Idea

Most web applications provide REST API architecture.

REST APIs are good “data containers”, they provide a summary of application data.

REST API cannot connect with applications that work under OSLC specifications.
Motivation

In order to create OSLC API developers need to employ a certain amount of time to accomplish it but...

What if:

Develop another OSLC API?
Need the OSLC API now?

Partial solution: we can reuse some OSLC API components such as:

- **Data source components** → Fetch data
- **OSLC Components** → Parse data into RDF
- **API endpoints** → HTTP requests
Inspiration

GraphQL:

- Query language for APIs
- Clients requests only data that they need

GraphQL server such as Apollo or Graphene can be employed to wrap a REST API and make requests to the REST API.

In order to wrap REST APIs the server uses modules:

- **Type definitions** → Data model
- **Resolvers** → Functions to perform mapping
Inspiration

PyOSLC:

- SDK based on Python
- Expose databases as OSLC SP
- Mapping of data sources into RDF

PyOSLC perform its operations by using modules:

- RDF Mapping
- OSLC resources described as Python objects
Inspiration

**OpenAPI Specification (OA Spec)** → API description format for REST APIs.

**Swagger** → Set of open-source tools for designing, building and documenting REST APIs.

**Swagger Codegen** → Generates server stub, client libraries from an OpenAPI spec, create/use custom generators.

[Diagram showing data flow from OpenAPI spec to REST and GraphQL APIs, with Swagger Codegen and PyOSLC nodes.]
What about creating a similar approach using OSLC?
Solution

Coatl → OSLC wrapper for REST APIs

Based on **Swagger Codegen / Python**

Necessary:
- REST API
- Swagger Codegen libraries
- Open API document
How Coatl works?

Coatl works by documenting REST API data into OA spec with extensions.

**Input:** OpenAPI spec file

**Output:** OSLC API

Specify:
- OSLC API endpoints
- Coatl extensions:
  - REST API → OSLC API data source
  - JSON keys of REST → RDF properties
How to use Coatl?

You need to perform two operations:

1. **Create OpenAPI spec**
2. **Generate OSLC API** by using Swagger Codegen
1.- OpenAPI document

Specify an OpenAPI document:

OSLC API endpoints

Via vendor extensions:

- REST API URLs → OSLC API data source
- JSON keys → RDF properties

```yaml
paths:
  /catalog:
    get:
      tags:
        - "OSLC Services"
      summary: "Get the OSLC Service Provider Catalog"
      operationId: "get_catalog"
      responses:
        200:
          description: "successful operation"
      x-OSLC-ServiceProviderCatalog:
        dataSource: "https://api.github.com/orgs/koneksys"
        description: "description"
        title: "name"
        domain:
          - "http://open-services.net/ns/rm#"
```
OpenAPI Coatl Extensions for OSLC

The following extensions are employed to map REST API URL/data into OSLC Structure:

**Operations**
- x-Jazz-Root Services
- x-OSLC-Service Provider Catalog
- x-OSLC-ServiceProvider
- x-OSLC-Services
  - Query Capability
  - UI dialog
- x-OSLC-UI-Preview

**Endpoints**
- x-OSLC-QueryCapability-endpoint
- x-OSLC-Publisher-endpoint
- x-OSLC-ResourceShape-endpoint
- x-OSLC-SelectDialog-endpoint
- x-Jazz-Configuration-Catalog

**Models**
- x-OSLC-ResourceProperties
- x-RDF-Vocabulary

**Configuration Management**
- x-OSLC-Components
- x-OSLC-Baseline
- x-OSLC-Stream
- x-OSLC-Configuration-Selection
- x-OSLC-Configuration-Versioned-Resource
Example: REST API endpoints

The data provided by Github API endpoints will be employed to create a Github OSLC API resources.

