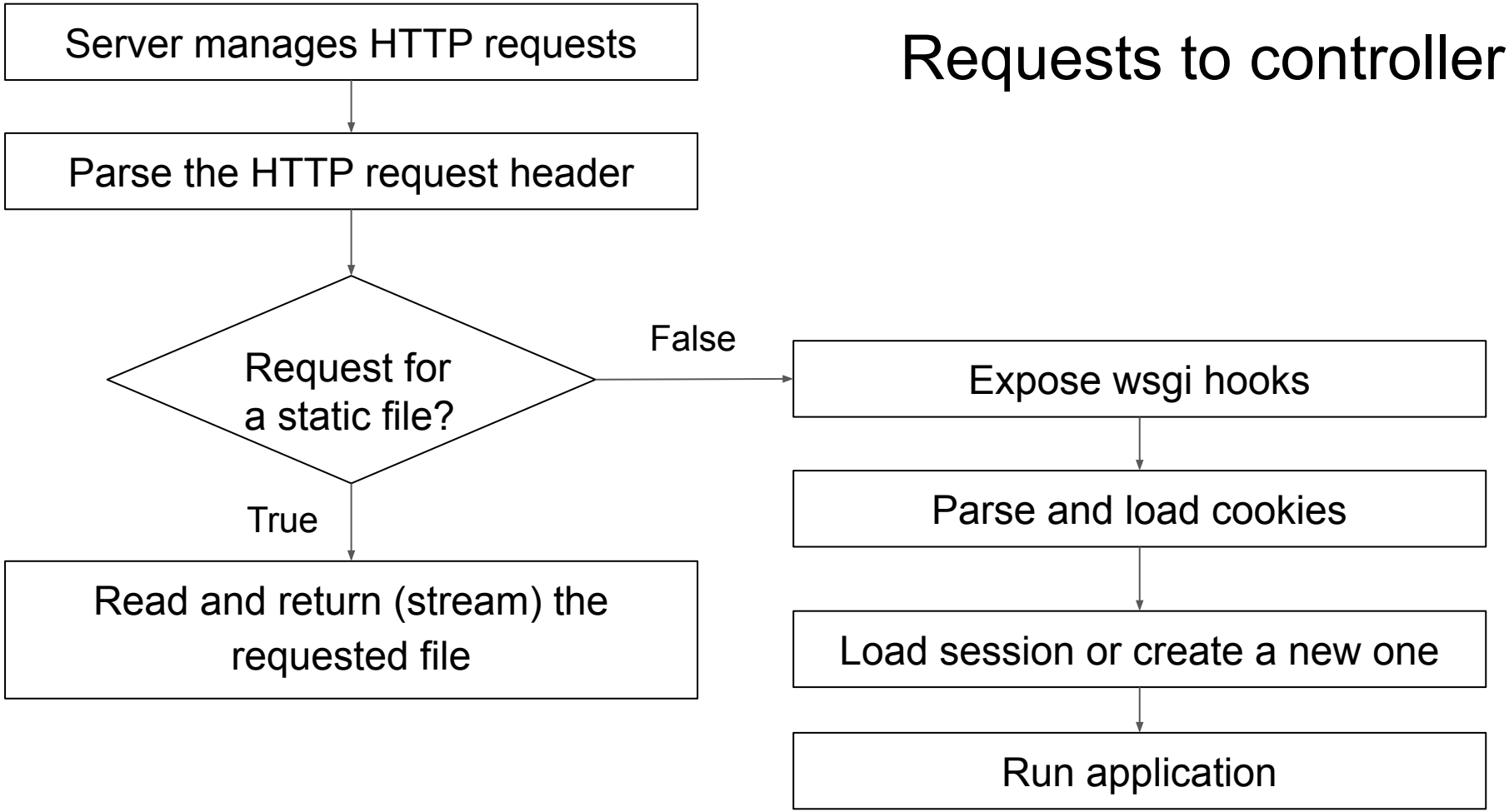
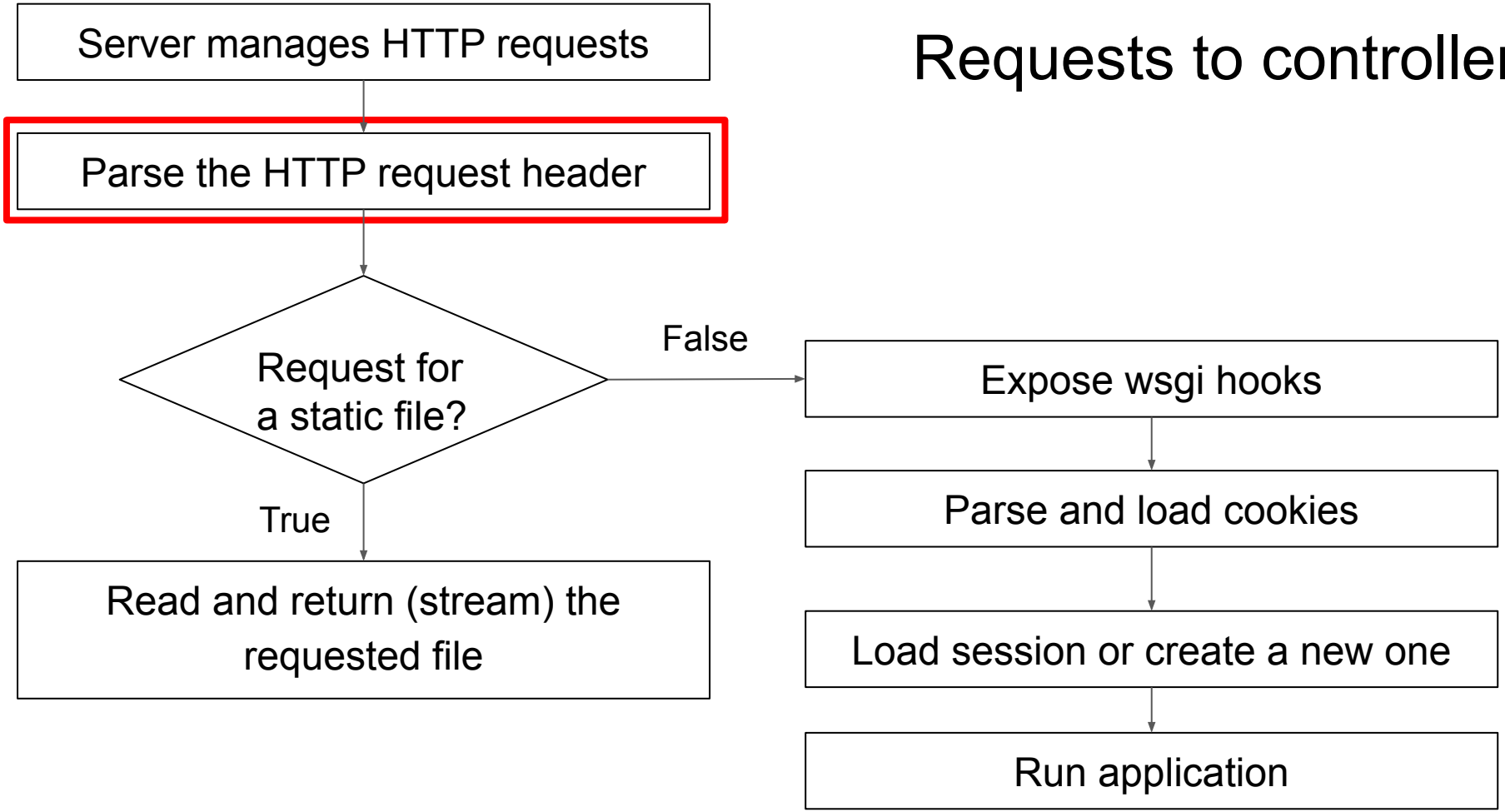
SED1219

