Dependencies

PydanticAI uses a dependency injection system to provide data and services to your agent's system prompts, tools and result validators.

Matching PydanticAI's design philosophy, our dependency system tries to use existing best practice in Python development rather than inventing esoteric "magic", this should make dependencies type-safe, understandable easier to test and ultimately easier to deploy in production.

Defining Dependencies

Dependencies can be any python type. While in simple cases you might be able to pass a single object as a dependency (e.g. an HTTP connection), dataclasses are generally a convenient container when your dependencies included multiple objects.

Here's an example of defining an agent that requires dependencies.

(**Note:** dependencies aren't actually used in this example, see Accessing Dependencies below)

unused_dependencies.py

```python
from dataclasses import dataclass

import httpx

from pydantic_ai import Agent


@dataclass
class MyDeps:  (1)
    api_key: str
    http_client: httpx.AsyncClient


agent = Agent(
    'openai:gpt-4o',
    deps_type=MyDeps,  (2)
)


async def main():
    async with httpx.AsyncClient() as client:
        deps = MyDeps('foobar', client)
        result = await agent.run(
            'Tell me a joke.',
            deps=deps,  (3)
        )
        print(result.data)
        #> Did you hear about the toothpaste scandal? They called it Colgate.
```

1. Define a dataclass to hold dependencies.
2. Pass the dataclass type to the `deps_type` argument of the `Agent` constructor. **Note**: we're passing the type here, NOT an instance, this parameter is not actually used at runtime, it's here so we can get full type checking of the agent.
3. When running the agent, pass an instance of the dataclass to the `deps` parameter.

(This example is complete, it can be run "as is")

Accessing Dependencies

Dependencies are accessed through the `RunContext` type, this should be the first parameter of system prompt functions etc.

system_prompt_dependencies.py

```python
from dataclasses import dataclass

import httpx

from pydantic_ai import Agent, RunContext


@dataclass
class MyDeps:
    api_key: str
    http_client: httpx.AsyncClient


agent = Agent(
    'openai:gpt-4o',
    deps_type=MyDeps,
)


@agent.system_prompt  (1)
async def get_system_prompt(ctx: RunContext[MyDeps]) -> str:  (2)
    response = await ctx.deps.http_client.get(  (3)
        'https://example.com',
        headers={'Authorization': f'Bearer {ctx.deps.api_key}'},  (4)
    )
    response.raise_for_status()
    return f'Prompt: {response.text}'


async def main():
    async with httpx.AsyncClient() as client:
        deps = MyDeps('foobar', client)
        result = await agent.run('Tell me a joke.', deps=deps)
        print(result.data)
        #> Did you hear about the toothpaste scandal? They called it Colgate.
```

1. `RunContext` may optionally be passed to a `system_prompt` function as the only argument.
2. `RunContext` is parameterized with the type of the dependencies, if this type is incorrect, static type checkers will raise an error.
3. Access dependencies through the `.deps` attribute.
4. Access dependencies through the `.deps` attribute.

(This example is complete, it can be run "as is")

Asynchronous vs. Synchronous dependencies

System prompt functions, function tools and result validators are all run in the async context of an agent run.

If these functions are not coroutines (e.g. `async def`) they are called with `run_in_executor` in a thread pool, it's therefore marginally preferable to use `async` methods where dependencies perform IO, although synchronous dependencies should work fine too.

`run` vs. `run_sync` and Asynchronous vs. Synchronous dependencies

Whether you use synchronous or asynchronous dependencies, is completely independent of whether you use `run` or `run_sync` — `run_sync` is just a wrapper around `run` and agents are always run in an async context.

Here's the same example as above, but with a synchronous dependency:

sync_dependencies.py

```python
from dataclasses import dataclass

import httpx

from pydantic_ai import Agent, RunContext


@dataclass
class MyDeps:
    api_key: str
    http_client: httpx.Client  (1)


agent = Agent(
    'openai:gpt-4o',
    deps_type=MyDeps,
)


@agent.system_prompt
def get_system_prompt(ctx: RunContext[MyDeps]) -> str:  (2)
    response = ctx.deps.http_client.get(
        'https://example.com', headers={'Authorization': f'Bearer {ctx.deps.api_key}'}
    )
    response.raise_for_status()
    return f'Prompt: {response.text}'


async def main():
    deps = MyDeps('foobar', httpx.Client())
    result = await agent.run(
        'Tell me a joke.',
        deps=deps,
    )
    print(result.data)
    #> Did you hear about the toothpaste scandal? They called it Colgate.
```

1. Here we use a synchronous `httpx.Client` instead of an asynchronous `httpx.AsyncClient`.
2. To match the synchronous dependency, the system prompt function is now a plain function, not a coroutine.

