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 collection (dictionary or Map) 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**
    - Data is stored as `key: value`.
    - **Keys:** Always strings (`str`). Use clear names (e.g., `'departure_city'`, `'user:language_preference'`).
    - **Values:** Must be **serializable**. This means they can be easily saved and loaded by the `SessionService`. Stick to basic types in the specific languages (Python/ Java) like strings, numbers, booleans, and simple lists or dictionaries containing *only* these basic types. (See API documentation for precise details).

- ⚠️ **Avoid Complex Objects: Do not store non-serializable objects** (custom class instances, functions, connections, etc.) directly in the state. Store simple identifiers if needed, and retrieve the complex object elsewhere.

2. **Mutability: It Changes**
   - The contents of the `state` are expected to change as the conversation evolves.
3. **Persistence: Depends on `SessionService`**
   - Whether state survives application restarts depends on your chosen service:
   - `InMemorySessionService`: **Not Persistent.** State is lost on restart.
   - `DatabaseSessionService` / `VertexAiSessionService`: **Persistent.** State is saved reliably.

> **Note**
>
> The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., `session.state['current_intent'] = 'book_flight'` in Python, `session.state().put("current_intent", "book_flight)` in Java). Refer to the language-specific API documentation for details.

Organizing State with Prefixes: Scope Matters

Prefixes on state keys define their scope and persistence behavior, especially with persistent services:

- **No Prefix (Session State):**
  - **Scope:** Specific to the *current* session (`id`).
  - **Persistence:** Only persists if the `SessionService` is persistent (`Database`, `VertexAI`).
  - **Use Cases:** Tracking progress within the current task (e.g., `'current_booking_step'`), temporary flags for this interaction (e.g., `'needs_clarification'`).
  - **Example:** `session.state['current_intent'] = 'book_flight'`
- `user:` **Prefix (User State):**

- **Scope:** Tied to the `user_id`, shared across *all* sessions for that user (within the same `app_name`).
- **Persistence:** Persistent with `Database` or `VertexAI`. (Stored by `InMemory` but lost on restart).
- **Use Cases:** User preferences (e.g., `'user:theme'`), profile details (e.g., `'user:name'`).
- **Example:** `session.state['user:preferred_language'] = 'fr'`

- `app:` **Prefix (App State):**
  - **Scope:** Tied to the `app_name`, shared across *all* users and sessions for that application.
  - **Persistence:** Persistent with `Database` or `VertexAI`. (Stored by `InMemory` but lost on restart).
  - **Use Cases:** Global settings (e.g., `'app:api_endpoint'`), shared templates.
  - **Example:** `session.state['app:global_discount_code'] = 'SAVE10'`
- `temp:` **Prefix (Temporary Session State):**
  - **Scope:** Specific to the *current* session processing turn.
  - **Persistence: Never Persistent.** Guaranteed to be discarded, even with persistent services.
  - **Use Cases:** Intermediate results needed only immediately, data you explicitly don't want stored.
  - **Example:** `session.state['temp:raw_api_response'] = {...}`

How the Agent Sees It: Your agent code interacts with the *combined* state through the single `session.state` collection (dict/ Map). The `SessionService` handles fetching/merging state from the correct underlying storage based on prefixes.

How State is Updated: Recommended Methods

State should **always** be updated as part of adding an `Event` to the session history using `session_service.append_event()`. This ensures changes are tracked, persistence works correctly, and updates are thread-safe.

1. The Easy Way: `output_key` **(for Agent Text Responses)**

This is the simplest method for saving an agent's final text response directly into the state. When defining your `LlmAgent`, specify the `output_key`:

Python

```
from google.adk.agents import LlmAgent
from google.adk.sessions import InMemorySessionService, Session
from google.adk.runners import Runner
from google.genai.types import Content, Part

# Define agent with output_key
greeting_agent = LlmAgent(
    name="Greeter",
    model="gemini-2.0-flash", # Use a valid model
    instruction="Generate a short, friendly greeting.",
    output_key="last_greeting" # Save response to
state['last_greeting']
)

# --- Setup Runner and Session ---
app_name, user_id, session_id = "state_app", "user1", "session1"
session_service = InMemorySessionService()
runner = Runner(
    agent=greeting_agent,
    app_name=app_name,
    session_service=session_service
)
session = await session_service.create_session(app_name=app_name,
                                    user_id=user_id,
                                    session_id=session_id)
print(f"Initial state: {session.state}")

# --- Run the Agent ---
# Runner handles calling append_event, which uses the output_key
# to automatically create the state_delta.
user_message = Content(parts=[Part(text="Hello")])
for event in runner.run(user_id=user_id,
                        session_id=session_id,
                        new_message=user_message):
    if event.is_final_response():
      print(f"Agent responded.") # Response text is also in
event.content

# --- Check Updated State ---
updated_session = await
session_service.get_session(app_name=APP_NAME, user_id=USER_ID,
session_id=session_id)
print(f"State after agent run: {updated_session.state}")
# Expected output might include: {'last_greeting': 'Hello there!
How can I help you today?'}
```

Java

```
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.RunConfig;
import com.google.adk.events.Event;
import com.google.adk.runner.Runner;
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import java.util.List;
import java.util.Optional;

public class GreetingAgentExample {

  public static void main(String[] args) {
    // Define agent with output_key
    LlmAgent greetingAgent =
        LlmAgent.builder()
            .name("Greeter")
            .model("gemini-2.0-flash")
            .instruction("Generate a short, friendly greeting.")
            .description("Greeting agent")
            .outputKey("last_greeting") // Save response to
state['last_greeting']
            .build();

    // --- Setup Runner and Session ---
    String appName = "state_app";
    String userId = "user1";
    String sessionId = "session1";

    InMemorySessionService sessionService = new
InMemorySessionService();
    Runner runner = new Runner(greetingAgent, appName, null,
sessionService); // artifactService can be null if not used

    Session session =
        sessionService.createSession(appName, userId, null,
sessionId).blockingGet();
    System.out.println("Initial state: " +
session.state().entrySet());

    // --- Run the Agent ---
    // Runner handles calling appendEvent, which uses the
output_key
    // to automatically create the stateDelta.
    Content userMessage =
Content.builder().parts(List.of(Part.fromText("Hello"))).build();

    // RunConfig is needed for runner.runAsync in Java
```

```
    RunConfig runConfig = RunConfig.builder().build();

    for (Event event : runner.runAsync(userId, sessionId,
userMessage, runConfig).blockingIterable()) {
      if (event.finalResponse()) {
        System.out.println("Agent responded."); // Response text
is also in event.content
      }
    }

    // --- Check Updated State ---
    Session updatedSession =
        sessionService.getSession(appName, userId, sessionId,
Optional.empty()).blockingGet();
    assert updatedSession != null;
    System.out.println("State after agent run: " +
updatedSession.state().entrySet());
    // Expected output might include: {'last_greeting': 'Hello
there! How can I help you today?'}
  }
}
```

Behind the scenes, the `Runner` uses the `output_key` to create the necessary `EventActions` with a `state_delta` and calls `append_event`.

2. The Standard Way: `EventActions.state_delta` (for Complex Updates)

For more complex scenarios (updating multiple keys, non-string values, specific scopes like `user:` or `app:`, or updates not tied directly to the agent's final text), you manually construct the `state_delta` within `EventActions`.

