PyOSLC

Latest Improvements

Frank Patz-Brockmann (CONTACT Software GmbH)
Mario Jiménez Carrasco (Koneksys)

Nov 2nd, 2021

Frank Patz-Brockmann

Director R&D at Contact Software

Leading the team responsible for Contact's Elements technology-platform, and the PLM and IoT-solutions based on it. He looks back on more than 25 years of experience in enterprise and engineering software, helping to define Contact's product strategy and advising key customers in automotive and other industries.

CONTACT Software

Mario Jiménez Carrasco

Koneksys

Senior Developer at Koneksys

Main contributor to the open-source PyOSLC project.

Developed several data integration solutions to connect domain-specific applications in Application Lifecycle Management (ALM) and Product Lifecycle Management (PLM).

email: mario.carrasco@koneksys.com

The PyOSLC project

PyOSLC

- **What is it?**
  - PyOSLC is an SDK to implement OSLC APIs in Python.
  - In scope: server-side SDK & client-side SDK, priority server-side for now
  - CONTACT Software funds OSS development by Koneksys, to support OSLC with product CONTACT Elements (PLM & more): https://www.contact-software.com/en/
- **Where can I find PyOSLC?**

  https://github.com/cslab/pyoslc

What happened so far ...

- PyOSLC has been developed during the last two years but with some intermittence in which we have reviewed and tested the integrated features, up to working demonstrators.
- Spent a lot of time on testing interoperability & reverse engineering not-so-standard implementations of OLSC-supporting tools (Core and Vocabularies are very robust -- Auth & Service advertising not so much)
- Early this year, we decided to change course a bit and rework APIs to make it more framework-like
- Meanwhile: independent team at Contact working on adopting PyOSLC into our standard product

CONTACT **Elements**

The world's best platform to design and operate smart products

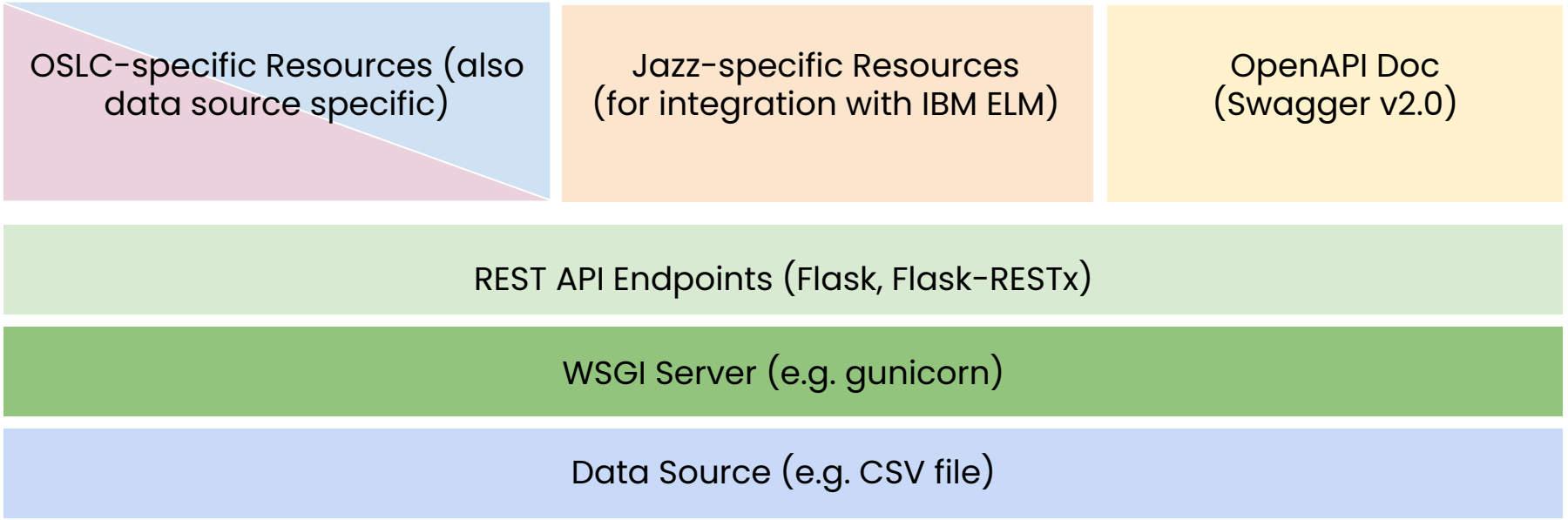
Product Mgmt.										
Innovations **IN**	**Project Mgmt.**								**Industriali-zation**	
Portfolio Mgmt. **PO**	Process Mgmt. **PM**		**Quality & Compli-ance**		**Product Architect.**		**Product Elaboration**	Tool Design **TD**	xBOM **XB**	**Digital Product Application**
Technical Sales **TS**	Project Costing **JC**	Agile Methods **AM**	Best Practice Templates **PT**	Change Records & Audit Trail **CR**	Module & Platform Mgmt. **MP**	Variants **VA**	CAx Document Mgmt. **CX**	Simulation Mgmt. **SI**	Ramp-up **RU**	Device Mgmt. **DE**
Product Costing **DC**	Work Breakdown **W**	Resource Planning **RP**	Risk Mgmt. **RM**	Material Compliance **MC**	Require-ments Engineering **RE**	Part Classi-fication & Reuse **PA**	Materials Mgmt. **MM**	Prototyping & Test **TE**	Closed Loop Engineering **CL**	Customer Services **CS**
Product Informa-tion Mgmt. **PI**	Scheduling **SC**	Documen-tation **DM**	Controlling & Maturity Mgmt. **CO**	Quality Issue Mgmt. **QI**	Product Structure **PS**	DMU **DU**	Bill of Materials **BM**	Engineering Change Mgmt. **EM**	Digital Twin **DT**	Monitoring & Analytics **MA**
Core Services	Configura-tion Mgmt. **CM**	Document Mgmt. **DM**	Enterprise Collabora-tion **EC**	Workflows **WF**	Tasks **TA**	Activity Stream **AS**	3D Visualiza-tion **3D**	Metrics & Reporting **MR**	Universal Classifica-tion **UC**	
Techno-logies	Component Architecture **CA**	Security **SE**	Roles & Access Rights **RA**	Digital Signatures **DS**	Webservices & Interope-rability **WI**	Multi-Cloud & on-Premise Operations **OP**	Global Data Mgmt. **GD**	UX for Web & Mobile **UX**	Application Development Framework **AD**	

