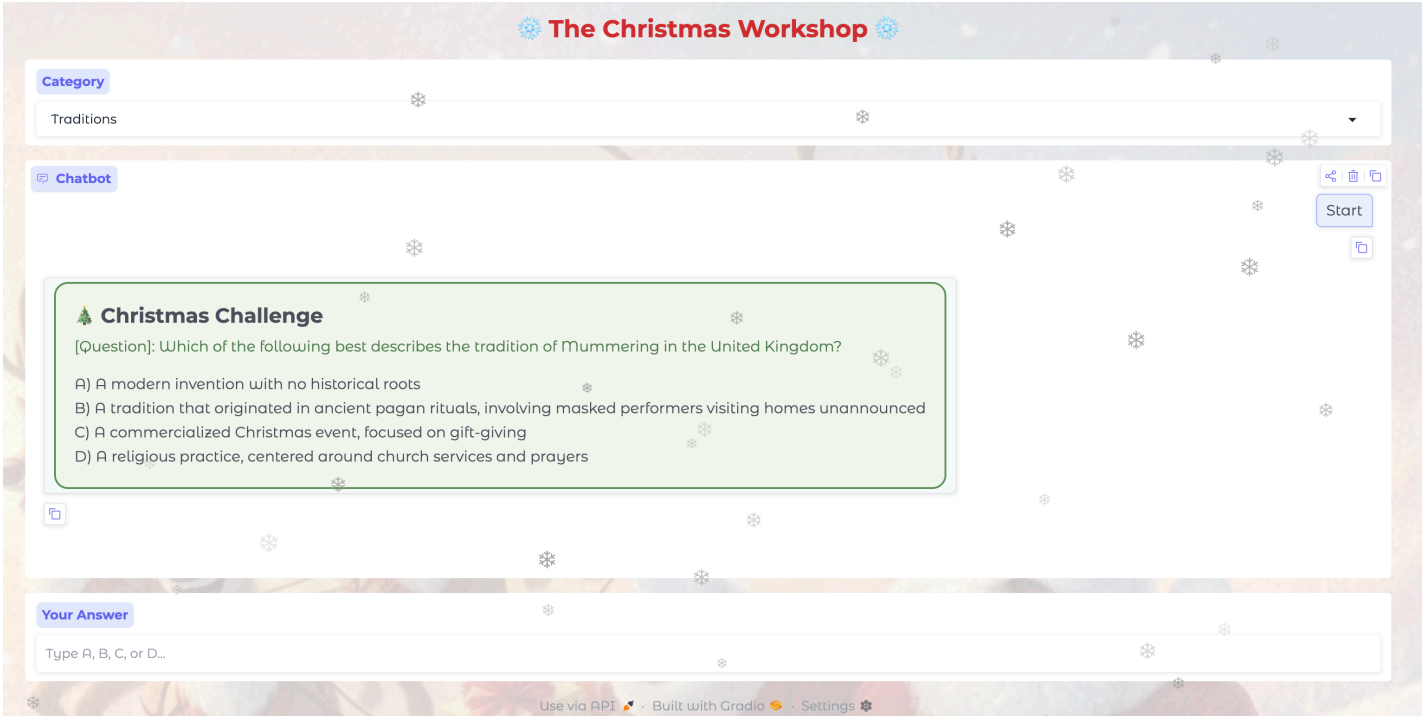
Santa Challenge Quest with Multi-Agent System

Estimated time needed: 45 minutes

Picture yourself huddled near a glowing fireplace as heavy snow falls outside. You think you know the secrets of the season, but as the North Pole's archives creak open, a question appears: Do you really know the magic behind the world's most obscure holiday traditions?

Standard trivia feels repetitive, but Santa Challenge Quest transforms your holiday gathering into a high-stakes workshop. Powered by multi-agent AI with specialized Researcher Elves and Santa himself, this app scours global folklore to judge your festive IQ in real-time.

Through a Gradio-based app featuring realistic snowfall and premium card designs, you will journey through "Festive Food," "Ancient Myths," and more. By the end, you'll have an interactive santa challenge game app that is shareable to your family and friends!

