`pydantic_ai.models.test`

Utility model for quickly testing apps built with PydanticAI.

TestModel `dataclass`

Bases: `Model`

A model specifically for testing purposes.

This will (by default) call all tools in the agent, then return a tool response if possible, otherwise a plain response.

How useful this model is will vary significantly.

Apart from `__init__` derived by the `dataclass` decorator, all methods are private or match those of the base class.

Source code in `pydantic_ai_slim/pydantic_ai/models/test.py`

```python
@dataclass
class TestModel(Model):
    """A model specifically for testing purposes.

    This will (by default) call all tools in the agent, then return a tool response if possible,
    otherwise a plain response.

    How useful this model is will vary significantly.

    Apart from `__init__` derived by the `dataclass` decorator, all methods are private or match those
    of the base class.
    """

    # NOTE: Avoid test discovery by pytest.
    __test__ = False

    call_tools: list[str] | Literal['all'] = 'all'
    """List of tools to call. If `'all'`, all tools will be called."""
    custom_result_text: str | None = None
    """If set, this text is return as the final result."""
    custom_result_args: Any | None = None
    """If set, these args will be passed to the result tool."""
    seed: int = 0
    """Seed for generating random data."""
    agent_model_function_tools: list[ToolDefinition] | None = field(default=None, init=False)
    """Definition of function tools passed to the model.

    This is set when the model is called, so will reflect the function tools from the last step of the last run.
    """
    agent_model_allow_text_result: bool | None = field(default=None, init=False)
    """Whether plain text responses from the model are allowed.

    This is set when the model is called, so will reflect the value from the last step of the last run.
    """
    agent_model_result_tools: list[ToolDefinition] | None = field(default=None, init=False)
    """Definition of result tools passed to the model.

    This is set when the model is called, so will reflect the result tools from the last step of the last run.
    """

    async def agent_model(
        self,
        *,
        function_tools: list[ToolDefinition],
        allow_text_result: bool,
        result_tools: list[ToolDefinition],
    ) -> AgentModel:
        self.agent_model_function_tools = function_tools
        self.agent_model_allow_text_result = allow_text_result
        self.agent_model_result_tools = result_tools

        if self.call_tools == 'all':
            tool_calls = [(r.name, r) for r in function_tools]
        else:
            function_tools_lookup = {t.name: t for t in function_tools}
            tools_to_call = (function_tools_lookup[name] for name in self.call_tools)
            tool_calls = [(r.name, r) for r in tools_to_call]

        if self.custom_result_text is not None:
            assert allow_text_result, 'Plain response not allowed, but `custom_result_text` is set.'
            assert self.custom_result_args is None, 'Cannot set both `custom_result_text` and `custom_result_args`.'
            result: _utils.Either[str | None, Any | None] = _utils.Either(left=self.custom_result_text)
        elif self.custom_result_args is not None:
            assert result_tools is not None, 'No result tools provided, but `custom_result_args` is set.'
            result_tool = result_tools[0]

            if k := result_tool.outer_typed_dict_key:
                result = _utils.Either(right={k: self.custom_result_args})
            else:
                result = _utils.Either(right=self.custom_result_args)
        elif allow_text_result:
            result = _utils.Either(left=None)
        elif result_tools:
            result = _utils.Either(right=None)
        else:
            result = _utils.Either(left=None)

        return TestAgentModel(tool_calls, result, result_tools, self.seed)

    def name(self) -> str:
        return 'test-model'
```

call_tools `class-attribute` `instance-attribute`

```python
call_tools: list[str] | Literal['all'] = 'all'
```

List of tools to call. If `'all'`, all tools will be called.

custom_result_text `class-attribute` `instance-attribute`

```python
custom_result_text: str | None = None
```

If set, this text is return as the final result.

custom_result_args `class-attribute` `instance-attribute`

```
custom_result_args: Any | None = None
```

If set, these args will be passed to the result tool.

seed `class-attribute` `instance-attribute`

```
seed: int = 0
```

Seed for generating random data.

agent_model_function_tools `class-attribute` `instance-attribute`

```
agent_model_function_tools: list[ToolDefinition] | None = (
    field(default=None, init=False)
)
```

Definition of function tools passed to the model.

This is set when the model is called, so will reflect the function tools from the last step of the last run.

agent_model_allow_text_result `class-attribute` `instance-attribute`

```
agent_model_allow_text_result: bool | None = field(
    default=None, init=False
)
```

Whether plain text responses from the model are allowed.

This is set when the model is called, so will reflect the value from the last step of the last run.

agent_model_result_tools `class-attribute` `instance-attribute`

```
agent_model_result_tools: list[ToolDefinition] | None = (
    field(default=None, init=False)
)
```

Definition of result tools passed to the model.

This is set when the model is called, so will reflect the result tools from the last step of the last run.

TestAgentModel `dataclass`

Bases: `AgentModel`

Implementation of `AgentModel` for testing purposes.

