A Command Prompt Funk.

http://www.commandprompt.com/

Vertically Challenged

Aurynn Shaw

Project location: https://public.commandprompt.com/projects/verticallychallenged

Strong Application Permissions, using PostgreSQL

Aurynn Shaw, Command Prompt, Inc.

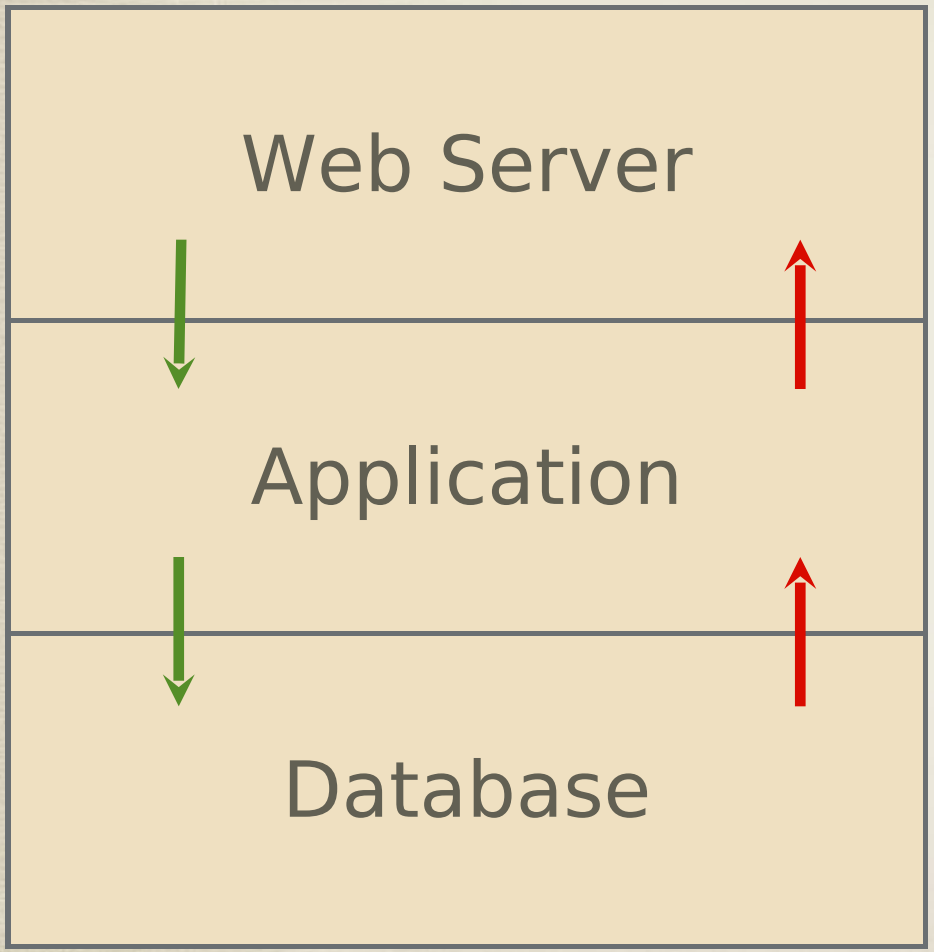
PostgreSQL Conference West, 2009

Application Architecture

Wherein

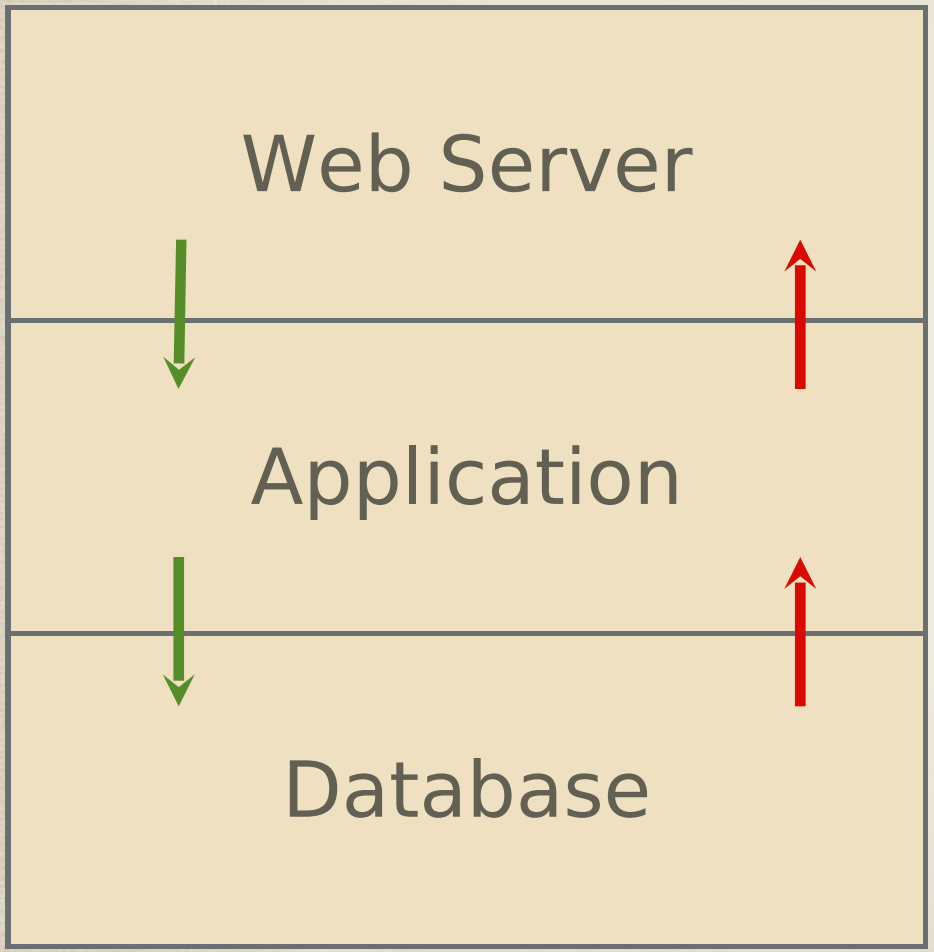
- Basic Architecture
- Limitations
- Points of Authority
- Why use DB permissions?
- Implementation

