Embabel Agent Cookbook

Rod Johnson; Jasper Blues; Igor Dayen; Alexander Hein-Heifetz

2026-07-04

Table of Contents

Chapter 1. Introduction

Executable recipes for building agents with Embabel.

For contributors, the chapter manuscript files use lower-kebab-case and the matching test classes use UpperCamelCase; `book.adoc` controls the chapter order.

Part 1: Agent Behavior

This part covers planner behavior: type chaining, conditions, stuck state, heuristics, looping, and debugging.

Chapter 2. Action Domain Type Chaining

2.1. Introduction

This chapter shows how an agent can move through a typed chain of actions without ad hoc branching. The example starts with a `UserInput`, narrows it into a request type, turns that into a concrete travel activity, and finishes with a travel recommendation. The main idea is that action return types drive the next step in the plan. `TravelRequest` is the common input shape, `FlightRequest` and `ItineraryRequest` split the flow, and `Flight` or `Itinerary` becomes the intermediate domain result before the final `TravelRecommendation`.

2.2. Key Concepts

`@Agent` marks the class that Spring discovers and Embabel deploys.

`@Action` marks each step in the plan and gives the planner a typed transition.

`@AchievesGoal` marks the terminal action that satisfies the invocation goal.

Embabel uses a planner behind the scenes. The default planner is GOAP. This chapter only introduces the idea; more planner detail comes later.

`Ai.withDefaultLlm().createObject(…)` is the LLM boundary in this chapter. Each action uses it to turn prompt text into a typed result.

2.3. How It Works

2.3.1. Bootstrap

The shared test application is a normal Spring Boot entry point for the cookbook test sources. It is the only test bootstrap the chapter needs.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles({"cookbook-test", "action-domain-type-chaining"})
class ActionDomainTypeChainingTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private AgentPlatform agentPlatform;
```

It scans `com.embabel.cookbook` so the chapter agent is discovered as a Spring bean and then deployed into the `AgentPlatform`. The agent is deployed automatically because the Spring scanner sees the `@Agent` stereotype.

2.3.2. Typed Flow

The action chain is intentionally typed so the planner can move from one domain object to the next. The first action classifies the user request, the next action returns either a `Flight` or an `Itinerary`, and the final action converts that into a `TravelRecommendation`.

```
    @Action(description = "Classify the user input as a flight or itinerary request")
①
    TravelRequest classifyRequest(UserInput userInput) {
        var requestType = ai.withDefaultLlm()
                .createObject("""
                        You are a helpful travel assistant classifying whether the
request needs a flight or an itinerary.

                        # User input
                        %s
                        """.formatted(userInput.getContent()).trim(),
TravelRequestType.class); ②

        return switch (requestType) {
            case FLIGHT -> new FlightRequest(userInput);
            case ITINERARY -> new ItineraryRequest(userInput);
        };
    }
    @Action(description = "Find a flight based on user input") ③
    Flight findFlight(FlightRequest request) {
        return ai.withDefaultLlm()
                .createObject("""
                        You are a travel assistant. Always recommend a specific flight
to go to. Use airport codes.

                        # User input
                        %s
                        """.formatted(request.userInput().getContent()).trim(),
Flight.class);
    }
    @Action(description = "Build a travel itinerary based on user input") ④
    Itinerary buildItinerary(ItineraryRequest request) {
        return ai.withDefaultLlm()
                .createObject("""
                        You are a travel assistant, so build a concise itinerary based
on the user input.

                        # User input
                        %s
                        """.formatted(request.userInput().getContent()).trim(),
Itinerary.class);
    }
    @AchievesGoal(description = "The user has received a travel recommendation") ⑤
    @Action(description = "Summarize the selected travel activity")
```

```
    TravelRecommendation summarize(TravelActivity activity) {
        return ai.withDefaultLlm()
                .createObject("""
                        Summarize this travel recommendation for the user.

                        # Recommendation
                        %s
                        """.formatted(activity.recommendation()).trim(),
TravelRecommendation.class);
    }
```

① `classifyRequest` is the first typed action; it turns raw `UserInput` into a `TravelRequest`.

② The LLM classifies the request as a `TravelRequestType`, which determines the next domain object.

③ `findFlight` is eligible only when the classifier returned a `FlightRequest`.

④ `buildItinerary` is eligible only when the classifier returned an `ItineraryRequest`.

⑤ `summarize` is the goal action; it converts the selected `TravelActivity` into the final `TravelRecommendation`.

The important part is the type flow: `UserInput` → `TravelRequest` → `FlightRequest` or `ItineraryRequest` → `Flight` or `Itinerary` → `TravelRecommendation`. That flow is what the planner follows when it chooses the next action. When the return type changes, the planner can advance to the next action without hard-coded branching.

2.3.3. Test Wiring

The chapter test exercises the chain from the outside. It autowires `AgentPlatform`, which is the entry point for runtime invocation. It also uses `Ai` inside the agent actions so each typed step can call the configured LLM directly. The default LLM comes from `src/test/resources/application-cookbook-test.properties`, which is loaded by the `cookbook-test` profile. That file sets `embabel.models.default-llm=gpt-4.1-mini` for the chapter tests. You can make other LLM provider's model as default by changing the `embabel.models.default-llm` property and updating the `pom.xml` with the corresponding supported module from (here)[https://github.com/embabel/embabel-agent/blob/main/embabel-agent-starters/pom.xml#L22].

```
    @Test
    void flightRequestChainsThroughFlightActivity() {
        logger.info("Running flight request domain type chaining test");

        var recommendation = AgentInvocation.create(agentPlatform,
TravelRecommendation.class) ①
                .invoke(new UserInput("Find me a flight from New York to London"));

        logger.info("Flight recommendation: {}", recommendation.text());
        assertNotNull(recommendation);
        assertFalse(recommendation.text().isBlank());
    }
    @Test
```

```
    void itineraryRequestChainsThroughItineraryActivity() {
        logger.info("Running itinerary request domain type chaining test");

        var recommendation = AgentInvocation.create(agentPlatform,
TravelRecommendation.class) ①
                .invoke(new UserInput("Plan a day in London around museums and
food"));

        logger.info("Itinerary recommendation: {}", recommendation.text());
        assertNotNull(recommendation);
        assertFalse(recommendation.text().isBlank());
    }
```

① `AgentInvocation` starts from the target result type and lets the planner find a path from `UserInput` to `TravelRecommendation`.

The test only checks that the result exists and is nonblank. That keeps the recipe stable even though the model output itself can vary. The included source snippets show the bootstrapped agent, the injected `Ai`, the action boundaries, the LLM call, and the `AgentInvocation` entry point.

2.3.4. LLM Invocation

Each action uses `ai.withDefaultLlm().createObject(…)` to turn prompt text into a typed domain object. This is the key boundary in the recipe: the agent chooses the step, and the LLM fills in the typed result for that step. The model choice itself is not hard-coded in the action; it comes from the active test profile configuration.

2.4. Conclusion

This recipe demonstrates how typed return values drive the planner forward without hard-coded branching. Each action's return type determines what the next step can be, and `AgentInvocation` runs the full typed chain end to end.

Chapter 3. Action Condition

3.1. Introduction

This chapter shows how Embabel uses `@Condition` to choose between mutually exclusive actions. The user asks for either a direct flight or a flight with at least one stop, and the planner picks the matching action after the request is analyzed. The main idea is straightforward: the agent analyzes the user input, sets the relevant conditions on the blackboard, and then the planner chooses exactly one branch. If the user asks for a direct flight, only the direct-flight action is eligible. If the user asks for a flight with stops, only the with-stops action is eligible.

3.2. Key Concepts

`@Agent` marks the class that Spring discovers and Embabel deploys for this chapter.

`@Condition` returns a boolean gate that the planner can use before it executes the next action.

The chapter uses two branch actions: one for direct flights and one for flights with stops.

The branch actions return a chapter-local `FlightPlan`, which keeps the summary step deterministic and avoids polluting the shared travel domain.

- `@Agent` marks the Spring component that Embabel deploys.
- `@Condition` marks the boolean gate that lets the planner choose a branch.
- `FlightPlan` is the chapter-local branch output that keeps the summary deterministic.
- The two branch actions are mutually exclusive and model the direct-flight and with-stops choices.

3.3. How It Works

3.3.1. Bootstrap

The chapter uses the same shared cookbook test bootstrap as the rest of the book. Spring scans the cookbook test package, Embabel deploys the chapter agent, and the test invokes the agent through `AgentInvocation`.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles({"cookbook-test", "action-condition"})
class ActionConditionTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private AgentPlatform agentPlatform;
```

3.3.2. Request Analysis

The first action reads `UserInput` and turns it into a small routing request. That request carries the direct-flight decision that later conditions use.

```
record FlightRoutingRequest(UserInput userInput, boolean directFlightOnly) {

    boolean stopsAllowed() {
        return !directFlightOnly;
    }
}
```

The user input is not enough by itself. The chapter converts it into an explicit request object so the planner can reason over a concrete fact instead of a prompt guess.

