Session: Tracking Individual Conversations

Following our Introduction, let's dive into the `Session`. Think back to the idea of a "conversation thread." Just like you wouldn't start every text message from scratch, agents need context regarding the ongoing interaction. **`Session`** is the ADK object designed specifically to track and manage these individual conversation threads.

The `Session` Object

When a user starts interacting with your agent, the `SessionService` creates a `Session` object (`google.adk.sessions.Session`). This object acts as the container holding everything related to that *one specific chat thread*. Here are its key properties:

- **Identification (`id`, `appName`, `userId`):** Unique labels for the conversation.
    - `id`: A unique identifier for *this specific* conversation thread, essential for retrieving it later. A SessionService object can handle multiple `Session`(s). This field identifies which particular session object are we referring to. For example, "test_id_modification".
    - `app_name`: Identifies which agent application this conversation belongs to. For example, "id_modifier_workflow".
    - `userId`: Links the conversation to a particular user.
- **History (`events`):** A chronological sequence of all interactions (`Event` objects – user messages, agent responses, tool actions) that have occurred within this specific thread.
- **Session State (`state`):** A place to store temporary data relevant *only* to this specific, ongoing conversation. This acts as a scratchpad for the agent during the interaction. We will cover how to use and manage `state` in detail in the next section.

- **Activity Tracking (`lastUpdateTime`):** A timestamp indicating the last time an event occurred in this conversation thread.

Example: Examining Session Properties

Python

```python
from google.adk.sessions import InMemorySessionService,
Session

# Create a simple session to examine its properties
temp_service = InMemorySessionService()
example_session = await temp_service.create_session(
    app_name="my_app",
    user_id="example_user",
    state={"initial_key": "initial_value"} # State can be
initialized
)

print(f"--- Examining Session Properties ---")
print(f"ID (`id`):                {example_session.id}")
print(f"Application Name (`app_name`):
{example_session.app_name}")
print(f"User ID (`user_id`):
{example_session.user_id}")
print(f"State (`state`):           {example_session.state}") #
Note: Only shows initial state here
print(f"Events (`events`):         {example_session.events}")
# Initially empty
print(f"Last Update (`last_update_time`):
{example_session.last_update_time:.2f}")
print(f"---------------------------------")

# Clean up (optional for this example)
temp_service = await
temp_service.delete_session(app_name=example_session.app_name,
                             user_id=example_session.user_id,
session_id=example_session.id)
print("The final status of temp_service - ", temp_service)
```

Java

```java
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;

String sessionId = "123";
String appName = "example-app"; // Example app name
String userId = "example-user"; // Example user id
```

```
 ConcurrentMap<String, Object> initialState = new
ConcurrentHashMap<>(Map.of("newKey", "newValue"));
 InMemorySessionService exampleSessionService = new
InMemorySessionService();

 // Create Session
 Session exampleSession = exampleSessionService.createSession(
     appName, userId, initialState,
Optional.of(sessionId)).blockingGet();
 System.out.println("Session created successfully.");

 System.out.println("--- Examining Session Properties ---");
 System.out.printf("ID (`id`): %s%n", exampleSession.id());
 System.out.printf("Application Name (`appName`): %s%n",
exampleSession.appName());
 System.out.printf("User ID (`userId`): %s%n",
exampleSession.userId());
 System.out.printf("State (`state`): %s%n",
exampleSession.state());
 System.out.println("------------------------------------");


 // Clean up (optional for this example)
 var unused = exampleSessionService.deleteSession(appName,
userId, sessionId);
```

(Note: The state shown above is only the initial state. State updates happen via events, as discussed in the State section.)

Managing Sessions with a `SessionService`

As seen above, you don't typically create or manage `Session` objects directly. Instead, you use a **`SessionService`**. This service acts as the central manager responsible for the entire lifecycle of your conversation sessions.

Its core responsibilities include:

- **Starting New Conversations:** Creating fresh `Session` objects when a user begins an interaction.
- **Resuming Existing Conversations:** Retrieving a specific `Session` (using its ID) so the agent can continue where it left off.
- **Saving Progress:** Appending new interactions (`Event` objects) to a session's history. This is also the mechanism through which session `state` gets updated (more in the `State` section).

- **Listing Conversations:** Finding the active session threads for a particular user and application.
- **Cleaning Up:** Deleting `Session` objects and their associated data when conversations are finished or no longer needed.

`SessionService` Implementations

ADK provides different `SessionService` implementations, allowing you to choose the storage backend that best suits your needs:

1. **`InMemorySessionService`**
    - **How it works:** Stores all session data directly in the application's memory.
    - **Persistence:** None. **All conversation data is lost if the application restarts.**
    - **Requires:** Nothing extra.
    - **Best for:** Quick development, local testing, examples, and scenarios where long-term persistence isn't required.

    Python

    ```
    from google.adk.sessions import InMemorySessionService
    session_service = InMemorySessionService()
    ```

    Java

    ```
     import com.google.adk.sessions.InMemorySessionService;
     InMemorySessionService exampleSessionService = new
    InMemorySessionService();
    ```

2. **`VertexAiSessionService`**
    - **How it works:** Uses Google Cloud's Vertex AI infrastructure via API calls for session management.
    - **Persistence:** Yes. Data is managed reliably and scalably via Vertex AI Agent Engine.
    - **Requires:**
        - A Google Cloud project (`pip install vertexai`)

    - A Google Cloud storage bucket that can be configured by this step.
    - A Reasoning Engine resource name/ID that can setup following this tutorial.
- **Best for:** Scalable production applications deployed on Google Cloud, especially when integrating with other Vertex AI features.