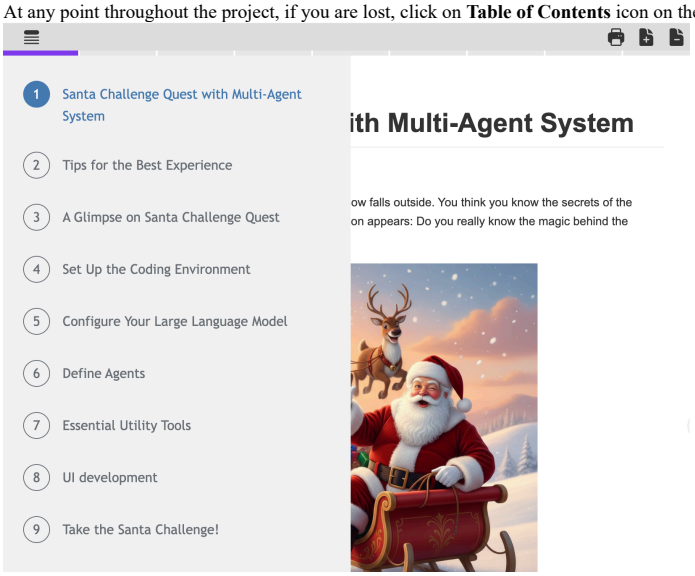


Tips for the Best Experience

Keep the following tips handy and refer to them at any point of confusion throughout the tutorial. Do not worry if they seem irrelevant now. We will go through everything step by step later.

- You can choose to follow the **Glimpse version** of building and running the application on the next page if you are in a hurry or are too excited to test the final result!

- At any point throughout the project, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.



- Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.
- To stop execution of the chatbot app in addition to closing the application tab, hit `Ctrl+C` in terminal.

A Glimpse on Santa Challenge Quest

On this page, you can build and run the chatbot in **less than 10 minutes**!

If you like to build the app from the scratch, you may directly go to the next page!

Execution Instructions

To execute the code snippet, click directly on the run button >_.

Step 1: Clone the Github repository:

```
git clone https://github.com/jianpingy/Christmas-Quiz-Chatbot.git
cd Christmas-Quiz-Chatbot
```

Step 2: Install all required libraries:

```
pip install -r requirements.txt
```

Step 2.5: Wait 3-5 minutes for libraries to be installed…

Step 3: Run the UI

```
gradio app.py
```

Step 3.5: Wait 20-30 seconds…

Step 4: CLICK the button and LAUNCH!

Try below!

Start **Santa Challenge Quest**

If you encounter the following error, please **close the tab and reopen it**.

```
Error occurred while trying to proxy: http://172.22.181.92:8888/

Please ensure your application is running and set to the correct port.
```

Set Up the Coding Environment

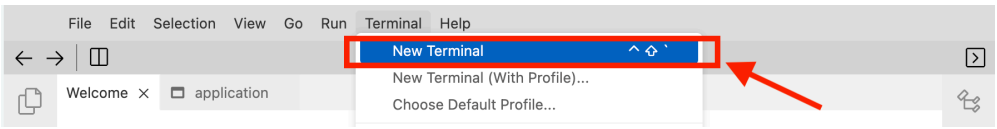
Great to have you here! I believe you are very excited about building the app on your own!

From this page, you will start your journey to develop the chatbot for the interview coach.

Step 0: Open a new terminal

We will use a new **terminal** to execute all the codes.

You can start a new terminal by clicking new terminal in the dropdown from Terminal tab.



Step 1: Create the directories (a.k.a. folders) to put your codes and files (e.g. databases)

Run the following codes to create the directories and the empty files.

Click on the Execute button >_ to run the lines.

```
mkdir santa-challenge # Main directory for the app and files
cd santa-challenge # Enter the created directory
wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/5pjyIc2IWWP_2ij-IOqAeg/requirements.txt # Get requirements.txt
touch app.py   # Main interview coach ui python file
```

Step 2: Set up a Python virtual environment

Initialize a new Python virtual environment to keep the required library versions tidy. Note, you can run the snippet directly by clicking on the run button >_.

```
python3.11 -m venv venv
source venv/bin/activate
```

Step 3: Install the required libraries

With your virtual environment activated, install all the required libraries via >_:

```
pip install -r requirements.txt
```

Approximately it will take **3-5 minutes** to install all the required libraries.

