PMR 3412 - Redes Industriais

# Criptografia na Internet

Gustavo Marangoni Rubo - 4584080

## 1 Exercício 1 - Senhas e Login

### 1.1 API RESTful de usuários

Foi criado um servidor em python que recebe requisições REST para gerenciar um banco de dados de usuários. As requisições foram feitas com a extensão "RESTED Client"do firefox. Segue o código do servidor:

```python
from flask import Flask
from flask import request
import json

app = Flask(__name__)

# Inicializando a lista de usu rios
users = [
    {
        "user_id": 1,
        "user_name": "Gustavo Rubo",
        "user_password": "4584080",
        "user_email": "gustavo.rubo@usp.br"
    }
]

id_counter = 2

# Rota para cole  o de usu rios
@app.route("/users", methods=["POST", "GET", "PUT", "DELETE"])
def process_users():
    global id_counter, users
    if request.method == "POST":
        # Criar um usu rio e adicion -lo    lista
        user = {
            "user_id": id_counter,
            "user_name": request.json["user_name"],
            "user_password": request.json["user_password"],
            "user_email": request.json["user_email"]
        }
        id_counter += 1
        users.append(user)

        return json.dumps(user), 201

    elif request.method == "GET":
        # Retornar todos os usu rios
        return json.dumps(users), 200

    elif request.method == "PUT":
        return "Method not allowed", 405
    elif request.method == "DELETE":
        return "Method not allowed", 405

# Rota espec fica para usu rios
@app.route("/users/<user_id>", methods=["POST", "GET", "PUT", "DELETE"])
def process_users_id(user_id):
    global id_counter, users
    if request.method == "POST":
        return "Method not allowed", 405

    elif request.method == "GET":
```

```
        # Retornar o usu rio com o id especificado
        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            return json.dumps(user), 200
        else:
            return "User not found", 404

    elif request.method == "PUT":
        # Modificar o usu rio com o id especificado

        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            user["user_name"] = request.json["user_name"]
            user["user_password"] = request.json["user_password"]
            user["user_email"] = request.json["user_email"]
            return json.dumps(user), 200
        else:
            return "User not found", 404

    elif request.method == "DELETE":
        # Remover o usu rio com o id especificado
        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            users[:] = [u for u in users if u["user_id"] != int(user_id)]
            return "User deleted succesfully", 200
        else:
            return "Usu rio n o encontrado", 404

# Rota para login de usu rio
@app.route("/login", methods=["POST"])
def process_login():
    global users
    if request.method == "POST":
        user_name = request.json["user_name"]
        user_password = request.json["user_password"]
        user = next((u for u in users if u["user_name"] == user_name), None)

        if user["user_password"] == user_password:
            return json.dumps({"login": "true"}), 200
        else:
            return json.dumps({"login": "false"}), 200
```

Em seguida, foram feitos as requisições teste para validar o funcionamento do servidor. Todas as requisições mostradas através dos pintscreens a seguir foram feitas sucessivamente.

