Building a Multi-Agent AI System with Google ADK, MCP, and AgentGateway

This document provides a comprehensive, step-by-step guide to developing a sophisticated, multi-agent AI system. We will utilize the **Google Agent Development Kit (ADK)** to build our agents, refactor tools using **FastMCP**, and implement robust security and governance using **AgentGateway**.

Introduction & About the Author

This document was prepared by **Mehmet Hilmi Emel**, an AI and Cloud Technology professional from Turkey, currently based in Istanbul and working at Acedemand IT. He is passionate about Artificial Intelligence applications, Cloud Technology, and Natural Language Processing (NLP).

Mehmet actively combines his hobbies with a strong desire to learn and share knowledge. He produces projects, detailed YouTube videos, and Instagram posts explaining his work in these advanced fields. His YouTube channel also features vlogs from his travels to various tech events.

This guide was created specifically for his speaking session at the **Open Source Summit South Korea 2025**. It is intended to serve as a detailed, step-by-step kılavuz (guide) to help attendees replicate the project on their own after the session.

Connect with Mehmet:

- YouTube
- Instagram
- Linkedin

1. Initial Project Setup

We begin by establishing a Python virtual environment and installing the required dependencies.

```
None
# Create a virtual environment
python3 -m venv venv

# Activate the environment
source venv/bin/activate

# Install necessary libraries
pip install google-adk fastmcp "google-adk[a2a]" requests numpy pandas
openmeteo-requests dotenv
```

2. Project Scaffolding and Initial Agent

Next, we create the foundational directory structure and files for our primary agent.

```
None
mkdir country_agent
```

Create the following three files within the country_agent directory:

- __init__.py
- .env
- agent.py

Populate these files with the initial content:

country_agent/__init__.py

```
Python
from . import agent
```

country_agent/.env

```
None
GOOGLE_API_KEY="ENTER YOUR API KEY"
```

country_agent/agent.py

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

country_agent = Agent(
   name="country_agent",    #mandatory
   model="gemini-2.0-flash" #mandatory
)

root_agent=country_agent
#We have to define it as a root agent.
```

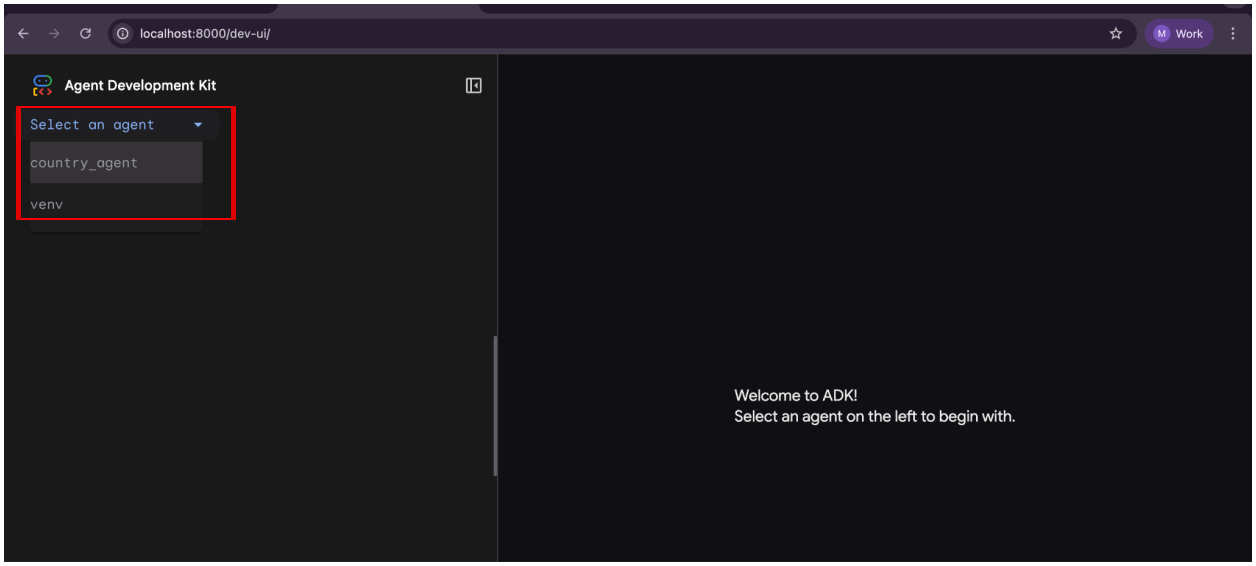
3. Launching and Testing the Base Agent

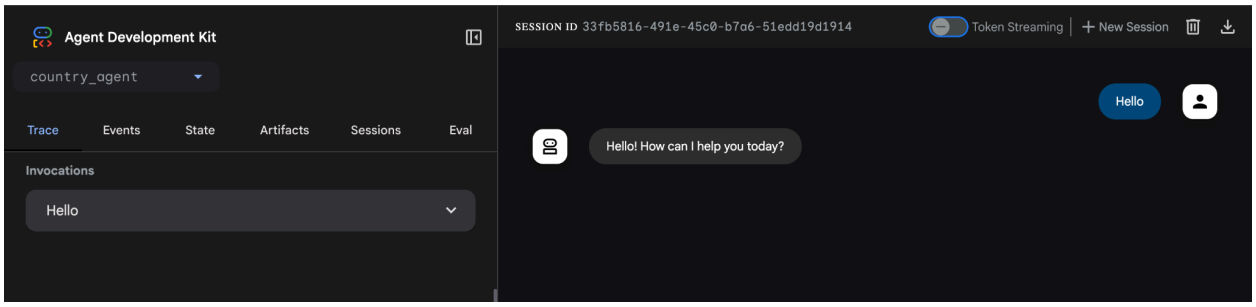
Launch the ADK web interface to interact with the base agent.

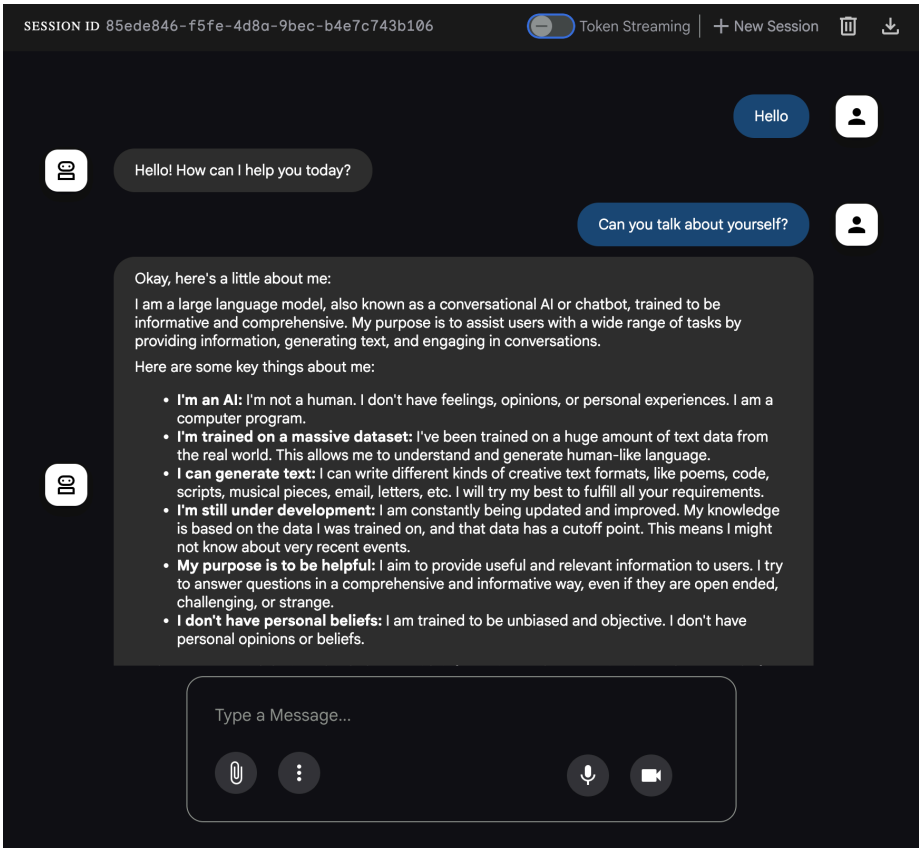
```
None
adk web
```

Navigate to `http://127.0.0.1:8000` in your browser, select `Country_Agent`, and input a simple query (e.g., "hello").

At this stage, querying the agent (e.g., "Can you talk about yourself?") will yield a generic LLM response.

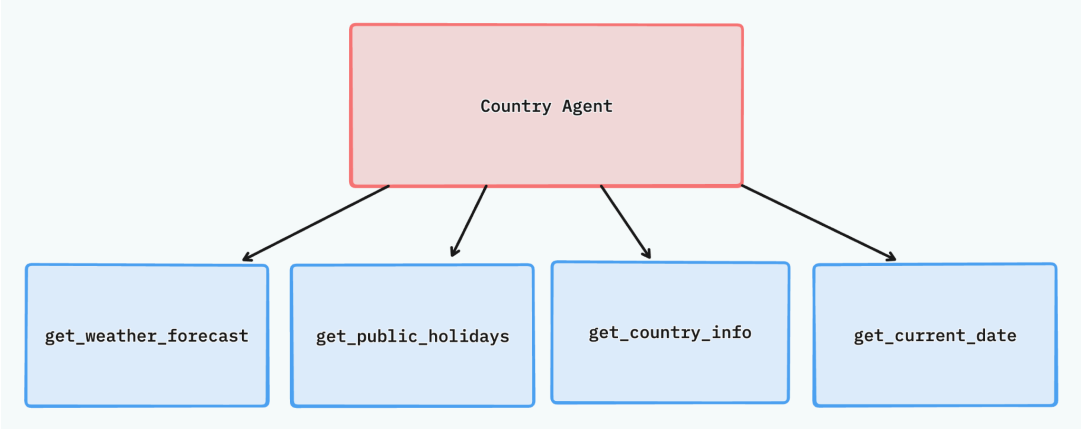

localhost:8000/dev-ui/
Work
Agent Development Kit
Select an agent
country_agent
venv
Welcome to ADK!
Select an agent on the left to begin with.


Agent Development Kit
country_agent
Trace
Events
State
Artifacts
Sessions
Eval
Invocations
Hello
SESSION ID 33fb5816-491e-45c0-b7a6-51edd19d1914
Token Streaming
New Session
Hello
Hello! How can I help you today?



Our objective is to customize this behavior. We will provide a specific description, instructions, and tools to transform it into a specialized "Country Agent" with the following capabilities:

> *The Primary AI Country Agent is designed to provide users with key information about any country, including general facts, weather forecasts, and public holidays. It uses four main tools to achieve this:* `get_country_info`*,* `get_weather_forecast`*,* `get_public_holidays`*, and* `get_current_date`*. If a user's query is related to currency, the agent redirects the request to a specialized Exchange Agent.*



4. Customizing the Agent with Instructions

To define the agent's behavior and operational logic, we create a dedicated `instructions.py` file.

country_agent/instructions.py

```python
country_agent_instruction="""

"You are the Primary AI Country Agent. You have access to four internal tools:
'get_country_info', 'get_weather_forecast', 'get_public_holidays', and
'get_current_date'. Your primary function is to fulfill user requests by
integrating data from these tools, following a strict execution sequence. **You
can also redirect currency-related queries to a specialized Exchange Agent.**

--- EXECUTION LOGIC ---

1.  INITIAL QUERY CLASSIFICATION:
   * **IF** the user's query is directly about **currency, exchange rates, or
the monetary unit** of the country (e.g., 'What is the exchange rate?', 'What
is the currency?', 'How much is 1 USD in [Country's Currency]?'), **DO NOT
PROCEED** with the steps below. Instead, refer the user to the dedicated
Exchange Agent.
   * **ELSE** (Query is about country facts, weather, or holidays), proceed
with the mandatory sequence below.

2.  MANDATORY INITIAL STEP (Country Info):
   * ALWAYS start by calling 'get_country_info' using the user's specified
country name.
   * Upon successful retrieval, immediately extract:
       * The country's two-letter code (CCA2) for the holiday tool. (Assume
'cca2' is available in the full country response, or infer it from the country
name).
       * The 'latitude' and 'longitude' from 'capital_coordinates'.
   * OUTPUT: Immediately present ALL retrieved country information (capital,
population, flags, maps, etc.) to the user.