Now your environment is set up, you're ready to start building your game for Santa Challenges!

Configure Your Large Language Model

Let's first configure the Large Language Model (LLM) that power the santa challenges. We are going to use the lightweight IBM Granite LLM for best experience.

Open `app.py` by clicking the button below so that we can edit the python file.

Open **app.py** in IDE

Important: Copy and paste the codes in the following steps to fill in `app.py`.

Step 1: Import Libraries

This step imports all the necessary libraries for the project. In essence, we will import:

- `os`: Provides an interface for interacting with the **operating system**.
- `gradio as gr`:
  - A library for quickly creating **shareable web user interfaces** for machine learning models and demos.
- `Agent`: The fundamental building block of a **CrewAI** system.
- `Task`: Defines a specific, actionable **job** that an agent must complete.
- `Crew`: The **orchestrator** that manages the entire multi-agent system.
- `Process`: Specifies the **collaboration methodology** for the agents within the Crew.
- `LLM`: Refers to the **Large Language Model** itself (e.g., GPT-4, Llama).

```
import os
import gradio as gr
from crewai import Agent, Task, Crew, Process, LLM
```

Step 2: Setup the base LLM

In this step, we configure and initialize the **Large Language Model (LLM)** that powers the santa challenges.

This code snippet focuses exclusively on the initial, crucial step of **securely retrieving authentication and configuration parameters** from the operating system's environment. This is a prerequisite for connecting any application, including a CrewAI system, to a cloud service like IBM watsonx.ai.

- **The Role of Environment Variables:**
  - The `os.environ.get()` method is used to read values from the system's **environment variables**.
  - This practice is paramount for **security**, preventing sensitive data like the `WATSONX_APIKEY` from being exposed if the code is shared publicly (e.g., on GitHub).
- **Variables Being Retrieved:**
  - **`project_id`**: Retrieves the value of the `WATSONX_AI_PROJECT_ID` environment variable. This ID links the application's LLM usage to a specific project space within the IBM Cloud account.
  - **`api_key`**: Retrieves the value of the `WATSONX_APIKEY` environment variable. This is the **authentication token** required to prove the application's identity and gain access to the watsonx services.
- **Outcome:**
  The variables `project_id` and `api_key` now hold the credentials needed for the next step—initializing and authenticating the CrewAI's `LLM` client to the watsonx.ai service.

```
# --- 1. LLM Configuration ---
project_id = os.environ.get('WATSONX_AI_PROJECT_ID')
api_key = os.environ.get("WATSONX_APIKEY")
watsonx_llm = LLM(
    model="watsonx/ibm/granite-3-8b-instruct",
    base_url="https://us-south.ml.cloud.ibm.com",
    project_id=project_id,
    temperature=0.7)
```

Define Agents

This project uses a team of specialized **LLM-based agents** that work together to give you challenging Christmas questions.

Let's keep furnishing `app.py` by adding these core LLM agents that power the interview simulation.

Open **app.py** in IDE

Important: Copy and paste the codes in each of the following steps to `app.py`.

In the CrewAI framework, an **Agent** is an autonomous unit powered by an LLM that performs specific roles. Below is an explanation of the specialized agents defined for this holiday-themed system.

- `noel_researcher`:
  - **Role**: Christmas Folklore Historian.
  - **Goal**: To uncover rare and obscure Christmas facts filtered by a specific category.
  - **Backstory**: An expert in global traditions, ensuring the information is culturally rich and historically accurate.
  - **Logic**: This agent acts as the **Data Gatherer**, providing the raw material (facts) for the rest of the crew.
- `holly_host`:
  - **Role**: Head Elf of Entertainment.
  - **Goal**: To transform dry facts into an engaging Multiple Choice Question (MCQ) using a specific `|||` separator for parsing.
  - **Backstory**: A high-energy game show host persona that adds flair and excitement to the output.
  - **Logic**: This agent serves as the **Content Creator/Formatter**, translating research into a user-facing game format.
- `santa_judge`:
  - **Role**: Santa Claus.
  - **Goal**: To validate user answers, determine correctness, and provide festive feedback.
  - **Backstory**: The ultimate authority on the "Nice List," delivering responses with the warmth and cheer of Father Christmas.
  - **Logic**: This agent functions as the **Evaluator and Feedback Loop**, closing the interaction with the user.