Contact's requirements

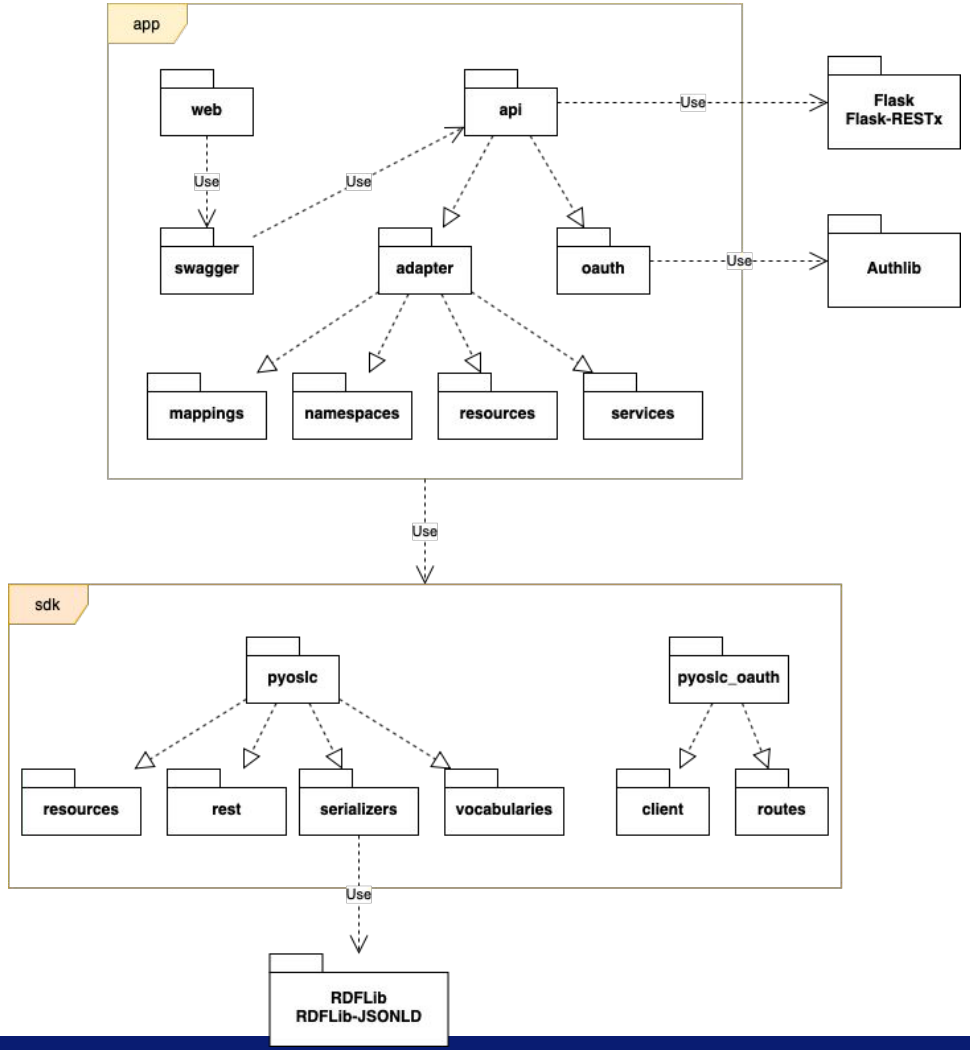
- Contact Elements (CE) is heavily based on Python → therefore PyOSLC
- OSLC server-side is more or less REST/LDP conventions plus vocabularies
- CE uses WSGI, Python's standard for web application. WSGI supports composability, therefore server-side PyOSLC should be a **pluggable WSGI component**, that can run stand-alone or in combination with a bigger application like CE
- PyOSLC should **support the dynamic nature of Python**: very little boilerplate code, supporting introspection
- **De-coupled binding** between PyOSLC and stored objects in the hosting application (or whatever backend), to support different use-cases

