

SED Final Project

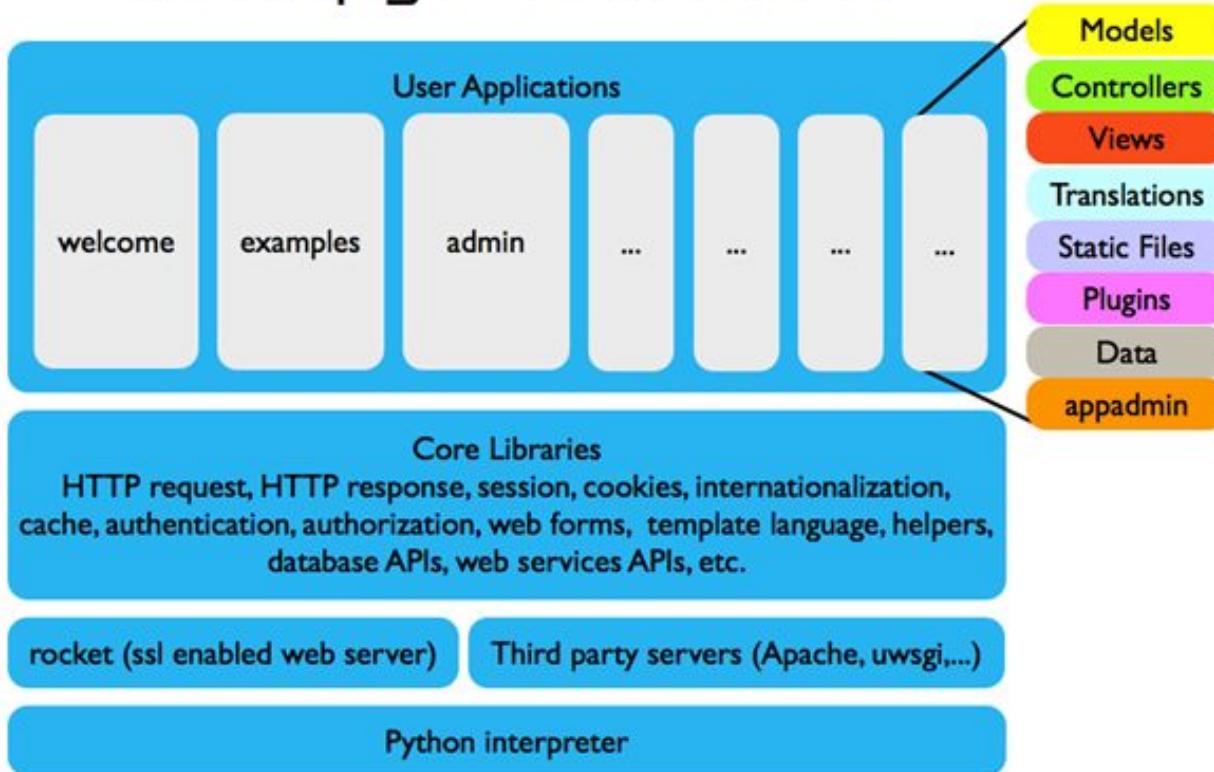
web2py

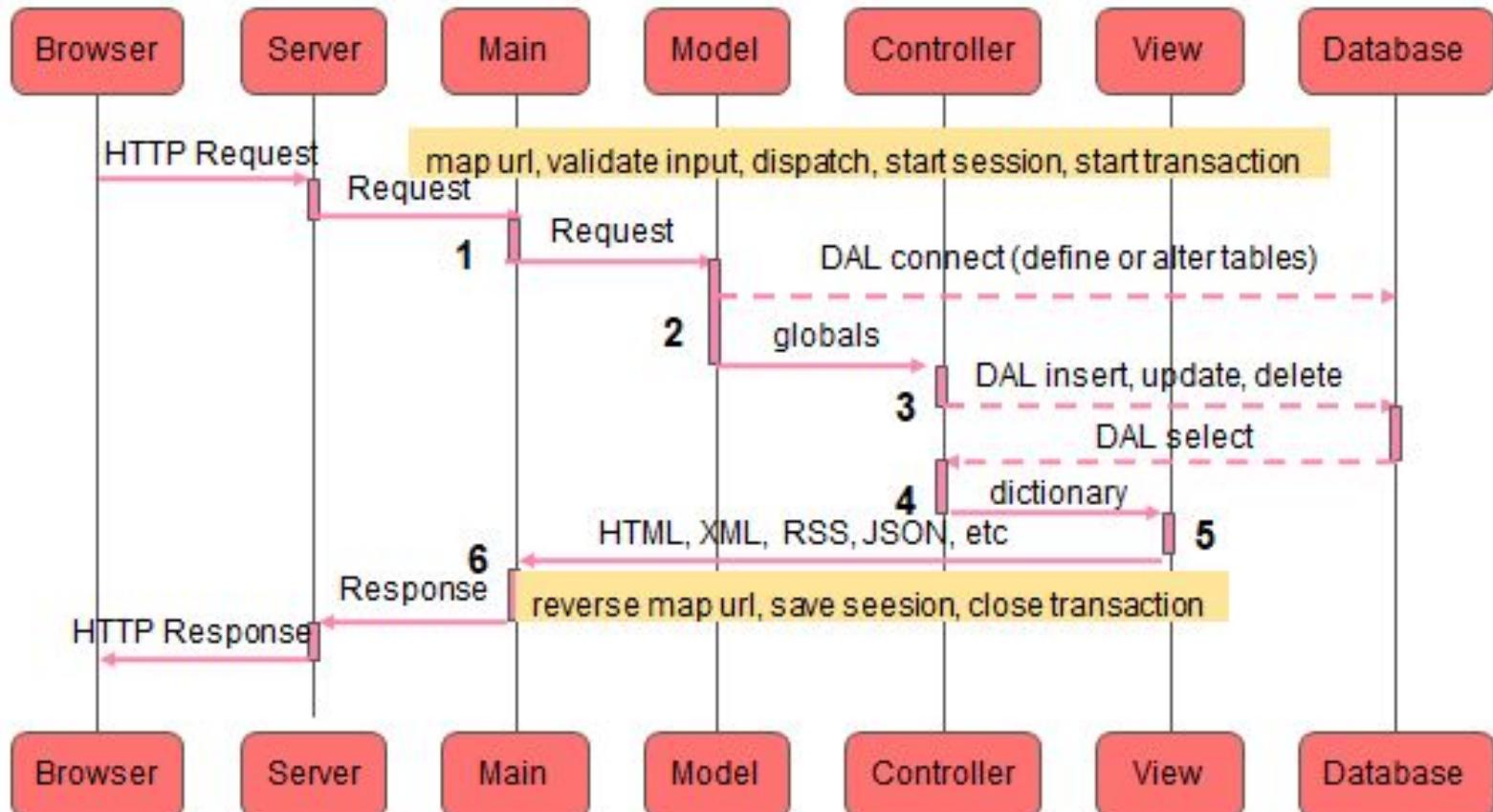
Team2

Outline

- **Briefly introduce web2py**
- Request to Controller
 - Server
 - Rewrite and parsing incoming URLs
 - WSGI hooks
 - Cookies & Session
 - Run Application
- Services
- Views
- Cron & Scheduler
- PyDAL
- Conclusion

web2py Architecture





Server manages HTTP requests

Parse the HTTP request header

Request for
a static file?

True

Read and return (stream) the
requested file

False

Requests to controller

Expose wsgi hooks

Parse and load cookies

Load session or create a new one

Run application

Server manages HTTP requests

Parse the HTTP request header

Request for
a static file?

Read and return (stream) the
requested file

False

True

Request to controller

Expose wsgi hooks

Parse and load cookies

Load session or create a new one

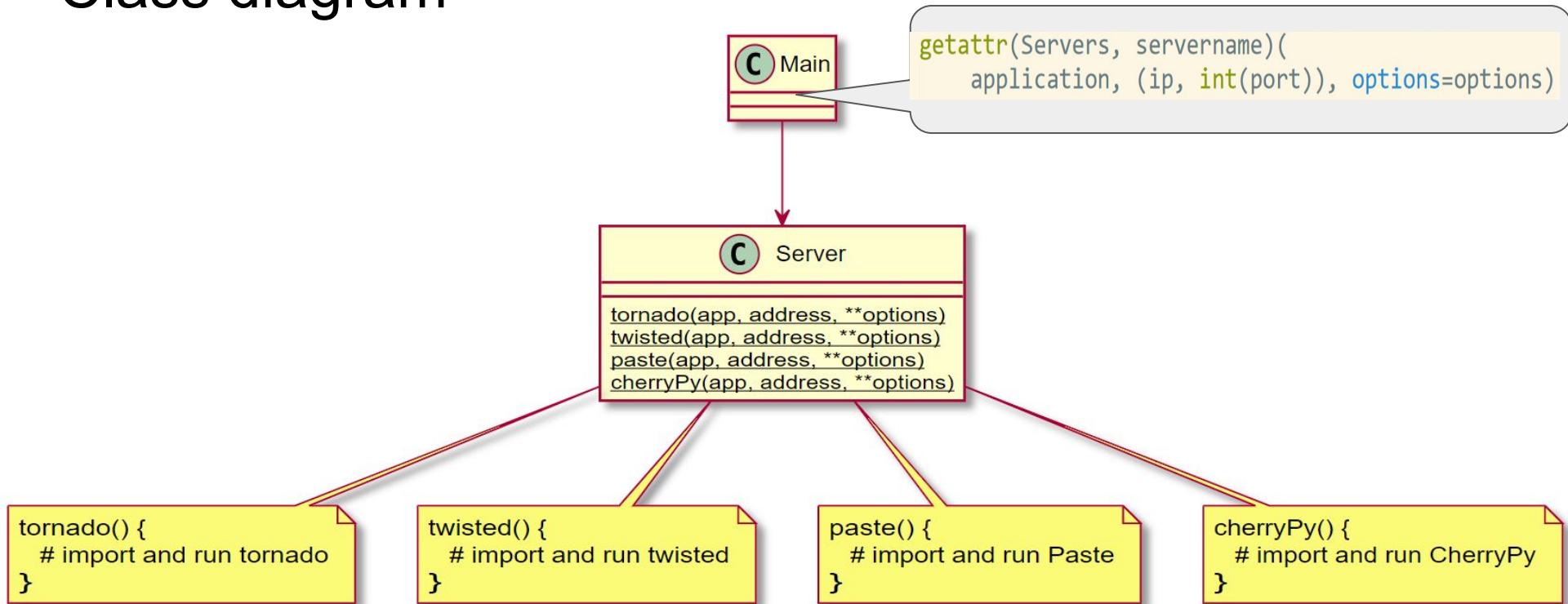
Run application

Server

- web2py includes a built-in server Rocket.
- web2py can also be runned with WSGI compatible servers:
E.g. paste, tornado, twisted, etc.
- Run anyserver.py to start

```
python anyserver.py -s tornado -i 127.0.0.1 -p 8000 -l -P
```

Class diagram



Code

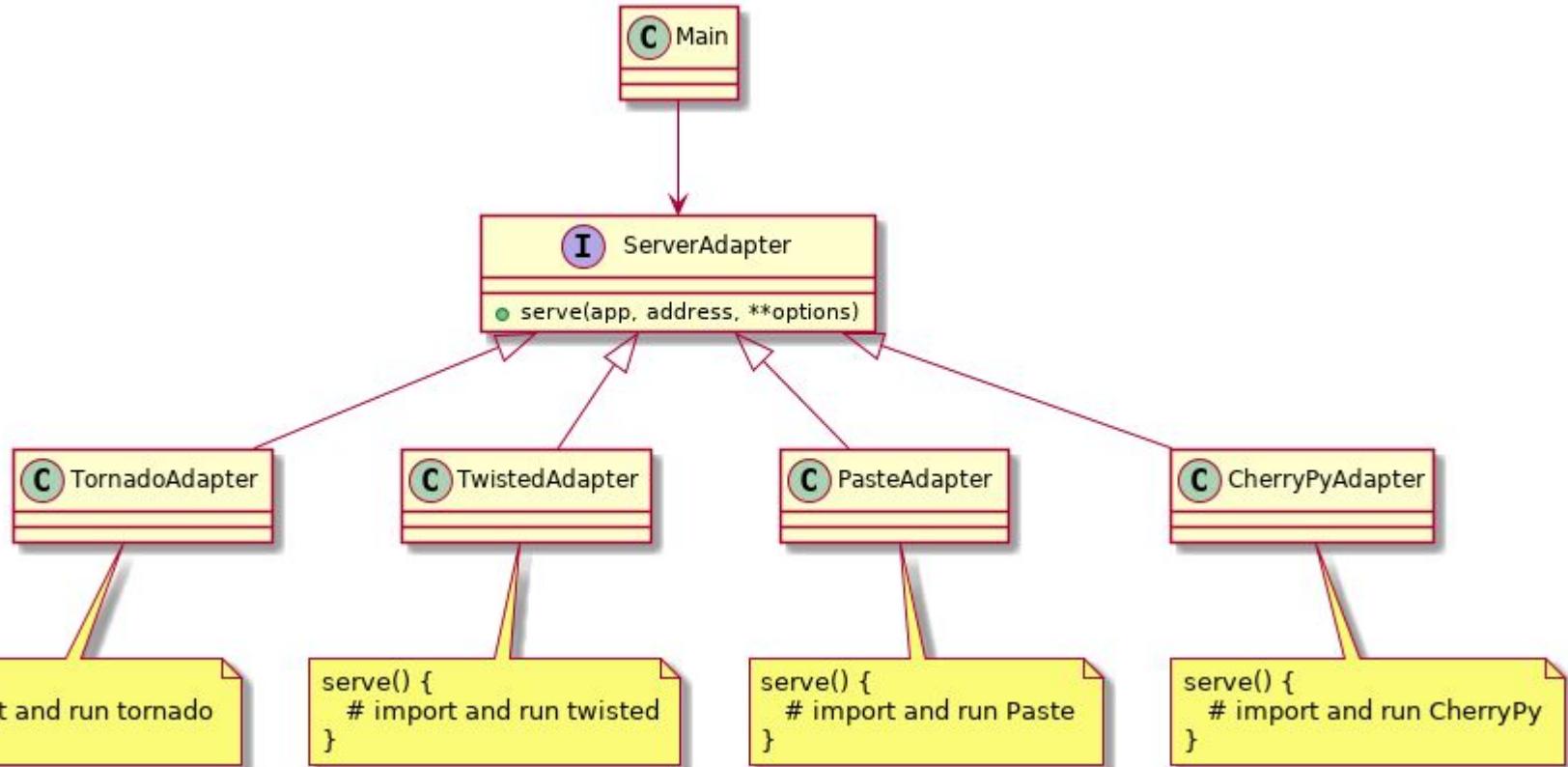
```
def paste(app, address, **options): ← Adapter
    options = {}
    from paste import httpserver
    from paste.translogger import TransLogger
    httpserver.serve(app, host=address[0], port=address[1], **options) ← Adaptee

def cherrypy(app, address, **options): ← Adapter
    ...
def twisted(app, address, **options): ← Adapter
    ...
def tornado(app, address, **options): ← Adapter
    ...
    ...
    ...
```

Interface of adapter:

```
def serve(app, address, **options):
```

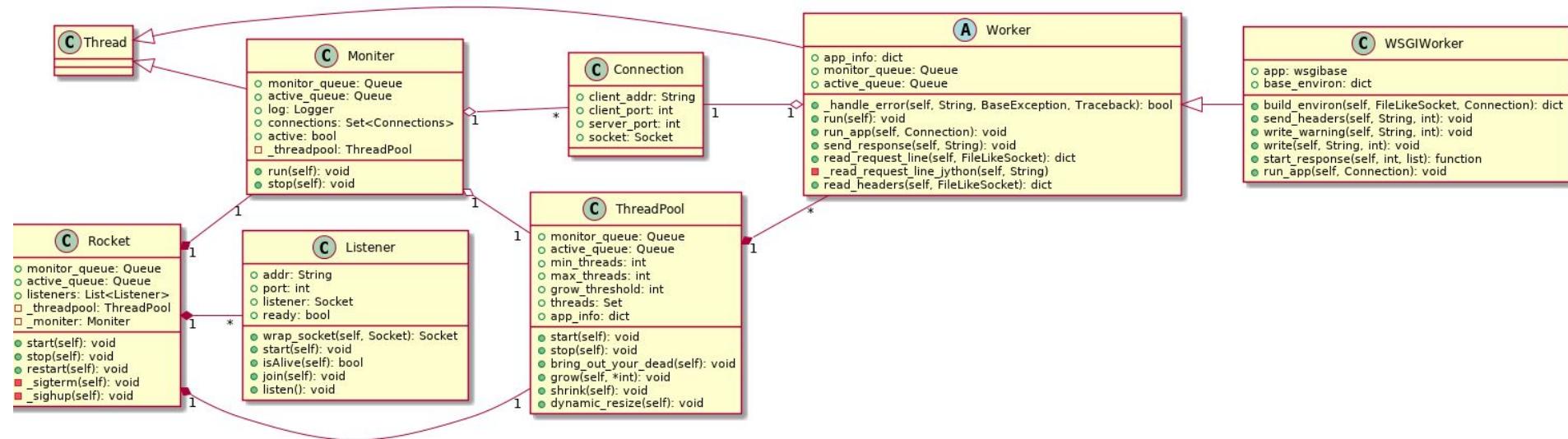
Adapter class diagram -- refactored



Rocket server

- Rocket is a lightweight open source server
- Included in web2py as the default web server

Design of Rocket: class diagram



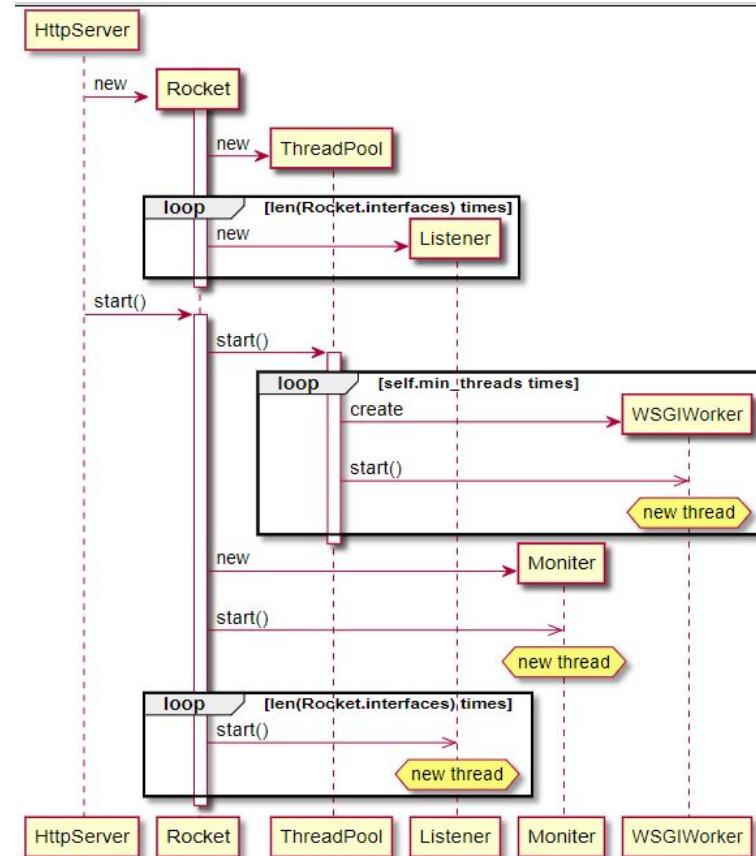
Rocket: responsibility of each class

- Listener: listens on an interface and accepts incoming connections.
- Worker (subclass of Thread): Reads request headers & contents from socket.
- WSGIWorker (subclass of Worker): It implements `run_app()` and is responsible of executing the WSGI application.
- ThreadPool: pool of worker threads.
- Monitor: It monitors connections; manages the size of ThreadPool.

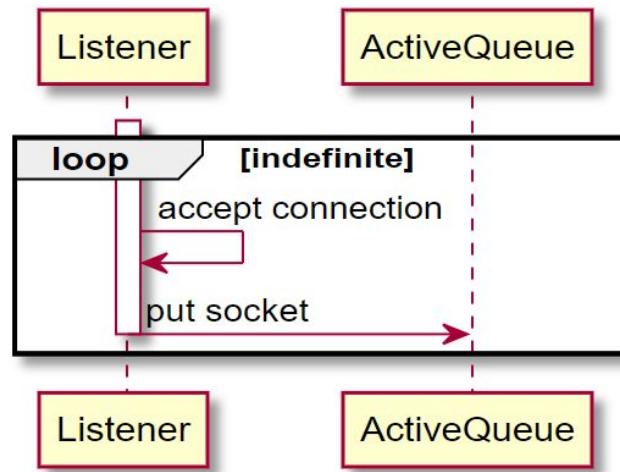
Rocket: two queues

- Active queue:
Active connections are put into this queue by Listeners, and retrieved from this queue by Workers
- Monitor queue:
Stalled or timeout connections are put into this queue, and are monitored by the monitor class.