Next Step: Would you like me to show you how to define the **Tasks** that these agents will perform to create the full holiday game workflow?

```
noel_researcher = Agent(
    role='Christmas Folklore Historian',
    goal='Discover obscure Christmas facts based on a category.',
    backstory="You are an expert in global traditions.",
    llm=watsonx_llm
)
holly_host = Agent(
    role='Head Elf of Entertainment',
    goal='Turn facts into a festive MCQ with a ||| separator.',
    backstory="You are a high-energy game show host elf.",
    llm=watsonx_llm
)
santa_judge = Agent(
    role='Santa Claus',
    goal='Validate answers and spread Christmas cheer.',
    backstory="You are Santa. You check the Nice List.",
    llm=watsonx_llm
)
```

Essential Utility Tools

Before building our multi-agent HR system, we need a few core utilities that allow the app to *read resumes, talk, and listen*.

These functions act as the building blocks for our interactive AI interviewer — handling file reading, voice synthesis, and speech recognition seamlessly within the Gradio interface.

Again, open the `app.py` by clicking the button below so that we can add these essential tools to the file.

Open **app.py** in IDE

Important: Copy and paste the codes in the following steps to fill in `app.py`.

Step 1. Define Logic of Tasks

We define two Python functions that manage the operational logic of the application, utilizing the **CrewAI** framework to coordinate agents and tasks.

🎄 `generate_christmas_challenge(category)`

This function serves as the **content engine**. It manages a multi-step pipeline where information flows from a researcher to a creative host.

- `task_research`:
  - Instructs the `noel_researcher` to find a specific fact based on the user-provided `category`.
  - *Output:* A raw summary of a Christmas tradition.
- `question_task_format`:
  - A string template that enforces a strict structure for the game. The `|||` separator is critical for the program to later split the question from the hidden answer.
- `task_format`:
  - Assigns the formatting job to the `holly_host`.
  - **Context**: It uses `context=[task_research]`, meaning the host receives the researcher's findings as input.
- `Crew` & `kickoff()`:
  - Assembles the agents and tasks into a team and executes the process.
  - *Role:* Acts as the **execution trigger** for the entire generation sequence.

🎅 `ask_santa(user_input, secret_key)`

This function handles the **interactive evaluation** phase, determining if the user's guess was correct.

- `task_judge`:
  - Provides the `santa_judge` with two critical pieces of data: the `user_input` (what the player clicked) and the `secret_key` (the actual truth).
  - *Instruction:* Explicitly tells the agent to "Reply as Santa," ensuring the persona is maintained.
- `Crew & kickoff()`:
  - Creates a micro-crew consisting only of Santa to process this single evaluation task.
  - *Role:* Acts as the **final judge** and provides the closing narrative for the user experience.

```
def generate_christmas_challenge(category):
    task_research = Task(description=f"Find one specific Christmas fact about {category}.", expected_output="A summary.", agent=noel_researcher)

    question_task_format = """Create MCQ. Format:
[Question]:
A)
B)
C)
D)
||| [Answer Letter]: [Fact]
"""
    task_format = Task(description=question_task_format, expected_output="Question block separated by |||", agent=holly_host, context=[task_research])
    crew = Crew(agents=[noel_researcher, holly_host], tasks=[task_research, task_format])
    return str(crew.kickoff())
def ask_santa(user_input, secret_key):
    task_judge = Task(description=f"User answered '{user_input}'. Truth is '{secret_key}'. Reply as Santa.", expected_output="Santa's response.", agent=santa_jud
    crew = Crew(agents=[santa_judge], tasks=[task_judge])
    return str(crew.kickoff())
```

Step 2: Define Output Format

Two helper functions are responsible for the **visual presentation** of the application. They take raw text generated by the AI agents and wrap it in styled HTML "cards" to create a festive user experience.