Requests to controller



Requests to controller



Rewrite and parse incoming URL

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

Rewrite and parse incoming URL (cont.)

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

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

```
http://domain.com/myapp/default/myapp
http://domain.com/myapp/myapp/index
```

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

Rewrite and parse incoming URL (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>'),
)
```

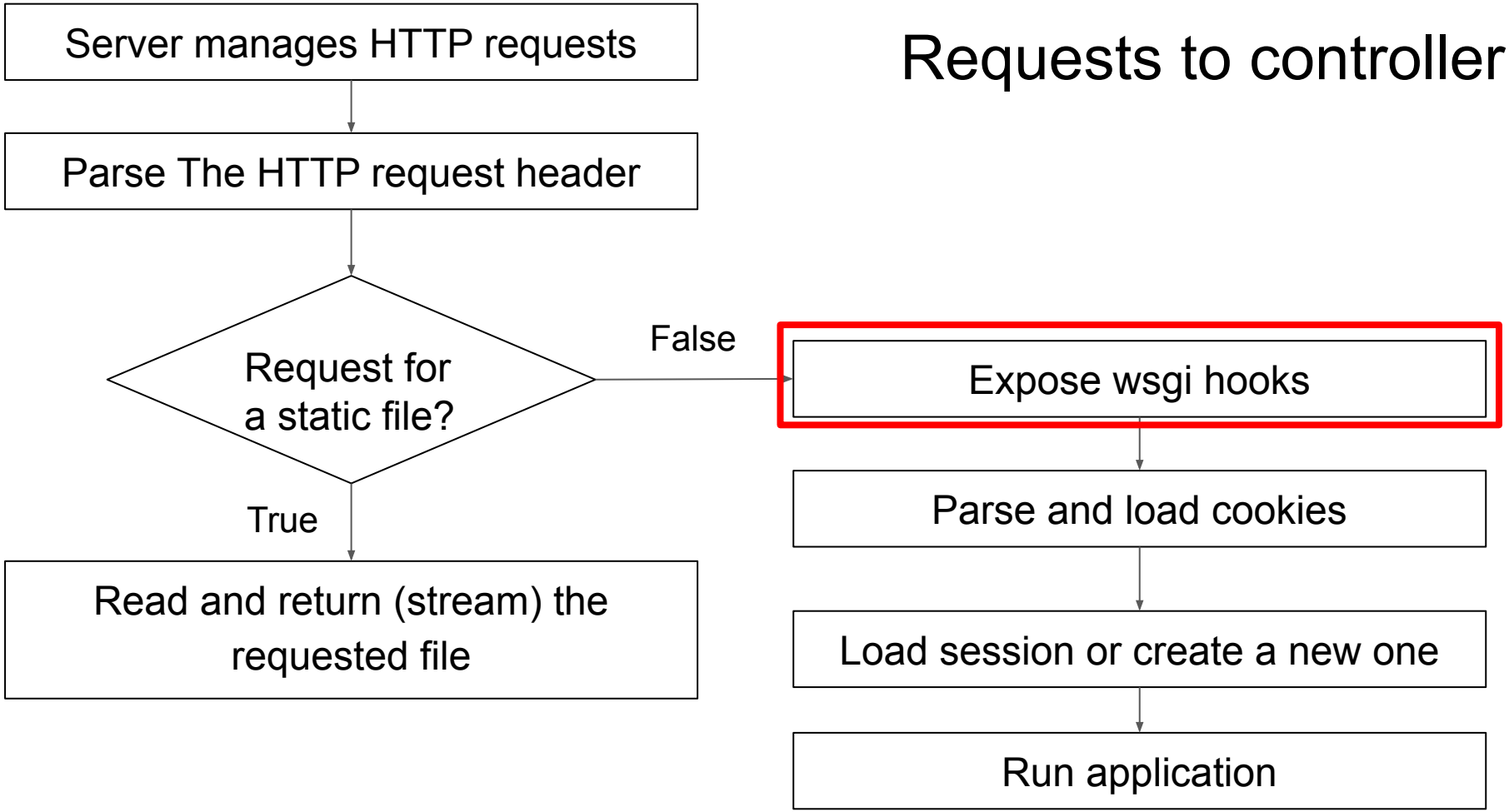
Get rid of the application prefix from the URLs

Rewrite and parse incoming URL

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

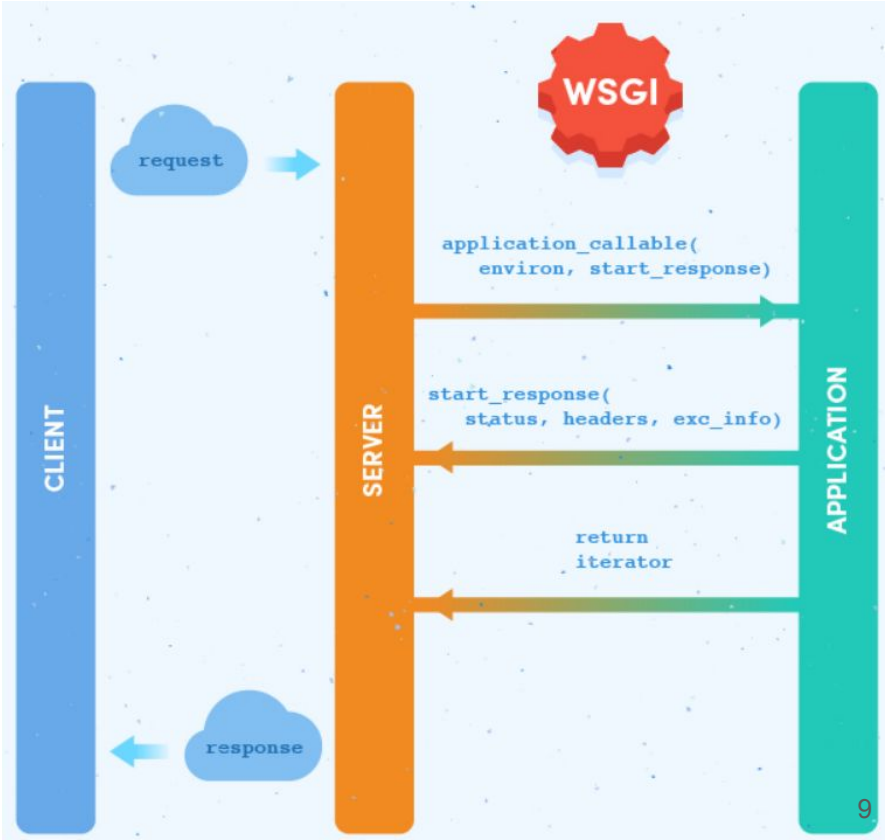
```
request.application = match.group('a') or routes.default_application
request.controller = match.group('c') or routes.default_controller
request.function = match.group('f') or routes.default_function