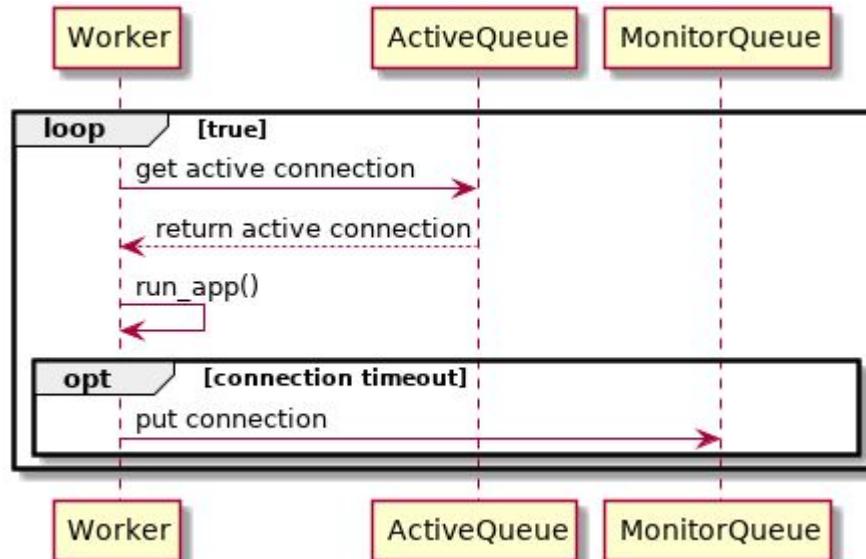
Sequence diagram: start-up



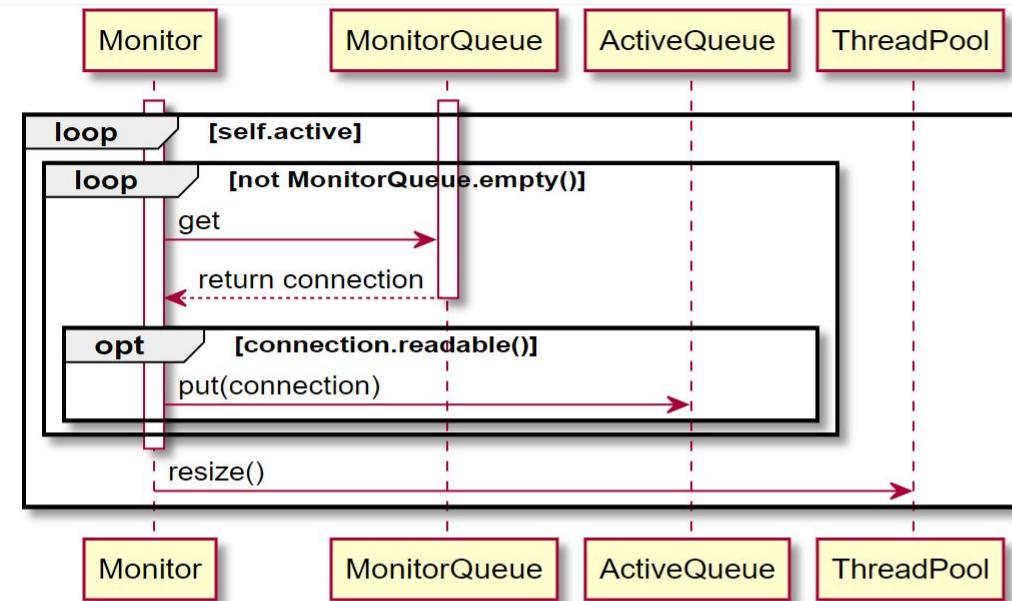
Sequence diagram: listening for incoming connections



Sequence diagram: process connection

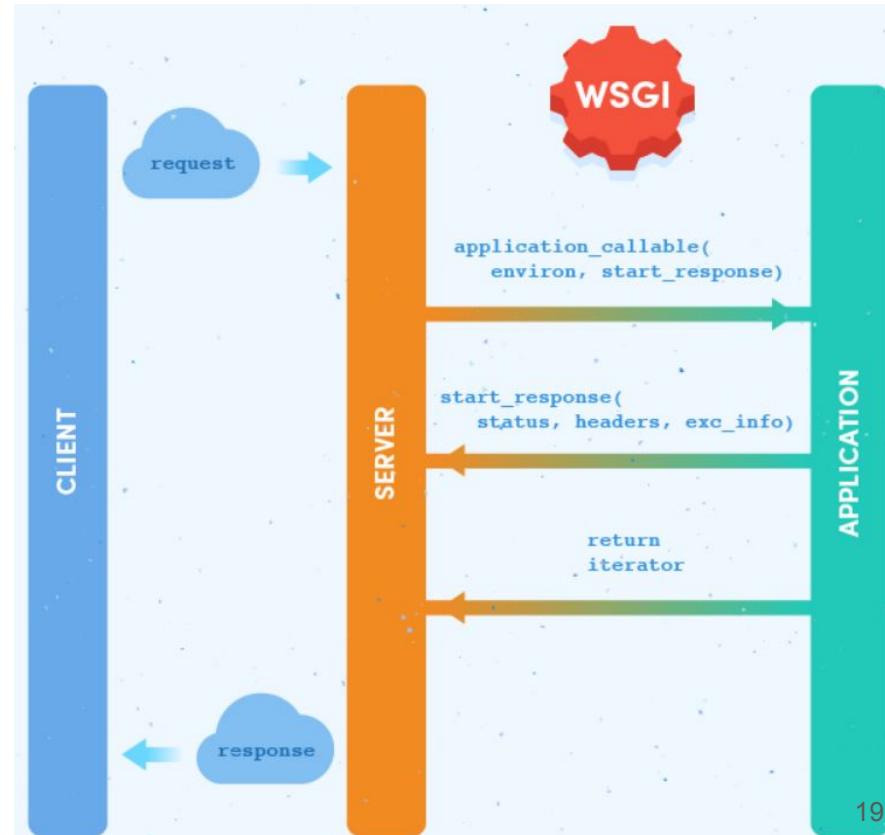


Sequence diagram: monitor connection status

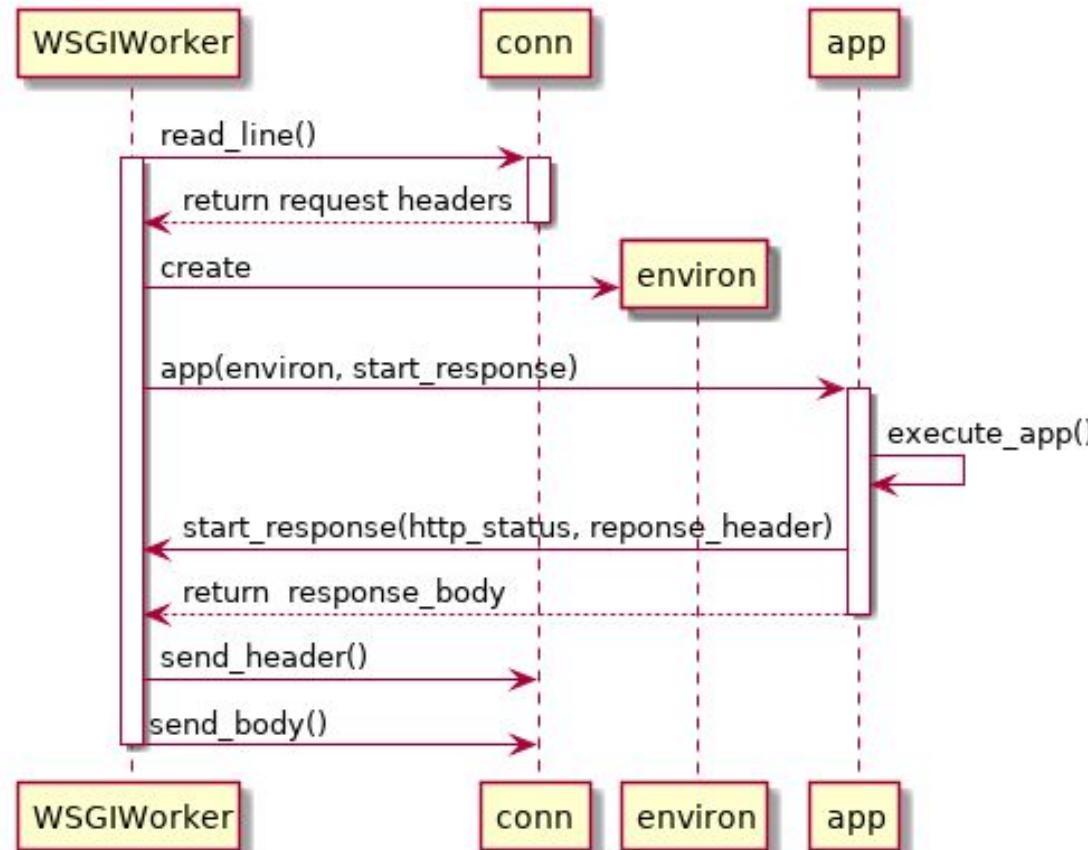


WSGI (Web Server Gateway Interface)

- A simple **calling convention** for web servers to forward requests to web applications or frameworks written in the Python programming language



Sequence diagram: running the WSGI application



Server manages HTTP requests

Parse the HTTP request header

Request for
a static file?

True

Read and return (stream) the
requested file

False

Request to controller

Expose wsgi hooks

Parse and load cookies

Load session or create a new one

Run application

Rewrite and parsing incoming URLs

- URL Rewrite
 - Motivation to rewrite incoming URLs
 - Handle legacy URLs
 - Simplify paths and make them shorter
 - Two distinct URL rewrite systems
 - The parameter-based system
 - The pattern-based system

Rewrite and parsing incoming URLs (cont.)

- Parameter-based system
 - Omit default application, controller and function names from externally-visible URLs (those created by the URL() function)
 - Map domains (and/or ports) to applications or controllers
 - Embed a language selector in the URL

```
routers = dict(  
    BASE = dict(default_application='myapp'),  
)
```

The application name is no longer part of the URL

```
routers = dict(  
    BASE = dict(  
        domains = {  
            'domain1.com' : 'app1',  
            'domain2.com' : 'app2',  
    })
```

Domain mapping

Rewrite and parsing incoming URLs (cont.)

- Pattern-based system
 - Provide some additional flexibility for more complex cases
 - Instead of defining routers as dictionaries of routing parameters, you define two lists (or tuples) of 2-tuples
 - Each tuple contains two elements: the pattern to be replaced and the string that replaces it

```
routes_in = (
    ('/testme', '/examples/default/index'),
)
routes_out = (
    ('/examples/default/index', '/testme'),
)
```

To the visitor, all links to the page URL looks like /testme.

```
routes_in = (
    ('/(?P<any>.*)', '/init/\g<any>'),
)
routes_out = (
    ('/init/(?P<any>.*)', '/\g<any>'),
)
```

Get rid of the application prefix from the URLs

Rewrite and parsing incoming URLs

- URL Parsing
 - For static pages:
 - /<application>/static/<file>
 - For dynamic pages:
 - /<a:application>[<c:controller>[<f:function>[.<e:ext>][<s:args>]]]

```
request.application = match.group('a') or routes.default_application
request.controller = match.group('c') or routes.default_controller
request.function = match.group('f') or routes.default_function
request.raw_extension = match.group('e')
request.extension = request.raw_extension or 'html'
```

Server manages HTTP requests

Parse The HTTP request header

Request for
a static file?

True

Read and return (stream) the
requested file

False

Request to controller

Expose wsgi hooks

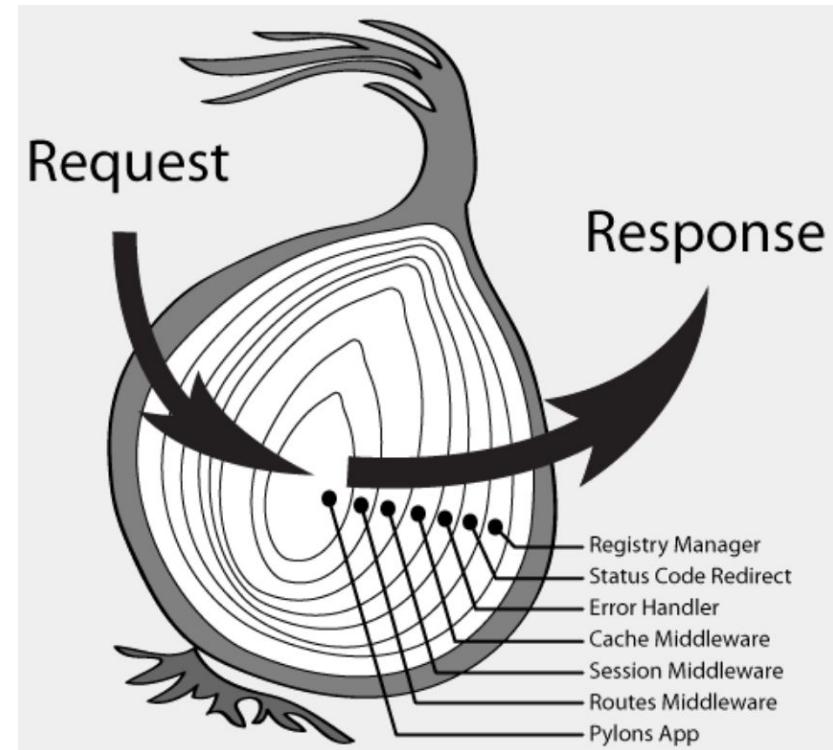
Parse and load cookies

Load session or create a new one

Run application

WSGI (Web Server Gateway Interface) (cont.)

- Some developers have pushed WSGI to its limits as a protocol for middleware communications and develop web applications as an onion with many layers (each layer being a WSGI middleware developed independently of the entire framework)



WSGI (Web Server Gateway Interface) (cont.)

- web2py does not adopt this structure internally. This is because they feel the core functionality of a frameworks (handling cookies, session, errors, transactions, dispatching) can be better optimized for **speed** and **security** if they are handled by a **single** comprehensive layer.

```
def wsgibase(environ, responder):
    """
    The gluon wsgi application. The first function called when a page
    is requested (static or dynamic). It can be called by paste.httpserver
    or by apache mod_wsgi (or any WSGI-compatible server).
    """

    # If the URL ends with a slash, add a trailing slash to the path
    if environ['PATH_INFO'][-1] != '/':
        environ['PATH_INFO'] += '/'
```

web2py at its core is a WSGI application: gluon.main.wsgibase

Third party WSGI applications and middleware

- web2py allows you to use third party WSGI applications and middleware in **three ways** (and their combinations)
 - External middleware
 - Internal middleware
 - Calling WSGI applications

Third party WSGI applications and middleware (cont.)

- External middleware (Add any third party middleware)

- ❑ wsgibase is wrapped by the middleware function appfactory
- ❑ In a similar fashion you can add any third party middleware

```
def appfactory(wsgiapp=wsgibase,
               logfilename='httpserver.log',
               profiler_dir=None,
               profilerfilename=None):
    """
    generates a wsgi application that does logging and profiling and calls
    wsgibase

    Args: ...
    """

    if profilerfilename is not None: ...
    if profiler_dir: ...

    def app_with_logging(environ, responder): ...

    return app_with_logging
```

```
app_info = {'wsgi_app': appfactory(wsgibase,
                                    log_filename,
                                    profiler_dir)}
```

Third party WSGI applications and middleware (cont.)

- `request.wsgi`
 - A hook that allows you to call third party WSGI applications from inside actions or decorate functions with WSGI middleware

```
class LazyWSGI(object):
    def __init__(self, environ, request, response): ...

    @property
    def environ(self): ...

    def start_response(self, status='200', headers=[], exec_info=None):
        ...

    def middleware(self, *middleware_apps): ...
```

```
# ##### expose wsgi hooks for convenience #####
# #####
request.wsgi = LazyWSGI(environ, request, response)
```

Third party WSGI applications and middleware (cont.)

- Internal middleware
 - You can use a web2py decorator to apply the middleware to the action in your controllers

```
class MyMiddleware:  
    """converts output to upper case"""  
    def __init__(self,app):  
        self.app = app  
    def __call__(self, environ, start_response):  
        items = self.app(environ, start_response)  
        return [item.upper() for item in items]  
  
@request.wsgi.middleware(MyMiddleware)  
def index():  
    return 'hello world'
```

Third party WSGI applications and middleware (cont.)

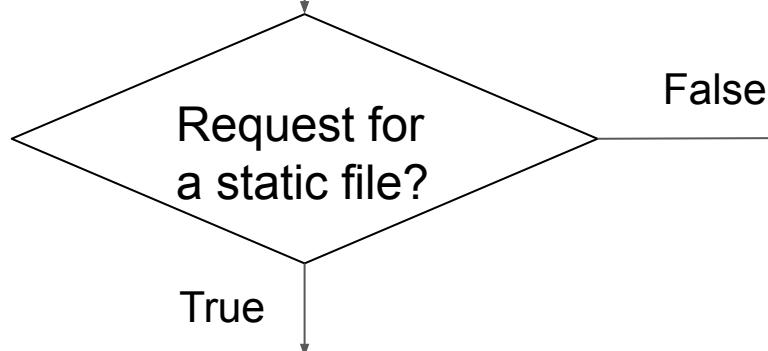
- Calling WSGI applications
 - Call WSGI app from a web2py action
 - An example:

```
def test_wsgi_app(environ, start_response):
    """this is a test WSGI app"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', '13')]
    start_response(status, response_headers)
    return ['hello world!\n']

def index():
    """a test action that calls the previous app and escapes output"""
    items = test_wsgi_app(request.wsgi.environ,
                          request.wsgi.start_response)
    for item in items:
        response.write(item, escape=False)
    return response.body.getvalue()
```

Server manages HTTP requests

Parse The HTTP request header



Read and return (stream) the requested file

Request to controller

Expose wsgi hooks

Parse and load cookies

Load session or create a new one

Run application

Cookies & Session

- Use the Python cookies modules for handling cookies
- Session is an instance of the **Storage class**
 - Trying to access an attribute/key that has not been set does not raise an exception; it returns None instead
- By default sessions are stored on the filesystem and a session cookie is used to store and to retrieve the session.id

```
if env.http_cookie:  
    for single_cookie in env.http_cookie.split(';'):   
        single_cookie = single_cookie.strip()  
        if single_cookie:  
            try:  
                request.cookies.load(single_cookie)  
            except Cookie.CookieError:  
                pass # single invalid cookie ignore
```

Load cookies

```
if not env.web2py_disable_session:  
    session.connect(request, response)
```

Try to load session or create new session file

Storage: the ubiquitous web2py class

- Extends the Python dict class
- Basically a dictionary, except that the item values can be accessed as attributes
- Unlike dict, trying to access a item/attribute that has not been set, it returns None instead of raising an exception

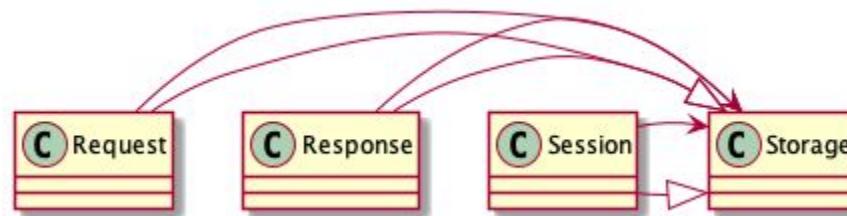


```
request[env]  
request[cookies]  
request[vars]  
request[restful()]
```

```
request.env  
request.cookies  
request.vars  
request.restful()
```

Storage: the ubiquitous web2py class

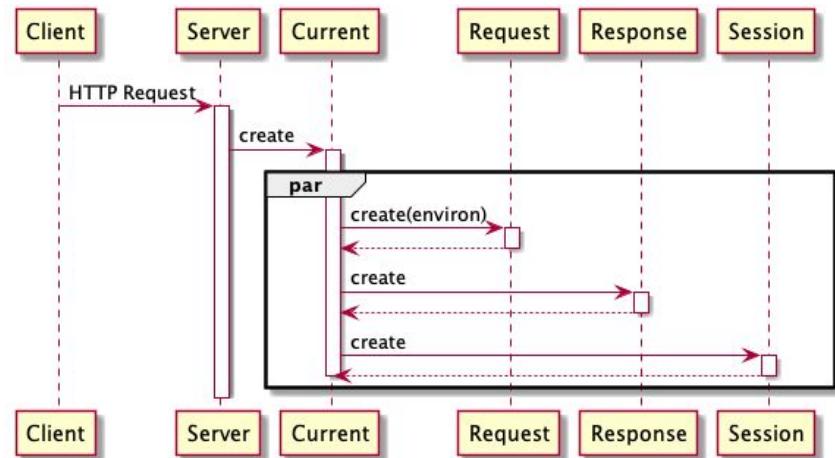
- Multiple instances throughout web2py: Request, Response, Session, env, vars, global_settings, user, router, etc., and these instances can include other Storage items/attributes
- Instead of segregating implementation from interface, web2py often opts to check if one Storage attribute exists



Current Container

- A thread-local container that stores some global objects request, response, session, T, cache, etc.
- Modules can only access them when they are called from an application

```
current = threading.local() # thread-local storage for request-scope global
```



Current Container

- Different users can call the same action index concurrently, which calls the function in the module
 - Yet with no conflicts because `current.request` is a different object in different threads
- `current` and `import` create a powerful mechanism to build extensible and reusable modules for applications

```
auth = Auth(db)
from gluon import current
current.auth = auth
```

Now all modules can access `current.auth`!