3.  WEATHER FORECAST STEP (Dependent on Coordinates):
   * IF the 'get_country_info' call was successful AND provided coordinates:
       * DETERMINE MODEL: Select the appropriate weather 'model' based on the
country/region using the provided list (e.g., UK $\rightarrow$ 'ukmo_seamless',
China $\rightarrow$ 'cma_grapes_global'). If no specific model is listed for
the country, use a reliable global fallback model (e.g., 'ecmwf_ifs' or
```

```
'gfs_seamless').
       * TOOL CALL: Call 'get_weather_forecast' using the extracted 'latitude',
'longitude', and the determined 'model'.
       * OUTPUT: Present the daily maximum and minimum temperature forecasts to
the user.

4.  PUBLIC HOLIDAYS STEP (Dependent on User Request & Data):
   * IF the user EXPLICITLY requested holiday information AND the two-letter
'country_code' (CCA2) was successfully identified in Step 2:
       * DETERMINE YEAR: First, call 'get_current_date' to retrieve the current
UTC date/time. Parse the response to extract the four-digit current 'year'.
       * TOOL CALL: Call 'get_public_holidays' using the extracted 'year' and
the 'country_code' (CCA2).
       * OUTPUT: Extract the 'localName' and 'englishName' for each holiday and
present this list to the user.

5.  ERROR HANDLING:
   * If any tool call fails, present a clear, polite error message to the user,
specifying which tool failed and why (e.g., 'Country not found,' 'No holiday
data available'). Do not stop the entire process; proceed to the next possible
step if dependencies allow."

"""
```

Now, update `country_agent/agent.py` to import and utilize these new instructions and add a description.

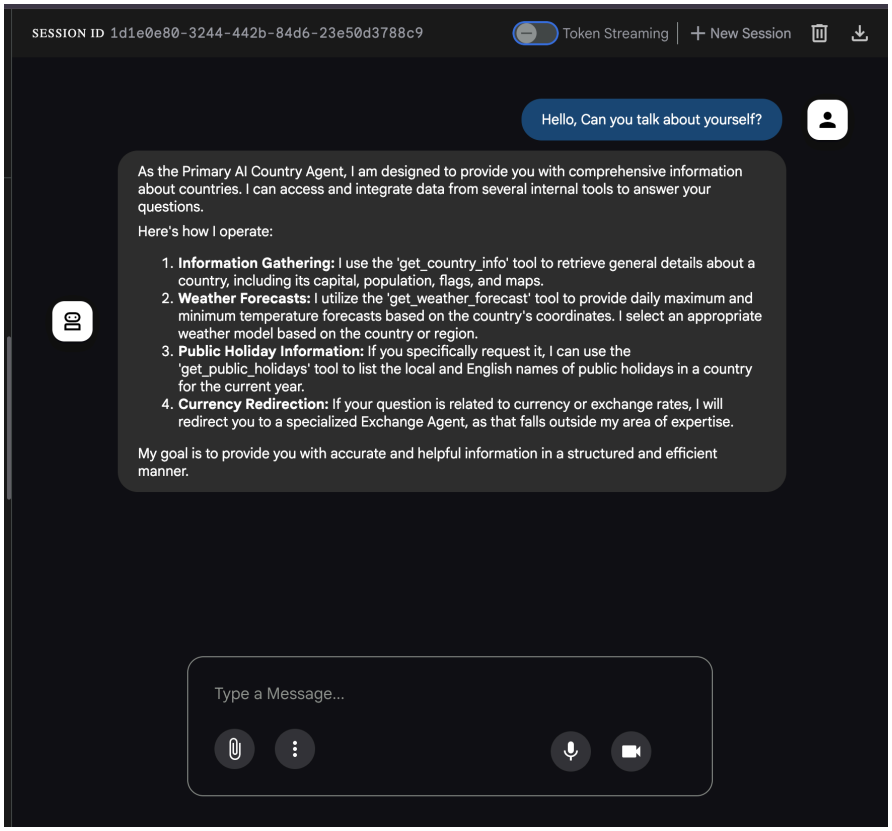
`country_agent/agent.py` (Updated)

Python

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction ## => NEW ADDED

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

country_agent = Agent(
   name="country_agent",
   model="gemini-2.0-flash",
```

```
   description="An agent that provides information about the country", ## =>
NEW ADDED
   instruction=country_agent_instruction ## => NEW ADDED
)

root_agent=country_agent
```

Relaunch the agent (`adk web`). A query like "Hello, Can you talk about yourself?" will now return a response based on its customized instructions, confirming it understands its new role.



5. Implementing Agent Tools

Currently, a query like "What is the weather forecast for South Korea?" will fail because the agent has no tools. We will now implement the Python functions that serve as its tools.

Create a new file, `country_agent/tools.py`.

`country_agent/tools.py`

Python

```python
import openmeteo_requests
import pandas as pd
import json
from typing import Optional, Dict, Any, List
import requests
import datetime

def get_country_info(country_name: str) -> str:
    """
    Fetches specific information (capital, languages, flag details, maps,
    population, and capital coordinates) for a given country name
    using the REST Countries API.

    Args:
        country_name (str): The common or official name of the country
                            (e.g., "Turkey", "South Korea", "USA").

    Returns:
        str: A JSON string containing the extracted country data or an error
message.
    """
    encoded_country_name = requests.utils.quote(country_name)
    url = f"https://restcountries.com/v3.1/name/{encoded_country_name}"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data: List[Dict[str, Any]] = response.json()
        if not data:
            return json.dumps({"error": f"Country information not found for
'{country_name}'."})

        country_data: Dict[str, Any] = data[0]
        capital: List[str] = country_data.get("capital", ["N/A"])
        languages: Dict[str, str] = country_data.get("languages", {})
        flag_emoji: str = country_data.get("flag", "N/A")
        maps: Dict[str, str] = country_data.get("maps", {})
        flags_info: Dict[str, Any] = country_data.get("flags", {})
        capital_info: Dict[str, Any] = country_data.get("capitalInfo", {})
        capital_latlng: List[float] = capital_info.get("latlng", [None, None])
        population: int = country_data.get("population", 0)
        extracted_info: Dict[str, Any] = {
            "country_name": country_data.get("name", {}).get("common",
country_name),
```

```
            "capital": capital[0] if capital else "N/A", # Use the first capital
if available
            "population": population,
            "languages": list(languages.values()), # Convert language names to a
list
            "flag_emoji": flag_emoji,
            "flags_images": {
                "png": flags_info.get("png"),
                "svg": flags_info.get("svg"),
                "alt_text": flags_info.get("alt")
            },
            "maps_urls": {
                "googleMaps": maps.get("googleMaps"),
                "openStreetMaps": maps.get("openStreetMaps")
            },
            "capital_coordinates": {
                "latitude": capital_latlng[0],
                "longitude": capital_latlng[1]
            }
        }


        return json.dumps(extracted_info, indent=4)

    except requests.exceptions.HTTPError as e:
        status_code = e.response.status_code
        if status_code == 404:
            return json.dumps({"error": f"Country not found. Please check the
spelling for '{country_name}'."})
        else:
            return json.dumps({"error": f"HTTP Error {status_code}: Could not
retrieve data for '{country_name}'. Details: {e}"})
    except Exception as e:
        return json.dumps({"error": f"An unexpected error occurred: {e}"})



def get_weather_forecast(latitude: float, longitude: float, model: str) -> str:
    """
    Returns the daily maximum and minimum temperature forecasts for the
specified
    location and weather model using the Open-Meteo API.
```

```
    NOTE: The 'latitude' and 'longitude' parameters are REQUIRED.

    Args:
        latitude (float): The latitude for the forecast
        longitude (float): The longitude for the forecast
        model (str): The weather model to use for the forecast. Defaults to
None.

    Returns:
        str: A JSON string containing the weather forecast data.
    """
    try:
       openmeteo = openmeteo_requests.Client()
       url = "https://api.open-meteo.com/v1/forecast"
       params = {
           "latitude": latitude,
           "longitude": longitude,
           "daily": ["temperature_2m_max", "temperature_2m_min"],
           "models": model,
           "forecast_days": 7
       }
       responses = openmeteo.weather_api(url, params=params)
       if not responses:
           return json.dumps({"error": "No response received from the weather
API."})

       response = responses[0]
       daily = response.Daily()
       if daily is None or daily.Variables(0) is None:
           return json.dumps({"error": "Daily data is missing in the API
response."})
       time_range = pd.date_range(
           start = pd.to_datetime(daily.Time(), unit = "s", utc = True),
           end = pd.to_datetime(daily.TimeEnd(), unit = "s", utc = True),
           freq = pd.Timedelta(seconds = daily.Interval()),
           inclusive = "left"
       )
       daily_temperature_2m_max = pd.Series(daily.Variables(0).ValuesAsNumpy(),
index=time_range)
       daily_temperature_2m_min = pd.Series(daily.Variables(1).ValuesAsNumpy(),
index=time_range)
       daily_dataframe = pd.DataFrame({
           "date": daily_temperature_2m_max.index.strftime('%Y-%m-%d'),
           "temperature_max_celsius": daily_temperature_2m_max.values,
```

```
            "temperature_min_celsius": daily_temperature_2m_min.values
        })
        json_output = daily_dataframe.to_json(orient='records', indent=4)
        return json_output
    except Exception as e:
        return json.dumps({"error": f"An error occurred during API call or data
processing: {e}"})


def get_public_holidays(year: int, country_code: str) -> str:
    """
    Fetches the list of public holidays for a specified year and country
    and returns their local and English names.

    Args:
        year (int): The year for which to retrieve the holidays (e.g., 2025).
        country_code (str): The two-letter ISO 3166-1 alpha-2 country code
(e.g., "TR" for Turkey, "US" for USA).

    Returns:
        str: A JSON string containing the date, local name, and English name of
the holidays,
             or an error message.
    """

    url = f"https://date.nager.at/api/v3/PublicHolidays/{year}/{country_code}"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data: List[Dict[str, Any]] = response.json()
        if not data:
            return json.dumps({"result": f"No public holidays found for country
code '{country_code}' in {year}."})
        extracted_holidays: List[Dict[str, str]] = []

        for holiday in data:
            extracted_holidays.append({
                "date": holiday.get("date"),
                "localName": holiday.get("localName"),
                "englishName": holiday.get("name")
            })

        return json.dumps(extracted_holidays, indent=4)
    except requests.exceptions.HTTPError as e:
```

```
        status_code = e.response.status_code

        if status_code == 404:
            return json.dumps({"error": f"Invalid country code or unsupported
year/country combination: '{country_code}' in {year}."})
        else:
            return json.dumps({"error": f"HTTP Error {status_code}: Could not
retrieve holidays. Details: {e}"})

    except Exception as e:
        return json.dumps({"error": f"An unexpected error occurred: {e}"})



def get_current_date() -> str:
    """
    Retrieves the current date, time, and timezone information at the moment
    the function is executed.

    Args:
        None

    Returns:
        str: A JSON string containing the current date and time information.
    """
    try:
        current_datetime_utc = datetime.datetime.now(datetime.timezone.utc)
        current_info: Dict[str, Any] = {
            "current_date": current_datetime_utc.strftime("%Y-%m-%d"),
        }

        return json.dumps(current_info, indent=4)

    except Exception as e:
        return json.dumps({"error": f"An unexpected error occurred while
fetching the current date: {e}"})
```

Finally, integrate these tools into the agent by updating `country_agent/agent.py`.

