

Building a Hack Planner

GHW July 2024

Why build a hack planner?

A hack planner is a tool or application designed to help hackers and participants effectively plan, organize, and manage their activities during a hackathon.

Why build a hack planner?

Benefits:

1. Time Management
2. Task and Goal Tracking / Reminders
3. Team Collaboration
4. Resource Management

Our hack planner features:

1. Task Management (Day 1)

We will learn CRUD operations, routing, and databases.

Our hack planner features:

2. Time Tracking and Reminders (Day 2)

We will learn to connect our app to an API.

Our hack planner features:

3. Team Collaboration (Day 3)

We will learn to create and authenticate user accounts.

Our hack planner features:

4. “Get Started” (Day 4)

We will learn to create a search feature to get your project started, using ChatGPT AI.

First: Set up your Python coding environment!

Day 1: Task Management / CRUD Operations

What is CRUD?

C - Create

R - Read

U - Update

D - Delete

Step 1

Type 'pip install Flask' in your terminal

Step 2

Create a new directory for your project and navigate to it using your code editor.

Step 3

Create a new Python file called 'app.py'

Step 4

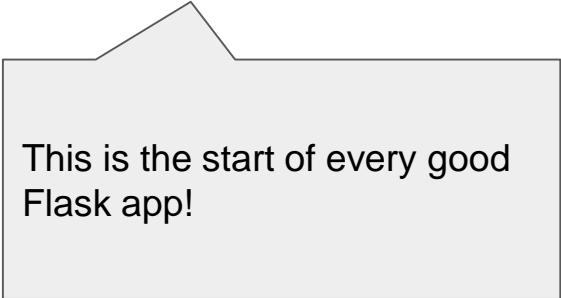
Import the modules

```
1 from flask import Flask, render_template, request, redirect
```

Step 5

Initialize the Flask application

```
1  from flask import Flask, render_template, request, redirect
2
3  app = Flask(__name__)
```



This is the start of every good
Flask app!

Routes

What are routes?

Routes in Flask determine the URL paths that the application can respond to.

A route specifies a URL pattern and associates it with a specific function, known as a view function.

When a user accesses a specific URL in their browser, Flask matches the URL to a route and triggers the associated view function to handle the request.

Routes are defined using the `@app.route` decorator in Flask.

Here's an example

```
@app.route('/')
```

```
def index():
```

```
    return 'Hello, world!'
```

In the above example, the / route is associated with the index() view function. Whenever a user visits the root URL of the application, Flask calls the index() function and returns the string "Hello, world!".

Views

What is a view?

In Flask, views are Python functions that handle HTTP requests and generate responses to be sent back to the client.

A view function is responsible for processing the user's request, performing necessary actions, and returning an appropriate response.

The response can be in various forms, such as HTML, JSON, or redirects.

Views are typically associated with routes, as the route determines which view function is called for a specific URL.

What is a view? Example:

```
@app.route('/tasks')
```

```
def tasks():
```

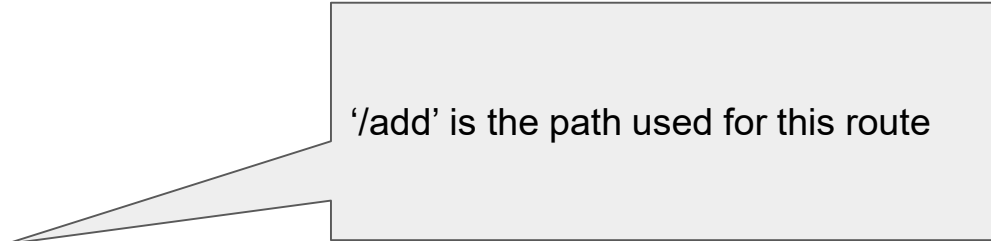
```
    tasks = fetch_tasks_from_database() # Retrieve tasks from the database
```

```
    return render_template('tasks.html', tasks=tasks)
```

In the above example, the /tasks route is associated with the tasks() view function. When a user visits the /tasks URL, Flask calls the tasks() function, retrieves tasks from the database, and returns the rendered tasks.html template with the tasks passed as a variable.

Part 1. Implementing CRUD

CreateRUD



`'/add'` is the path used for this route

```
@app.route('/add', methods=['POST'])
```

```
def add_item():
```

```
    item = request.form['item']
```

```
    items.append(item) # Append the new item to the list (not stored in a database)
```

```
    return redirect('/')
```

CreateRUD

```
@app.route('/add', methods=['POST'])
```

```
def add_item():
```

```
    item = request.form['item']
```

```
    items.append(item) # Append the new item to the list (not stored in a database)
```

```
    return redirect('/')
```

POST is an HTTP method. It allows the user to **send AND receive** data (request and response)

CReadUD

```
@app.route('/')
```

```
def checklist():
```

```
    return render_template('checklist.html', items=items)
```

CReadUD

```
@app.route('/')  
def checklist():
```

By default, route uses the **GET** method. That only lets the user receive, not send, data.

```
    return render_template('checklist.html', items=items)
```

CRUpdated

```
@app.route('/edit/<int:item_id>', methods=['GET', 'POST'])
```

```
def edit_item(item_id):
```

```
    item = items[item_id - 1] # Retrieve the item based on its index in the list (not updated in a database)
```

```
    if request.method == 'POST':
```

```
        new_item = request.form['item']
```

```
        items[item_id - 1] = new_item # Update the item in the list (not stored in a database)
```

```
        return redirect('/')
```

```
    return render_template('edit.html', item=item, item_id=item_id)
```

CRUDelete

```
@app.route('/delete/<int:item_id>')
```

```
def delete_item(item_id):
```

```
    del items[item_id - 1] # Delete the item from the list (not stored in a database)
```

```
    return redirect('/')
```