Cookies & Session

- **Cookies** and **Session** are dealt within `main`, the main WSGI application that performs all common tasks and wraps user applications
- Session cookies enable the website to keep track of user movement from page to page
 - In web2py, sessions are stored on the filesystem and a session cookie is used to store and retrieve the `session_id` by default
 - Cookies from the browser are in `request.cookies` and those sent by the server are in `response.cookies`

```
cookies = request.cookies
```

```
response.cookies[response.session_id_name] = response.session_id  
response.cookies[response.session_id_name]['path'] = '/'
```

Session

- An instance of **Storage**: whatever is stored can be retrieved as an attribute
- The dispatcher decides which of the installed action will handle the request and maps the path in the URL into a function call

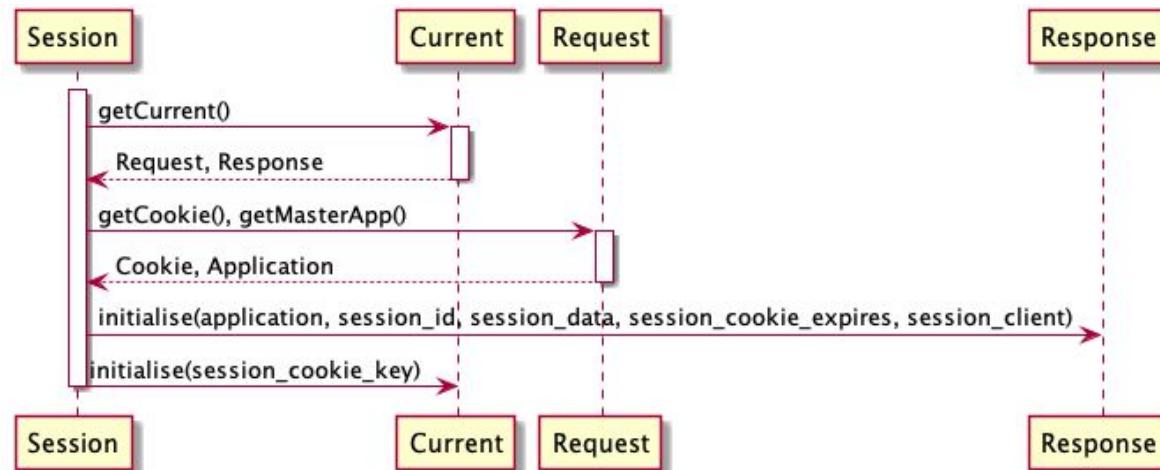
```
# [controller].py
def index():
    images = db().select(db.image.ALL, orderby=db.image.title)
    return dict(images=images)
```

[application]/[controller]/index

- Static files - stored in the application's static subfolder
 - Handled directly, while large files are streamed directly to the client
- Anything else - mapped into an action, where session comes into play before the action is called

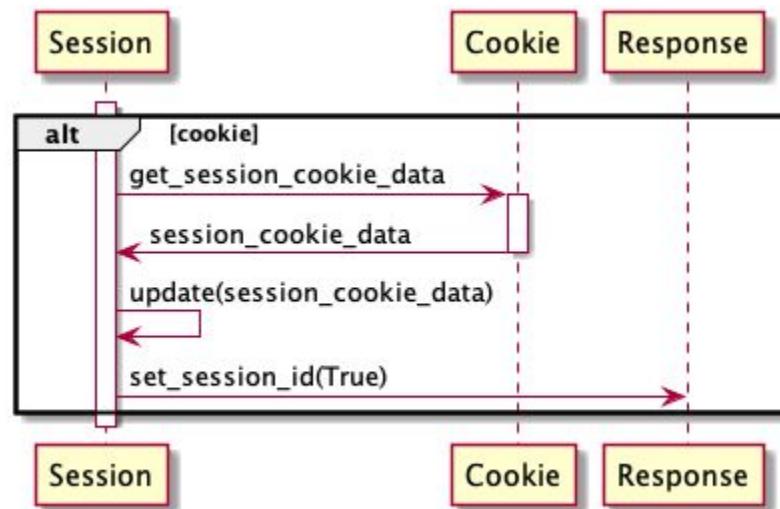
Connect Session

- Try to load session or create new session file
 - Parse cookies
 - If the request header contains a session cookie for the app, the session object is retrieved
 - If not, a session id is created (but the session file is not saved until later)



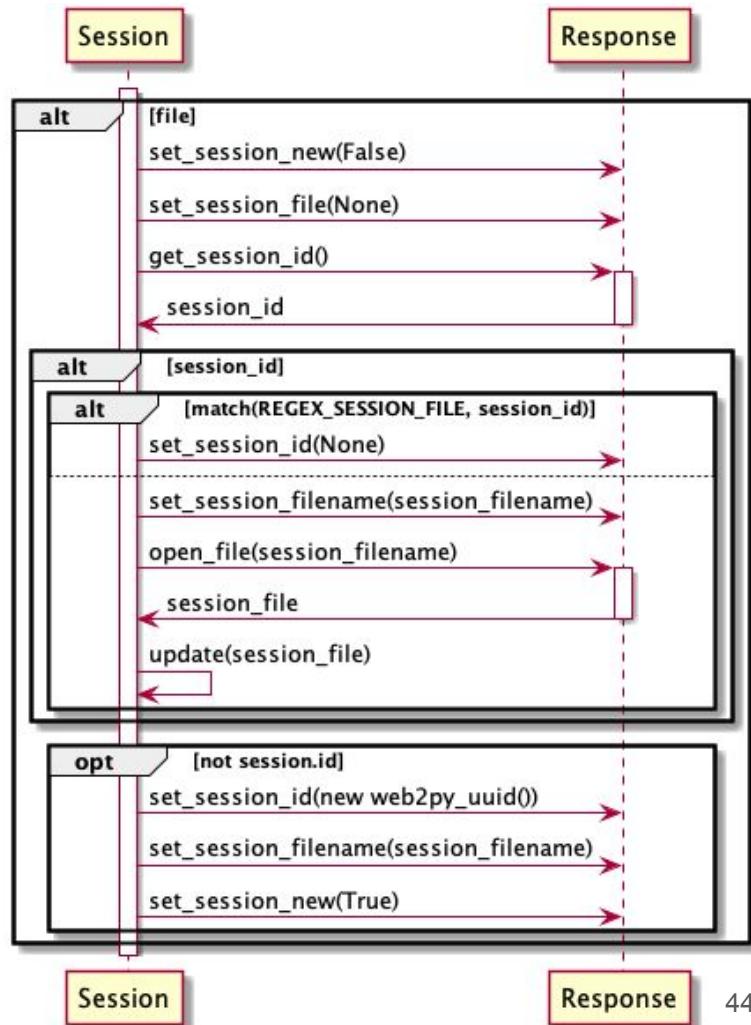
Connect Session

- Store session in cookie
 - Check if there is session data in cookies



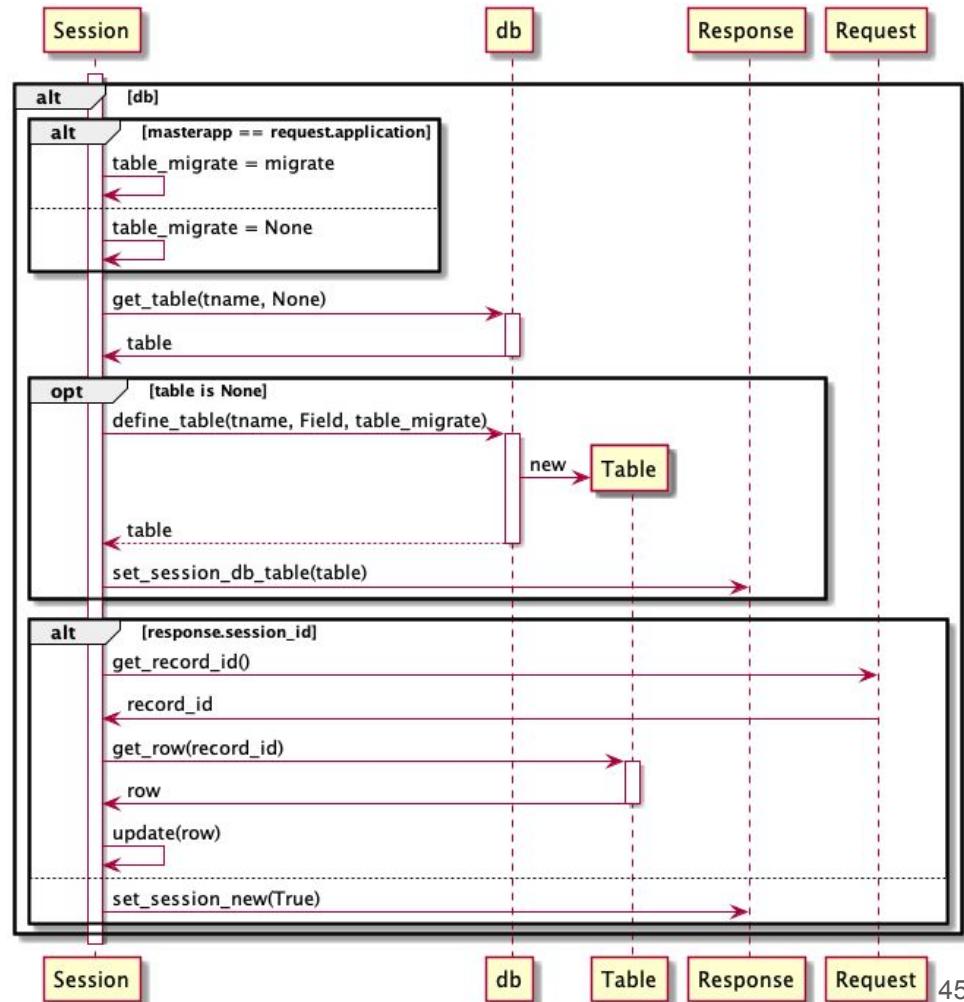
Connect Session

- Store session in filesystem
 - Check if session_id points to a valid session filename
 - check_client: if True, sessions can only come from the same ip. Used to detect cookie attacks
 - separate: if True, creates a folder with the 2 initials of session_id. Useful when file system access becomes a bottleneck.



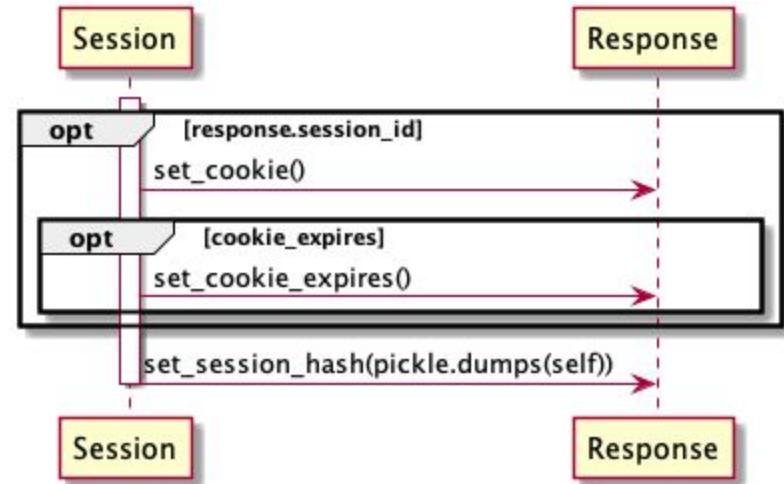
Connect Session

- Store session in db
 - masterapp: point to another app's session
 - migrate: passed to the underlying db
 - Get session data out of the database



Connect Session

- Wrap up
 - Set cookies if we know the session_id so user can set cookie attributes in controllers/models
 - cookies_expires: cookie may be reset later



Forget & Secure Session

- `Session.forget()`
 - Tell web2py not to save the session
 - Prevent the session file from being written
 - Used in controllers whose actions are called often and do not need to track user activity
- `Session.secure`
 - Tell web2py to save the session cookie to be a secure cookie
 - Prevent the browser from sending the session cookie back to the server unless over an https connection
 - Set if the app is going over https

```
def forget(self, response=None):  
    self._close(response)  
    self._forget = True
```

```
def secure(self):  
    self._secure = True
```

Server manages HTTP requests

Parse The HTTP request header

Request for
a static file?

True

Read and return (stream) the
requested file

False

Request to controller

Expose wsgi hooks

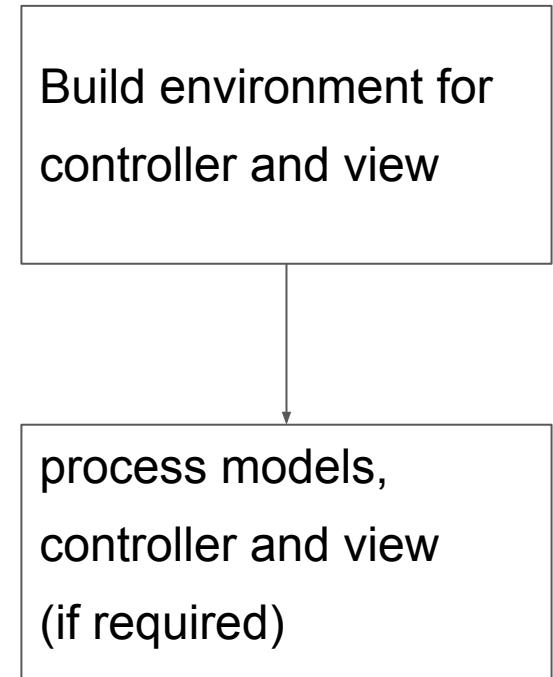
Parse and load cookies

Load session or create a new one

Run application

Run Application

- Build environment for controller and view
 - web2py model and controller files are not Python modules
 - models and controllers are designed to be executed in a prepared environment that has been pre-populated with web2py global objects (request, response, session, cache and T) and helper functions.



Run Application (cont.)

- The controller and the view are executed in different copies of the same environment
 - The view does not see the controller, but it sees the models and it sees the variables returned by the controller action function
- The view is only called if the action returns a dictionary

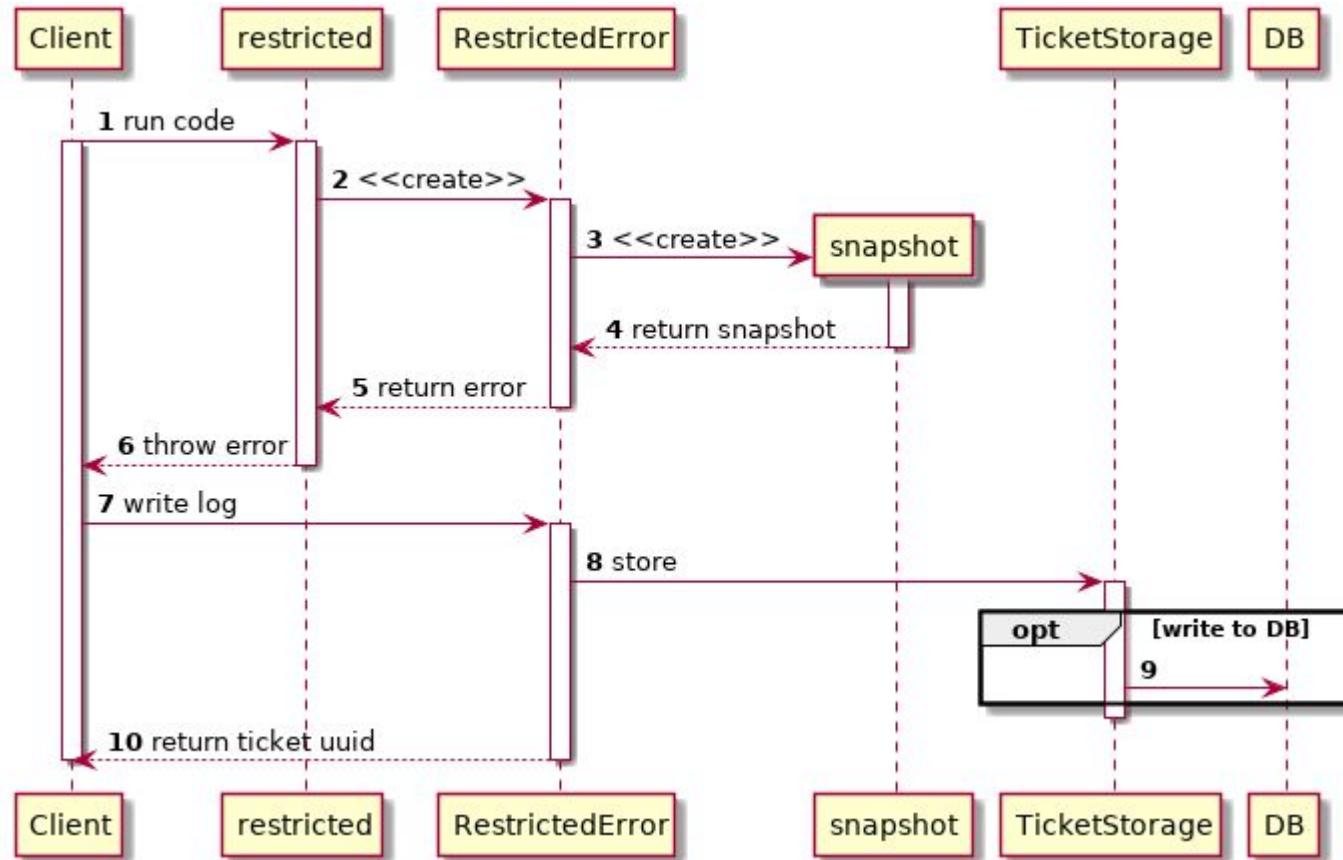
```
run_models_in(environment)
response._view_environment = copy.copy(environment)
page = run_controller_in(request.controller, request.function, environment)
if isinstance(page, dict):
    response._vars = page
    response._view_environment.update(page)
    page = run_view_in(response._view_environment)
```

Execute application code in restricted environment

- Application code is compiled and executed with exec() in a try...except... block
- Any error in the application code is catched, and the error information is stored into a database table

```
def restricted(ccode, environment=None, layer='Unknown', scode=None):  
  
    ...  
    try:  
        exec(ccode, environment)  
        ...  
  
    except Exception as error:  
        # extract the exception type and value (used as output message)  
        etype, evalue, tb = sys.exc_info()  
        ...  
  
        # Save source code in ticket when available  
        scode = scode if scode else ccode  
        raise RestrictedError(layer, scode, output, environment)
```

Sequence diagram of handling error



Outline

- Briefly introduce web2py
- Request to Controller
 - Server
 - Rewrite and parsing incoming URLs
 - WSGI hooks
 - Cookies & Session
 - Run Application
- **Services**
- Views
- Cron & Scheduler
- PyDAL
- Conclusion

Restful Web Services

- A client communicates with the resource by making an HTTP request to the URL that identifies the resource and using the HTTP method POST/GET/PUT/DELETE to pass instructions to the resource
 - For example, we might want to send a GET request to

```
http://127.0.0.1/myapp/default/simple_rest/persons.json
```

- To create a restful action in web2py

```
def simple_rest():
    return locals()
```

- `request.restful` provides a very useful decorator that can be used to change default web2py behaviour

