hmis-codegen Guide

Overview

`hmis-codegen` is a Python toolchain that bridges semantic web ontologies with practical API development. It parses YAML-LD specifications and JSON-LD ontologies to generate:

- **JSON-LD contexts** for semantic API responses
- **Mockoon configurations** for sandbox/mock servers
- **Effect handler interfaces** with algebraic effect annotations
- **Mock handlers** for unit testing

Installation

```
# Clone the repository
git clone https://github.com/HUD-Data-Lab/hmis-codegen.git
cd hmis-codegen

# Install in development mode
pip install -e .

# Or install with dev dependencies
pip install -e ".[dev]"

# Verify installation
hmis-codegen --help
```

Basic Usage

1. Generate All Outputs

The simplest way to use the toolchain:

```
hmis-codegen generate \
  --ontology path/to/FY26HMIS_JSON-LD_v1.jsonld \
  --spec path/to/hmis_api_ld_0.4.yaml \
  --output-dir ./generated
```

This produces:

```
generated/
├── context.jsonld           # JSON-LD @context
├── openapi.yaml             # Enhanced OpenAPI spec
├── mockoon-env.json         # Mockoon environment
├── effect_handlers.py       # Handler interfaces
└── effect_mocks.py          # Mock handlers for testing
```

2. Validate Your Specification

Before generating outputs, validate that your YAML-LD spec is correct:

```
hmis-codegen validate \
  --ontology ontology.jsonld \
  --spec hmis_api_ld_0.4.yaml
```

Output:

```
✓ Validation passed!

✓ 3 schemas validated
✓ 3 operations validated
✓ 9 effects validated
✓ All semantic URIs reference valid ontology terms
```

3. Extract Semantic Mappings

Useful for debugging or understanding the semantic structure:

```
# JSON format (default)
hmis-codegen extract --spec api.yaml --output mappings.json

# YAML format for readability
hmis-codegen extract --spec api.yaml --output mappings.yaml --
format yaml

# CSV format for spreadsheet analysis
hmis-codegen extract --spec api.yaml --output mappings.csv --
format csv
```

4. View Specification Info

Get a quick overview of your specification:

```
hmis-codegen info --spec hmis_api_ld_0.4.yaml
```

Output includes:

- Basic metadata (title, version, context URL)
- Statistics (schemas, operations, effects)
- Experimental features in use
- Compliance references

Generated Artifacts

JSON-LD Context (`context.jsonld`)

Maps OpenAPI field names to ontology properties:

```
{
  "@context": {
    "@vocab": "http://hmis.hud.gov/ontology#",
    "FirstName": "hasFirstName",
    "LastName": "hasLastName",
    "SSN": "hasSSN",
    "DOB": "hasDateOfBirth",
    "VeteranStatus": "hasVeteranStatus"
  }
}
```

Use Case: Vendors can include this context in API responses to make them RDF-compatible:

```
{
  "@context": "https://hmis.hud.gov/context.jsonld",
  "@type": "Client",
  "PersonalID": "123",
  "FirstName": "John",
  "LastName": "Doe"
}
```

Effect Handlers (`effect_handlers.py`)

Type-safe Protocol classes for implementing effects:

```
from effect_handlers import ConsentCheckHandler,
DatabaseWriteHandler

class MyConsentChecker:
    def handle(self, client_id: str, purpose: str) →
ConsentStatus:
        # Check consent records
        roi = get_roi(client_id, purpose)
        if roi and roi.is_active():
            return ConsentStatus.GRANTED
        return ConsentStatus.DENIED

# Register handlers
from effect_handlers import register_handler
register_handler("ConsentCheck", MyConsentChecker())
```

Mock Handlers (`effect_mocks.py`)

Pre-built mocks for unit testing:

```
from effect_mocks import setup_mock_handlers, ConsentStatus

def test_client_creation():
    # Setup mocks
    handlers = setup_mock_handlers()
    handlers['ConsentCheck'].default_result =
ConsentStatus.GRANTED

    # Run your code that triggers effects
    result = create_client(data)

    # Verify effects were called
    assert len(handlers['ConsentCheck'].calls) == 1
    assert handlers['ConsentCheck'].calls[0]['purpose'] ==
'case_management'
    assert len(handlers['DatabaseWrite'].calls) == 1
```

Mockoon Configuration (`mockoon-env.json`)

Import directly into Mockoon for a working mock server:

1. Open Mockoon
2. Settings → Import/Export → Import environment

3. Select `mockoon-env.json`
4. Start the mock server on `localhost:3000`

Now vendors can test against your API without writing any backend code!