3.3.3. Branching

The agent sets two conditions from the routing request:

- `directFlightRequested`
- `stopsAllowed`

The planner then chooses the matching action.

```
    FlightRoutingRequest analyzeFlightRequest(UserInput userInput, OperationContext
context) {
        var request = new FlightRoutingRequest(userInput, isDirectFlightOnly(
userInput.getContent()));
        context.setCondition("directFlightRequested", request.directFlightOnly());
        context.setCondition("stopsAllowed", request.stopsAllowed());
        return request;
    }
    @Condition ①
    boolean directFlightRequested(FlightRoutingRequest request, OperationContext
context) {
        logger.info("Evaluating direct-flight request against blackboard: {}",
context.getObjects());
        return request.directFlightOnly();
    }
    @Condition ②
    boolean stopsAllowed(FlightRoutingRequest request, OperationContext context) {
        logger.info("Evaluating with-stops request against blackboard: {}", context
.getObjects());
        return request.stopsAllowed();
    }
    @Action(description = "Build a direct flight", pre = {"directFlightRequested"})
    FlightPlan buildDirectFlight(FlightRoutingRequest request, OperationContext
context) {
        logger.info("User intent is classified as a request for direct flight.");
```

```
        return context.ai()
                .withDefaultLlm()
                .createObject("""
                        You are a travel assistant. Recommend a specific direct
flight.
                        Do not include any stopover or connection city.
                        Use airport codes.

                        # User input
                        %s
                        """.formatted(request.userInput().getContent()).trim(),
DirectFlightPlan.class);
    }
    @Action(description = "Build a flight with stops", pre = {"stopsAllowed"})
    FlightPlan buildFlightWithStops(FlightRoutingRequest request, OperationContext
context) {
        logger.info("User intent is classified as a request for flight with stops.");
        return context.ai()
                .withDefaultLlm()
                .createObject("""
                        You are a travel assistant. Recommend a flight with at least
one stop.
                        Include exactly one connection city.
                        Do not recommend a nonstop or direct flight.
                        Use airport codes.

                        # User input
                        %s
                        """.formatted(request.userInput().getContent()).trim(),
ConnectingFlightPlan.class);
    }
    @Action(description = "Summarize the flight")
    @AchievesGoal(description = "The user has received a flight recommendation")
    TravelRecommendation summarize(FlightPlan flightPlan, OperationContext context) {
        return new TravelRecommendation(flightPlan.recommendation());
    }
```

① `directFlightRequested` is true only when the analyzed request requires a direct or nonstop flight.

② `stopsAllowed` is true when the analyzed request permits a connecting flight.

The interesting part is the branch selection:

- direct requests execute `buildDirectFlight`
- requests that allow stops execute `buildFlightWithStops`

The branch actions return chapter-local plan types instead of forcing the shared domain to carry chapter-specific behavior. The tagged source snippets show the analysis action, the two `@Condition` gates, the two branch actions, and the final summary step.

```
sealed interface FlightPlan permits DirectFlightPlan, ConnectingFlightPlan {

    String recommendation();
}
record DirectFlightPlan(String flightNumber, String departure, String arrival)
implements FlightPlan {

    @Override
    public String recommendation() {
        return "Direct flight %s departing from %s and arriving at %s.".formatted(
                flightNumber,
                departure,
                arrival
        );
    }
}
record ConnectingFlightPlan(String flightNumber, String departure, String
connectionCity, String arrival)
        implements FlightPlan {

    @Override
    public String recommendation() {
        return "Connecting flight %s departing from %s, stopping in %s, and arriving
at %s.".formatted(
                flightNumber,
                departure,
                connectionCity,
                arrival
        );
    }
}
```

3.3.4. Test Wiring

The test exercises both branches independently. One test feeds the direct-flight wording, the other feeds the with-stops wording. Both check that the process completes and that the recommendation is nonblank.

```
    @Test
    void directFlightOnlyCompletes() {
        logger.info("Running direct-flight condition test");

        var process = AgentInvocation.create(agentPlatform, TravelRecommendation.
class) ①
                .runAsync(new UserInput("Find me a direct flight from New York to
London"))
                .join();

        logger.info("Direct-flight blackboard: {}", process.getBlackboard().
```

```
infoString(true, 1));
        logger.info("Direct-flight history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.COMPLETED, process.getStatus());
        var recommendation = process.last(TravelRecommendation.class);
        assertNotNull(recommendation);
        assertFalse(recommendation.text().isBlank());
    }
    @Test
    void flightWithStopsCompletes() {
        logger.info("Running with-stops condition test");

        var process = AgentInvocation.create(agentPlatform, TravelRecommendation.
class)
                .runAsync(new UserInput("Find me a flight from New York to London with
stops"))
                .join();

        logger.info("With-stops blackboard: {}", process.getBlackboard().infoString
(true, 1));
        logger.info("With-stops history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.COMPLETED, process.getStatus());
        var recommendation = process.last(TravelRecommendation.class);
        assertNotNull(recommendation);
        assertFalse(recommendation.text().isBlank());
    }
```

① `AgentInvocation` asks the planner for a `TravelRecommendation`; the conditions decide which branch can run for the supplied input.

3.4. Conclusion

This recipe demonstrates practical `@Condition` usage. The agent reads a structured request, the planner evaluates the condition, and exactly one branch runs before the final recommendation is produced.

Chapter 4. Agent Stuck State

4.1. Introduction

This chapter shows what happens when the planner runs out of eligible actions. The agent first classifies the user input into either `TravelInquiry` or `NonTravelInquiry`. If the input looks like travel, the planner can continue and produce a travel recommendation. If the input is not travel-related, the classifier returns `NonTravelInquiry` and no action matches it, so the process becomes stuck. The chapter also logs the blackboard through `OperationContext` so you can see the facts the planner is reasoning over. In the stuck case, the process does not fail before the first action. It fails after `classifyRequest` has run, returned `NonTravelInquiry`, and left `buildTravelRecommendation` with no matching `TravelInquiry`.

4.2. Key Concepts

`@Agent` marks the class that Spring discovers and Embabel deploys for this chapter.

The chapter does not need a separate `@Condition`; the type returned by the classifier determines whether the next action is eligible.

`OperationContext` exposes the blackboard so the agent can log the facts that are already available.

`STUCK` is the state the planner reaches when no remaining action can satisfy the current blackboard facts.

- `@Agent` marks the Spring component that Embabel deploys.
- `OperationContext` gives the agent access to the blackboard for logging the current facts.
- `TravelInquiry` and `NonTravelInquiry` model the two branches of the classifier.
- `STUCK` is the failure state used when no eligible action remains.