🎄 `format_question_card(text)`

This function styles the trivia question to look like a Christmas-themed game card.

- **Visual Style**:
  - **Border**: Uses a solid green (`#2e7d32`) border for a "Christmas Tree" aesthetic.
  - **Background**: A very light green (`#f1f8e9`) to ensure readability while maintaining the theme.
- **Logic**:
  - `text.replace('|||', '')`: This is a crucial step that **strips out the hidden answer key**. It ensures the user only sees the question and options, not the correct answer which is hidden behind the `|||` separator.
- **Role**: Acts as the **Front-end Stylist** for the challenge phase.

🎅 `format_santa_card(text)`

This function styles Santa's response to look like a piece of holiday parchment or a letter from the North Pole.

- **Visual Style**:
  - **Border**: Uses a dashed red (`#d32f2f`) border to mimic the look of holiday stationary or a postage stamp.
  - **Background**: A warm, off-white/pinkish red (`#fff5f5`) to differentiate the feedback from the question.
- **Logic**:
  - Wraps Santa's AI-generated verdict in an `<h2>` header and a stylized container.
- **Role**: Acts as the **Narrative Wrapper** for the evaluation phase.

```
def format_question_card(text):
    """Wraps the question in a festive Markdown card."""
    return f"""
<div style="border: 2px solid #2e7d32; border-radius: 15px; padding: 20px; background-color: #f1f8e9; color: #1b5e20;">
    <h2 style="margin-top: 0;">🎄 Christmas Challenge</h2>
    {text.replace('|||', '').strip()}
</div>
"""
def format_santa_card(text):
    """Wraps Santa's response in a parchment-style card."""
    return f"""
<div style="border: 2px dashed #d32f2f; border-radius: 15px; padding: 20px; background-color: #fff5f5; color: #b71c1c;">
    <h2 style="margin-top: 0;">🎅 Santa's Verdict</h2>
    {text}
</div>
"""
```

Step 3: Define the Game Logic Engine

We define a function that acts as the **Controller** for the entire application. It manages the "State Machine," deciding whether the game is currently generating a new question or evaluating a user's answer.

- **`state` and `history` Tracking**:
  - **`state`**: A dictionary that persists data between turns (e.g., whether we are in `IDLE` or `WAITING_FOR_ANSWER` mode and what the correct answer is).
  - **`history`**: Maintains the list of messages for the Gradio Chatbot component to display.
- **`if state['status'] == 'IDLE'` (The Generation Phase)**:
  - **Loading State**: It first yields an interim "Consulting the North Pole…" message to give the user immediate feedback while the AI works.
  - **The Split Logic**: It calls `generate_christmas_challenge` and uses the `|||` delimiter to split the AI's response. The first part (Question) is shown to the user, while the second part (Answer) is saved in `state['secret_key']`.
  - **Transition**: Changes the status to `WAITING_FOR_ANSWER`.
- **`elif state['status'] == 'WAITING_FOR_ANSWER'` (The Evaluation Phase)**:
  - **Validation**: It takes the `user_message` and compares it against the `secret_key` using the `ask_santa` function.
  - **UI Update**: Once Santa gives his verdict, it updates the chat history with the festive parchment card.
  - **Reset**: Sets the status back to `IDLE` so the player can start a new round.
- **`yield` keywords**:
  - This function is a **Generator**. It "yields" multiple times to update the UI progressively (showing loading messages before the final content).
  - It controls the **Visibility** of Gradio buttons, hiding the "Start" button while the user is answering and revealing a "Next" button once the round is over.

```
def game_logic(user_message, category, history, state):
    if state is None: state = {'status': 'IDLE', 'secret_key': ''}
    if history is None: history = []
    # If message is empty (from a button click), provide a default
    input_text = user_message if user_message else "Let's play!"
    if state['status'] == 'IDLE':
        history.append({"role": "user", "content": input_text})
        history.append({"role": "assistant", "content": f"✨ *Consulting the North Pole library for {category}...*"})
        yield history, state, gr.update(visible=False), gr.update(visible=False)

        full_output = generate_christmas_challenge(category)
        parts = full_output.split("|||")
        public_q = parts[0].strip() if len(parts) > 1 else full_output
        state['secret_key'] = parts[1].strip() if len(parts) > 1 else "Hidden"
        state['status'] = 'WAITING_FOR_ANSWER'

        history[-1] = {"role": "assistant", "content": format_question_card(public_q)}
        # Hide start/next, show text box for answer
        yield history, state, gr.update(visible=False), gr.update(visible=False)
    elif state['status'] == 'WAITING_FOR_ANSWER':
        history.append({"role": "user", "content": input_text})
        history.append({"role": "assistant", "content": "Checking the Nice List... 📜"})
        yield history, state, gr.update(visible=False), gr.update(visible=False)

        verdict = ask_santa(input_text, state['secret_key'])
        state['status'] = 'IDLE'

        history[-1] = {"role": "assistant", "content": format_santa_card(verdict)}
        # Show 'Next Challenge' button now that we are IDLE
        yield history, state, gr.update(visible=False), gr.update(visible=True)
```

