Manual: App Creation using Hugging Face Spaces + Gradio

Using ChatGPT Prompts to Generate `app.py` and `requirements.txt`

Prof. Santhi
SRC – SASTRA

Contents

1 What This Manual Covers **2**

2 App Types: Mobile Apps vs Web Apps **2**
2.1 Mobile Apps . . . 2
2.2 Web Apps . . . 2

3 Hugging Face Spaces + Gradio: What You Can Build **2**
3.1 Frameworks Available in Spaces (Typical) . . . 2
3.2 Why Gradio . . . 2

4 Create a Space in Hugging Face (Step-by-Step) **3**
4.1 Create Account and Login . . . 3
4.2 Create a New Space . . . 3
4.3 Add Files to the Space . . . 3
4.4 How Deployment Works . . . 3

5 Using ChatGPT to Generate `app.py` and `requirements.txt` **3**
5.1 Recommended Prompt Structure . . . 3
5.2 Golden Rules for Success . . . 3

6 Common Errors in Spaces and How to Fix **4**
6.1 Build-time Errors (Dependency issues) . . . 4
6.2 Runtime Errors (App crashes on startup) . . . 4
6.3 Tips to Debug Quickly . . . 4

7 Five Ready Prompts + Generated Files (Copy-Paste) **4**
7.1 Prompt 1 (Computer Engineering): CPU Scheduling Calculator . . . 4
7.2 Prompt 2 (Finance): Simple Loan EMI Calculator . . . 6
7.3 Prompt 3 (Biochemistry): DNA GC Content Calculator . . . 7
7.4 Prompt 4 (English Department): Essay Rubric Score Calculator . . . 8
7.5 Prompt 5 (Research General): Classification Metrics Calculator . . . 9

8 Checklist Before You Share the Space **10**

9 Best Practices for Teaching and Workshops **10**

1 What This Manual Covers

This manual explains how to build and deploy simple web apps using **Hugging Face Spaces** and the **Gradio** framework. It includes:

- Types of apps (mobile vs web) and what you can build in Spaces
- Step-by-step procedure to create a Space and deploy a Gradio app
- How to use ChatGPT to generate `app.py` and `requirements.txt`
- Five ready-to-copy prompts (Computer Engineering, Finance, Biochemistry, English, General Research)
- For each prompt: corresponding `app.py` + `requirements.txt`
- Common errors and debugging tips

2 App Types: Mobile Apps vs Web Apps

2.1 Mobile Apps

Mobile apps run on Android/iOS and are typically distributed via app stores. They often require:

- Mobile UI frameworks (e.g., Flutter, React Native, native Android/iOS)
- Build pipelines, signing keys, store approvals

Hugging Face Spaces is not a mobile app store. However, a Space provides a **web link** that can be opened on mobile browsers and shared instantly (very useful for demos, labs, and teaching).

2.2 Web Apps

Web apps run in browsers. Hugging Face Spaces is primarily for web deployment. Benefits:

- Shareable URL for students/participants
- No local installation for the end user
- Fast iteration: update code and redeploy quickly

3 Hugging Face Spaces + Gradio: What You Can Build

Spaces supports several SDKs; this manual focuses on **Gradio**.

3.1 Frameworks Available in Spaces (Typical)

- **Gradio** (Python, easiest UI for ML demos)
- **Streamlit** (Python dashboard-style apps)
- **Static HTML** (simple websites)
- **Docker** (advanced, any stack inside a container)

3.2 Why Gradio

Gradio is excellent for academic demos:

- UI components like textbox, file upload, buttons, tables, plots
- Quick callback functions (`button.click(fn=...)`)
- Works well in Spaces without heavy setup

4 Create a Space in Hugging Face (Step-by-Step)

4.1 Create Account and Login

1. Create an account and log in to Hugging Face.

4.2 Create a New Space

1. Go to Spaces: `https://huggingface.co/spaces`
2. Click **Create new Space**.
3. Choose:
   - **Space name**: meaningful (e.g., `src-aiml-readyref`)
   - **Visibility**: Public (for sharing) or Private (restricted)
   - **SDK**: **Gradio**
   - **Hardware**: CPU is enough for most non-heavy apps
4. Click **Create Space**.

4.3 Add Files to the Space

A Gradio Space typically needs:

- `app.py` (main app code)
- `requirements.txt` (dependencies)
- Optional assets: `logo.jpg`, `resources.xlsx`, sample CSV, etc.