4.3. How It Works

4.3.1. Bootstrap

The chapter uses the shared cookbook test bootstrap. Spring scans the cookbook test package, Embabel deploys the chapter agent, and the test invokes the agent through `AgentInvocation`.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles({"cookbook-test", "action-stuck-state"})
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class AgentStuckStateTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());
```

```
    @Autowired
    private AgentPlatform agentPlatform;
```

4.3.2. Blackboard Logging

The first action inspects the user input, classifies it, and logs the current blackboard. That is the point where you can see exactly what the planner knows before it tries to continue.

```
    @Action(description = "Classify whether the request is travel-related")
    RequestClassification classifyRequest(UserInput userInput, OperationContext
context) {
        var request = isTravelRequest(userInput.getContent())
                ? new TravelInquiry(userInput)
                : new NonTravelInquiry(userInput);
        logger.info("Stuck-state blackboard: {}", context.getObjects());
        return request;
    }
```

The analysis step logs the blackboard through `OperationContext`, so you can inspect the current facts before the planner continues.

4.3.3. Recommendation Action

If the classifier returns a travel request, the planner can continue into the recommendation action. That action is the only one eligible in the positive case.

```
    @Action(description = "Build a travel recommendation")
    @AchievesGoal(description = "The user has received a travel recommendation")
    TravelRecommendation buildTravelRecommendation(TravelInquiry request,
OperationContext context) {
        return context.ai()
                .withDefaultLlm()
                .createObject("""
                        You are a travel assistant. Write a short travel
recommendation.

                        # User input
                        %s
                        """.formatted(request.userInput().getContent()).trim(),
TravelRecommendation.class);
    }
```

4.3.4. Stuck Path

The travel request test completes because `classifyRequest` returns `TravelInquiry`, which makes `buildTravelRecommendation` eligible. The non-travel test gets stuck after `classifyRequest` returns

`NonTravelInquiry`. At that point the planner reevaluates the blackboard, sees that `buildTravelRecommendation` requires a `TravelInquiry`, and has no eligible next action left. The invocation from `TravelRecommendation` starts the run from the target output type, but the dog input is not travel-related, so the planner reaches `STUCK` immediately after the classification step. The `<1>` marker on the test line points to the invocation that starts the run.

stuck at: `keen_wilbur` has a `NonTravelInquiry`, so no action can continue.

```
20:37:59.299 [main] INFO  AgentStuckStateTest - Running dog stuck-state test
20:37:59.319 [main] INFO  Embabel - [interesting_lewin] created
20:37:59.320 [main] INFO  Embabel - [interesting_lewin] object added: UserInput
20:37:59.326 [embabel-platform-0] INFO  Embabel - [interesting_lewin] ready to plan
from:
  {it:com.embabel.agent.domain.io.UserInput=TRUE,
it:com.embabel.cookbook.RequestClassification=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.classifyRequest=FALSE,
it:com.embabel.cookbook.TravelInquiry=FALSE,
it:com.embabel.cookbook.travel.domain.TravelRecommendation=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.buildTravelRecommendation=FALSE,
it:com.embabel.cookbook.NonTravelInquiry=FALSE, it:java.lang.Record=FALSE}
20:37:59.356 [embabel-platform-0] INFO  Embabel - [interesting_lewin] formulated plan:
  com.embabel.cookbook.ActionStuckStateAgent.classifyRequest ->
com.embabel.cookbook.ActionStuckStateAgent.buildTravelRecommendation
  from:
  {it:com.embabel.agent.domain.io.UserInput=TRUE,
it:com.embabel.cookbook.RequestClassification=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.classifyRequest=FALSE,
it:com.embabel.cookbook.TravelInquiry=FALSE,
it:com.embabel.cookbook.travel.domain.TravelRecommendation=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.buildTravelRecommendation=FALSE,
it:com.embabel.cookbook.NonTravelInquiry=FALSE, it:java.lang.Record=FALSE}
20:37:59.357 [embabel-platform-0] INFO  Embabel - [interesting_lewin] executing action
com.embabel.cookbook.ActionStuckStateAgent.classifyRequest
20:37:59.361 [embabel-platform-0] INFO  ActionStuckStateAgent - Stuck-state
blackboard: [UserInput(content=Tell me about my dog, timestamp=2026-07-
08T00:37:59.303086564Z)]
20:37:59.362 [embabel-platform-0] INFO  Embabel - [interesting_lewin] object bound
it:NonTravelInquiry
20:37:59.363 [embabel-platform-0] INFO  Embabel - [interesting_lewin] executed action
com.embabel.cookbook.ActionStuckStateAgent.classifyRequest in PT0.001S
20:37:59.365 [embabel-platform-0] INFO  Embabel - [interesting_lewin] ready to plan
from:
  {it:com.embabel.agent.domain.io.UserInput=TRUE,
it:com.embabel.cookbook.RequestClassification=TRUE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.classifyRequest=TRUE,
it:com.embabel.cookbook.TravelInquiry=FALSE,
it:com.embabel.cookbook.travel.domain.TravelRecommendation=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.buildTravelRecommendation=FALSE,
it:com.embabel.cookbook.NonTravelInquiry=TRUE, it:java.lang.Record=TRUE}
20:37:59.389 [embabel-platform-0] INFO  Embabel - [interesting_lewin] stuck at:
```

```
  {it:com.embabel.agent.domain.io.UserInput=TRUE,
it:com.embabel.cookbook.RequestClassification=TRUE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.classifyRequest=TRUE,
it:com.embabel.cookbook.TravelInquiry=FALSE,
it:com.embabel.cookbook.travel.domain.TravelRecommendation=FALSE,
hasRun_com.embabel.cookbook.ActionStuckStateAgent.buildTravelRecommendation=FALSE,
it:com.embabel.cookbook.NonTravelInquiry=TRUE, it:java.lang.Record=TRUE}
20:37:59.389 [embabel-platform-0] WARN  SimpleAgentProcess - Process interesting_lewin
is stuck with no StuckHandler. This may or may not be an error. History (1):
    com.embabel.cookbook.ActionStuckStateAgent.classifyRequest
```

```
    @Test
    void travelRequestCompletes() {
        logger.info("Running travel-request stuck-state test");

        var process = AgentInvocation.create(agentPlatform, TravelRecommendation.
class)
                .runAsync(new UserInput("Plan a travel itinerary from London to
Paris"))
                .join();

        logger.info("Travel blackboard: {}", process.getBlackboard().infoString(true,
1));
        logger.info("Travel history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.COMPLETED, process.getStatus());
        var recommendation = process.last(TravelRecommendation.class);
        assertNotNull(recommendation);
        assertFalse(recommendation.text().isBlank());
    }
    @Test
    void dogRequestStaysStuck() {
        logger.info("Running dog stuck-state test");

        var process = AgentInvocation.create(agentPlatform, TravelRecommendation.
class) ①
                .runAsync(new UserInput("Tell me about my dog"))
                .join();

        logger.info("Dog blackboard: {}", process.getBlackboard().infoString(true,
1));
        logger.info("Dog history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.STUCK, process.getStatus());
        assertEquals(1, process.getHistory().size());
        assertEquals("com.embabel.cookbook.ActionStuckStateAgent.classifyRequest",
process.getHistory().getFirst().getActionName());
```

```
    }
```

① `AgentInvocation` starts the run from `TravelRecommendation`; the dog input is not travel-related, so the classifier returns `NonTravelInquiry` and the process reaches `STUCK`.

4.3.5. Request Model

The chapter uses a tiny pair of request types to model the branch. That keeps the failure case deterministic and easy to read.

