`pydantic_ai.models.openai`

Setup

For details on how to set up authentication with this model, see model configuration for OpenAI.

OpenAIModelName `module-attribute`

```
OpenAIModelName = Union[ChatModel, str]
```

Using this more broad type for the model name instead of the ChatModel definition allows this model to be used more easily with other model types (ie, Ollama)

OpenAIModel `dataclass`

Bases: `Model`

A model that uses the OpenAI API.

Internally, this uses the OpenAI Python client to interact with the API.

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

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

```python
@dataclass(init=False)
class OpenAIModel(Model):
    """A model that uses the OpenAI API.

    Internally, this uses the [OpenAI Python client](https://github.com/openai/openai-python) to interact with the API.

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

    model_name: OpenAIModelName
    client: AsyncOpenAI = field(repr=False)

    def __init__(
        self,
        model_name: OpenAIModelName,
        *,
        api_key: str | None = None,
        openai_client: AsyncOpenAI | None = None,
        http_client: AsyncHTTPClient | None = None,
    ):
        """Initialize an OpenAI model.

        Args:
            model_name: The name of the OpenAI model to use. List of model names available
                [here](https://github.com/openai/openai-python/blob/v1.54.3/src/openai/types/chat_model.py#L7)
                (Unfortunately, despite being ask to do so, OpenAI do not provide `.inv` files for their API).
            api_key: The API key to use for authentication, if not provided, the `OPENAI_API_KEY` environment variable
                will be used if available.
            openai_client: An existing
                [`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
                client to use, if provided, `api_key` and `http_client` must be `None`.
            http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
        """
        self.model_name: OpenAIModelName = model_name
        if openai_client is not None:
            assert http_client is None, 'Cannot provide both `openai_client` and `http_client`'
            assert api_key is None, 'Cannot provide both `openai_client` and `api_key`'
            self.client = openai_client
        elif http_client is not None:
            self.client = AsyncOpenAI(api_key=api_key, http_client=http_client)
        else:
            self.client = AsyncOpenAI(api_key=api_key, http_client=cached_async_http_client())

    async def agent_model(
        self,
        *,
        function_tools: list[ToolDefinition],
        allow_text_result: bool,
        result_tools: list[ToolDefinition],
    ) -> AgentModel:
        check_allow_model_requests()
        tools = [self._map_tool_definition(r) for r in function_tools]
        if result_tools:
            tools += [self._map_tool_definition(r) for r in result_tools]
        return OpenAIAgentModel(
            self.client,
            self.model_name,
            allow_text_result,
            tools,
        )

    def name(self) -> str:
        return f'openai:{self.model_name}'

    @staticmethod
    def _map_tool_definition(f: ToolDefinition) -> chat.ChatCompletionToolParam:
        return {
            'type': 'function',
            'function': {
                'name': f.name,
                'description': f.description,
                'parameters': f.parameters_json_schema,
            },
        }
```

`__init__`

```
__init__(
    model_name: OpenAIModelName,
    *,
    api_key: str | None = None,
    openai_client: AsyncOpenAI | None = None,
    http_client: AsyncClient | None = None
)
```

Initialize an OpenAI model.

Parameters:

Name	Type	Description	Default
`model_name`	`OpenAIModelName`	The name of the OpenAI model to use. List of model names available here (Unfortunately, despite being ask to do so, OpenAI do not provide `.inv` files for their API).	*required*
`api_key`	`str | None`	The API key to use for authentication, if not provided, the `OPENAI_API_KEY` environment variable will be used if available.	`None`
`openai_client`	`AsyncOpenAI | None`	An existing `AsyncOpenAI` client to use, if provided, `api_key` and `http_client` must be `None`.	`None`
`http_client`	`AsyncClient | None`	An existing `httpx.AsyncClient` to use for making HTTP requests.	`None`