Part 2. Creating HTML Templates

Template directory

You must place all templates in a directory called “template”



checklist.html, the main/front page

```
<!DOCTYPE html>
<html>
<head>
  <title>Hack Planner App</title>
</head>
<body>
  <h1>Hack Planner</h1>
  <ul>
    {% for item in items %}
      <li>
        {{ item }}
        <a href="/edit/{{ loop.index }}">Edit</a>
        <a href="/delete/{{ loop.index }}">Delete</a>
      </li>
    {% endfor %}
  </ul>
  <form action="/add" method="post">
    <input type="text" name="item" placeholder="Enter item">
    <button type="submit">Add</button>
  </form>
</body>
</html>
```

Edit.html, the edit task page

```
<!DOCTYPE html>
<html>
<head>
  <title>Hack Planner App</title>
</head>
<body>
  <h1>Edit Item</h1>
  <form action="/edit/{{ item_id }}" method="post">
    <input type="text" name="item" value="{{ item }}">
    <button type="submit">Save</button>
  </form>
</body>
</html>
```


Run the Flask App

- `export FLASK_APP=app.py` (Mac OS)
- `set FLASK_APP=app.py` (Windows)

- `flask run`

Part 3. Increasing the visual appeal 🤗

New, “static” folder

Used for assets that do not change over the course of the app’s functionality.



checklist.html, the main/front page

```
<!DOCTYPE html>
<html>
<head>
  <title>Hack Planner App</title>
</head>
<body>
  <h1>Hack Planner</h1>
  <ul>
    {% for item in items %}
      <li>
        {{ item }}
        <a href="/edit/{{ loop.index }}">Edit</a>
        <a href="/delete/{{ loop.index }}">Delete</a>
      </li>
    {% endfor %}
  </ul>
  <form action="/add" method="post">
    <input type="text" name="item" placeholder="Enter item">
    <button type="submit">Add</button>
  </form>
</body>
</html>
```

Edit.html, the edit task page

```
<!-- edit.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Hack Planner App</title>
  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
  <div class="container">
    <h1>Edit Item</h1>
    <form class="edit-form" action="/edit/{{ item_id }}" method="post">
      <input type="text" name="item" value="{{ item }}">
      <button type="submit">Save</button>
    </form>
  </div>
</body>
</html>
```

Part 4. Adding a database connection

CREATE TABLE

```
db_path = 'checklist.db'
```

```
def create_table():  
    conn = sqlite3.connect(db_path)  
    c = conn.cursor()  
    c.execute("CREATE TABLE IF NOT EXISTS checklist  
              (id INTEGER PRIMARY KEY AUTOINCREMENT, item TEXT)")  
    conn.commit()  
    conn.close()
```

Let's revisit our CRUD operations!

CreateRUD

```
def add_item(item):  
    conn = sqlite3.connect(db_path)  
    c = conn.cursor()  
    c.execute("INSERT INTO checklist (item) VALUES (?)", (item,))  
    conn.commit()  
    conn.close()
```

...

```
@app.route('/add', methods=['POST'])  
def add():  
    item = request.form['item']  
    add_item(item)  
    return redirect('/')
```

CReadUD

```
def get_items():  
    conn = sqlite3.connect(db_path)  
    c = conn.cursor()  
    c.execute("SELECT * FROM checklist")  
    items = c.fetchall()  
    conn.close()  
    return items  
  
...  
  
@app.route('/')  
def checklist():  
    create_table()  
    items = get_items()  
    return render_template('checklist.html', items=items)
```

CRUpdated

```
def update_item(item_id, new_item):  
    conn = sqlite3.connect(db_path)  
    c = conn.cursor()  
    c.execute("UPDATE checklist SET item = ? WHERE id = ?", (new_item, item_id))  
    conn.commit()  
    conn.close()
```

...

```
@app.route('/edit/<int:item_id>', methods=['GET', 'POST'])  
def edit(item_id):  
    if request.method == 'POST':  
        new_item = request.form['item']  
        update_item(item_id, new_item)  
        return redirect('/')  
    else:  
        items = get_items()  
        item = next((x[1] for x in items if x[0] == item_id), None)  
        return render_template('edit.html', item=item, item_id=item_id)
```

CRUDelete

```
def delete_item(item_id):  
    conn = sqlite3.connect(db_path)  
    c = conn.cursor()  
    c.execute("DELETE FROM checklist WHERE id = ?", (item_id,))  
    conn.commit()  
    conn.close()
```

...

```
@app.route('/delete/<int:item_id>')  
def delete(item_id):  
    delete_item(item_id)  
    return redirect('/')
```

Let's revisit our front-end

Checklist

(1, 'aef')

Edit

Delete

(2, 'arg')

Edit

Delete

(3, 'r')

Edit

Delete

(4, 'a')

Edit

Delete

Enter item

Add

Can anyone fix this?

Let's fix it

```
<ul class="checklist-items">
  {% for item in items %}
  <li>
    {{ item[1] }}
    <a href="/edit/{{ loop.index }}">Edit</a>
    <a href="/delete/{{ loop.index }}">Delete</a>
  </li>
  {% endfor %}
</ul>
```


Now, “delete” and “edit” aren’t working! Any ideas?

Checklist

aefragr	Edit	Delete
adyt	Edit	Delete
aydn3	Edit	Delete
atshy	Edit	Delete

Add

Let's fix it

```
<ul class="checklist-items">
  {% for item in items %}
    <li>
      {{ item[1] }}
      <a href="/edit/{{ item[0] }}">Edit</a>
      <a href="/delete/{{ item[0] }}">Delete</a>
    </li>
  {% endfor %}
</ul>
```

