Google ADK library and BaseSessionService

Exported on 2/7/2026 at 22:08:35 GMT+2 from Cursor (2.4.27)

User

Briefly explain the purpose and capabilities of the @google-adk library, and explain the role of classes derived from BaseSessionService.

Cursor

Searching the ADK docs and your project for BaseSessionService and session handling.

Checking the ADK overview and Python SessionService/BaseSessionService API:

Checking the ADK Python API for the session service base class:

Here's a concise summary:

Google ADK – purpose and capabilities

Agent Development Kit (ADK) is an open-source, code-first toolkit for building, evaluating, and deploying AI agents. It targets both conversational and non-conversational agents and multi-agent systems.

Purpose: Give developers a structured way to build agents with clear primitives (agents, tools, sessions, state, memory) and to run them locally or in production (e.g. Vertex AI, Cloud Run).

Main capabilities:

- **Agents:** LLM-based agents and workflow-style agents (e.g. `SequentialAgent`, `ParallelAgent`, `LoopAgent`).
- **Tools:** Custom functions, agents-as-tools, built-ins (e.g. code execution, search), MCP, OpenAPI.
- **Session management:** One **Session** per conversation thread: history (events), per-conversation **State**, and optional long-term **Memory** across sessions.
- **Runner:** Runs the agent loop, uses session/state/events, and talks to backend services (including the session store).
- **Streaming:** Bidirectional streaming (e.g. text/audio with Gemini).
- **Deployment & observability:** Vertex AI Agent Engine, Cloud Run, GKE; integration with tracing, logging, evaluation.

So ADK is the layer that defines how agents, tools, sessions, and storage plug together; the actual storage for sessions is pluggable via a **SessionService**.

Role of classes derived from `BaseSessionService` (SessionService implementations)

SessionService is the abstraction that owns the full lifecycle of **Session** objects (conversation threads). Classes that implement this abstraction (in Python these are typically subclasses of **BaseSessionService** or otherwise implement the same interface) are the only place that creates, reads, updates, and deletes sessions. The Runner and your app always go through a SessionService to get or change session data.

Concrete responsibilities:

1. **Create sessions** – Create a new `Session` (with `app_name`, `user_id`, optional initial `state`) and return it (e.g. when a user starts a new chat).
2. **Retrieve sessions** – Load an existing `Session` by id (and usually `app_name` / `user_id`) so the agent can resume a conversation.
3. **Append events and update state** – When the Runner finishes a turn, it calls the service to append the new `Event` to the session's history and to apply any state changes implied by that event; the service is responsible for persisting both.
4. **List sessions** – List sessions for a given app and/or user (e.g. "my conversations").
5. **Delete sessions** – Remove a session and its data when the conversation is over or for compliance.

Built-in implementations:

- **InMemorySessionService** – In-process only; data is lost on restart (dev/testing).
- **DatabaseSessionService** – Persistent storage in a relational DB (e.g. PostgreSQL, SQLite with an async driver).
- **VertexAiSessionService** – Managed persistence via Vertex AI / Agent Engine (production on GCP).

Custom implementations (e.g. your `mongo_session_service.py`):
By implementing a class derived from **BaseSessionService** (or the equivalent SessionService interface in your ADK language), you provide a **custom backend** for sessions (e.g. MongoDB). Your class must implement the same lifecycle operations (create, get, append_event, list, delete) so the Runner and the rest of ADK can manage conversation threads without caring whether storage is in memory, SQL, Vertex AI, or MongoDB. That's the role of SessionService and its derived classes: **they are the pluggable persistence layer for conversation threads**.

User

Give me a high-level overview of how DatabaseSessionService is implemented

---

Cursor

Checking the schema definitions to complete the picture:

Here's a concise high-level overview of how **DatabaseSessionService** is implemented in ADK Python.

