Two-Factor Authentication

Step 1: UserPass Authentication

```
with Yosai.context(yosai):
   new_subject = Yosai.get_current_subject()

   password_token = UsernamePasswordToken(username='thedude',
                                          credentials='letsgobowling')
   try:
       new_subject.login(password_token)
   except AdditionalAuthenticationRequired:
       # this is where your application responds to the second-factor
       # request from Yosai
       # this is pseudocode:
       request_totp_token_from_client()
   except IncorrectCredentialsException:
       # incorrect username/password provided
   except LockedAccountException:
       # too many failed authentication attempts, account locked
```

Step 2: TOTP Authentication

```
with Yosai.context(yosai):
   new_subject = Yosai.get_current_subject()

   totp_token = TOTPToken(client_provided_totp_token)

   try:
       new_subject.login(totp_token)
   except IncorrectCredentialsException:
       # incorrect totp token provided
   except LockedAccountException:
       # too many failed TOTP authentication attempts, account locked
   except InvalidAuthenticationSequenceException:
       # when 2FA is attempted prior to username/password
```

Authorization

Role-Level Access Control

Imperative Style

```
def remove_comment(self, yosai, submission):
    with Yosai.context(yosai):
        subject = Yosai.get_current_subject()

        try:
            subject.check_role(['moderator', 'creator'], logical_operator=any)
        except UnauthorizedException:
            print('Cannot remove comment:  Access Denied.')

        self.comment_service.remove_comment(submission)
```

Declarative Style

```
@Yosai.requires_role(roleid_s=['moderator', 'admin'], logical_operator=any)
def remove_comment(self, submission):
   self.database_handler.delete(submission)
```

Permission-Level Access Control

Imperative Style

```
def write_prescription(self, patient, med_id=None):
    with Yosai.context(yosai):
        subject = Yosai.get_current_subject()

        try:
            subject.check_permission(['prescription:write:{}'.format(med_id)])
        except UnauthorizedException:
            print('Cannot write prescription:  Access Denied.')

        self.rx_service.write_rx(patient, med_id)
```

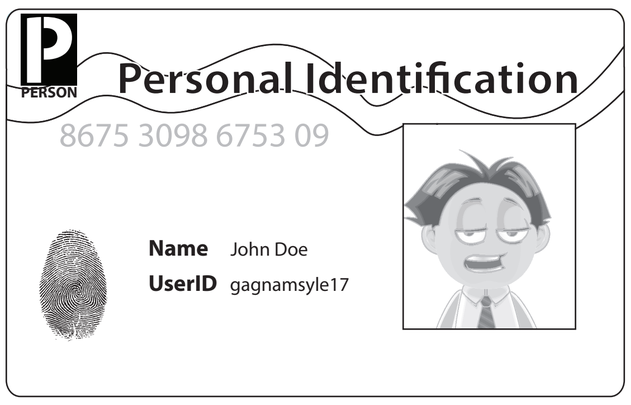
Declarative Style

```
@Yosai.requires_dynamic_permission(['prescription:write:{med_id}'])
def write_prescription(self, patient, med_id=None):
    self.rx_service.write_rx(patient, med_id)
```