request.raw_extension = match.group('e')
request.extension = request.raw_extension or 'html'
```

Requests to controller



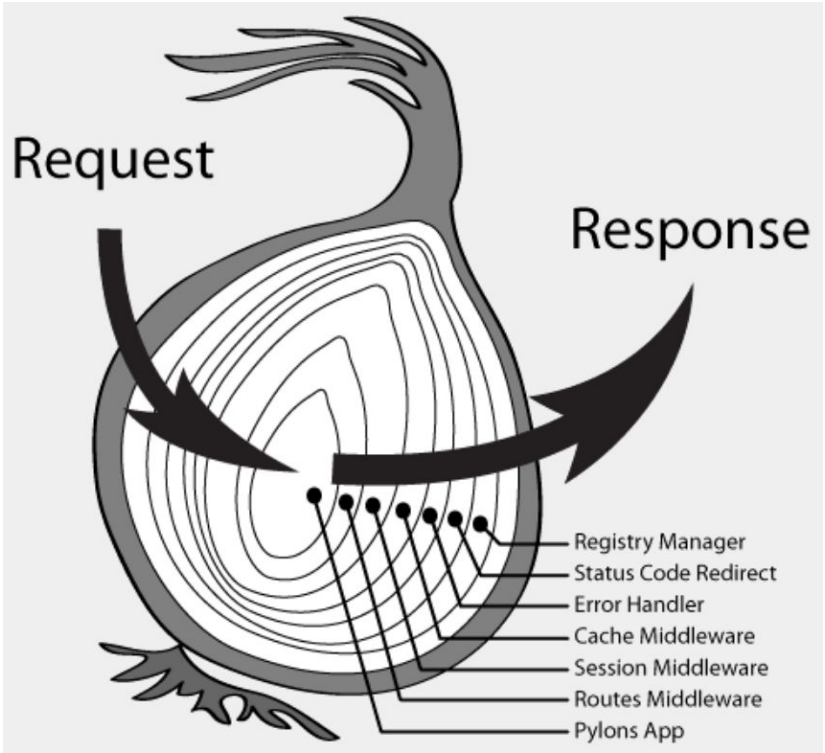
WSGI (Web Server Gateway Interface)

- A simple **calling convention** for web servers to forward requests to web applications or frameworks written in the Python programming language
- Between the server and the application, there may be one or more WSGI middleware components, which implement both sides of the API, typically in Python code



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 we 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).
```

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 actions with WSGI middleware

```
# ################################################
# expose wsgi hooks for convenience
# ################################################

request.wsgi = LazyWSGI(environ, request, response)
```

  -

```
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):…
```

