Google Agent Development Kit (ADK): Sessions, Memory, and Runtime

Introduction to Conversational Context

Meaningful, multi-turn conversations require agents to understand context. Just like humans, they need to recall what's been said and done to maintain continuity and avoid repetition. The Agent Development Kit (ADK) provides structured ways to manage this context through `Session`, `State`, and `Memory`.

Think of interacting with your agent as having distinct **conversation threads**, potentially drawing upon **long-term knowledge**. ADK's context management system is designed to handle both immediate conversation context and longer-term knowledge retention.

Sessions: Tracking Individual Conversations

A `Session` is the ADK object designed specifically to track and manage individual conversation threads. Following the idea of a "conversation thread," just like you wouldn't start every text message from scratch, agents need context from the ongoing interaction.

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:

1. **Identification (`id`, `app_name`, `user_id`)**: Unique labels for the conversation.
2. `id`: A unique identifier for this specific conversation thread, essential for retrieving it later.
3. `app_name`: Identifies which agent application this conversation belongs to.

4. `user_id`: Links the conversation to a particular user.

5. **History (`events`)**: A chronological sequence of all interactions (`Event` objects – user messages, agent responses, tool actions) that have occurred within this specific thread.

6. **Session Data (`state`)**: A place to store temporary data relevant only to this specific, ongoing conversation. This acts as the agent's "scratchpad" for the current interaction.

Managing Sessions with a SessionService

The `SessionService` handles the lifecycle of `Session` objects:

1. **Creating Sessions**: Initializes a new conversation thread with a unique ID.
2. **Retrieving Sessions**: Loads an existing conversation by its ID.
3. **Updating Sessions**: Adds new events or modifies state.
4. **Deleting Sessions**: Removes conversations that are no longer needed.

SessionService Implementations

ADK offers different implementations for the `SessionService`:

1. **InMemorySessionService**: Stores sessions in memory.
2. **Pros**: Simple, fast, no setup required.
3. **Cons**: All data is lost when the application restarts.

4. **Best for**: Local development and testing.

5. **Database-backed implementations**: Store sessions in persistent storage.

6. **Pros**: Data persists across application restarts.
7. **Cons**: Requires database setup and configuration.

8. **Best for**: Production environments.

9. **Cloud-based implementations**: Store sessions in cloud services.

10. **Pros**: Scalable, managed infrastructure.
11. **Cons**: Requires cloud service setup and configuration.
12. **Best for**: Production environments with high scalability needs.

State: The Session's Scratchpad

Within each `Session` (our conversation thread), the `state` attribute acts like the agent's dedicated scratchpad for that specific interaction. While `session.events` holds the full history, `session.state` is where the agent stores and updates dynamic details needed during the conversation.

What is session.state?

Conceptually, `session.state` is a dictionary holding key-value pairs. It's designed for information the agent needs to recall or track to make the current conversation effective:

- **Personalize Interaction**: Remember user preferences mentioned earlier (e.g., `"user_preference_theme": "dark"`).
- **Track Task Progress**: Keep tabs on steps in a multi-turn process (e.g., `"booking_step": "confirm_payment"`).
- **Accumulate Information**: Build lists or summaries (e.g., `"shopping_cart_items": ["book", "pen"]`).
- **Make Informed Decisions**: Store flags or values influencing the next response (e.g., `"user_is_authenticated": True`).

Key Characteristics of State

1. **Structure: Serializable Key-Value Pairs**
2. Data is stored as `key: value`.
3. Keys: Always strings (`str`). Use clear names (e.g., `"departure_city"`, `"user:language_preference"`).

4. Values: Must be serializable. This means primitive types (strings, numbers, booleans) or collections of these (lists, dictionaries).

5. **Scope: Session-Specific**

6. Data in `state` is only relevant to the current session.

7. When the session ends or is deleted, this data is typically lost (unless explicitly saved elsewhere).

8. **Persistence: Depends on SessionService**

9. With `InMemorySessionService`, state is lost when the application restarts.
10. With database or cloud implementations, state persists as long as the session exists.