Restful

-

```
def restful(self, ignore_extension=False):
    def wrapper(action, request=self):
        def f(_action=action, *a, **b):
            request.is_restful = True
            env = request.env
            is_json = env.content_type == 'application/json'
            method = env.request_method
            if not ignore_extension and len(request.args) and '.' in request.args[-1]:
                request.args[-1], _, request.extension = request.args[-1].rpartition('.')
                current.response.headers['Content-Type'] = \
                    contenttype('.' + request.extension.lower())
            rest_action = _action().get(method, None)
            if not (rest_action and method == method.upper()
                    and callable(rest_action)):
                raise HTTP(405, "method not allowed")
            try:
                res = rest_action(*request.args, **request.vars)
                if is_json and not isinstance(res, str):
                    res = json(res)
                return res
            except TypeError as e:
                exc_type, exc_value, exc_traceback = sys.exc_info()
                if len(traceback.extract_tb(exc_traceback)) == 1:
                    raise HTTP(400, "invalid arguments")
                else:
                    raise
            f.__doc__ = action.__doc__
            f.__name__ = action.__name__
            return f
    return wrapper
```

Restful Web Services

- web2py uses inner functions to define factory functions
 - A function that creates other objects (functions)
- Add function `request.restful()` dynamically to another object or class

```
other_request.restful = types.MethodType(request.restful.__func__, other_request)

def restful(self, ignore_extension=False):
    def wrapper(action, request=self):
        def f(_action=action, *a, **b):
            request.is_restful = True
            env = request.env
            is_json = env.content_type == 'application/json'
            method = env.request_method
            if not ignore_extension and len(request.args) and '.' in request.args[-1]:
                request.args[-1], _, request.extension = request.args[-1].rpartition('.')
                current.response.headers['Content-Type'] = \
                    contenttype('.' + request.extension.lower())
            rest_action = _action().get(method, None)
            if not (rest_action and method == method.upper() and callable(rest_action)):
                raise HTTP(405, "method not allowed")
            if is_json and not isinstance(res, str):
                res = json(res)
            return res
        except TypeError as e:
            exc_type, exc_value, exc_traceback = sys.exc_info()
            if len(traceback.extract_tb(exc_traceback)) == 1:
                raise HTTP(400, "invalid arguments")
            else:
                raise
        f.__doc__ = action.__doc__
        f.__name__ = action.__name__
        return f
    return wrapper
```

Restful Web Services

- Modify restful action in web2py

```
def simple_rest():
    return locals()
```



- @request.restful() decorator

- Make sure that the extension (.html, .xml, .json) is stored in request.extension
- Map the request method into the corresponding function within the action (GET, POST, PUT, DELETE)
- Pass request.args and request.vars to the selected function

```
@request.restful()
def rest():
    def GET(*args,**vars):
        patterns = [
            "/friends[person]",
            "/{person.name}/:field",
            "/{person.name}/pets[pet.ownedby]",
            "/{person.name}/pets[pet.ownedby]/"
            "{pet.name}",
            "/{person.name}/pets[pet.ownedby]/"
            "{pet.name}/:field",
            (""/dogs[pet]", db.pet.info=='dog'),
            (""/dogs[pet]/{pet.name.startswith}",",
            db.pet.info=='dog'),
        ]
        parser = db.parse_as_rest(patterns,args,
                                   vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,
                       parser.error)

    def POST(table_name,**vars):
        if table_name == 'person':
            return db.person.validate_and_insert
            (**vars)
        elif table_name == 'pet':
            return db.pet.validate_and_insert
            (**vars)
        else:
            raise HTTP(400)
    return locals()
```

Remote Procedure Calls

- web2py provides a mechanism to turn any function into a web service
 - Work for a fixed set of protocols; not easily extensible
 - May take arguments or be defined in a model instead of controller
 - Decorate functions to expose as a service

```
http://127.0.0.1:8000/app/default/call/run/myfunction?a=10&b=11  
http://127.0.0.1:8000/app/default/call/run/myfunction/10/11
```

```
http://127.0.0.1:8000/app/default/call/xml/myfunction/10/11
```

```
http://127.0.0.1:8000/app/default/call/json/myfunction/10/11
```

```
http://127.0.0.1:8000/app/default/call/jsonrpc/myfunction/10/11
```

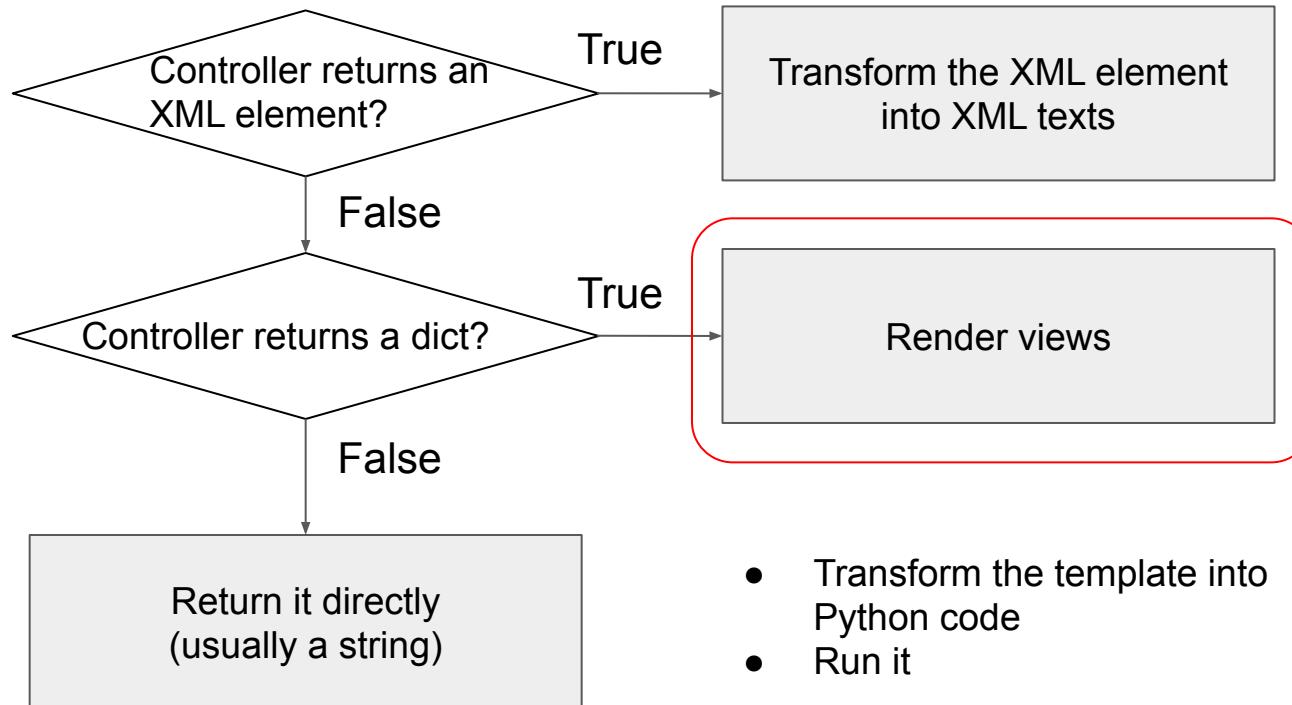
```
http://127.0.0.1:8000/app/default/call/xmlrpc/myfunction/10/11
```

```
from gluon.tools import Service  
service = Service()  
@service.run  
def myfunction(a,b):  
    return a + b  
# expose service handler in the controller  
def call():  
    return service()  
  
service = Service()  
@service.xml  
def myfunction(a,b):  
    return a + b  
def call():  
    return service()  
  
service = Service()  
@service.json  
def myfunction(a,b):  
    return [a: b]  
def call():  
    return service()  
  
service = Service()  
@service.jsonrpc  
def myfunction(a,b):  
    return a + b  
def call():  
    return service()  
  
service = Service()  
@service.xmlrpc  
def myfunction(a,b):  
    return a + b  
def call():  
    return service()
```

Outline

- Briefly introduce web2py
- Request to Controller
 - Server
 - Rewrite and parsing incoming URLs
 - WSGI hooks
 - Cookies & Session
 - Run Application
- Services
- **Views**
- Cron & Scheduler
- PyDAL
- Conclusion

How the Client Gets Data

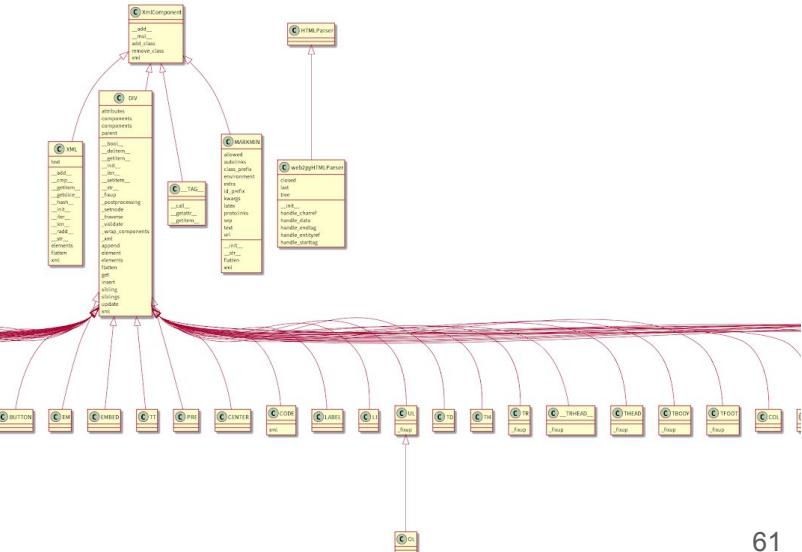


- Transform the template into Python code
- Run it

XML Elements: Helpers for HTML Tag

- Create HTML tags with Python functions
- Doing miscellaneous stuffs (e.g., escaping)

```
def index():
    import gluon.html
    return gluon.html.SPAN("Hello World <3")
# client gets <span>Hello World &lt;3</span>
```



Yet Another Template Language (YATL)

- A template language similar to Django's templates or Flask's Jinja2
- Allow using control flow statements of Python
- A template can extend another one
- Often used with HTML tag helpers to generate HTML with ease
- The main class, TemplateParser, does both parsing and transformation to Python

```
 {{ if authenticated: }}  
   {{=T("Good!")}}  
 {{ else: }}  
   {{=T("Oops")}}  
 {{ pass }}
```



```
if authenticated:  
    response.write(T("Good!"))  
else:  
    response.write(T("Oops"))  
pass
```

Extension of Templates

layout.html

```
<html><head>
{{ block head }}
<title>{{ block title }}web2py view{{ end }}</title>
{{ end }}
</head><body></body></html>
```

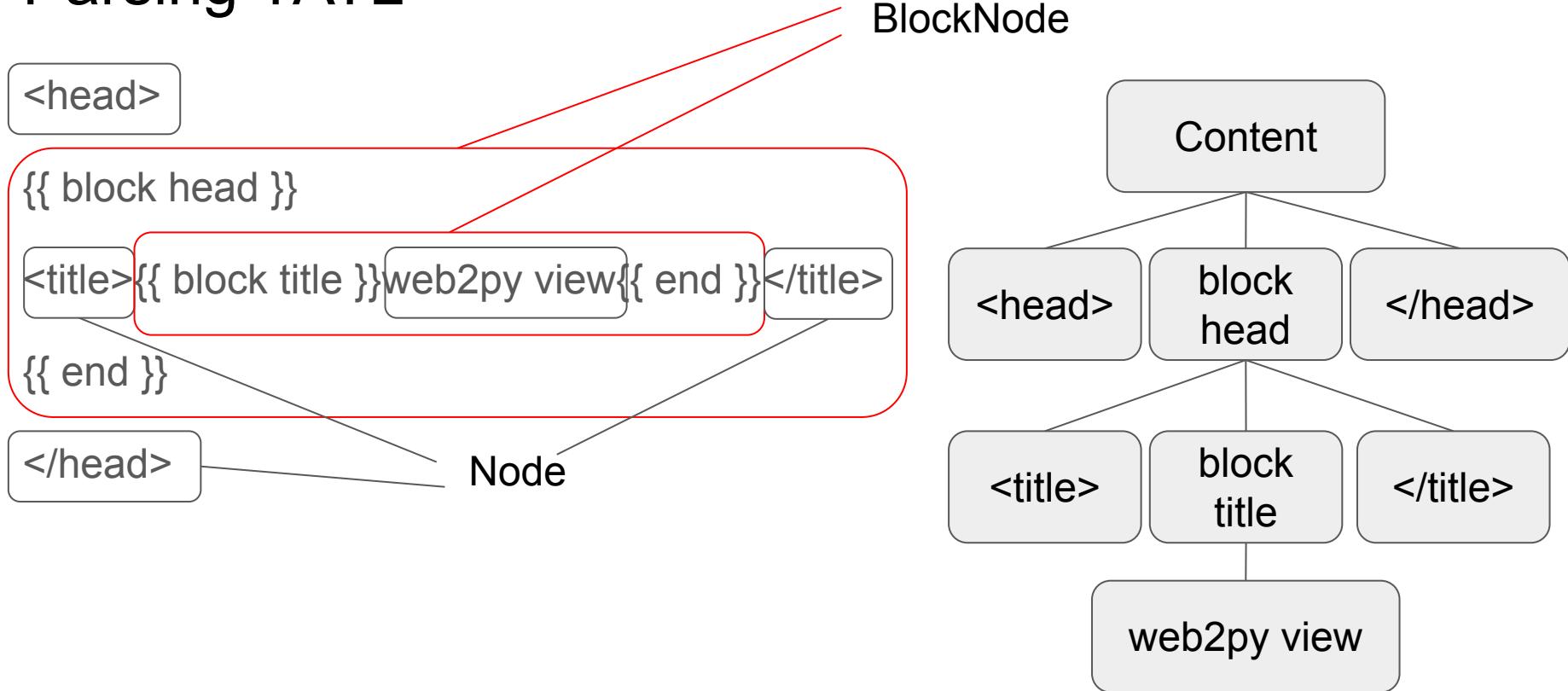
- **block**: contents that can be replaced
- **super**: place the content from the base
- **extends**: for template reuse

index.html

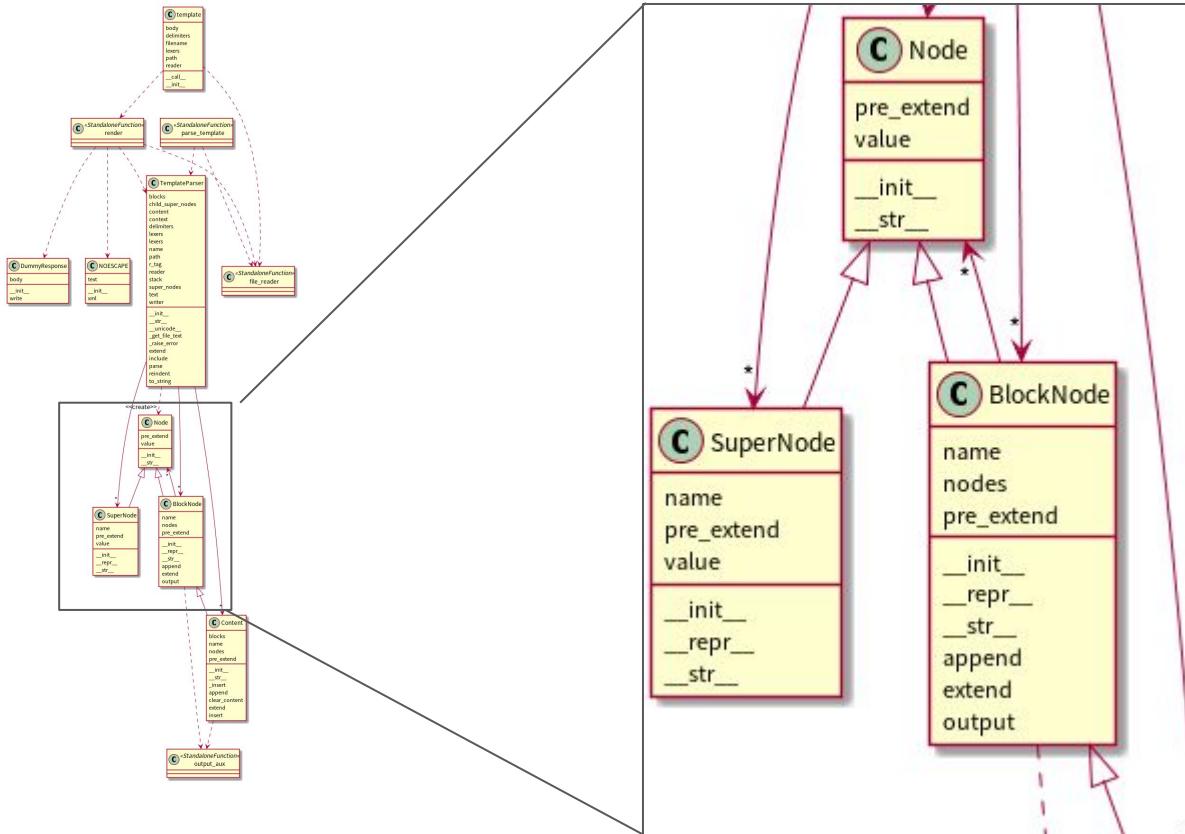
```
{{ extends "layout.html" }}
{{ block head }}
{{ super }}
<link rel="stylesheet" href="index.css" />
{{ end }}
{{ block title }}Index{{ end }}
```

```
<html><head>
<title>Index</title>
<link rel="stylesheet" href="index.css" />
</head><body></body></html>
```

Parsing YATL



Class Diagram for YATL



Composite Pattern

How the Parser works

```
if name == 'block' and not value.startswith('='):
    # Make a new node with name.
    node = BlockNode(name=value.strip(),
                      pre_extend=pre_extend,
                      delimiters=self.delimiters)
    # Append this node to our active node
    top.append(node)
    # Make sure to add the node to the stack.
    # so anything after this gets added
    # to this node. This allows us to
    # "nest" nodes.
    stack.append(node)
elif name == 'end' and not value.startswith('='):
    # We are done with this node.
    # Save an instance of it
    self.blocks[top.name] = top
    # Pop it.
    stack.pop()
elif name == 'super' and not value.startswith('='):
    # Get our correct target name
    # If they just called {{super}} without a name
    # attempt to assume the top blocks name.
    if value:
        target_node = value
    else:
        target_node = top.name
    # Create a SuperNode instance
    node = SuperNode(name=target_node,
                      pre_extend=pre_extend)
    # Add this to our list to be taken care of
    self.super_nodes.append(node)
    # And put in in the tree
    top.append(node)
```

For {{ block }}

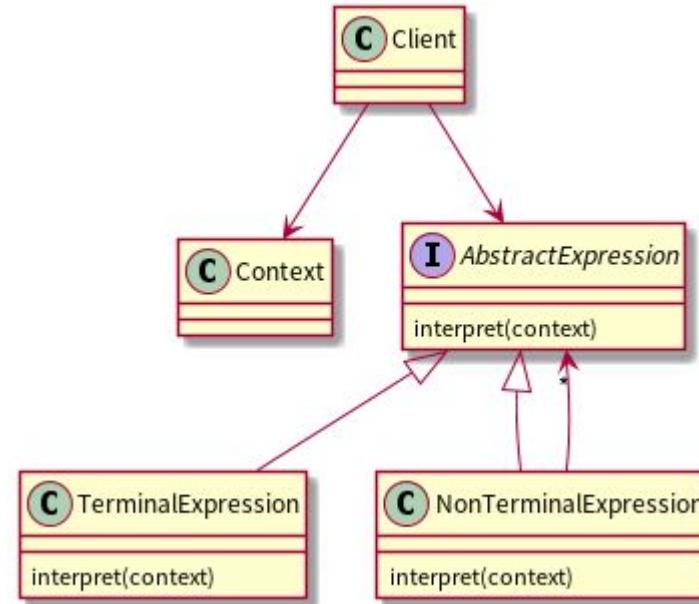
For {{ end }}

For {{ super }}

Problem: need to change
the parser whenever a
expression changes

No Interpreter Pattern

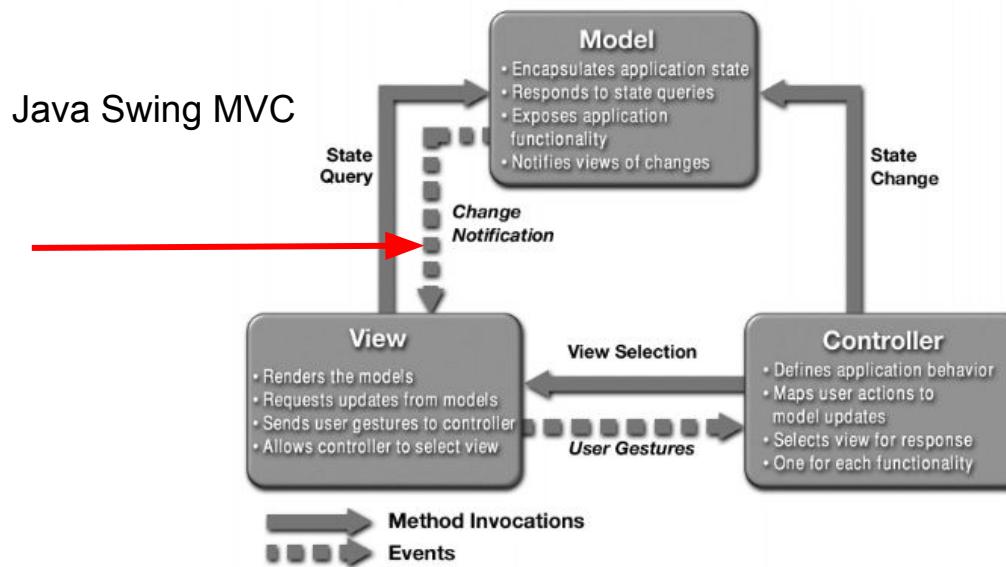
- Encapsulate what varies
 - Actions for different nodes to different Expression classes
- Abstract common behaviors
 - An AbstractExpression class for various Expression classes



No Observer Pattern

- MVC frameworks use the observer pattern
- But what web2py returns to the client are static HTML
- Actually “Model 2”

Where the observer pattern appears, but web2py does not implement it.



