`pydantic_ai.messages`

Message `module-attribute`

```
Message = Union[
    SystemPrompt,
    UserPrompt,
    ToolReturn,
    RetryPrompt,
    ModelTextResponse,
    ModelStructuredResponse,
]
```

Any message send to or returned by a model.

SystemPrompt `dataclass`

A system prompt, generally written by the application developer.

This gives the model context and guidance on how to respond.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class SystemPrompt:
    """A system prompt, generally written by the application developer.

    This gives the model context and guidance on how to respond.
    """

    content: str
    """The content of the prompt."""
    role: Literal['system'] = 'system'
    """Message type identifier, this type is available on all message as a discriminator."""
```

content `instance-attribute`

```
content: str
```

The content of the prompt.

role `class-attribute` `instance-attribute`

```
role: Literal['system'] = 'system'
```

Message type identifier, this type is available on all message as a discriminator.

UserPrompt `dataclass`

A user prompt, generally written by the end user.

Content comes from the `user_prompt` parameter of `Agent.run`, `Agent.run_sync`, and `Agent.run_stream`.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class UserPrompt:
    """A user prompt, generally written by the end user.

    Content comes from the `user_prompt` parameter of [`Agent.run`][pydantic_ai.Agent.run],
    [`Agent.run_sync`][pydantic_ai.Agent.run_sync], and [`Agent.run_stream`][pydantic_ai.Agent.run_stream].
    """

    content: str
    """The content of the prompt."""
    timestamp: datetime = field(default_factory=_now_utc)
    """The timestamp of the prompt."""
    role: Literal['user'] = 'user'
    """Message type identifier, this type is available on all message as a discriminator."""
```

content `instance-attribute`

```
content: str
```

The content of the prompt.

timestamp `class-attribute` `instance-attribute`

```
timestamp: datetime = field(default_factory=now_utc)
```

The timestamp of the prompt.

role `class-attribute` `instance-attribute`

```
role: Literal['user'] = 'user'
```

Message type identifier, this type is available on all message as a discriminator.

ToolReturn `dataclass`

A tool return message, this encodes the result of running a tool.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class ToolReturn:
    """A tool return message, this encodes the result of running a tool."""

    tool_name: str
    """The name of the "tool" was called."""
    content: Any
    """The return value."""
    tool_call_id: str | None = None
    """Optional tool call identifier, this is used by some models including OpenAI."""
    timestamp: datetime = field(default_factory=_now_utc)
    """The timestamp, when the tool returned."""
    role: Literal['tool-return'] = 'tool-return'
    """Message type identifier, this type is available on all message as a discriminator."""

    def model_response_str(self) -> str:
        if isinstance(self.content, str):
            return self.content
        else:
            return tool_return_ta.dump_json(self.content).decode()

    def model_response_object(self) -> dict[str, Any]:
        # gemini supports JSON dict return values, but no other JSON types, hence we wrap anything else in a dict
        if isinstance(self.content, dict):
            return tool_return_ta.dump_python(self.content, mode='json')  # pyright: ignore[reportUnknownMemberType]
        else:
            return {'return_value': tool_return_ta.dump_python(self.content, mode='json')}
```

tool_name `instance-attribute`

```python
tool_name: str
```

The name of the "tool" was called.

content `instance-attribute`

```python
content: Any
```

The return value.

tool_call_id `class-attribute` `instance-attribute`

```python
tool_call_id: str | None = None
```

Optional tool call identifier, this is used by some models including OpenAI.

timestamp `class-attribute` `instance-attribute`

```python
timestamp: datetime = field(default_factory=now_utc)
```

The timestamp, when the tool returned.

role `class-attribute` `instance-attribute`

```python
role: Literal['tool-return'] = 'tool-return'
```

Message type identifier, this type is available on all message as a discriminator.

RetryPrompt `dataclass`

A message back to a model asking it to try again.

This can be sent for a number of reasons:

- Pydantic validation of tool arguments failed, here content is derived from a Pydantic `ValidationError`
- a tool raised a `ModelRetry` exception
- no tool was found for the tool name
- the model returned plain text when a structured response was expected
- Pydantic validation of a structured response failed, here content is derived from a Pydantic `ValidationError`
- a result validator raised a `ModelRetry` exception

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class RetryPrompt:
    """A message back to a model asking it to try again.

    This can be sent for a number of reasons:

    * Pydantic validation of tool arguments failed, here content is derived from a Pydantic
      [`ValidationError`][pydantic_core.ValidationError]
    * a tool raised a [`ModelRetry`][pydantic_ai.exceptions.ModelRetry] exception
    * no tool was found for the tool name
    * the model returned plain text when a structured response was expected
    * Pydantic validation of a structured response failed, here content is derived from a Pydantic
      [`ValidationError`][pydantic_core.ValidationError]
    * a result validator raised a [`ModelRetry`][pydantic_ai.exceptions.ModelRetry] exception
    """

    content: list[pydantic_core.ErrorDetails] | str
    """Details of why and how the model should retry.

    If the retry was triggered by a [`ValidationError`][pydantic_core.ValidationError], this will be a list of
    error details.
    """
    tool_name: str | None = None
    """The name of the tool that was called, if any."""
    tool_call_id: str | None = None
    """Optional tool call identifier, this is used by some models including OpenAI."""
    timestamp: datetime = field(default_factory=_now_utc)
    """The timestamp, when the retry was triggered."""
    role: Literal['retry-prompt'] = 'retry-prompt'
    """Message type identifier, this type is available on all message as a discriminator."""

    def model_response(self) -> str:
        if isinstance(self.content, str):
            description = self.content
        else:
            json_errors = ErrorDetailsTa.dump_json(self.content, exclude={'__all__': {'ctx'}}, indent=2)
            description = f'{len(self.content)} validation errors: {json_errors.decode()}'
        return f'{description}\n\nFix the errors and try again.'
```

content `instance-attribute`

```python
content: list[ErrorDetails] | str
```

Details of why and how the model should retry.