4.4 How Deployment Works

- When you commit/update files, HF builds the environment from `requirements.txt`
- Then it runs `app.py`
- If errors occur, check the **Logs** tab

5 Using ChatGPT to Generate `app.py` and `requirements.txt`

5.1 Recommended Prompt Structure

To get high-quality code, include:

1. **Goal**: what the app does
2. **Inputs**: what user enters or uploads
3. **Outputs**: what the app shows / downloads
4. **UI layout**: header, buttons, tables, charts
5. **Constraints**: "single `gr.Blocks()`, no external API calls, safe defaults"
6. **Return format**: "Output only `app.py` and `requirements.txt` in code blocks"

5.2 Golden Rules for Success

- Keep code in one file unless needed
- Avoid unused components (common runtime errors)
- Pin minimal dependencies
- Test locally (optional) or use Space logs to debug

6 Common Errors in Spaces and How to Fix

6.1 Build-time Errors (Dependency issues)

Symptoms: build fails, pip errors.

- **Cause:** wrong package name or incompatible versions
- **Fix:** keep `requirements.txt` minimal; avoid conflicting version pins

6.2 Runtime Errors (App crashes on startup)

Symptoms: "Application Startup... Traceback..." in logs. Common causes:

- **NameError**: referenced component variable not defined
- **Wrong Gradio argument**: parameter not supported in your Gradio version
- **Missing file**: `logo.jpg`, `resources.xlsx` not in repo root

6.3 Tips to Debug Quickly

1. Open **Logs** in Space and read the first real error line
2. Confirm file names (case-sensitive): `logo.jpg` vs `Logo.jpg`
3. Print diagnostics inside code:

```
import os
print("Files:", os.listdir("."))
```

4. If Gradio argument fails, remove it or check supported arguments (version mismatch)
5. Restart Space after big changes (in Settings or via rebuild)

7 Five Ready Prompts + Generated Files (Copy-Paste)

Each example below contains:

- One simple prompt to give ChatGPT
- A working `app.py`
- A minimal `requirements.txt`

7.1 Prompt 1 (Computer Engineering): CPU Scheduling Calculator

Copy Prompt

```
Create a Hugging Face Gradio app. Output ONLY app.py and requirements.txt.
App: CPU scheduling calculator for FCFS.
Inputs: list of processes with arrival time and burst time (user enters as CSV-like
    text).
Outputs: table with start time, completion time, waiting time, turnaround time, and
    averages.
Use a simple UI: textbox input, button "Compute", dataframe output, markdown for
    averages.
Use only one gr.Blocks() and demo.launch().
```

app.py

```
import gradio as gr
import pandas as pd

def parse_text(s):
    # Expected lines: PID,Arrival,Burst
    rows = []
    for line in (s or "").strip().splitlines():
        if not line.strip():
            continue
        parts = [p.strip() for p in line.split(",")]
        if len(parts) != 3:
            raise ValueError("Each line must be: PID,Arrival,Burst")
        pid, at, bt = parts[0], int(parts[1]), int(parts[2])
        rows.append((pid, at, bt))
    if not rows:
        raise ValueError("Enter at least one process line.")
    return rows

def fcfs_compute(text):
    try:
        procs = parse_text(text)
        procs.sort(key=lambda x: x[1]) # by arrival
        t = 0
        out = []
        for pid, at, bt in procs:
            if t < at:
                t = at
            start = t
            comp = t + bt
            tat = comp - at
            wt = tat - bt
            out.append([pid, at, bt, start, comp, wt, tat])
            t = comp

        df = pd.DataFrame(out, columns=[
            "PID","Arrival","Burst","Start","Completion","Waiting","Turnaround"
        ])
        avg_wt = df["Waiting"].mean()
        avg_tat = df["Turnaround"].mean()

        md = f"### Averages\n- Average Waiting Time: {avg_wt:.2f}\n- Average Turnaround
             Time: {avg_tat:.2f}"
        return md, df
    except Exception as e:
        return f" {e}", pd.DataFrame()

with gr.Blocks(title="FCFS Scheduling Calculator") as demo:
    gr.Markdown("## FCFS CPU Scheduling Calculator\nEnter one process per line: `PID,
        Arrival,Burst`")
    inp = gr.Textbox(label="Processes", lines=8, value="P1,0,5\nP2,1,3\nP3,2,8")
    btn = gr.Button("Compute")
    out_md = gr.Markdown()
    out_df = gr.Dataframe(interactive=False)
    btn.click(fcfs_compute, inputs=inp, outputs=[out_md, out_df])

demo.launch()
```

requirements.txt

```
gradio>=4.0.0
pandas
```

7.2 Prompt 2 (Finance): Simple Loan EMI Calculator

Copy Prompt

```
Create a Hugging Face Gradio app. Output ONLY app.py and requirements.txt.
App: Loan EMI calculator.
Inputs: principal (INR), annual interest rate (%), tenure (years).
Outputs: markdown summary (EMI, total interest, total payable) and an amortization
    table.
Use one gr.Blocks() and demo.launch().
```