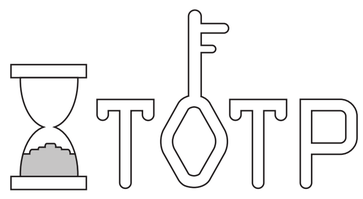
yosai

A Security Framework for python™

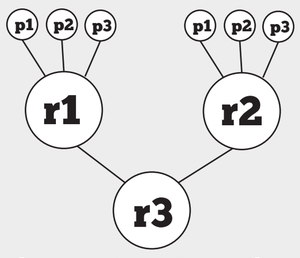
Featuring



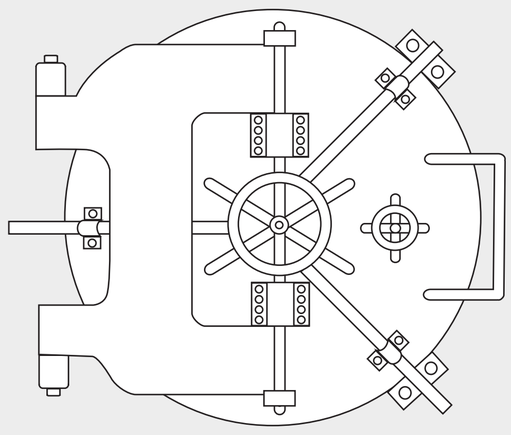
Authentication



A complete, TOTP-based two-factor authentication workflow



Enables RBAC policies using granular permissions or high-level roles

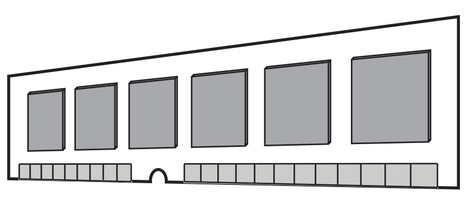


Authorization

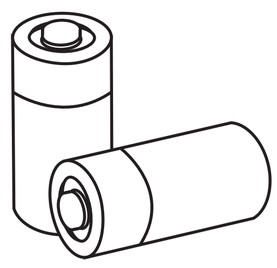
Session Management

Manage server-side sessions, including idle and absolute timeouts

Includes

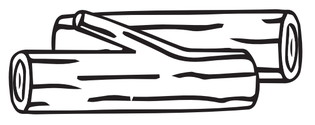


Native cache support, including serialization



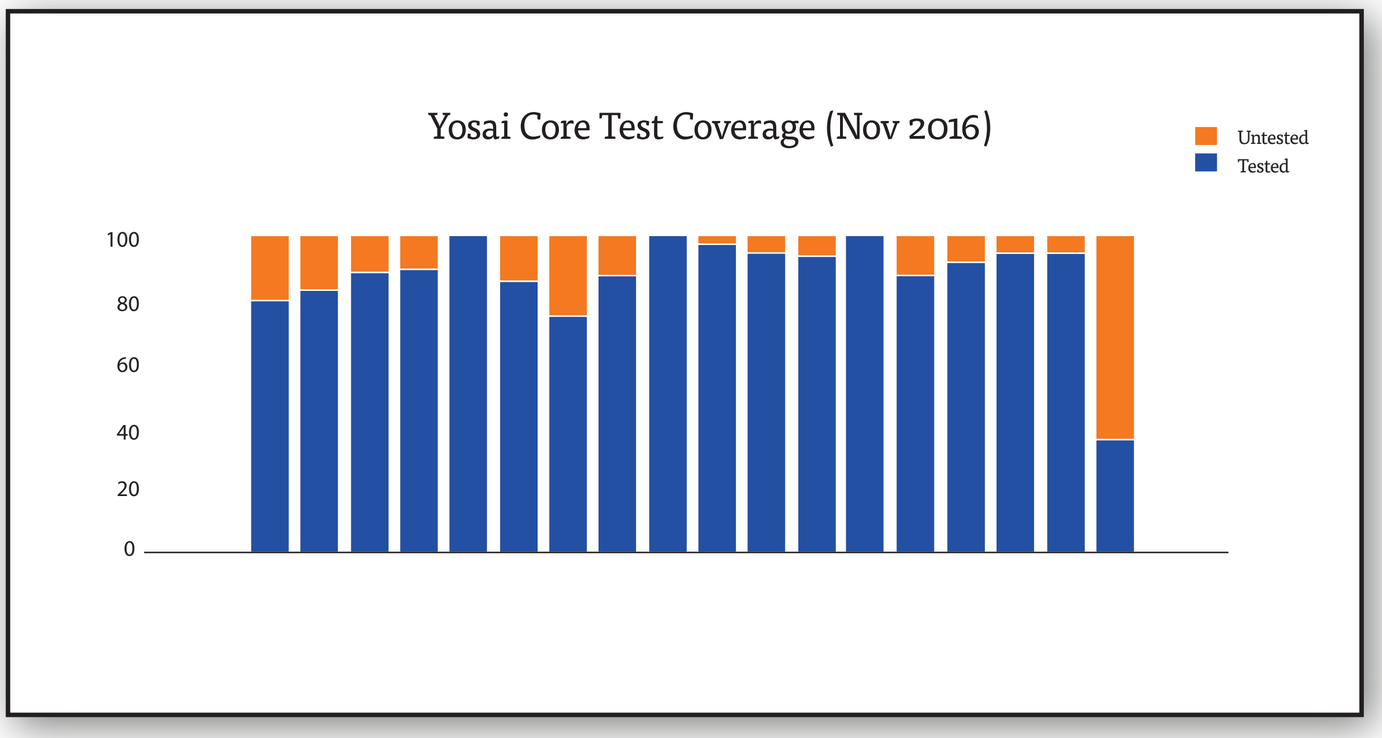
Extension and Integration libraries

Impersonate while testing by using “Run As” tool



Structured logging of every major event

Project Status and References



yosai v0.3 released to pypi in Nov 2016

Project Page

Github Page

Project Demonstration

MonsterRX is a fully-functional proof of concept web application written with the Pyramid web framework and secured using a **pyramid_yosai** integration library.

It is a medical prescription workflow application, intended for monster-use, only.

Powered By

Pyramid

SQLAlchemy

Github Page

Porting Apache Shiro

How would YOU Port Mount Shiro?

- ★ Unnecessary design patterns - builder, resolver, setter/getter
- ★ Inheritance vs Composition
- ★ Premature Unit Testing
- ★ LBYL vs EAFP
- ★ Value objects
- ★ Idiomatic Python