If the retry was triggered by a `ValidationError`, this will be a list of error details.

tool_name `class-attribute` `instance-attribute`

```python
tool_name: str | None = None
```

The name of the tool that was called, if any.

tool_call_id `class-attribute` `instance-attribute`

```python
tool_call_id: str | None = None
```

Optional tool call identifier, this is used by some models including OpenAI.

timestamp `class-attribute` `instance-attribute`

```python
timestamp: datetime = field(default_factory=now_utc)
```

The timestamp, when the retry was triggered.

role `class-attribute` `instance-attribute`

```python
role: Literal['retry-prompt'] = 'retry-prompt'
```

Message type identifier, this type is available on all message as a discriminator.

ModelAnyResponse `module-attribute`

```python
ModelAnyResponse = Union[
    ModelTextResponse, ModelStructuredResponse
]
```

Any response from a model.

ModelTextResponse `dataclass`

A plain text response from a model.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class ModelTextResponse:
    """A plain text response from a model."""

    content: str
    """The text content of the response."""
    timestamp: datetime = field(default_factory=_now_utc)
    """The timestamp of the response.

    If the model provides a timestamp in the response (as OpenAI does) that will be used.
    """
    role: Literal['model-text-response'] = 'model-text-response'
    """Message type identifier, this type is available on all message as a discriminator."""
```

content `instance-attribute`

```python
content: str
```

The text content of the response.

timestamp `class-attribute` `instance-attribute`

```
timestamp: datetime = field(default_factory=now_utc)
```

The timestamp of the response.

If the model provides a timestamp in the response (as OpenAI does) that will be used.

role `class-attribute` `instance-attribute`

```
role: Literal["model-text-response"] = "model-text-response"
```

Message type identifier, this type is available on all message as a discriminator.

ModelStructuredResponse `dataclass`

A structured response from a model.

This is used either to call a tool or to return a structured response from an agent run.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```
@dataclass
class ModelStructuredResponse:
    """A structured response from a model.

    This is used either to call a tool or to return a structured response from an agent run.
    """

    calls: list[ToolCall]
    """The tool calls being made."""
    timestamp: datetime = field(default_factory=_now_utc)
    """The timestamp of the response.

    If the model provides a timestamp in the response (as OpenAI does) that will be used.
    """
    role: Literal['model-structured-response'] = 'model-structured-response'
    """Message type identifier, this type is available on all message as a discriminator."""
```

calls `instance-attribute`

```
calls: list[ToolCall]
```

The tool calls being made.

timestamp `class-attribute` `instance-attribute`

```
timestamp: datetime = field(default_factory=now_utc)
```

The timestamp of the response.

If the model provides a timestamp in the response (as OpenAI does) that will be used.

role `class-attribute` `instance-attribute`

```
role: Literal["model-structured-response"] = (
    "model-structured-response"
)
```

Message type identifier, this type is available on all message as a discriminator.

ToolCall `dataclass`

Either a tool call from the agent.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```
@dataclass
class ToolCall:
    """Either a tool call from the agent."""

    tool_name: str
    """The name of the tool to call."""
    args: ArgsJson | ArgsDict
    """The arguments to pass to the tool.

    Either as JSON or a Python dictionary depending on how data was returned.
    """
    tool_call_id: str | None = None
    """Optional tool call identifier, this is used by some models including OpenAI."""

    @classmethod
    def from_json(cls, tool_name: str, args_json: str, tool_call_id: str | None = None) -> ToolCall:
        return cls(tool_name, ArgsJson(args_json), tool_call_id)

    @classmethod
    def from_dict(cls, tool_name: str, args_dict: dict[str, Any]) -> ToolCall:
        return cls(tool_name, ArgsDict(args_dict))

    def has_content(self) -> bool:
        if isinstance(self.args, ArgsDict):
            return any(self.args.args_dict.values())
        else:
            return bool(self.args.args_json)
```

tool_name `instance-attribute`

```
tool_name: str
```

The name of the tool to call.

args `instance-attribute`

```
args: ArgsJson | ArgsDict
```

The arguments to pass to the tool.

Either as JSON or a Python dictionary depending on how data was returned.

tool_call_id `class-attribute` `instance-attribute`

```
tool_call_id: str | None = None
```

Optional tool call identifier, this is used by some models including OpenAI.

ArgsJson `dataclass`

Tool arguments as a JSON string.

Source code in `pydantic_ai_slim/pydantic_ai/messages.py`

```python
@dataclass
class ArgsJson:
    """Tool arguments as a JSON string."""

    args_json: str
    """A JSON string of arguments."""
```

args_json `instance-attribute`

```
args_json: str
```

A JSON string of arguments.

MessagesTypeAdapter `module-attribute`

```
MessagesTypeAdapter = LazyTypeAdapter(
    list[Annotated[Message, Field(discriminator="role")]]
)
```

Pydantic `TypeAdapter` for (de)serializing messages.