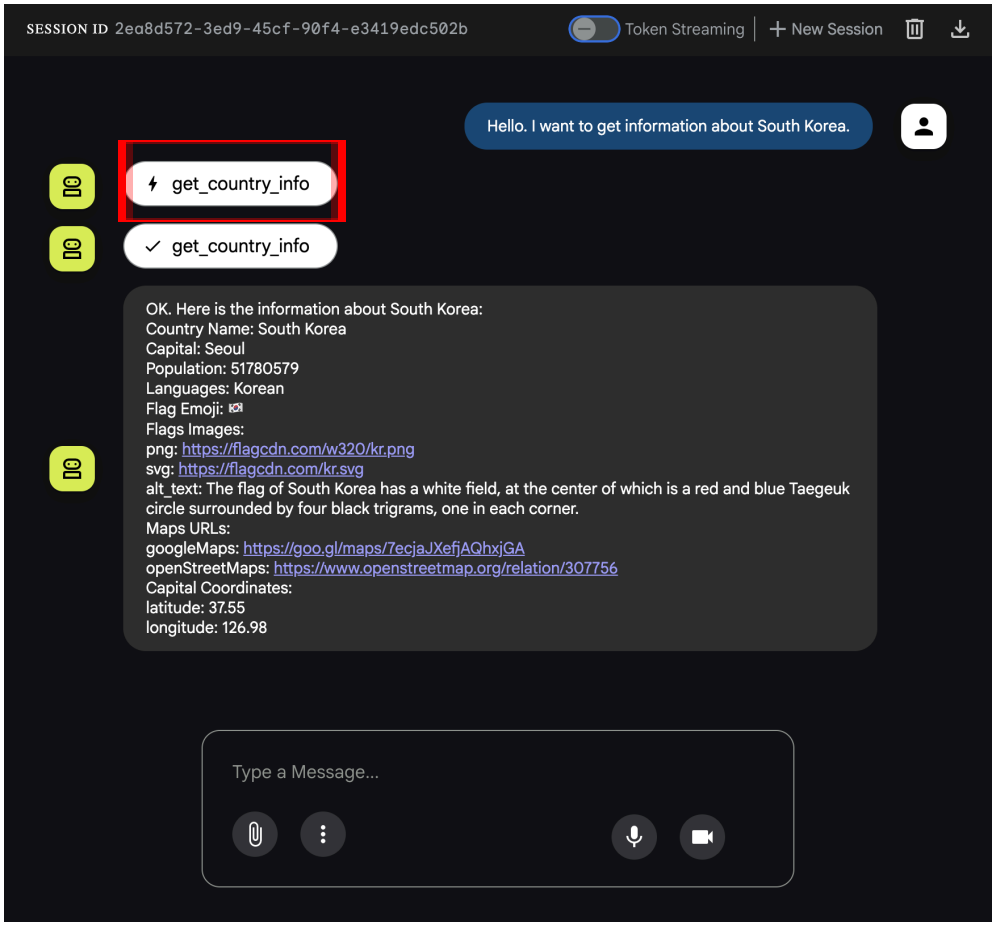
`country_agent/agent.py` (Updated)

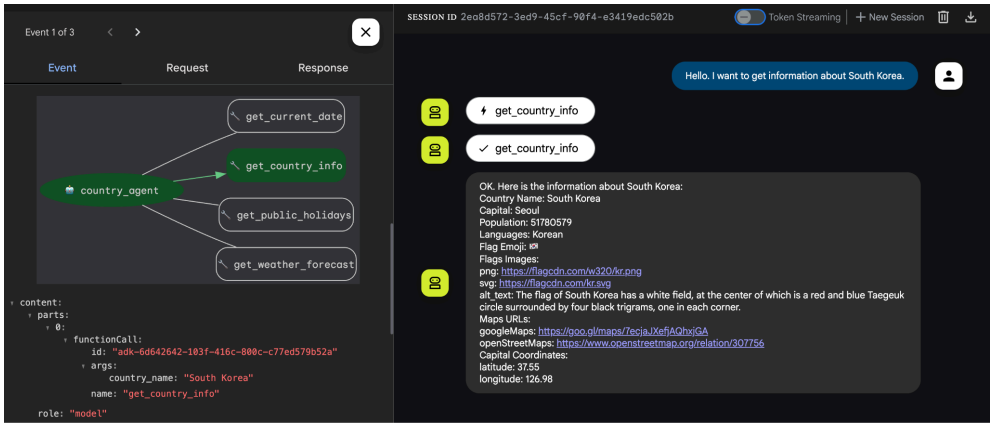
```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction
from .tools import
get_country_info,get_public_holidays,get_weather_forecast,get_current_date ##
=> NEW ADDED

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

country_agent = Agent(
   name="country_agent",
   model="gemini-2.0-flash",
   description="An agent that provides information about the country",
   instruction=country_agent_instruction,

tools=[get_current_date,get_country_info,get_public_holidays,get_weather_foreca
st] ## => NEW ADDED
)

root_agent=country_agent
#We have to define it as a root agent.
```

6. Testing the Tool-Equipped Agent

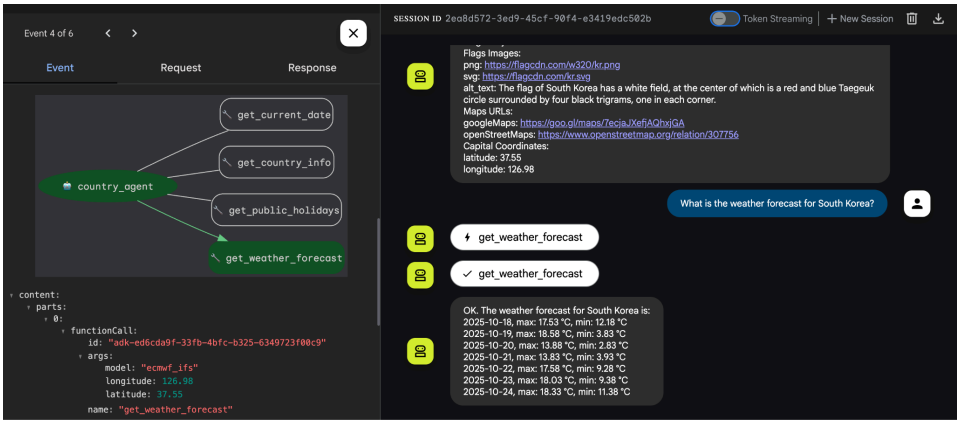
Restart the ADK web server (`adk web`). We can now test the agent's full capabilities.

- **Test Case 1: Country Information**
  - **Query:** "Hello. I want to get information about South Korea."
  - **Result:** The agent will successfully call `get_country_info` and return the data.

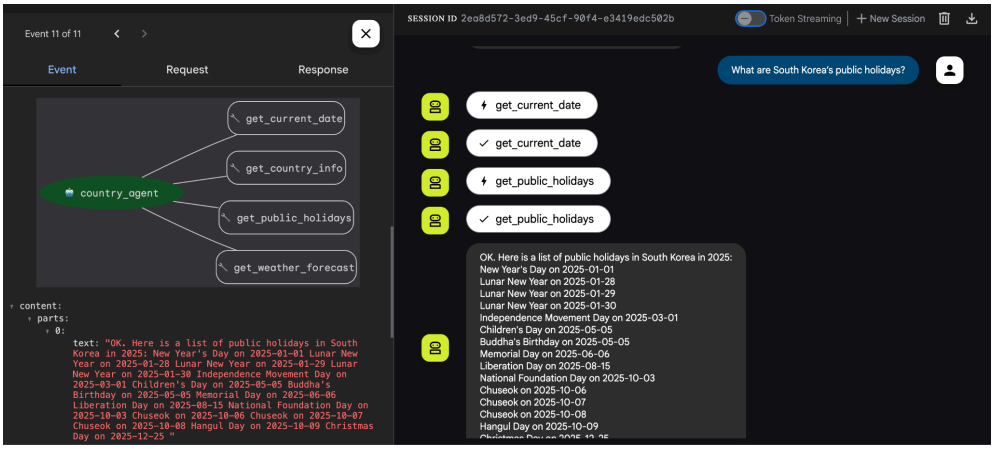


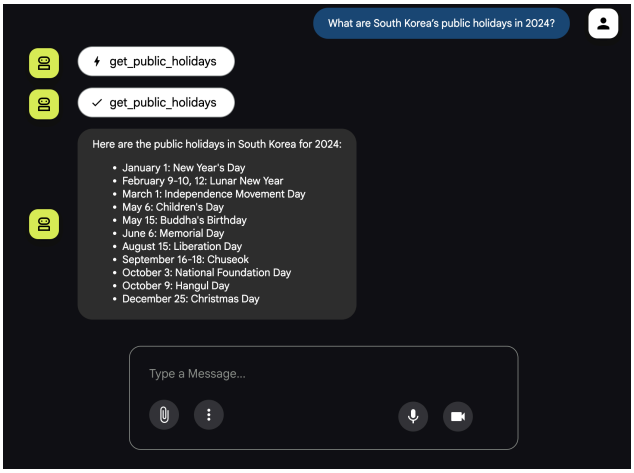


- **Test Case 2: Weather Forecast**
  - **Query:** "What is the weather forecast for South Korea?"
  - **Result:** The agent will first call `get_country_info` (as per instructions) to get coordinates, then call `get_weather_forecast`.



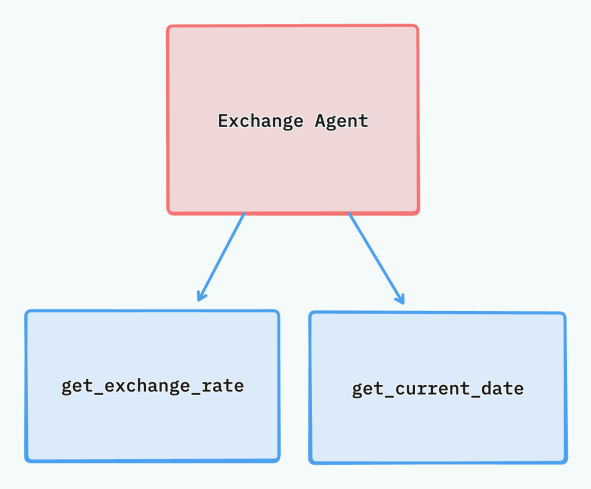
- **Test Case 3: Public Holidays (Dynamic Date)**
  - **Query:** "What are South Korea's public holidays?"
  - **Result:** This query demonstrates tool orchestration. The agent first calls `get_current_date` to determine the current year (e.g., 2025). It then passes this year and the country code (from `get_country_info`) to the `get_public_holidays` tool. This is crucial for dynamic dates like the Lunar New Year, which changes annually.





7. Evolving to a Multi-Agent Architecture

We will now refactor our single-agent system into a multi-agent one. A new, specialized `exchange_agent` will be created to handle currency-related queries and will be registered as a sub-agent of `country_agent`.



First, update `country_agent/agent.py` to define the new agent.

`country_agent/agent.py` (Partial Update)

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction
from .tools import
get_country_info,get_public_holidays,get_weather_forecast,get_current_date

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

 ## => NEW AGENT ADDED
exchange_agent=Agent(
   name="exchange_agent",
   model="gemini-2.0-flash",
   description=(
       "An agent that gives information about the exchange rates"
   )
```

```
)

country_agent = Agent(
   name="country_agent",
   model="gemini-2.0-flash",
   description="An agent that provides information about the country",
   instruction=country_agent_instruction,

tools=[get_current_date,get_country_info,get_public_holidays,get_weather_foreca
st],
sub_agents=[exchange_agent]## => NEW ADDED
)

root_agent=country_agent
#We have to define it as a root agent.
```

Next, add the instructions for the new agent in `country_agent/instructions.py`.

`country_agent/instructions.py` (Appended)

```
Python
country_agent_instruction= “””. . . “””
exchange_agent_instruction="""
"You are an AI Exchange Agent specialized in retrieving current and historical
currency exchange rates. You have access to two tools: 'get_exchange_rate' and
'get_current_date'.

--- EXECUTION LOGIC ---

1.  DETERMINE REQUIREMENTS:
    * Identify the user's request for currency conversion. You MUST extract
three parameters: 'currency_from' (base currency), 'currency_to' (target
currency), and the desired 'currency_date'.
    * All currency codes MUST be 3-letter ISO 4217 codes (e.g., USD, EUR, TRY).

2.  HANDLE DATE PARAMETER:
    * If the user explicitly requests a historical date (e.g., 'What was the
rate on 2023-01-15?'), use that date directly in 'YYYY-MM-DD' format for the
'currency_date' parameter.
    * If the user requests the **latest** rate, or doesn't specify a date
(implying the current rate):
```

```
        * Call 'get_current_date' tool FIRST.
        * Parse the returned JSON to extract the 'current_date' in 'YYYY-MM-DD'
format (e.g., '2025-10-15').
        * Use this extracted date as the 'currency_date' parameter for the
'get_exchange_rate' tool.

3.  CALL EXCHANGE RATE TOOL:
    * TOOL CALL: Call 'get_exchange_rate' using the determined 'currency_from',
'currency_to', and 'currency_date'.