```
interface RequestClassification {
}

record TravelInquiry(UserInput userInput) implements RequestClassification {
}

record NonTravelInquiry(UserInput userInput) implements RequestClassification {
}
```

4.4. Conclusion

This recipe shows the planner failure mode in a controlled way. You can see the blackboard before the branch decision, and you can see `STUCK` when no action is eligible to continue the run.

Chapter 5. Action Heuristics

5.1. Introduction

This chapter shows how Embabel chooses between two eligible actions that have the same input and output shape. The difference is not a condition gate. It is the action cost. Both actions are eligible, and the planner prefers the cheaper action when both can satisfy the goal.

5.2. Key Concepts

`cost` gives the planner a static preference for one action over another.

`@AchievesGoal` tells the planner that either action can finish the run.

5.3. How It Works

5.3.1. Bootstrap

The chapter uses the shared cookbook test bootstrap. Spring loads the cookbook application and the heuristics profile, then Embabel deploys the agent.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles({"cookbook-test", "action-heuristics"})
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class ActionHeuristicsTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private AgentPlatform agentPlatform;
```

5.3.2. Main Flow

Both actions accept the same `UserInput` and return the same `TravelRecommendation`. The only difference is the cost value, so the planner picks the cheaper branch.

```
@Agent(description = "Show how action cost steers the planner", planner = PlannerType
.GOAP)
@Profile("action-heuristics")
class ActionHeuristicsAgent {
```

```
    @AchievesGoal(description = "The user has received a travel recommendation")
    @Action(description = "Build the cheapest recommendation", cost = 0.1)
    TravelRecommendation buildCheapestRecommendation(UserInput userInput) {
        return new TravelRecommendation("Cheapest travel recommendation selected.");
    }

    @AchievesGoal(description = "The user has received a travel recommendation")
    @Action(description = "Build the premium recommendation", cost = 0.9)
    TravelRecommendation buildPremiumRecommendation(UserInput userInput) {
        return new TravelRecommendation("Premium travel recommendation selected.");
    }
}
```

5.3.3. Test Wiring

The test asks for a travel recommendation and checks that the cheaper action was the one executed.

```
    @Test
    void plannerChoosesCheapestAction() {
        logger.info("Running action heuristics test");

        var process = AgentInvocation.create(agentPlatform, TravelRecommendation.
class)
                .runAsync(new UserInput("Choose a travel recommendation"))
                .join();

        logger.info("Heuristics blackboard: {}", process.getBlackboard().infoString
(true, 1));
        logger.info("Heuristics history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.COMPLETED, process.getStatus());
        assertEquals(1, process.getHistory().size());
        assertEquals
("com.embabel.cookbook.ActionHeuristicsAgent.buildCheapestRecommendation", process
.getHistory().getFirst().getActionName());
        var recommendation = process.last(TravelRecommendation.class);
        assertNotNull(recommendation);
        assertEquals("Cheapest travel recommendation selected.", recommendation.
text());
    }
```

5.4. Conclusion

`cost` is the right tool when multiple actions are eligible and one should be preferred. This chapter keeps the comparison deterministic by using two actions with the same signature and different static costs.

Chapter 6. Repeat Until Acceptable

6.1. Introduction

This chapter shows the repeat-until pattern Embabel uses when one pass is not enough. The agent generates a travel itinerary, evaluates it, and repeats until the score crosses an acceptance threshold. The loop is deterministic here because the evaluator is part of the code, not an external judgment call.

6.2. Key Concepts

`RepeatUntilAcceptableBuilder` builds a loop that keeps revising a result until the evaluator accepts it.

`Feedback` is the loop contract. `TextFeedback` is the framework's built-in score-plus-message feedback type in this API. It is not the business result; it is the value returned by the evaluator lambda inside `withEvaluator(…)`. The evaluator returns `TextFeedback(score, feedback)`. `withScoreThreshold(0.8)` reads `Feedback.getScore()` from that object. The loop compares that score to `0.8` to decide whether to repeat or stop, and it carries the `feedback` text into the next revision prompt. In this API jar, `Feedback` is the interface and `TextFeedback` is the concrete implementation shown by the repeat-until examples.

`withScoreThreshold(0.8)` compares against `Feedback.getScore()`. That means any score below `0.8` repeats, and any score at or above `0.8` stops the loop. `withMaxIterations` is the safety cap that stops the loop even if the score never reaches the threshold.

6.3. How It Works

6.3.1. Bootstrap

The chapter uses the shared cookbook test bootstrap. Spring loads the cookbook application and the repeat-until profile, then Embabel deploys the agent.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles({"cookbook-test", "repeat-until-acceptable"})
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class RepeatUntilAcceptableTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private AgentPlatform agentPlatform;
```

6.3.2. Main Flow

The action builds a repeat-until-acceptable sub-process. It revises the itinerary, evaluates the latest attempt, and stops once the score reaches the threshold. The loop returns an `Itinerary`, not a `TextFeedback`, because the feedback is internal control data while the itinerary is the actual business result.

```
@Agent(description = "Show how repeat-until acceptable retries a result until it
passes the score threshold")
@Profile("repeat-until-acceptable")
class RepeatUntilAcceptableAgent {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @AchievesGoal(description = "The user has received an acceptable itinerary")
    @Action(description = "Revise the itinerary until it is acceptable")
    Itinerary reviseUntilAcceptable(UserInput userInput, ActionContext actionContext)
{ ①
        return RepeatUntilAcceptableBuilder
                .returning(Itinerary.class) ②
                .withMaxIterations(3) ③
                .withScoreThreshold(0.8) ④
                .repeating(context -> {
                    var lastAttempt = context.lastAttempt();
                    var itineraryText = lastAttempt == null
                            ? """
                            Day 1: Depart from London and arrive in Paris.
                            """.strip()
                            : """
                            Day 1: Depart from London and arrive in Paris.
                            Day 2: Visit the Eiffel Tower and the Louvre.
                            Day 3: Take a Seine cruise and depart.
                            """.strip();
                    logger.info("Repeat-until attempt: {}", itineraryText);
                    return new Itinerary(itineraryText);
                }) ⑤
                .withEvaluator(context -> {
                    var text = context.getResultToEvaluate().text();
                    var score = text.contains("Day 2") && text.contains("Day 3") ? 0.9
: 0.2;
                    var feedback = score >= 0.8 ? "Three-day itinerary accepted" :
"Needs the full three-day itinerary";
                    logger.info("Repeat-until evaluation: score={} feedback={}",
score, feedback);
                    return new TextFeedback(score, feedback);
                }) ⑥
                .build() ⑦
                .asSubProcess(actionContext, Itinerary.class); ⑧
    }
```

```
}
```

① `reviseUntilAcceptable` is the action entry point that owns the repeat-until loop.

② `returning(Itinerary.class)` says the loop's final product is an itinerary.

③ `withMaxIterations(3)` caps the number of drafts the loop may produce.

④ `withScoreThreshold(0.8)` is the accept line: scores below it repeat, scores at or above it stop the loop.

⑤ `repeating(…)` generates each itinerary draft, using `lastAttempt()` to decide whether this is the first pass or a revision. `lastAttempt()` returns the previous loop `Attempt`, not `TextFeedback`. That `Attempt` contains the earlier `Itinerary` result and the `TextFeedback` that was returned for it. On the first pass it is `null`, so the action emits only `Day 1`.

⑥ `withEvaluator(…)` scores the latest itinerary and returns `TextFeedback`. The first draft gets `0.2`, which is below the `0.8` threshold, because it only contains `Day 1`. The revision gets `0.9`, which is above the threshold, because it contains the full three-day itinerary. That is why the loop accepts the revised itinerary and stops. `TextFeedback` carries the numeric score plus the explanatory message. The evaluator creates it. The loop reads only the score for control through `withScoreThreshold(0.8)`. The `feedback` message is carried forward as guidance for the next revision. `TextFeedback` is the evaluator output, while `lastAttempt()` is the previous input-output pair that lets the repeater see what happened last time.

⑦ `build()` turns the loop definition into an executable scope.

⑧ `asSubProcess(…)` runs the loop inside the enclosing action and returns the final `Itinerary`.

6.3.3. Test Wiring

The test starts the agent, waits for the sub-process to complete, and checks that the acceptable itinerary won the loop. The test proves two things: the loop finishes as `COMPLETED`, and the returned itinerary is the revised version that reached the acceptance score. The final itinerary is the full three-day version, because that is the one the evaluator scores above the threshold. The test does not assert the intermediate draft directly; the intermediate draft is only visible through the evaluation log.