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'
```

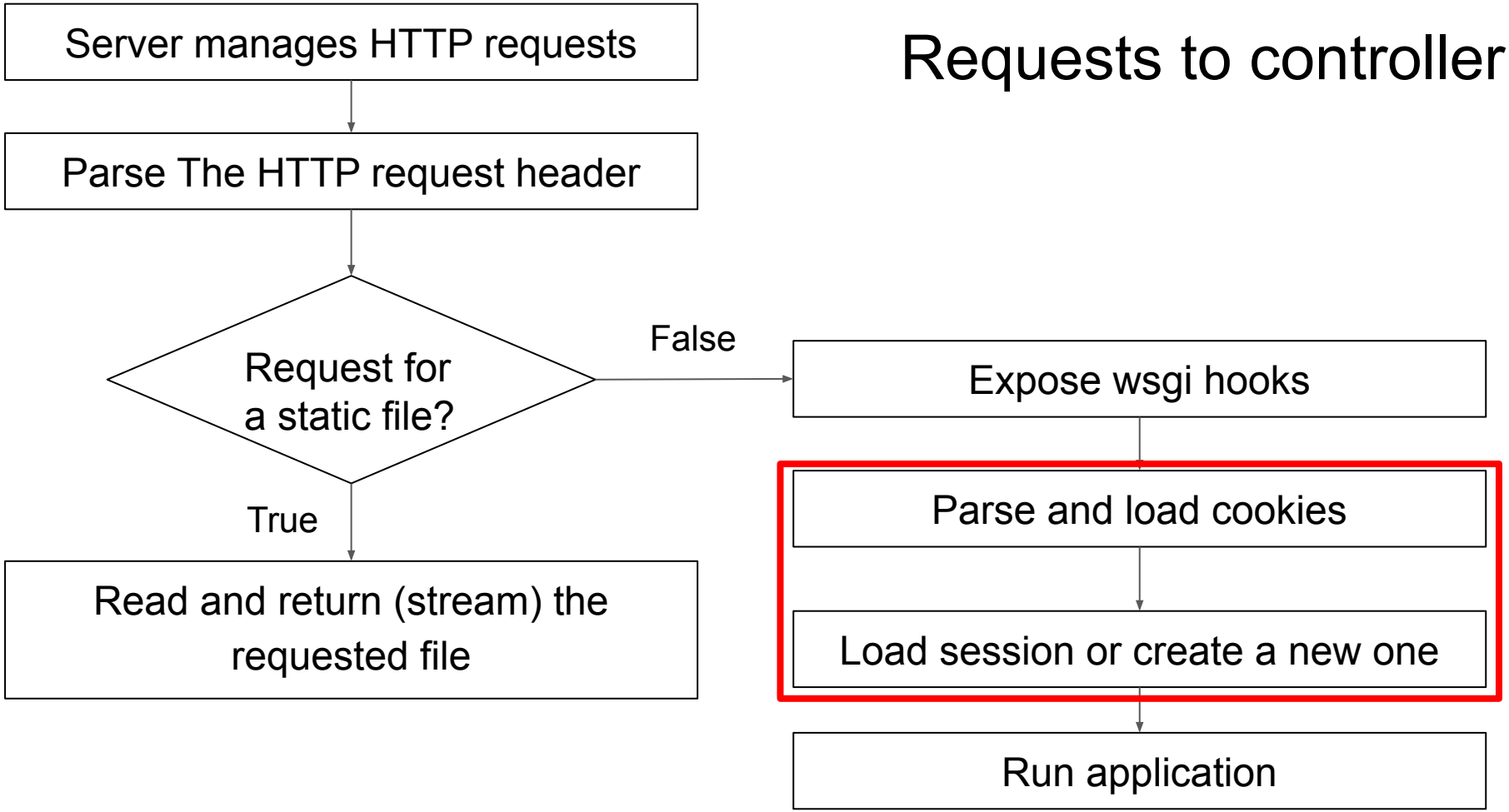
Third party WSGI applications and middleware (cont.)

- 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 outpu
t"""
    items = test_wsgi_app(request.wsgi.environ,
                          request.wsgi.start_response)
    for item in items:
        response.write(item, escape=False)
    return response.body.getvalue()
```