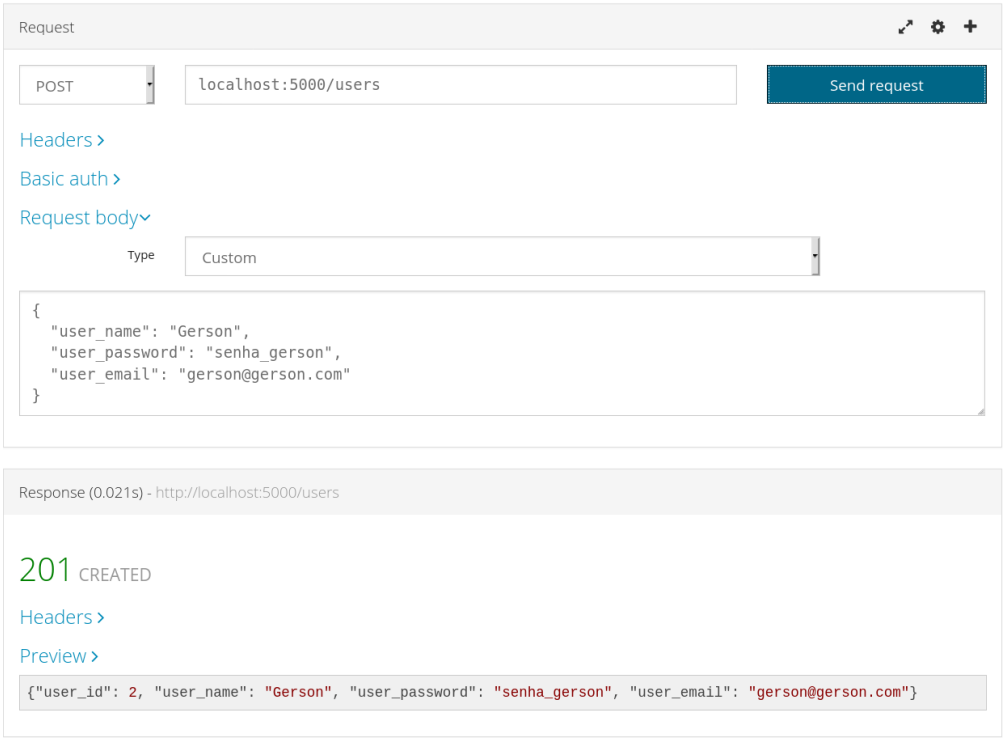


Figura 1: POST /users

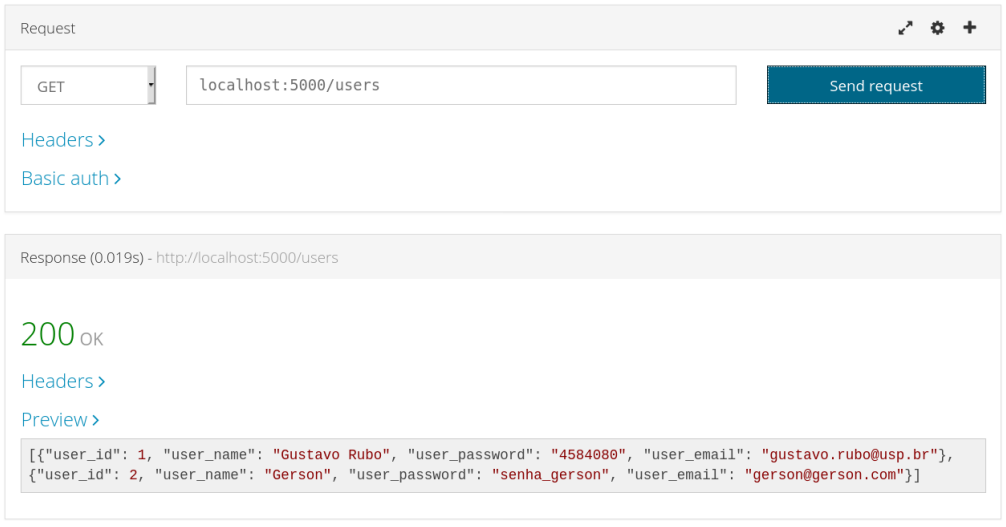


Figura 2: GET /users

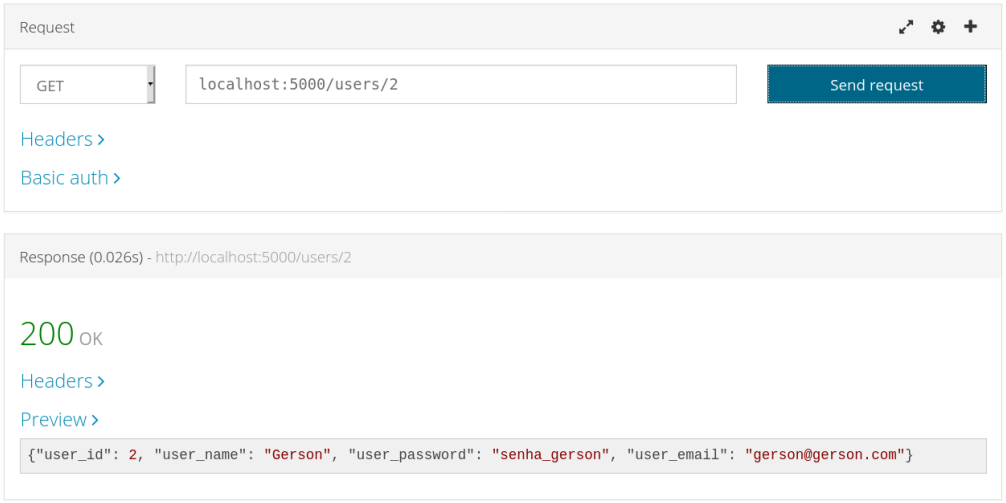


Figura 3: GET /users/2

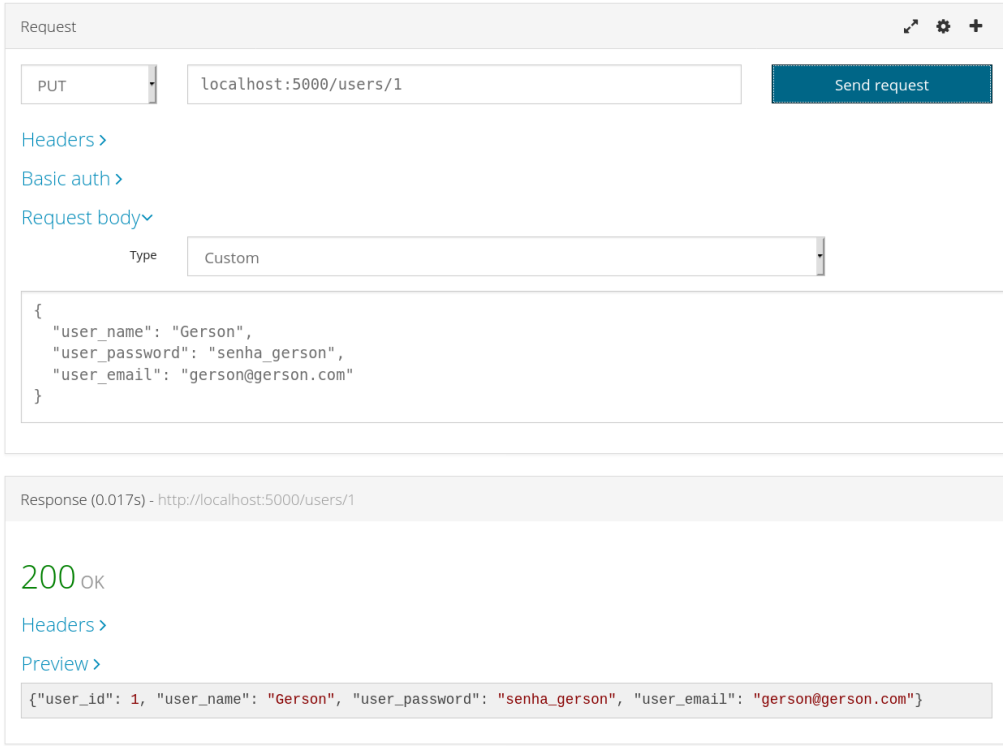


Figura 4: PUT /users/1

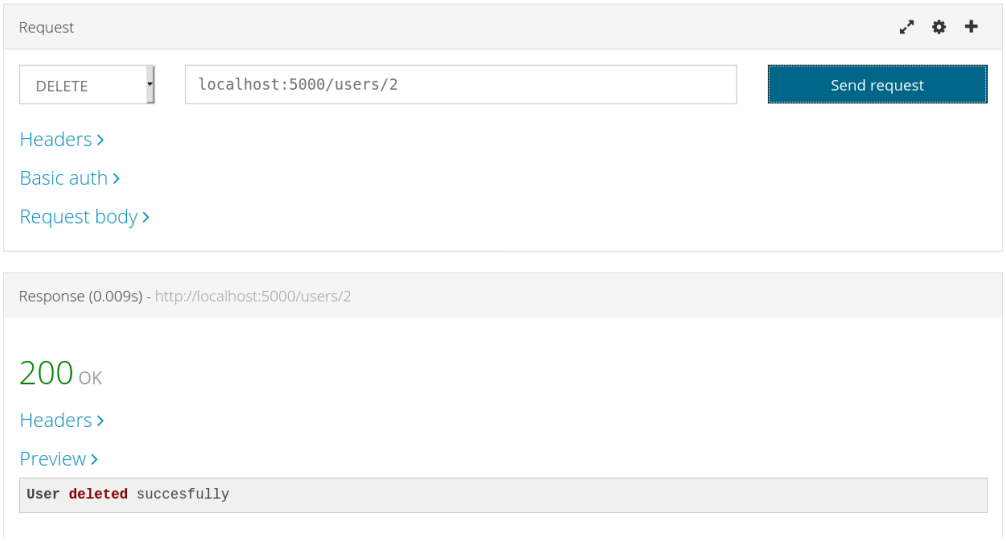


Figura 5: DELETE /users/2

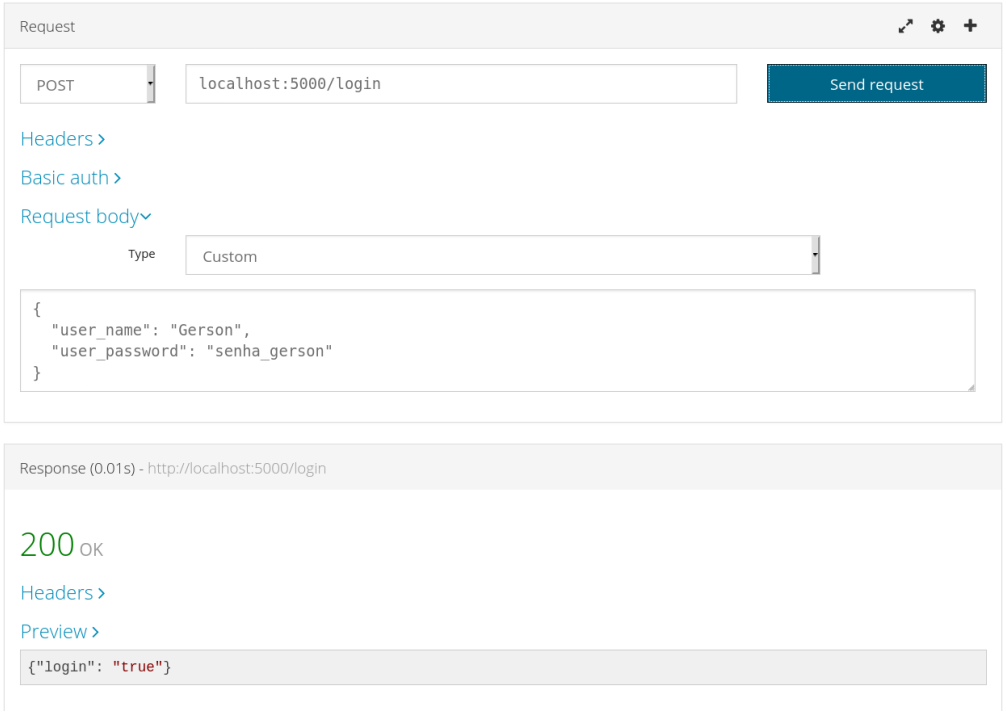