```
    @Test
    void loopRetriesUntilTheResultIsAcceptable() {
        logger.info("Running repeat-until acceptable test");

        var process = AgentInvocation.create(agentPlatform, Itinerary.class) ①
                .runAsync(new UserInput("Plan a 3-day itinerary from London to
Paris"))
                .join();

        logger.info("Loop blackboard: {}", process.getBlackboard().infoString(true,
1));
        logger.info("Loop history: {}", process.getHistory().stream().map
(ActionInvocation::getActionName).toList());

        assertEquals(AgentProcessStatusCode.COMPLETED, process.getStatus());
```

```
        assertEquals(1, process.getHistory().size());
        assertEquals
("com.embabel.cookbook.RepeatUntilAcceptableAgent.reviseUntilAcceptable",
                process.getHistory().getFirst().getActionName());
        var itinerary = process.last(Itinerary.class);
        assertNotNull(itinerary);
        assertEquals("""
                Day 1: Depart from London and arrive in Paris.
                Day 2: Visit the Eiffel Tower and the Louvre.
                Day 3: Take a Seine cruise and depart.
                """.strip(), itinerary.text());
    }
```

① `AgentInvocation` starts the loop from `Itinerary`; the travel input drives the repeat-until run. The repeated drafts do not appear as separate top-level actions in the history, because the loop is running inside the single `reviseUntilAcceptable` action.

6.4. Conclusion

The repeat-until-acceptable pattern is the right choice when the first itinerary may be close, but not good enough. The loop keeps the work local to the agent and makes the acceptance rule explicit.

Chapter 7. Agent Debugging

7.1. Introduction

When an agent application starts correctly, a lot of wiring happens before the first invocation runs. This chapter traces that boot path, from the starter dependency in `pom.xml` through the auto-configurations, property loaders, and metadata readers that make agent deployment work. The boundary matters: Spring still creates the `@Agent` bean and manages its lifecycle, while Embabel reads the annotations and deploys the bean as an agent.

7.2. Key Concepts

`com.embabel.agent:embabel-agent-starter-openai` is the cookbook dependency that pulls in the platform and OpenAI auto-configuration.

`AgentOpenAiAutoConfiguration` and `AgentPlatformAutoConfiguration` are the auto-configuration entry points that Spring Boot imports during startup.

`AgentPlatformPropertiesLoader`, `AgentPlatformProperties`, and `OpenAiProperties` are the property-loading and property-binding types that turn configuration files into runtime settings.

`ScanConfiguration`, `AgentScanningPostProcessorDelegate`, `AgentMetadataReader`, `DelegatingAgentScanningBeanPostProcessor`, and `AgentDeployer` are the types that discover agent classes and deploy them into the platform. The `@Agent` class is still a normal Spring bean, so the Spring container is responsible for instantiating it, injecting its dependencies, and keeping it in the application context.

7.3. How It Works

- `pom.xml` declares `com.embabel.agent:embabel-agent-starter-openai`, so the cookbook app gets both the platform runtime and the OpenAI model wiring on the classpath.
- Spring Boot reads `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` from the starter dependencies and imports `com.embabel.agent.autoconfigure.models.openai.AgentOpenAiAutoConfiguration` before `com.embabel.agent.autoconfigure.platform.AgentPlatformAutoConfiguration`.
- `AgentOpenAiAutoConfiguration` imports `com.embabel.agent.config.models.openai.OpenAiModelsConfig`, which binds `OpenAiProperties` under `embabel.agent.platform.models.openai.*` and provides the OpenAI model setup used by the platform.
- `AgentPlatformAutoConfiguration` imports `CommonPlatformPropertiesLoader`, `ScanConfiguration`, `AgentPlatformConfiguration`, `ToolGroupsConfiguration`, and `EmbeddingTrackingConfiguration`.
- `CommonPlatformPropertiesLoader` and `AgentPlatformPropertiesLoader` load the platform properties from classpath resources; `AgentPlatformPropertiesLoader` reads `classpath:agent-platform.properties` and `classpath:agent-application.properties`.

- `ScanConfiguration` applies `@ConfigurationPropertiesScan` and `@ComponentScan` to the core Embabel packages: `com.embabel.agent.api`, `com.embabel.agent.core`, `com.embabel.agent.experimental`, `com.embabel.agent.prompt`, `com.embabel.agent.spi`, `com.embabel.agent.test`, `com.embabel.agent.tools`, and `com.embabel.agent.web`.
- `AgentPlatformProperties` binds the platform configuration tree, including `scanning`, `ranking`, `llmOperations`, `processIdGeneration`, `autonomy`, `models`, `sse`, `rest`, `test`, `actionQos`, and `threading`.
- `AgentPlatformConfiguration` contributes the core runtime beans: `nameGenerator`, `agentInstrumentation`, `toolDecorator`, `templateRenderer`, `defaultLogger`, `eventListener`, `defaultColorPalette`, `embabelJacksonObjectMapper`, `ranker`, `agentProcessRepository`, `contextRepository`, `toolGroupResolver`, `toolsStats`, `actionScheduler`, `modelProvider`, `autoLlmSelectionCriteriaResolver`, and `outputChannel`.
- `ToolGroupsConfiguration` contributes the tool group beans that the platform can expose, including `mathToolGroup`, `mcpWebToolsGroup`, `mapsToolsGroup`, `browserAutomationWebToolsGroup`, and `githubToolsGroup`.
- `EmbeddingTrackingConfiguration` contributes `embeddingTrackingBeanPostProcessor`, which lets the platform observe embedding-related work when the relevant listeners are present.
- Spring then instantiates the `@Agent` class like any other bean, wires its constructor arguments, and registers it in the application context.
- `DelegatingAgentScanningBeanPostProcessor` buffers beans during startup, then, after the context refresh, looks up `AgentScanningPostProcessorDelegate` from the application context and hands the queued beans to it.
- `AgentScanningPostProcessorDelegate` performs the actual scan of candidate beans and uses `AgentMetadataReader` to inspect `@Agent`, `@Action`, `@Condition`, and `@AchievesGoal`.
- `AgentMetadataReader` reads the Spring-managed bean class and turns those annotations into deployable agent metadata.
- `AgentDeployer` takes the discovered agents, the `AgentPlatform`, and the scanning properties, then deploys the agents into the running platform.
- Spring owns the bean instance; Embabel uses the metadata and deployment path to make that bean available as an agent.

7.4. Conclusion

When startup goes wrong, the first places to inspect are the starter dependency in `pom.xml`, the auto-configuration imports, the property loader logs, and the deployment logs.

Part 2: Agentic AI APIs

This part covers object creation, nullable creation, prompt contributions, thinking, streaming, and tool calls.

- Prompt Contributors
- Thinking
- Streaming
- Tool Call

Chapter 8. Object Creation

8.1. Introduction

Object creation is the part of the API that turns a prompt into a strongly typed Java object. This chapter uses the `Ai` gateway and the `PromptRunner.creating(…)` fluent API to create a structured travel summary from plain text. The next chapter will cover the nullable variant, `createObjectIfPossible(…)`.

8.2. Key Concepts

`Ai` is the entry point for LLM work in application code. It gives you a `PromptRunner` with a chosen model configuration.

`LlmOptions` controls which model is selected and how the call behaves. It can set model selection, temperature, max tokens, top-p, top-k, thinking, and timeout.

`PromptRunner.creating(…)` opens the fluent object-creation builder. The builder can add examples and validation before the prompt is executed. In this chapter, the target object has five fields and the chapter uses a travel system prompt to bias the request toward the right domain.

`createObjectIfPossible(…)` is the nullable variant. It belongs to the next chapter because this chapter focuses on the successful creation path first.

8.3. How It Works

8.3.1. Bootstrap

The test uses the cookbook Spring Boot application and injects `Ai` directly. No agent class is needed for this chapter because the point is the LLM object-creation API itself.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles("cookbook-test")
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class ObjectCreationTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private Ai ai;
```

