Generating Python based service components for web frameworks

網媒所 - 陳耀新

2020/8/14

Outline

- Introduction
- Related work
- PythonPoet
- Service Code Generation
- Conclusion
- Future work

Introduction

- web service
- service oriented architecture
- extend service
    - craw from open source
- analyzer python code
- PythonPoet
- generate controller, invoker and WSDL

Related work

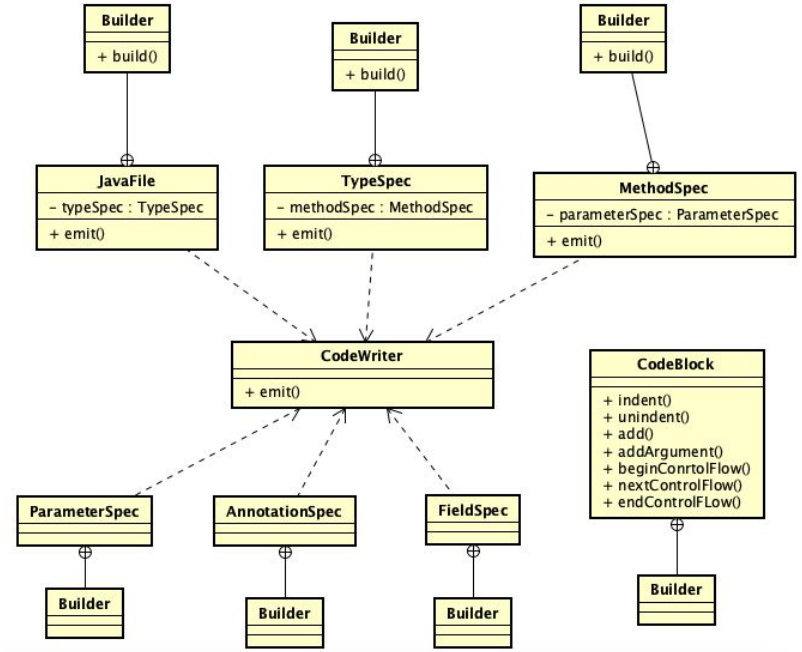
- JavaPoet
- Web service component
- Flask
- Python AST
- Service Code Generation for JAVA

JavaPoet

- third party library
- class structure
- example
- pros and cons

JavaPoet - class structure

- JavaFile
- TypeSpec
- MethodSpec
- CodeBlock
- CodeWriter
- ...



Example

API rules（input）

```
public class Test{
    public static void main(String[] args){
        MethodSpec main = MethodSpec.methodBuilder( name: "main")
                .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
                .returns(void.class)
                .addParameter(String[].class,  name: "args")
                .addStatement("$T.out.println($S)", System.class, "Hello, JavaPoet!")
                .build();

        TypeSpec helloWorld = TypeSpec.classBuilder( name: "HelloWorld")
                .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
                .addMethod(main)
                .build();

        JavaFile javaFile = JavaFile.builder( packageName: "com.example.helloworld", helloWorld)
                .build();
```

output

```
public final class HelloWorld {
  public static void main(String[] args) { System.out.println("Hello, JavaPoet!"); }
}
```

JavaPoet pro and cons

- generate java code easily by using different SPECs
- provide many APIs for user to customize their code
- the generated code is not checked(or compiled) by system

Web service component

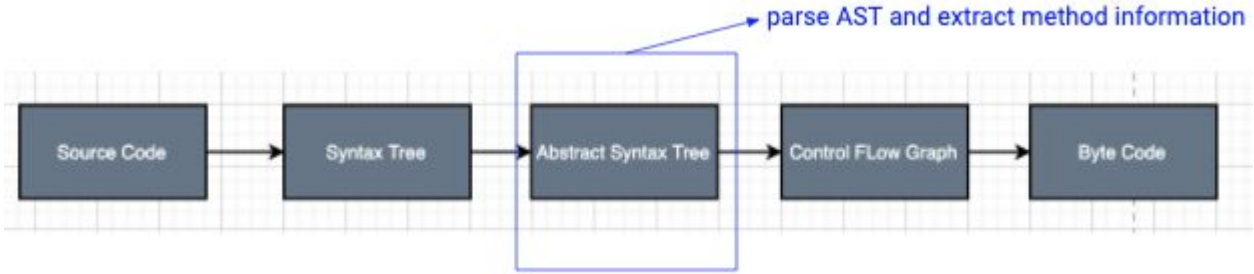
- treat web service as class, message as attribute and service logic as method

Flask

- microframework
- costumize extension
- ...