GET: https://api.github.com/orgs/koneksys

```
{
  "login": "koneksys",
  "id": 16889825,
  ...
  "repos_url": "https://api.github.com/orgs/koneksys/repos",
  "description": "Software services based on state-of-the-art non-proprietary",
  "name": "Koneksys"
}
```

GET: https://api.github.com/orgs/koneksys/repos

```
[
  {
    "name": "KFE",
    "full_name": "koneksys/KFE",
    "trees_url": "https://api.github.com/repos/koneksys/KFE/git/trees{/sha}"
  }
  ...
],
[
  {
    "name": "Web-Based-SysML-Block-Diagram",
    "full_name": "koneksys/Web-Based-SysML-Block-Diagram",
    "trees_url": "https://api.github.com/repos/koneksys/Web-Based-SysML-Bloc"
  },
  ...
]
```
OSLC API data source

REST resource describing github account.
Github API URL:
https://api.github.com/orgs/koneksys

OpenAPI spec with Coatl extensions for OSLC Service Provider Catalog creation.

OSLC Service Provider Catalog resource describing github account.
**JSON keys → RDF**

**JSON Query Language (JAQL)** is employed to parse JSON data into RDF properties.

**JAQL** → Handle stored data in JSON documents.

JAQL provides tools to perform other operations such as:

- Get keys from JSON nested data
- Multiple Selection
- Functions expressions

Coatl contains libraries for managing JAQL queries.
OpenAPI spec with Coatl extensions for OSLC Service Provider Catalog creation.

GitHub API URL:
https://api.github.com/orgs/koneksys

OSLC Service Provider Catalog resource describing github account.
OpenAPI spec with Coatl extensions for OSLC Service Provider creation.

REST resource describing github repository.
Github API URL:
https://api.github.com/orgs/koneksys/repo

JSON keys → RDF

OSLC Service Provider resource describing github repository.
REST resource describing github files.
Github API URL:

```
https://api.github.com/repos/koneksys/KFE/git/trees/master?recursive=1
```

OpenAPI spec with Coatl extensions for OSLC Query Capability service creation.

```
x-OSLC-Services:
  - queryCapability: "github_files"
  dataSource: '[[:].join(``, [to_string(trees url),`(`/master?recursive=1`)])]
query: "tree[::].id: replace_all_substrings(path, `/`, `~`) title: replace_all_substrings(path, `.py`, `~`)"
```

OSLC artifact describing a single github file.

```
<https://github-oslc-api/oslc/provider/KFE/resources/github/100000000/FEA_model-FEA Model 1.py
ns3:instanceShape <https://github-oslc-api/oslc/provider/KFE/resources/github/100000000/FEA_model-FEA Model 1.py
ns3:serviceProvider <https://github-oslc-api/oslc/provider/KFE/resources/github/100000000/FEA_model-FEA Model 1.py
ns2:identifier FEA_model-FEA Model 1.py
ns2:title FEA_model-FEA Model 1.py
ns1:size 5028
```
2.- Create OSLC wrapper

OSLC API will be created through Swagger Codegen and Coatl library.
The information provided by REST API is exposed through OSLC Resources.

Github API example:
Coatl extensions for integration with IBM ELM™
IBM ELM™

- Provides integrated end-to-end solution across all engineering data.
- Optimizes collaboration and communication across all stakeholders.

**OSLC APIs integration**

- Enables extensibility through open standards such as OSLC
- Optimize communication, collaboration and verification of artifacts from different vendors
- Improve decision-making by linking artifacts from different vendors to Jazz.
Support for Oauth v1.0a for registering external OSLC API as Jazz friend application which enables:

- To associate providers (e.g. Github repo/OSLC SP) to a IBM ELM project
- To use delegated dialogs from external OSLC API within IBM ELM apps
- To define contributing components from external OSLC API to a global component in IBM GCM
Next steps
Next steps (OSLC)

- Improve RDF mapping definition in OpenAPI Spec
- OSLC Configuration Management compatible with IBM ELM
- Tracked Resource Set
Next steps (Code)

- Deploy new OSLC APIs in cloud / container
- Include in Coatl library more automated tests
- Test Coatl library with more REST APIs
- Include Coatl library support for OpenAPI 3.0
Conclusion

- Coatl → OSLC wrapper for REST APIs
- Automatic code generation
- Coatl significantly reduce the implementation effort for developers creating OSLC APIs
- Developers only need to understand OSLC concepts, they don’t need to program them
Questions about Coatl?
See a Coatl demo?

Contact us

axel.reichwein@koneksys.com
juan.quintanar@koneksys.com