Figura 6: POST /login (com senha correta)

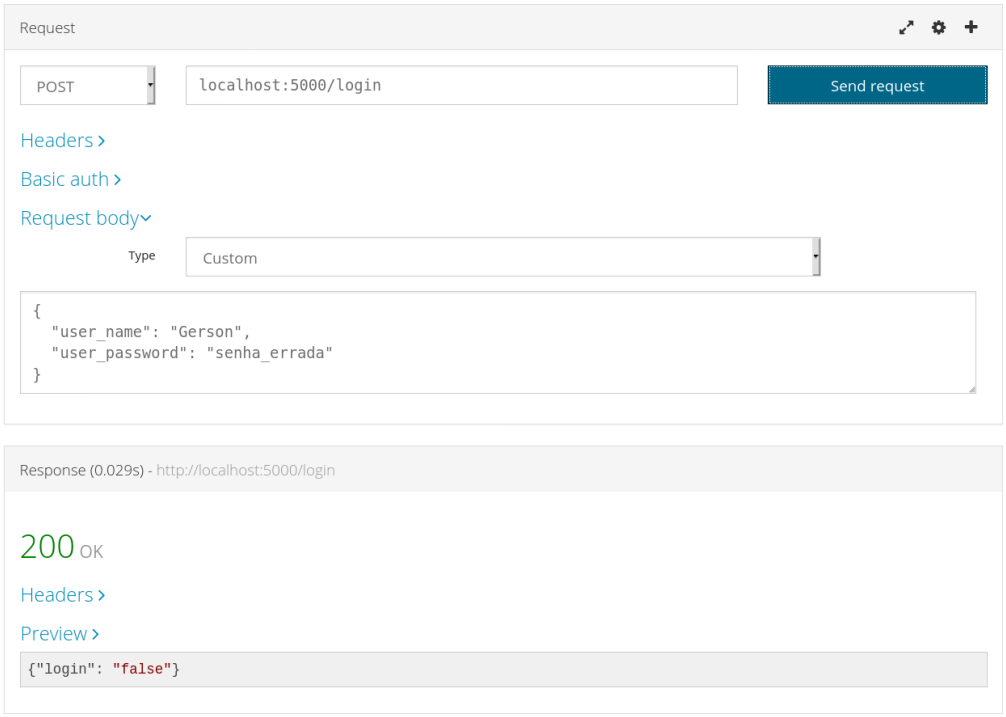


Figura 7: POST /login (com senha incorreta)

## 1.2 Hashing de senhas

Foi feito um servidor que gerencia um banco de dados de usuários como o anterior, porém agora as senhas não são armazenadas em sua forma pura, são criptografadas usando a biblioteca cryptography do Python.