High-level overview of DatabaseSessionService

DatabaseSessionService is a concrete **BaseSessionService** that persists sessions, events, and state in a relational database via **SQLAlchemy** (async engine and session factory). It uses separate storage tables and supports two schema versions (v0 and v1) with lazy table creation and optional schema migration.

---

1. Construction and engine

- **Input:** A database URL (e.g. `sqlite+aiosqlite:///file.db` or `postgresql+asyncpg://...`).
- **Engine:** Uses `create_async_engine()` so all DB access is async. For SQLite in-memory it uses `StaticPool` and enables foreign keys via a `connect` event; for other backends it can use `pool_pre_ping`.
- **Session factory:** An `async_sessionmaker` bound to that engine is used for all operations (no sync SQLAlchemy session APIs in this service).
- **Lazy setup:** Tables are not created in `__init__`. A `_prepare_tables()` step runs before the first operation and uses locks so schema detection and table creation are done once, safely.

---

2. Schema and storage model

- **Schema versions:** Two schema modules exist: **v0** (legacy, pickle-based) and **v1** (JSON-based). The service detects which version the DB is using (or assumes v0 if no metadata) and then uses the matching ORM classes (`_SchemaClasses`).
- **Storage tables (conceptually):**
  - **Sessions** – One row per conversation: `app_name`, `user_id`, session `id`, session-level **state** (dict), `create_time`, `update_time`.
  - **Events** – One row per event in a session: linked by `(app_name, user_id, session_id)`, with event payload and **timestamp** (used for ordering and optional filtering in `get_session`).
  - **App-level state** – One row per `app_name`; key-value state shared across sessions of that app.
  - **User-level state** – One row per `(app_name, user_id)`; key-value state for that user across their sessions.
- **State merging:** At read time, session state, user state, and app state are merged (with prefixes) into the single `Session.state` dict that the rest of ADK expects.

---

3. BaseSessionService API implementation

- **create_session:**
  Ensures tables exist, then in a transaction: loads or creates app and user state rows, applies initial state deltas (split into app/user/session via `_session_util.extract_state_delta`), inserts a new session row, commits, and returns a `Session` with merged state. Reuses existing `session_id` if provided; otherwise the storage layer generates one. Raises if a session with that id already exists.

- **get_session:**
  Loads the session row by `(app_name, user_id, session_id)`. If `GetSessionConfig` is provided, loads events with optional `after_timestamp` and/or `num_recent_events`, ordered by timestamp. Fetches app and user state, merges with session state, builds `Event` list from storage events, and returns a `Session` (or `None` if not found).

- **list_sessions:**
  Queries session rows by `app_name` and optional `user_id`, loads app and (per-user) user state, merges state for each session, and returns `ListSessionsResponse(sessions=...)`.

- **delete_session:**
  Deletes the session row for `(app_name, user_id, session_id)`. Event and state table design may allow cascades or leave orphan events depending on schema; the main contract is that the session row is removed.

- **append_event:**
  Ensures tables exist, then in a transaction: loads the session and app/user state rows. Optionally **reloads** the in-memory `Session` from storage if the stored `update_time` is newer than `session.last_update_time` (optimistic concurrency). Applies `event.actions.state_delta` to app, user, and session state in storage. Persists the new event via `StorageEvent.from_event(session, event)`, updates the session's `update_time`, commits, then updates `session.last_update_time` and calls `super().append_event()` so in-memory session and event list stay in sync.

---

4. Concurrency and transaction handling

- **Transactions:** Each operation runs inside an async context that provides a DB session and **rolls back on any exception** (`_rollback_on_exception_session`) to avoid leaving failed transactions in the pool.
- **Lazy table creation:** `_prepare_tables()` uses two locks: one to detect/set the DB schema version, and one to create tables once. All of this is async and avoids duplicate creation under concurrency.
- **Optimistic check in append_event:** Before applying an event, the code compares stored `update_time` with the in-memory session's `last_update_time` and, if the DB is newer, reloads session and events from the DB so concurrent updates are not lost.