Previous API Design

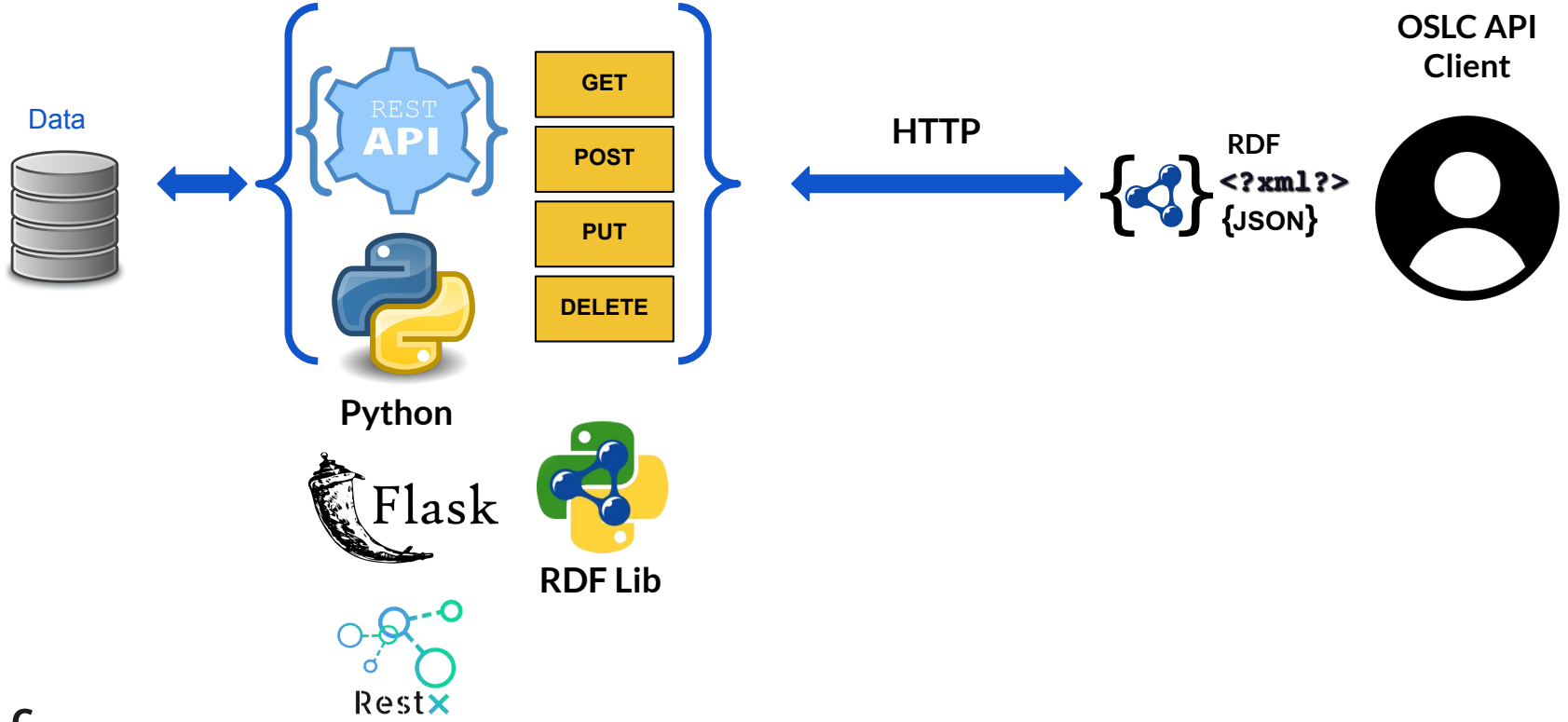
PyOSLC Building Blocks / Current



Packages


app
web
api
Use
Flask
Flask-RESTx
Use
Use
swagger
adapter
oauth
Use
Authlib
mappings
namespaces
resources
services
Use
sdk
pyoslc
pyoslc_oauth
resources
rest
serializers
vocabularies
client
routes
Use
RDFLib
RDFLib-JSONLD

Architecture

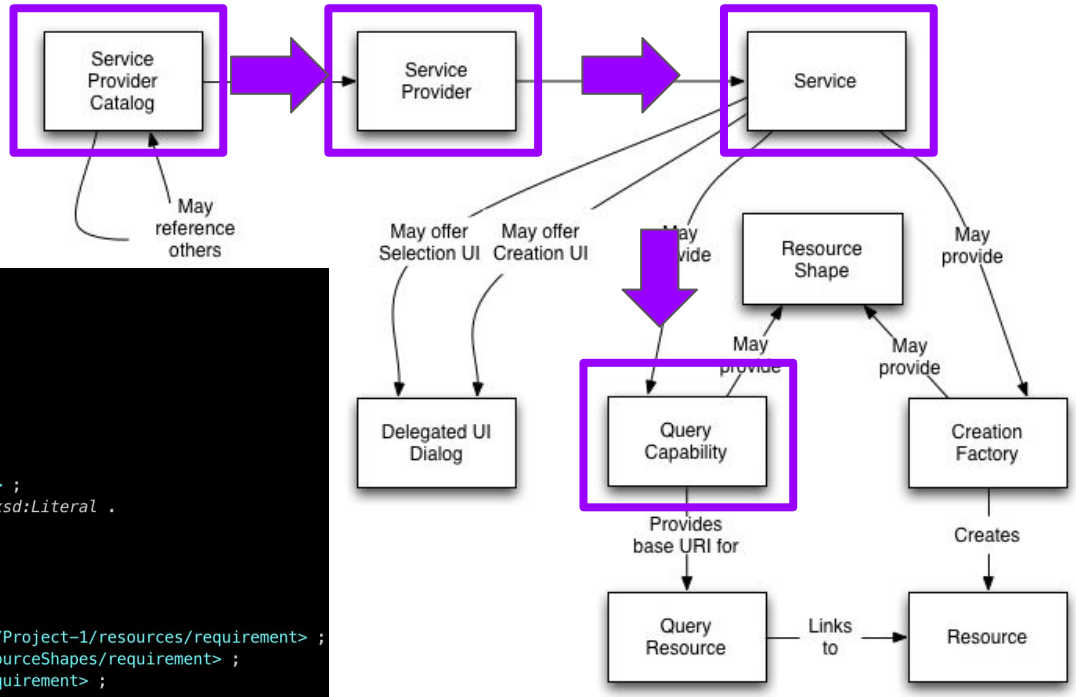


OSLC Resources

Curl for GET on SPC

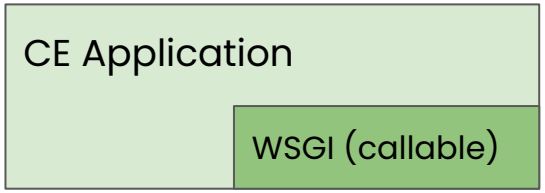
```
➜ curl -X GET http://baseurl/oslc/services/catalog \
-H "Content-Type: text/turtle" \
-H "Accept: text/turtle"
```

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix oslc: <http://open-services.net/ns/core#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://baseurl/oslc/services/catalog> a oslc:ServiceProviderCatalog ;
    oslc:domain <http://open-services.net/ns/rm#> ;
    oslc:serviceProvider <http://baseurl/oslc/services/catalog/Project-1> ;
    dcterms:title "Contact Software Platform Service Provider Catalog"^^xsd:Literal .

