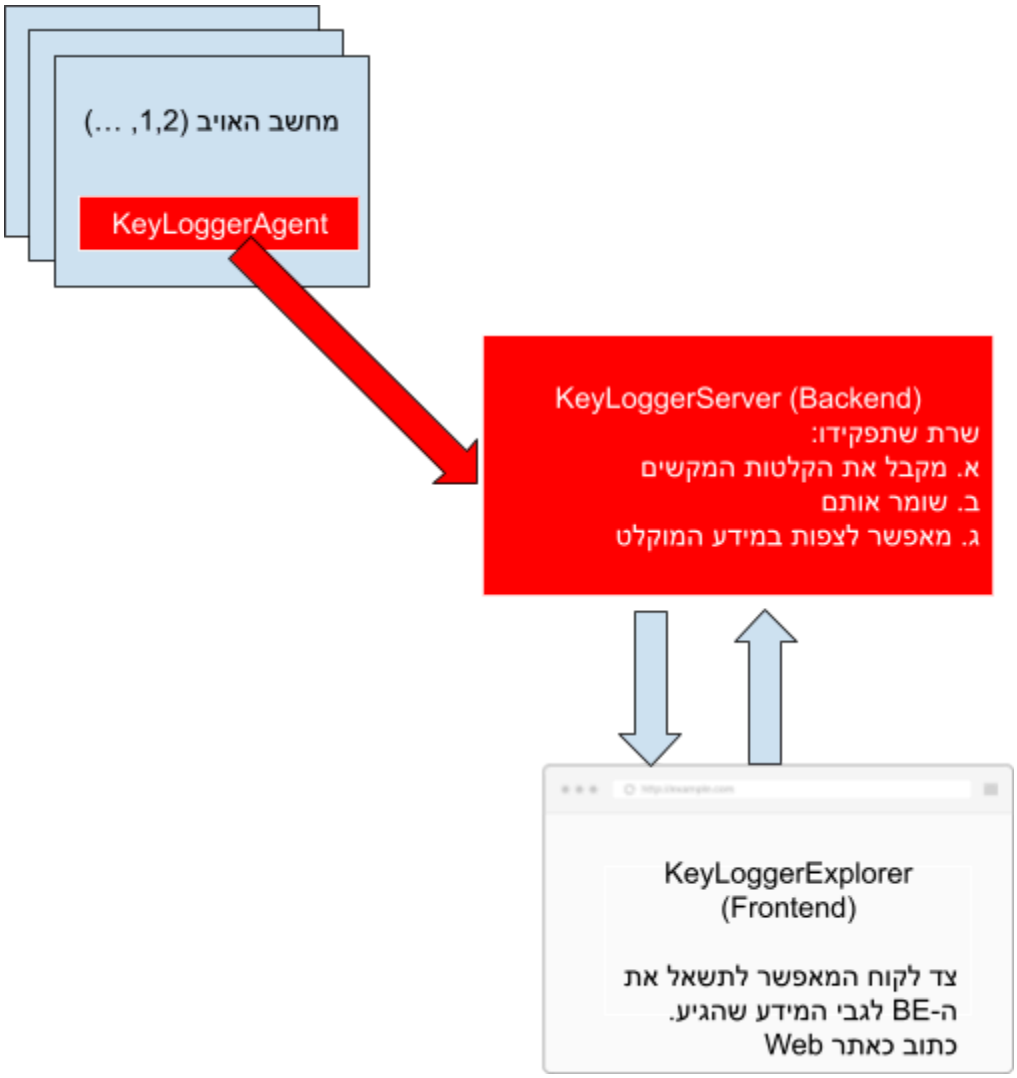
בפרויקט זה נבנה **מערכת מלאה** המדמה כלי סייבר בסיסי:

1. **כלי השטח (KeyLoggerAgent):** תוכנה הרצה על מחשב היעד, אוספת הקשות מקלדת, מצפינה אותן ושולחת לשרת.
2. **צד השרת (Backend):** שרת Flask המקבל את הנתונים, מפענח אותם ושומר אותם.
3. **ממשק משתמש (Frontend):** דף אינטרנט המציג את הנתונים שנאספו.