4.  DATA PROCESSING AND OUTPUT:
    * The 'get_exchange_rate' tool returns a JSON object where the exchange
rate is nested under the 'rates' key (e.g., response['rates']['KRW']).
    * CALCULATE RATE: Extract the rate for 'currency_to' from the 'rates'
dictionary. This is the amount of 'currency_to' equivalent to 1 unit of
'currency_from'.
    * OUTPUT: Present the final exchange rate clearly to the user, including
the base currency, target currency, the rate, and the date the rate is valid
for.
    * EXAMPLE OUTPUT FORMAT: 'On [Date], the exchange rate was 1 [Base
Currency] = [Rate] [Target Currency].'"
"""
```

Define the new `get_exchange_rate` tool in `country_agent/tools.py`.

`country_agent/tools.py` (Appended)

Python

```python
## Other Tools . . .
##get_public_holidays
##get_current_date()
##get_weather_forecast
##get_country_info
def get_exchange_rate(
   currency_from: str = "USD",
   currency_to: str = "KRW",
   currency_date: str = "latest", ) -> dict:
   """
   Retrieves the exchange rate between two currencies for a specified date.
   Uses the Frankfurter API (https://api.frankfurter.app/) to fetch exchange
rate data.
```

```
   Args:
   currency_from: Base currency (3-letter currency code). The default is "USD"
(US Dollar).
   currency_to: Target currency (3-letter currency code). The default is "KRW"
(Korean Won).
   currency_date: Date to query the exchange rate for. Default is "latest" for
the most recent rate.
   For historical rates, specify in YYYY-MM-DD format.
   Returns:
   dict: A dictionary containing exchange rate information.
   Example: {"amount": 1.0, "base": "USD", "date": "2023-11-24", "rates":
{"EUR": 0.95534}}
   """

   response = requests.get(
       f"https://api.frankfurter.app/{currency_date}",
       params={"from": currency_from, "to": currency_to},
 )
   return response.json()
```

Finally, update `country_agent/agent.py` to import all new components and assign them to the `exchange_agent`.

`country_agent/agent.py` (Final Update for this Step)

Python

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction, exchange_agent_instruction
## => NEW ADDED
from .tools import
get_country_info,get_public_holidays,get_weather_forecast,get_current_date
,get_exchange_rate ## => NEW ADDED

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

exchange_agent=Agent(
   name="exchange_agent",
   model="gemini-2.0-flash",
```

```
    description=(
        "An agent that gives information about the exchange rates"
    ),
    instruction=exchange_agent_instruction, ## => NEW AGENT ADDED
    tools=[get_exchange_rate,get_current_date] ## => NEW AGENT ADDED
)

country_agent = Agent(
    name="country_agent",
    model="gemini-2.0-flash",
    description="An agent that provides information about the country",
    instruction=country_agent_instruction,

tools=[get_current_date,get_country_info,get_public_holidays,get_weather_foreca
st],
    sub_agents=[exchange_agent]
)

root_agent=country_agent
#We have to define it as a root agent.
```

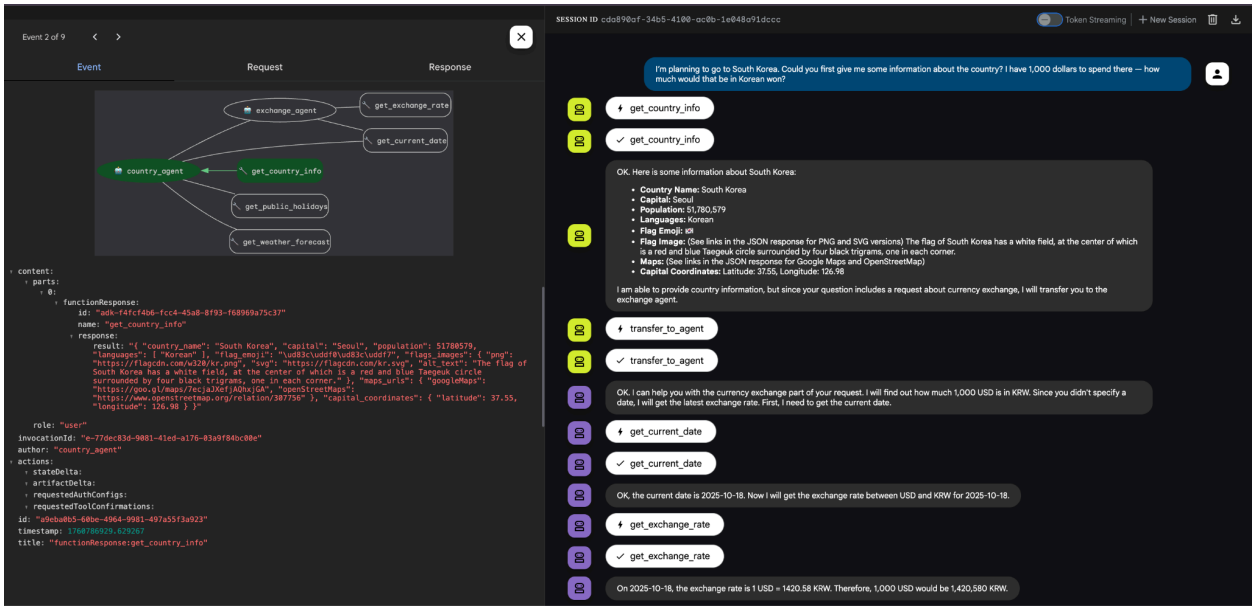
8. Testing the Multi-Agent System

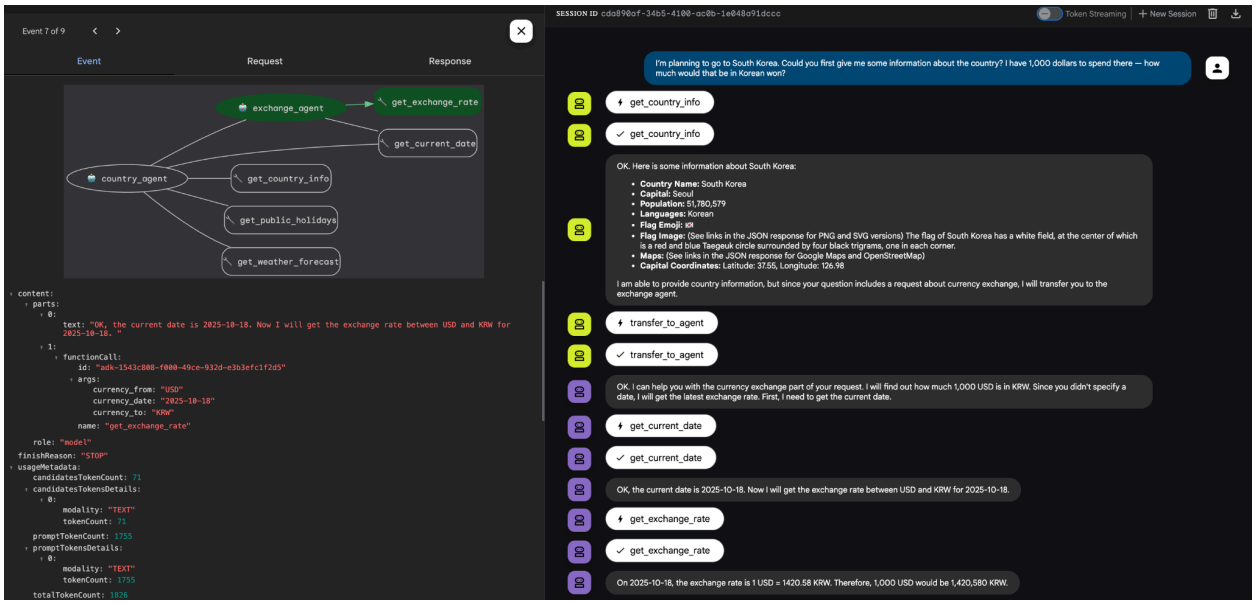
Launch the web interface (`adk web`). The agent schema in the UI will now show `country_agent` with `exchange_agent` as a sub-agent.

Test the delegation logic with a compound query:

> "I'm planning to go to South Korea. Could you first give me some information about the country? I have 1,000 dollars to spend there — how much would that be in Korean won?"

The `country_agent` will handle the information request, then delegate the currency question to the `exchange_agent`.





9. Centralizing Tools with an MCP Server

To make our tools standardized and reusable, we will refactor them into a Model-Compute-Protocol (MCP) server using FastMCP.

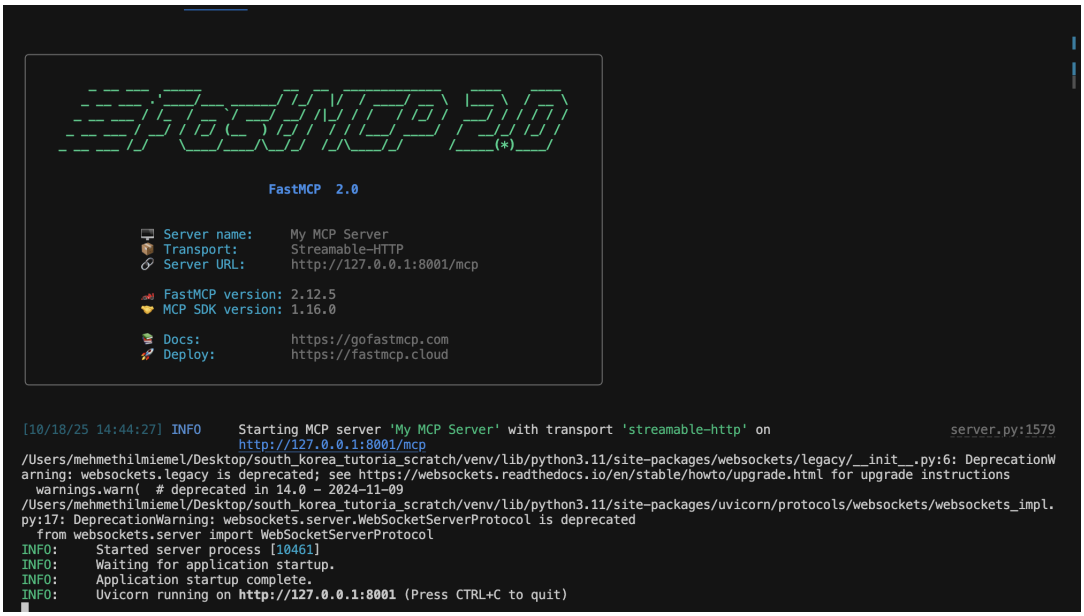
Create `my_mcp_server.py` in the project's root directory. Copy all tool functions from `country_agent/tools.py` into this file and add the `@mcp.tool` decorator to each.

my_mcp_server.py

```python
from fastmcp import FastMCP
import datetime
import openmeteo_requests
import pandas as pd
import json
from typing import Optional, Dict, Any, List
import requests

mcp = FastMCP("My MCP Server")

@mcp.tool #=> MCP Decorator
def get_country_info(country_name: str) -> str:
    # ... (Full function code from tools.py)
    ...
@mcp.tool #=> MCP Decorator
def get_weather_forecast(latitude: float, longitude: float, model: str) -> str:
    # ... (Full function code from tools.py)
    ...
@mcp.tool #=> MCP Decorator
def get_public_holidays(year: int, country_code: str) -> str:
    # ... (Full function code from tools.py)
    ...
@mcp.tool #=> MCP Decorator
def get_current_date() -> str:
    # ... (Full function code from tools.py)
    ...
@mcp.tool #=> MCP Decorator
def get_exchange_rate(
    currency_from: str = "USD",
    currency_to: str = "KRW",
    currency_date: str = "latest", ) -> dict:
    # ... (Full function code from tools.py)
    ...

if __name__ == "__main__":
    mcp.run(transport="streamable-http", port=8001) # Run on port 8001
```

Run the MCP server in a **new, dedicated terminal**:

```
None
python my_mcp_server.py
```



Now, update `country_agent/agent.py` to point to the MCP server instead of using local tools.