<http://baseurl/oslc/services/catalog/Project-1> a oslc:ServiceProvider ;
    oslc:service [ a oslc:Service ;
            oslc:domain <http://open-services.net/ns/rm#> ;
            oslc:queryCapability [ a oslc:QueryCapability ;
                    oslc:queryBase <http://baseurl/oslc/services/project/Project-1/resources/requirement> ;
                    oslc:resourceShape <http://baseurl/oslc/services/resourceShapes/requirement> ;
                    oslc:resourceType <http://open-services.net/ns/rm#Requirement> ;
                    dcterms:title "Query Capability" ] ;
            dcterms:title "Service for Resources"^^xsd:Literal ] ;
    dcterms:identifier "Project-1" .
```



New API Design

CONTACT Elements (CE) Application



CE Application interacts with CIM database

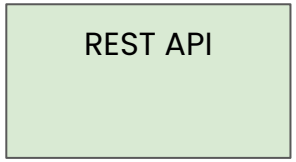
WSGI callable allows to run the CE application within a web server as an API

PyOSLC SDK

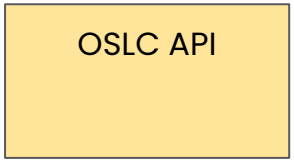
Old	New	
pyoslc-sdk	pyoslc	pyoslc is the base package, including facilities for mapping to and from RDF, defining domains and vocabularies etc.
	pyoslc-server	pyoslc-server is the server SDK, utilizing Werkzeug and the base package to provide building blocks for creating OSLC servers
	pyoslc-client	More packages could be added to the pyoslc sdk in the future

OSLC

REST App / OSLC App

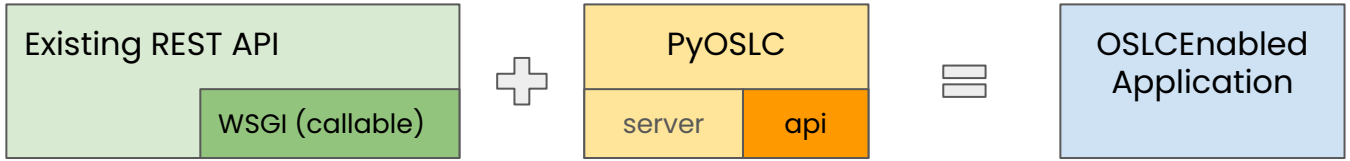


1. Stateless
2. Supports JSON and XML
3. Resource methods (GET, POST, etc)
4. Uniform Interface
5. Resource Identifier
6. etc.



1. Uniform self-descriptive REST API
2. Linked data model for standard domains (ALM, PLM)
3. Exchange data in RDF (JSON-LD and others)
4. Self discoverability
5. Traceability
6. etc.

OSLC Enabled Application



Adding PyOSLC to an existing REST API, generates an OSLC Enabled Application

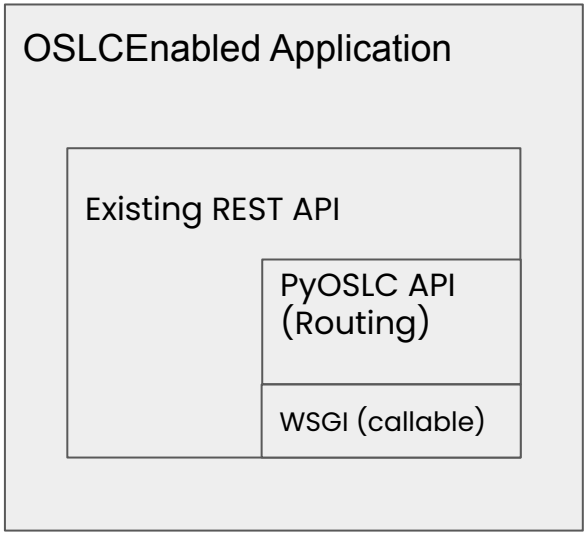
By using PyOSLC it is possible to deploy the application within a WSGI web server using its own WSGI callable or the PyOSLC server implementation.

Demo: Initializing OSLC Application

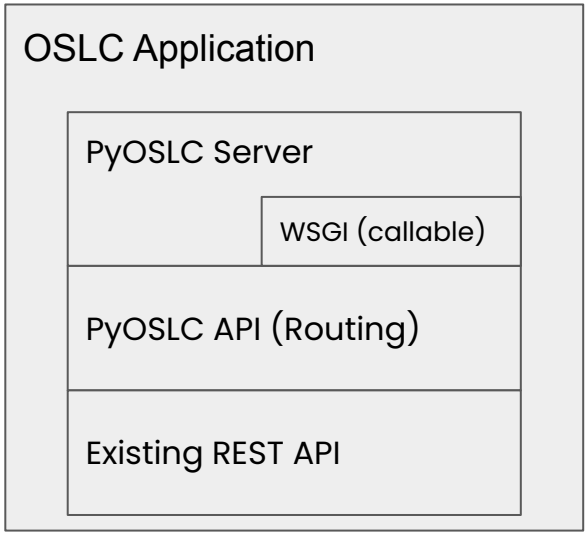
- Importing the OSLCAPP class
- Instantiating the OSLC API
- Initializing the adapters

```python
from pyoslc_server import OSLCAPP

from apposlc.adapter import RequirementAdapter
from apposlc.adapter import TestCaseAdapter
from apposlc.adapter import REQ_TO_RDF


class OSLCEnabled:

    def __init__(self):
        self.app = OSLCAPP(prefix='/oslc')
        self.__init_adapters()

    def __init_adapters(self):...

    def __call__(self, environ, start_response):...
```

OSLC Deployment options



pyoslc, implements the endpoints and routings to process the requests, running within an existing web server



pyoslc-server, implements callable to execute the API directly in a web server

Demo: Running the OSLC App

- Running the application by instantiating the OSLC Enabled application