Organizing State with Prefixes: Scope Matters

It's recommended to use prefixes in state keys to organize data by scope or ownership:

- **Tool-specific data**: `"tool_name.key"` (e.g., `"calendar.last_viewed_date"`)
- **Agent-specific data**: `"agent.key"` (e.g., `"agent.current_task"`)

- **User-specific data**: `"user.key"` (e.g., `"user.preferred_language"`)

How State is Updated: Recommended Methods

ADK provides several methods to update state safely:

1. **Using SessionService.update_session_state()**: `python session_service.update_session_state( session_id=session.id, updates={"key1": "new_value1", "key2": "new_value2"} )`

2. **Using Runner.update_session_state()**: `python runner.update_session_state( session_id=session.id, updates={"key1": "new_value1", "key2": "new_value2"} )`

3. **From within a Tool function**: ```python def my_tool_function(session): # Read from state current_value = session.state.get("my_key", "default")

   # Update state (will be committed after tool execution) session.state["my_key"] = "new_value" return "Tool result" ```

Warning About Direct State Modification

While direct modification of `session.state` is possible, it's important to understand when changes are committed:

- Changes made directly to `session.state` during tool execution are automatically committed when the tool completes.
- Changes made outside the normal execution flow (e.g., in custom code) need to be explicitly committed using one of the update methods.

Memory: Long-Term Knowledge with MemoryService

We've seen how `Session` tracks the history (`events`) and temporary data (`state`) for a single, ongoing conversation. But what if an agent needs to recall information from past conversations or access external knowledge bases? This is where the concept of Long-Term Knowledge and the `MemoryService` come into play.

Think of it this way: - `Session` / `State`: Like your short-term memory during one specific chat. - Long-Term Knowledge (`MemoryService`): Like a searchable archive or knowledge library the agent can consult, potentially containing information from many past chats or other sources.

The MemoryService Role

The `BaseMemoryService` defines the interface for managing this searchable, long-term knowledge store. Its primary responsibilities are:

1. **Ingesting Information (`add_session_to_memory`)**: Taking the contents of a (usually completed) `Session` and adding relevant information to the long-term knowledge store.

2. **Searching Information (`search_memory`)**: Allowing an agent (typically via a `Tool`) to query the knowledge store and retrieve relevant snippets or context based on a search query.

MemoryService Implementations

ADK offers different implementations for the `MemoryService`:

1. **InMemoryMemoryService**: Stores memory in RAM.
2. **Pros**: Simple, fast, no setup required.
3. **Cons**: All data is lost when the application restarts.

4. **Best for**: Local development and testing.

5. **Vector database implementations**: Store embeddings for semantic search.

6. **Pros**: Powerful semantic search capabilities.
7. **Cons**: Requires database setup and configuration.

8. **Best for**: Production environments requiring sophisticated memory retrieval.

9. **Cloud-based implementations**: Leverage cloud services for memory storage.

10. **Pros**: Scalable, managed infrastructure.
11. **Cons**: Requires cloud service setup and configuration.
12. **Best for**: Production environments with high scalability needs.

How Memory Works in Practice

The typical workflow for using memory in an agent application involves:

1. **Adding to Memory**: After a conversation completes, relevant information is extracted and added to the memory store. `python memory_service.add_session_to_memory(session)`

2. **Searching Memory**: During a conversation, the agent can search the memory to recall relevant information. `python results = memory_service.search_memory( query="What was the user's preferred shipping method?", limit=5 )`

3. **Using Memory Results**: The agent incorporates the retrieved information into its responses or decision-making process.

Runtime: The Engine Powering Agent Interactions

The ADK Runtime is the underlying engine that powers your agent application during user interactions. It's the system that takes your defined agents, tools, and callbacks and orchestrates their execution in response to user input, managing the flow of information, state changes, and interactions with external services like LLMs or storage.

Think of the Runtime as the "engine" of your agentic application. You define the parts (agents, tools), and the Runtime handles how they connect and run together to fulfill a user's request.

Core Idea: The Event Loop

At its heart, the ADK Runtime operates on an Event Loop. This loop facilitates a back-and-forth communication between the `Runner` component and your defined "Execution Logic" (which includes your Agents, the LLM calls they make, Callbacks, and Tools).

In simple terms:

1. The `Runner` receives a user query and asks the main `Agent` to start processing.
2. The `Agent` (and its associated logic) runs until it has something to report (like a response, a request to use a tool, or a state change) – it then `yields` an `Event`.
3. The `Runner` receives this `Event`, processes any associated actions (like saving state changes via `Services`), and forwards the event onwards (e.g., to the user interface).
4. Only `after` the `Runner` has processed the event does the `Agent`'s logic `resume` from where it paused, now potentially seeing the effects of the changes committed by the Runner.
5. This cycle repeats until the agent has no more events to yield for the current user query.