A senha "hardcoded"do usuário inicial é criptografada. As senhas de usuários que são adicionados com o POST e modificados com o PUT também são criptografadas. Para todos esses casos, usamos o mesmo sal, que é definido na inicialização do servidor.

As senhas armazenadas tem o tipo de variável "bytes", e portanto não são "JSON serializable". Por isso, as respostas do servidor não são mais convertidas em JSON. É possível escrever uma função de serialização que contorna esse problema, mas não vale o esforço.

A seguir, mostramos a listagem do código.

```
from flask import Flask
from flask import request
import json

app = Flask(__name__)

# Inicializando a lista de usu rios
users = [
    {
        "user_id": 1,
        "user_name": "Gustavo Rubo",
        "user_password": "4584080",
        "user_email": "gustavo.rubo@usp.br"
    }
]

id_counter = 2

# Rota para cole  o de usu rios
@app.route("/users", methods=["POST", "GET", "PUT", "DELETE"])
def process_users():
    global id_counter, users
```

```
    if request.method == "POST":
        # Criar um usu rio e adicion -lo    lista
        user = {
            "user_id": id_counter,
            "user_name": request.json["user_name"],
            "user_password": request.json["user_password"],
            "user_email": request.json["user_email"]
        }
        id_counter += 1
        users.append(user)

        return json.dumps(user), 201

    elif request.method == "GET":
        # Retornar todos os usu rios
        return json.dumps(users), 200

    elif request.method == "PUT":
        return "Method not allowed", 405
    elif request.method == "DELETE":
        return "Method not allowed", 405

# Rota espec fica para usu rios
@app.route("/users/<user_id>", methods=["POST", "GET", "PUT", "DELETE"])
def process_users_id(user_id):
    global id_counter, users
    if request.method == "POST":
        return "Method not allowed", 405

    elif request.method == "GET":
        # Retornar o usu rio com o id especificado
        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            return json.dumps(user), 200
        else:
            return "User not found", 404

    elif request.method == "PUT":
        # Modificar o usu rio com o id especificado

        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            user["user_name"] = request.json["user_name"]
            user["user_password"] = request.json["user_password"]
            user["user_email"] = request.json["user_email"]
            return json.dumps(user), 200
        else:
            return "User not found", 404

    elif request.method == "DELETE":
        # Remover o usu rio com o id especificado
        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            users[:] = [u for u in users if u["user_id"] != int(user_id)]
            return "User deleted succesfully", 200
        else:
            return "Usu rio  n  o  encontrado", 404

# Rota para login de usu rio
@app.route("/login", methods=["POST"])
def process_login():
    global users
    if request.method == "POST":
        user_name = request.json["user_name"]
        user_password = request.json["user_password"]
```

```
91          user = next((u for u in users if u["user_name"] == user_name), None)
92
93          if user["user_password"] == user_password:
94              return json.dumps({"login": "true"}), 200
95          else:
96              return json.dumps({"login": "false"}), 200
```

Os prints a seguir mostram a criação e login de um usuário com nome "Gerson"e senha 123456.

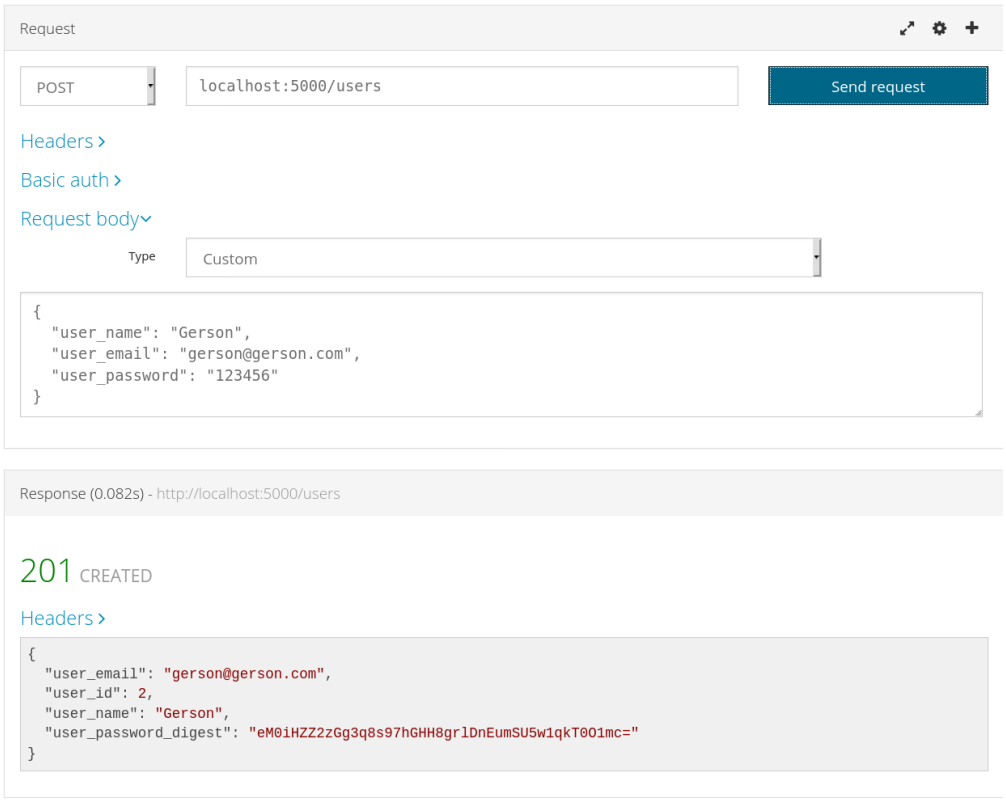


Figura 8: Criação do usuário.

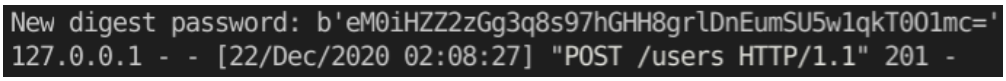


Figura 9: O resultado do digest da senha "123456".