```python
from apposlc.oslc_enabled import OSLCEnabled


def create_oslc_app():
    app = OSLCEnabled()
    return app
```

```python
from werkzeug import run_simple

from apposlc import create_oslc_app

app = create_oslc_app()

if __name__ == "__main__":
    run_simple('127.0.0.1',
               5000,
               app,
               use_debugger=True,
               use_reloader=True)
```

OSLC

What is WSGI

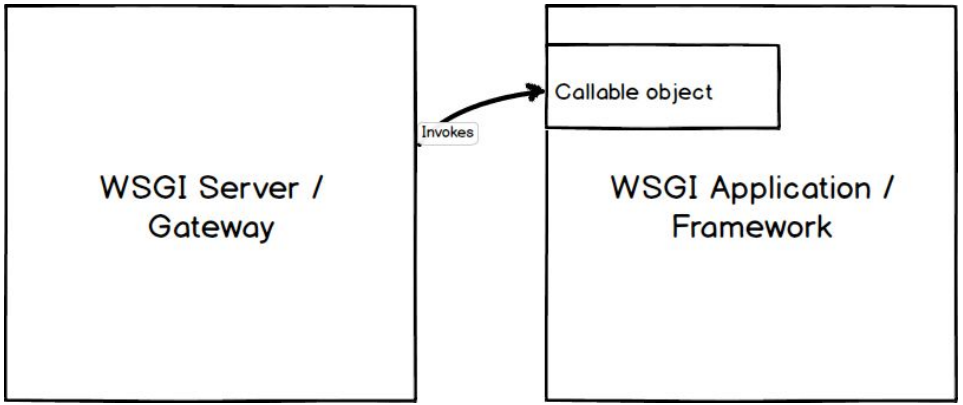
- Interface specification for connecting web apps and web servers (PEP 3333)
- Define rules for the web app and the web server to interact.
- Requires a callable function
- Receive Environment variables and callback function

An example:

A Java similar API

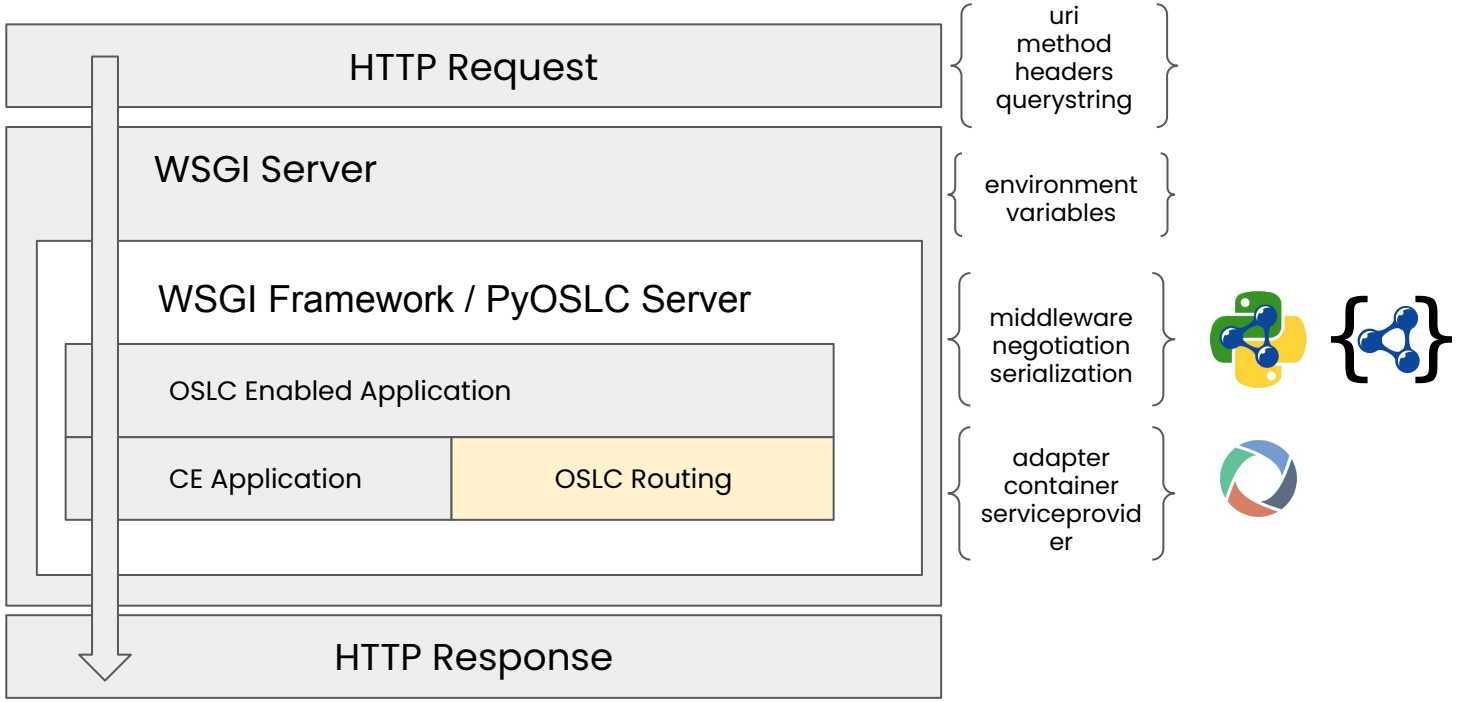
Java Servlet API

WSGI Compliance

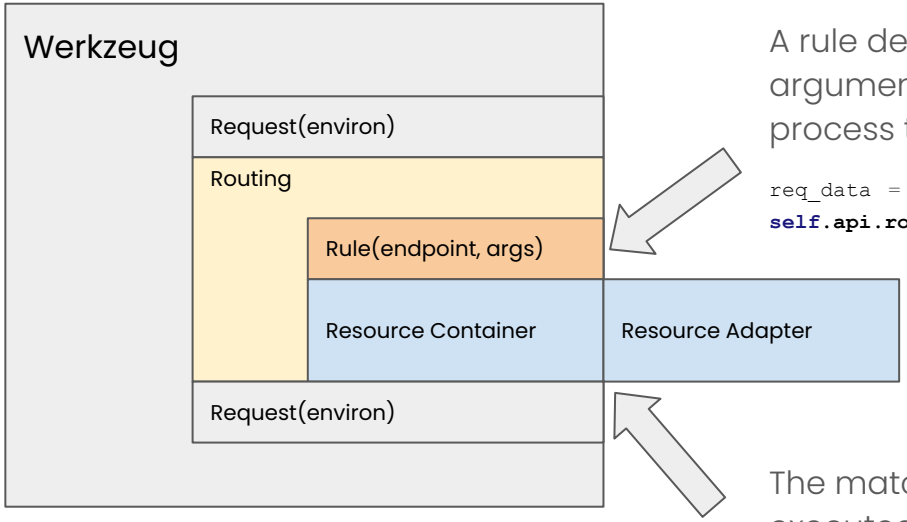


- WSGI de facto should be used.
- PyOSLC could create or become into a Framework (to run upon to REST APIs)
- PyOSLC could or could not be an application, it could be an extension of an application instead.

PyOSLC Request Flow



Routing with Werkzeug



A rule defines the endpoint “path” with the arguments and the method or class that should process the request

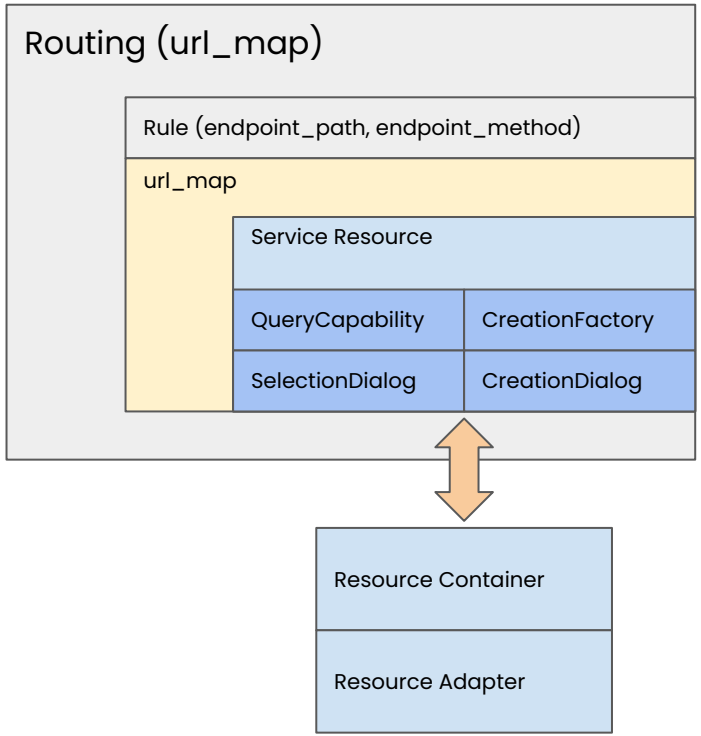
```
req_data = RequirementsAdapter(REQSTORE)
self.api.route("/requirements/<int:identifier>", req_data)
```

The matched endpoint and method will be executed in the resource container or resource adapter class or instance.