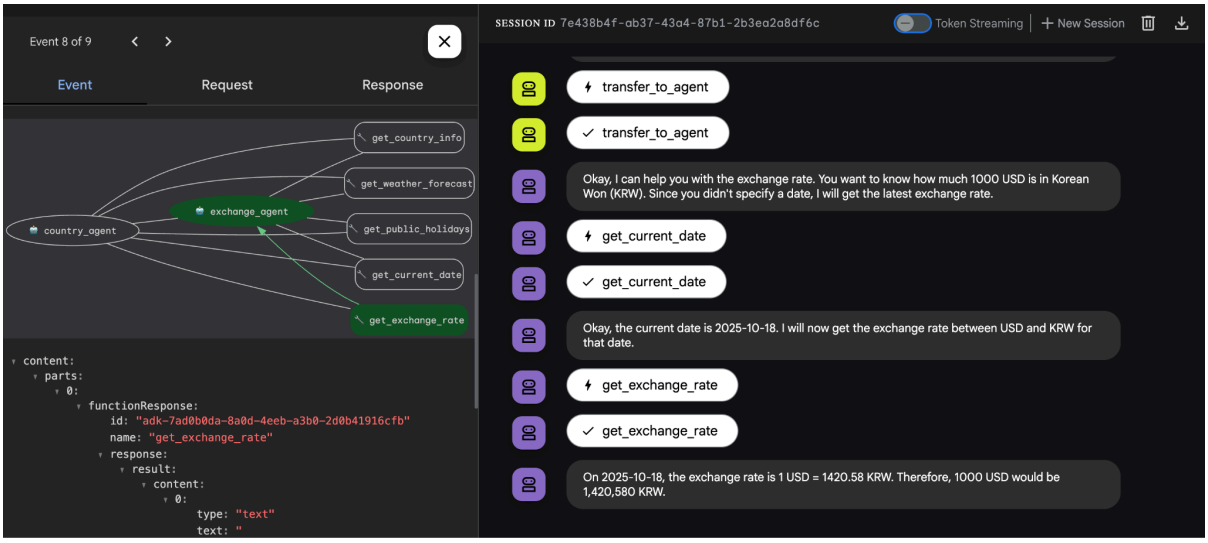
`country_agent/agent.py` (Updated)

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction, exchange_agent_instruction
from google.adk.tools.mcp_tool import MCPToolset,
StreamableHTTPConnectionParams ## => NEW ADDED
# from .tools import ... (Local imports are no longer needed)

## => NEW Toolset Added
tool_set=MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url="http://localhost:8001/mcp" ## => Our MCP Server
    ),
)
```

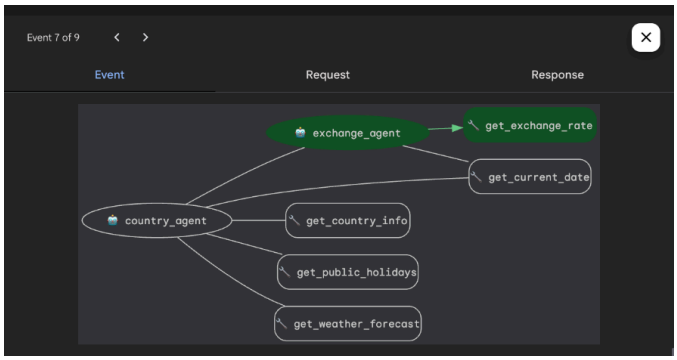
```
load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

exchange_agent=Agent(
    name="exchange_agent",
    model="gemini-2.0-flash",
    description=(
        "An agent that gives information about the exchange rates"
    ),
    instruction=exchange_agent_instruction,
    tools=[tool_set] ## => Tools updated
)
country_agent = Agent(
    name="country_agent",
    model="gemini-2.0-flash",
    description="An agent that provides information about the country",
    instruction=country_agent_instruction,
    tools=[tool_set], ## => Tools updated
    sub_agents=[exchange_agent]
)
root_agent=country_agent
```

9.1. Addressing the MCP Architecture Security Vulnerability

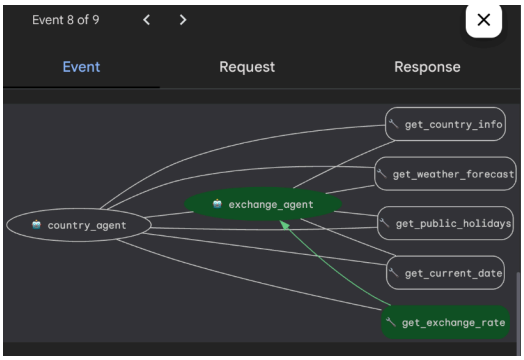
While we have successfully centralized our tools on an MCP server, this new architecture introduces a significant security vulnerability. Currently, both the `country_agent` and the `exchange_agent` connect to the *same* MCP server (`tool_set`), which means both agents have access to *all* tools hosted on it.



Before MCP



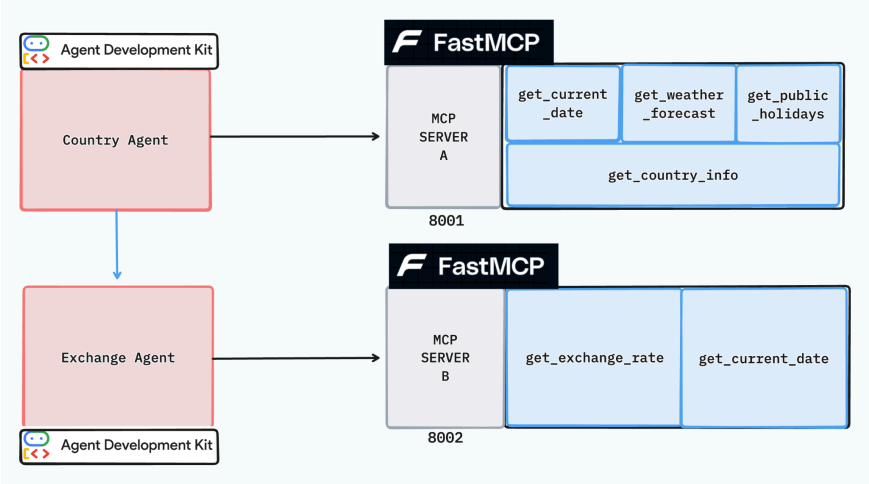
After MCP



For example, the `country_agent` could now potentially invoke the `get_exchange_rate` tool, violating our design principles.

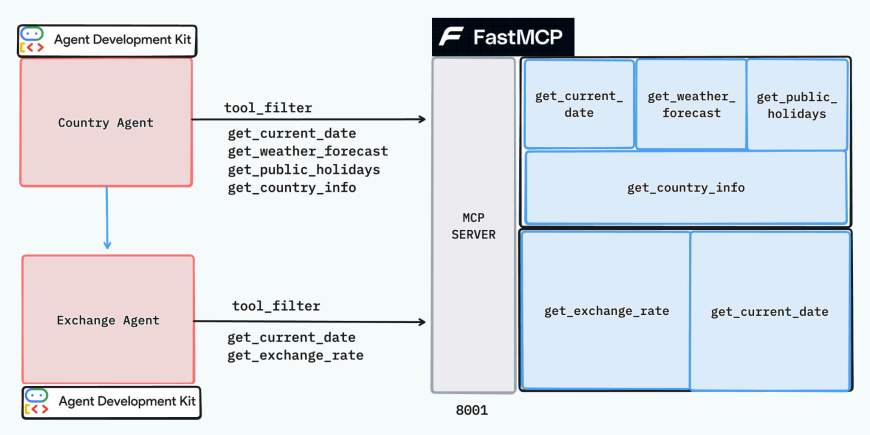
We have several options for addressing this scenario:

1. **Instruction-Based Restriction (Insecure):** We could attempt to add an instruction to the `country_agent` stating, "Do not ever use the `get_exchange_rate` tool." However, this approach is not secure. It is highly vulnerable to prompt injection techniques, where a user could trick the agent into bypassing its restrictions.
2. **Creating Multiple MCP Servers (Inefficient):** A second option is to create separate, isolated MCP servers for each agent. For example:
    - **MCP Server A (Port 8001):** Hosts only the `country_agent` tools (`get_country_info`, `get_public_holidays`, `get_weather_forecast`, `get_current_date`).
    - **MCP Server B (Port 8002):** Hosts only the `exchange_agent` tools (`get_exchange_rate`, `get_current_date`).



This method, however, is inefficient and scales poorly. A system with many specialized agents would require a separate MCP server for each, dramatically increasing maintenance overhead. Furthermore, this approach is not viable if we want to consume tools from third-party MCP servers.

3. **MCPToolset Filtering (Preferred Solution):** A far superior, more secure, and scalable approach is to continue using a single MCP server but apply a `tool_filter` within the `MCPToolset` definition for each agent. This allows us to enforce which tools are available to which agent at the connection level. This is the method we will implement in the next section.



10. Implementing Secure Tool Access with Filtering

This new architecture creates a security concern: both agents can now access *all* tools on the MCP server. We will fix this by applying a `tool_filter` to each agent's `MCPToolset`.

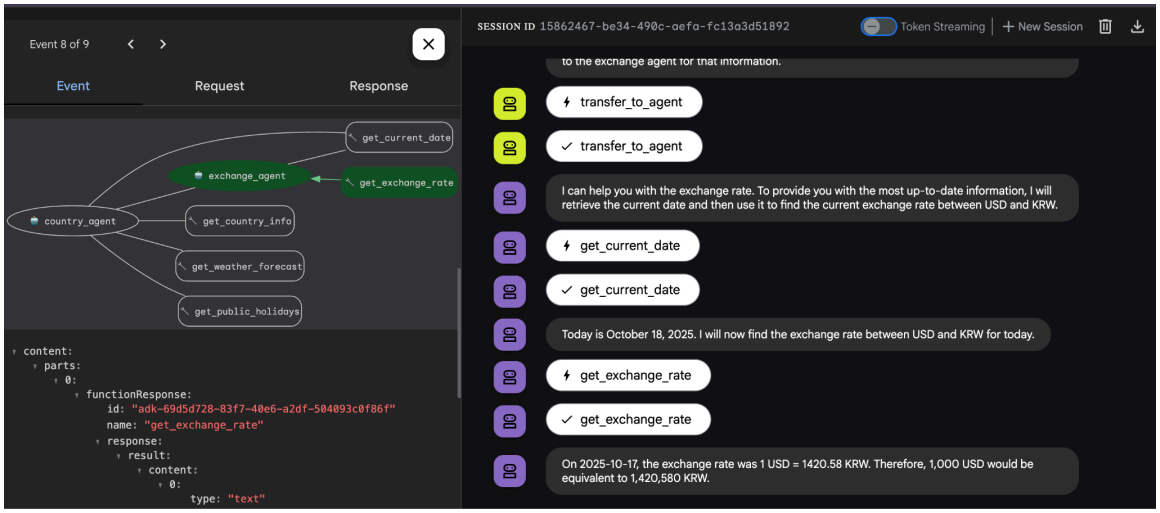
Update `country_agent/agent.py` to define two distinct, filtered toolsets.

`country_agent/agent.py` (Updated)

Python

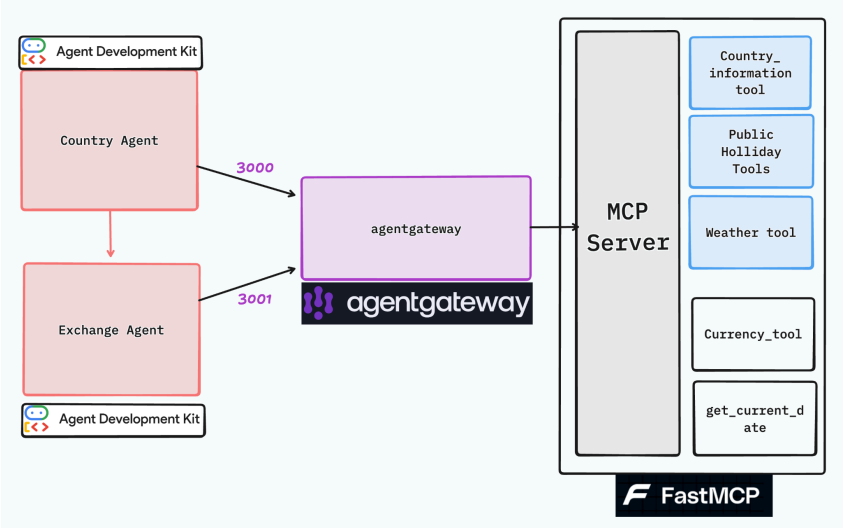
```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction,
exchange_agent_instruction
from google.adk.tools.mcp_tool import MCPToolset,
StreamableHTTPConnectionParams

## => NEW country_agent_tool_set Added
country_agent_tool_set=MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url="http://localhost:8001/mcp"
    ),

tool_filter=["get_country_info","get_public_holidays","get_weather_foreca
st","get_current_date"]
)

## => NEW exchange_agent_tool_set Added
exchange_agent_tool_set=MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url="http://localhost:8001/mcp"
    ),
    tool_filter=["get_current_date","get_exchange_rate"]
)

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

exchange_agent=Agent(
    name="exchange_agent",
    # ... (rest of definition)
    instruction=exchange_agent_instruction,
    tools=[exchange_agent_tool_set] ## => tools updated
)

country_agent = Agent(
    name="country_agent",
    # ... (rest of definition)
    instruction=country_agent_instruction,
    tools=[country_agent_tool_set], ## => tools updated
```

```
    sub_agents=[exchange_agent]
)

root_agent=country_agent
```



11. Managing Communication with AgentGateway

For a production-grade solution, we introduce **AgentGateway** for security, governance, and observability. Agentgateway connects AI agents, MCP tool servers, and LLM providers in any environment. These two-way connections are secure, scalable, and stateful.



First, install AgentGateway (in a new terminal):