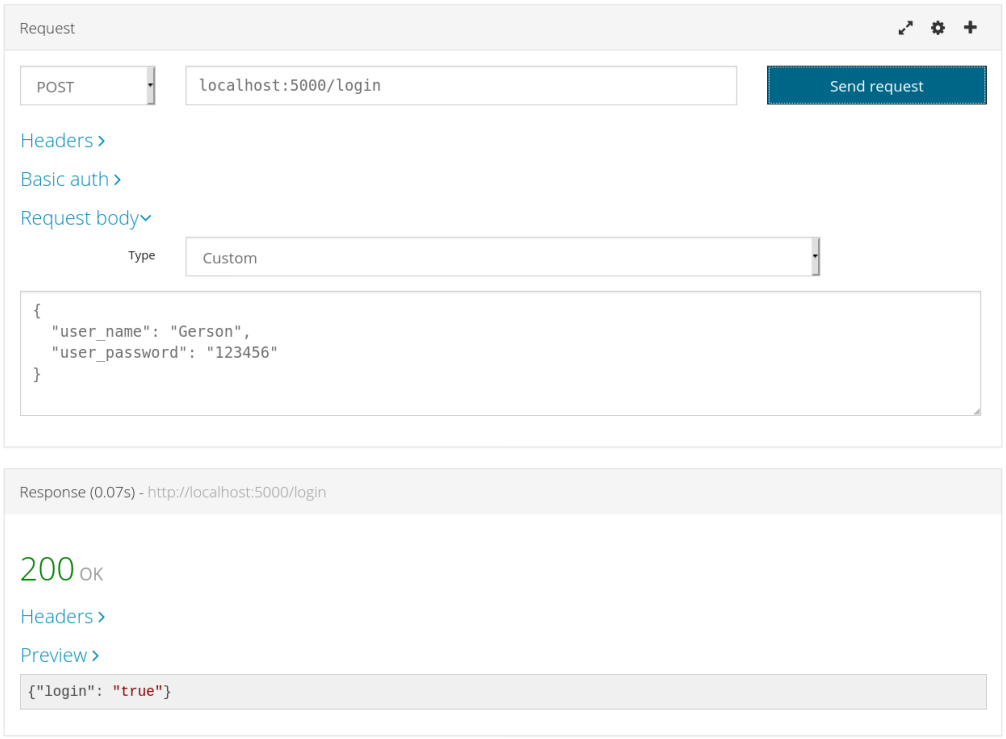


Figura 10: Login efetuado com sucesso.

Como hashes são operações que sempre tem a mesma saída para cada entrada, pode-se criar um banco de dados para senhas comuns. Então, caso sua senha seja uma senha comum (como *password*), ela é vulnerável a um **ataque de dicionário**.

O site Have I been Pwned? tem um registro de mais de 10 bilhões de contas comprometidas. Dessas contas, mais de 3,8 milhões usaram a senha *password*, e 24 milhões usaram a senha *123456*.

Podemos usar o site MD5 Reverse para obter uma senha comum dessas a partir do hash vazado.

## 1.3 Login com Cookie

Para essa parte do exercício, fazemos com que o servidor armazene um cookie de usuário no navegador do cliente, assim fazendo com que o login do usuário seja lembrado e mantido através de várias páginas.

O RESTED Client não tem a funcionalidade de edição de cookies, então a partir de agora o Postman será usado.

No código, a rota de login foi editada, e duas rotas (de boas vindas e de logout) foram criadas. Apenas essas três rotas serão mostradas a seguir.

```
# Rota para login de usu rio
@app.route("/login", methods=["POST", "GET"])
def process_login():
    global users
    if request.method == "POST":

        user_name = request.json["user_name"]
        user_password = request.json["user_password"]
        user = next((u for u in users if u["user_name"] == user_name), None)
        kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1, backend=default_backend())

        try:
            # Caso o nome de usu rio e senha existam no banco de dados
            kdf.verify(str.encode(user_password), base64.b64decode(user["
    user_password_digest"]))

```

```
            resp = make_response(f'Signing in as {request.json["user_name"]}')
            resp.set_cookie("user_name", request.json["user_name"])
            return resp

        except cryptography.exceptions.InvalidKey:
            # Caso de senha inv lida
            return json.dumps({"message": "Invalid password", "login": "false"}), 200
        except:
            # Outros casos (ex: usu rio inexistente)
            return json.dumps({"message": "Invalid user/password", "login": "false"}), 200

    elif request.method == "GET":
        user_name =request.cookies.get("user_name")

        if (user_name):
            return f'Logged in as {user_name}'
        else:
            return f'Not logged in yet'

# Rota para p gina de boas vindas. Usu rio deve estar logado para acessar.
@app.route("/welcome", methods=["GET"])
def process_welcome():

    if request.method == "GET":
        user_name = request.cookies.get("user_name")

        if (user_name):
            return f'Welcome {user_name}'
        else:
            return f'You need to log in to access this page.'

# Rota para logout. Apaga os cookies de usu rio.
@app.route("/logout", methods=["DELETE"])
def process_logout():

    if request.method == "DELETE":
        resp = make_response("Cookie removed")
        resp.set_cookie("user_name", "", max_age=0)
        return resp
```

Em seguida, verificamos o funcionamento do servidor fazendo um login, conferindo nossa página de boas vindas, fazendo o logout, e conferindo nossa página de boas vindas novamente.

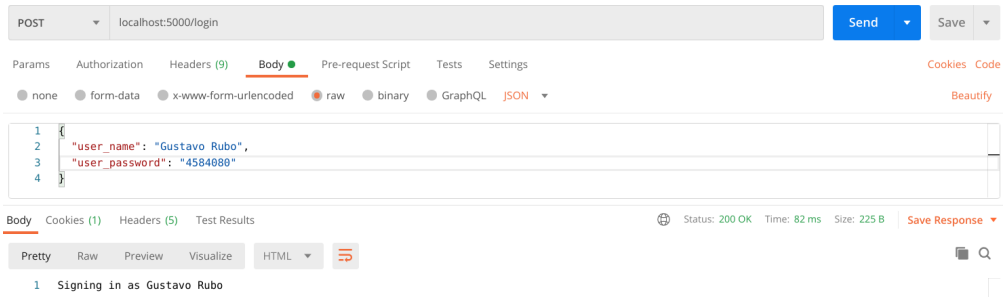


Figura 11: Login efetuado com sucesso.