Python AST

- Python interpret process



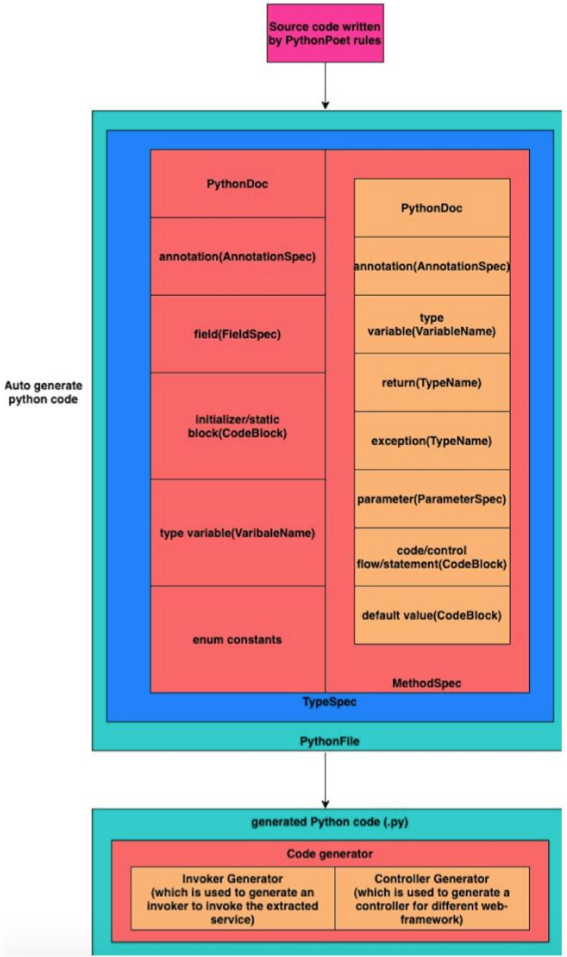
- A module helps Python applications to process trees of the Python abstract syntax grammar

Service Code Generation for JAVA

- Controller
  - routing
- Invoker
  - invoke service
- API server
  - interact with BPEL engine
- WSDL
  - a service description

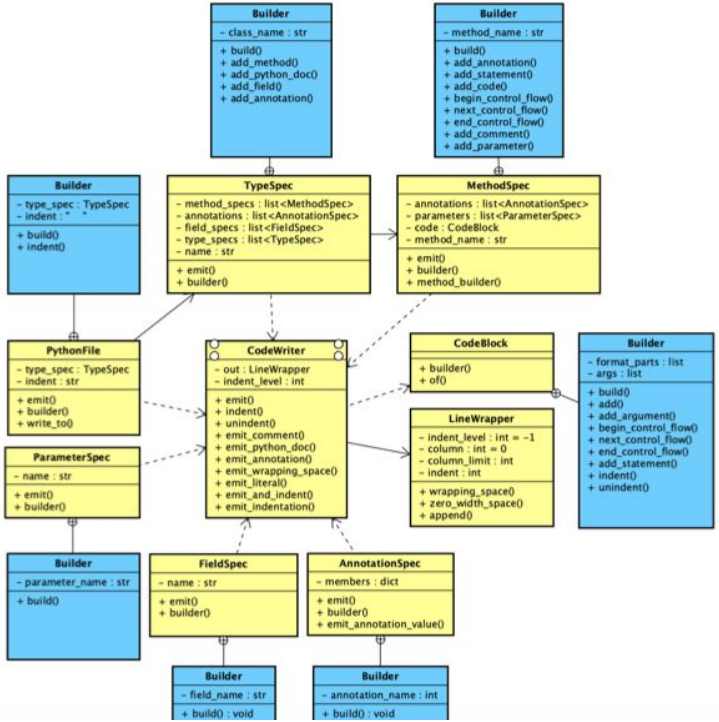
PythonPoet

- treat JavaPoet as a template
- API rules and example
- ...



PythonPoet class structure

- PythonFile
- TypeSpec
- MethodSpec
- AnnotationSpec(used by controller)
- ParameterSpec
- CodeBlock
- CodeWriter
- ...