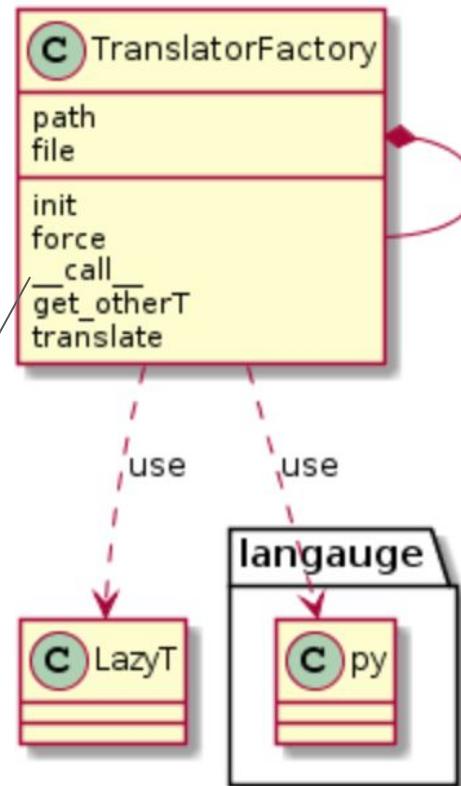
Languages.py

```
T = languages.TranslatorFactory(self.langpath, self.http_accept_language)  
self.assertEqual(str(T('Hello World')),  
                 'Salve Mondo')
```

- Translation system
- Only one class named TranslatorFactory is used
- Its work is translating key words into appointed language
- TranslatorFactory is not factory pattern, but like an initial design

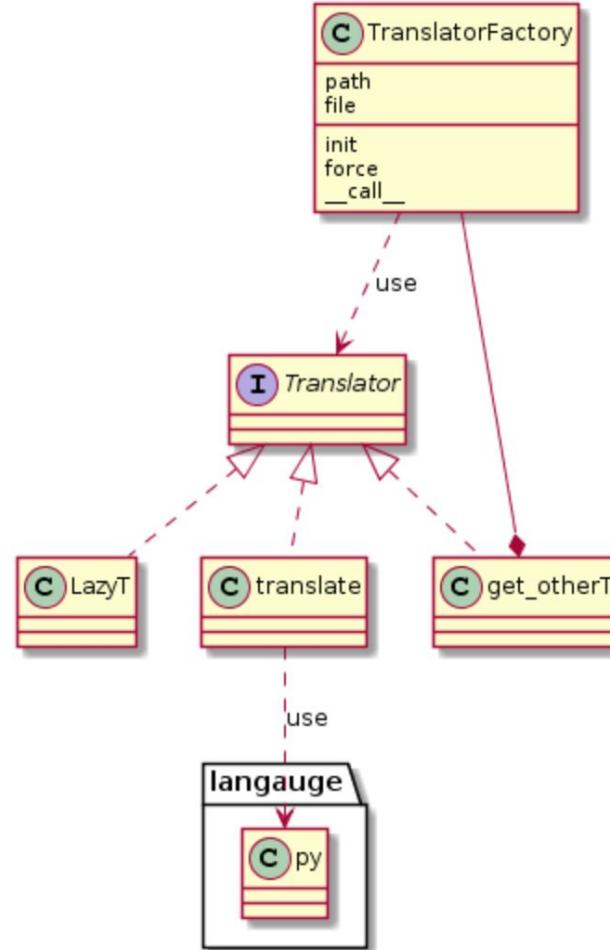
Languages.py

```
if lazy is None:
    lazy = self.lazy
if not language and not ns:
    if lazy:
        return lazyT(message, symbols, self)
    else:
        return self.translate(message, symbols)
else:
    if ns:
        if ns != self.ns:
            self.langpath = os.path.join(self.langpath, ns)
        if self.ns is None:
            self.ns = ns
otherT = self.__get_otherT__(language, ns)
return otherT(message, symbols, lazy=lazy)
```



Languages.py

- Re-design with strategy pattern
- Because of Python's property , omitting the interface is allowed



Outline

- Briefly introduce web2py
- Request to Controller
 - Server
 - Rewrite and parsing incoming URLs
 - WSGI hooks
 - Cookies & Session
 - Run Application
- Services
- Views
- **Cron & Scheduler**
- PyDAL
- Conclusion

Time-Based Job Scheduler - Cron

- Provides the ability for applications to execute tasks at preset times
- Running **periodically** tasks in the background
 - Each task runs in its own process
 - Can't control how many tasks run
- Three types of cron
 - Softcorn
 - Hardcorn
 - External corn

Time-Based Job Scheduler - Cron(cont.)

- Call class or method directly by options

```
if options.with_cron:  
    if options.soft_cron:  
        if global_settings.web2py_crontype == 'soft':  
            cmd_opts = global_settings.cmd_options  
            newcron.softcron(global_settings.applications_parent,  
                             apps=cmd_opts and cmd_opts.crontabs)
```

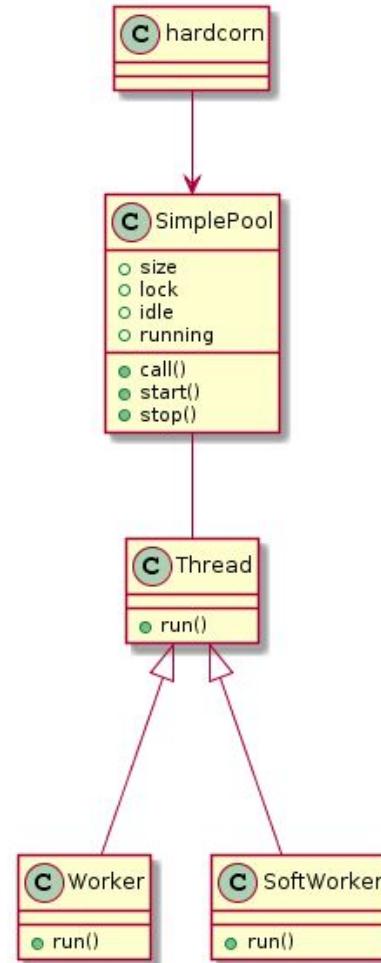
Class diagram of Cron

- A threadpool contain worker or softworker

```
class SimplePool(object):
    """
    Very simple thread pool,
    (re)uses a maximum number of threads to launch cron tasks.

    Pool size can be incremented after initialization,
    this allows delayed configuration of a global instance
    for the case you do not want to use lazy initialization.
    """
```

- Lazy initialization
- Object pool pattern



Time-Based Job Scheduler - Cron(cont.)

- Build 2 pools at initialization (use only 1 pool)

```
# set size of cron thread pools
newcron.dancer_size(options.min_threads)
newcron.launcher_size(options.cron_threads)
```

- For softcorn

```
_dancer = SimplePool(5, worker_cls=SoftWorker)
if not _dancer((applications_parent, apps)):
```

- For hardcorn

```
_launcher = SimplePool(5, worker_cls=Worker)
if not _launcher(commands):
```

Lazy Initialization and Object Pool

- Create an object when it was called and no other objects are available

```
if len(self.running) == self.size:  
    # no worker available  
    return False  
idle_num = len(self.idle)  
if idle_num:  
    # use an existing (idle) thread  
    t = self.idle.pop(0)  
else:  
    # create a new thread  
    t = self.worker_cls(self)
```

- Reuse the created object
 - create a thread consume many resources

Scheduler

```
class Scheduler(threading.Thread):  
    ...."""Scheduler.object
```

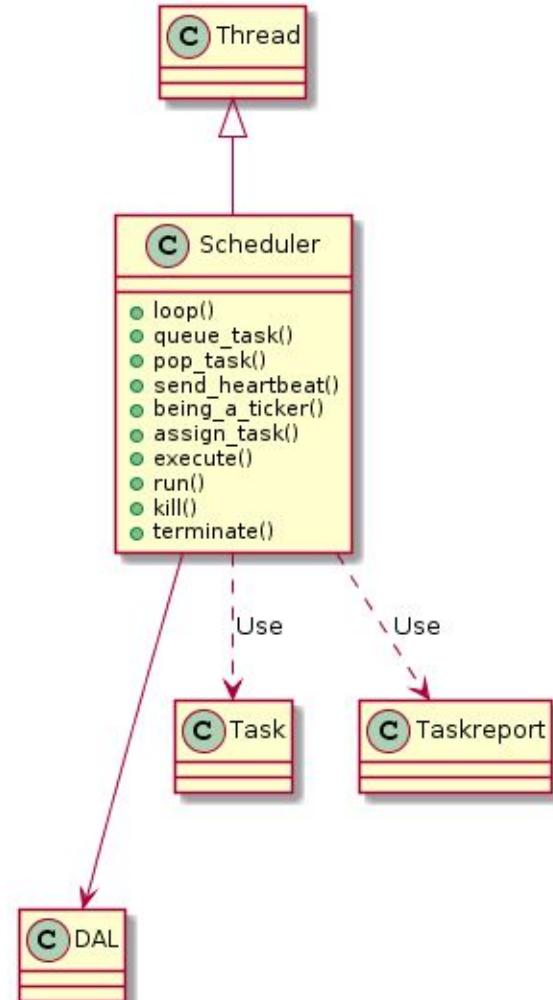
- An optional and independent class in web2py
- Access information from pyDAL
- Dependency injection - property Injection

```
current = threading.local()  
    .from gluon import current  
    .current._scheduler = self
```

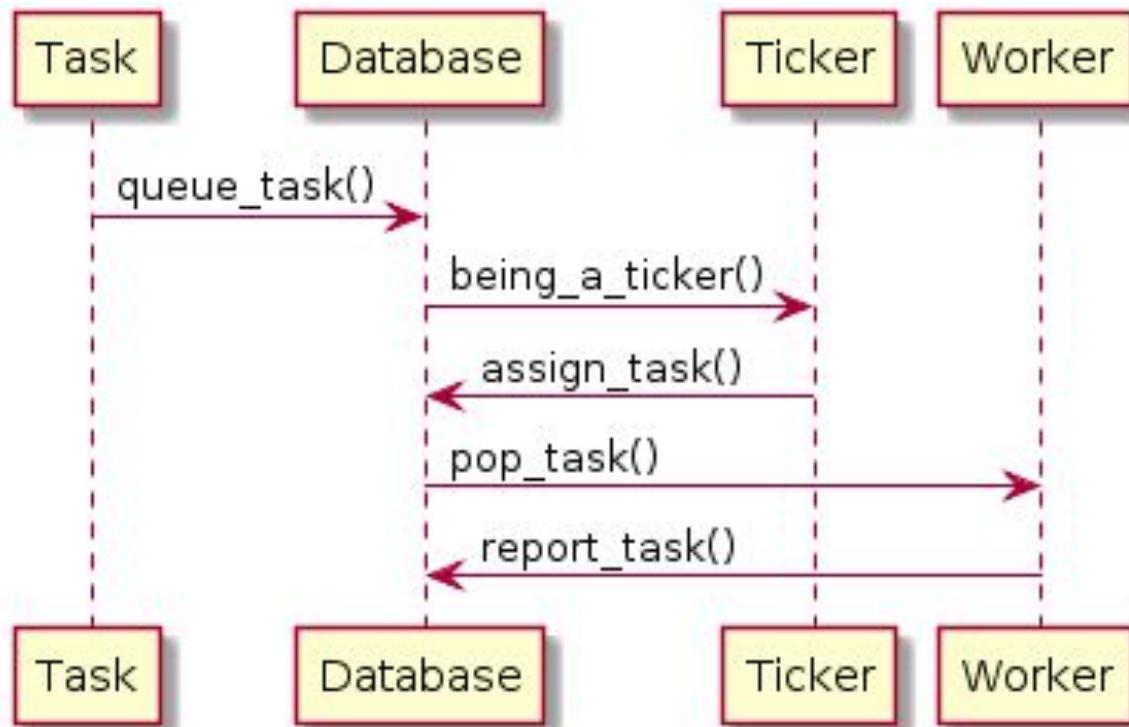
- Monitor & assign tasks to each worker
- What about original web2py? (without scheduler)
 - Task may stuck at worker
 - Get response only when worker finish job

Class Diagram of Scheduler

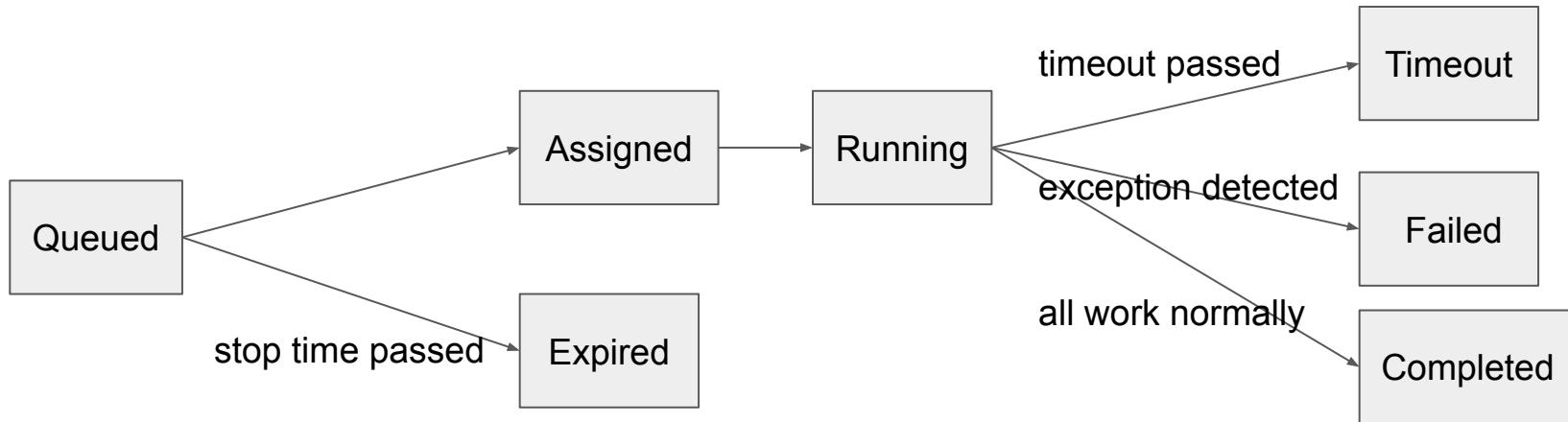
- Number of running processes(worker) is fixed
- They can run on different machines
- Check all worker after a heartbeat
- Any worker may be a ticker
- Only ticker can assign work to other workers



Sequence Diagram of Scheduler



Task State Transition



```
class Task(object):
    """Defines a "task" object that gets passed from the main thread to the
    executor's one
    """

```

Outline

- Briefly introduce web2py
- Request to Controller
 - Server
 - Rewrite and parsing incoming URLs
 - WSGI hooks
 - Cookies & Session
 - Run Application
- Services
- Views
- Cron & Scheduler
- **PyDAL**
- Conclusion

PyDAL outline

- PyDAL introduction
- PyDAL workflow
- conclusion

What is PyDAL

- PyDAL stands for Python Data Abstraction Layer
- A Python module that can access different types of databases
- Written by web2py author and can be used alone

```
from pydal import DAL, Field
db = DAL('sqlite://storage.sqlite', pool_size=10)
db.define_table('person', Field('name'), Field('age', 'integer'))
db.person.insert(name="Alex", age="19")
query = db.person.name.startswith('A')
rows = db(query).select()
print(rows[0].age) # >>> 19
db.commit()
```

Person

id	name	age
1	Alex	10
...

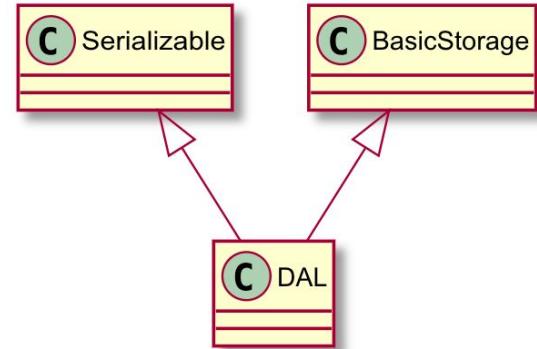
DAL

- Able to access different types of databases
 - SQLite, MySQL, Oracle, MongoDB, PostgreSQL, MSSQL, FireBird, ...

```
from pydal import DAL, Field
db = DAL('sqlite://storage.sqlite', pool_size=10)
db.define_table('person', Field('name'), Field('age', 'integer'))
db.person.insert(name="Alex", age="19")
query = db.person.name.startswith('A')
rows = db(query).select()
print(rows[0].age) # >>> 19
db.commit()
```

```
db = DAL('sqlite://storage.sqlite', pool_size=10)
db = DAL('postgres://username:password@localhost/test', pool_size=10)
```