Basic Architecture



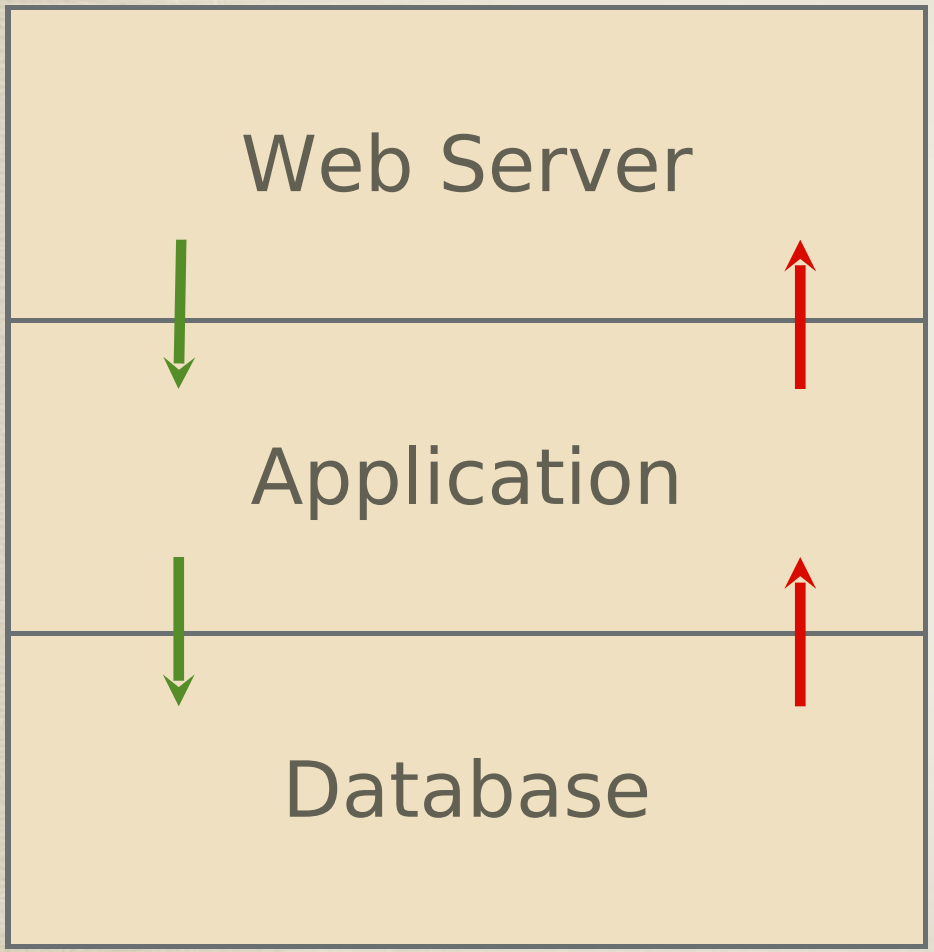
- Standard tiered design

Basic Architecture



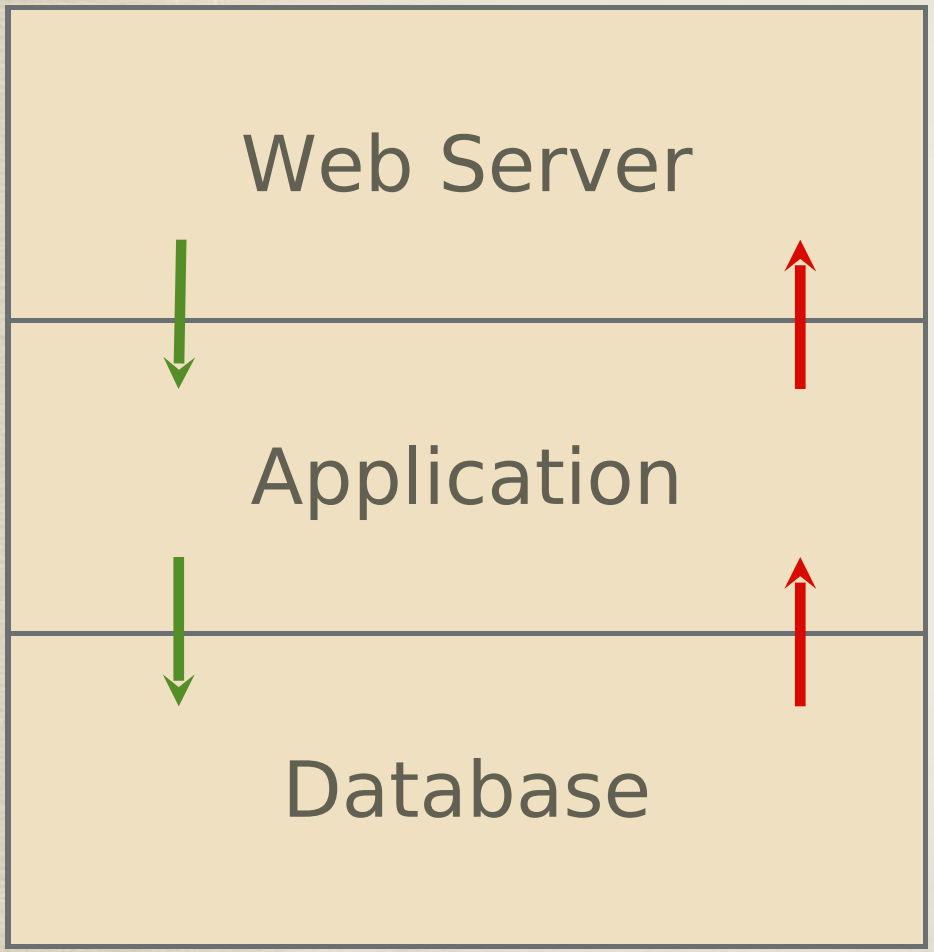
- Standard tiered design
- Standard ingress, logic, egress cycle.

