

Minicurso UFSJ - SECOMP 2025



Professor: Bernardo de Castro Monteiro Franco Gomes

Minicurso de Desenvolvimento Prático de Sistemas com JavaScript e Flask

Resumo

O minicurso irá ser focado no desenvolvimento full-stack de uma página de Login e Registro. O sistema irá funcionar a partir de um backend feito em Flask com um sistema CRUD com o banco de dados SQLite e uma parte front-end, onde será feita por um código básico de HTML e CSS para a parte do "corpo" do site e o processamento dos dados vai ser desenvolvido a partir de um código em JavaScript.

Temáticas a serem abordadas

As principais temáticas a serem abordadas neste minicurso são:

- Desenvolvimento ágil de projetos;
- Capacitação full-stack de desenvolvedores;
- Tratamento de dados em relação a cibersegurança;
- Desenvolvimento de um Sistema CRUD (Create, Read, Update e Delete);

Estas temáticas estão diretamente relacionadas ao dia a dia de um desenvolvedor, independente de sua stack. O desenvolvimento ágil de projetos atualmente é algo muito requisitado pelas empresas, a capacitação full-stack hoje é algo extremamente necessário pela volatilidade dos projetos atuais e o tratamento de dados utilizados tanto pelo front-end quanto back-end, que é algo que está em foco nas discussões atuais.

Descrição

Este minicurso prático é ideal para quem deseja aprender a construir uma aplicação web completa, desde o backend até o frontend, utilizando tecnologias modernas e acessíveis. Você irá desenvolver um sistema de Login e Registro com funcionalidades básicas de CRUD, usando FlaskAPI com SQLite no backend para gerenciar dados, e HTML, CSS, Bootstrap e JavaScript no frontend para criar uma interface interativa e responsiva.

Durante o curso, você será guiado passo a passo no desenvolvimento ágil de projetos, compreendendo a importância da segurança no tratamento de dados e aplicando boas práticas essenciais para um desenvolvedor.

Este curso é perfeito para quem quer uma introdução objetiva e aplicada ao desenvolvimento web full-stack, além de servir como uma base sólida para quem deseja aprofundar-se no tema.

Conhecimentos prévios necessários [Recomendados]

Os conhecimentos prévios necessários para um bom entendimento do curso são:

- Noção básica em lógica da programação;
- Noção básica do desenvolvimento de sites utilizando HTML e CSS;
- Noção básica do uso de python;
- Noção básica de uso do VSCode;

Objetivos de aprendizagem

O objetivo de aprendizagem é que os alunos que participarem sejam capazes de criarem as suas próprias aplicações utilizando as funcionalidades básicas do CRUD com FlaskAPI e SQLite, e HTML, CSS, Bootstrap e JavaScript, além de criarem uma melhor lógica de como desenvolverem sistemas web.

Metodologia

O minicurso irá ser feito a partir de uma metodologia com foco no aprendizado prático dos alunos e participantes do minicurso.

Material de apoio e recursos

Os materiais de apoio que vão ser utilizados para este minicurso são:

- Este documento;
- Apresentação de slides;
- O seguinte repositório do github:

<https://github.com/Se7enzito/Minicurso-UFSJ---SECOMP-2025>

Se algum participante deste minicurso tiver dúvidas ou desejar conversar sobre qualquer tema relacionado ao conteúdo apresentado, estou à disposição para ajudar.

Você pode entrar em contato comigo pelos seguintes canais:

- Telefone: (31) 99998-1418
- E-mail: bernardocmfgomes@gmail.com

Fique à vontade para me procurar!

Estrutura do conteúdo

Estrutura 2 horas - Tempo aproximado por tópico

1. Introdução (5 minutos)

- Apresentação sobre mim;
- Problema: Desenvolvimento de software atualmente precisa ser algo ágil;
- Stack proposta: Flask + SQLite + HTML/CSS + Javascript + Bootstrap;