Requests to controller



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 retrieve the session.id

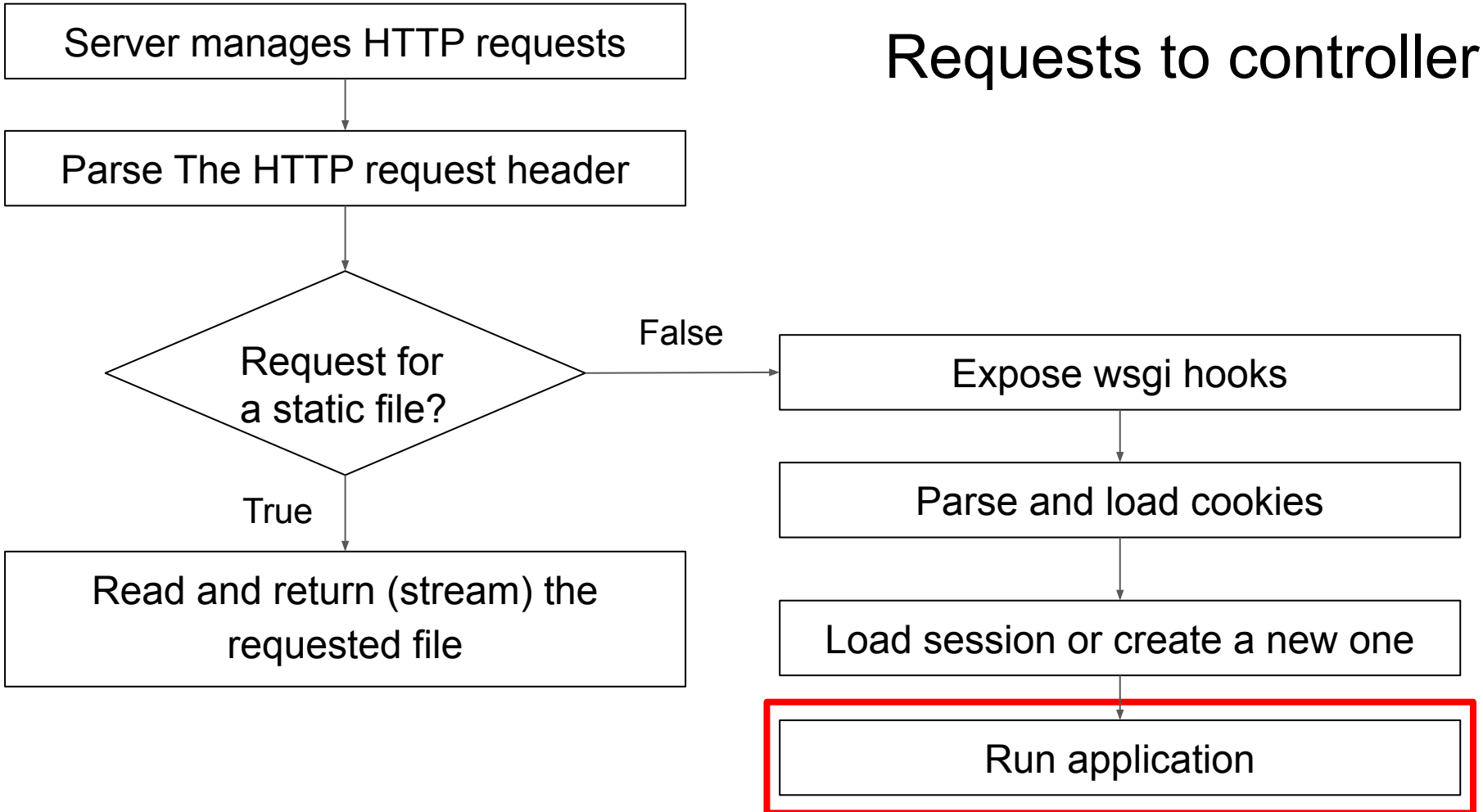
```
if env.http_cookie:
    for single_cookie in env.http_cookie.split(';'):
        single_cookie = single_cookie.strip()
        if single_cookie:
            try:
                request.cookies.load(single_cookie)
            except Cookie.CookieError:
                pass  # single invalid cookie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

Load cookies

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

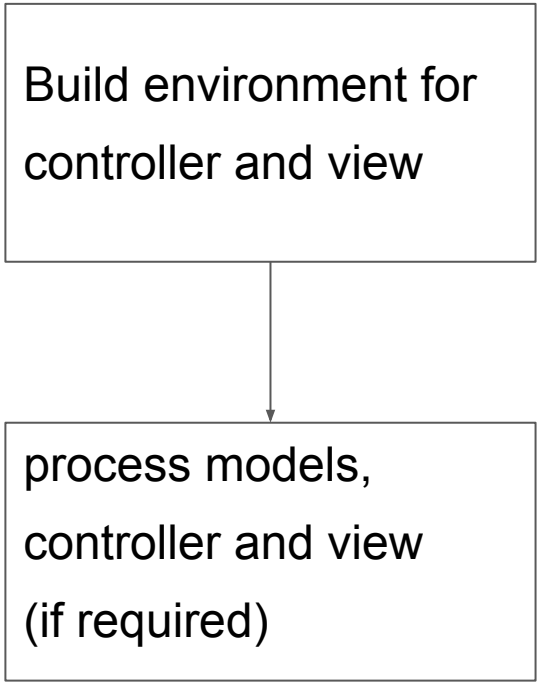
Try load session or create new session file

Requests to controller



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.



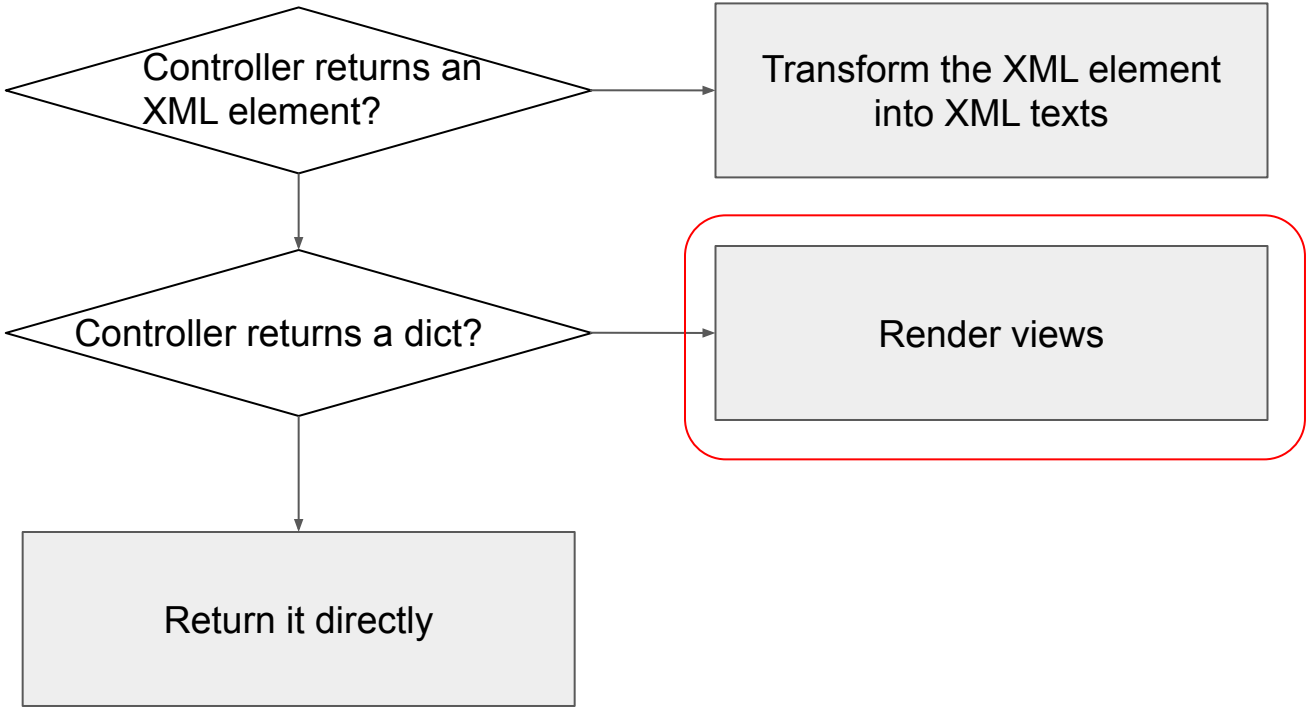
Run Application (cont.)

- 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)
```

