Using Dataclasses

FastAPI is built on top of **Pydantic**, and I have been showing you how to use Pydantic models to declare requests and responses.

But FastAPI also supports using `dataclasses` the same way:

```
{!../../../docs_src/dataclasses/tutorial001.py!}
```

This is still thanks to **Pydantic**, as it has internal support for `dataclasses` .

So, even with the code above that doesn't use Pydantic explicitly, FastAPI is using Pydantic to convert those standard dataclasses to Pydantic's own flavor of dataclasses.

And of course, it supports the same:

- data validation
- data serialization
- data documentation, etc.

This works the same way as with Pydantic models. And it is actually achieved in the same way underneath, using Pydantic.

!!! info Have in mind that dataclasses can't do everything Pydantic models can do.

```
So, you might still need to use Pydantic models.

But if you have a bunch of dataclasses laying around, this is a nice trick to use them
to power a web API using FastAPI. 🤓
```

Dataclasses in `response_model`

You can also use `dataclasses` in the `response_model` parameter:

```
{!../../../docs_src/dataclasses/tutorial002.py!}
```

The dataclass will be automatically converted to a Pydantic dataclass.

This way, its schema will show up in the API docs user interface:

Dataclasses in Nested Data Structures

You can also combine `dataclasses` with other type annotations to make nested data structures.

In some cases, you might still have to use Pydantic's version of `dataclasses` . For example, if you have errors with the automatically generated API documentation.

In that case, you can simply swap the standard `dataclasses` with `pydantic.dataclasses` , which is a drop-in replacement:

```
{!../../../docs_src/dataclasses/tutorial003.py!}
```

1. We still import `field` from standard `dataclasses`.
2. `pydantic.dataclasses` is a drop-in replacement for `dataclasses`.
3. The `Author` dataclass includes a list of `Item` dataclasses.
4. The `Author` dataclass is used as the `response_model` parameter.
5. You can use other standard type annotations with dataclasses as the request body.

   In this case, it's a list of `Item` dataclasses.
6. Here we are returning a dictionary that contains `items` which is a list of dataclasses.

   FastAPI is still capable of serializing the data to JSON.
7. Here the `response_model` is using a type annotation of a list of `Author` dataclasses.

   Again, you can combine `dataclasses` with standard type annotations.
8. Notice that this *path operation function* uses regular `def` instead of `async def`.

   As always, in FastAPI you can combine `def` and `async def` as needed.

   If you need a refresher about when to use which, check out the section *"In a hurry?"* in the docs about `async` and `await`.
9. This *path operation function* is not returning dataclasses (although it could), but a list of dictionaries with internal data.

   FastAPI will use the `response_model` parameter (that includes dataclasses) to convert the response.

You can combine `dataclasses` with other type annotations in many different combinations to form complex data structures.

Check the in-code annotation tips above to see more specific details.

Learn More

You can also combine `dataclasses` with other Pydantic models, inherit from them, include them in your own models, etc.

To learn more, check the Pydantic docs about dataclasses.

Version

This is available since FastAPI version `0.67.0`.