Create and manage a Cal.diy booking

Use this guide to create one booking through slot discovery, an immutable plan, trusted approval, a single commit attempt, and durable verification. It then shows how to choose a safe follow-up operation without replaying an uncertain provi

SECTION	Connectors
TYPE	Connector
PROTOCOL	AIP 1.0
SOURCE	docs/connectors/cal-diy/guides/create-and-manage-a-booking.md

This guide uses the `eventTypeId` branch and no custom duration. Use the capability reference when you need slug, team, recurrence, seating, routing, phone-only attendee, or custom-location input.

Prerequisites

You need:

- an admitted and healthy Cal.diy connector for the intended tenant;
- a native AIP endpoint and bearer token already bound to that tenant;
- a caller authorized to discover and invoke Cal.diy capabilities;
- a trusted approval process for tenant-policy decisions;
- a known Cal.diy event-type id;
- a durable place to retain action, transaction, plan, approval, idempotency,

and provider-operation identifiers.

Set client coordinates without putting provider credentials in the shell:

```
SH
export AIP_URL='https://aip.example.com'
export AIP_TOKEN_FILE='/run/secrets/aip-native-token'
export EVENT_TYPE_ID='42'
```

The deployment resolves tenant, account, connector route, and provider credential from authenticated context. Do not add them to booking input.

1. Confirm the admitted contracts

Inspect the two capabilities used before any mutation:

```
SH
aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  capability list "$AIP_URL" \
  --capability-id cap:cal_diy:slot.list \
  --limit 1

aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  capability list "$AIP_URL" \
  --capability-id cap:cal_diy:booking.create \
  --limit 1
```

Record the catalog revision plus each contract and schema digest. Confirm that `booking.create` is medium risk, approval-required, requires external-account idempotency, and requires plan before commit.

Catalog visibility proves admission at one revision. It does not prove that a connector replica or Cal.diy account is currently reachable.

2. Discover a bounded slot

Choose a future interval and write `slot-query.json`:

```
JSON
{
  "start": "2030-01-03",
  "end": "2030-01-04",
  "eventTypeId": 42,
  "timeZone": "Europe/London",
  "format": "time"
}
```

Submit a low-risk read with a stable action id:

```
SH
export SLOT_ACTION_ID='act_cal_booking_slots_001'

aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  action call "$AIP_URL" \
  cap:cal_diy:slot.list \
  --action-id "$SLOT_ACTION_ID" \
  --mode sync \
  --input @slot-query.json
```

Choose an exact date-time from the provider result. In the remaining examples, the chosen start is `2030-01-03T10:00:00Z`.

A slot result is an observation, not a hold. If the workflow needs a temporary provider reservation, use the separate reservation lifecycle and preserve its opaque uid. Booking create does not consume a reservation uid in its published input.

3. Freeze one booking input

Write `booking-create.json` once:

```
JSON
{
  "start": "2030-01-03T10:00:00Z",
  "eventTypeId": 42,
  "attendee": {
    "name": "Avery Patel",
    "email": "avery@example.com",
    "timeZone": "Europe/London"
  }
}
```

Keep this file byte-stable for the plan and commit. The runtime binds the capability and canonical input hash, not the filename. Reformatting equivalent JSON is normally canonicalized, but changing any semantic value requires a new plan and a newly reviewed commit.

Reserve stable identifiers and keep them together:

```
SH
export PLAN_ACTION_ID='act_cal_booking_plan_001'
export COMMIT_ACTION_ID='act_cal_booking_commit_001'
export TRANSACTION_ID='txn_cal_booking_001'
export BOOKING_KEY='cal-booking-001'
```

Never reuse `BOOKING_KEY` with different canonical input or another external account.

4. Create the downstream plan

Submit the plan action:

```
SH
aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  action call "$AIP_URL" \
  cap:cal_diy:booking.create \
  --action-id "$PLAN_ACTION_ID" \
  --idempotency-key "$BOOKING_KEY" \
  --transaction-mode plan \
  --transaction-id "$TRANSACTION_ID" \
  --input @booking-create.json
```

A successful result has validation class `downstream_slot_availability` and records `side_effect_committed: false`. It checks that the exact requested start appears in a one-day slot query. It does not reserve the slot.

Read the durable transaction and copy the returned plan id exactly:

```
SH
aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  transaction get "$AIP_URL" \
  --transaction-id "$TRANSACTION_ID" \
  --include-result \
  --include-receipts

export PLAN_ID='paste-the-returned-plan-id'
```

Do not infer a plan id from the transaction id. The implementation-generated plan expires after 15 minutes.

Understand the reviewed plan limit

The connector's create planner forwards an id selector and the requested start, but it does not forward `teamSlug`, `lengthInMinutes`, or nested `attendee.timeZone` into its slot probe. This guide avoids the first two fields; the attendee time zone remains required booking input but is not part of the reviewed provider slot query.

Treat the plan as an exact-start observation, not full validation of team, custom duration, time-zone interpretation, attendee details, or provider acceptance.