8.3.2. Main Flow

`Ai.withLlm(…)` produces a `PromptRunner` with the chosen LLM options. The fluent builder starts with `withSystemPrompt(…)` and `creating(…)`, then optionally adds an example object and optionally enables validation before the prompt is executed. Those steps are helpers, not requirements: each one can be used independently depending on how much guidance the model needs.

```
    @Test
    void creatingBuildsATravelSummary() {
        logger.info("Running object creation test");

        PromptRunner promptRunner = ai.withLlm(LlmOptions.withDefaultLlm
().withTemperature(0.0)); ①
        var summary = promptRunner
                .withSystemPrompt("You are a travel assistant.") ②
                .creating(TripSummary.class) ③
                .withExample(
                        "Weekend in Rome",
                        new TripSummary(
                                "Rome",
                                "plane",
                                "Visit the Colosseum and finish with dinner in
Trastevere.",
                                "Colosseum",
                                "150 EUR"
                        )
                ) ④
                .withValidation(true) ⑤
                .fromPrompt("""
                        Create a short travel summary for a weekend trip to Paris.
                        Return the destination, the transport, a short itinerary
description, one highlight, and a budget.
                        """); ⑥

        logger.info("Created summary: {}", summary);

        assertNotNull(summary);
        assertNotNull(summary.destination());
        assertNotNull(summary.transport());
        assertNotNull(summary.itineraryDescription());
        assertNotNull(summary.highlight());
        assertNotNull(summary.budget());
    }
```

① `ai.withLlm(LlmOptions.withDefaultLlm().withTemperature(0.0))` picks the default LLM route and sets temperature for the prompt runner used by the call.

② `withSystemPrompt(…)` adds a travel-assistant system instruction before the object-creation request starts.

③ `creating(TripSummary.class)` tells Embabel the exact type to create.

④ `withExample(…)` is an optional example object that shows the LLM the expected shape. In this test, the example is a different trip from the requested Paris summary and it includes a concrete itinerary description, so it demonstrates the structure without copying the answer.

⑤ `withValidation(true)` is optional validation; it tells Embabel to validate the generated object before returning it, but the chapter could skip validation if it only wanted raw object creation.

⑥ `fromPrompt(…)` is the call that actually triggers the LLM request and creates the object.

8.3.3. Test Wiring

The test asserts that the returned object is present and that all five fields were populated. That is enough to prove that the object-creation path produced a structured result rather than plain text, while the system prompt and example guide the model toward the travel domain and the expected shape.

8.4. Conclusion

`Ai` and `PromptRunner.creating(…)` are the right tools when you want typed output from a prompt. The next chapter will show the nullable creation path when the model may not have enough information to build the object.

Chapter 9. Create Object If Possible

9.1. Introduction

`createObjectIfPossible(…)` is the nullable form of object creation. It is the right choice when a prompt may or may not contain enough structured facts to build the target object. This chapter uses the travel domain and focuses on graceful failure when the input is incomplete.

9.2. Key Concepts

`PromptRunner` is the API entry point. It exposes `createObjectIfPossible(…)` so code can ask for a typed object and get `null` when the prompt is not sufficient.

`ItineraryRequest` is the target type in this chapter. It gives the chapter a concrete travel object to extract from plain text.

The nullable return type is the point of the chapter. The positive case returns a populated request; the negative case returns `null`. The important detail is that the API does not throw on missing information; it logs a warning and returns `null`.

9.3. How It Works

9.3.1. Test Shape

This chapter is test-driven, like the other cookbook chapters, but it does not need an agent class. The test uses the LLM API directly and shows both a successful extraction and an insufficient input case.

```
    @Test
    void createObjectIfPossibleReturnsItineraryRequestWhenPromptIsSufficient() {
        logger.info("Running createObjectIfPossible positive test");

        var request = ai.withDefaultLlm() ①
                .createObjectIfPossible("""
                        Plan a three-day itinerary from London to Paris for Friday to
Sunday.
                        Include Eurostar travel, the Eiffel Tower, the Louvre, and a
Seine river walk.
                        """, ItineraryRequest.class); ②

        logger.info("Positive request: {}", request);

        assertNotNull(request);
    }

    @Test
    void createObjectIfPossibleReturnsNullWhenPromptIsInsufficient() {
        logger.info("Running createObjectIfPossible negative test");
```

```
        var request = ai.withDefaultLlm() ③
                .createObjectIfPossible("""
                        Plan a trip.
                        """, ItineraryRequest.class); ④

        logger.info("Negative request: {}", request);

        assertNull(request);
    }
```

9.3.2. Step by Step

① `createObjectIfPossible(…)` is the nullable object-creation API used in this chapter; it asks the LLM to build a typed result and returns `null` when the prompt is insufficient.

② The positive prompt includes enough itinerary detail to build a request: travel dates, destination, and a few concrete activities.

③ `createObjectIfPossible(…)` is the same nullable API in the negative case.

④ The negative prompt is intentionally incomplete, so the API has nothing reliable to extract.

9.3.3. Failure Proof

The negative case is visible in the runtime log:

```
17:35:16.286 [main] WARN  OperationContextDelegate - Failed to create object of type
com.embabel.cookbook.travel.domain.ItineraryRequest with messages
[UserMessage(from='User', content='Plan a trip.
')]: Insufficient details to plan a trip
```

That warning is the proof point for the chapter: the API returns `null` when the prompt is insufficient, and it does so without raising an exception.

9.4. Conclusion

`createObjectIfPossible(…)` is the right API when the application wants a typed result but must also handle missing information gracefully.

Chapter 10. Prompt Contributors

10.1. Introduction

Prompt contributors let you build a prompt from structured message objects instead of a single concatenated string. This chapter uses `SystemMessage` and `UserMessage` to create a travel plan from a message list.

10.2. Key Concepts

`SystemMessage` carries system-level instructions for the LLM. `UserMessage` carries the request from the user. Together they form a message list that the runner can send to the model.

`PromptRunner.creating(⋯)` opens the fluent object-creation builder. `fromMessages(⋯)` executes the request using a list of message objects instead of a single string.

10.3. How It Works

10.3.1. Bootstrap

The test uses the cookbook Spring Boot application and injects `Ai` directly. No agent class is needed because the chapter is about composing prompt messages before object creation.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles("cookbook-test")
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class PromptContributorsTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private Ai ai;
```

10.3.2. Main Flow

The test builds a `List<Message>` from a `SystemMessage` and two `UserMessage` objects, then hands that message list to `PromptRunner.creating(⋯).fromMessages(⋯)`. That keeps the prompt contributions visible as separate pieces instead of hiding them in one large string.