DAL class



```
class DAL(with_metaclass(MetaDAL, Serializable, BasicStorage)):
```

```
class Serializable(object):
    def as_dict(self, flat=False, sanitize=True):
        return self.__dict__

    def as_xml(self, sanitize=True):
        return serializers.xml(self.as_dict(flat=True, sanitize=sanitize))

    def as_json(self, sanitize=True):
        return serializers.json(self.as_dict(flat=True, sanitize=sanitize))

    def as_yaml(self, sanitize=True):
        return serializers.yaml(self.as_dict(flat=True, sanitize=sanitize))
```

```
class BasicStorage(object):
    def __init__(self, *args, **kwargs):
        return self.__dict__.__init__(*args, **kwargs)

    def __getitem__(self, key):
        return self.__dict__.__getitem__(str(key))

    __setitem__ = object.__setattr__
```

```
db.person.insert(name="Alex", age="19")
db['person'].insert(name="Bob", age="20")
```

parse

```
db = DAL('sqlite://storage.sqlite', pool_size=10)
```

```
if uri:
    uris = isinstance(uri, (list, tuple)) and uri or [uri]
    connected = False
    for k in range(attempts):
        for uri in uris:
            try:
                from .adapters import adapters
                if is_jdbc and not uri.startswith('jdbc:'):
                    uri = 'jdbc:' + uri
                self._dbname = REGEX_DBNAME.match(uri).group()
                # notice that driver_args or {} else driver_args
                # defaults to {} global, not correct
                kwargs = dict(db=self,
                              uri=uri,
                              pool_size=pool_size,
                              folder=folder,
                              db_codec=db_codec,
                              credential_decoder=credential_decoder,
                              driver_args=driver_args or {},
                              adapter_args=adapter_args or {},
                              do_connect=do_connect,
                              after_connection=after_connection,
                              entity_quoting=entity_quoting)
                adapter = adapters.get_for(self._dbname)
                self._adapter = adapter(**kwargs)
            except:
                if connected:
                    break
                if k < attempts - 1:
                    time.sleep(0.1)
                else:
                    raise
            else:
                connected = True
                break
```

```
class Adapters(Dispatcher):
    def register_for(self, *uris):
        def wrap(dispatch_class):
            for uri in uris:
                self._registry_[uri] = dispatch_class
            return dispatch_class
        return wrap

    def get_for(self, uri):
        try:
            return self._registry_[uri]
        except KeyError:
            raise SyntaxError(
                'Adapter not found for %s' % uri
            )

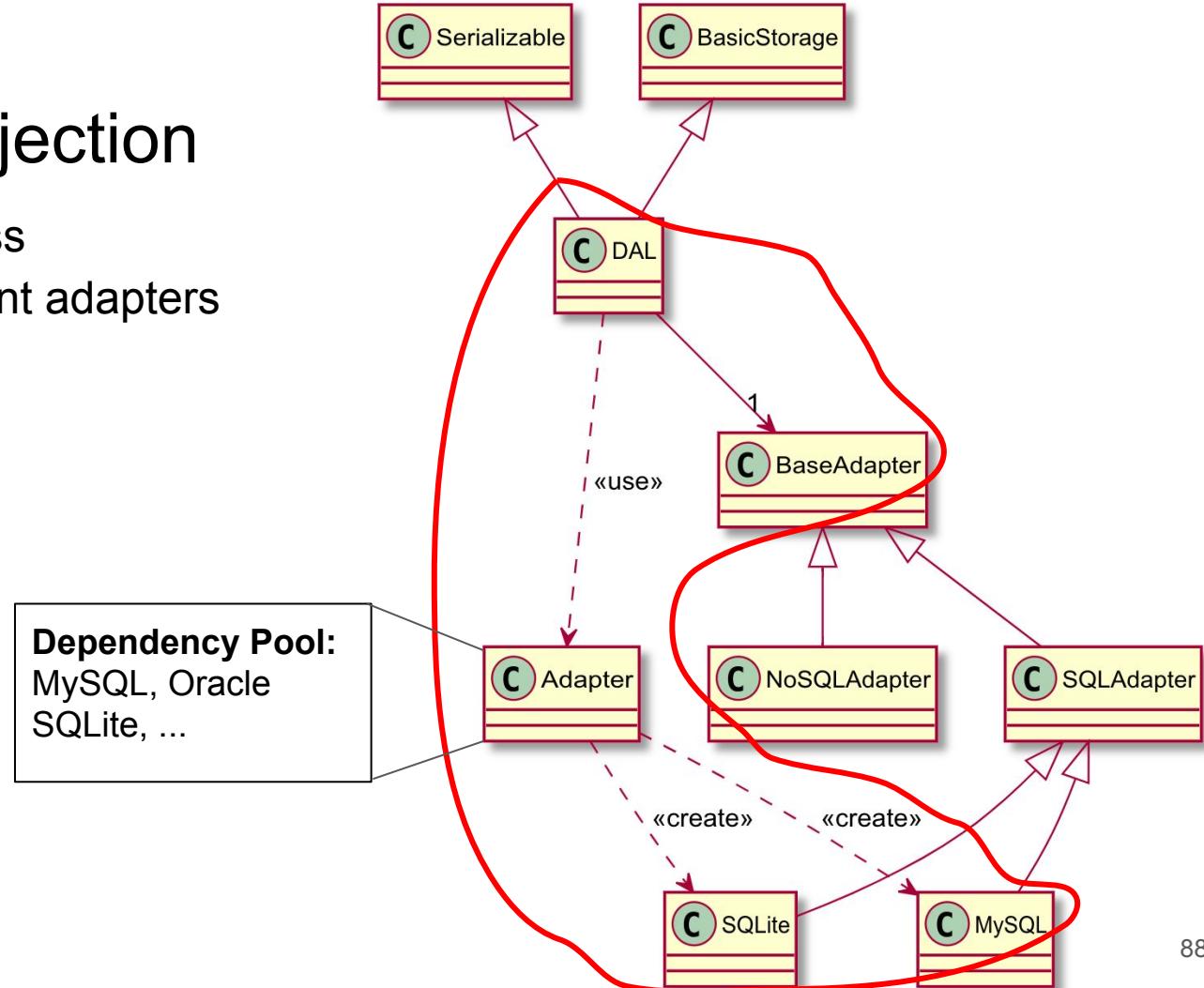
adapters = Adapters('adapters')
```

```
@adapters.register_for('sqlite', 'sqlite:memory')
class SQLite(SQLAdapter):
    database = 'sqlite'
    drivers = ('sqlite2', 'sqlite3')

    def __initialize__(self, do_connect):
```

Dependency Injection

- Inject a SQLite class
- Adapter has different adapters

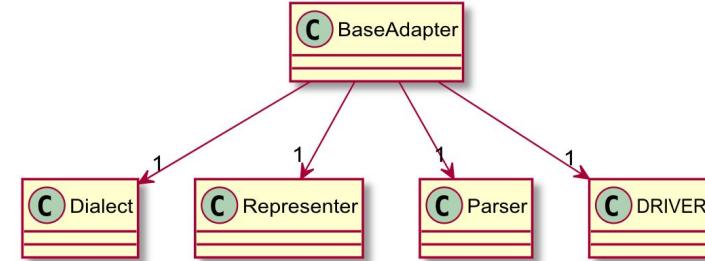


What adapters can do

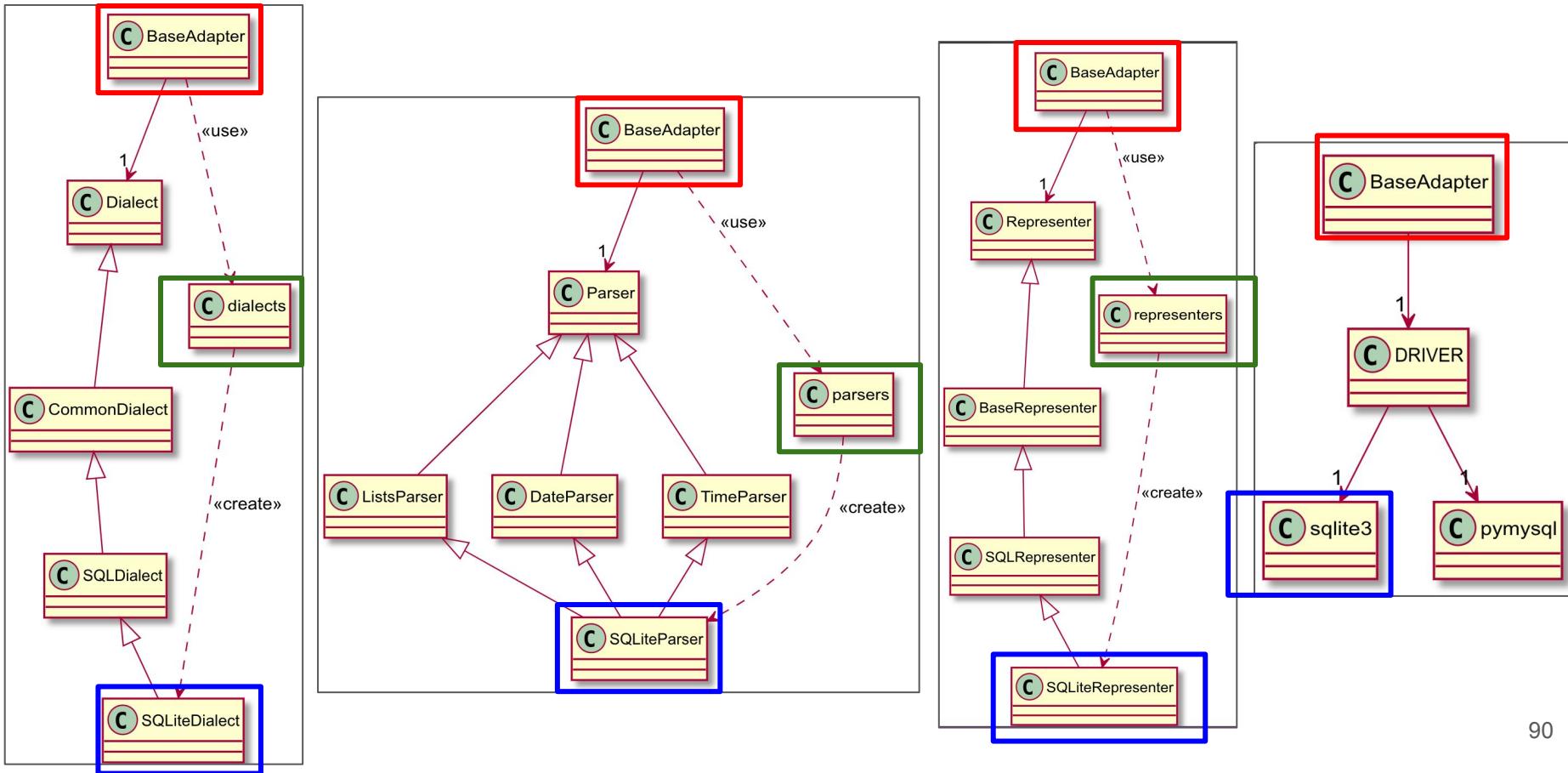
- Translate all fields' operations into its own dialect (SQLiteDialect)
- Connect to the database via specific module (from sqlite3 import dbapi2)
- Parse response string using (if needed) based on its own parser (SQLiteParser)
- Extra query (ST_Geomtry) using its representers (SQLiteRepresenter)

```
| class BaseAdapter(with_metaclass(AdapterMeta, ConnectionPool)):
```

```
def _load_dependencies(self):  
    from ..dialects import dialects  
    from ..parsers import parsers  
    from ..representers import representers  
    self.dialect = dialects.get_for(self)  
    self.parser = parsers.get_for(self)  
    self.representer = representers.get_for(self)
```

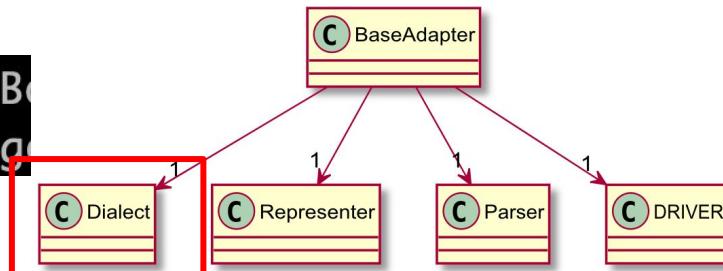


Injection of Dialect, Parser, Representer, Connection module

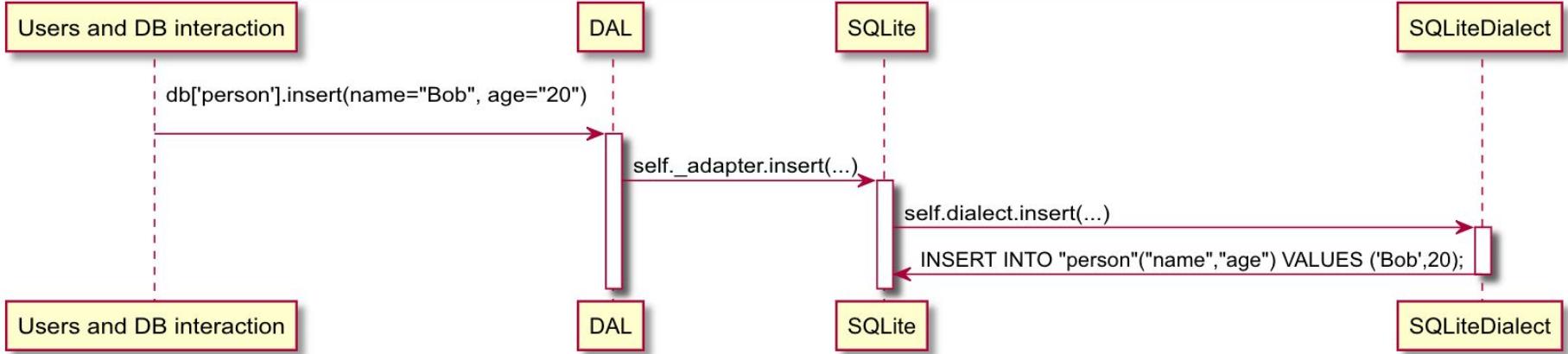


Dialect

```
>>> db['person'].insert(name="Bob", age=20)
INSERT INTO "person"("name", "age") VALUES ('Bob', 20);
```



- Map types and operations into specific strings
- Prepare query strings



```
@sqltype_for('float')
def type_float(self):
    return 'FLOAT'

@sqltype_for('double')
def type_double(self):
    return 'DOUBLE'
```

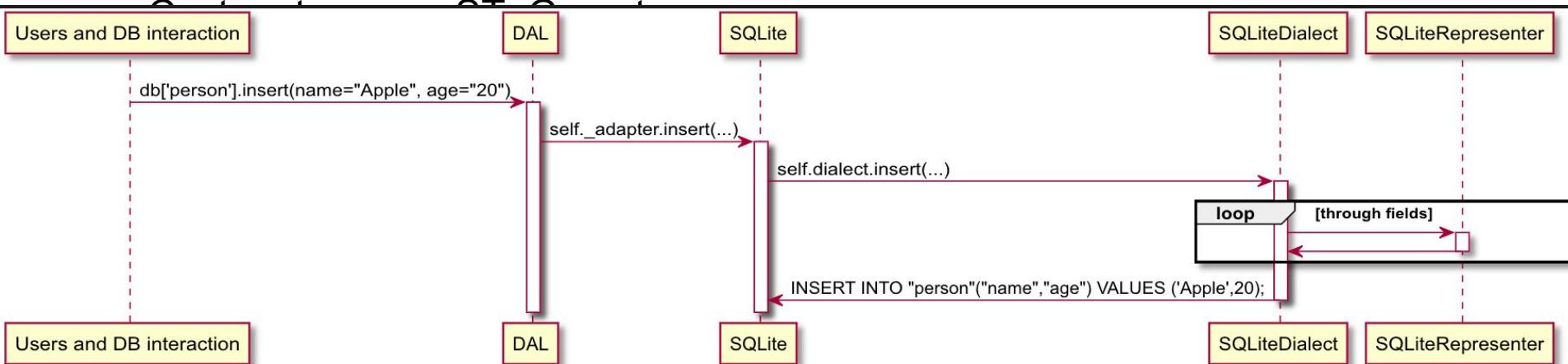
```
        whr = ' %s' % self.where(where)
        return 'UPDATE %s SET %s%s;' % (tablename, values, whr)

    def delete(self, table, where=None):
        tablename = self.writing_alias(table)
        whr = ''
        if where:
            whr = ' %s' % self.where(where)
        return 'DELETE FROM %s%s;' % (tablename, whr)
```

Representer

```
>>> db['person'].insert(name="Apple", age="20")
representer Apple string
representer 20 integer
```

- Type definition (string, int)

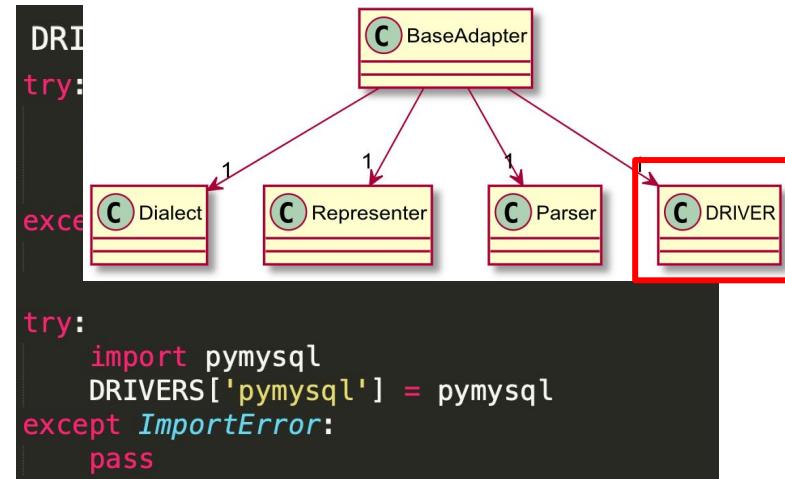


```
@for_type('integer', adapt=False)
def _integer(self, value):
    return str(long(value))
```

```
@for_type('geometry', adapt=False)
def _geometry(self, value, srid):
    return "ST_GeomFromText('%s', %s)" % (value, srid)
```

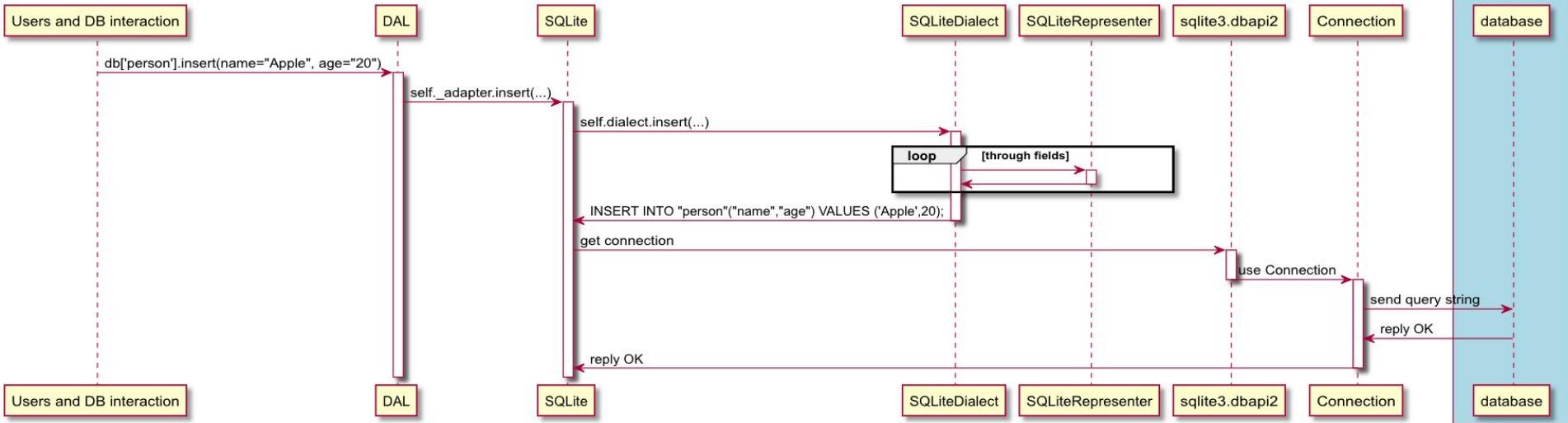
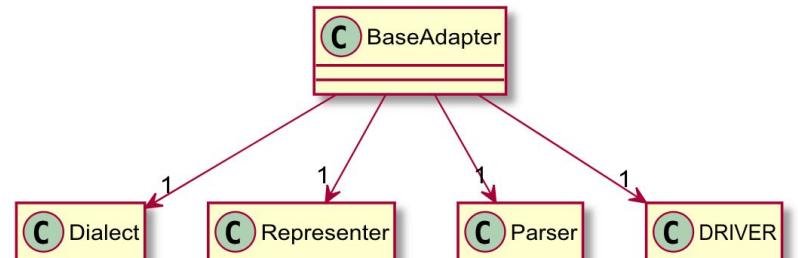
DRIVER

- Connect to database
- Send requests and receive responses



```
class BaseAdapter(with_metaclass(AdapterMeta, ConnectionPool)):  
  
    def find_driver(self):  
        if getattr(self, 'driver', None) is not None:  
            return  
        requested_driver = self._driver_from_uri() or \  
            self.adapter_args.get('driver')  
        if requested_driver:  
            if requested_driver in self._available_drivers:  
                self.driver_name = requested_driver  
                self.driver = self.db._drivers_available[requested_driver]  
            else:  
                raise RuntimeError(  
                    'Driver %s is not available' % requested_driver)
```

Connection



```

...
ret = SQLEndTran(SQL_HANDLE_DBC, selfdbc_h, SQL_COMMIT)
...
def rollback(self):
...
ret = SQLEndTran(SQL_HANDLE_DBC, selfdbc_h, SQL_ROLLBACK)
...
...
  