2. Sistema CRUD (10 minutos)
 - Como funciona um CRUD?;
 - Como criar um CRUD;
3. FlaskAPI (15 minutos)
 - O que é FlaskAPI;
 - Como utilizar FlaskAPI;
 - Importância da FlaskAPI;
4. Tratamento de dados com FlaskAPI (7 minutos)
 - Como funciona o tratamento de dados com FlaskAPI?;
 - Como funciona a Flask Session?;
5. HTML, CSS e Bootstrap (5 minutos)
 - Básico de sites HTML e CSS;
 - Como utilizar bootstrap;
6. JavaScript (10 minutos)
 - Uso de APIs no JavaScript;
 - Eventos DOM;
 - Como utilizar: Async e Await | Try e Catch;
7. Conhecimentos básicos sobre uma API (10 minutos)
 - Como funciona uma API?;
 - Como criar uma API utilizando FlaskAPI;
 - Como enviar dados em uma porta da API;
8. Intervalo (15 minutos)
 - Intervalo para os participantes do minicurso conversarem entre si, irem no banheiro e tomar água sem perder o conteúdo do minicurso;
9. Dúvidas (13 minutos)
 - Momento para tirar dúvidas dos participantes do minicurso;
10. Caso prático (30 minutos)
 - Os participantes irão implementar o sistema por conta própria, acompanhados pelo palestrante, que estará disponível para esclarecer dúvidas individualmente durante o processo (15 minutos);
 - Em seguida, o palestrante irá reproduzir o desenvolvimento do mesmo sistema passo a passo, reforçando os conceitos e respondendo às questões levantadas durante a prática (15 minutos);

Estrutura 4 horas - Tempo aproximado por tópico

1. Introdução (5 minutos)
 - Apresentação sobre mim;
 - Problema: Desenvolvimento de software atualmente precisa ser algo ágil;
 - Stack proposta: Flask + SQLite + HTML/CSS + Javascript + Bootstrap;

2. Sistema CRUD (10 minutos)
 - Como funciona um CRUD?;
 - Como criar um CRUD;
3. FlaskAPI (15 minutos)
 - O que é FlaskAPI;
 - Como utilizar FlaskAPI;
 - Importância da FlaskAPI;
4. Tratamento de dados com FlaskAPI (7 minutos)
 - Como funciona o tratamento de dados com FlaskAPI?;
 - Como funciona a Flask Session?;
5. HTML, CSS e Bootstrap (5 minutos)
 - Básico de sites HTML e CSS;
 - Como utilizar bootstrap;
6. JavaScript (10 minutos)
 - Uso de APIs no JavaScript;
 - Eventos DOM;
 - Como utilizar: Async e Await | Try e Catch;
7. Conhecimentos básicos sobre uma API (15 minutos)
 - Como funciona uma API?;
 - Como criar uma API utilizando FlaskAPI;
 - Como enviar dados em uma porta da API;
8. Intervalo (15 minutos)
 - Intervalo para os participantes do minicurso conversarem entre si, irem no banheiro e tomar água sem perder o conteúdo do minicurso;
9. Dúvidas (15 minutos)
 - Momento para tirar dúvidas dos participantes do minicurso;
10. Caso prático (60 minutos)
 - Os participantes irão implementar o sistema por conta própria, acompanhados pelo palestrante, que estará disponível para esclarecer dúvidas individualmente durante o processo (30 minutos);
 - Em seguida, o palestrante irá reproduzir o desenvolvimento do mesmo sistema passo a passo, reforçando os conceitos e respondendo às questões levantadas durante a prática (30 minutos);
11. Intervalo (15 minutos)
 - Intervalo para os participantes do minicurso conversarem entre si, irem no banheiro e tomar água sem perder o conteúdo do minicurso;
12. Introdução temas avançados (10 minutos)
 - Explicar os temas avançados que iremos abordar;
13. Aprofundamento SQLite (10 minutos)

- Melhor explicação sobre SQLite;
- Explicação sobre outros modelos de banco de dados;

14. Aprofundamento FlaskAPI e desenvolvimento de APIs (10 minutos)

- Como funciona em si uma API com Flask;
- Dicas para criar uma API com Flask;
- Dicas para desenvolvimento de APIs;

15. Tratamento de dados avançado (28 minutos)

- Criptografia de dados;
- Modelos hash de dados + Criptografia dos dados;
- Exemplo real de como funciona o modelo hash junto da criptografia;

16. Dúvidas (10 minutos)

- Momento para tirar dúvidas dos participantes do minicurso;

Conteúdo Minicurso

1 - Sistema CRUD

1.1 - Como funciona um CRUD?

CRUD, um acrônimo para Create (Criar), Read (Ler), Update (Atualizar) e Delete (Deletar), descreve as quatro operações básicas realizadas em dados armazenados em bancos de dados ou sistemas de informação. Essas operações são fundamentais para a interação do usuário com os dados, permitindo a criação de novos registros, a visualização e busca de informações, a modificação de dados existentes e a remoção de registros.