UI development

Now, we have prepared all the ingredients for building the interview coach.

Gradio is a Python library that allows developers to easily create **interactive web interfaces** for Python code.

You can think of it as a way to turn Python scripts into simple web apps with buttons, text boxes, and image uploaders.

Open the `app.py` again to code for the interface.

Open **app.py** in IDE

Important: Copy and paste the codes in each of the following steps to complete `app.py`.

Step 1: Design Snowfall Visual Effects and Set Santa Background

These string variables contain the **Frontend Code** (JavaScript and CSS) used to inject a winter atmosphere directly into the web browser, transforming the standard Gradio interface into a holiday-themed experience.

❄️ **`snow_js`: The Dynamic Snowfall Script**

`snow_js` is a raw JavaScript snippet that creates a "live" snowfall effect on the webpage.

- **Style Injection**: It dynamically creates a `<style>` tag to define the `.sn` (snowflake) class, using **CSS Keyframe Animations** to move flakes from the top (`-10%`) to the bottom (`100vh`) of the screen.
- **The Snowflake Factory**: It uses `setInterval` to generate a new snowflake every 150 milliseconds.
  - **`Math.random()`**: Randomizes the horizontal position (`left`), the size (`fontSize`), and the transparency (`opacity`) to make the snowfall look natural and organic.
  - **Automatic Cleanup**: Uses `setTimeout` to remove each snowflake from the document after 8 seconds, preventing the browser from slowing down due to too many elements.
- **Role**: Acts as the **Animation Engine** for visual immersion.

🎅 **`santa_css`: The Visual Theme Styling**

`santa_css` is a variable that contains **Cascading Style Sheets (CSS)** used to override the default look of the Gradio container.

- **Layered Backgrounds**: It uses a sophisticated triple-layered background technique:
  1. **Top Layer**: A transparent snow texture (`snow.png`) for added depth.
  2. **Middle Layer**: A semi-transparent white `linear-gradient` to ensure the text remains readable against the background image.
  3. **Bottom Layer**: A festive image of Santa Claus provided via a URL.
- **`background-attachment: fixed`**: Implements a **Parallax-like effect** where the background stays stationary while the user scrolls through the chat history.
- **Role**: Acts as the **Skin or Theme** that defines the overall aesthetic of the application.

```
snow_js = """
function createSnow() {
    const s = document.createElement('style');
    s.innerHTML = `.sn { color: #a0a0a0; position: fixed; top: -10%; z-index: 9999; animation: f 8s linear infinite; pointer-events: none; } @keyframes f { to {
    document.head.appendChild(s);
    setInterval(() => {
        const b = document.createElement('div');
        b.className = 'sn'; b.innerHTML = '❄';
        b.style.left = Math.random() * 100 + 'vw';
        b.style.fontSize = (Math.random() * 15 + 15) + 'px'; // Larger flakes
        b.style.opacity = Math.random() * 0.8 + 0.2; // More opaque
        document.body.appendChild(b);
        setTimeout(() => b.remove(), 8000);
    }, 150); // Faster interval for "heavier" snow
}
setTimeout(createSnow, 1000);
"""
santa_css = """
.gradio-container {
    background-image: url('https://www.transparenttextures.com/patterns/snow.png'),
                      linear-gradient(rgba(255,255,255,0.8), rgba(255,255,255,0.8)),
                      url('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/HPVuGgmOV81JMulqTtrukQ/santa.png');
    background-size: cover;
    background-attachment: fixed;
}
"""
```

Step 2: Design the app interface

This final block of code assembles all the previous components—the AI agents, the logic, and the festive styling—into a fully functional, interactive web application.