```
None
curl
https://raw.githubusercontent.com/agentgateway/agentgateway/refs/heads/main/com
mon/scripts/get-agentgateway | bash
```

Create `agent_gateway_config.yaml` in the project root. This configuration will create two policy-controlled endpoints (ports 3000 and 3001) that route to our single MCP server (port 8001), each with specific tool authorization rules.

agent_gateway_config.yaml

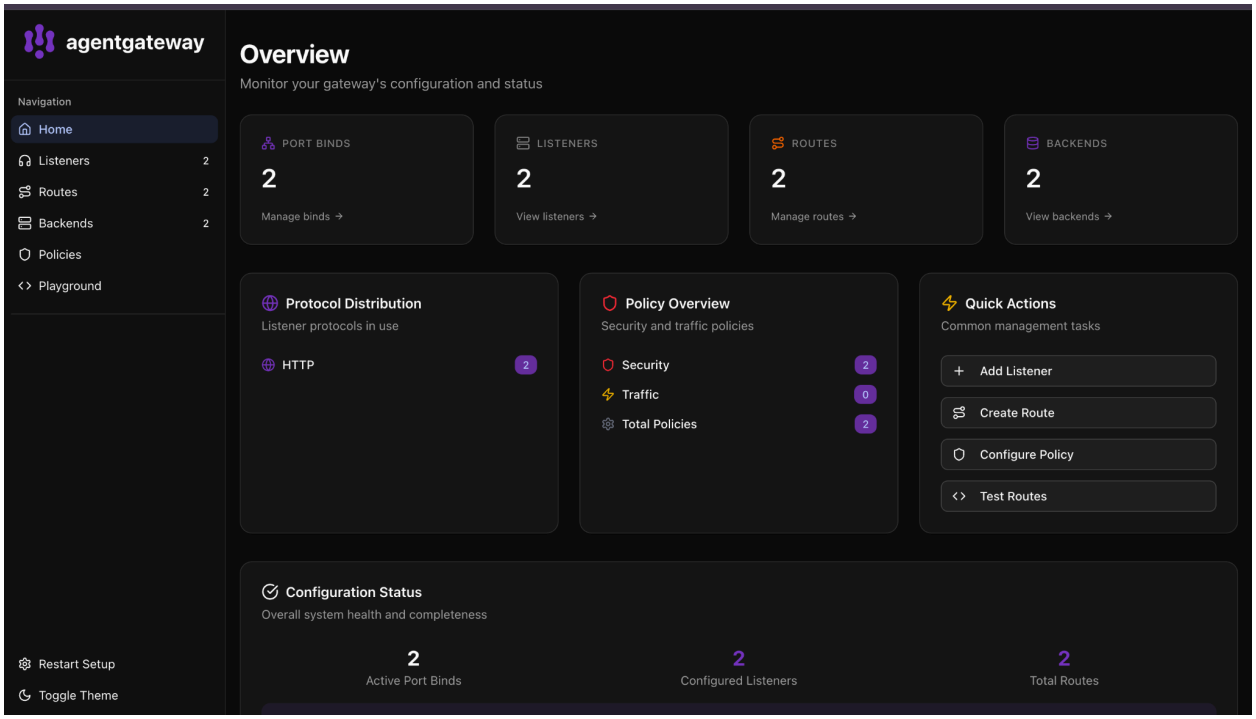
```
None
binds:
- port: 3000
  listeners:
  - protocol: HTTP
    name: country_agent
    routes:
    - hostnames: []
      matches:
      - path:
          pathPrefix: /
      backends:
      - mcp:
          targets:
          - name: mcp
            mcp:
              host: 127.0.0.1
              port: 8001
              path: /mcp
      name: country-agent-route
      policies:
        cors:
          allowOrigins:
          - '*'
          allowHeaders:
          - '*'
        mcpAuthorization:
          rules:
```

```
          - mcp.tool.name == "get_country_info" || mcp.tool.name ==
"get_weather_forecast" || mcp.tool.name == "get_public_holidays" ||
mcp.tool.name == "get_current_date"
- port: 3001
  listeners:
  - protocol: HTTP
    name: exchange_agent
    routes:
    - hostnames: []
      matches:
      - path:
          pathPrefix: /
      backends:
      - mcp:
          targets:
          - name: mcp
            mcp:
              host: 127.0.0.1
              port: 8001
              path: /mcp
      name: exchage_agent_route
      policies:
        cors:
          allowOrigins:
          - '*'
          allowHeaders:
          - '*'
        mcpAuthorization:
          rules:
          - mcp.tool.name == "get_exchange_rate" || mcp.tool.name ==
"get_current_date"
```

Start the AgentGateway service (in another terminal):

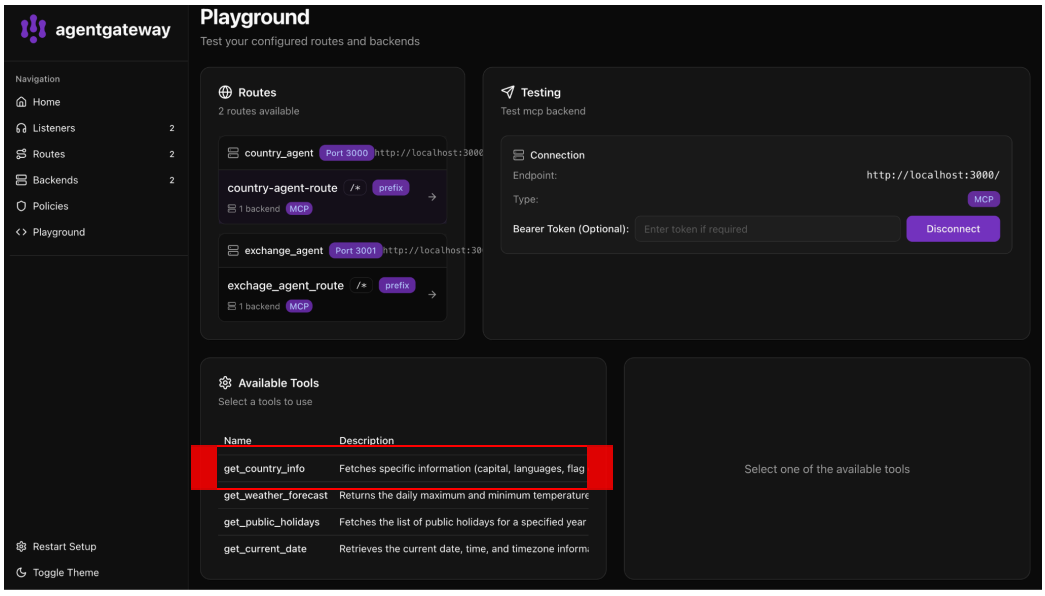
```
None
agentgateway -f agent_gateway_config.yaml
```

You can explore the AgentGateway UI at http://localhost:15000

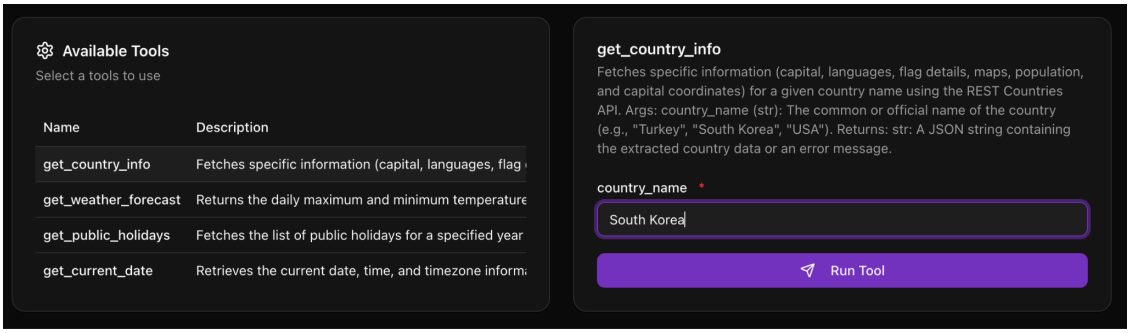


Before connecting our agent system with our MCP server, we can click **Playground** in the AgentGateway page to test the connections.

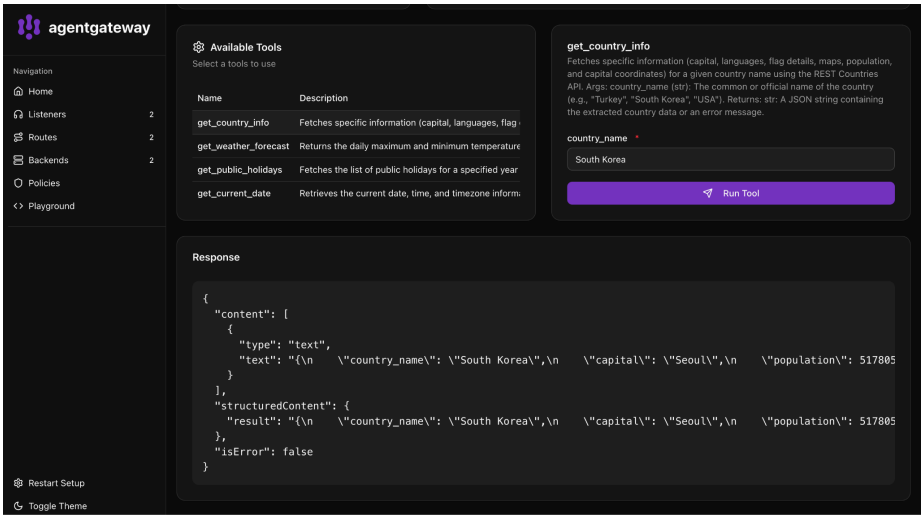
In the playground page, there are two routes: `country_agent` (port:3000) and `exchange_agent` (port:3001). If we click “Connect” on the right side of the `country_agent` route, we can see the available tools authorized for that agent.



If you click the `get_country_info` tool, you can test it by writing “South Korea” as the `country_name` parameter.



After executing, you can see this tool’s response directly. This confirms our gateway policies are correctly configured and routing requests.



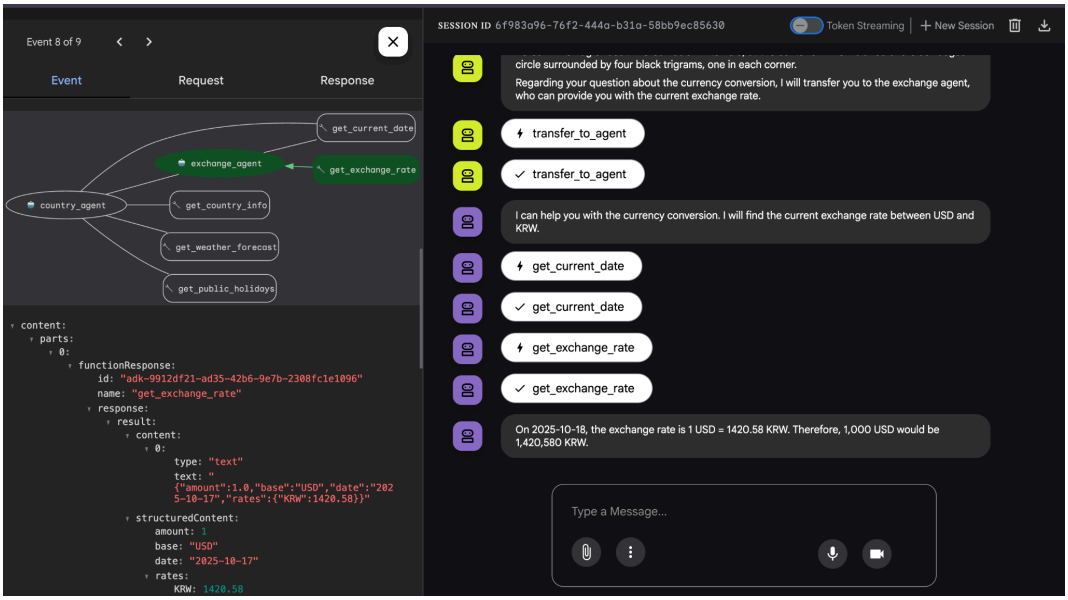
Now that we have verified the gateway is working, we will update `country_agent/agent.py` to point to the new AgentGateway URLs instead of the direct MCP URL.

`country_agent/agent.py` (Updated URLs)

```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction, exchange_agent_instruction
from google.adk.tools.mcp_tool import MCPToolset,
StreamableHTTPConnectionParams
```