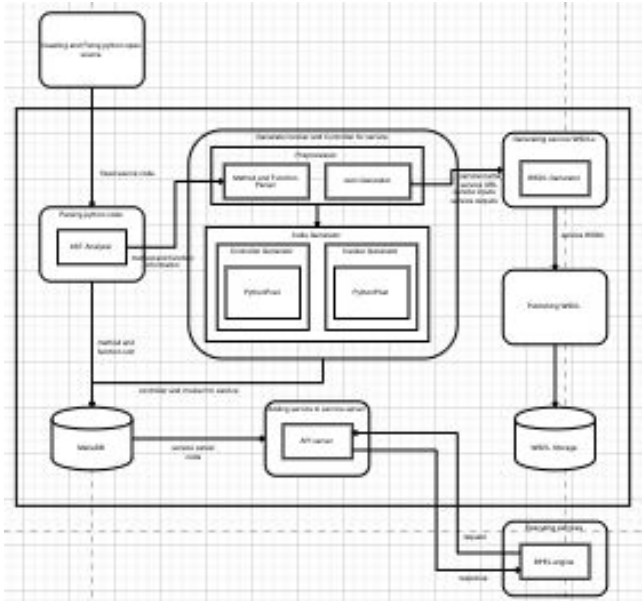
PythonPoet API rules and example

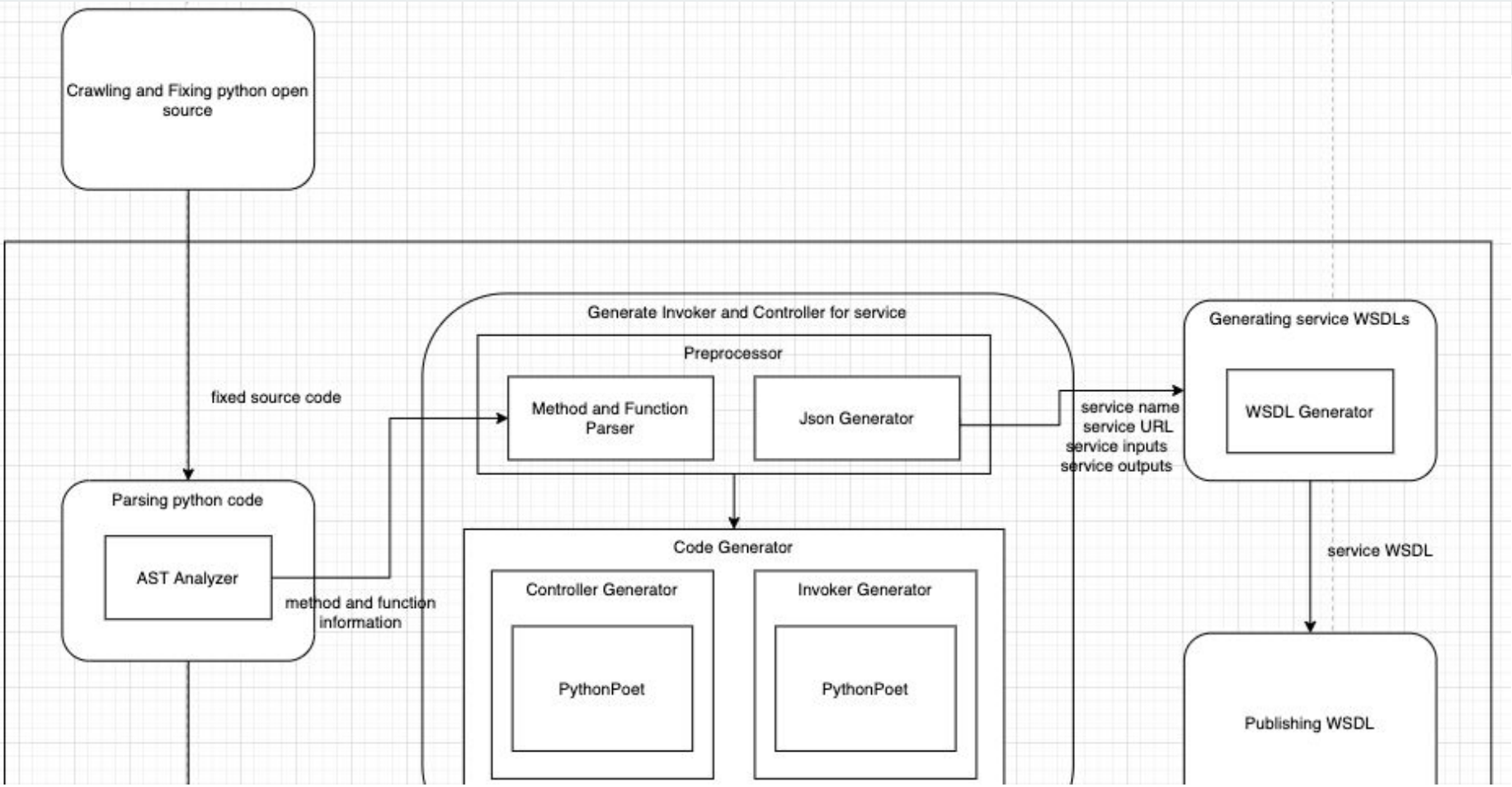
```python
def add_output_code(self, builder, service_unit):
    # add code: return ServiceUtil.serialize(serviceResultObject)
    builder.add_code_segment("return $L.", 'ServiceUtil')
    builder.add_code_segment("$L(", Constant.SERVICE_UTIL_SERIALIZE_NAME)
    builder.add_code_segment("$L)", Constant.OUTPUT_JSON_NAME)
    return builder
```