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

```python
def __init__(
    self,
    model_name: OpenAIModelName,
    *,
    api_key: str | None = None,
    openai_client: AsyncOpenAI | None = None,
    http_client: AsyncHTTPClient | None = None,
):
    """Initialize an OpenAI model.

    Args:
        model_name: The name of the OpenAI model to use. List of model names available
            [here](https://github.com/openai/openai-python/blob/v1.54.3/src/openai/types/chat_model.py#L7)
            (Unfortunately, despite being ask to do so, OpenAI do not provide `.inv` files for their API).
        api_key: The API key to use for authentication, if not provided, the `OPENAI_API_KEY` environment variable
            will be used if available.
        openai_client: An existing
            [`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
            client to use, if provided, `api_key` and `http_client` must be `None`.
        http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
    """
    self.model_name: OpenAIModelName = model_name
    if openai_client is not None:
        assert http_client is None, 'Cannot provide both `openai_client` and `http_client`'
        assert api_key is None, 'Cannot provide both `openai_client` and `api_key`'
        self.client = openai_client
    elif http_client is not None:
        self.client = AsyncOpenAI(api_key=api_key, http_client=http_client)
    else:
        self.client = AsyncOpenAI(api_key=api_key, http_client=cached_async_http_client())
```

OpenAIAgentModel `dataclass`

Bases: `AgentModel`

Implementation of `AgentModel` for OpenAI models.

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

```python
@dataclass
class OpenAIAgentModel(AgentModel):
    """Implementation of `AgentModel` for OpenAI models."""

    client: AsyncOpenAI
    model_name: OpenAIModelName
    allow_text_result: bool
    tools: list[chat.ChatCompletionToolParam]

    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, result.Cost]:
        response = await self._completions_create(messages, False)
        return self._process_response(response), _map_cost(response)

    @asynccontextmanager
    async def request_stream(self, messages: list[Message]) -> AsyncIterator[EitherStreamedResponse]:
        response = await self._completions_create(messages, True)
        async with response:
            yield await self._process_streamed_response(response)

    @overload
    async def _completions_create(
        self, messages: list[Message], stream: Literal[True]
    ) -> AsyncStream[ChatCompletionChunk]:
        pass

    @overload
    async def _completions_create(self, messages: list[Message], stream: Literal[False]) -> chat.ChatCompletion:
        pass

    async def _completions_create(
        self, messages: list[Message], stream: bool
    ) -> chat.ChatCompletion | AsyncStream[ChatCompletionChunk]:
        # standalone function to make it easier to override
        if not self.tools:
            tool_choice: Literal['none', 'required', 'auto'] | None = None
        elif not self.allow_text_result:
            tool_choice = 'required'
        else:
            tool_choice = 'auto'

        openai_messages = [self._map_message(m) for m in messages]
        return await self.client.chat.completions.create(
            model=self.model_name,
            messages=openai_messages,
            n=1,
            parallel_tool_calls=True if self.tools else NOT_GIVEN,
            tools=self.tools or NOT_GIVEN,
            tool_choice=tool_choice or NOT_GIVEN,
            stream=stream,
            stream_options={'include_usage': True} if stream else NOT_GIVEN,
        )

    @staticmethod
    def _process_response(response: chat.ChatCompletion) -> ModelAnyResponse:
        """Process a non-streamed response, and prepare a message to return."""
        timestamp = datetime.fromtimestamp(response.created, tz=timezone.utc)
        choice = response.choices[0]
        if choice.message.tool_calls is not None:
            return ModelStructuredResponse(
                [ToolCall.from_json(c.function.name, c.function.arguments, c.id) for c in choice.message.tool_calls],
                timestamp=timestamp,
            )
        else:
            assert choice.message.content is not None, choice
            return ModelTextResponse(choice.message.content, timestamp=timestamp)

    @staticmethod
    async def _process_streamed_response(response: AsyncStream[ChatCompletionChunk]) -> EitherStreamedResponse:
        """Process a streamed response, and prepare a streaming response to return."""
        timestamp: datetime | None = None
        start_cost = Cost()
        # the first chunk may contain enough information so we iterate until we get either `tool_calls` or `content`
        while True:
            try:
                chunk = await response.__anext__()
            except StopAsyncIteration as e:
                raise UnexpectedModelBehavior('Streamed response ended without content or tool calls') from e

            timestamp = timestamp or datetime.fromtimestamp(chunk.created, tz=timezone.utc)
            start_cost += _map_cost(chunk)

            if chunk.choices:
                delta = chunk.choices[0].delta

                if delta.content is not None:
                    return OpenAIStreamTextResponse(delta.content, response, timestamp, start_cost)
                elif delta.tool_calls is not None:
                    return OpenAIStreamStructuredResponse(
                        response,
                        {c.index: c for c in delta.tool_calls},
                        timestamp,
                        start_cost,
                    )
                # else continue until we get either delta.content or delta.tool_calls

    @staticmethod
    def _map_message(message: Message) -> chat.ChatCompletionMessageParam:
        """Just maps a `pydantic_ai.Message` to a `openai.types.ChatCompletionMessageParam`."""
        if message.role == 'system':
            # SystemPrompt ->
            return chat.ChatCompletionSystemMessageParam(role='system', content=message.content)
        elif message.role == 'user':
            # UserPrompt ->
            return chat.ChatCompletionUserMessageParam(role='user', content=message.content)
        elif message.role == 'tool-return':
            # ToolReturn ->
            return chat.ChatCompletionToolMessageParam(
                role='tool',
                tool_call_id=_guard_tool_call_id(message),
                content=message.model_response_str(),
            )
        elif message.role == 'retry-prompt':
            # RetryPrompt ->
            if message.tool_name is None:
                return chat.ChatCompletionUserMessageParam(role='user', content=message.model_response())
            else:
                return chat.ChatCompletionToolMessageParam(
                    role='tool',
                    tool_call_id=_guard_tool_call_id(message),
                    content=message.model_response(),
                )
        elif message.role == 'model-text-response':
            # ModelTextResponse ->
            return chat.ChatCompletionAssistantMessageParam(role='assistant', content=message.content)
        elif message.role == 'model-structured-response':
            assert (
                message.role == 'model-structured-response'
            ), f'Expected role to be "llm-tool-calls", got {message.role}'
            # ModelStructuredResponse ->
            return chat.ChatCompletionAssistantMessageParam(
                role='assistant',
                tool_calls=[_map_tool_call(t) for t in message.calls],
            )
        else:
            assert_never(message)
```