אזהרה משפטית ואתית:
תוכן תרגיל זה מיועד למטרות לימודיות בלבד. אין להשתמש בכלי לצרכים בלתי חוקיים או לא־אתיים. העבודה מתבצעת בסביבה אישית או וירטואלית בלבד, ולאחר אישור בעלי המחשב. כל המשתתפים נדרשים להקפיד על כללי אתיקה מקצועית והגבלות משפטיות.

תכן כללי של המערכת

התרשים הבא מראה את **כל רכיבי המערכת.** שימו לב שנעבוד על כל רכיב בנפרד.



חלק א׳ - ״כלי סייבר״ (KeyLoggerAgent)

מטרה: לפתח מערכת המדמה כלי Keylogger מתקדם.

שלב 0: עיצוב (Design)

משימה:

עליכם לתכנן את חלוקת האחריות במערכת.

חשבו על השאלות הבאות ותנו תשובות שיכוונו את המחשבה שלכם:

- איך תחלקו את תפקיד איסוף הקשות (KeyLoggerService) לעומת תפקיד ניהול הכתיבה לקובץ?

- באיזה אופן תבחרו לנהל את הזיכרון של הכלי? מה יהיה המקום המתאים ביותר לעשות זאת?
- כיצד תוכלו לעצב את הממשק כך שבעתיד יהיה קל להוסיף מחלקת NetworkWriter?
- איפה לדעתכם ההצפנה צריכה להיכנס בכתיבת הכלי?

רציונל:

המטרה היא לא "להכתיב" פתרון סופי, אלא לתת לכם כמה רמזים שיסייעו לחשיבה עצמאית:

- נסו לחשוב על ארכיטקטורה שבה המחלקה המרכזית (KeyLoggerManager) אוספת את הנתונים מה־KeyLoggerService ומחליטה מתי להעבירם לכתיבה.
- שקלו להשתמש במנגנון תזמון (Timer או Thread) במחלקת המנהל על מנת לשלוט בתדירות העדכון.
- חשבו על איזה ממשקים עליכם להגדיר במידה ויש לכם מחלקות שאמורות לממש פונקציונליות דומה.

תוצאה נדרשת:

תרשים זרימה או תיאור טקסטואלי קצר שמתאר את חלוקת האחריות, לדוגמה:

- **KeyLoggerService:** "איסוף נתונים והחזרתם לפי בקשה".
- **KeyLoggerManager:** "פועל בלולאה, בכל X שניות קורא את ההקשות, מאגד אותן ל־Buffer, מעביר את המידע אל FileWriter ואופציונלית מעביר אותו גם ל־NetworkWriter".
- **FileWriter:** "מקבל מחרוזת טקסט ומבצע כתיבה לקובץ עם חותמת זמן, לאחר שעבר הצפנה (באמצעות Encryptor)".
- **NetworkWriter:** "מקבל נתונים ומעביר אותם לשרת באמצעות הממשק IWriter".
- **Encryptor:** אחראית לבצע XOR או הצפנה אחרת.

שלב 1: מימוש KeyLoggerService

בשלב זה נבצע יישור קו ונממש את KeyLoggerService בממשק אחיד. המטרה היא ליצור מחלקה שמאזינה להקשות מקלדת ושומרת אותן באופן זמני בזיכרון (Buffer) לצורך שימוש בהמשך.

דרישות המימוש:

1. נגדיר ממשק `IKeyLogger` עם הפונקציות הבאות:
   - `start_logging()`: מפעיל את האזנה להקשות.
   - `stop_logging()`: עוצר את האזנה.
   - `get_logged_keys() -> list[str]`: מחזיר רשימה של ההקשות שנאספו.
2. נממש מחלקה `KeyLoggerService` אשר תיישם את הממשק `IKeyLogger`.
3. נשתמש בספריית `pynput` כדי להאזין להקשות מקלדת.