---

5. Lifecycle and compatibility

- **close() / aexit:** Disposes the SQLAlchemy engine and closes connections when the service is used as an async context manager.
- **Schema migration:** From v1.22.0 onward, the recommended schema is v1 (JSON). The docs describe a separate migration path (e.g. `adk migrate session`) from v0 (pickle) to v1; the service itself just uses whichever schema version it detects in the DB.

In short: **DatabaseSessionService** implements the full BaseSessionService API by mapping sessions, events, and app/user/session state to SQLAlchemy-backed tables, with async-only access, lazy schema/table setup, two schema versions (v0/v1), and careful transaction and concurrency handling so it can be used as a drop-in persistent backend for ADK sessions.

User

List all the tables where DatabaseSessionService stores session data for schema v1. For each table, list the primary keys (PKs), then explain how the tables are connected to each other (FKs?).

Cursor

Here's a concise summary of the **v1 schema** used by `DatabaseSessionService`: tables, primary keys, and how they relate.

Tables in schema v1

1. `adk_internal_metadata` (StorageMetadata)

Column	Type	Role
key	String(128)	**PK**
value	String(256)	

- **PK:** `key`
- **FKs:** None. Used for internal metadata (e.g. schema version).

---

2. `sessions` (StorageSession)

Column	Type	Role
app_name	String(128)	**PK**
user_id	String(128)	**PK**
id	String(128)	**PK** (session id, default uuid4)
state	DynamicJSON	Session-level state
create_time	PreciseTimestamp	
update_time	PreciseTimestamp	

- **PK:** `(app_name, user_id, id)` (composite)
- **FKs:** None. Referenced by `events`.

---

3. `events` (StorageEvent)

Column	Type	Role
id	String(128)	**PK** (event id)
app_name	String(128)	**PK**
user_id	String(128)	**PK**

Column	Type	Role
session_id	String(128)	**PK**
invocation_id	String(256)	
timestamp	PreciseTimestamp	
event_data	DynamicJSON	Serialized Event (JSON)

- **PK:** `(id, app_name, user_id, session_id)` (composite)
- **FK:** `(app_name, user_id, session_id) → (sessions.app_name, sessions.user_id, sessions.id)` with **ON DELETE CASCADE** (deleting a session deletes its events).

---

4. `app_states` (StorageAppState)

Column	Type	Role
app_name	String(128)	**PK**
state	DynamicJSON	App-level state
update_time	PreciseTimestamp	

- **PK:** `app_name`
- **FKs:** None. Logically "one row per app".

---

5. `user_states` (StorageUserState)

Column	Type	Role
app_name	String(128)	**PK**
user_id	String(128)	**PK**
state	DynamicJSON	User-level state
update_time	PreciseTimestamp	

- **PK:** `(app_name, user_id)` (composite)
- **FKs:** None. Logically "one row per (app, user)".

How the tables are connected

- **Only declared foreign key:**
  `events` → `sessions`
  - Columns: `(app_name, user_id, session_id)` in `events`.
  - References: `(sessions.app_name, sessions.user_id, sessions.id)`.
  - `ON DELETE CASCADE`: deleting a session row removes all its event rows.
- **No FKs between:**
  `sessions`, **`app_states`**, and **`user_states`**.
  They are linked only by **application logic** using the same identifiers:
  - **`sessions`** and **`app_states`** share `app_name`.
  - **`sessions`** and **`user_states`** share `(app_name, user_id)`.
  - At runtime, `DatabaseSessionService` loads session + app_state + user_state and merges their `state` dicts (with prefixes) into a single `Session.state`.
- **`adk_internal_metadata`** is standalone (key-value store for schema version, etc.).

So in v1 there is **one explicit FK**: `events` → `sessions` with CASCADE. The other tables are related only by shared keys (`app_name`, `user_id`) in code, not by database FKs.

