Run your first AIP action

In this tutorial, you will start one product-neutral AIP daemon on loopback, discover its manifest, invoke the built-in health capability, and inspect the durable action state. You will also view the same capability surface through the MCP

SECTION	Start Here
TYPE	Getting started
PROTOCOL	AIP 1.0
SOURCE	docs/getting-started/quickstart.md

This path is for developers learning the native request lifecycle. It uses a literal development bearer token and local file-backed state at source revision `d7cce13d1d555644d04a4d73c66c95b113737635`. Do not expose this configuration on a shared network or treat its success as connector-fleet or production qualification.

What you will observe

By the end of the tutorial, you will have:

- a ready local `getaip-server` process;
- a manifest containing `cap:aip:server:health`;
- an `aip.core.v1.action_result` with status `completed`;
- a durable lifecycle view for action `act_quickstart_health_001`;
- an AIP manifest projection produced through MCP.

No external product, connector host, PostgreSQL database, or NATS server is required for this path.

Prerequisites

You need:

- Rust `1.88` or newer;
- a clean checkout at the source revision above;
- `curl`;
- two terminal windows.

Run every command from the repository root. Complete Install AIP first if the pinned checkout or Rust toolchain is not ready.

1. Start the daemon

In the first terminal, start `getaip-server` on loopback with an explicit development identity and a dedicated state directory:

```
SH
cargo run --locked -p getaip-server -- \
  --bind 127.0.0.1:18080 \
  --service-id agent:getaip:server:quickstart \
  --native-bearer-token local-development-token \
```

```
  --native-principal service:quickstart:client \
  --storage-dir .getaip-server-quickstart
```

Leave this process running. The bearer token authenticates native HTTP calls as `service:quickstart:client`. The storage directory selects the durable local runtime and preserves action, event, receipt, and replay state across process restarts.

Expected result: the command remains running and reports no startup error.

2. Check daemon readiness

In the second terminal, request the unauthenticated readiness endpoint:

```
SH
curl --fail --silent --show-error http://127.0.0.1:18080/ready
```

Expected result: HTTP `200` with a JSON body whose `status` is `ready`. At this revision, top-level readiness means the gateway is ready, the runtime worker is running, every required module is ready, and the NATS listener is running if NATS was configured as required. It does not prove that a connector replica or external product is reachable.

3. Discover the manifest

Fetch the participant manifest with the same development credential:

```
SH
cargo run --locked -p getaip-cli -- \
  --native-bearer-token local-development-token \
  manifest fetch http://127.0.0.1:18080
```

Expected result: a JSON manifest for `agent:getaip:server:quickstart`. Its capabilities include `cap:aip:server:health`, and its profiles include the bindings enabled by this daemon. Discover this contract instead of hard-coding assumptions about an unfamiliar deployment.

4. Invoke the health capability

Submit a native action with a stable tutorial identifier:

```
SH
cargo run --locked -p getaip-cli -- \
  --native-bearer-token local-development-token \
  action call http://127.0.0.1:18080 \
  cap:aip:server:health \
  --action-id act_quickstart_health_001 \
  --input '{}'
```

Expected result: an AIP envelope with message type `aip.core.v1.action_result`. Its `body.action_result` contains `"action_id": "act_quickstart_health_001"`, `"status": "completed"`, and a health payload in `output`.

The explicit action ID makes the next query copyable and gives an interrupted client a stable logical action to resume. For a replay-sensitive mutation, the caller must also preserve its original idempotency and transaction identities; the health capability is a low-risk read and requires no approval.

5. Read the durable lifecycle view

Query the action independently of the original HTTP response:

```
SH
cargo run --locked -p getaip-cli -- \
  --native-bearer-token local-development-token \
  action status http://127.0.0.1:18080 \
  act_quickstart_health_001 \
```

```
  --include-result \
  --include-receipts
```

Expected result: the lifecycle view identifies the same action and capability, reports a terminal lifecycle state and `result_status` of `completed`, and includes the final result. This read model remains available after the client that submitted the action disconnects.

6. Inspect the MCP projection

The same daemon exposes a Streamable HTTP MCP endpoint backed by the AIP gateway. Initialize an MCP client and print its AIP manifest projection:

```
SH
cargo run --locked -p getaip-cli -- \
  mcp inspect --url http://127.0.0.1:18080/mcp
```

Expected result: JSON describing the MCP peer as an AIP manifest. This confirms the MCP mapping implemented by the local daemon; it is not an MCP conformance or independent-client qualification result.

7. Verify the complete path

Repeat the two read-only checks that prove the daemon is still ready and the action remains queryable:

```
SH
curl --fail --silent --show-error http://127.0.0.1:18080/ready >/dev/null
cargo run --locked -p getaip-cli -- \
  --native-bearer-token local-development-token \
  action status http://127.0.0.1:18080 \
  act_quickstart_health_001 \
  --include-result >/dev/null
```

Both commands must exit with status `0`. You have now completed discovery, authenticated submission, execution, durable lookup, and compatibility-profile inspection for one native AIP action.

Stop and clean up

Return to the first terminal and stop `getaip-server` with `Control-C`. Keep `.getaip-server-quickstart` if you want to restart the daemon and inspect the same action. When that local history is no longer needed, inspect and remove only this tutorial directory:

```
SH
du -sh .getaip-server-quickstart
rm -rf -- .getaip-server-quickstart
```

This deletion removes the tutorial's local action and replay history. It does not affect Cargo build artifacts or external services.

What happened internally

`getaip` wrapped the typed health `Action` in an AIP `Envelope` and sent it to `/aip/v1/messages` with the bearer credential. The daemon bound the request to the configured principal, validated and executed the built-in handler, stored the lifecycle records, and returned an `ActionResult`. The later status query read the stored operational view rather than replaying the action.

The MCP step initialized a compatibility client and projected the daemon's AIP manifest into MCP concepts. It did not create a second runtime or change the native action semantics.

Next steps

- Run the connector fleet quickstart

- Understand how AIP works
- Learn capabilities and contracts
- Use native AIP
- Prepare a production deployment