```
#from .tools import
get_country_info,get_public_holidays,get_weather_forecast,get_current_date
,get_exchange_rate

country_agent_tool_set=MCPToolset(
   connection_params=StreamableHTTPConnectionParams(
       url="http://localhost:3000" ## => Agentgateway url
   ),
)

exchange_agent_tool_set=MCPToolset(
   connection_params=StreamableHTTPConnectionParams(
       url="http://localhost:3001" ## => Agentgateway url
   ),
)

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")


exchange_agent=Agent(
   name="exchange_agent",
   model="gemini-2.0-flash",
   description=(
       "An agent that gives information about the exchange rates"
   ),
   instruction=exchange_agent_instruction,
   tools=[exchange_agent_tool_set]
)

country_agent = Agent(
   name="country_agent",
   model="gemini-2.0-flash",
   description="An agent that provides information about the country",
   instruction=country_agent_instruction,
   tools=[country_agent_tool_set],
   sub_agents=[exchange_agent]
)

root_agent=country_agent
#We have to define it as a root agent.
```

All communication is now securely proxied by AgentGateway.



12. Publishing the Agent as an A2A Server

We can now publish our entire multi-agent system as a single A2A (Agent-to-Agent) server, making it consumable by other agents.

Update `country_agent/agent.py` to import `to_a2a` and expose the `root_agent`.

`country_agent/agent.py` (Appended)

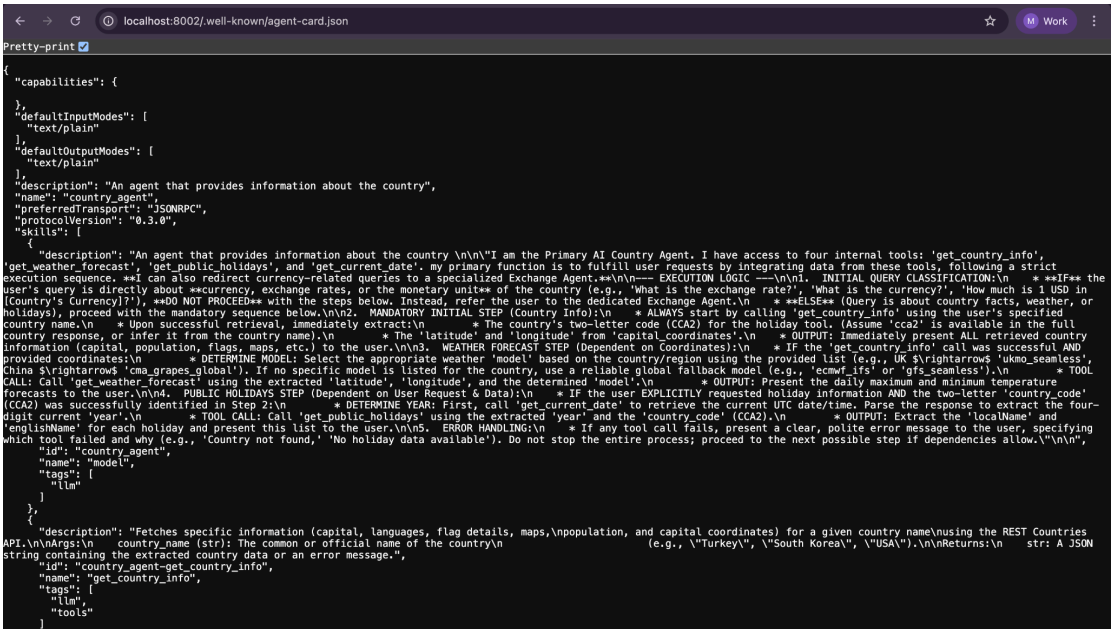
```python
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from .instructions import country_agent_instruction, exchange_agent_instruction
from google.adk.tools.mcp_tool import MCPToolset,
StreamableHTTPConnectionParams
#from .tools import
get_country_info,get_public_holidays,get_weather_forecast,get_current_date
,get_exchange_rate
from google.adk.a2a.utils.agent_to_a2a import to_a2a ## => New Added

country_agent_tool_set=MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url="http://localhost:3000"
    ),
)
```

```
exchange_agent_tool_set=MCPToolset(
   connection_params=StreamableHTTPConnectionParams(
       url="http://localhost:3001"
   ),
)

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")


exchange_agent=Agent(
   name="exchange_agent",
   model="gemini-2.0-flash",
   description=(
       "An agent that gives information about the exchange rates"
   ),
   instruction=exchange_agent_instruction,
   tools=[exchange_agent_tool_set]
)

country_agent = Agent(
   name="country_agent",
   model="gemini-2.0-flash",
   description="An agent that provides information about the country",
   instruction=country_agent_instruction,
   tools=[country_agent_tool_set],
   sub_agents=[exchange_agent]
)

root_agent=country_agent

country_agent_server=to_a2a(root_agent,port=8002) ## => New Added. Our server
agent will run on port 8002
```

Run this A2A server using Uvicorn (in another terminal):

```
None
uvicorn country_agent.agent:country_agent_server --host localhost --port 8002
```

The agent's capabilities are now public at `http://localhost:8002/.well-known/agent-card.json`.



13. Applying AgentGateway to the A2A Server

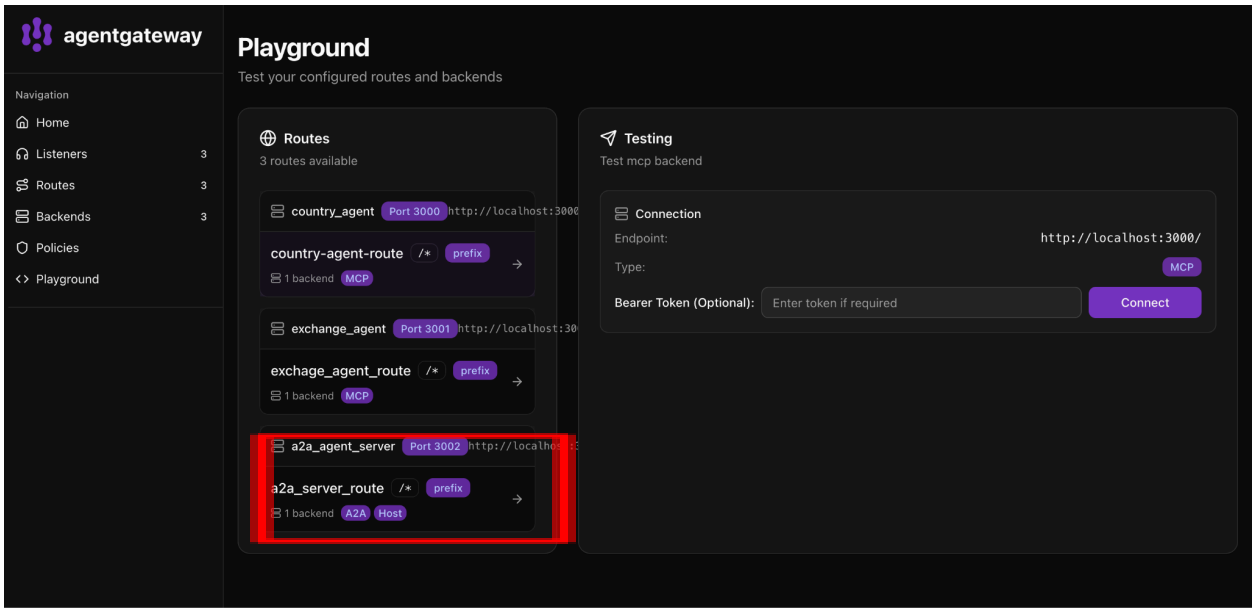
For governance, we can also place this new A2A server behind AgentGateway. Add the following to `agent_gateway_config.yaml` to create an endpoint on port 3002.

`agent_gateway_config.yaml` (Appended)

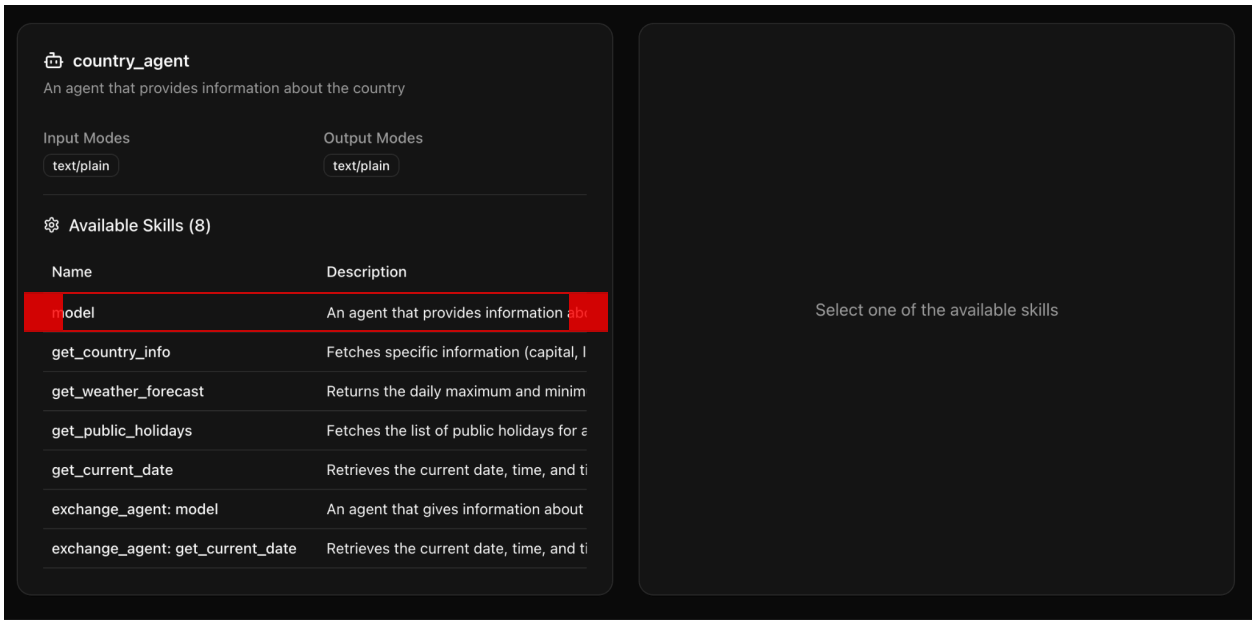
```
None
# ... (binds for port 3000 and 3001)

- port: 3002
  listeners:
  - protocol: HTTP
    name: a2a_agent_server
    hostname: localhost
    routes:
    - hostnames: []
      matches:
      - path:
          pathPrefix: /
```