רציונל

בשלב זה, המטרה היא "לנקות רעש" ולספק נקודת התחלה זהה כך שבהמשך ניתן יהיה להוסיף יכולות כמו הצפנה, כתיבה לקובץ ושליחה לשרת. שימוש בממשק מוגדר מאפשר קוד ברור, תחזוקתי ומודולרי, כך שניתן יהיה להחליף או לשדרג את מימוש `KeyLoggerService` בקלות.

מחלקת עזר:

```
from abc import ABC, abstractmethod
from typing import List

class IKeyLogger(ABC):
    @abstractmethod
    def start_logging(self) -> None:
        pass

    @abstractmethod
    def stop_logging(self) -> None:
        pass

    @abstractmethod
    def get_logged_keys(self) -> List[str]:
        pass
```

תוצאה נדרשת

- `KeyLoggerService` תקני שמאזין להקשות מקלדת ושומר אותן בזיכרון.
- המימוש יתבצע לפי ממשק אחיד (`IKeyLogger`), כך שבעתיד ניתן יהיה להרחיב את הקוד בקלות.

שלב 2: מימוש FileWriter

משימה

לממש מחלקה בשם **FileWriter** אשר אחראית על כתיבה לקובץ.

רציונל

הפרדת אחריות: FileWriter מטפל אך ורק בכתיבה לקובץ, והמחלקה שמנהלת את הזרימה (KeyLoggerManager) תדאג לעבד את הנתונים לפני שליחתם.

תוצאה נדרשת

FileWriter שמקבל מחרוזת טקסט ומבצע כתיבה לקובץ.

דוגמת עזר

```python
# iwriter.py
from abc import ABC, abstractmethod
class IWriter(ABC):
    @abstractmethod
    def send_data(self, data: str, machine_name: str) -> None:
        pass
```

שלב 3: כתיבת XOR Encryption

משימה

1. כתבו מחלקת Encryptor אשר מבצעת הצפנת XOR בסיסית על המידע שנשמר.
2. שמרו מפתח פשוט (למשל מחרוזת או מספר) המשמש לפעולת הXOR.

רציונל

הצפנה היא חלק מרכזי בתרגיל - המידע הרגיש לא אמור להישמר כטקסט גלוי בדיסק. XOR היא שיטה פשוטה הממחישה עקרונות של הצפנה.

תוצאה נדרשת

הקובץ יישמר בצורה מוצפנת - כאשר משתמש שינסה לפתוח את הקובץ ישירות יראה טקסט "משובש" (Garbage). רק מי שיש לו את פונקציית הפענוח והמפתח יוכל לקרוא את המידע.

נספח XOR

בנספח שיופיע בהמשך, נסביר מהם הפעולות הבסיסיות הדרושות למימוש XOR. הרעיון העיקרי:

- נתון מפתח (מספר או ASCII code).
- לעבור על כל בית (byte) בטקסט ולבצע byte ^ key.
- אותה פעולה ממש משמשת גם להצפנה וגם לפענוח.

שלב 4: מימוש KeyLoggerManager

משימה:

לממש מחלקה **KeyLoggerManager** אשר תקבל את ה־KeyLoggerService ואת ה־FileWriter, ותנהל את ה־Buffer המרכזי.

תבצע, בכל פרק זמן מוגדר (למשל, כל 5 שניות):

- איסוף ההקשות מה־KeyLoggerService.
- איגוד הנתונים ל־Buffer.
- כאשר עוצרים את המערכת (או בסיום תקופתי):
  - להוסיף חותמת זמן לנתונים.
  - לבצע הצפנה (באמצעות פונקציית Encryptor).
  - להעביר את המחרוזת המעובדת ל־FileWriter (ולאופציונלי ל־NetworkWriter).