```
class RequirementsAdapter (ServiceResourceAdapter) :
    ...

    def get item( self, identifier):
        for item in self. items:
            if item.identifier == identifier:
                return item
```

OSLC

Service Resource Adapter



Configuring the Router using a Service Resource Adapter class in which the method should be configured to be implemented in the Containers or Adapters

The methods of the Service Resource could be bind to the methods of the Container or Adapter.

```
class RequirementsAdapter(ServiceResourceAdapter):
    ...

    def get_item(self, identifier):
        for item in self._items:
            if item.identifier == identifier:
                return item
```

OSLC

Benefits of Service Resource Adapter

- Adapters will create the OSLC endpoints
- Adapters will manage the header negotiation for query strings
- Adapters will manage the serialization from/to rdf and python objects
- Developers just need to implement the methods required for theirs OSLC API
- Methods will talk with datasource using python objects
- Developers do not need to take care about RDF just python objects

New Components

Domain-Specific Resources

- Requirement
- Product
- Document
- TestCase

Resources are defined in data classes and will be Python objects

```
@dataclass
class Requirement:
      identifier: str = ""
      title: str = ""
      description: str = ""

REQSTORE = [
    Requirement("1", "Provide WSGI implementation", "..."),
    Requirement("2", "Capability to add resources", "..."),
    # and so on ...
]
```

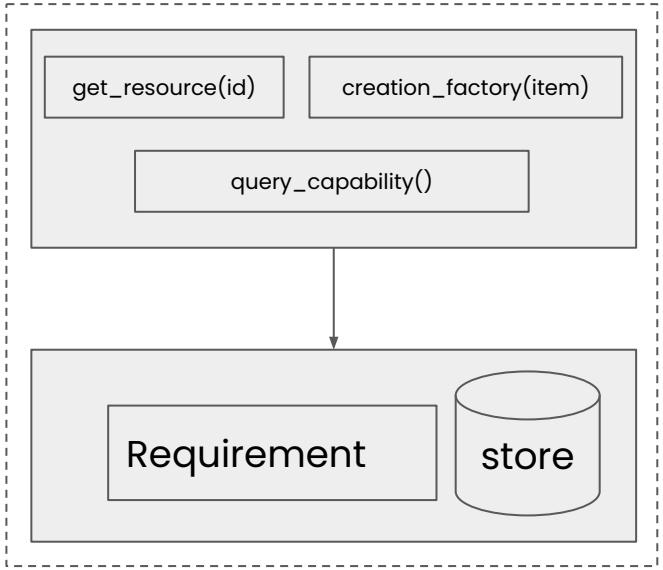
Resource classes should be implemented in the CE Application side.

Demo: Defining a DS Resource

- Domain-Specific Resource defines the attributes that will be used
- The class could have more attributes
- The class could have methods if the developer need them for using within the adapter.
- The class will be used only for the adapter

```python
class Requirement:

    identifier = ""
    title = ""
    description = ""

    def __init__(self, identifier, title, description):
        self.identifier = identifier
        self.title = title
        self.description = description

    ...
```

Resource Adapter



Adapter is the implementation of methods on the CE or Application side to manipulate the resources.