```

ConnectionPool

- Instantiate a connection beforehand
- Pop up an available worker

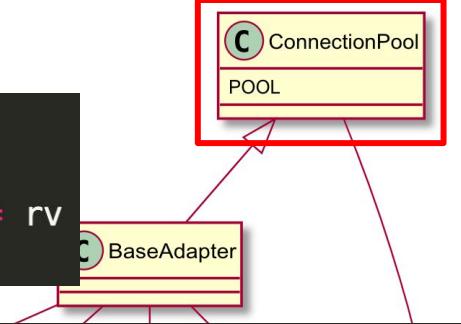
```
class ConnectionPool(object):
    POOLS = {}
    check_active_connection = True

    def __init__(self):
        _iid_ = str(id(self))
        self._connection_thname_ = '_pydal'
        self._cursors_thname_ = '_pydal_cui'

    def _get_or_build_free_cursor(self):
        for handler in itervalues(self.POOLS):
            if handler.available:
                return handler
        return self._build_cursor()
```

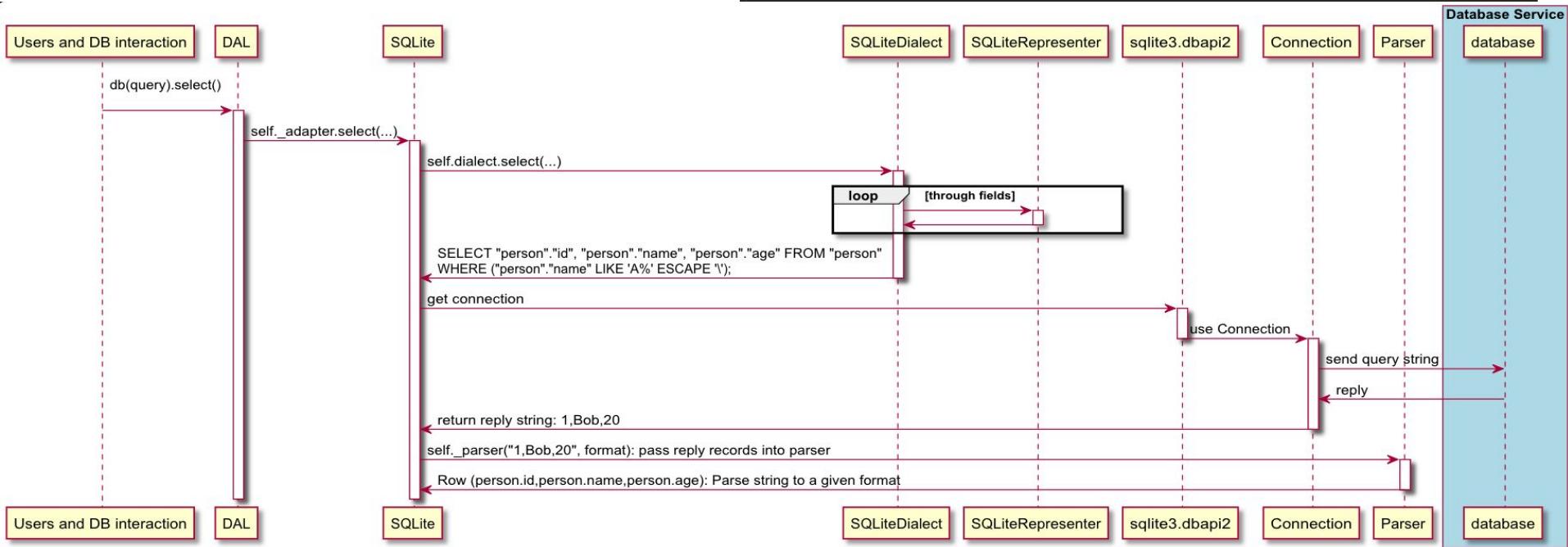
```
def _build_cursor(self):
    rv = Cursor(self.connection)
    self.cursors[id(rv.cursor)] = rv
    return rv
```

```
# The Cursor Class.
class Cursor:
    def __init__(self, conx, row_type_callable=None):
        """ Initialize self.stmt_h, which is the handle of a statement
        A statement is actually the basis of a python "cursor" object
        """
        self.stmt_h = ctypes.c_void_p()
        self.connection = conx
        self.ansi = conx.ansi
        self.row_type_callable = row_type_callable or TupleRow
        self.statement = None
        self._last_param_types = None
        self._ParamBufferList = []
        self._ColBufferList = []
        self._row_type = None
        self._buf_cvt_func = []
        self.rowcount = -1
        self.description = None
        self.autocommit = None
        self._ColTypeCodeList = []
        self._outputsizes = {}
        self._inputsizers = []
        self.arraysize = 1
        ret = ODBC_API.SQLAllocHandle(SQL_HANDLE_STMT, self.connection.dbc_h, ADDR)
        check_success(self, ret)
```



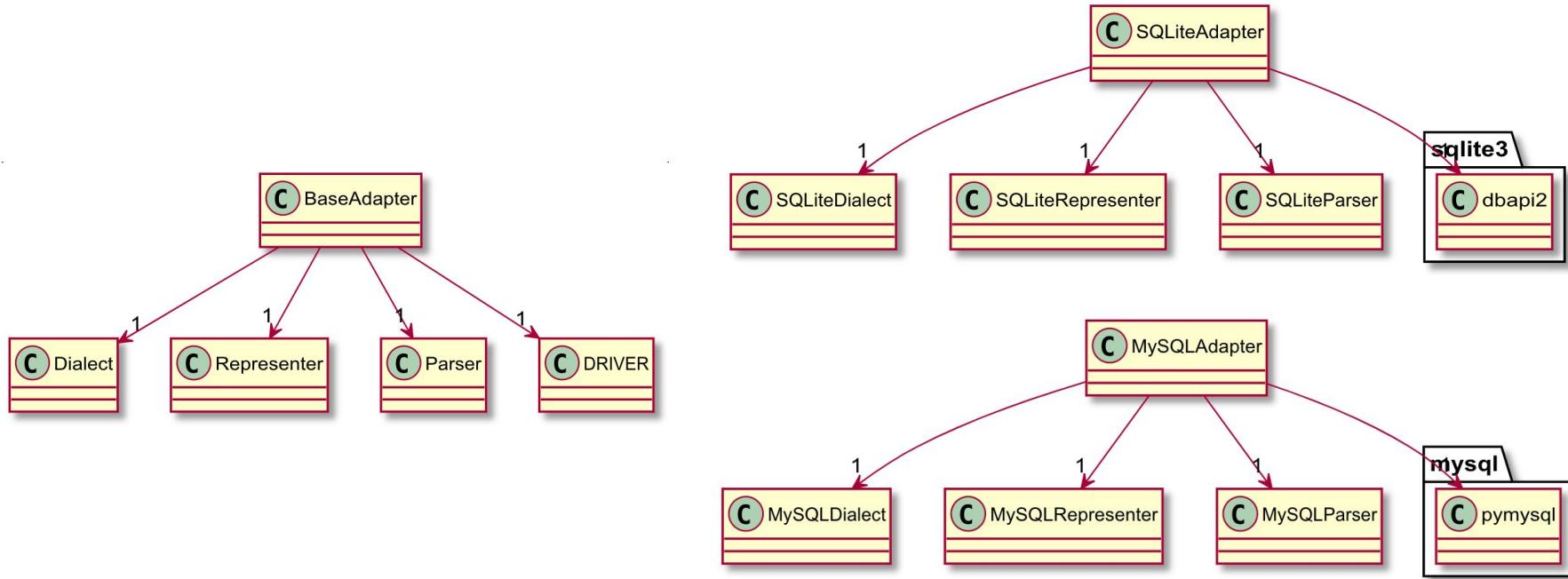
Parser

```
class BasicParser(Parser):
    @for_type('id')
    def _id(self, value):
        return long(value)
```

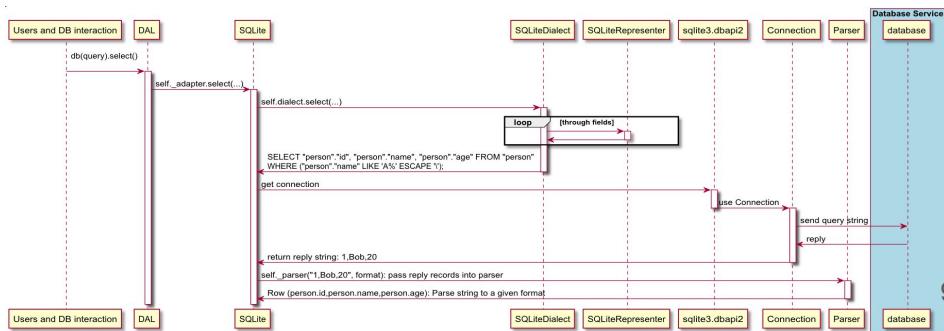
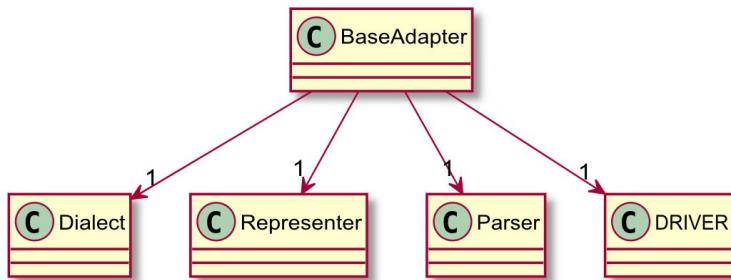
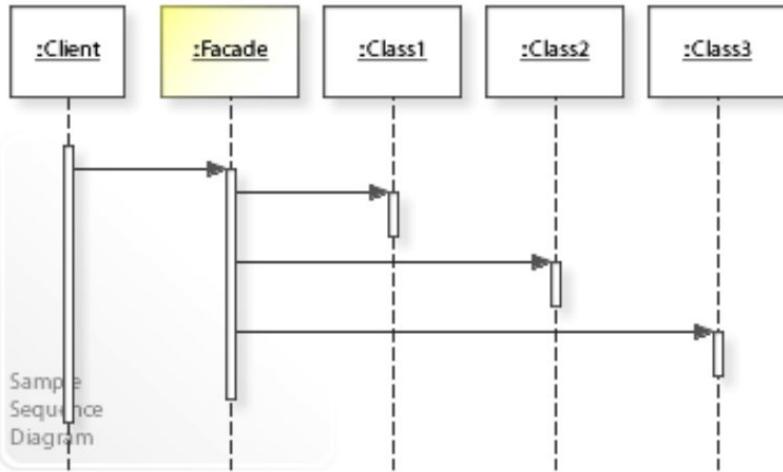
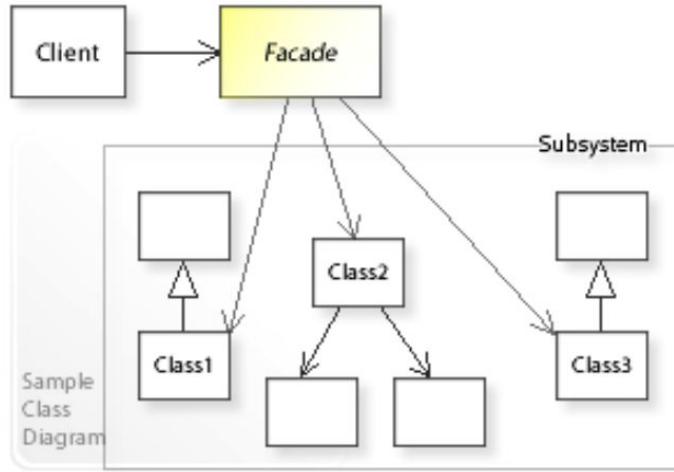


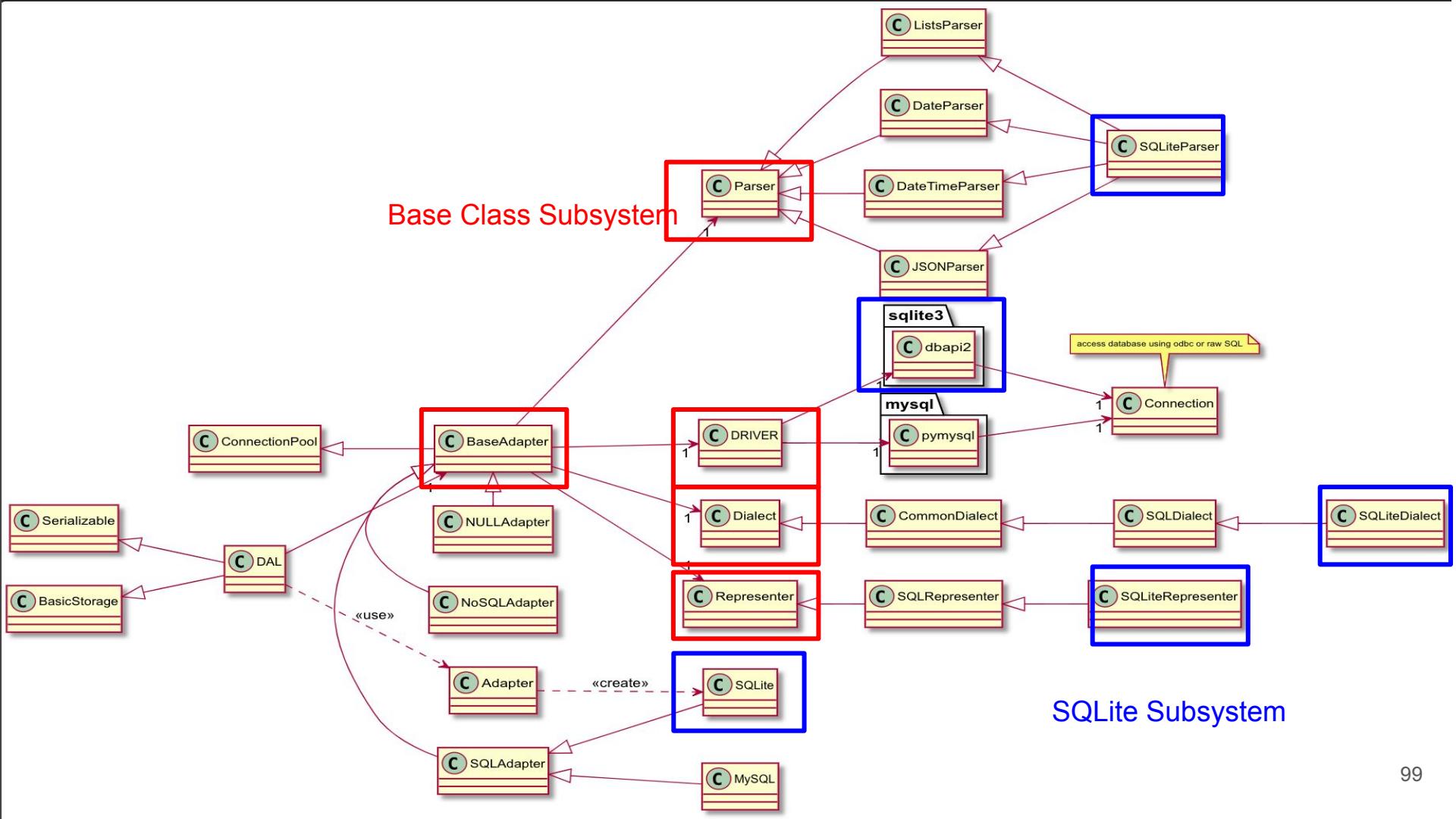
```
    fields_virtual, fields_lazy)
for row in rows
]
rowsobj = self.db.Rows(self.db, new_rows, colnames, rawrows=rows,
                      fields=fields)
print(rowsobj, 'here')
```

Subsystem



Facade





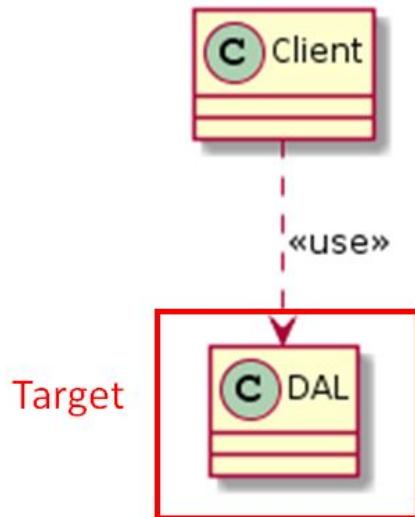
Database Abstraction Layer (DAL)

```
289         db.define_table(
290             settings.table_user_name,
291             Field('first_name', length=128, default='',
292                   label=self.messages.label_first_name,
293                   requires=is_not_empty),
294             Field('last_name', length=128, default='',
295                   label=self.messages.label_last_name,
296                   requires=is_not_empty),
297             Field('email', length=512, default='',
298                   label=self.messages.label_email,
299                   requires=is_unique_email),
300             Field('username', length=128, default='',
301                   label=self.messages.label_username,
302                   requires=is_unique_username),
303             Field(passfield, 'password', length=512,
304                   readable=False, label=self.messages.label_password,
305                   requires=[is_crypted]),
306             Field('registration_key', length=512,
307                   writable=False, readable=False, default='',
308                   label=self.messages.label_registration_key),
309             Field('reset_password_key', length=512,
310                   writable=False, readable=False, default='',
311                   label=self.messages.label_reset_password_key),
312             Field('registration_id', length=512,
313                   writable=False, readable=False, default='',
314                   label=self.messages.label_registration_id),
315             *extra_fields,
316             **dict(
317                 migrate=self._get_migrate(settings.table_user_name,
318                                         migrate),
319             fake_migrate=fake_migrate,
320             format='%(username)s'))
```

Maps Python objects into database objects

```
185         while True:
186             ordered = set(item for item, dep in nested_dict.items() if not dep)
187             if not ordered:
188                 break
189             rtn.append(ordered)
190             nested_dict = dict(
191                 (item, (dep - ordered)) for item, dep in nested_dict.items()
192                 if item not in ordered
193             )
194             assert not nested_dict, "A cyclic dependency exists amongst %r" % nested_dict
195             db.commit()
196             return rtn
197         except:
198             db.rollback()
199             return None
```

Target



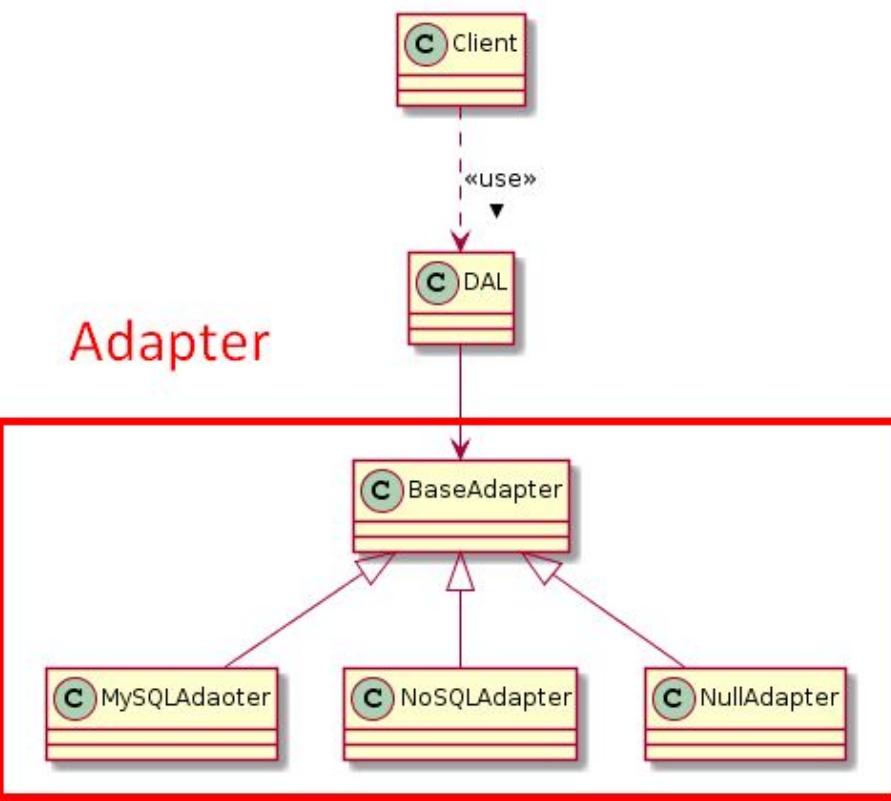
```
chris@chris:~$ python3
Python 3.4.3 (default, Nov 12 2018, 22:25:49)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pydal
>>> from pydal import DAL
>>> from pydal import Field
>>> db = DAL('sqlite://storage.db', pool_size=6)
>>> db.define_table('person', Field('name'), format='%(name)s')
<Table person (id, name)>
>>> db.person.insert(name="Alex")
1
>>> db.commit()
>>> db.rollback()
>>> █
```

Initialization

```
class DAL(with_metaclass(MetaDAL, Serializable, BasicStorage)):

    def __init__(self, uri='sqlite://dummy.db',
                 pool_size=0, folder=None,
                 db_codec='UTF-8', check_reserved=None,
                 migrate=True, fake_migrate=False,
                 migrate_enabled=True, fake_migrate_all=False,
                 decode_credentials=False, driver_args=None,
                 adapter_args=None, attempts=5, auto_import=False,
                 bigint_id=False, debug=False, lazy_tables=False,
                 db_uid=None, do_connect=True,
                 after_connection=None, tables=None, ignore_field_case=True,
                 entity_quoting=True, table_hash=None):
```