Create

Permite adicionar novos dados a um sistema, como a criação de um novo usuário em um site ou a adição de um novo produto a um catálogo online.

Update

Permite modificar dados existentes. Por exemplo, alterar o nome de usuário, atualizar o preço de um produto ou editar informações de contato.

Read

Permite a visualização e busca de dados existentes. Por exemplo, ver o perfil de um usuário, pesquisar produtos em um e-commerce ou ler notícias em um portal.

Delete

Permite remover dados de um sistema. Por exemplo, excluir uma conta de usuário, remover

um produto de um catálogo ou apagar um post de um fórum.

1.2 - Como criar um CRUD?

Para criar um CRUD é preciso seguir alguns passos que vão ser nossos primeiros passos cuidando de um banco de dados:

1. Escolher uma tecnologia;
 - a. No nosso caso vai ser o Python + SQLite;
2. Crie o banco de dados;
3. Conecte-se ao banco de dados;
4. Implemente as operações CRUD;
5. Crie uma interface;

Aqui vou colocar alguns exemplos básicos de códigos que podem ser utilizados em um CRUD no SQLite:

- Criar uma tabela no banco de dados:

!! No SQLite não é preciso criar um banco de dados em si, somente as tabelas, ao arquivo do banco de dados ser criado é somente adicionar as tabelas!!

```
CREATE TABLE IF NOT EXISTS tabelinha (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    user TEXT NOT NULL,  
    email TEXT NOT NULL  
);
```

- Adicionar algo à essa tabela:

```
INSERT INTO usuarios (nome, email) VALUES ('João da Silva', 'joao@email.com');
```

- Ler algo dessa tabela:

```
SELECT * FROM usuarios;
```

- Atualizar algo dessa tabela:

```
UPDATE usuarios SET nome = 'João Silva' WHERE id = 5;
```

- Remover algo dessa tabela:

```
DELETE FROM usuarios WHERE id = 5;
```

2 - FlaskAPI

2.1 - O que é FlaskAPI

FlaskAPI é uma extensão ou abordagem para o framework Flask (Python) que facilita o desenvolvimento de APIs RESTful. O Flask é minimalista, simples de configurar e bastante flexível, sendo ideal para criar aplicações web ou APIs de forma rápida.



APIs RESTful são usadas para fornecer acesso a recursos (dados, serviços, etc.) através de URLs e métodos HTTP como GET, POST, PUT e DELETE. Elas são projetadas para serem leves, escaláveis e fáceis de usar, tornando-as populares para o desenvolvimento de aplicações web e móveis.



FlaskAPI permite criar endpoints para que outros sistemas ou o frontend possam consumir os dados do backend.

2.2 - Como utilizar FlaskAPI

- Instalação com pip:

```
pip install Flask
```

- Exemplo básico:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api')
def hello_api():
    return jsonify({'mensagem': 'Olá, API!'})

if __name__ == '__main__':
    app.run(debug=True)
```



Esse código cria um endpoint `/api` que retorna um JSON.

2.3 - Importância do FlaskAPI

- Permite integrar aplicações diferentes;
- Torna os dados acessíveis via HTTP;
- Facilita a construção de sistemas modernos (front-end separado do back-end);
- É simples e rápido para prototipar;

3 - Tratamento de Dados com FlaskAPI

3.1 - Como funciona o tratamento de dados com FlaskAPI?

FlaskAPI permite:

- Receber dados via requisições HTTP (GET, POST, PUT, DELETE);
- Validar e tratar os dados recebidos;
- Retornar respostas estruturadas (JSON);

Exemplo recebendo dados:

```
from flask import request

@app.route('/api/somar', methods=['POST'])
def somar():
    dados = request.json
    resultado = dados['a'] + dados['b']
    return jsonify({'resultado': resultado})
```


3.2 - Como funciona a Flask Session?

Flask Session é usada para armazenar informações temporárias sobre o usuário (por exemplo, login) durante a navegação.

- Exemplo:

```
from flask import session

session['usuario'] = 'joao'
print(session['usuario']) # Exibe 'joao'
```

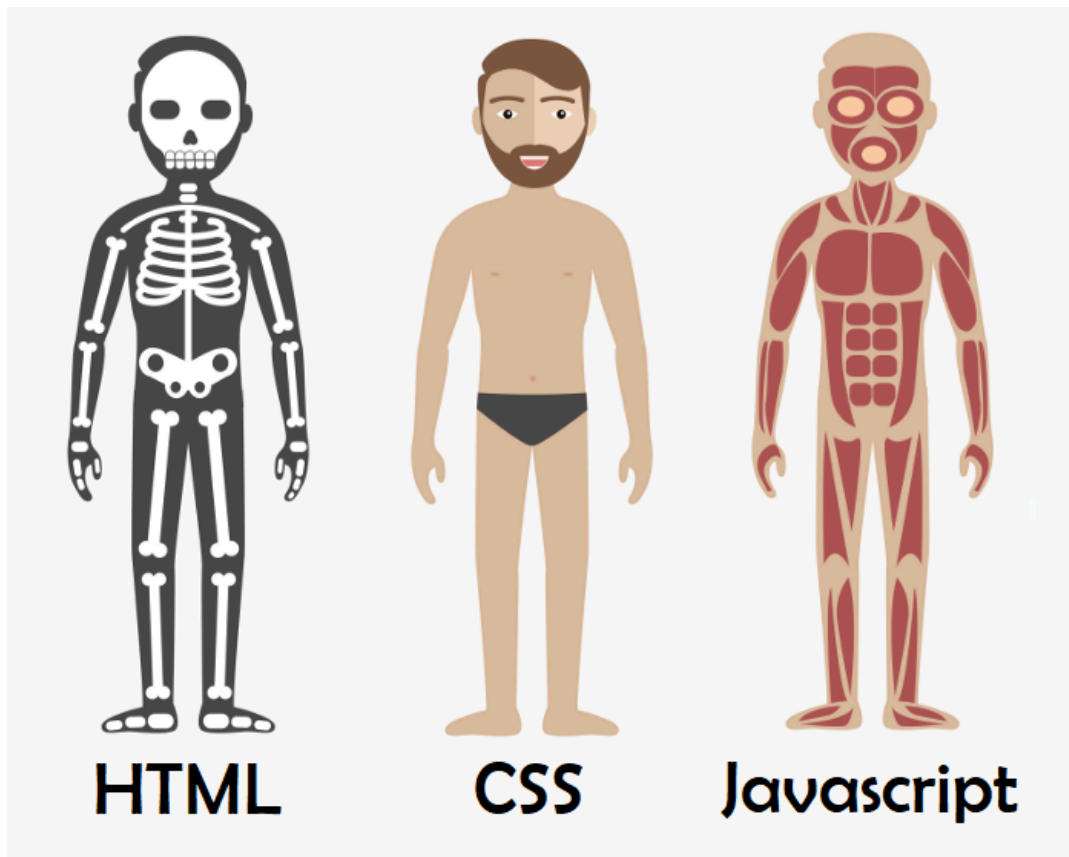
|  As sessões são seguras e permitem personalizar a experiência do usuário.

Em Flask, os dados da sessão são armazenados no lado do cliente, dentro de um cookie, que é assinado digitalmente e criptografado pelo servidor para garantir a integridade. Esse cookie é enviado com cada solicitação ao servidor, onde é decodificado e acessado pelo aplicativo Flask.

4 - HTML, CSS e Bootstrap

4.1 - Básico de sites HTML e CSS

Muitas pessoas colocam o HTML, CSS e JavaScript como se fosse o corpo humano, HTML sendo a estrutura (ossos), CSS a estilização (pele) e JavaScript sendo as funções (músculos e sistema nervoso):



HTML cria a estrutura do site:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
  initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Bem-vindo ao meu site!</h1>
  <p>Este é um parágrafo.</p>
</body>

</html>
```

CSS estiliza:

```
h1 {
  color: blue;
  text-align: center;
}
```

4.2 - Como utilizar Bootstrap

O Bootstrap é uma extensão do CSS e do JavaScript, com ela fica mais fácil de se criar a estilização de um site, ele já possui funções pré-definidas que podem ser utilizadas no código HTML.

Site do Bootstrap: <https://getbootstrap.com/>

Bootstrap facilita a criação de layouts responsivos:

- Instalação via CDN:

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">
```

- Exemplo de botão:

```
<button class="btn btn-primary">Clique aqui</button>
```

5 - JavaScript

5.1 - Uso de APIs no JavaScript

O uso de APIs no JavaScript é feito a partir das funções `fetch` e suas sub-funções, onde você utiliza `fetch('link')` e depois os sub-comandos do `fetch`, como `.json()` para pegar os dados.