This event-driven loop is the fundamental pattern governing how ADK executes your agent code.

The Heartbeat: The Event Loop - Inner workings

The Event Loop is the core operational pattern defining the interaction between the `Runner` and your custom code (Agents, Tools, Callbacks, collectively referred to as "Execution Logic" or "Logic Components" in the design document). It establishes a clear division of responsibilities:

Runner's Role (Orchestrator)

The `Runner` is responsible for:

1. **Initializing the execution environment**: Setting up the session, state, and other resources.
2. **Processing events yielded by execution logic**: Taking appropriate actions based on event type.
3. **Committing state changes**: Ensuring state updates are properly saved.
4. **Managing the execution flow**: Deciding when to resume execution logic.
5. **Handling errors and exceptions**: Providing graceful error handling.

Execution Logic's Role (Agent, Tool, Callback)

Your execution logic (Agents, Tools, Callbacks) is responsible for:

1. **Processing user input**: Understanding and responding to user queries.
2. **Making decisions**: Determining what actions to take.
3. **Yielding events**: Communicating results, requests, and state changes to the Runner.
4. **Resuming execution**: Continuing from where it left off after the Runner processes events.

Key components of the Runtime

1. **Runner**: The orchestrator that manages the execution flow.
2. **Execution Logic Components**:
3. **Agent**: The main component that processes user input and generates responses.
4. **Tool**: Components that perform specific tasks (e.g., searching, calculating).
5. **Callback**: Components that are notified of specific events.
6. **Event**: The communication mechanism between the Runner and Execution Logic.
7. **Services**: Components that provide access to external resources (e.g., SessionService, MemoryService).
8. **Session**: The container for conversation state and history.
9. **Invocation**: A single execution of the Runtime in response to a user query.

How It Works: A Simplified Invocation

When a user sends a query to your agent application, the following steps occur:

1. The application calls `runner.run_async(session_id, user_id, new_message)`.
2. The Runner retrieves the session and prepares the execution environment.
3. The Runner adds the user's message as an event to the session.
4. The Runner invokes the main Agent with the updated session.
5. The Agent processes the input, potentially using LLMs and Tools.
6. When the Agent has a response or needs to use a Tool, it yields an Event.
7. The Runner processes the Event, updates the session, and resumes the Agent.
8. This cycle continues until the Agent has no more events to yield.
9. The Runner returns the final result to the application.

Important Runtime Behaviors

1. **State Updates & Commitment Timing**: State changes are only committed when the Runner processes events, not when they're made by execution logic.
2. **"Dirty Reads" of Session State**: Execution logic may see uncommitted state changes, which can lead to inconsistencies if not handled properly.
3. **Streaming vs. Non-Streaming Output**: The Runtime supports both streaming (partial=True) and non-streaming output modes.
4. **Async is Primary**: The primary API for running agents is asynchronous (`run_async`), which allows for better performance and responsiveness.

Artifacts

Artifacts in ADK are a way to store and manage binary data or large text content that doesn't fit well in the regular session state. They provide a mechanism for handling files, images, audio, or other binary data within the agent system.

Key characteristics of Artifacts:

1. **Binary Data Storage**: Designed for storing binary data or large text content.
2. **Separate from State**: Stored separately from session state to avoid performance issues.
3. **Reference-Based**: Referenced in session state via artifact IDs.
4. **Lifecycle Management**: Can be created, retrieved, and deleted.

Callbacks

Callbacks in ADK provide a way to hook into the agent execution lifecycle, allowing you to execute custom code at specific points during the agent's operation. They're useful for logging, monitoring, debugging, and extending the agent's functionality.

Key characteristics of Callbacks:

1. **Lifecycle Hooks**: Execute at specific points in the agent execution lifecycle.
2. **Custom Logic**: Allow you to add custom behavior without modifying the core agent code.
3. **Multiple Callbacks**: Multiple callbacks can be registered and will execute in registration order.
4. **Synchronous and Asynchronous**: Support both sync and async execution models.

Events

Events in ADK are the primary communication mechanism between the Runner and Execution Logic components. They represent discrete occurrences during agent execution, such as user messages, agent responses, tool calls, and state changes.

Key characteristics of Events:

1. **Communication Mechanism**: Used to communicate between Runne (Content truncated due to size limit. Use line ranges to read in chunks)