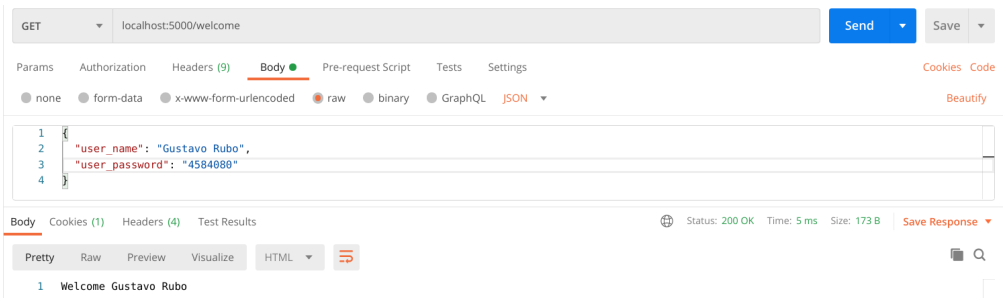


Figura 12: Página de boas vindas para usuário logado.

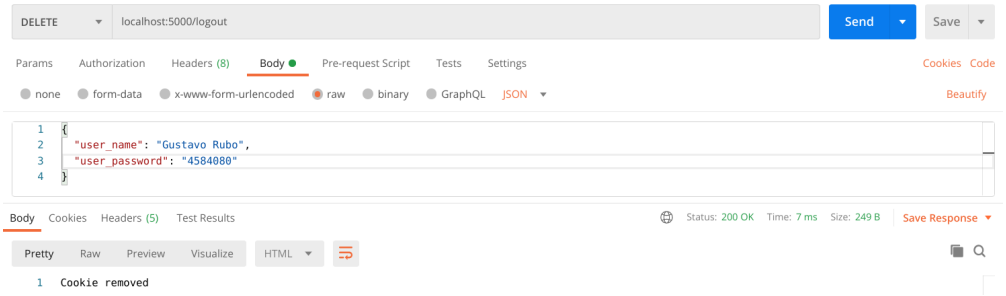


Figura 13: Logout efetuado (Cookies de usuário deletados)

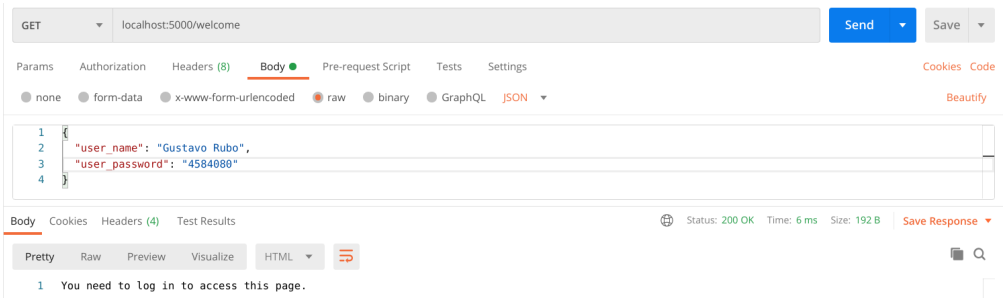


Figura 14: Página de boas vindas para usuário não logado

Os cookies que estamos armazenando por enquanto não são criptografados. Por isso, o sistema é vulnerável a uma alteração manual de cookies.

Com um usuário logado, podemos conferir no Postman qual é o formato de cookies de usuário (Figura [15]). Então, com o postman podemos criar um cookie como ele com o nome do usuário que desejarmos (Figuras 16, 17).


Figura 15: Podemos ver o formato dos cookies de um usuário logado.


Figura 16: Adicionando cookies para o domínio localhost.


Figura 17: Criando um cookie malicioso.

Com o ataque mostrado acima, pudemos acessar a página de boas vindas de um usuário sabendo apenas o nome de usuário dele.

## 1.4 Login com Cookie criptografado

Para finalizar o exercício 1, fizemos o servidor enviar cookies criptografados para o cliente. A criptografia é simétrica, e apenas o servidor pode criptografar e descriptografar os cookies. A mensagem criptografada é a junção de uma string *counter*, gerada aleatóriamente e hardcoded no código, com o id do usuário.

Os valores de *key*, *nonce* e *counter* foram gerados com python, como mostrado a seguir.

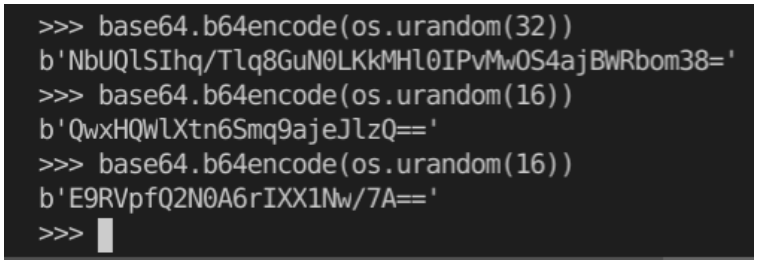


Figura 18: Gerando valores de key, nonce e counter.