```
1 fetch('https://api.exemplo.com/dados', {
2   method: 'GET', // ou 'POST', 'PUT', 'DELETE', etc.
3   headers: {
4     'Authorization': 'Bearer sua_token_de_autenticação',
5     'Content-Type': 'application/json' // dependendo da API
6   }
7 })
8 .then(response => response.json())
9 .then(data => {
10   // Faça algo com os dados da resposta da API
11 })
12 .catch(error => {
13   // Lida com erros
14 });
```

5.2 - Eventos DOM

DOM (Document Object Model) é uma representação da nossa árvore de elementos do HTML. Ela nos traz todo o modelo do nosso documento HTML, nos permitindo manipulá-lo de diversas formas.

Ela se assemelha muito a uma árvore genealógica, onde cada elemento tem seus respectivos pai e filhos. No entanto, cada elemento pode ter apenas um elemento pai, mas diversos elementos filhos.

Exemplo:

```
document.getElementById('botao').addEventListener('click', () => {
  alert('Botão clicado!');
});
```

5.3 - Como utilizar: Async e Await | Try e Catch

5.3.1 - Async e Await

Função Síncrona: Quando falamos ao telefone, as informações chegam e saem em sequência, uma após a outra; fazemos uma pergunta, recebemos logo em seguida a resposta, com os dados dessa resposta fazemos outro comentário, etc.

Função Assíncrona: uma conversa online via algum mensageiro, como o WhatsApp, enviamos uma mensagem e não ficamos olhando para a tela, esperando, até a outra pessoa responder. Afinal de contas, não temos como saber quando, e se, essa resposta vai chegar.

Simplifica código assíncrono:

```
async function buscarDados() {  
  const resposta = await fetch('/api');  
  const dados = await resposta.json();  
  console.log(dados);  
}
```

5.3.2 - Try e Catch

`try` e `catch` são usados para lidar com exceções ou erros que podem ocorrer durante a execução do seu código.

Em JavaScript, utilizamos `try` e `catch` porque eles permitem lidar com erros e exceções de maneira controlada, evitando que o código pare de funcionar abruptamente.

Tratamento de erros:

```
try {  
  const resposta = await fetch('/api');  
  const dados = await resposta.json();  
} catch (erro) {  
  console.error('Erro ao buscar dados:', erro);  
}
```

6 - Conhecimentos básicos sobre uma API

6.1 - Como funciona uma API?

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e protocolos que permite que diferentes sistemas de software se comuniquem e compartilhem informações entre si. É como um garçom em um restaurante, que recebe os pedidos dos clientes (aplicativos) e os entrega ao cozinheiro (servidor), que prepara a comida (dados) e a devolve ao garçom para que ele a entregue ao cliente.

6.2 - Como criar uma API utilizando FlaskAPI?

Para criar uma API com Flask, você precisará instalar o Flask, criar um projeto com a estrutura básica, definir endpoints para as requisições HTTP (GET, POST, etc.), e rodar a aplicação. O Flask permite facilmente manipular dados em JSON, o que é comum em APIs.

Exemplo:

```
@app.route('/api/usuarios')  
def listar_usuarios():
```

```
return jsonify([{'id': 1, 'nome': 'João'}])
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///mydatabase.db'
db = SQLAlchemy(app)