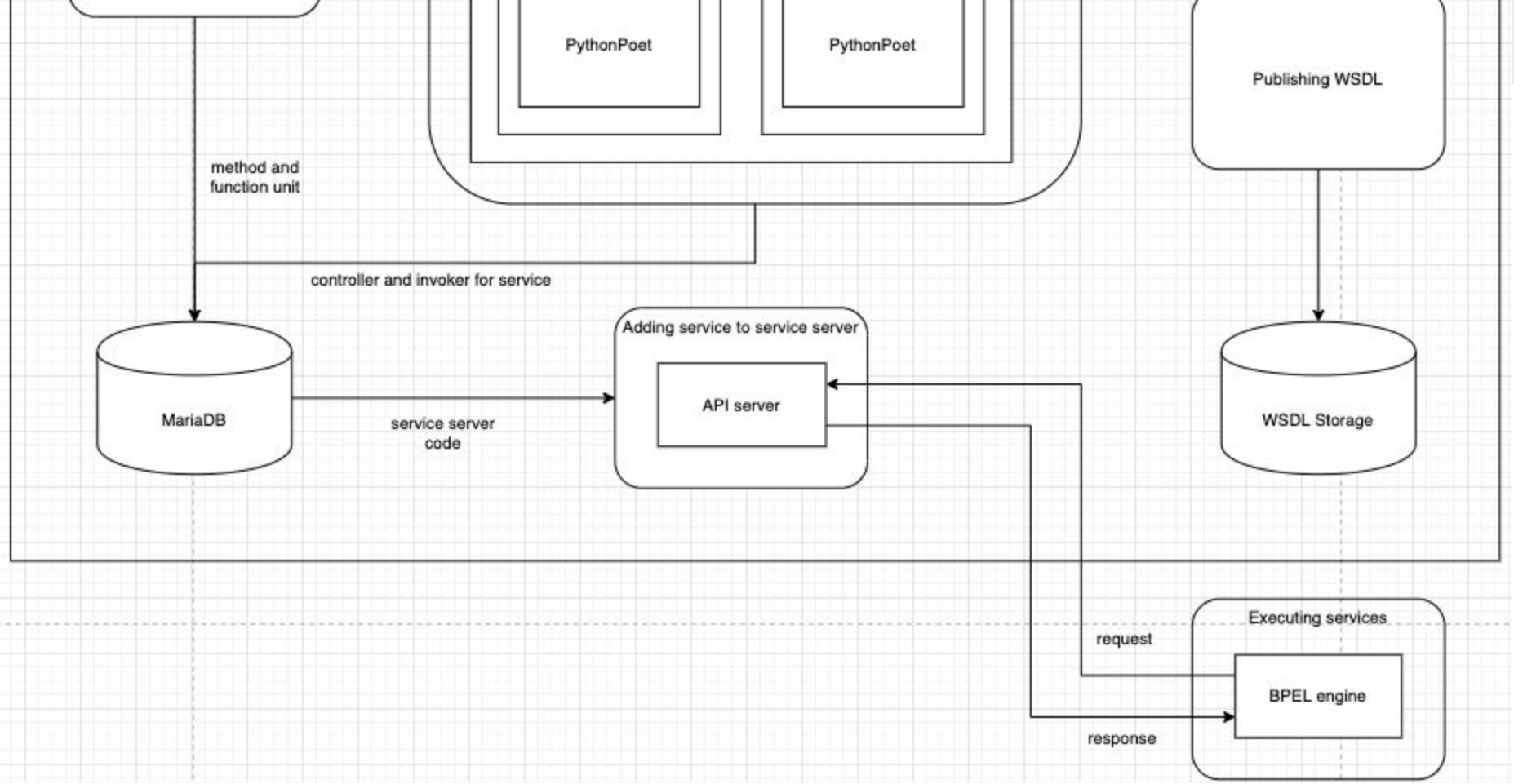
```python
@staticmethod
def monitor__FactorMonitor(self, serviceUtilObject):
    self__Instance = serviceUtilObject.deserializeInstance(self)
    executable = serviceUtilObject.getDeclaredMethod(self__Instance, 'monitor')
    serviceResultObject = serviceUtilObject.invokeService(executable)
    return ServiceUtil.serialize(serviceResultObject)
```