(This example is complete, it can be run "as is")

Full Example

As well as system prompts, dependencies can be used in tools and result validators.

full_example.py

```python
from dataclasses import dataclass

import httpx

from pydantic_ai import Agent, ModelRetry, RunContext


@dataclass
class MyDeps:
    api_key: str
    http_client: httpx.AsyncClient


agent = Agent(
    'openai:gpt-4o',
    deps_type=MyDeps,
)


@agent.system_prompt
async def get_system_prompt(ctx: RunContext[MyDeps]) -> str:
    response = await ctx.deps.http_client.get('https://example.com')
    response.raise_for_status()
    return f'Prompt: {response.text}'


@agent.tool  (1)
async def get_joke_material(ctx: RunContext[MyDeps], subject: str) -> str:
    response = await ctx.deps.http_client.get(
        'https://example.com#jokes',
        params={'subject': subject},
        headers={'Authorization': f'Bearer {ctx.deps.api_key}'},
    )
    response.raise_for_status()
    return response.text


@agent.result_validator  (2)
async def validate_result(ctx: RunContext[MyDeps], final_response: str) -> str:
    response = await ctx.deps.http_client.post(
        'https://example.com#validate',
        headers={'Authorization': f'Bearer {ctx.deps.api_key}'},
        params={'query': final_response},
    )
    if response.status_code == 400:
        raise ModelRetry(f'invalid response: {response.text}')
    response.raise_for_status()
    return final_response
```

```python
async def main():
    async with httpx.AsyncClient() as client:
        deps = MyDeps('foobar', client)
        result = await agent.run('Tell me a joke.', deps=deps)
        print(result.data)
        #> Did you hear about the toothpaste scandal? They called it Colgate.
```

1. To pass `RunContext` to a tool, use the `tool` decorator.
2. `RunContext` may optionally be passed to a `result_validator` function as the first argument.

(This example is complete, it can be run "as is")

Overriding Dependencies

When testing agents, it's useful to be able to customise dependencies.

While this can sometimes be done by calling the agent directly within unit tests, we can also override dependencies while calling application code which in turn calls the agent.

This is done via the `override` method on the agent.

joke_app.py

```python
from dataclasses import dataclass

import httpx

from pydantic_ai import Agent, RunContext


@dataclass
class MyDeps:
    api_key: str
    http_client: httpx.AsyncClient

    async def system_prompt_factory(self) -> str:  (1)
        response = await self.http_client.get('https://example.com')
        response.raise_for_status()
        return f'Prompt: {response.text}'


joke_agent = Agent('openai:gpt-4o', deps_type=MyDeps)


@joke_agent.system_prompt
async def get_system_prompt(ctx: RunContext[MyDeps]) -> str:
    return await ctx.deps.system_prompt_factory()  (2)


async def application_code(prompt: str) -> str:  (3)
    ...
    ...
    # now deep within application code we call our agent
    async with httpx.AsyncClient() as client:
        app_deps = MyDeps('foobar', client)
        result = await joke_agent.run(prompt, deps=app_deps)  (4)
    return result.data
```

1. Define a method on the dependency to make the system prompt easier to customise.
2. Call the system prompt factory from within the system prompt function.
3. Application code that calls the agent, in a real application this might be an API endpoint.
4. Call the agent from within the application code, in a real application this call might be deep within a call stack. Note `app_deps` here will NOT be used when deps are overridden.

test_joke_app.py

```python
from joke_app import MyDeps, application_code, joke_agent


class TestMyDeps(MyDeps):  (1)
    async def system_prompt_factory(self) -> str:
        return 'test prompt'


async def test_application_code():
    test_deps = TestMyDeps('test_key', None)  (2)
    with joke_agent.override(deps=test_deps):  (3)
        joke = await application_code('Tell me a joke.')  (4)
    assert joke.startswith('Did you hear about the toothpaste scandal?')
```

1. Define a subclass of `MyDeps` in tests to customise the system prompt factory.
2. Create an instance of the test dependency, we don't need to pass an `http_client` here as it's not used.
3. Override the dependencies of the agent for the duration of the `with` block, `test_deps` will be used when the agent is run.
4. Now we can safely call our application code, the agent will use the overridden dependencies.

Agents as dependencies of other Agents

Since dependencies can be any python type, and agents are just python objects, agents can be dependencies of other agents.

agents_as_dependencies.py

```python
from dataclasses import dataclass

from pydantic_ai import Agent, RunContext


@dataclass
class MyDeps:
    factory_agent: Agent[None, list[str]]


joke_agent = Agent(
    'openai:gpt-4o',
    deps_type=MyDeps,
    system_prompt=(
        'Use the "joke_factory" to generate some jokes, then choose the best. '
        'You must return just a single joke.'
    ),
)
```

```python
factory_agent = Agent('gemini-1.5-pro', result_type=list[str])


@joke_agent.tool
async def joke_factory(ctx: RunContext[MyDeps], count: int) -> str:
    r = await ctx.deps.factory_agent.run(f'Please generate {count} jokes.')
    return '\n'.join(r.data)


result = joke_agent.run_sync('Tell me a joke.', deps=MyDeps(factory_agent))
print(result.data)
#> Did you hear about the toothpaste scandal? They called it Colgate.
```

Examples

The following examples demonstrate how to use dependencies in PydanticAI:

- Weather Agent
- SQL Generation
- RAG