רציונל:

- מרכזת את הלוגיקה של איסוף, עיבוד והעברת הנתונים.
- מאפשרת גמישות בשינויים - לדוגמה, שינוי תדירות העדכון, שינוי שיטת ההצפנה, הוספת ניהול Buffer וכו'.

תוצאה נדרשת:

- KeyLoggerManager שמתחילה את האיסוף, אוגדת את הנתונים, מוסיפה חותמת זמן ומבצעת הצפנה לפני שליחת הנתונים למחלקת הכתיבה (וגם, אם מוגדר, למחלקת השליחה לרשת

שלב 4: סוכן חשאי מפענח את הקובץ

משימה

1. כתבו סקריפט (לדוגמה decrypt_file.py) שמקבל מהמשתמש ב־CLI (שורת הפקודה) את הנתיב לקובץ המוצפן, ואת מפתח הXOR.
2. הסקריפט יטען את הקובץ, יבצע פענוח (אותה פעולה של XOR), וידפיס את התוכן המפוענח למסך (std output).

רציונל

אנחנו מדמים מצב שבו הסוכן "גונב" את הקובץ המוצפן באמצעות Disk-On-Key, ולאחר מכן מפעיל כלי צד ג' לפענוח.

תוצאה נדרשת

קובץ Python המאפשר לשחזר את המידע הגלוי מהקובץ שהKeyLogger יצר.

שלב 5: כתיבה רשתית

משימה:

לממש מחלקה בשם **NetworkWriter** העומדת בממשק **IWriter**, אשר תשתמש בספריית requests לשליחת נתונים לשרת.

רציונל:

באמצעות ממשק IWriter, ניתן להחליף את המימוש בקלות בעתיד, מבלי להשפיע על שאר המערכת.
שימו לב שאתם משתמשים בהצפנה שרשמתם!

תוצאה נדרשת:

- NetworkWriter שמיישם את המתודה:

```
send_data(data: str, machine_name: str) -> None
```

הערות נוספות

תיעוד ועצות עבודה:

- מומלץ לצרף README עם הוראות הפעלה, תרשים זרימה ותיאור חלוקת המערכת.
- חשבו על טיפול בשגיאות (Exceptions) וניהול לוגים (Logging) בכל רכיב.

חלק ב׳ - בניית Backend

הקדמה

כעת, לאחר שיש לנו כלי שיכול לתקשר בצורה רשתית, נעבור לכתיבת ה-Backend.
השרת שלנו ישמש ל2 שימושים מרכזיים:

1. קבלת נתוני Keylogger מהכלי (שנשלחים מהמערכת המפותחת בחלק "כלי הסייבר") ושמירתם לצורך ניתוח עתידי.
2. מתן אפשרות לשליפת נתוני ההקשות, כך שהמשתמש יוכל לגשת לנתונים אלו דרך ממשק המשתמש (Frontend).

ה-Backend ייכתב ב-**Python** ומומלץ לעבוד עם **Flask** (למרות שתוכלו לעבוד עם כל גישה אחרת גם כן).

שלב 0: Design - רמזים לתכנון מערכת אחסון הנתונים

משימה:

- הגדירו את ארכיטקטורת הנתונים עבור ה-Backend.
- חשבו כיצד תרצו לארגן את המידע במערכת הקבצים כך שתוכלו להבדיל בין כל מכונה למכונה אחרת, ולשמור בצורה היררכית קבצי טקסט המכילים את ההקשות.

רציונל:

- רמז: שקלו האם אתם מעדיפים ליצור קובץ חדש בכל POST או לאגד מספר שליחות לאותו קובץ (למשל, קובץ עם שם שמבוסס על חותמת זמן).
- נסו לתעד את הרעיון באמצעות תרשים זרימה קצר או תיאור טקסטואלי, כדי להבהיר את חלוקת האחריות בין התיקיות והקבצים.