Service Code Generation

- Analyze Python code
- API server
- Generate controller
- Generate invoker
- Generate WSDL

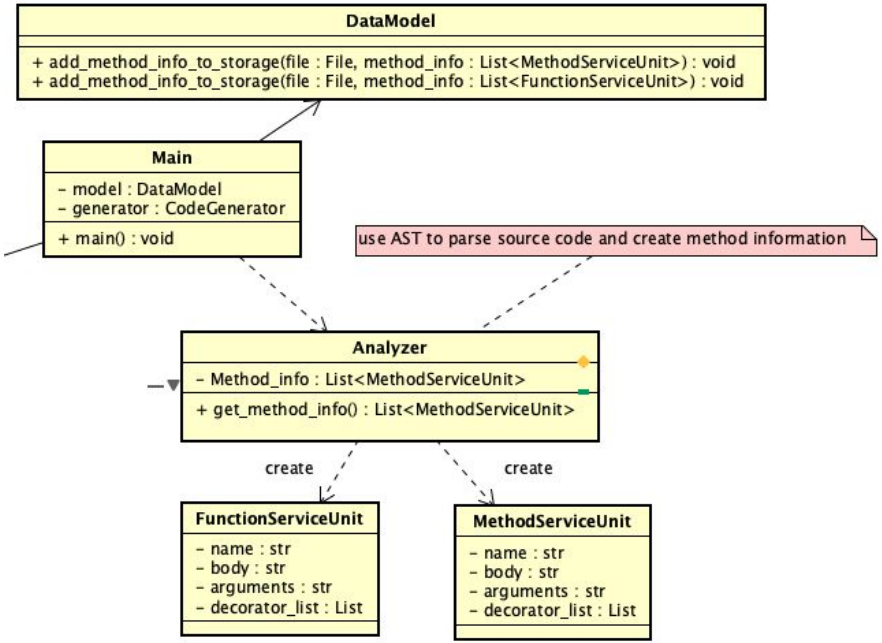



Crawling and Fixing python open source
fixed source code
Parsing python code
AST Analyzer
method and function information
Generate Invoker and Controller for service
Preprocessor
Method and Function Parser
Json Generator
Code Generator
Controller Generator
PythonPoet
Invoker Generator
PythonPoet
service name
service URL
service inputs
service outputs
Generating service WSDLs
WSDL Generator
service WSDL
Publishing WSDL


PythonPoet
PythonPoet
Publishing WSDL
method and
function unit
controller and invoker for service
Adding service to service server
API server
MariaDB
service server
code
WSDL Storage
Executing services
request
BPEL engine
response

Analyze Python code

- AST module use visitor pattern to visit all nodes in a program
- extract method information from AST
- save information to data model



Method information in python AST

Method Information	Descriptions	Use or not in service component
name	a raw string of the function name.	O
args	a arguments node	O
body	the list of nodes inside the function.	X
decorator_list	the list of decorators to be applied, stored outermost first	O
returns	the return annotation	X
type_comment	a string containing the PEP 484 type comment of the function (added in Python 3.8)	X