5. Submit the matching commit

Use a distinct commit action id, the same capability and input, the original transaction and idempotency key, and the returned plan id:

```
SH
aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  action call "$AIP_URL" \
  cap:cal_diy:booking.create \
  --action-id "$COMMIT_ACTION_ID" \
  --idempotency-key "$BOOKING_KEY" \
  --transaction-mode commit \
  --transaction-id "$TRANSACTION_ID" \
  --plan-id "$PLAN_ID" \
  --input @booking-create.json
```

The public commit path requires `plan_id`. The runtime rechecks the capability, input hash, planning principal, tenant and external account, transaction id, and plan expiry before it can claim the plan.

Without an admitted approval decision, the commit action enters `requires_human`. Store the returned approval id and leave the commit action in place. Do not submit a replacement commit.

6. Complete trusted approval

Use the deployment's approver-facing process to inspect the frozen request and submit a decision from an authenticated principal with tenant-policy authority. The decision needs the request's policy hash and required reason, input snapshot, and policy-decision evidence.

The Cal.diy approval validity window is 900,000 ms. The plan also expires after 15 minutes. If either boundary closes before commit ownership is claimed, stop and create a new plan for the current intent. Do not weaken policy or reuse an expired authorization.

Approval resumes the same queued commit. It does not authorize a second action, prove that the plan is still valid, or prove that Cal.diy accepted the booking.

7. Wait for the durable result

Read the original commit action independently of its submission connection:

```
SH
aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  action status "$AIP_URL" \
  "$COMMIT_ACTION_ID" \
  --include-result \
  --include-receipts \
  --wait-ms 30000
```

Proceed only after a terminal `completed` result. Retain the provider request id and provider operation id when present. The common connector output requires `status`, but it does not guarantee a booking-id field. Validate a non-empty provider booking uid from the actual response contract before continuing.

If the uid is absent, stop and treat the result as provider-contract drift. Do not derive a uid from the AIP action id, transaction id, or idempotency key.

8. Verify by reading the booking

Write `booking-get.json` with the validated provider uid:

```
JSON
{
  "booking_uid": "booking-provider-uid"
}
```

Read it under a new action id:

```
SH
export VERIFY_ACTION_ID='act_cal_booking_verify_001'

aipctl \
  --native-bearer-token-file "$AIP_TOKEN_FILE" \
  action call "$AIP_URL" \
  cap:cal_diy:booking.get \
  --action-id "$VERIFY_ACTION_ID" \
  --mode sync \
  --input @booking-get.json
```

Compare only the intended start, event type, attendee, and status fields exposed by the provider response. Keep the complete input, output, approval evidence, and receipts under restricted PII controls.

A completed create and matching read still do not prove that every calendar or notification was delivered.

9. Choose a follow-up operation

Use the provider booking uid and select the smallest operation that matches the new intent:

Intent	Capability	New plan required	Important boundary
Read current state	`booking.get`	No	Retry-safe read
Move the start	`booking.reschedule`	Yes	New exact-start plan and approval
Cancel	`booking.cancel`	Yes	Destructive; no rollback
Confirm pending booking	`booking.confirm`	No	Approval and stable mutation key
Decline pending booking	`booking.decline`	No	Destructive and approval-required
Change location	`booking.location.update`	No	Read preflight; no empty update
Add attendee or guests	`booking.attendee.add`, `booking.guest.add`	No	May affect calendars or notifications
Remove attendee	`booking.attendee.delete`	No	Destructive; no compensation
Change absence or owner	`booking.mark_absent`, `booking.reassign`	No	High risk and approval-required

Every mutation needs a new action id and a stable idempotency key scoped to that exact intent. Reschedule and cancel need their own current plan; the create plan cannot authorize either change.

Do not use the create capability's compensation declaration as ordinary cancellation. Compensation is a new, approval-required best-effort action against the original create and has a 24-hour contract window. It is neither automatic nor atomic.

Recover without replaying a mutation

Observation	Response
`connector.cal_diy.slot_unavailable` during plan	Choose a current start and create a new plan
Plan expired or input hash changed	Create a new plan; do not alter the old commit
Commit is `requires_human`	Complete the existing approval workflow
Commit is pending or in flight	Read the same action and transaction; wait
`connector.cal_diy.idempotency_collision`	Stop; the key already owns different input
`connector.cal_diy.outcome_unknown` or ambiguous transport failure	Reconcile the original provider operation
Definitive provider rejection with no uncertain outcome	Correct the cause and begin a new governed intent

The CLI at this revision does not expose transaction mode `reconcile`, even though the protocol and runtime define it. Use the deployment's authorized reconciliation integration and retain the original action, transaction, idempotency, and provider-operation identities. Never replace the key to force another provider call.

Verify the complete workflow

Before treating the task as complete, confirm that you retained:

- the catalog revision plus contract and schema digests;
- slot-read action and exact selected start;
- immutable booking input and its hash;
- plan action, plan id, transaction id, and expiry;
- approval id, policy hash, authority evidence, and terminal decision;
- commit action, original idempotency key, durable result, and receipts;
- provider request and operation ids when present;
- validated provider booking uid and the follow-up read.

This evidence demonstrates one governed application workflow. It does not by itself establish live-provider qualification, notification delivery, calendar delivery, availability guarantees, or production readiness.

Related documentation

- Booking capability reference
- Slots and reservations
- Approvals and policy
- Transactions and compensation
- Observe and recover