```
class RequirementsAdapter(ServiceResourceAdapter):
    ...

    def get_resource(self, identifier):
        for item in self._items:
            if item.identifier == identifier:
                return item
```

"To PyOSLC, data items are completely opaque, they are simple received from and passed back into the application adapter."

Demo: Defining an Adapter

- Importing and extending ServiceResourceAdapter class
- Configuring the OSLC attributes of the class
- Implementing the methods that will be managed by the adapter
- The class could contain other methods

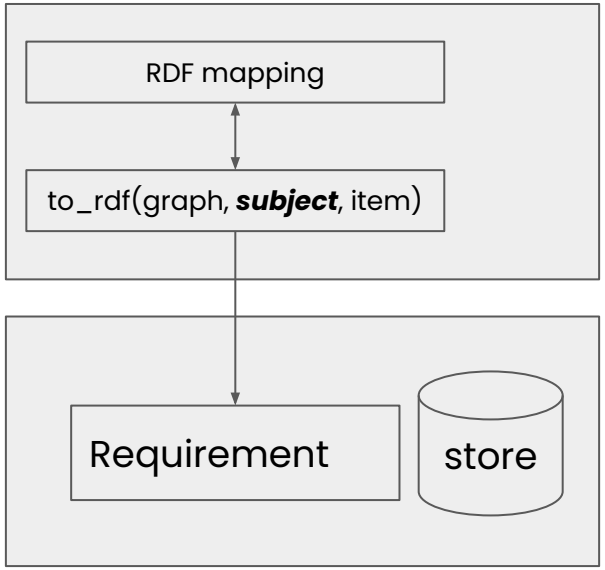
```python
from pyoslc_server.specification import ServiceResourceAdapter
from .resource import REQSTORE

REQ_TO_RDF = {...}


class RequirementAdapter(ServiceResourceAdapter):

    domain = OSLC_RM
    types = [OSLC_RM.Requirement]
    items = REQSTORE
    mapping = REQ_TO_RDF

    def __init__(self, **kwargs):...

    def set(self, data_items):...

    def query_capability(self, paging=False, page_size=50, page_no=1,
                         prefix=None, where=None, select=None,
                         *args, **kwargs):...

    def creation_factory(self):...

    def get_resource(self, resource_id):...
```

Adding Adapters to OSLC

- The adapter should be instantiated to initialize the attributes
- The mapping should be defined
- The adapter is added to the OSLC API
- This shows one, but more than one adapter could be added to the OSLC API.

```
class OSLCEnabled:

    def __init__(self):...

    def __init_adapters(self):

        requirement_adapter = RequirementAdapter(
            identifier='adapter',
            title='Requirement Adapter',
            description='Requirement Adapter for OSLC'
            mapping=REQ_TO_RDF

        )

        self.app.api.add_adapter(
            adapter=requirement_adapter
        )
```

Mapping RDF for Adapter



The RDF mapping should define the attribute mapping from Python object attributes to RDF attributes

```
REQ_TO_RDF = {
    "identifier": DCTERMS.description,
    "title": DCTERMS.title,
    "description": DCTERMS.identifier,
}

class RequirementsAdapter(ServiceResourceAdapter):
        ...
    def to_rdf(self, graph, subject, item):
        graph.add((subject, RDF.type, RM.Requirement))
        for field, property in REQ_TO_RDF.items():
            graph.add((subject, property, Literal(getattr(item, field))))
        return graph
```

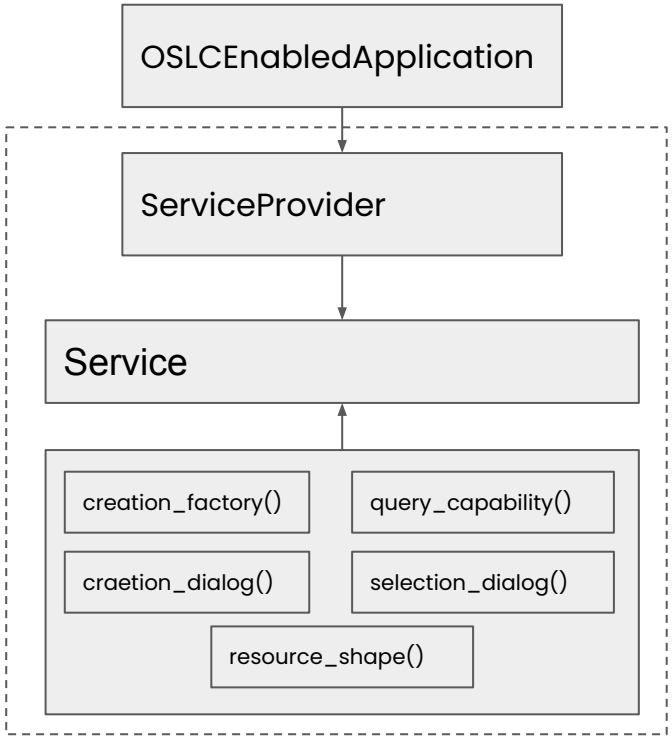
"Note that subject is computed by PyOSLC and given to to_rdf as an argument."

Demo: Mapping

- Importing standard vocabularies
- Importing domain-specific vocabularies
- Dictionary with python attributes and OSLC terms

```python
from pyoslc.vocabularies.rm import OSLC_RM
from pyoslc.vocabularies.qm import OSLC_QM
from rdflib import DCTERMS