app.py

```
import math
import pandas as pd
import gradio as gr

def emi_calc(P, annual_rate, years):
    if P is None or P <= 0: return "Enter valid principal.", pd.DataFrame()
    if annual_rate is None or annual_rate < 0: return "Enter valid rate.", pd.
        DataFrame()
    if years is None or years <= 0: return "Enter valid tenure.", pd.DataFrame()

    P = float(P); r = (float(annual_rate)/100)/12
    n = int(round(float(years)*12))
    if r == 0:
        emi = P/n
    else:
        emi = P*r*((1+r)**n)/(((1+r)**n)-1)

    total = emi*n
    interest_total = total - P

    md = (f"### Results\n"
          f"- Monthly EMI: {emi:,.2f}\n"
          f"- Total Interest: {interest_total:,.2f}\n"
          f"- Total Payable: {total:,.2f}\n"
          f"- Tenure: {n} months")

    bal = P
    rows=[]
    for m in range(1, n+1):
        interest = bal*r
        principal = emi - interest
        if m == n: # adjust last row
            principal = bal
            emi_last = principal + interest
            bal = 0.0
            rows.append([m, round(emi_last,2), round(principal,2), round(interest,2),
                round(bal,2)])
            break
```

```
        bal -= principal
        rows.append([m, round(emi,2), round(principal,2), round(interest,2), round(bal
            ,2)])

    df = pd.DataFrame(rows, columns=["Month","EMI","Principal Paid","Interest Paid","
        Balance"])
    return md, df

with gr.Blocks(title="Loan EMI Calculator") as demo:
    gr.Markdown("## Simple Loan EMI Calculator")
    with gr.Row():
        P = gr.Number(label="Loan Amount ()", value=500000, precision=0)
        rate = gr.Number(label="Annual Interest Rate (%)", value=9.0, precision=2)
        years = gr.Number(label="Tenure (Years)", value=5, precision=2)

    btn = gr.Button("Calculate")
    out_md = gr.Markdown()
    out_df = gr.Dataframe(interactive=False)
    btn.click(emi_calc, inputs=[P, rate, years], outputs=[out_md, out_df])

demo.launch()
```

requirements.txt

```
gradio>=4.0.0
pandas
```

7.3 Prompt 3 (Biochemistry): DNA GC Content Calculator

Copy Prompt

```
Create a Hugging Face Gradio app. Output ONLY app.py and requirements.txt.
App: DNA sequence analyzer.
Input: DNA sequence (string).
Output: GC% and base counts (A,C,G,T) in a table.
UI: textbox + button + dataframe + markdown.
One gr.Blocks() only.
```

app.py

```
import gradio as gr
import pandas as pd

def gc_calc(seq):
    s = (seq or "").strip().upper()
    s = "".join([c for c in s if c in "ACGT"])
    if not s:
        return "Enter a DNA sequence containing A/C/G/T.", pd.DataFrame()
    counts = {b: s.count(b) for b in "ACGT"}
    gc = counts["G"] + counts["C"]
    gc_pct = (gc / len(s)) * 100
    df = pd.DataFrame([["A", counts["A"]],
                       ["C", counts["C"]],
                       ["G", counts["G"]],
```

```
                     ["T", counts["T"]],
                     ["Length", len(s)],
                     ["GC%", f"{gc_pct:.2f}%"]],
                    columns=["Metric","Value"])
    return f"### GC Content = {gc_pct:.2f}%", df

with gr.Blocks(title="DNA GC Content Calculator") as demo:
    gr.Markdown("## DNA GC Content Calculator\nPaste DNA sequence (A/C/G/T).")
    seq = gr.Textbox(lines=6, label="DNA Sequence", value="ACGTTGCAACGT")
    btn = gr.Button("Compute")
    out_md = gr.Markdown()
    out_df = gr.Dataframe(interactive=False)
    btn.click(gc_calc, inputs=seq, outputs=[out_md, out_df])

demo.launch()
```

requirements.txt

```
gradio>=4.0.0
pandas
```

7.4 Prompt 4 (English Department): Essay Rubric Score Calculator

Copy Prompt

```
Create a Hugging Face Gradio app. Output ONLY app.py and requirements.txt.
App: Essay rubric scoring tool.
Inputs: Content(0-10), Grammar(0-10), Vocabulary(0-10), Coherence(0-10), Originality
    (0-10).
Output: total/50, percentage, and a table with each criterion and score.
One gr.Blocks(), clean UI.
```