Segue a listagem completa do código final. A classe *EncryptionManager* foi feita de acordo com o exemplo do enunciado.

```
import os
import base64
import cryptography
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

from flask import Flask
from flask import request
from flask import make_response
import json

counter = 'E9RVpfQ2N0A6rIXX1Nw/7A=='

class EncryptionManager:
    def __init__(self):
        key = base64.b64decode(b'NbUQlSIhq/Tlq8GuN0LKkMHl0IPvMwOS4ajBWRbom38=')
        nonce = base64.b64decode(b'QwxHQWlXtn6Smq9ajeJlzQ==')
        aes_context = Cipher(algorithms.AES(key), modes.CTR(nonce), backend=default_backend
    ())
        self.encryptor = aes_context.encryptor()
        self.decryptor = aes_context.decryptor()

    def updateEncryptor(self, plaintext):
        return self.encryptor.update(plaintext)

    def finalizeEncryptor(self):
        return self.encryptor.finalize()

    def updateDecryptor(self, ciphertext):
        return self.decryptor.update(ciphertext)

    def finalizeDecryptor(self):
        return self.decryptor.finalize()


# Definir o sal. Enquanto o servidor rodar, ele s   ir   usar este sal.
salt = os.urandom(16)
kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1, backend=default_backend())

app = Flask(__name__)

# Inicializando a lista de usu rios
users = [
    {
        "user_id": 1,
        "user_name": "Gustavo Rubo",
        "user_password_digest": base64.b64encode(kdf.derive(b"4584080")),
        "user_email": "gustavo.rubo@usp.br"
    }
]

```

```python
id_counter = 2

# Rota para cole  o de usu rios
@app.route("/users", methods=["POST", "GET", "PUT", "DELETE"])
def process_users():
    global id_counter, users
    if request.method == "POST":
        # Criar um usu rio e adicion -lo    lista

        kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1, backend=default_backend())
        user = {
            "user_id": id_counter,
            "user_name": request.json["user_name"],
            "user_password_digest": base64.b64encode(kdf.derive(str.encode(request.json["
    user_password"]))),
            "user_email": request.json["user_email"]
        }
        id_counter += 1
        users.append(user)

        print("New digest password:", user["user_password_digest"])

        return user, 201

    elif request.method == "GET":
        # Retornar todos os usu rios
        return {"users": users}, 200

    elif request.method == "PUT":
        return "Method not allowed", 405
    elif request.method == "DELETE":
        return "Method not allowed", 405

# Rota espec fica para usu rios
@app.route("/users/<user_id>", methods=["POST", "GET", "PUT", "DELETE"])
def process_users_id(user_id):
    global id_counter, users
    if request.method == "POST":
        return "Method not allowed", 405

    elif request.method == "GET":
        # Retornar o usu rio com o id especificado

        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            return user, 200
        else:
            return "User not found", 404

    elif request.method == "PUT":
        # Modificar o usu rio com o id especificado

        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            user["user_name"] = request.json["user_name"]
            user["user_password_digest"] = base64.b64encode(kdf.derive(request.json["
    user_password"]))
            user["user_email"] = request.json["user_email"]

            print("New digest password:", user["user_password_digest"])

            return user, 200
        else:
            return "User not found", 404

    elif request.method == "DELETE":
```

```
        # Remover o usu rio com o id especificado

        user = next((u for u in users if u["user_id"] == int(user_id)), None)

        if user:
            users[:] = [u for u in users if u["user_id"] != int(user_id)]
            return "User deleted succesfully", 200
        else:
            return "User not found", 404

# Rota para login de usu rio
@app.route("/login", methods=["POST", "GET"])
def process_login():
    global users
    if request.method == "POST":

        user_name = request.json["user_name"]
        user_password = request.json["user_password"]
        user = next((u for u in users if u["user_name"] == user_name), None)
        kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1, backend=default_backend())
        enc_manager = EncryptionManager()

        try:
            # Caso o nome de usu rio e senha existam no banco de dados
            kdf.verify(str.encode(user_password), base64.b64decode(user["
    user_password_digest"]))

            plaintext = counter+str(user["user_id"])
            ciphertext = enc_manager.updateEncryptor(bytes(str.encode(plaintext)))
            enc_manager.finalizeEncryptor()

            print("Cookie encriptado:", base64.b64encode(ciphertext))

            resp = make_response(f'Signing in as {request.json["user_name"]}')
            resp.set_cookie("user_cookie", base64.b64encode(ciphertext))
            return resp

        except cryptography.exceptions.InvalidKey:
            # Caso de senha inv lida
            return json.dumps({"message": "Invalid password", "login": "false"}), 200
        except:
            # Outros casos (ex: usu rio inexistente)
            return json.dumps({"message": "Invalid user/password", "login": "false"}), 200

    elif request.method == "GET":
        user_name =request.cookies.get("user_name")

        if (user_name):
            return f'Logged in as {user_name}'
        else:
            return f'Not logged in yet'

# Rota para p gina de boas vindas. Usu rio deve estar logado para acessar.
@app.route("/welcome", methods=["GET"])
def process_welcome():

    if request.method == "GET":
        if(request.cookies.get("user_cookie")):

            enc_manager = EncryptionManager()
            ciphertext = base64.b64decode(request.cookies.get("user_cookie"))
            plaintext = str(enc_manager.updateDecryptor(ciphertext))
            enc_manager.finalizeDecryptor()

            print("Cookie encriptado:", base64.b64encode(ciphertext))

            user_id = int(plaintext[26:][:-1])
            user = next((u for u in users if u["user_id"] == user_id), None)
```

```
185
186             return f'Welcome {user["user_name"]}'
187         else:
188             return f'You need to log in to access this page.'
189
190 # Rota para logout. Apaga os cookies de usu rio.
191 @app.route("/logout", methods=["DELETE"])
192 def process_logout():
193
194     if request.method == "DELETE":
195         resp = make_response("Cookie removed")
196         resp.set_cookie("user_cookie", "", max_age=0)
197         return resp
```

Os cookie são encriptados no servidor quando o usuário faz o login, e são decriptados também no servidor quando o usuário entra na página welcome. A imagem a seguir mostra o cookie encriptado no momento do login e do welcome, respectivamente.

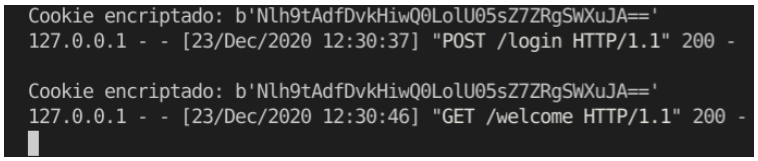


Figura 19: Cookie encriptado.