תוצאה נדרשת:

תרשים זרימה או תיאור טקסטואלי של מבנה data/, לדוגמה:

- /data
  - /machine1
    - Log_2025-02-03_10-15-00.txt
    - Log_2025-02-03_10-20-00.txt
  - /machine2
    - Log_2025-02-03_11-00-00.txt

דוגמה למבנה תיקיות:

```
backend/
 ├── app.py
 ├── data/        <-- תיקייה זו תכיל תיקיות לכל מכונה
 └── README.md
```

שלב 1: התקנת Flask והכנת מבנה הפרויקט

משימה:

1. ודאו ש-Flask מותקן (אם לא, התקינו עם pip install flask).
2. צרו תיקייה חדשה בשם backend ונווטו אליה.
3. בתוך התיקייה, צרו את הקבצים הבאים:
   a. app.py (הקובץ הראשי של השרת).
   b. /data (תיקייה שבה יאוחסנו קובצי ה-KeyLogger).
      i. החליטו על מבנה קבצים פנימי שיאפשר לכם לשמור עבור כל מחשב תיקייה שמלאה בתוצרי ההקלדות שלו.

רציונל:

הכנה נכונה של מבנה הפרויקט תאפשר לכם הרחבה וניהול קל של הקוד בהמשך.

תוצאה נדרשת:

מבנה פרויקט הכולל תיקייה לנתונים וקובץ app.py ריק.

שלב 2: יצירת שרת Flask בסיסי

משימה:

1. פתחו את app.py והוסיפו את הקוד הבא:

```
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/')
def home():
    return "KeyLogger Server is Running"
if __name__ == '__main__':
    app.run(debug=True)
```

2. הריצו את השרת (python app.py) וודאו שהוא פועל.
3. גשו לכתובת http://127.0.0.1:5000 ובדקו שהעמוד נטען בהצלחה.

חומרי עזר:

אתם מוזמנים להשתמש בתיעוד הרשמי של flask, אשר זמין בכתובת https://flask.palletsprojects.com/en/stable/quickstart.

רציונל:

יצירת שרת בסיסי הוא הצעד הראשון ליצירת API פונקציונלי.

תוצאה נדרשת:

שרת Flask פעיל שמציג הודעה בדפדפן.

שלב 3: יצירת API לקבלת נתוני Keylogger

משימה:

צרו האזנה באמצעות flask שתאזין לשליחות מידע של הכלי.

רציונל:

ארגון הנתונים בצורה היררכית (תיקייה למכונה, קבצים ל־session) מאפשר גישה קלה וניהול ברור של הנתונים.

תוצאה נדרשת:

מידע שישלח מהכלי יתקבל ע"י השרת, יפוענח וישמר לדיסק תחת תיקיית /data.

- POST ל-/api/upload שקולט נתונים ושומר אותם במבנה:
  - data/<machine>/log_<timestamp>.txt

להלן קוד ממנו ניתן להתחיל:

```
# app.py (המשך)
from flask import request
import time

def generate_log_filename():
    # מחזירה שם קובץ המבוסס על חותמת זמן
    return "log_" + time.strftime("%Y-%m-%d_%H-%M-%S") + ".txt"

@app.route('/api/upload', methods=['POST'])
def upload():
    data = request.get_json()
    if not data or "machine" not in data or "data" not in data:
        return jsonify({"error": "Invalid payload"}), 400

    machine = data["machine"]
    log_data = data["data"]

    # יצירת תיקייה עבור המכשיר אם אינה קיימת
    machine_folder = os.path.join(DATA_FOLDER, machine)
    if not os.path.exists(machine_folder):
        os.makedirs(machine_folder)

    # יצירת שם קובץ חדש לפי חותמת זמן
    filename = generate_log_filename()
    file_path = os.path.join(machine_folder, filename)

    # ניתן להוסיף כאן עיבוד נוסף, למשל הוספת חותמת זמן נוספת בתוך הקובץ
    with open(file_path, "w", encoding="utf-8") as f:
        f.write(log_data)

    return jsonify({"status": "success", "file": file_path}), 200
```

שלב 4: יצירת API להחזרת רשימת המחשבים

משימה:

צרו API בשרת שלכם שיקרא `get_target_machines_list()`, ויאפשר לקבל את רשימת המחשבים שעליהם הורץ הכלי עד כה.