```
    @Test
    void creatingFromMessagesBuildsTravelPlan() {
```

```
        logger.info("Running prompt contributors test");

        PromptRunner runner = ai.withDefaultLlm(); ①
        List<Message> messages = List.of( ②
                new SystemMessage("You are a travel assistant."),
                new UserMessage("Create a structured travel plan for a three-day trip
from London to Paris."),
                new UserMessage("Include Eurostar travel, the Eiffel Tower, the
Louvre, and a Seine walk.")
        );

        var plan = runner
                .creating(TravelPlan.class) ③
                .fromMessages(messages); ④

        logger.info("Created travel plan: {}", plan);

        assertNotNull(plan);
        assertTrue( plan.destination().toLowerCase().contains("paris"),
                "Paris is missing from the destination " + plan.destination());
        assertNotNull(plan.transport());
        assertNotNull(plan.itineraryDescription());
        assertNotNull(plan.highlight());
    }
```

① `ai.withDefaultLlm()` selects the cookbook default model from the test properties.

② The message list is built from `SystemMessage` and `UserMessage` objects so the prompt stays structured.

③ `creating(TravelPlan.class)` tells Embabel which type to create.

④ `fromMessages(messages)` executes the object creation from the assembled message list.

10.3.3. Test Wiring

The test checks that the returned `TravelPlan` is present and that its fields were populated from the message list. That proves the prompt was composed from message contributors rather than a single raw prompt string.

10.4. Conclusion

Use message lists when you want prompt composition to be explicit. `SystemMessage` and `UserMessage` make the request easier to inspect, reuse, and debug.

Chapter 11. Thinking

11.1. Introduction

Thinking is the API for capturing the model's reasoning content while still getting a typed result. This chapter stays on the direct object-creation path: it shows a successful travel plan and a nullable failure case, both through `ThinkingResponse`. The cookbook test uses `ai.withDefaultLlm()` and then checks whether that runner supports thinking before turning the call into a thinking-aware request.

11.2. Key Concepts

`PromptRunner.thinking()` enables reasoning extraction for the current LLM call. It only works when the selected model advertises thinking support.

`ThinkingResponse<T>` is the framework wrapper that carries both the typed result and the extracted reasoning content. The `result` is the created object or `null`; the `thinkingBlocks` list contains the reasoning text the model produced. The chapter uses the shared `TravelPlan` record from the cookbook domain so the result type is reusable across chapters.

`createObject(…)` returns a `ThinkingResponse<T>` when the prompt contains enough information. The chapter uses it for the positive case.

`createObjectIfPossible(…)` returns a `ThinkingResponse<T>` (where the result `T` may be `null`) when the prompt may be incomplete. The chapter uses it for the nullable case.

11.3. How It Works

11.3.1. Bootstrap

The test uses the cookbook Spring Boot application and injects `Ai` directly. No agent class is needed because the chapter is about the thinking-aware object-creation API.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles("cookbook-test")
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
    listeners = DependencyInjectionTestExecutionListener.class,
    mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class ThinkingTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
```

```
    private Ai ai;
```

11.3.2. Positive Path

The positive test selects the cookbook default LLM, adds a system prompt that asks for a separate `<decision_reasoning>` block, checks that the runner supports thinking, and then creates a typed `TravelPlan`. The returned `ThinkingResponse` contains the created object, and it also surfaces the extracted reasoning content when the model emits it. In this run, the extracted block is tagged `decision_reasoning`, and the chapter log shows both the block content and the tag metadata. `getThinkingContent()` prints the reasoning text, while the underlying `ThinkingBlock` also carries the tag name.

```
    @Test
    void createObjectWithThinkingResult() {
        logger.info("Running thinking positive test");

        PromptRunner runner = ai.withDefaultLlm(); ①
        assertTrue(runner.supportsThinking(), "Expected the prompt runner to support
thinking"); ②

        ThinkingResponse<TravelPlan> response = runner
            .withSystemPrompt( ③
                """
                     You are travel assistant.
                     You MUST:
                   1. Provide reasoning inside
<decision_reasoning>...</decision_reasoning>
                   2. Keep reasoning concise (3-5 bullet points)
                   3. Explain why the itinerary is optimal.

                   <decision_reasoning>
                   Explain:
                   - time constraint
                   - risk of being late
                   - trade-offs
                   </decision_reasoning>
                   """)
            .thinking() ④
            .createObject("""
                Create a short travel plan for a three-day trip from London to Paris.
                Include major landmarks.
                Make the most optimal and balanced itinerary.
                """, TravelPlan.class); ⑤

        logger.info("Positive thinking response: {}", response);
        logger.info("Positive thinking content: {}", response.getThinkingContent());

        assertNotNull(response);
        assertTrue(response.hasResult());
```

```
        assertNotNull(response.getResult());
        assertTrue(response.hasThinking()); ⑤
    }
```

① `ai.withDefaultLlm()` selects the cookbook default model, which is configured in `application-cookbook-test.properties`.

② `supportsThinking()` verifies that the selected model can actually produce thinking blocks.

③ `withSystemPrompt(…)` injects the instruction that reasoning must appear inside `<decision_reasoning>`.

④ `runner.thinking()` switches the prompt runner into thinking-aware object creation.

⑤ `createObject(…)` asks for a typed travel plan and returns a `ThinkingResponse`. `hasResult()` and `hasThinking()` prove that the call produced both a structured result and extracted reasoning content. The logged reasoning content comes from the `decision_reasoning` block in the response.

11.3.3. Nullable Path

The nullable test uses the same thinking-aware runner and the same system prompt, but passes an intentionally incomplete user prompt. This is different from the plain nullable API, which would return `null` directly. Here, the thinking-aware call returns a `ThinkingResponse` whose `result` is `null` and whose exception carries the failure information for the impossible request. In this chapter, the useful text is the exception message explaining that a half-day New York to Sydney trip is impossible or infeasible.

```
    @Test
    void createObjectIfPossibleWithThinkingResult() {
        logger.info("Running thinking nullable test");

        PromptRunner runner = ai.withDefaultLlm(); ①
        assertTrue(runner.supportsThinking(), "Expected the prompt runner to support
thinking"); ②

        ThinkingResponse<TravelPlan> response = runner
            .withSystemPrompt( ③
                """
                     You are travel assistant.
                     You MUST:
                   1. Provide reasoning inside
<decision_reasoning>...</decision_reasoning>
                   2. Keep reasoning concise (3-5 bullet points)
                   3. Explain why the itinerary is optimal.

                   <decision_reasoning>
                   Explain:
                   - time constraint
                   - risk of being late
                   - trade-offs
                   </decision_reasoning>
                   """)
```

```
            .thinking() ④
            .createObjectIfPossible("""
                Create a short travel plan for a half-day trip from New York to
Sydney.
                Include major landmarks.
                Make the most optimal and balanced itinerary.
                """, TravelPlan.class); ⑤

        logger.info("Nullable thinking response: {}", response);
        logger.info("Nullable thinking content: {}", response.getThinkingContent());
        logger.info("Nullable thinking exception: {}", response.getException() == null
? null : response.getException().getMessage());

        assertNotNull(response);
        assertNull(response.getResult());
        assertNotNull(response.getException());
        assertInstanceOf(ThinkingException.class, response.getException());
        String exceptionMessage = response.getException().getMessage().toLowerCase();
        assertTrue(exceptionMessage.contains("impossible"));
        assertTrue(exceptionMessage.contains("half-day"));
        assertTrue(exceptionMessage.contains("new york"));
        assertTrue(exceptionMessage.contains("sydney"));
    }
```

① `ai.withDefaultLlm()` matches the positive test and keeps the model selection centralized in the test properties.

② `supportsThinking()` verifies that the selected model can actually produce thinking blocks.

③ `withSystemPrompt(…)` injects the same reasoning instruction as the positive case.

④ `runner.thinking()` switches the prompt runner into thinking-aware object creation.

⑤ `createObjectIfPossible(…)` is the nullable thinking-aware API. The prompt is intentionally incomplete, so the model has too little travel detail to build a typed plan. `result` is expected to be `null`, but the response object itself still exists. The failure is surfaced through `response.getException()`, whose message explains why the object could not be created. That is the key difference from the plain nullable API, which would return `null` directly.

11.4. Conclusion

`ThinkingResponse` is useful when you want typed output and the reasoning that produced it. Use `createObject(…)` when the prompt should be enough, and `createObjectIfPossible(…)` when the prompt may be incomplete. If the nullable thinking-aware path fails, the wrapper still exists and the failure is carried in `response.getException()`.

Chapter 12. Streaming

12.1. Introduction

Streaming allows you to consume typed results as they arrive instead of waiting for a single final object. This chapter streams three travel itineraries, records them as they arrive, and checks that the stream completes.

12.2. Key Concepts

`StreamingPromptRunnerBuilder` adapts a `PromptRunner` to the streaming API. `supportsStreaming()` tells you whether the selected model can stream. `createObjectStream(…)` parses a newline-delimited JSON response into typed objects. `doOnNext(…)` lets the test handle each streamed itinerary as soon as it appears. `Itinerary` is the shared travel-domain record used for the streamed results.

