Using Kubernetes operators & resource definitions for ngUML runtime generation

Max Boone

LIACS

25th of May 2023

Contents

Current situation and background

ngUML end-to-end is "prose(-to-metadata)-to-prototype". This project focuses on how to get from metadata to prototype, in production.

- Runtime through code generation
- Generation by `str.format` templating
- Formatted and structured as Django Application (v3.1)
- Stored in-tree / in modeler's Django Project

Complications

- Same directory as main server directory:
  - Exceptions bubble up, killing entire system
  - Race conditions, as same thread is utilized
- Code generation to flat files:
  - Production environments' FS immutable
  - Large risk for injection of malicious code
  - Migration impractical, as entire system is overwritten

(Research) Question

Is a CRD-based approach a viable alternative to templated code generation for the ngUML runtime?

Objectives

1. Kubernetes Deployment
2. Runtime CRDs & Controller
3. Django ORM construction without code generation

Objectives

Kubernetes Deployment

Re-write the current `docker-compose.yaml` to a helm chart with a `values.yaml`. Add service account as well.

Update `README.md` with instructions to helm deploy locally (k3s, minikube, ...). Use **Gefyra**?

Objectives

Runtime CRDs & Controller

Kubernetes Documentation: Operator Pattern

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

Write CRDs for runtime metadata & runtime configuration. The custom controller should then build, upgrade or destroy a runtime (with database) from this definition.

Objectives

Django ORM construction

Currently, the system generates Python code and stores it as files. We should be able to use the Django ORM without first rendering all metadata as Python code. We can find inspiration for this in **Baserow**: the **SchemaEditor**.

Django Documentation: SchemaEditor

It's unlikely that you will want to interact directly with SchemaEditor as a normal developer using Django, but if you want to write your own migration system, or have more advanced needs, it's a lot nicer than writing SQL.

Impact & Risks

- **Helm Chart**
  Must ensure no degradation in DX (and only required iff runtime needed). Will impact code from all developers.
- **Runtime CRDs & Controller**
  Will be written in separate app (orchestrator).
- **Django ORM Construction**
  Will be written in separate app (runtime).

Something else

Problem

Currently, the database structure is absurdly complex. We have 43-ish tables to store nodes in a diagram, node type has it's own table & django model.

Polymorphism - What is a chair?

Polymorphism, Jerry! It's like having a closet full of clothes, but every time you reach for a shirt, it magically turns into a pair of pants!

Django models don't support polymorphism, so if you call `objects.all()` or `objects.get(pk=id)` on `Classifier` it does not return for any models that inherit `Classifier`.

Something else

What to do?

NoSQL to the rescue. We're not going to fully implement the abstract structure in a relational database. Instead: `JSONField`.

However, no validation is dangerous. Instead, we write JSON Schemas for nodes, extend Django ORM to use `pg-jsonschema` and generate type definitions for Python & TypeScript from the JSON Schemas.

Major change, as all code that directly calls the current classes will need changes. If you stick to using the internal API, you should be fine. PR must be non-breaking and requires review from **everyone currently working**. Expected in two weeks.