רציונל:

רשימת המחשבים נשלפת ישירות לפי שמות התיקיות, כך שאין צורך במסד נתונים כרגע.

תוצאה נדרשת:

קריאת GET ל־/api/get_target_machines_list תחזיר מערך JSON עם שמות המכונות.

שלב 5: שליפת נתוני ההקלדה של מחשב ספציפי

משימה:

צרו API בשרת שלכם שיקרא `get_target_machine_key_strokes(target_machine)`, ויאפשר לקבל את הקלדות המקלדת ממחשב ספציפי.

רציונל:

API זה מאפשר לשלוף את המידע הרלוונטי לכל מחשב שממנו נאספו נתונים.

תוצאה נדרשת:

קריאה ל-/api/get_keystrokes?machine=computer1 תחזיר JSON עם הנתונים של אותו מחשב.

הערות ותיעוד נוספים

README ותיעוד:

צרפו קובץ README המסביר:

- כיצד להריץ את השרת (לדוגמה: python app.py).
- מבנה תיקיות הפרויקט (הדגש על data/ עם תיקיות לכל מכונה).
- תיאור קצר של כל Endpoint - הקלט והפלט שלו.
- הפניות לתיעוד Flask Quickstart (Flask).

טיפול בשגיאות וניהול לוגים:

שקלו להוסיף טיפול מתקדם בשגיאות ולוגים (ניתן להשתמש במודול logging של Python).

סיכום תוצרי הגשה (עבור חלק ה-Backend):

- קוד המקור בקובץ app.py הכולל את ה-Endpoints הבאים:
  - / - הודעת מצב.
  - /api/upload - קבלת נתוני Keylogger ושמירתם במבנה תיקיות שבו כל מכונה היא תיקייה עם קבצי טקסט.
  - /api/get_target_machines_list - החזרת רשימת המכונות (תיקיות).
  - /api/get_keystrokes - שליפת תוכן קבצי הלוג עבור מכונה מסוימת.
- מבנה תיקיות שמדגיש את ארגון הנתונים תחת data/.
- README עם הוראות הפעלה, תרשים זרימה ותיאור המערכת.

חלק ג': בניית ממשק Frontend (HTML, CSS, JS)

הקדמה

כעת, לאחר שכתבנו את רכיב ה-KeyLogger וBackend שמתאים לו ושמרנו את הנתונים, נבנה ממשק Frontend בסיסי שמאפשר לצפות במידע שנאסף.

מטרות השלב:

- הבנה בסיסית של HTML ו-CSS ליצירת דפי אינטרנט פשוטים.
- שימוש ב-JavaScript כדי לתקשר עם השרת ולהציג מידע רלוונטי.
- תרגול קריאות API בסיסיות באמצעות fetch.

שלב 1: תכנון הממשק

משימה:

- תכננו דף המציג רשימת מחשבי יעד עליהם רץ הכלי.
- תכננו תצוגה המציגה את הקשות המקלדת לכל מחשב יעד.

רציונל:

תכנון נכון ימנע עבודה כפולה בהמשך.

תוצאה נדרשת:

Wireframe בסיסי של הממשק.

שלב 2: מימוש דף תצוגה של מחשבי היעד

משימה:

בהתאם לתכנון שלכם, צרו דף אשר מאפשר להציג את רשימת מחשבי היעד עליהם רץ הכלי.