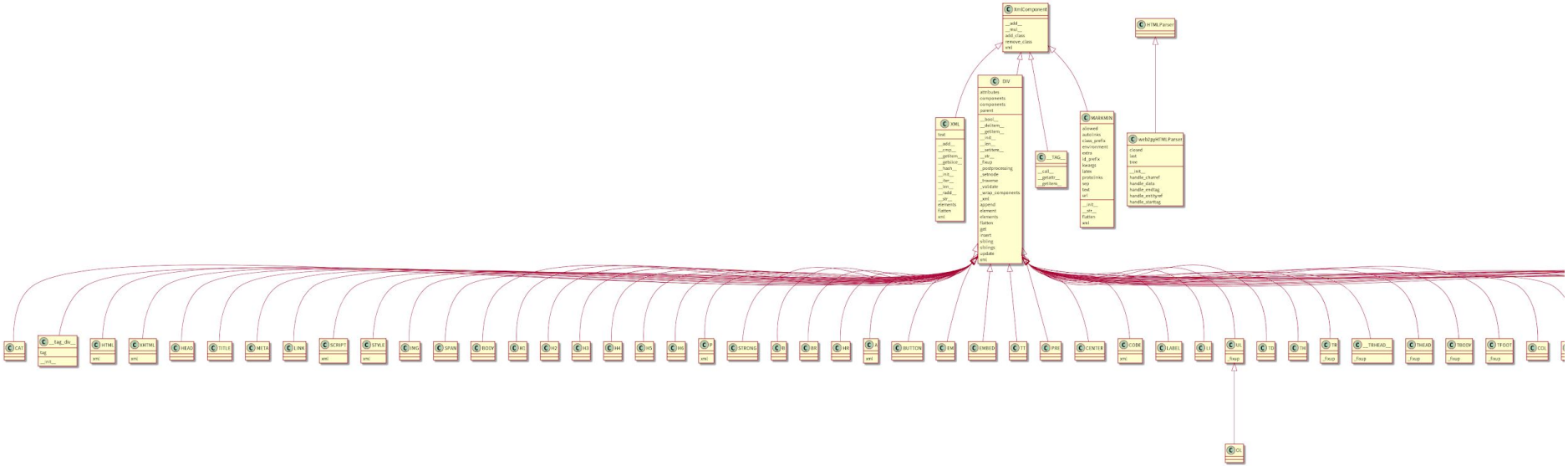
Views

How the Client Gets Data



XML Elements: Helpers for HTML Tag

- SPAN("World") -> World



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

Example Templates

layout.html

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

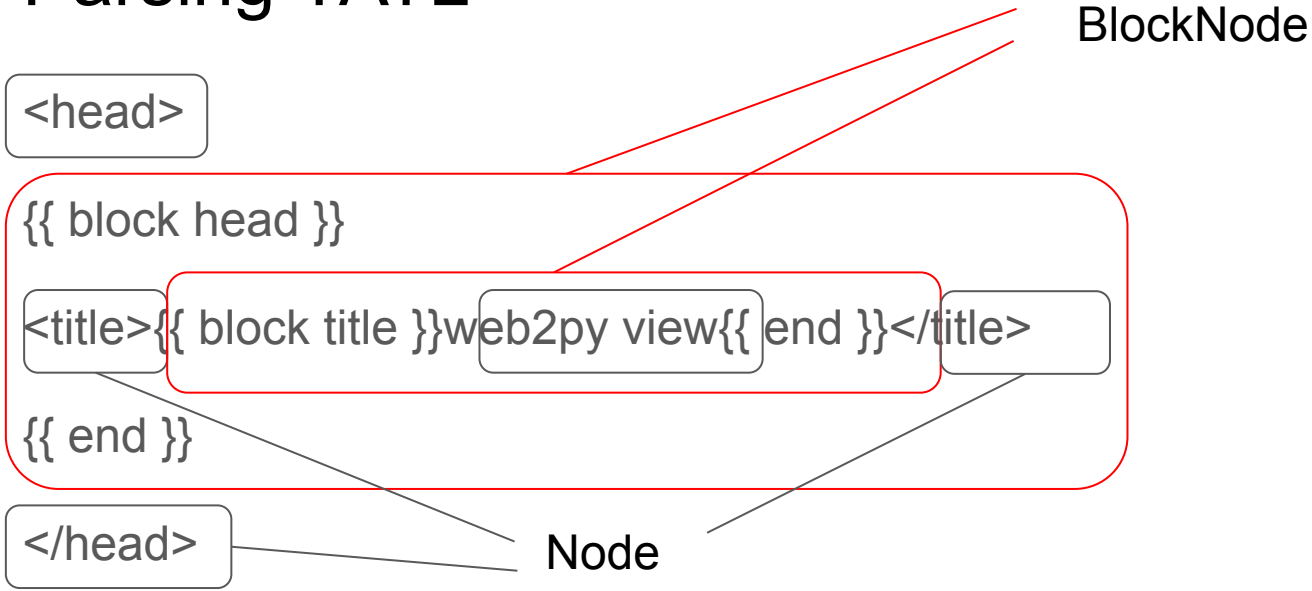
index.html

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

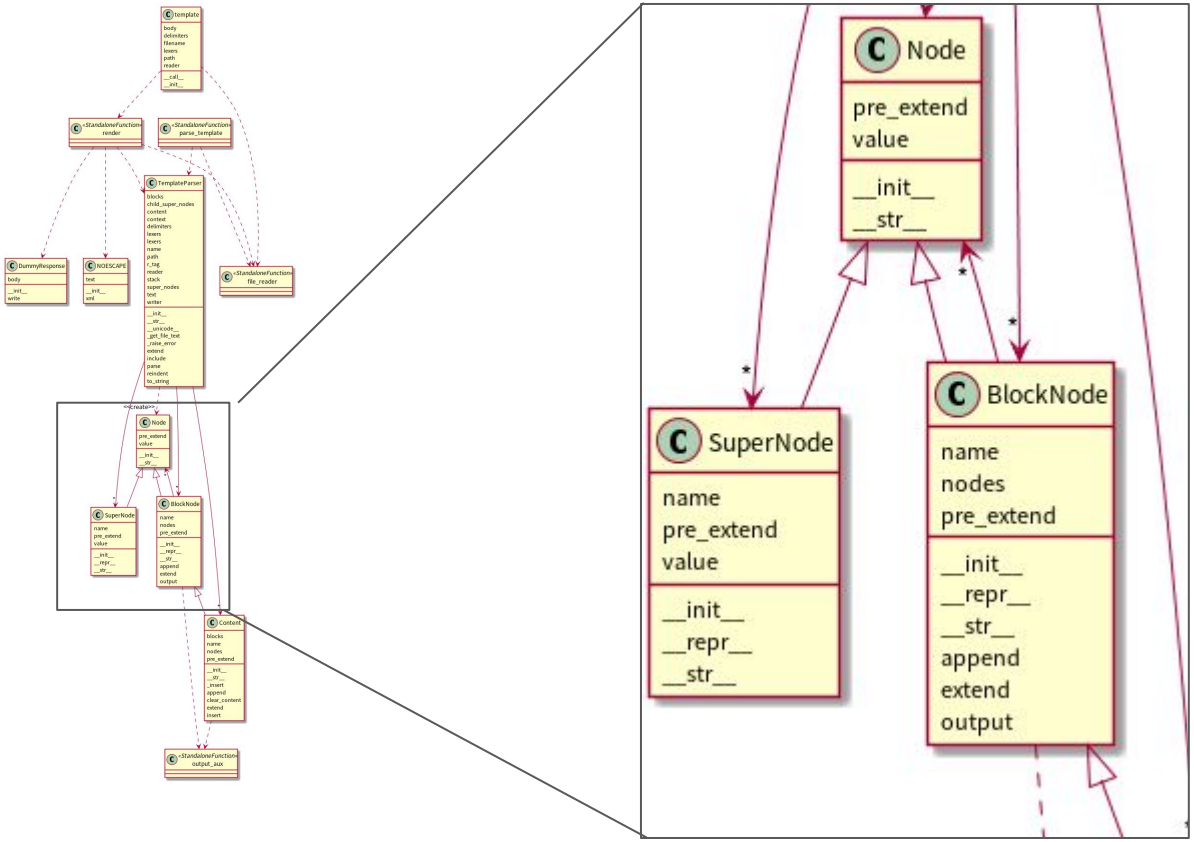
- block: contents that can be replaced
- super: place the content from the base
- extends: for template reuse