REQ_TO_RDF = {
    "identifier": DCTERMS.identifier,
    "title": DCTERMS.title,
    "description": DCTERMS.description,
}
```

Capabilities for Adapters



By adding an Adapter the OSLC Application will create a Service Provider with the corresponding set of services.

The methods that will process the information will be implemented on the adapter

The OSCL Application is only responsible of the generation of the endpoints for SPC, SP, QC

Demo: Methods of Adapter

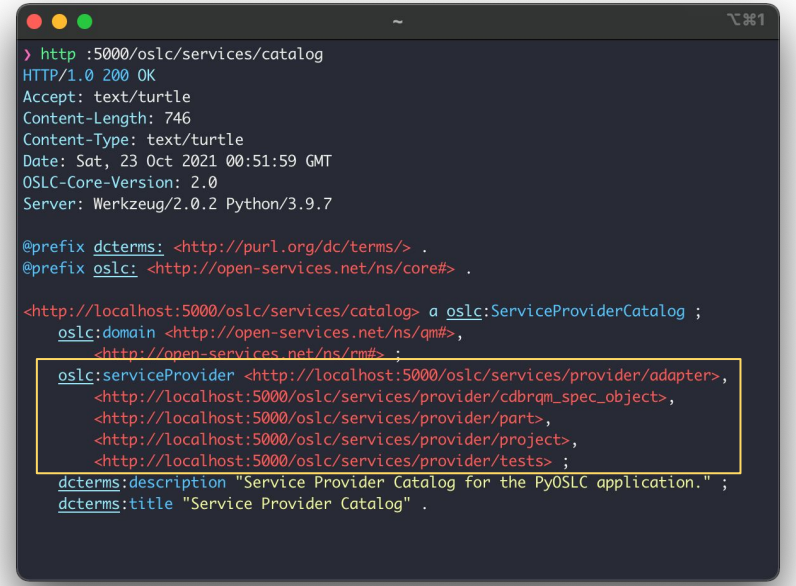
- Adapter should implement the methods for OSLC endpoints like QC, CF
- Adapter will manage python objects
- Adapter does not need to implement all OSLC methods or endpoints

```
def query_capability(self, paging=False, page_size=50, page_no=1,
                     prefix=None, where=None, select=None,
                     *args, **kwargs):...

def creation_factory(self):...

def get_resource(self, resource_id):...
```

The Result

Getting the Catalog

- The catalog will show each adapter as a ServiceProvider
- The identifier of the adapter is part of the URL for the SP



Getting the Service Provider

- The Service Provider will show the services enabled in the adapter cla

```
def query_capability(self, paging=False, page_size=50, page_no=1,
                     prefix=None, where=None, select=None,
                     *args, **kwargs):...

def creation_factory(self):...

def get_resource(self, resource_id):...
```

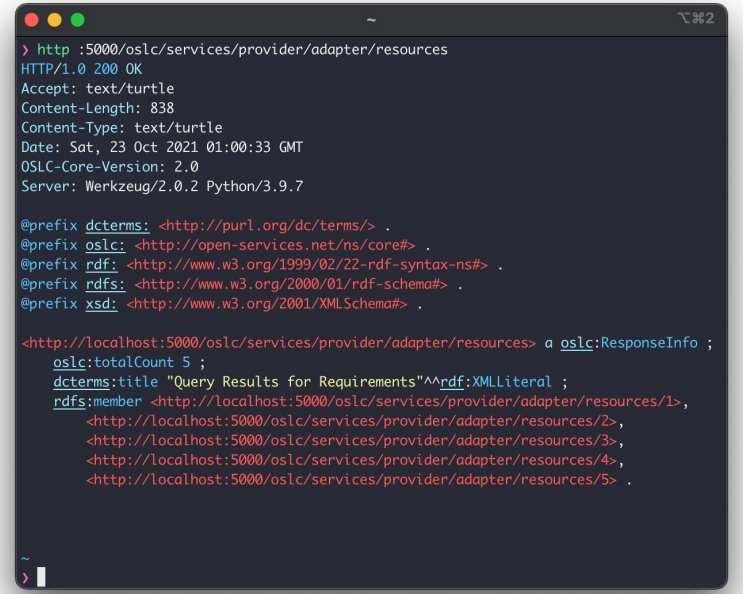
```
❯ http :5000/oslc/services/provider/adapter
HTTP/1.0 200 OK
Accept: text/turtle
Content-Length: 1582
Content-Type: text/turtle
Date: Sat, 23 Oct 2021 00:56:51 GMT
OSLC-Core-Version: 2.0
Server: Werkzeug/2.0.2 Python/3.9.7

@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix oslc: <http://open-services.net/ns/core#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://localhost:5000/oslc/services/provider/adapter> a oslc:ServiceProvider ;
    oslc:details <http://localhost:5000/oslc/services/provider/adapter> ;
    oslc:service [ a oslc:Service ;
            oslc:creationFactory [ a oslc:CreationFactory ;
                    oslc:creation <http://localhost:5000/oslc/services/provider/adapter/resources> ;
                    oslc:label "Creation Factory"^^xsd:string ;
                    oslc:resourceShape <http://localhost:5000/oslc/services/catalog/resourceShapes/requirement> ;
                    oslc:resourceType <http://open-services.net/ns/rm#Requirement> ;
                    dcterms:title "Creation Factory" ] ;
            oslc:domain <http://open-services.net/ns/rm#> ;
            oslc:queryCapability [ a oslc:QueryCapability ;
                    oslc:label "Query Capability"^^xsd:string ;
                    oslc:queryBase <http://localhost:5000/oslc/services/provider/adapter/resources> ;
                    oslc:resourceShape <http://localhost:5000/oslc/services/resourceShapes/requirement> ;
                    oslc:resourceType <http://open-services.net/ns/rm#Requirement> ;
                    dcterms:title "Query Capability" ] ] ;
    dcterms:description "Requirement Adapter for OSLC" ;
    dcterms:identifier "adapter"^^xsd:string ;
    dcterms:title "Requirement Adapter"^^rdf:XMLLiteral .
```

OSLC

Query Capability

- The query capability and the other methods of the service, will be called by the OSLC API

- OSLC API just process the input / output to serialize and deserialize from python objects to RDF and vice versa



Conclusion

PyOSLC New API

- PyOSLC supports the development of OSLC APIs for any domain-specific resources
- Developer only needs to have a basic understanding of OSLC concepts
- Developer thereby does not need to modify nor manage any code related to the translation of Python objects into RDF
- PyOSLC now supports two options to deploy an OSLC API

Thanks