Python

```
# Requires: pip install google-adk[vertexai]
# Plus GCP setup and authentication
from google.adk.sessions import VertexAiSessionService

PROJECT_ID = "your-gcp-project-id"
LOCATION = "us-central1"
# The app_name used with this service should be the Reasoning
Engine ID or name
REASONING_ENGINE_APP_NAME = "projects/your-gcp-project-
id/locations/us-central1/reasoningEngines/your-engine-id"

session_service = VertexAiSessionService(project=PROJECT_ID,
location=LOCATION)
# Use REASONING_ENGINE_APP_NAME when calling service methods,
e.g.:
# session_service = await
session_service.create_session(app_name=REASONING_ENGINE_APP_NAME,
...)
```

Java

```
// Please look at the set of requirements above, consequently
export the following in your bashrc file:
// export GOOGLE_CLOUD_PROJECT=my_gcp_project
// export GOOGLE_CLOUD_LOCATION=us-central1
// export GOOGLE_API_KEY=my_api_key

import com.google.adk.sessions.VertexAiSessionService;
import java.util.UUID;

String sessionId = UUID.randomUUID().toString();
String reasoningEngineAppName = "123456789";
String userId = "u_123"; // Example user id
ConcurrentMap<String, Object> initialState = new
    ConcurrentHashMap<>(); // No initial state needed for this
example

VertexAiSessionService sessionService = new
VertexAiSessionService();
Session mySession =
    sessionService
        .createSession(reasoningEngineAppName, userId,
initialState, Optional.of(sessionId))
        .blockingGet();
```

3. `DatabaseSessionService`

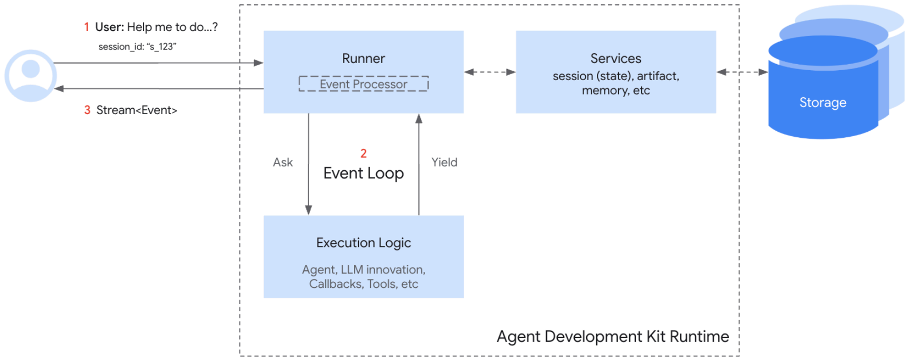
Currently supported in Python

- **How it works:** Connects to a relational database (e.g., PostgreSQL, MySQL, SQLite) to store session data persistently in tables.
- **Persistence:** Yes. Data survives application restarts.
- **Requires:** A configured database.
- **Best for:** Applications needing reliable, persistent storage that you manage yourself.

```
from google.adk.sessions import DatabaseSessionService
# Example using a local SQLite file:
db_url = "sqlite:///./my_agent_data.db"
session_service = DatabaseSessionService(db_url=db_url)
```

Choosing the right `SessionService` is key to defining how your agent's conversation history and temporary data are stored and persist.

The Session Lifecycle


1 User: Help me to do...?
session_id: "s_123"
3 Stream<Event>
Runner
Event Processor
Services
session (state), artifact, memory, etc
Storage
Ask
2 Event Loop
Yield
Execution Logic
Agent, LLM innovation, Callbacks, Tools, etc
Agent Development Kit Runtime

Here's a simplified flow of how `Session` and `SessionService` work together during a conversation turn:

1. **Start or Resume:** Your application's `Runner` uses the `SessionService` to either `create_session` (for a new chat) or `get_session` (to retrieve an existing one).
2. **Context Provided:** The `Runner` gets the appropriate `Session` object from the appropriate service method, providing the agent with access to the corresponding Session's `state` and `events`.
3. **Agent Processing:** The user prompts the agent with a query. The agent analyzes the query and potentially the session `state` and `events` history to determine the response.
4. **Response & State Update:** The agent generates a response (and potentially flags data to be updated in the `state`). The `Runner` packages this as an `Event`.
5. **Save Interaction:** The `Runner` calls `sessionService.append_event(session, event)` with the `session` and the new `event` as the arguments. The service adds the `Event` to the history and updates the session's `state` in storage based on information within the event. The session's `last_update_time` also get updated.
6. **Ready for Next:** The agent's response goes to the user. The updated `Session` is now stored by the `SessionService`, ready for the next turn (which restarts the cycle at step 1, usually with the continuation of the conversation in the current session).
7. **End Conversation:** When the conversation is over, your application calls `sessionService.delete_session(...)` to clean up the stored session data if it is no longer required.

This cycle highlights how the `SessionService` ensures conversational continuity by managing the history and state associated with each `Session` object.