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

Parsing YATL



Class Diagram for YATL

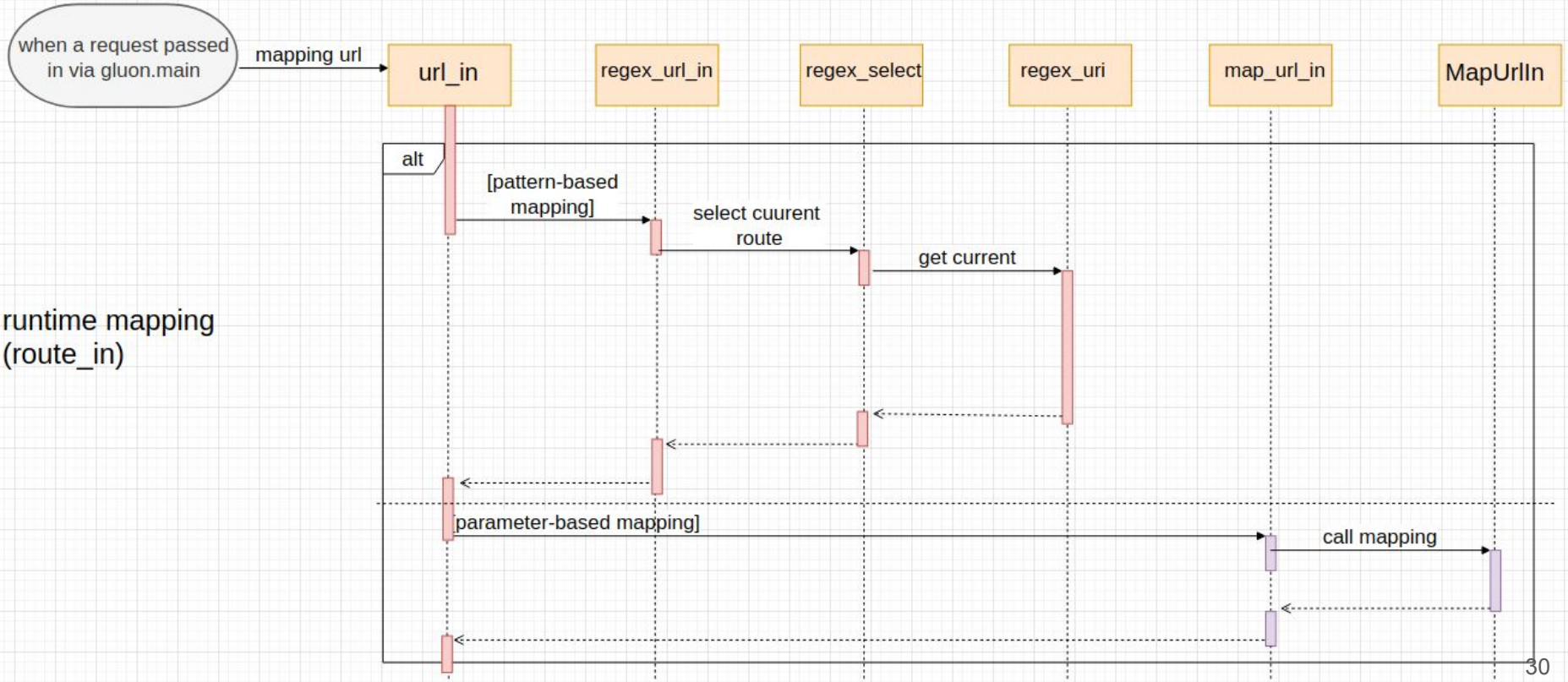


Composite Pattern

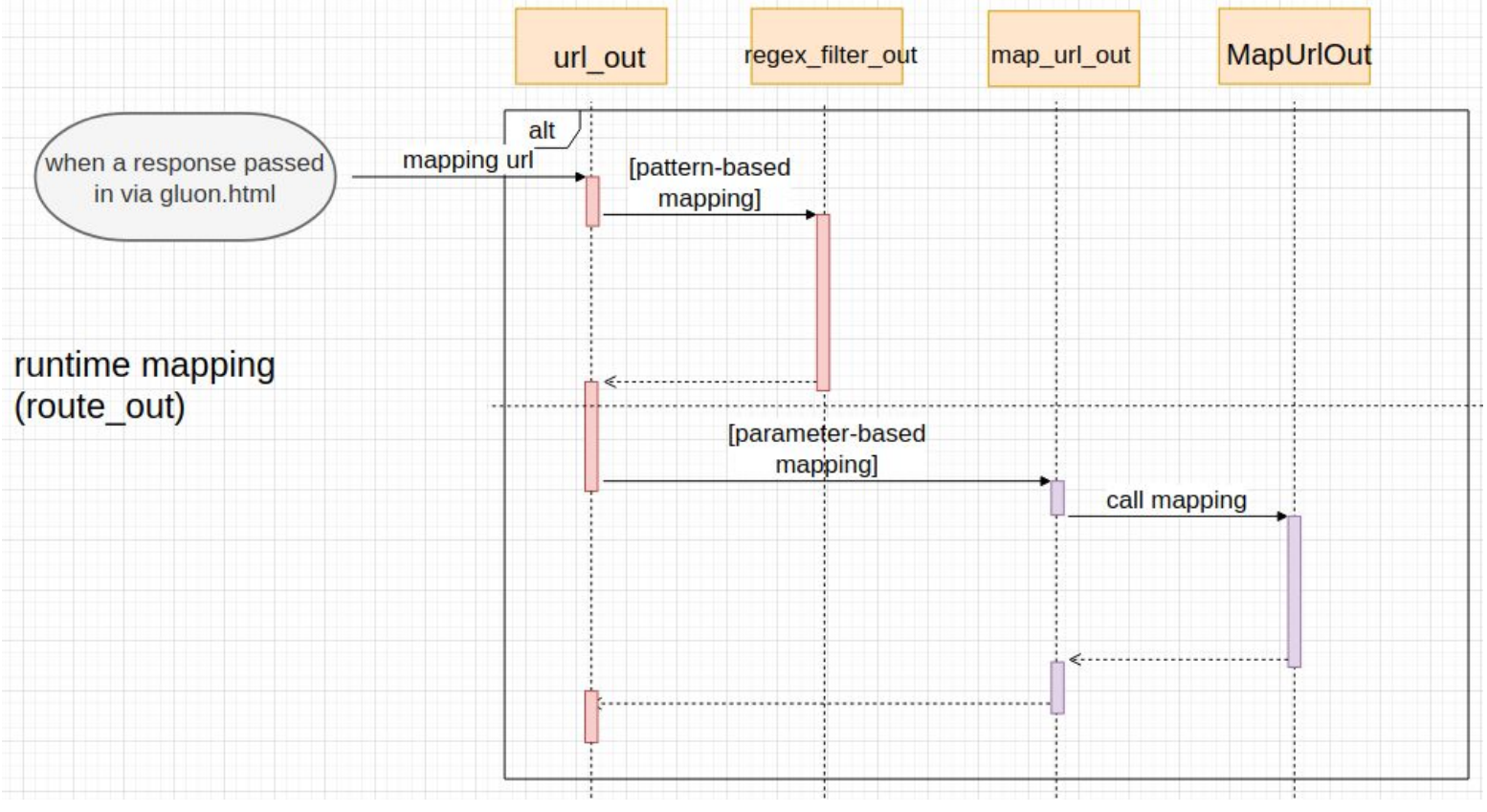
Thank you

Appendix

runtime mapping
(route_in)



Appendix


url_out
regex_filter_out
map_url_out
MapUrlOut
when a response passed in via gluon.html
alt
mapping url
[pattern-based mapping]
runtime mapping (route_out)
[parameter-based mapping]
call mapping