Layered Communications



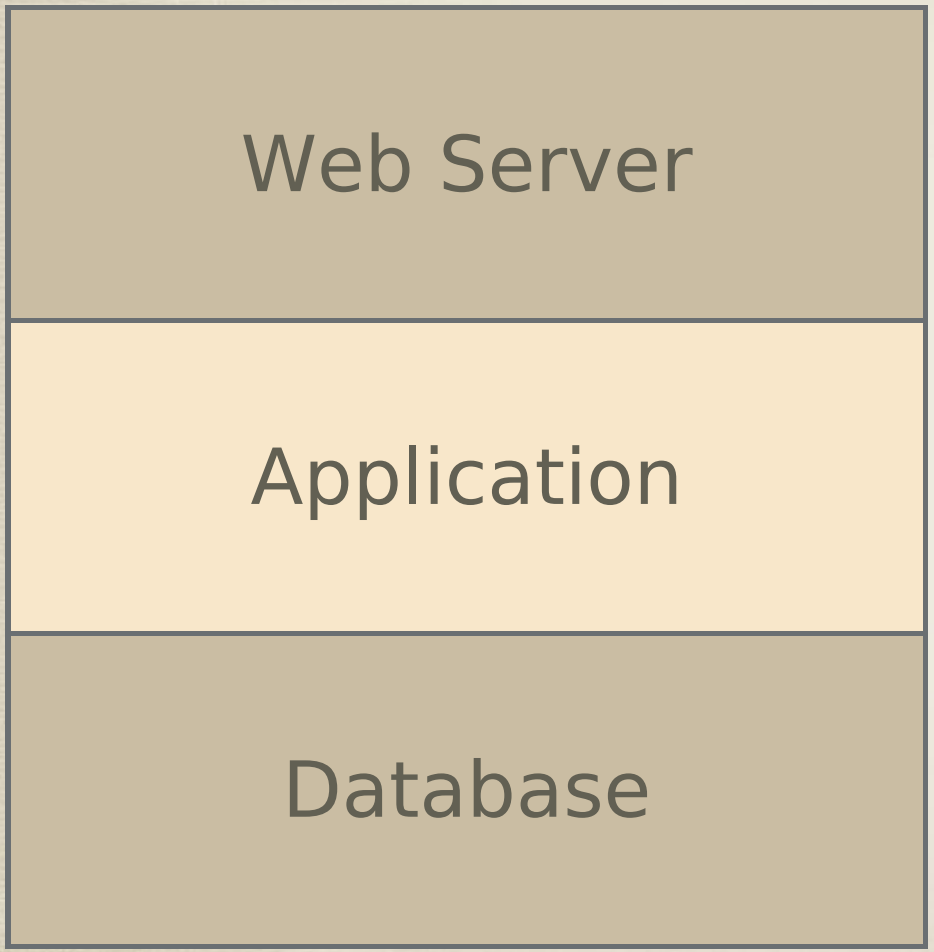
- Limitations in communication

Layered Communications



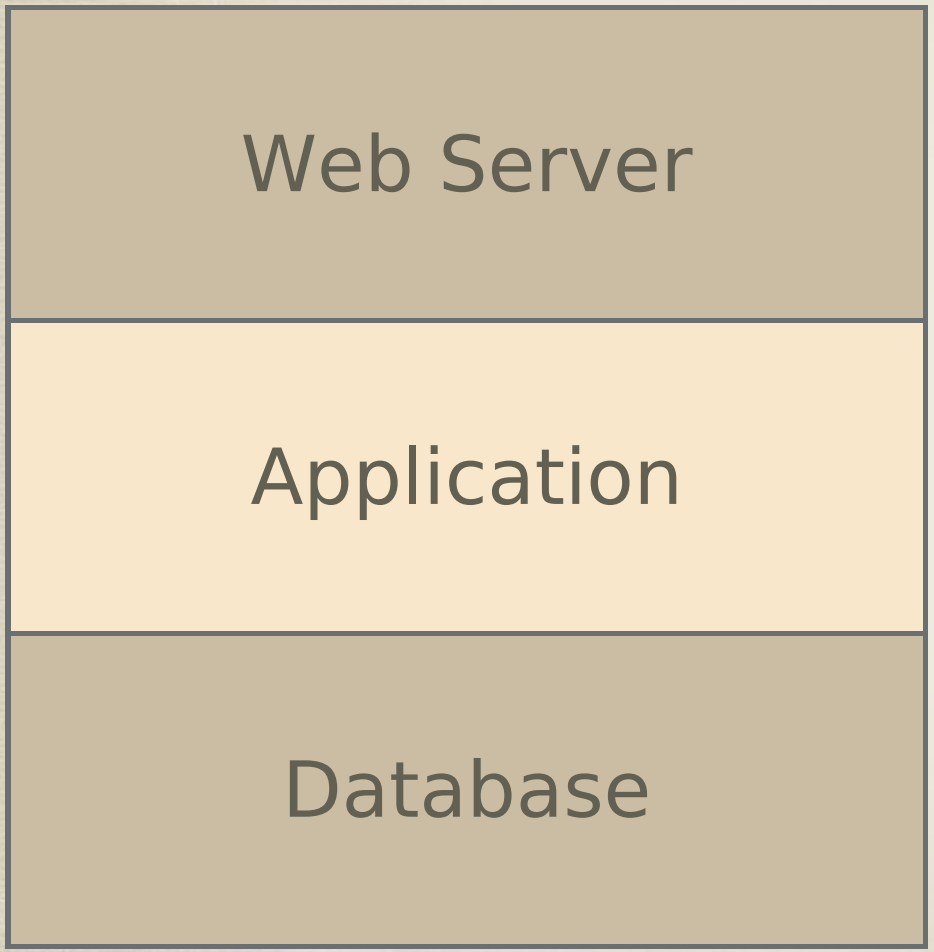
- Limitations in communication
- Influences on application design