app.py

```
import gradio as gr
import pandas as pd

def score(content, grammar, vocab, coherence, originality):
    vals = [content, grammar, vocab, coherence, originality]
    names = ["Content","Grammar","Vocabulary","Coherence","Originality"]
    for n,v in zip(names, vals):
        if v is None or v < 0 or v > 10:
            return "Enter scores between 0 and 10.", pd.DataFrame()
    total = sum(vals)
    pct = (total/50)*100
    df = pd.DataFrame({"Criterion": names, "Score (0-10)": vals})
    md = f"### Result\n- Total: {total:.0f}/50\n- Percentage: {pct:.2f}%"
    return md, df

with gr.Blocks(title="Essay Rubric Scorer") as demo:
    gr.Markdown("## Essay Rubric Score Calculator")
    with gr.Row():
        c = gr.Number(label="Content (0-10)", value=8, precision=0)
        g = gr.Number(label="Grammar (0-10)", value=7, precision=0)
```

```
        v = gr.Number(label="Vocabulary (0-10)", value=7, precision=0)
        co = gr.Number(label="Coherence (0-10)", value=8, precision=0)
        o = gr.Number(label="Originality (0-10)", value=8, precision=0)
    btn = gr.Button("Compute Score")
    out_md = gr.Markdown()
    out_df = gr.Dataframe(interactive=False)
    btn.click(score, inputs=[c,g,v,co,o], outputs=[out_md,out_df])

demo.launch()
```

requirements.txt

```
gradio>=4.0.0
pandas
```

7.5 Prompt 5 (Research General): Classification Metrics Calculator

Copy Prompt

```
Create a Hugging Face Gradio app. Output ONLY app.py and requirements.txt.
App: classification metrics calculator from TP, TN, FP, FN.
Input: TP, TN, FP, FN.
Output: table with columns Metric and Value (accuracy, precision, recall, specificity,
     F1, etc).
One gr.Blocks() only.
```

app.py

```
import math
import pandas as pd
import gradio as gr

def safe_div(a,b):
    return float("nan") if b==0 else a/b

def compute(tp, tn, fp, fn):
    for name, v in [("TP",tp),("TN",tn),("FP",fp),("FN",fn)]:
        if v is None or v < 0:
            return pd.DataFrame([["Error", f"{name} must be >= 0"]], columns=["Metric
                ","Value"])
    tp, tn, fp, fn = int(tp), int(tn), int(fp), int(fn)
    total = tp+tn+fp+fn

    acc = safe_div(tp+tn, total)
    prec = safe_div(tp, tp+fp)
    rec = safe_div(tp, tp+fn)
    spec = safe_div(tn, tn+fp)
    f1 = float("nan") if (math.isnan(prec) or math.isnan(rec) or (prec+rec)==0) else
        2*prec*rec/(prec+rec)

    def fmt(x):
        return "NA (div by zero)" if (isinstance(x,float) and math.isnan(x)) else f"{x
            :.6f} ({x*100:.2f}%)"
```

```
    rows = [
        ["TP", tp], ["TN", tn], ["FP", fp], ["FN", fn], ["Total", total],
        ["Accuracy", fmt(acc)],
        ["Precision", fmt(prec)],
        ["Recall (Sensitivity)", fmt(rec)],
        ["Specificity", fmt(spec)],
        ["F1-score", fmt(f1)],
    ]
    return pd.DataFrame(rows, columns=["Metric","Value"])

with gr.Blocks(title="Classification Metrics") as demo:
    gr.Markdown("## Classification Metrics Calculator (TP, TN, FP, FN)")
    with gr.Row():
        tp = gr.Number(label="TP", value=50, precision=0)
        tn = gr.Number(label="TN", value=40, precision=0)
        fp = gr.Number(label="FP", value=5, precision=0)
        fn = gr.Number(label="FN", value=10, precision=0)
    btn = gr.Button("Compute Metrics")
    out = gr.Dataframe(interactive=False)
    btn.click(compute, inputs=[tp, tn, fp, fn], outputs=out)
demo.launch()
```

requirements.txt

```
gradio>=4.0.0
pandas
```

8 Checklist Before You Share the Space

- App builds successfully (green status)
- UI loads on desktop and mobile browser
- Buttons work and outputs appear
- Add sample inputs (default values) for demos
- Add a short description in Space README

9 Best Practices for Teaching and Workshops

- Keep apps small and focused (one concept per app)
- Use consistent headers and institutional branding (logo + title)
- Provide downloadable outputs (CSV/PDF/MD) where useful
- Save prompts used to generate the app (for reproducibility)