Source code in `pydantic_ai_slim/pydantic_ai/models/test.py`

```python
@dataclass
class TestAgentModel(AgentModel):
    """Implementation of `AgentModel` for testing purposes."""

    # NOTE: Avoid test discovery by pytest.
    __test__ = False

    tool_calls: list[tuple[str, ToolDefinition]]
    # left means the text is plain text; right means it's a function call
    result: _utils.Either[str | None, Any | None]
    result_tools: list[ToolDefinition]
    seed: int

    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, Cost]:
        return self._request(messages), Cost()

    @asynccontextmanager
    async def request_stream(self, messages: list[Message]) -> AsyncIterator[EitherStreamedResponse]:
        msg = self._request(messages)
        cost = Cost()
        if isinstance(msg, ModelTextResponse):
            yield TestStreamTextResponse(msg.content, cost)
        else:
            yield TestStreamStructuredResponse(msg, cost)

    def gen_tool_args(self, tool_def: ToolDefinition) -> Any:
        return _JsonSchemaTestData(tool_def.parameters_json_schema, self.seed).generate()

    def _request(self, messages: list[Message]) -> ModelAnyResponse:
        # if there are tools, the first thing we want to do is call all of them
        if self.tool_calls and not any(m.role == 'model-structured-response' for m in messages):
            calls = [ToolCall.from_dict(name, self.gen_tool_args(args)) for name, args in self.tool_calls]
            return ModelStructuredResponse(calls=calls)

        # get messages since the last model response
        new_messages = _get_new_messages(messages)

        # check if there are any retry prompts, if so retry them
        new_retry_names = {m.tool_name for m in new_messages if isinstance(m, RetryPrompt)}
        if new_retry_names:
            calls = [
                ToolCall.from_dict(name, self.gen_tool_args(args))
                for name, args in self.tool_calls
                if name in new_retry_names
            ]
            return ModelStructuredResponse(calls=calls)

        if response_text := self.result.left:
            if response_text.value is None:
                # build up details of tool responses
                output: dict[str, Any] = {}
                for message in messages:
                    if isinstance(message, ToolReturn):
                        output[message.tool_name] = message.content
                if output:
                    return ModelTextResponse(content=pydantic_core.to_json(output).decode())
                else:
                    return ModelTextResponse(content='success (no tool calls)')
            else:
                return ModelTextResponse(content=response_text.value)
        else:
            assert self.result_tools, 'No result tools provided'
            custom_result_args = self.result.right
            result_tool = self.result_tools[self.seed % len(self.result_tools)]
            if custom_result_args is not None:
                return ModelStructuredResponse(calls=[ToolCall.from_dict(result_tool.name, custom_result_args)])
            else:
                response_args = self.gen_tool_args(result_tool)
                return ModelStructuredResponse(calls=[ToolCall.from_dict(result_tool.name, response_args)])
```

TestStreamTextResponse `dataclass`

Bases: `StreamTextResponse`

A text response that streams test data.

Source code in `pydantic_ai_slim/pydantic_ai/models/test.py`

```python
@dataclass
class TestStreamTextResponse(StreamTextResponse):
    """A text response that streams test data."""

    _text: str
    _cost: Cost
    _iter: Iterator[str] = field(init=False)
    _timestamp: datetime = field(default_factory=_utils.now_utc)
    _buffer: list[str] = field(default_factory=list, init=False)

    def __post_init__(self):
        *words, last_word = self._text.split(' ')
        words = [f'{word} ' for word in words]
        words.append(last_word)
        if len(words) == 1 and len(self._text) > 2:
            mid = len(self._text) // 2
            words = [self._text[:mid], self._text[mid:]]
        self._iter = iter(words)

    async def __anext__(self) -> None:
        self._buffer.append(_utils.sync_anext(self._iter))

    def get(self, *, final: bool = False) -> Iterable[str]:
        yield from self._buffer
        self._buffer.clear()

    def cost(self) -> Cost:
        return self._cost

    def timestamp(self) -> datetime:
        return self._timestamp
```

TestStreamStructuredResponse `dataclass`

Bases: `StreamStructuredResponse`

A structured response that streams test data.

Source code in `pydantic_ai_slim/pydantic_ai/models/test.py`

```python
@dataclass
class TestStreamStructuredResponse(StreamStructuredResponse):
    """A structured response that streams test data."""

    _structured_response: ModelStructuredResponse
    _cost: Cost
    _iter: Iterator[None] = field(default_factory=lambda: iter([None]))
    _timestamp: datetime = field(default_factory=_utils.now_utc, init=False)

    async def __anext__(self) -> None:
        return _utils.sync_anext(self._iter)

    def get(self, *, final: bool = False) -> ModelStructuredResponse:
        return self._structured_response

    def cost(self) -> Cost:
        return self._cost

    def timestamp(self) -> datetime:
        return self._timestamp
```