Advanced Usage

Generate Specific Outputs

```
# Only JSON-LD context
hmis-codegen generate -o ontology.jsonld -s api.yaml -f context

# Only Mockoon config
hmis-codegen generate -o ontology.jsonld -s api.yaml -f mockoon

# Only effect handlers
hmis-codegen generate -o ontology.jsonld -s api.yaml -f handlers

# Only mock handlers
hmis-codegen generate -o ontology.jsonld -s api.yaml -f mocks
```

Custom Output Directory

```
hmis-codegen generate \
  -o ontology.jsonld \
  -s api.yaml \
  --output-dir ~/my-project/generated
```

Batch Processing

Generate outputs for multiple specifications:

```
#!/bin/bash
for spec in specs/*.yaml; do
  echo "Processing $spec..."
  hmis-codegen generate \
    -o ontology.jsonld \
    -s "$spec" \
    -d "generated/$(basename $spec .yaml)"
done
```

Integration Examples

Using Generated Handlers in FastAPI

```python
from fastapi import FastAPI, HTTPException
from effect_handlers import register_handler, get_handler
from pydantic import BaseModel

app = FastAPI()

# Setup effect handlers
from my_handlers import (
    ConsentChecker,
    DatabaseWriter,
    ValidationChecker,
    AuditLogger
)

register_handler("ConsentCheck", ConsentChecker())
register_handler("DatabaseWrite", DatabaseWriter())
register_handler("ValidationCheck", ValidationChecker())
register_handler("AuditLog", AuditLogger())

class ClientCreate(BaseModel):
    FirstName: str
    LastName: str
    DOB: str

@app.post("/client")
async def create_client(client: ClientCreate):
    # Execute effects in order
    consent = get_handler("ConsentCheck").handle(
        client_id=None,
        purpose="intake"
    )
    if consent ≠ ConsentStatus.GRANTED:
        raise HTTPException(403, "Consent required")

    validation = get_handler("ValidationCheck").handle(
        data=client.dict(),
        schema="clientBase"
    )
    if not validation.valid:
        raise HTTPException(400, validation.errors)
```

```
    personal_id = get_handler("DatabaseWrite").handle(
        entity="Client",
        data=client.dict()
    )

    get_handler("AuditLog").handle(
        event={
            "type": "ClientCreated",
            "personal_id": personal_id,
            "timestamp": datetime.utcnow()
        }
    )

    return {"PersonalID": personal_id}
```

Using Mock Handlers in Tests

```
import pytest
from effect_mocks import setup_mock_handlers, ConsentStatus
from my_api import create_client

@pytest.fixture
def mock_handlers():
    handlers = setup_mock_handlers()
    # Set default behaviors
    handlers['ConsentCheck'].default_result =
ConsentStatus.GRANTED
    handlers['DatabaseWrite'].default_result = "client-123"
    return handlers

def test_client_creation_success(mock_handlers):
    """Test successful client creation"""
    result = create_client({
        "FirstName": "Jane",
        "LastName": "Doe"
    })

    assert result["PersonalID"] == "client-123"

    # Verify all effects were called
    assert len(mock_handlers['ConsentCheck'].calls) == 1
    assert len(mock_handlers['ValidationCheck'].calls) == 1
    assert len(mock_handlers['DatabaseWrite'].calls) == 1
```

```
    assert len(mock_handlers['AuditLog'].calls) == 1

def test_client_creation_consent_denied(mock_handlers):
    """Test client creation fails without consent"""
    mock_handlers['ConsentCheck'].default_result =
ConsentStatus.DENIED

    with pytest.raises(HTTPException) as exc:
        create_client({"FirstName": "Jane", "LastName": "Doe"})

    assert exc.value.status_code == 403
    # Verify downstream effects were NOT called
    assert len(mock_handlers['DatabaseWrite'].calls) == 0
```