❄️ `gr.Blocks`: The Main Container

The application is wrapped in a `gr.Blocks` context, which allows for highly customized layouts.

- **Head & CSS**: It injects the `snow_js` into the document header and applies the `santa_css` to the container.
- **Theme**: Uses `gr.themes.Soft()`, which provides a clean, rounded interface that complements the Christmas aesthetics.
- **`gr.HTML`**: Adds a centered, festive title at the very top of the page.

🕹️ Layout Components

- **`gr.Row()`**: Groups the **Dropdown** (for selecting trivia categories) and the **Buttons** side-by-side to save vertical space.
- **Visibility Control**: Notice `next_btn` is initialized with `visible=False`. It only appears after a user finishes one challenge, keeping the UI clean.
- **`gr.Chatbot`**: The central display where the `format_question_card` and `format_santa_card` outputs are rendered.
- **`gr.State`**: A hidden variable that tracks the game's "memory" (status and answer key) across different clicks.

⚡ Event Listeners (The "Wiring")

This section connects user actions to the `game_logic` function:

- **Button Clicks**: Both `start_btn` and `next_btn` trigger the game logic. They pass the current chatbot history and the chosen category into the function and receive updated UI states in return.
- **Text Submission**: The `msg.submit` allows players to press **Enter** to send their answer.
- **Clear Input**: The final `msg.submit(lambda: "", None, msg)` is a clever trick to instantly clear the textbox after the user submits their answer, making the experience feel snappy.

🚀 Deployment

- **`demo.launch(share=True)`**: This starts the local web server. The `share=True` parameter generates a public URL, allowing you to send your Christmas Workshop game to friends and family anywhere in the world!

```
with gr.Blocks(head=f"<script>{snow_js}</script>", css=santa_css, theme=gr.themes.Soft()) as demo:
    gr.HTML("<h1 style='text-align: center; color: #d32f2f;'>❄️ The Christmas Workshop ❄️</h1>")

    with gr.Row():
        category_drop = gr.Dropdown(label="Category", choices=["Traditions", "Food", "Clothing", "Myths", "Music"], value="Traditions")
        start_btn = gr.Button("🚀 Start Workshop", variant="primary")
        next_btn = gr.Button("🎁 Next Challenge", visible=False)

    chatbot = gr.Chatbot(height=450)
    msg = gr.Textbox(label="Your Answer", placeholder="Type A, B, C, or D...")
    state = gr.State({'status': 'IDLE', 'secret_key': ''})
    # Logic Triggers
    start_btn.click(game_logic, [gr.State("Start"), category_drop, chatbot, state], [chatbot, state, start_btn, next_btn])
    next_btn.click(game_logic, [gr.State("Next"), category_drop, chatbot, state], [chatbot, state, start_btn, next_btn])
    msg.submit(game_logic, [msg, category_drop, chatbot, state], [chatbot, state, start_btn, next_btn])
    msg.submit(lambda: "", None, msg)
if __name__ == "__main__":
    demo.launch(share=True)
```

Take the Santa Challenge!

Everything is ready, run the app by executing the code:

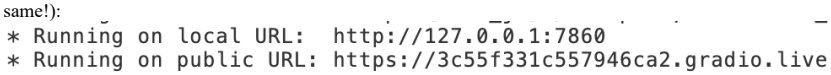
```
gradio app.py
```

Enter the web app by clicking the button:

Start **Santa Challenge**

If you encounter the following error, please **close the tab and reopen it**.

You may also enter your web app by clicking the public URL you generated! You can find it in your console, like in the following screenshot (NOTE: your URL will NOT be the same!):



Congrats, share this game by the public URL with your friends!

Author(s)

Jianping Ye | Data Scientist Intern @ IBM