Python

```
from google.adk.sessions import InMemorySessionService, Session
from google.adk.events import Event, EventActions
from google.genai.types import Part, Content
import time

# --- Setup ---
session_service = InMemorySessionService()
app_name, user_id, session_id = "state_app_manual", "user2",
"session2"
session = await session_service.create_session(
    app_name=app_name,
    user_id=user_id,
    session_id=session_id,
    state={"user:login_count": 0, "task_status": "idle"}
)
print(f"Initial state: {session.state}")
```

```
# --- Define State Changes ---
current_time = time.time()
state_changes = {
    "task_status": "active",              # Update session
state
    "user:login_count": session.state.get("user:login_count",
0) + 1, # Update user state
    "user:last_login_ts": current_time,   # Add user state
    "temp:validation_needed": True        # Add temporary state
(will be discarded)
}

# --- Create Event with Actions ---
actions_with_update = EventActions(state_delta=state_changes)
# This event might represent an internal system action, not
just an agent response
system_event = Event(
    invocation_id="inv_login_update",
    author="system", # Or 'agent', 'tool' etc.
    actions=actions_with_update,
    timestamp=current_time
    # content might be None or represent the action taken
)

# --- Append the Event (This updates the state) ---
await session_service.append_event(session, system_event)
print("`append_event` called with explicit state delta.")

# --- Check Updated State ---
updated_session = await
session_service.get_session(app_name=app_name,
                                            user_id=user_id,

session_id=session_id)
print(f"State after event: {updated_session.state}")
# Expected: {'user:login_count': 1, 'task_status': 'active',
'user:last_login_ts': <timestamp>}
# Note: 'temp:validation_needed' is NOT present.
```

Java

```
import com.google.adk.events.Event;
import com.google.adk.events.EventActions;
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import java.time.Instant;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class ManualStateUpdateExample {
```

```
  public static void main(String[] args) {
    // --- Setup ---
    InMemorySessionService sessionService = new
InMemorySessionService();
    String appName = "state_app_manual";
    String userId = "user2";
    String sessionId = "session2";

    ConcurrentMap<String, Object> initialState = new
ConcurrentHashMap<>();
    initialState.put("user:login_count", 0);
    initialState.put("task_status", "idle");

    Session session =
        sessionService.createSession(appName, userId,
initialState, sessionId).blockingGet();
    System.out.println("Initial state: " +
session.state().entrySet());

    // --- Define State Changes ---
    long currentTimeMillis = Instant.now().toEpochMilli(); //
Use milliseconds for Java Event

    ConcurrentMap<String, Object> stateChanges = new
ConcurrentHashMap<>();
    stateChanges.put("task_status", "active"); // Update
session state

    // Retrieve and increment login_count
    Object loginCountObj =
session.state().get("user:login_count");
    int currentLoginCount = 0;
    if (loginCountObj instanceof Number) {
      currentLoginCount = ((Number) loginCountObj).intValue();
    }
    stateChanges.put("user:login_count", currentLoginCount +
1); // Update user state

    stateChanges.put("user:last_login_ts", currentTimeMillis);
// Add user state (as long milliseconds)
    stateChanges.put("temp:validation_needed", true); // Add
temporary state

    // --- Create Event with Actions ---
    EventActions actionsWithUpdate =
EventActions.builder().stateDelta(stateChanges).build();

    // This event might represent an internal system action,
not just an agent response
    Event systemEvent =
        Event.builder()
            .invocationId("inv_login_update")
            .author("system") // Or 'agent', 'tool' etc.
```

```
            .actions(actionsWithUpdate)
            .timestamp(currentTimeMillis)
            // content might be None or represent the action
taken
            .build();

    // --- Append the Event (This updates the state) ---
    sessionService.appendEvent(session,
systemEvent).blockingGet();
    System.out.println("`appendEvent` called with explicit
state delta.");

    // --- Check Updated State ---
    Session updatedSession =
        sessionService.getSession(appName, userId, sessionId,
Optional.empty()).blockingGet();
    assert updatedSession != null;
    System.out.println("State after event: " +
updatedSession.state().entrySet());
    // Expected: {'user:login_count': 1, 'task_status':
'active', 'user:last_login_ts': <timestamp_millis>}
    // Note: 'temp:validation_needed' is NOT present because
InMemorySessionService's appendEvent
    // applies delta to its internal user/app state maps IF
keys have prefixes,
    // and to the session's own state map (which is then merged
on getSession).
  }
}
```

3. Via `CallbackContext` or `ToolContext` (Recommended for Callbacks and Tools)

Modifying state within agent callbacks (e.g., `on_before_agent_call`, `on_after_agent_call`) or tool functions is best done using the `state` attribute of the `CallbackContext` or `ToolContext` provided to your function.