E enfim, para demonstrar o funcionamento correto do servidor, fazemos uma sequência de login, página de welcome, logout e página de welcome

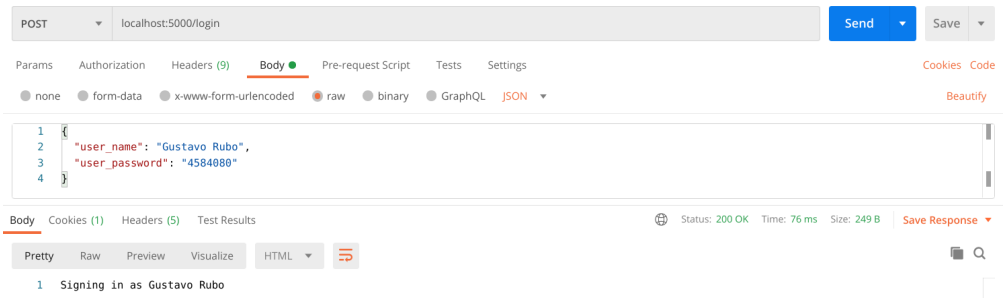


Figura 20: Login efetuado com sucesso.

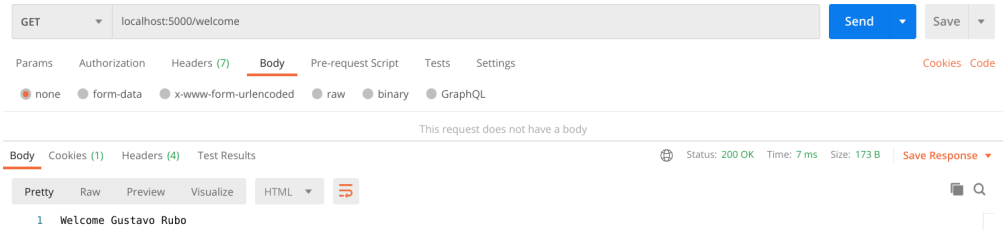


Figura 21: Página de boas vindas para usuário logado.

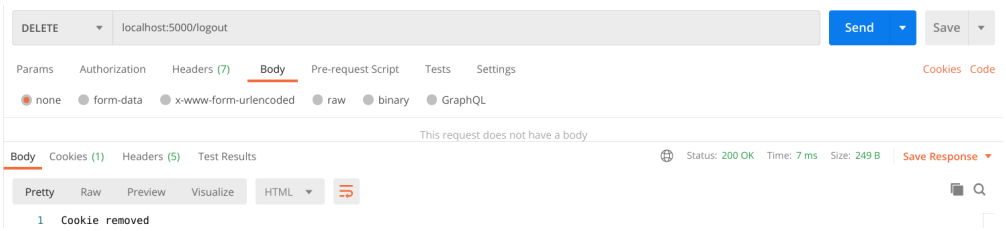


Figura 22: Logout efetuado (Cookies de usuário deletados)

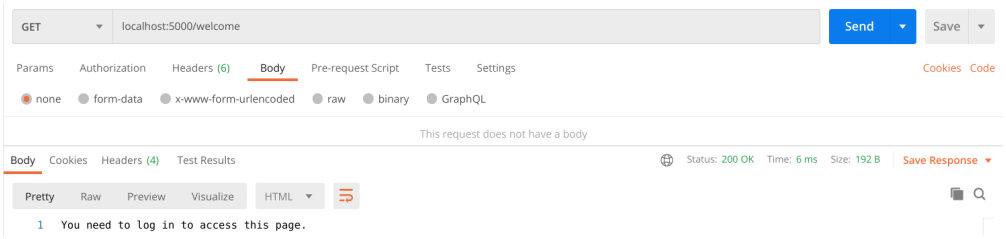


Figura 23: Página de boas vindas para usuário não logado

Caso não concatenarmos o *counter* aos ids, temos um resultado de textos criptografados como o seguinte:

```
Plaintext: b'1' Cipher (hex): 42
Plaintext: b'2' Cipher (hex): 53
Plaintext: b'3' Cipher (hex): 1c
Plaintext: b'4' Cipher (hex): d6
Plaintext: b'5' Cipher (hex): 42
```

Figura 24: Ids criptografados sem a concatenação do counter.

Do jeito que o counter está sendo utilizado neste programa (sendo concatenado ao plaintext), não encontrei que tipo de vulnerabilidade pode existir caso omitirmos ele. Inclusive, o livro texto não mostra uma string sendo contatenada a mensagens no diagrama da encriptação CTR.

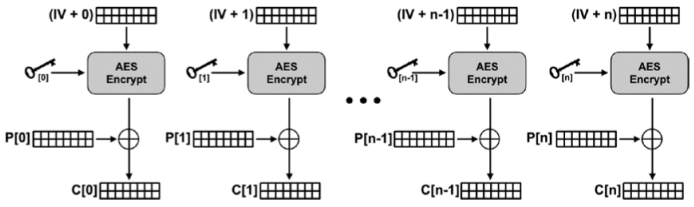


Figura 25: Esquema da encriptação CTR.

Caso um atacante conseguir interceptar a mensagem criptografada do id, sabendo qual é o id, ele pode conseguir informações sobre a chave utilizada para criptografá-lo, mas isso não deve ser um problema se a mesma chave nunca for repetida.

## 2 Exercício 2 - Aplicativo de troca de mensagens

Não feito.

## 3 Considerações sobre os exercícios

Ter os enunciados separados em requisitos funcionais e lista de coisas a apresentar no relatório é muito conveniente e fácil de seguir, não ficam dúvidas sobre o que temos que fazer.

Por ter uma quantidade de trabalho muito grande de programação, muitas vezes eu estava só fazendo o programa sem pensar na teoria do que eu estava fazendo, só colocando os exemplos no lugar certo e fazendo funcionar. Só as questões conceituais me fizeram revisar os slides e assistir pedaços das aulas de novo.

Além disso, confesso que é difícil ter motivação para completar todo o exercício tendo já passado da matéria com a nota dos anteriores, ainda mais com o saco cheio de final de ano. Tendo, dito isso, boas festas!