OpenAIStreamTextResponse `dataclass`

Bases: `StreamTextResponse`

Implementation of `StreamTextResponse` for OpenAI models.

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

```python
@dataclass
class OpenAIStreamTextResponse(StreamTextResponse):
    """Implementation of `StreamTextResponse` for OpenAI models."""

    _first: str | None
    _response: AsyncStream[ChatCompletionChunk]
    _timestamp: datetime
    _cost: result.Cost
    _buffer: list[str] = field(default_factory=list, init=False)

    async def __anext__(self) -> None:
        if self._first is not None:
            self._buffer.append(self._first)
            self._first = None
            return None

        chunk = await self._response.__anext__()
        self._cost += _map_cost(chunk)
        try:
            choice = chunk.choices[0]
        except IndexError:
            raise StopAsyncIteration()

        # we don't raise StopAsyncIteration on the last chunk because usage comes after this
        if choice.finish_reason is None:
            assert choice.delta.content is not None, f'Expected delta with content, invalid chunk: {chunk!r}'
        if choice.delta.content is not None:
            self._buffer.append(choice.delta.content)

    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
```

OpenAIStreamStructuredResponse `dataclass`

Bases: `StreamStructuredResponse`

Implementation of `StreamStructuredResponse` for OpenAI models.

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

```python
@dataclass
class OpenAIStreamStructuredResponse(StreamStructuredResponse):
    """Implementation of `StreamStructuredResponse` for OpenAI models."""

    _response: AsyncStream[ChatCompletionChunk]
    _delta_tool_calls: dict[int, ChoiceDeltaToolCall]
    _timestamp: datetime
    _cost: result.Cost

    async def __anext__(self) -> None:
        chunk = await self._response.__anext__()
        self._cost += _map_cost(chunk)
        try:
            choice = chunk.choices[0]
        except IndexError:
            raise StopAsyncIteration()

        if choice.finish_reason is not None:
            raise StopAsyncIteration()

        assert choice.delta.content is None, f'Expected tool calls, got content instead, invalid chunk: {chunk!r}'

        for new in choice.delta.tool_calls or []:
            if current := self._delta_tool_calls.get(new.index):
                if current.function is None:
                    current.function = new.function
                elif new.function is not None:
                    current.function.name = _utils.add_optional(current.function.name, new.function.name)
                    current.function.arguments = _utils.add_optional(current.function.arguments, new.function.arguments)
            else:
                self._delta_tool_calls[new.index] = new

    def get(self, *, final: bool = False) -> ModelStructuredResponse:
        calls: list[ToolCall] = []
        for c in self._delta_tool_calls.values():
            if f := c.function:
                if f.name is not None and f.arguments is not None:
                    calls.append(ToolCall.from_json(f.name, f.arguments, c.id))

        return ModelStructuredResponse(calls, timestamp=self._timestamp)

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

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