- `callback_context.state['my_key'] = my_value`
- `tool_context.state['my_key'] = my_value`

These context objects are specifically designed to manage state changes within their respective execution scopes. When you modify `context.state`, the ADK framework ensures that these changes are automatically captured and correctly routed into the `EventActions.state_delta` for the event being generated by the callback or tool. This delta is then processed by the `SessionService` when the event is appended, ensuring proper persistence and tracking.

This method abstracts away the manual creation of `EventActions` and `state_delta` for most common state update scenarios within callbacks and tools, making your code cleaner and less error-prone.

For more comprehensive details on context objects, refer to the Context documentation.

Python

```
# In an agent callback or tool function
from google.adk.agents import CallbackContext # or ToolContext

def my_callback_or_tool_function(context: CallbackContext, # Or
ToolContext
                                 # ... other parameters ...
                                ):
    # Update existing state
    count = context.state.get("user_action_count", 0)
    context.state["user_action_count"] = count + 1

    # Add new state
    context.state["temp:last_operation_status"] = "success"

    # State changes are automatically part of the event's
state_delta
    # ... rest of callback/tool logic ...
```

Java

```
// In an agent callback or tool method
import com.google.adk.agents.CallbackContext; // or ToolContext
// ... other imports ...

public class MyAgentCallbacks {
    public void onAfterAgent(CallbackContext callbackContext) {
        // Update existing state
        Integer count = (Integer)
callbackContext.state().getOrDefault("user_action_count", 0);
        callbackContext.state().put("user_action_count", count
+ 1);

        // Add new state

callbackContext.state().put("temp:last_operation_status",
"success");

        // State changes are automatically part of the event's
state_delta
        // ... rest of callback logic ...
    }
```

```
}
```

What `append_event` Does:

- Adds the `Event` to `session.events`.
- Reads the `state_delta` from the event's `actions`.
- Applies these changes to the state managed by the `SessionService`, correctly handling prefixes and persistence based on the service type.
- Updates the session's `last_update_time`.
- Ensures thread-safety for concurrent updates.

⚠️ A Warning About Direct State Modification

Avoid directly modifying the `session.state` collection (dictionary/Map) on a `Session` object that was obtained directly from the `SessionService` (e.g., via `session_service.get_session()` or `session_service.create_session()`) *outside* of the managed lifecycle of an agent invocation (i.e., not through a `CallbackContext` or `ToolContext`). For example, code like `retrieved_session = await session_service.get_session(...); retrieved_session.state['key'] = value` is problematic.

State modifications *within* callbacks or tools using `CallbackContext.state` or `ToolContext.state` are the correct way to ensure changes are tracked, as these context objects handle the necessary integration with the event system.

Why direct modification (outside of contexts) is strongly discouraged:

1. **Bypasses Event History:** The change isn't recorded as an `Event`, losing auditability.
2. **Breaks Persistence:** Changes made this way **will likely NOT be saved** by `DatabaseSessionService` or `VertexAiSessionService`. They rely on `append_event` to trigger saving.
3. **Not Thread-Safe:** Can lead to race conditions and lost updates.
4. **Ignores Timestamps/Logic:** Doesn't update `last_update_time` or trigger related event logic.

Recommendation: Stick to updating state via `output_key`, `EventActions.state_delta` (when manually creating events), or by modifying the `state` property of `CallbackContext` or `ToolContext` objects when within their respective scopes. These methods ensure reliable, trackable, and persistent state management. Use direct access to `session.state` (from a `SessionService`-retrieved session) only for *reading* state.

Best Practices for State Design Recap

- **Minimalism:** Store only essential, dynamic data.
- **Serialization:** Use basic, serializable types.
- **Descriptive Keys & Prefixes:** Use clear names and appropriate prefixes (`user:`, `app:`, `temp:`, or none).
- **Shallow Structures:** Avoid deep nesting where possible.
- **Standard Update Flow:** Rely on `append_event`.