12.3. How It Works

12.3.1. Bootstrap

The test uses the cookbook Spring Boot application and injects `Ai` directly. No agent class is needed because the chapter is about streaming typed objects from a prompt.

```
@SpringBootTest(classes = CookbookTestApplication.class)
@ActiveProfiles("cookbook-test")
// Replace Spring's default test listeners so this test keeps dependency injection
// without triggering the broader reset/mock listeners that are noisy here.
@TestExecutionListeners(
        listeners = DependencyInjectionTestExecutionListener.class,
        mergeMode = TestExecutionListeners.MergeMode.REPLACE_DEFAULTS
)
class StreamingTest {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private Ai ai;
```

12.3.2. Main Flow

The test asks for exactly three itinerary objects, one per JSON line, and handles each one in `doOnNext(…)`. That keeps the request explicit and lets the test assert on the streamed objects and completion callback.

```
    @Test
    void streamingTopThreeItineraries() {
```

```
        logger.info("Running streaming test");

        PromptRunner runner = ai.withDefaultLlm(); ①
        assertTrue(runner.supportsStreaming(), "Expected the prompt runner to support
streaming"); ②

        List<Itinerary> itineraries = new CopyOnWriteArrayList<>(); ③
        AtomicReference<Throwable> errorOccurred = new AtomicReference<>(); ④
        AtomicBoolean completionCalled = new AtomicBoolean(false); ⑤

        new StreamingPromptRunnerBuilder(runner) ⑥
                .streaming()
                .withPrompt("""
                        Return exactly three itinerary options for a three-day trip
from London to Paris.
                        Emit each itinerary as a separate JSON object with a single
text field.
                        Keep the options distinct and concise.
                        """) ⑦
                .createObjectStream(Itinerary.class) ⑧
                .timeout(Duration.ofSeconds(240))
                .doOnNext(itinerary -> {
                    itineraries.add(itinerary);
                    logger.info("Streamed itinerary: {}", itinerary.text());
                })
                .doOnError(error -> errorOccurred.set(error))
                .doOnComplete(() -> completionCalled.set(true))
                .blockLast(Duration.ofSeconds(240)); ⑨

        logger.info("Streamed itineraries: {}", itineraries);

        assertNull(errorOccurred.get());
        assertNotNull(itineraries);
        assertTrue(completionCalled.get(), "Expected the streaming flux to complete");
        assertEquals(3, itineraries.size());
    }
```

① `ai.withDefaultLlm()` selects the cookbook default model from the test properties.

② `supportsStreaming()` confirms the selected model can stream before the builder is used.

③ `itineraries` is a thread-safe list that records objects as they stream in.

④ `errorOccurred` captures any streaming failure.

⑤ `completionCalled` records when the stream finishes normally.

⑥ `new StreamingPromptRunnerBuilder(runner).streaming()` opens the streaming API.

⑦ The prompt asks for exactly three itinerary objects as separate JSON lines.

⑧ `createObjectStream(Itinerary.class)` parses the response into `Itinerary` objects.

⑨ `blockLast(…)` keeps the subscription alive until the stream completes.

`doOnNext(…)` fires for each object the moment it is parsed from the stream. Streaming uses Project Reactor — `Flux<T>` pushes results to subscribers without blocking the calling thread.

12.3.3. Test Wiring

The test checks that three `Itinerary` objects arrive, that the stream completes, and that each itinerary contains text. That proves the stream is producing typed objects incrementally rather than one final blob of text. The staggered log timestamps show the items arriving on separate callback invocations over time.

```
10:25:45.694 [loomBoundedElastic-63] INFO  StreamingTest - Streamed itinerary: Option
1: Day 1: Eurostar to Paris, check in, evening stroll along the Seine and dinner in
Saint-Germain. Day 2: Louvre, Tuileries, and a sunset view from the Eiffel Tower. Day
3: Montmartre, café brunch, last-minute shopping, return to London.
10:25:46.268 [loomBoundedElastic-137] INFO  StreamingTest - Streamed itinerary: Option
2: Day 1: Arrive in Paris by Eurostar, explore Île de la Cité and Notre-Dame area,
then dinner in Le Marais. Day 2: Musée d'Orsay, Seine cruise, and evening at the
Champs-Élysées. Day 3: Latin Quarter walk, Luxembourg Gardens, relaxed lunch, back to
London.
10:25:46.983 [loomBoundedElastic-208] INFO  StreamingTest - Streamed itinerary: Option
3: Day 1: Travel from London to Paris, settle in, and spend the evening in Canal
Saint-Martin. Day 2: Visit the Arc de Triomphe, Eiffel Tower, and a food tour. Day 3:
Sacré-Cœur, boutique browsing in Montmartre, then return to London.
```

12.4. Conclusion

Use streaming when you want typed results as a `Flux` instead of a single completed object. For collections, ask the model for one JSON object per line and handle the emitted items as they arrive.

Chapter 13. Tool Call

13.1. Introduction

Tools let the LLM gather external information before producing a structured result. Instead of relying solely on training data, the model calls your methods, receives their output, and uses it to inform the final object it returns. This chapter shows how to expose tools to a `PromptRunner` and produce a `TravelPlan` backed by live tool results.

13.2. Key Concepts

`@LlmTool` marks a method as callable by the LLM during inference. `withToolObject(…)` registers an instance whose `@LlmTool` methods become available to the model. `withToolCallInspectors(…)` attaches callbacks that log each tool invocation and its result. `creating(TravelPlan.class).fromPrompt(…)` runs the prompt, allowing the model to call tools before returning the typed result.

13.3. How It Works

13.3.1. Tooling

The `TravelTooling` class exposes two methods to the LLM. Each method receives a destination string from the model and returns a hard-coded but realistic response.

```
    static class TravelTooling {

        @LlmTool(description = "Get current weather for a destination.")
        public String getWeather(String destination) {
            return "Sunny, 24°C, low humidity — ideal for outdoor sightseeing in " +
destination + ".";
        }

        @LlmTool(description = "Get top attractions for a destination.")
        public String getTopAttractions(String destination) {
            return "Top attractions in " + destination + ": Eiffel Tower, Louvre
Museum, Montmartre, Seine River cruise.";
        }
    }
```

13.3.2. Main Flow

The test wires the tooling into a `PromptRunner` and asks for a `TravelPlan`. The model calls the tools, receives their responses, and uses them to populate the result.

```
    @Test
```

```
    void planTripWithTools() {
        PromptRunner runner = ai.withDefaultLlm()
                .withToolObject(new TravelTooling()) ①
                .withToolCallInspectors(new ToolCallLoggingInspector(LogLevel.INFO,
logger)); ②

        TravelPlan plan = runner
                .creating(TravelPlan.class) ③
                .fromPrompt("""
                        Plan a 3-day trip to Paris for a traveler from London.
                        Use tools to check the weather and top attractions.
                        """); ④

        logger.info("Created travel plan: {}", plan);

        assertNotNull(plan.destination());
        assertNotNull(plan.itineraryDescription());
        assertNotNull(plan.highlight());
    }
```

① `withToolObject(new TravelTooling())` makes `getWeather` and `getTopAttractions` available to the model.

② `withToolCallInspectors(…)` logs each tool call and its result at INFO level.

③ `creating(TravelPlan.class)` tells the runner to produce a typed `TravelPlan`.

④ The prompt instructs the model to use tools before recommending a plan.

13.4. Conclusion

Annotate methods with `@LlmTool`, register the object with `withToolObject(…)`, and the model will call them as needed before returning the typed result. Tool call inspectors are optional but useful for observing which tools were called and what they returned.

Chapter 14. Further Reading

The recipes in this cookbook cover common patterns, but Embabel Agent has much more to offer.

- Embabel Agent Reference Guide: full documentation covering all APIs, configuration, and advanced features.
- Embabel Hub: community resources, examples, and published agents.