Using Generated Context with JSON-LD

```
import json
from pyld import jsonld

# Load generated context
with open('generated/context.jsonld') as f:
    context = json.load(f)

# API response with embedded context
response = {
    "@context": context['@context'],
    "@type": "Client",
    "@id": "http://hmis.example.org/client/123",
    "PersonalID": "123",
    "FirstName": "Jane",
    "LastName": "Doe",
    "VeteranStatus": 1
}

# Expand to full RDF
expanded = jsonld.expand(response)
print(json.dumps(expanded, indent=2))

# Convert to RDF triples
from rdflib import Graph
g = Graph()
g.parse(data=json.dumps(response), format='json-ld')
```

```
# Now you can query with SPARQL
query = """
SELECT ?client ?firstName WHERE {
    ?client <http://hmis.hud.gov/ontology#hasFirstName> ?
firstName .
}
"""
results = g.query(query)
for row in results:
    print(f"Client: {row.client}, Name: {row.firstName}")
```

Experimental Features

Algebraic Effect Annotations

The toolchain supports experimental effect annotations in Phase 1:

```
/client:
  post:
    x-effects:
      - effect: ConsentCheck
        handler: hmis:validateConsent
        signature: "clientID: ID, purpose: Purpose →
ConsentStatus"
        resumable: true
        required-by: "2004 HMIS Standards Section 4.2"
```

Benefits:

- **Documentation**: Explicitly shows what side effects operations perform
- **Test generation**: Auto-generate mock handlers
- **Effect reasoning**: "Does this operation require consent?" → Yes, has `ConsentCheck` effect
- **Provenance**: Effects map to PROV-O for audit trails

Phase 2 Vision (Future):

- Runtime effect handlers with delimited continuations
- Composable effect stacks
- Resumable computations

Troubleshooting

Validation Errors

```
Error: Property 'clientBase.FirstName' references unknown URI:
http://...
```

Solution: Ensure the URI exists in your ontology. Check spelling and namespace.

Missing Semantic Annotations

```
Warning: Schema 'clientBase' missing x-semantic-uri annotation
```

Solution: This is just a warning. Add semantic annotations to enable better tooling:

```yaml
clientBase:
  type: object
  x-semantic-uri: http://hmis.hud.gov/ontology#Client  # Add this
```

Effect Handler Not Found

```
Error: Operation postClient references undefined effect:
ConsentCheck
```

Solution: Define the effect in your `x-effect-system` section or ensure effect names match exactly.

Best Practices

1. **Version your ontology**: Use semantic versioning for ontology files
2. **Validate early**: Run `validate` before `generate` to catch errors
3. **Extract mappings**: Use `extract` to document semantic structure
4. **Mock everything**: Use generated mocks in all unit tests
5. **Progressive enhancement**: Start with plain OpenAPI, add semantic annotations incrementally

Development Workflow

```
# 1. Make changes to YAML-LD spec
vim hmis_api_ld_0.4.yaml

# 2. Validate changes
hmis-codegen validate -o ontology.jsonld -s hmis_api_ld_0.4.yaml
```

```
# 3. Generate outputs
hmis-codegen generate -o ontology.jsonld -s hmis_api_ld_0.4.yaml

# 4. Test with Mockoon
# Import generated/mockoon-env.json into Mockoon

# 5. Update vendor integration tests
pytest tests/ --handlers=generated/effect_mocks.py
```

CI/CD Integration

```
# .github/workflows/codegen.yml
name: Generate API Artifacts

on:
  push:
    paths:
      - 'specs/**.yaml'
      - 'ontology/**.jsonld'

jobs:
  generate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install hmis-codegen
        run: pip install -e .

      - name: Validate specification
        run: |
          hmis-codegen validate \
            -o ontology/FY26HMIS_JSON-LD_v1.jsonld \
            -s specs/hmis_api_ld_0.4.yaml

      - name: Generate artifacts
        run: |
```

```
          hmis-codegen generate \
            -o ontology/FY26HMIS_JSON-LD_v1.jsonld \
            -s specs/hmis_api_ld_0.4.yaml \
            -d generated/

      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: generated-api-artifacts
          path: generated/
```

Support & Contributing

- **Documentation**: See `docs/` directory
- **Issues**: GitHub Issues
- **Architecture**: See `docs/architecture.md`
- **Effect System**: See `docs/effect_system.md`

License

MIT License - See LICENSE file for details