רציונל:

שלב זה יאפשר להכיר את היסודות של HTML ויבנה שלד בסיסי לממשק המשתמש.

תוצאה נדרשת:

קובץ index.html המכיל עיצוב לטעמכם וכפתור רענון.

שלב 2: כתיבת קובץ CSS לעיצוב בסיסי

משימה:

השתמשו בcss על מנת לעצב את הדף שבניתם זה עתה בצורה יותר נעימה למשתמש.

רציונל:

שיפור חוויית המשתמש והנגשת הנתונים בצורה נוחה לקריאה.

תוצאה נדרשת:

העמוד המקורי שעיצבתם, כעת עם עיצוב.

שלב 3: משיכת רשימת המחשבים מהשרת

משימה:

השתמשו בAPI של השרת שכתבנו בשלב הקודם כדי להשיג את רשימת מחשבי היעד עליהם רצה התוכנה. השתמשו בJavascript על מנת לעשות זאת, ועדכנו את הטבלה שעיצבתם בשלב הקודם בהתאם.

```
fetch('/api/get_target_machines_list')
  .then(response => response.json())
  .then(data => console.log(data));
```

רציונל:

בשלב זה נתרגל עבודה עם ה-DOM ועדכון דינמי של HTML בעזרת JavaScript דרך API.

תוצאה נדרשת:

קריאה ל-API בעת לחיצה על הכפתור, כאשר המידע מוצג בקונסול הדפדפן. לאחר מכן, רשימת המחשבים מוצגת בטבלה, עם כפתור ליד כל מחשב.

שלב 4: משיכת נתוני ההקשות עבור מחשב מסוים

משימה:

השתמשו בAPI של השרת שכתבנו בשלב הקודם כדי להשיג את רשימת ההקלדות עבור מחשב יעד מסויים, לאחר שהמשתמש לוחץ על כפתור כפי שתכננתם. בעת הלחיצה יתבצע API נוסף לשרת, והעמוד שנראה למשתמש יתעדכן בהתאם.

רציונל:

נלמד כיצד לבצע קריאה ממוקדת יותר ולהציג נתונים בהתאם לבחירת המשתמש.

תוצאה נדרשת:

בלחיצה על כפתור "הצג הקשות מקלדת", הנתונים מוצגים באזור ייעודי.

נספחים

נספח א - XOR Encryption Basics

מה זה XOR

XOR (Exclusive OR) הוא שער לוגי בסיסי שמקבל שני ביטים ומחזיר 1 רק אם אחד מהם שונה מהשני.
בטבלה הלוגית:
0 0 = 0 XOR
1 0 = 1 XOR
0 1 = 0 XOR
1 1 = 1 XOR
כאשר משתמשים בXOR עבור ערכים בייטים (8 ביטים), הפעולה מבוצעת על כל ביט בנפרד.

למה XOR משמש להצפנה בסיסית

- אפשר לקחת כל בייט במידע ולבצע עליו XOR עם בייט של המפתח.
- אותה פעולה (XOR עם אותו מפתח) הופכת גם את המידע המוצפן חזרה למידע גלוי.

דוגמה

נניח שהמפתח הוא התו 'K' (ב־ASCII, הערך 75, 01001011, 0x4B).

- נקרא לכל בייט של הטקסט שלנו b.
- נפיק בייט מוצפן encryptedByte = b ^ keyByte.
- לשחזור, מבצעים שוב decryptedByte = encryptedByte ^ keyByte.

נספח B - הסבר על REST API Design with Flask

- **REST API:** עיצוב שירותי ווב המשתמשים במתודות HTTP (GET, POST, וכו').
- **Flask:** הוא Framework קל משקל בפייתון לבניית API.

עקרונות:

- לקבל קלט בפורמט JSON (באמצעות request.get_json()),
- להחזיר תגובות בפורמט JSON (באמצעות jsonify()),
- לטפל בשגיאות ולהחזיר קודי סטטוס מתאימים.