Access AST

```
-- ASDL's 5 builtin types are:
-- identifier, int, string, object, constant

module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment)
         | AsyncFunctionDef(identifier name, arguments args,
                            stmt* body, expr* decorator_list, expr? returns,
                            string? type_comment)

         | ClassDef(identifier name,
           expr* bases,
           keyword* keywords,
           stmt* body,
           expr* decorator_list)
         | Return(expr? value)
```

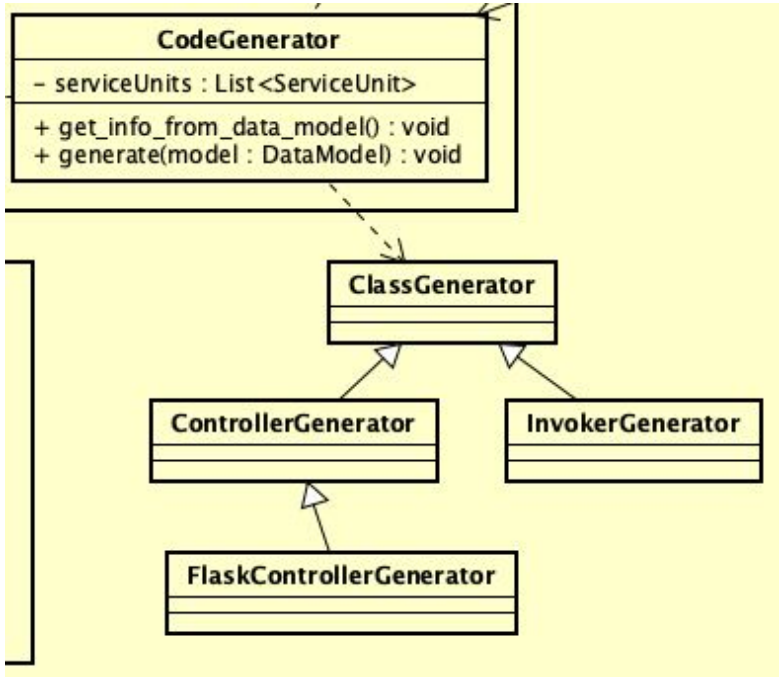
```python
def visit_ClassDef(self, node):
    methods = list()
    class_name = node.name
    for n in node.body:  # visit all child nodes in this class
        is_static = False
        if type(n) == ast.ClassDef:  # if n has child class node
            self.visit_ClassDef(n)  # process nested class
            self.nested_class_info.update({node.name: n.name})  # store inner classes
        if hasattr(n, 'name') and hasattr(n, 'args'):  # if n is a method or function
            # find static method
            if n.decorator_list:
                for func_node in n.decorator_list:
                    if type(func_node) == ast.Name:
                        if func_node.id == "staticmethod":
                            is_static = True
```

API server

- service library
  - serialize
  - deserialize
  - ...
- flask server
  - register blueprint
  - import controller

Generators

- generate controller by ControllerGenerator
- generate invoker by InvokerGenerator



Controller

- flask controller features
  - blueprint
  - routing
  - get parameter

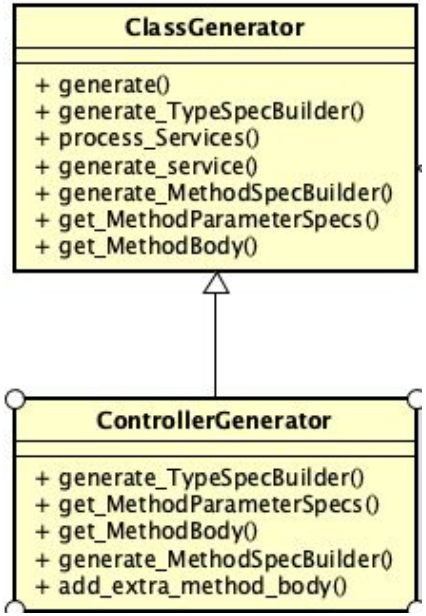
```
from flask import Blueprint, request

index_blueprint = Blueprint('index', __name__)

@index_blueprint.route("/", methods=['GET'])
def index():
    t = request.args.get('t')
    s = request.args.get('s')
    # testing url: http://127.0.0.1:5000/?t=5&s=2
    return str(t) + str(s)
```