Adapter



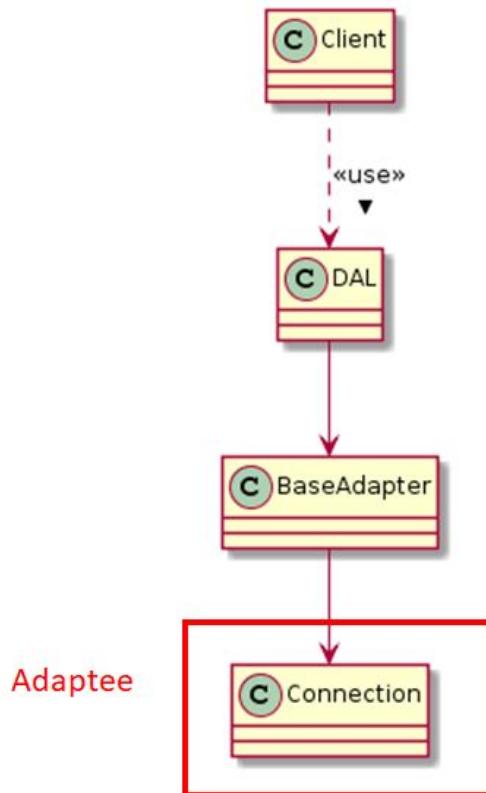
```
kwargs = dict(db=self,
              uri=uri,
              pool_size=pool_size,
              folder=folder,
              db_codec=db_codec,
              credential_decoder=credential_decoder,
              driver_args=driver_args or {},
              adapter_args=adapter_args or {},
              do_connect=do_connect,
              after_connection=after_connection,
              entity_quoting=entity_quoting)
adapter = adapters.get_for(self._dbname)
self._adapter = adapter(**kwargs)
```

```
def commit(self):
    self._adapter.commit()

def rollback(self):
    self._adapter.rollback()

def close(self):
    self._adapter.close()
    if self._db_uid in THREAD_LOCAL._pydal_db_instances_:
        db_group = THREAD_LOCAL._pydal_db_instances_[self._db_uid]
        db_group.remove(self)
        if not db_group:
            del THREAD_LOCAL._pydal_db_instances_[self._db_uid]
    self._adapter._clean_locals()
```

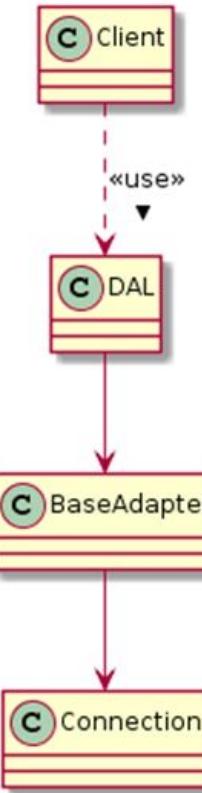
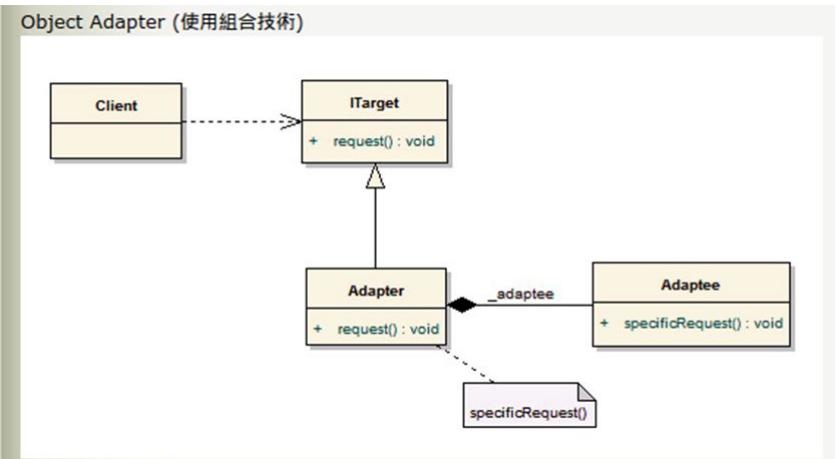
Adaptee



```
@with_connection  
def commit(self):  
    return self.connection.commit()  
  
@with_connection  
def rollback(self):  
    return self.connection.rollback()  
  
@with_connection  
def prepare(self, key):  
    self.connection.prepare()  
  
@with_connection  
def commit_prepared(self, key):  
    self.connection.commit()  
  
@with_connection  
def rollback_prepared(self, key):  
    self.connection.rollback()
```

Adapter Pattern

Object Adapter



Table, Row, Field

- PyDAL has classes like
 - Table
 - Row
 - Field

Person		
id	name	age
1	Alex	10
...

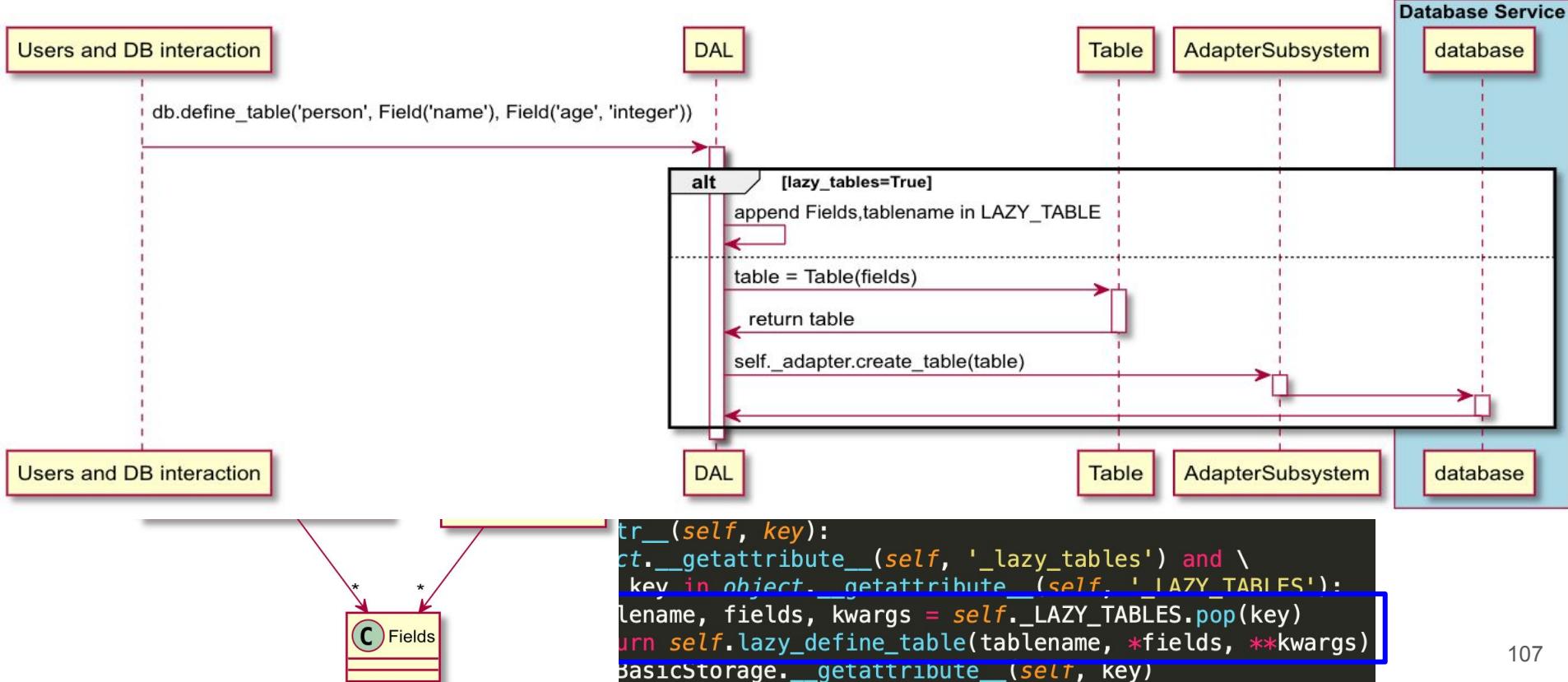
```
from pydal import DAL, Field
db = DAL('sqlite://storage.sqlite', pool_size=10)
db.define_table('person', Field('name'), Field('age', 'integer'))
db.person.insert(name="Alex", age=19)
query = db.person.name.startswith('A')
rows = db(query).select()
print(rows[0].age) # >>> 19
db.commit()
```

```

class DAL(with_metaclass(MetaDAL, Serializable, BasicStorage)):
    def define_table(self, tablename, *fields, **kwargs):
        if self._lazy_tables:
            if tablename not in self.LAZY_TABLES or redefine:
                self.LAZY_TABLES[tablename] = (tablename, fields, kwargs)

```

Define Table



Get records

- Parser returns includes lots of

```
def parse(self, rows, fields, colnames, blob_decode=True, cacheable=False):
    (fields_virtual, fields_lazy, tmpls) = \
        self._parse_expand_colnames(fields)
    new_rows = [
        self._parse(
            row, tmpls, fields, colnames, blob_decode, cacheable,
            fields_virtual, fields_lazy)
        for row in rows
    ]
    rowsobj = self.db.Rows(self.db, new_rows, colnames, rawrows=rows,
                           fields=fields)
    print(rowsobj, 'here')
```

```
from pydal import DAL, Field
db = DAL('sqlite://storage.sqlite', pool_size=10)
db.define_table('person', Field('name'), Field('age', 'integer'))
db.person.insert(name="Alex", age="19")
query = db.person.name.startswith('A')
rows = db(query).select()
print(rows[0].age) # >>> 19
db.commit()
```

Iterate records

BasicRows and IterRows

```
class Rows(BasicRows):
    def __init__(self, db=None, records=[], rawrows=None, fields=[]):
        self.db = db
        self.records = records
        self.fields = fields
        self.colnames = colnames
        self.compact = compact
        self.response = rawrows
```

```
def __iter__(self):
```

Iterator over records

....

```
for i in xrange(len(self)):
    yield self[i]
```

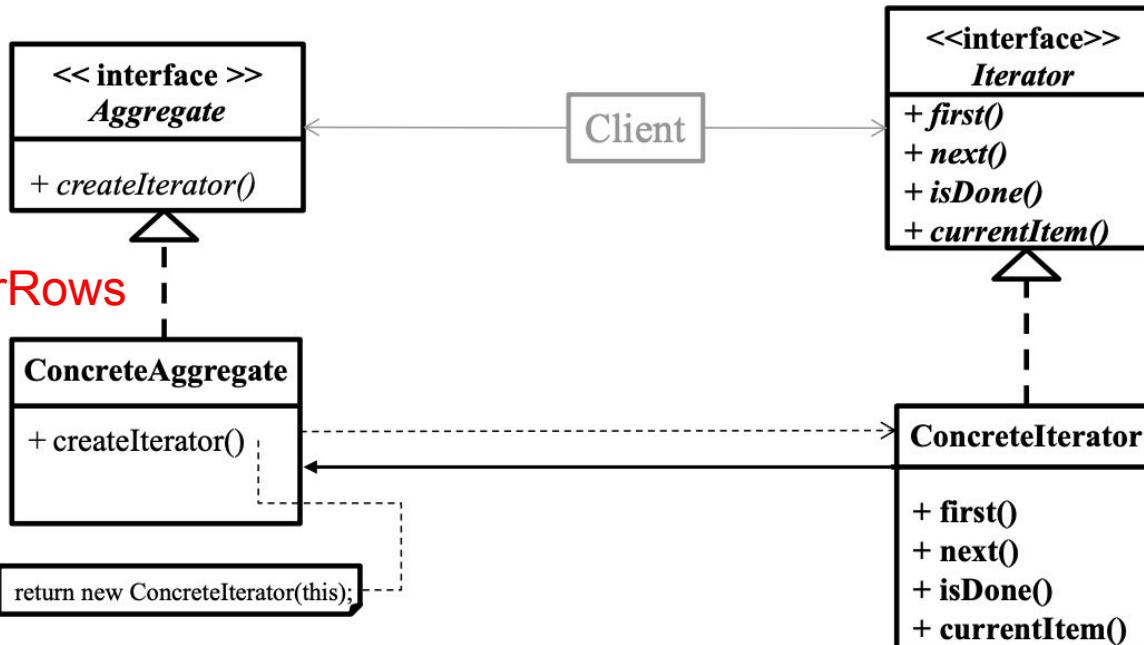
```
class IterRows(BasicRows):
```

```
def __iter__(self):
    if self._head:
        yield self._head
    try:
        row = next(self)
        while row is not None:
            yield row
            row = next(self)
    except StopIteration:
        # Iterator is over, adjust the cursor logic
        self.db._adapter.close_cursor(self.cursor)
        return
    return

def next_(self):
    db_row = self.cursor.fetchone()
    print(db_row)
    if db_row is None:
        raise StopIteration
    row = self.db._adapter._parse(db_row, self.tmps, self.fields,
                                  self.colnames, self.blob_decode,
                                  self.cacheable, self.fields_virtual,
                                  self.fields_lazy)
    if self.compact:
        # The following is to translate
        # <Row {'t0': {'id': 1L, 'name': 'web2py'}>
        # in
        # <Row {'id': 1L, 'name': 'web2py'}>
        # normally accomplished by Rows.__getitem__
        keys = list(row.keys())
        if len(keys) == 1 and keys[0] != '_extra':
            row = row[keys[0]]
    return row
```

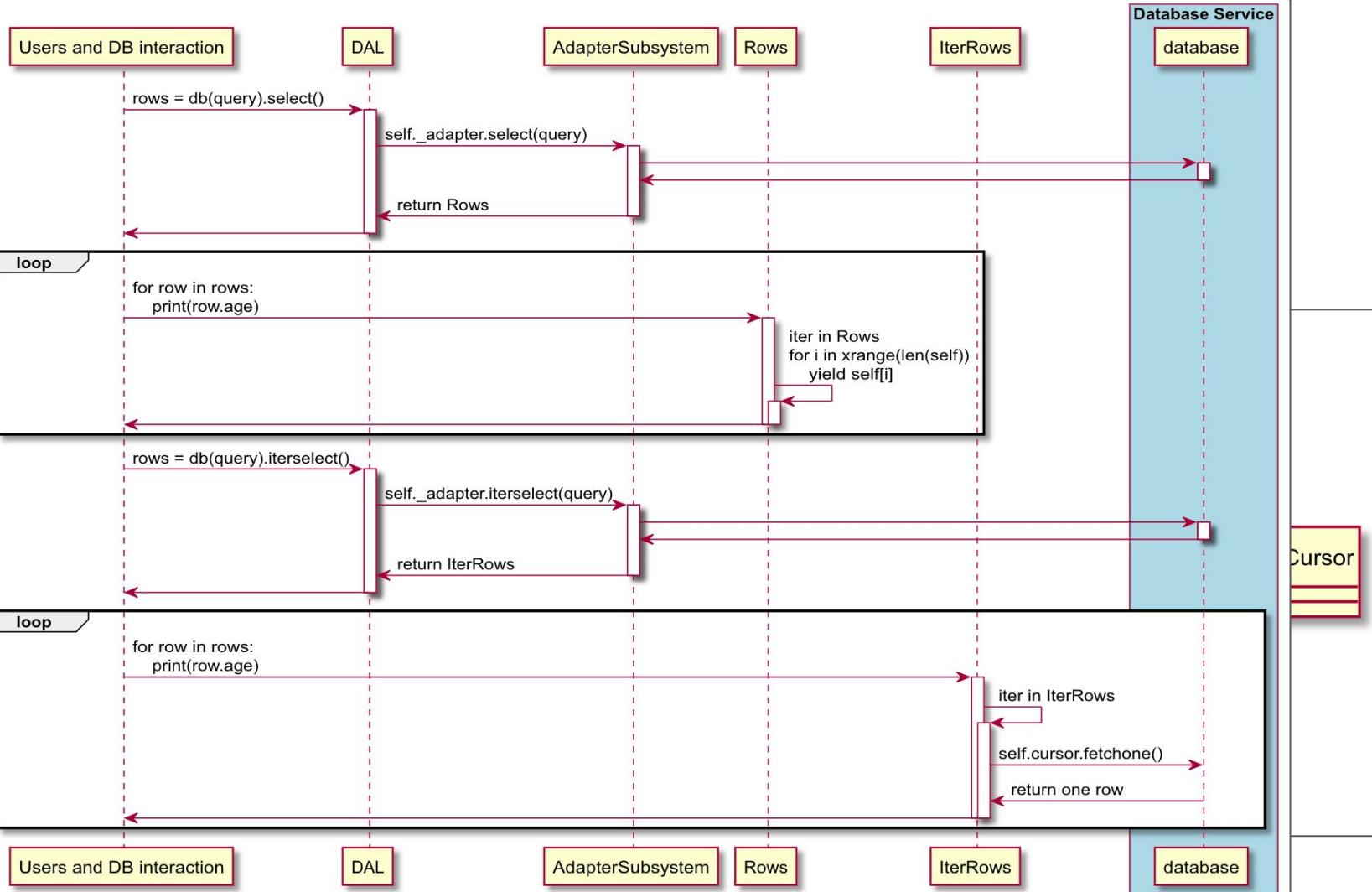
Iterator Pattern Structure₁

BasicRows



Rows / IterRows

from
db =
db.d
db.p
db.p
db.p
quer
ha
rows
ho
rows
for
for



PyDAL vs ORM

Person

id	name	age
1	Alex	10
...

```
class Person {  
    id,  
    name,  
    age  
}
```

Parse return string to **Row**
Aggregate all rows to **Rows**

PyDAL conclusion

- Use adapter to connect DAL with database
- Use dependency injection to inject different adapters
- Has connection pool to hold initialized connections
- Lazy table mechanism
- Different iteration implementation
- Not ORM

Conclusion (web2py)

- Model 2 architecture
- Most of design are **initial design**, web2py is hard to maintain. Breaking down the system into definable elements is difficult because it is not loosely coupled
- Provides some interesting mechanisms with Python
e.g. dynamically typed, function decorators, Storage

Conclusion (SED)

- Python及Java的特性差異使得design pattern不全然相同
- Code design 的重要性，讓程式有更好的維護性
- 團隊合作，共同完成程式碼的實作
- Web framework 的流程與設計有更多的了解
- Design process for change 的觀念可以套在各種情況
- 對人生有不一樣的領悟與啟發

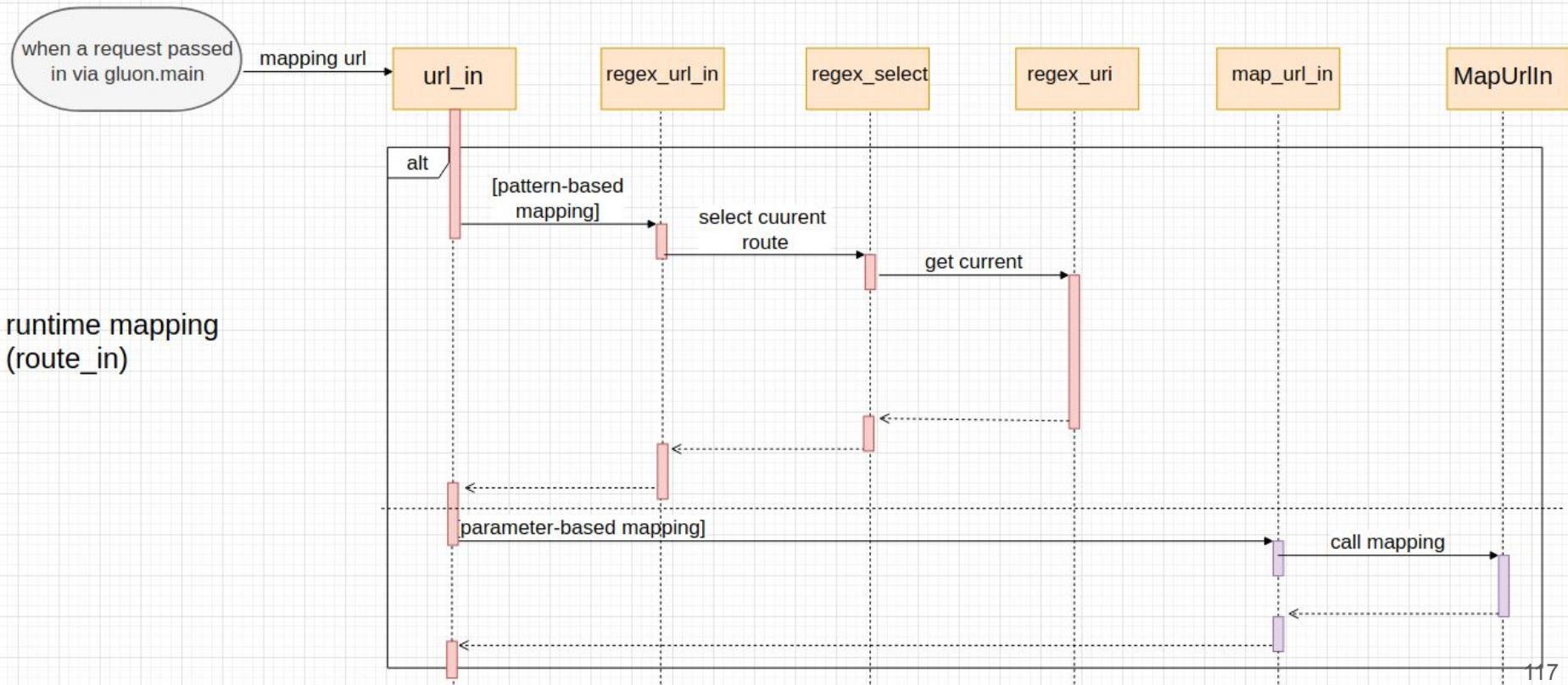
Merry Christmas !!!



Thank you



Appendix



Appendix

