

Building Agentic AI App with OpenAI Agents SDK

- Intro to OpenAI Agents SDK
- Creating a Single Agent
- Multi-Agent System (Orchestration)
- Guardrails
- Content
- Sessions
- Tools

* This is an Agentic AI framework by OpenAI

lets start with Single agent

→ in os
 from openai import OpenAI
 from dotenv import
 from agents import Agent, Runner

} setup api keys

no llm mentioned
so default chatgpt

```
agent = Agent (name="Assistant",  
               instructions="You _____")
```

Now Run the agent

```
Result = Runner.run_sync(agent, "Write a poem  
on programming.")
```

```
print (result.final_output)
```

→ _____

* Multi-Agent Systems

- a - Hand-off Pattern (Decentralized)
- b - Manager Pattern (Centralized)

Hand-off Pattern:

means there are multiple agent and one agent pass the user query to other agent which is more suitable for solving that particular query

```
→ history_tutor_agent = Agent(  
    name = "History Tutor"  
    handoff_description = "Specialist agent  
    for historical questions",  
    instructions = "_____")
```


Manager Patterner

coordinates with other agents and then collect all the info and create a final answer

```
→ Spanish agent = Agent(
    name = "Spanish Agent",
    inst = _____,
)
```

```
french agent = Agent(
    name = _____,
    inst = _____,
)
```

```
italian agent = Agent(
    name = _____,
    inst = _____,
)
```

```
manager agent = Agent(
    name = "Manager Agent",
    inst = {
```

"You are a translation manager."

"If the user asks for translation, you call the right tools"

"Always return the translations only"

),

~~_____~~

~~_____~~


```
tools = [
    Spanish_agent.as_tool(
        tool_name="Translate to Spanish",
        tool_desc=" ",
    ),
    French_agent.as_tool(
        tool_name="Translate to French",
        tool_desc=" ",
    ),
    Italian_agent.as_tool(
        tool_name="Translate to Italian",
        tool_desc=" ",
    ),
]
```

```
async def main():
    msg = " "
    # ...
```

```
orchestrator_output = await Runner.run(
    manager_agent, msg
)
```

```
print("\n== Final Answer ==")
print(orchestrator_output)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

→

Guardrail

Suppose someone asks chatgpt a hacking or unethical question, Chatgpt refuses that answer to be provided, or which are out of content and not safe.

This is what Guardrail does

lets Implement

```
→ from agents import GuardrailFunctionOutput, Agent, Runner
```

```
from pydantic import BaseModel
```

```
class HomeworkOutput(BaseModel):  
    is homework: bool  
    reasoning: str
```

this class
will be
used to
check if
question
is homework
or not

lets Great Guardrail Agent

```
guardrail_agent = Agent(  
    name = "Guardrail agent",  
    instructions = "Check if the user is  
                    asking about homework",  
    output_type = HomeworkOutput,  
)
```


history-agent =

math-agent =

```
handoff-agent = agent (
    name = "handoff agent"
    in                     
    handoffs = [math-agent, history-agent],
    input-guardrails = [
        InputGuardrail (guardrail-function
                        = homework-guardrail),
    ]
)
```

```
async def homework_guardrail (ctx, agent,
                             input-data):
    result = await Runner.run (guardrail-agent,
                              input-data, content=
                              ctx.content)
    final-output = result.final-output as (Homework
                                           Output)
```

```
return GuardrailFunctionOutput(
    output_info= final_output,
    tripwire_triggered= not final-
    output.is_homework,
```

↓
checking Homework or not
using homework class
and Guardrail
agent

```
async def main():
    result= await Runner.run(allocator
    -agent, " ")
    print(result.final_output)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```


Content:

we create class by this
to store important
info abt user

→ ~~for~~ dataclass in ~~python~~ dataclass

from agents in Agent, Runner, function tool,
RunContentWrapper

by this you
can make a
function as a
tool

@dataclass

class UserInfo:

name: str

uid: int

last purchase: str

@function tool

async def fetch_user_purchase(wrapper:
RunContentWrapper[UserInfo])

→ str:

return f"the user {wrapper.content.name} recently purchased a {wrapper.content.last_purchase}."

```

async def main():
    user_info = UserInfo(name="aroli", uid=123,
                          last_purchase="laptop")

```

```

    agent = Agent[UserInfo](
        name="Assistant",
        instructions=" ",
        tools=[fetch_user_purchase],
    )

```

```

    result = await Runner.run(
        starting_agent=agent,
        input=" ",
        content=user_info,
    )

```

```

    print(result.final_output)

```

```

if __name__ == '__main__':
    asyncio.run(main())

```

→

PAGE No. / /
DATE / /

Checkout the Session and tools
in the
youtube
or chatgpt

Session

→ session = SQLiteSession("Conversation 123",
conversation history.db)

result = am . run (agent, " ",
session=session)

print (result)

→

we see the memory chat
being stored