Generate Controller

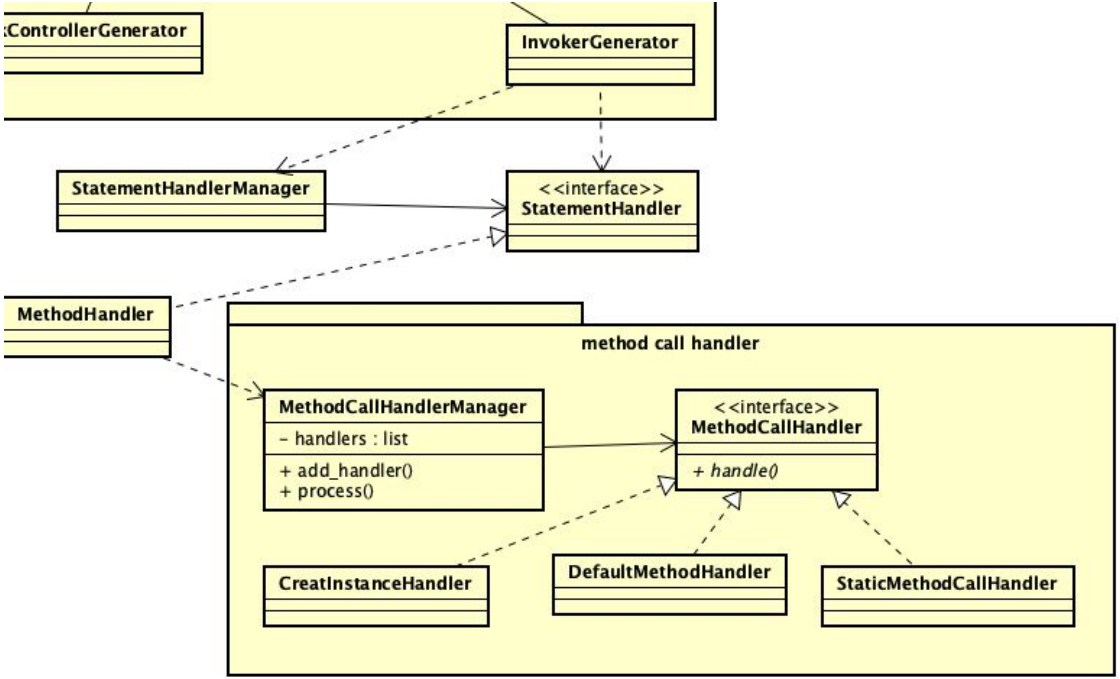
- different from invoker
- add language features for flask controller
    - url binding
    - blueprint



Invoker

- how to invoke method and function
  - invoke method
    - get object instance
    - use getattr() to get the method
    - invoke method and return result
  - invoke function
    - import module and invoke it directly

Generate Invoker



A generated example for invoker and controller

```python
class FactorMonitor:
    def __init__(self, readSec):
        self.readSec = readSec
        self.patients = list()
        self.datas = dict()

    def addPatient(self, patient):
        self.patients.append(patient)
```