# Defina os modelos (tabelas)
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)

# Rota inicial
@app.route('/')
def index():
    # Exemplo de consulta ao banco de dados
    users = User.query.all()
    return f'Total de usuários: {len(users)}'
```

FLASK PYTHON

6.3 - Como enviar dados em uma porta da API?

Para enviar dados através de uma porta de API, é necessário fazer uma requisição HTTP para o endpoint desejado, utilizando o método HTTP apropriado (como POST, PUT, ou PATCH) e incluindo os dados no corpo da requisição. O formato dos dados geralmente é JSON, mas pode variar dependendo da API.

!! No caso do Flask sempre iremos enviar dados utilizando JSON.

7 - Aprofundamento SQLite

7.1 - O que é SQLite? Curiosidades e afins

Basicamente, um banco de dados é uma planilha, como a do excel, gerenciada por um software, entretanto, além do software, há outras diferenças:

- Como os dados são armazenados e manipulados;
- Quem pode acessar os dados;
- Quantos dados podem ser armazenados;
- As planilhas foram originalmente projetadas para serem usadas por um único usuário, e suas características refletem esse propósito. Elas são ótimas para um único usuário ou para um pequeno número de usuários que não precisam realizar manipulações de dados muito complicadas;
- Um banco de dados é uma coleção organizada de informações, ou dados, estruturados, geralmente armazenados eletronicamente em um sistema de computador. Normalmente, um banco de dados é controlado por um Sistema de Gerenciamento de Banco de Dados (SGBD);

7.2 - Outros modelos de banco de dados

- **MySQL / PostgreSQL:** robustos e escaláveis.
- **MongoDB:** orientado a documentos (NoSQL).
- **Redis:** banco em memória para alta performance.

8 - Aprofundamento FlaskAPI e desenvolvimento de APIs

8.1 - Como funciona em si uma API com Flask

Uma API criada com Flask funciona basicamente assim:

- O Flask cria um **servidor web local** que fica “escutando” as requisições HTTP que chegam em determinados **endpoints** (URLs).
- Cada endpoint está associado a uma função Python que é executada quando aquela rota é acessada.
- Essas funções podem:
 - Consultar bancos de dados;
 - Processar dados recebidos na requisição;
 - Retornar dados para quem fez a requisição, geralmente em formato JSON.

Exemplo resumido:

```
@app.route('/api/produtos', methods=['GET'])
def listar_produtos():
    produtos = [{'id': 1, 'nome': 'Camiseta'}]
    return jsonify(produtos)
```

Nesse exemplo:

- A função `listar_produtos` é chamada quando alguém acessa `/api/produtos` usando método GET.
- Ela devolve os dados (uma lista de produtos) em JSON para quem chamou.

8.2 - Dicas para criar uma API com Flask

- Defina claramente quais serão os **recursos** e **rotas** (ex.: `/api/usuarios`, `/api/produtos`).
- Sempre use os métodos HTTP adequados:
 - GET para buscar dados;
 - POST para criar;
 - PUT/PATCH para atualizar;
 - DELETE para remover.
- Documente os endpoints (pode usar ferramentas como Swagger ou Postman).
- Valide os dados recebidos para evitar erros ou falhas de segurança.
- Use códigos de status HTTP adequados (`200 OK`, `201 Created`, `400 Bad Request`, etc.).
- Divida o código em arquivos e módulos para facilitar a manutenção.

8.3 - Dicas para desenvolvimento de APIs

- Pense na **segurança** desde o início: autenticação, autorização e criptografia.
- Use padrões como REST ou GraphQL conforme a necessidade.
- Considere adicionar **páginas de status**, logs e monitoramento.
- Se a API crescer, pense em técnicas como cache, paginação e versionamento.
- Mantenha as respostas **padronizadas** (mesma estrutura JSON para sucesso e erro).
- Teste a API com ferramentas como Postman ou Insomnia antes de publicar.

9 - Tratamento de dados avançado

9.1 - Criptografia de dados

Criptografia é o processo de transformar dados legíveis em dados cifrados, para que só quem possui a chave correta possa ler. É essencial para proteger informações sensíveis, como senhas e dados pessoais.

No Python, é comum usar bibliotecas como:

- `cryptography`
- `hashlib`

Exemplo de criptografia simétrica (mesma chave para cifrar e decifrar):

```
from cryptography.fernet import Fernet

# Gerar chave e salvar
chave = Fernet.generate_key()
fernet = Fernet(chave)

# Criptografar
mensagem = 'Segredo123'.encode()
mensagem_cifrada = fernet.encrypt(mensagem)

# Descriptografar
mensagem_original = fernet.decrypt(mensagem_cifrada)
print(mensagem_original.decode())
```

9.2 - Modelos hash de dados + Criptografia de dados

- **Hash** é uma função que transforma dados em um valor fixo (ex.: 32 ou 64 caracteres). Exemplo: SHA-256.
- É muito usada para armazenar senhas de forma segura, pois não é possível reverter diretamente o hash para a senha original.
- **Hash não serve para cifrar e depois decifrar**, apenas para verificar se o dado informado gera o mesmo hash.

Exemplo usando hashlib:

```
import hashlib

senha = 'minhaSenha123'
hash_senha = hashlib.sha256(senha.encode()).hexdigest()

print(hash_senha)
```

Combinar hash e criptografia é uma prática comum:

- Use **hash** para proteger senhas armazenadas;
- Use **criptografia** para proteger dados que precisam ser recuperados depois (como tokens, documentos ou informações pessoais).