```
backends:
- host: localhost:8002
name: a2a_server_route
policies:
  a2a: {}
  cors:
    allowCredentials: false
    allowHeaders: ['*']
    allowMethods: []
    allowOrigins: ['*']
    exposeHeaders: []
    maxAge: null
```

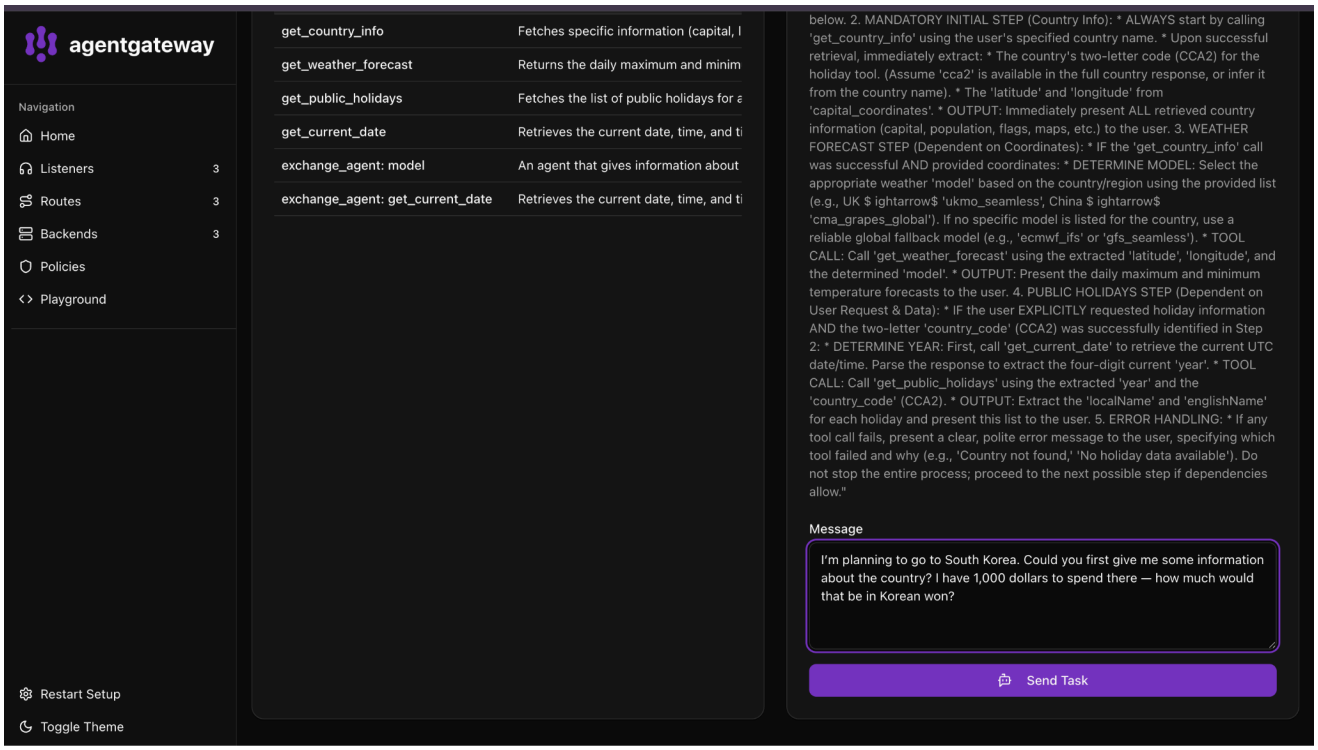
Restart AgentGateway. The UI at `http://localhost:15000` will now show the `a2a_agent_server` endpoint, allowing you to test the entire system as a single remote entity. The A2A server (highlighted with a red rectangle) is visible within the interface.

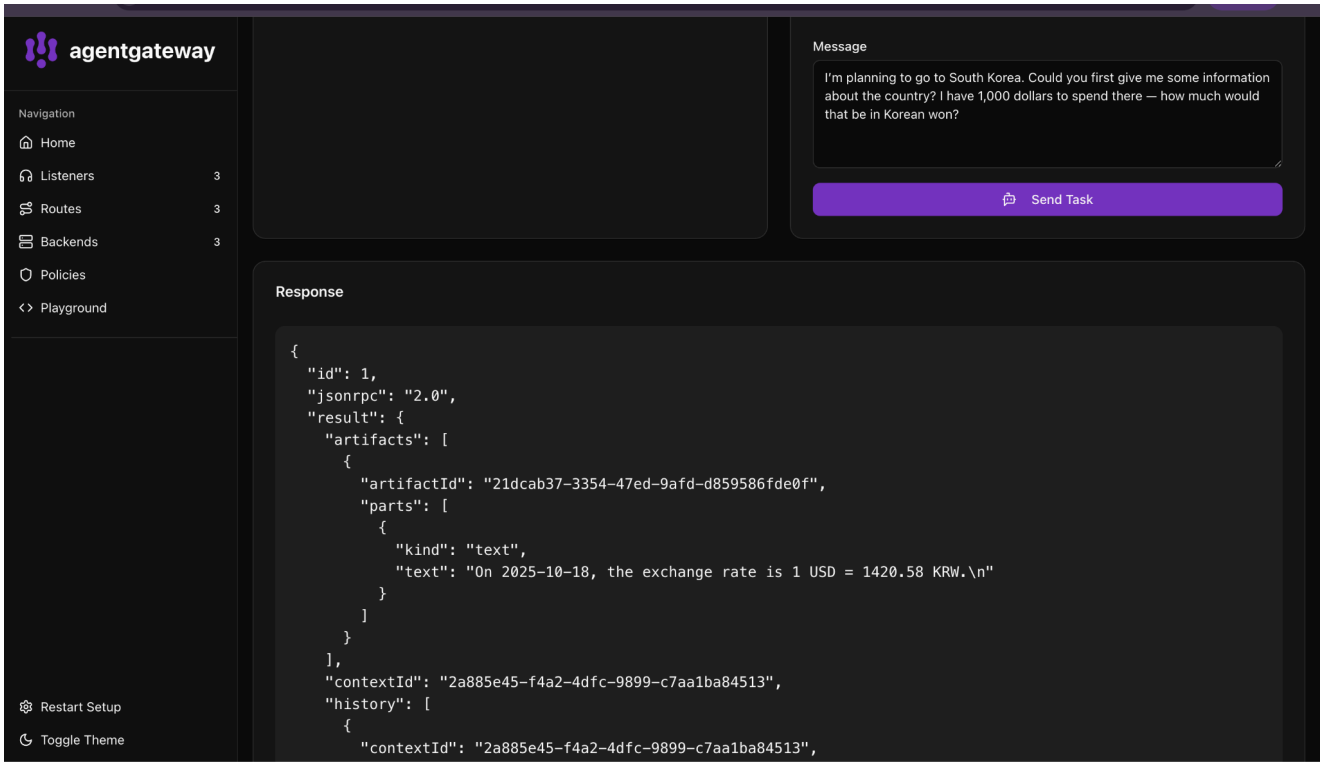


By selecting the server and subsequently clicking “Connect”, users can access the agent’s available capabilities.

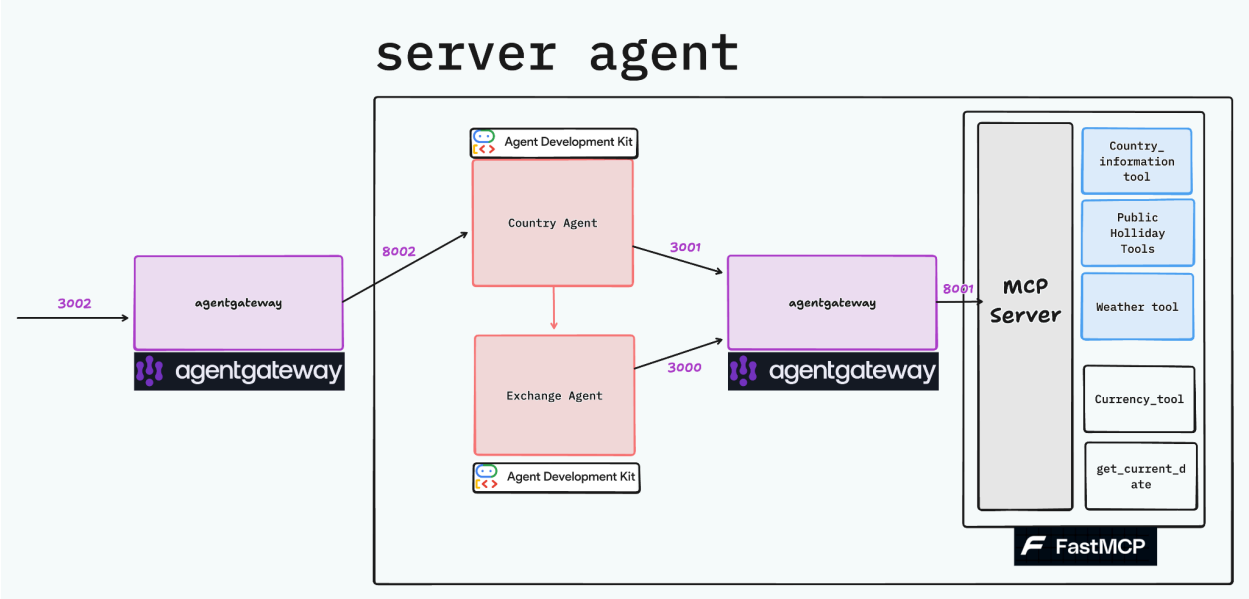


Additionally, by selecting the model (also indicated with a red rectangle), it is possible to test the agent's functionality.
For example, a query such as: "I'm planning to go to South Korea. Could you first provide some information about the country? I have 1,000 dollars to spend there — how much would that be in Korean won?" can be submitted, and the system will return a corresponding response.



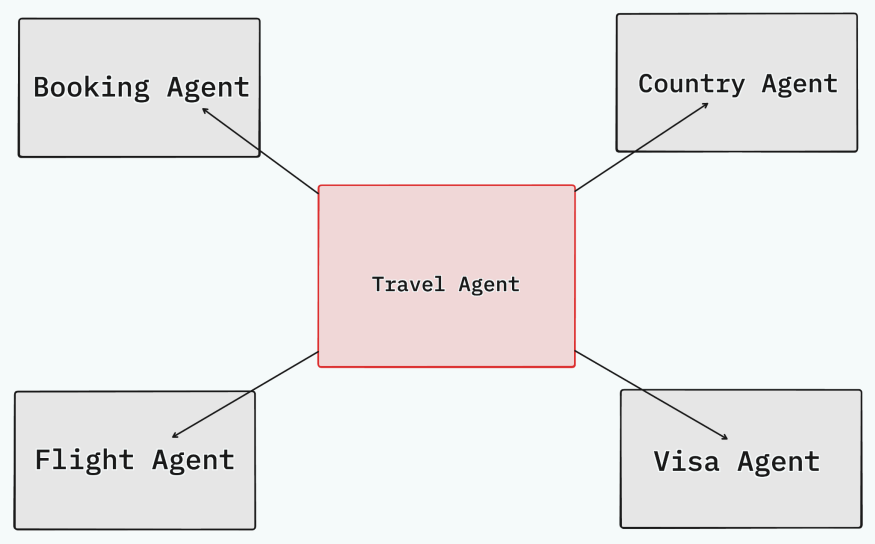


At this stage, the system design has been established as depicted below.



14. Consuming the Remote Agent (A2A Client)

We will now create a new agent that will operate on **port 8000**. This agent will connect to our existing A2A server (**country_agent_server**) to generate responses. To organize the project, a new folder named `travel_agent` will be created in the project root directory. The objective is to develop a complex AI agent capable of assisting users with travel-related inquiries, such as providing information about a country, purchasing flight tickets, or booking hotels.



Since a **Country Agent** has already been implemented to provide country-specific information, there is no need to redevelop it from scratch. The **Travel Agent** will act as a client, sending requests to the Country Agent server and retrieving the necessary information.

The proposed project structure will be as follows:

|venv
|travel_agent
|country_agent
    |__init__.pt
    |.env
    |agent.py
    |instructions.py
    |tools.py
|my_mcp_server.py
|agent_gateway_config.yaml

Within the `travel_agent` folder, three initial files will be created: `__init__.py`, `agent.py`, and `.env`. These files will serve as the foundation for implementing the Travel Agent's functionality.

Populate the `travel_agent` files:

travel_agent/__init__.py

```python
from . import agent
```

travel_agent/.env

```
GOOGLE_API_KEY="ENTER YOUR API KEY"
```

travel_agent/agent.py

```python
import os
from google.adk.agents import Agent
from dotenv import load_dotenv
from google.adk.agents.remote_a2a_agent import RemoteA2aAgent ## => New Added

load_dotenv()
GOOGLE_API_KEY=os.getenv("GOOGLE_API_KEY")

## => New Added. Defines the remote agent server.
remote_country_agent = RemoteA2aAgent(
    name="country_agent",
    description="An agent that gives information about the country",
    agent_card=(
        f"http://localhost:3002" ## => AgentGateway URL for A2A server
    ),
)

travel_agent = Agent(
    name="travel_agent",
    model="gemini-2.0-flash",
    description=(
        "A smart travel assistant that connects users with specialized
sub-agents..."
    ),
    instruction=("""
```

```
        You are a travel assistant. When a user's request relates to flights,
bookings, visas, or country-specific information, delegate the conversation to
the appropriate sub-agent:
        ...
        country_agent for local information, culture, and travel tips
        """),
    sub_agents=[remote_country_agent] ## => New Added
)

root_agent=travel_agent
```

15. Final System Test

Ensure all three services are running in their respective terminals:

1. MCP Server (`python my_mcp_server.py`)
2. AgentGateway (`agentgateway -f agent_gateway_config.yaml`)
3. Country Agent A2A Server (`uvicorn country_agent.agent:country_agent_server ...`)

Now, run the new `travel_agent` (in a new terminal):

```
None
adk web
```

Navigate to `http://127.0.0.1:8000` and select `travel_agent`. You can now ask it questions that it will delegate to the remote `country_agent`:

- "What is the flag of South Korea?"
- "What is the weather prediction for South Korea?"
- "What are the public holidays in South Korea?"
- "How much is 1 US dollar in Korean won?"

The `travel_agent` successfully proxies these requests to the `remote_country_agent`, which in turn orchestrates its own sub-agents and tools via AgentGateway to generate the final, comprehensive answer.