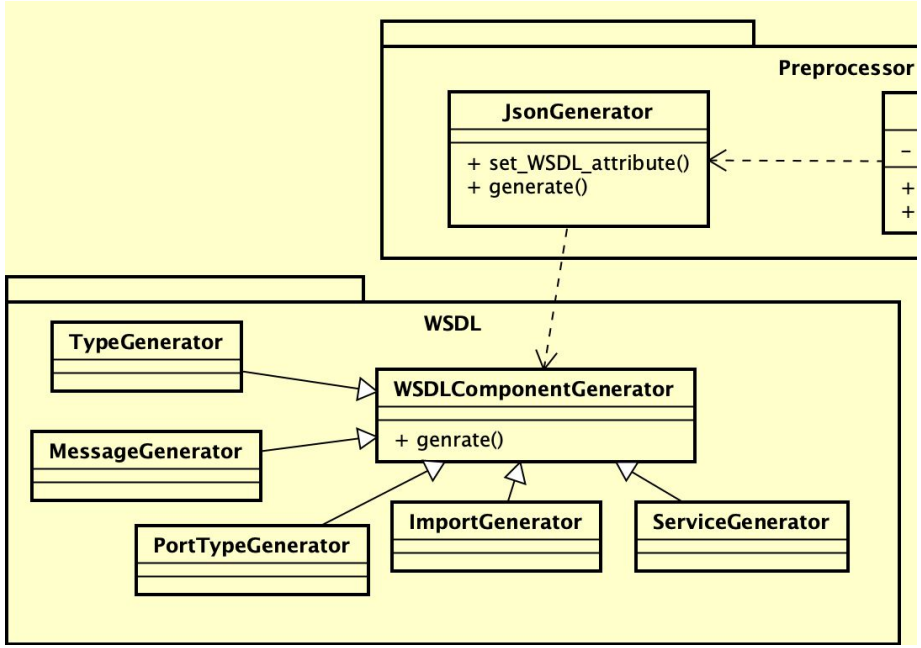
```python
class FactorMonitor_Invoker:
    @staticmethod
    def __init____FactorMonitor(readSec, serviceUtilObject):
        readSec__Instance = serviceUtilObject.deserializeInstance(readSec)
        self__Instance = FactorMonitor(readSec__Instance)
        executable = serviceUtilObject.getDeclaredConstructor(self__Instance)
        serviceResultObject = serviceUtilObject.invokeService(executable, readSec__Instance)
        return ServiceUtil.serialize(serviceResultObject)

    @staticmethod
    def addPatient__FactorMonitor(self, patient, serviceUtilObject):
        self__Instance = serviceUtilObject.deserializeInstance(self)
        patient__Instance = serviceUtilObject.deserializeInstance(patient)
        executable = serviceUtilObject.getDeclaredMethod(self__Instance, 'addPatient')
        serviceResultObject = serviceUtilObject.invokeService(executable, patient__Instance)
        return ServiceUtil.serialize(serviceResultObject)
```

```python
@app.route('/patient_monitoring/FactorMonitor/addPatient')
def addPatient__FactorMonitor():
    self = request.args.get('self')
    patient = request.args.get('patient')
    requestWrapper = ServiceUtil.createFlaskRequestWrapper(request)
    storage = FlaskStorage()
    if not ServiceUtil.hasID(requestWrapper, storage):
        return ServiceUtil.createErrorMessage('There is no ID: ' + ServiceUtil.getProcessID(requestWrapper))
    serviceUtilObject = ServiceUtilObject(requestWrapper, storage, 'patient_monitoring')
    invokerResult = FactorMonitor_Invoker.addPatient__FactorMonitor(self, patient, serviceUtilObject)
    return invokerResult
```

Generate WSDL

- JsonGenerator
  - TypeGenerator
  - PortTypeGenerator
  - MessageGenerator
  - ServiceGenerator
  - ...



Conclusion

- parse open source
- pythonPoet
- genetate controller and invoker
- generate wsdl
- api server

Future work

- integrate service components with mariaDB
- infer the data type for input and output
  - language comparison
  - solution

```
if statement1:
    for i in range(0, 100):
        ...
            ...
                return 1
elif statement2:
    return 1.0
else:
    return "1"
```

- A language is **dynamically-typed** if the type of a variable is checked during **run-time**.
- A language is **statically-typed** if the type of a variable is known at **compile-time** instead of at run-time.
- A **strongly-typed language** is one in which variables are **bound to specific data types, and will result in type errors** if types do not match up as expected in the expression — regardless of when type checking occurs.

C++	**JAVA**	**Python**
weakly typed	strongly typed	strongly typed
static typed	static typed	**dynamically-typed**

Solution

- use JEDI to infer the type for input and output

```
>>> from jedi import Script
>>> source = '''
... import keyword
...
... class C:
...     pass
...
... class D:
...     pass
...
... x = D()
...
... def f():
...     pass
...
... for variable in [keyword, f, C, x]:
...     variable'''
```

```
>>> script = Script(source)
>>> defs = script.infer()
```

Finally, here is what you can get from `type`:

```
>>> defs = [d.type for d in defs]
>>> defs[0]
'module'
>>> defs[1]
'class'
>>> defs[2]
'instance'
>>> defs[3]
'function'
```

Valid values for type are `module`, `class`, `instance`, `function`, `param`, `path`, `keyword`, `property` and `statement`.

Q&A