Body - Updates

Update replacing with `PUT`

To update an item you can use the HTTP `PUT` operation.

You can use the `jsonable_encoder` to convert the input data to data that can be stored as JSON (e.g. with a NoSQL database). For example, converting `datetime` to `str`.

Python 3.10+

```
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded
```

Python 3.8+

```
from typing import List, Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel
```

```python
app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: List[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded
```

`PUT` is used to receive data that should replace the existing data.

Warning about replacing

That means that if you want to update the item `bar` using `PUT` with a body containing:

```
{
    "name": "Barz",
    "price": 3,
    "description": None,
}
```

because it doesn't include the already stored attribute `"tax": 20.2`, the input model would take the default value of `"tax": 10.5`.

And the data would be saved with that "new" `tax` of `10.5`.

Partial updates with `PATCH`

You can also use the HTTP `PATCH` operation to *partially* update data.

This means that you can send only the data that you want to update, leaving the rest intact.

Note

`PATCH` is less commonly used and known than `PUT`.

And many teams use only `PUT`, even for partial updates.

You are **free** to use them however you want, **FastAPI** doesn't impose any restrictions.

But this guide shows you, more or less, how they are intended to be used.

Using Pydantic's `exclude_unset` parameter

If you want to receive partial updates, it's very useful to use the parameter `exclude_unset` in Pydantic's model's `.model_dump()`.

Like `item.model_dump(exclude_unset=True)`.

Info

In Pydantic v1 the method was called `.dict()`, it was deprecated (but still supported) in Pydantic v2, and renamed to `.model_dump()`.

The examples here use `.dict()` for compatibility with Pydantic v1, but you should use `.model_dump()` instead if you can use Pydantic v2.

That would generate a `dict` with only the data that was set when creating the `item` model, excluding default values.

Then you can use this to generate a `dict` with only the data that was set (sent in the request), omitting default values:

Python 3.10+

```python
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()
```

```python
class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

▸ Other versions and variants

Python 3.9+

```python
from typing import Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Python 3.8+

```python
from typing import List, Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: List[str] = []
```

```
items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Using Pydantic's `update` parameter

Now, you can create a copy of the existing model using `.model_copy()`, and pass the `update` parameter with a `dict` containing the data to update.

Info

In Pydantic v1 the method was called `.copy()`, it was deprecated (but still supported) in Pydantic v2, and renamed to `.model_copy()`.

The examples here use `.copy()` for compatibility with Pydantic v1, but you should use `.model_copy()` instead if you can use Pydantic v2.

Like `stored_item_model.model_copy(update=update_data)`:

Python 3.10+

```
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

▸ Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
```

```
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Python 3.8+

```
from typing import List, Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: List[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Partial updates recap

In summary, to apply partial updates you would:

- (Optionally) use `PATCH` instead of `PUT`.
- Retrieve the stored data.
- Put that data in a Pydantic model.
- Generate a `dict` without default values from the input model (using `exclude_unset`).
  - This way you can update only the values actually set by the user, instead of overriding values already stored with default values in your model.
- Create a copy of the stored model, updating its attributes with the received partial updates (using the `update` parameter).
- Convert the copied model to something that can be stored in your DB (for example, using the `jsonable_encoder`).
  - This is comparable to using the model's `.model_dump()` method again, but it makes sure (and converts) the values to data types that can be converted to JSON, for example, `datetime` to `str`.
- Save the data to your DB.
- Return the updated model.

Python 3.10+

```
from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel
```

```
app = FastAPI()


class Item(BaseModel):
    name: str | None = None
    description: str | None = None
    price: float | None = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

▸ 🤓 Other versions and variants

Python 3.9+

```
from typing import Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: list[str] = []


items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Python 3.8+

```
from typing import List, Union

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: Union[str, None] = None
    description: Union[str, None] = None
    price: Union[float, None] = None
    tax: float = 10.5
    tags: List[str] = []
```

```python
items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.patch("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True)
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item
```

Tip

You can actually use this same technique with an HTTP `PUT` operation.

But the example here uses `PATCH` because it was created for these use cases.

Note

Notice that the input model is still validated.

So, if you want to receive partial updates that can omit all the attributes, you need to have a model with all the attributes marked as optional (with default values or `None`).

To distinguish from the models with all optional values for **updates** and models with required values for **creation**, you can use the ideas described in Extra Models.