Dynamically serving REST endpoints for database stored procedures with FastAPI

Kim van Wyk

PyconZA 2023

5/6 October 2023

What is this about?

- My employer has a large number of Microsoft SQL Server stored procedures
  - The heart of several important backend systems
  - Built up over many years
  - Battle-hardened and extensively debugged and developed
  - Reliable and well-understood means of effecting controlled, auditable system changes
  - Consistently defined - documented params in, *Msg* param out

Accessing with other systems

- Would be beneficial if internal or third-party software could use these stored procs
  - No need to reinvent a perfectly good wheel
  - Direct access would require DB host access and credentials (albeit limited).
    - This does not delight security teams or DBAs (completely validly)
  - Many systems don't understand SQL or stored procs
- REST endpoints would do nicely
  - Greatly understood and supported by other systems
  - Securable
  - Swagger and similar make for good discoverability

Chosen approach

- FastAPI and underlying Pydantic models
  - Provides the REST benefits out of the box
  - Can be dynamically generated to cater for changing stored procs
- Turned out to be completely doable but not as smoothly as hoped
  - Stored proc query
  - ~~Dynamic FastAPI endpoints~~
  - Generated FastAPI code
  - Dockerisation

Retrieving stored proc details

- As would be expected for such a venerable and well-used technology, querying needed details about the stored procs is fairly easy

```
USE database;
SELECT o.name AS [proc_name], par.name AS [param_name],
types.name AS [param_dtype], par.is_output AS [is_output]
FROM sys.objects o
JOIN sys.schemas s ON o.schema_id = s.schema_id
INNER JOIN sys.procedures p ON o.object_id = p.object_id
INNER JOIN sys.parameters par ON par.object_id = p.object_i
INNER JOIN sys.types ON par.system_type_id = types.system_t
AND par.user_type_id = types.user_type_id
WHERE o.type_desc = 'SQL_STORED_PROCEDURE'
AND s.name = schema
ORDER BY s.name, o.name, par.name
```

Query result

proc_name	param_name	param_dtype	is_output
Proc1	@Param1	int	0
Proc1	@Param2	varchar	0
Proc1	@Param3	varchar	1
Proc2	@Param1	datetime	0
Proc2	@Param2	decimal	1

Dynamic FastAPI endpoints

- First plan was to dynamically generate FastAPI endpoints from a SQLAlchemied query result
- Pydantic's *create_model* method could create input and output models
- Could not find a reasonable way to have FastAPI use such Pydantic models
  - examples and documentation all refer to already-existing named models
  - Some clever solutions in discussions in GitHub issues
    - More Pythonic naval gazing than I was comfortable putting into my code

Plan B: code generation to the rescue

- Enter the Jinja2 templating engine
  - Commonly associated with Django HTML templating
  - Can be run standalone to generate any text
- Templates to generate Pydantic *models.py* and FastAPI *main.py* files are not overly complex

Models template

```
{% for (name,mods) in models.items() %}
    {% for (direction, params) in mods.items() %}
{% if params %}
    class {{ name }}{% if direction == "in" %}InputModel
                    {% else %}OutputModel{% endif %}(BaseModel):
        {% for (field, types) in params %}{{ field }}: {{ types[1] }}
            {% if direction == "out" %} = None{% endif %}
        {% endfor %}_db_fields: dict = PrivateAttr()

        def __init__(self, **data):
            self._db_fields = {
                {% for (field, types) in params %}
                    "{{ field }}": {{ types[0] }},
                {% endfor %} }
```

Resulting models file

```
class Proc1InputModel(BaseModel):
    Param1: int
    Param2: str
    _db_fields: dict = PrivateAttr()

    def __init__(self, **data):
        self._db_fields = {
            "Param1": XX,
            "Param2": XX,
             }

class Proc2OutputModel(BaseModel):
    Param3: str = None
    _db_fields: dict = PrivateAttr()

    def __init__(self, **data):
        self._db_fields = {
            "Param3": XX,
             }
```

FastAPI template

```
{% for (name, mods) in models.items() %}
@app.post("/{{ name }}")
def {{ name }}({% if mods["in"] %}
    input_model: models.{{ name }}InputModel{% endif %}):
    {% if not mods["in"] %}input_model = None{% endif %}
    output_model = {% if mods["out"] %}
        models.{{ name }}OutputModel(){% else %}None{% endif %}
    try:
        om = DBH.exec_stored_proc("{{ database }}.{{ schema }}.{{ name }}",
            input_model, output_model)
    except db.StoredProcError as e:
        raise HTTPException(status_code = 400, detail = str(e))
    return om
{% endfor %}
```

Resulting FastAPI file

```
@app.post("/Proc1")
def Proc1(input_model: models.Proc1InputModel):

    output_model = models.Proc1OutputModel()
    try:
        om = DBH.exec_stored_proc("db_name.schema.table",
            input_model, output_model)
    except db.StoredProcError as e:
        raise HTTPException(status_code = 400, detail = str(e))
    return om
```

- Small swagger demo goes here

Dockerisation

- Multi-step process lends itself well to Dockerisation
  - **RUN** layers to generate the code
  - **ENTRYPOINT** to execute the resulting code in FastAPI
- If stored procs change the image just needs to be re-deployed

Potential improvements, benefits and lessons learned

- Reading doc strings from stored procs and serving them in endpoints
- Stored procs could be seamlessly replaced with a different underlying system
- Consistent and well-structured stored procs aren't as common as they should be - thank your DB teams if yours are
- Code generation can be less complex than you may think
- A syntax highlighting text editor is really handy for this kind of thing

And finally

- This talk is a fleshed out version of a blog post I wrote some time ago
- Slides and source in this GitLab repo
- Thank you for listening
  - vanwykk@gmail.com
  - @kimvanwyk in the conference Discord