User

What concurrency considerations are taken into account in the @google-adk DatabaseSessionService implementation? Does it use any synchronization primitives when writing concurrently to the same tables from within the same process?

Cursor

Pulling the relevant concurrency-related parts from the DatabaseSessionService source.

Here's how concurrency is handled in **DatabaseSessionService** and what that means for concurrent writes in the same process.

---

1. Synchronization primitives (only for one-time setup)

The implementation uses **two `asyncio.Lock()` instances**, both only for lazy initialization, **not** for normal reads/writes:

- **`_db_schema_lock`**
  Used in `_prepare_tables()` so that schema version detection runs only once. Pattern: check `_db_schema_version is None` → acquire lock → double-check → detect version and set `_db_schema_version` → release. No lock is held during any data access.

- **`_table_creation_lock`**
  Used in `_prepare_tables()` so that table creation runs only once. Pattern: check `not _tables_created` → acquire lock → double-check → create tables and set `_tables_created = True` → release. Again, no lock around actual data operations.

So: **synchronization primitives are used only to make schema detection and table creation safe under concurrent first use.** They do **not** protect concurrent writes to the same tables or the same session.

---

2. No per-session or per-write locking

There are **no** application-level locks (no per-session, per-user, or per-table locks) around:

- `create_session`
- `get_session`
- `list_sessions`
- `delete_session`
- `append_event`

So **concurrent writes to the same tables from within the same process are not coordinated by any in-process synchronization primitive.** Multiple coroutines can call these methods at the same time; they do not wait on each other via locks.

---

3. How concurrent writes are handled instead

Separate transactions

Each logical operation uses its own SQLAlchemy async session from `_rollback_on_exception_session()` and commits (or rolls back on exception). So concurrent calls get different transactions. Concurrency is left to the **database** (isolation level, row locking, etc.), not to in-process locks.

Rollback on exception

The context manager always rolls back the transaction on any exception before re-raising. That avoids leaving failed transactions open and exhausting the connection pool; it doesn't serialize or coordinate concurrent writers.

Optimistic "reload if stale" in `append_event`

Before applying a new event and state deltas, the code compares the session's stored `update_time` with the in-memory `session.last_update_time`:

```
if (
    storage_session.get_update_timestamp(is_sqlite)
    > session.last_update_time
):
    # Reload the session from storage if it has been updated since it was
    # loaded.
    ...
    session.state = _merge_state(...)
    session.events = [e.to_event() for e in storage_events]
```

So if another writer has already committed an update to that session, the current writer **refreshes** the in-memory session from the DB and then applies its own event and state delta and commits. There is **no** retry loop, no row-level locking (e.g. `SELECT ... FOR UPDATE`), and no in-process lock. So:

- **Events:** Each append adds a new row (event id is unique); concurrent appends can both commit and both events are stored.
- **State:** Updates are read-modify-write in the transaction. Two concurrent appends can both read, merge their deltas, and commit; the last commit wins for the overlapping state. The “reload if stale” only makes the in-memory `Session` object reflect the latest DB state before this writer applies its delta; it does not prevent two writers from interleaving.

So: **concurrent writes to the same session from the same process are allowed; they rely on DB transactions and this optimistic refresh, with no in-process synchronization around the actual writes.**

---

4. Summary

Aspect	What the implementation does
Locks for setup	Uses `asyncio.Lock()` only for schema version detection and table creation (double-checked locking).
Locks for data writes	**None.** No per-session, per-table, or per-operation locks when writing to the same tables from the same process.
Concurrent writes	Handled by separate DB transactions plus the “reload if stale” logic in `append_event`; no in-process serialization of writes.

So: **yes**, it uses synchronization primitives, but **only** to make one-time setup safe. It **does not** use any synchronization primitives when writing concurrently to the same tables from within the same process; those writes are coordinated only by the database and the optimistic reload in `append_event`.