Application Authority



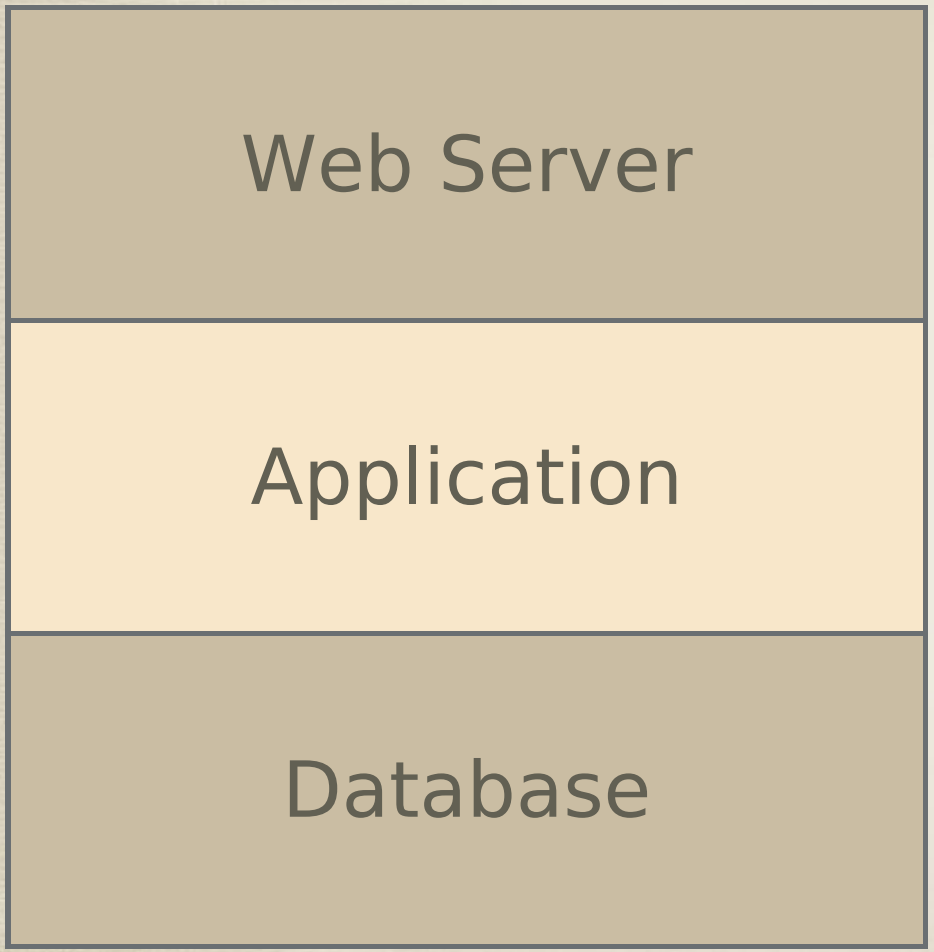
- The Application becomes permissions arbiter

Application Authority



- The Application becomes permissions arbiter
- Loss of Portability

Application Authority



- The Application becomes permissions arbiter
- Loss of Portability
- Increased work for other consumers

Using DB Permissions

- Why ?

Using DB Permissions

- Why ?
  - It’s better than what you have now

Using DB Permissions

- Why ?
  - It’s better than what you have now
  - Easy Integration

Using DB Permissions

- Why ?
  - It’s better than what you have now
  - Easy Integration
  - Once it’s set up, it’s easier to use

Basic Mechanica

- Entirely DB users
  - Why this is hard to implement

Basic Mechanica

- Entirely DB users
  - Why this is hard to implement
- DB users, with an external auth store
  - Still hard
  - Why it’s Worthwhile

Basic Mechanica

- Entirely DB users
  - Why this is hard to implement
- DB users, with an external auth store
  - Still hard
  - Why it’s Worthwhile
- The Hybrid approach

The Hybrid approach - use DB roles + rows in the database, instead of DB users.

Database Implementation

Wherein

- Basic Componentry
- Roles
- Stored Procedures

Componentry

- Postgres Implementation
  - SET ROLE
  - GRANT / NOINHERIT
  - NOLOGIN

Grant / noinherit - How this works together to provide VC.

Componentry

- Postgres Implementation
  - SET ROLE
  - GRANT / NOINHERIT
  - NOLOGIN
- sudo for the Database

Great for PCI compliance

Setup

```
vc=# CREATE ROLE vc NOINHERIT;
CREATE ROLE
vc=# CREATE ROLE auth_level NOLOGIN;
CREATE ROLE
vc=# GRANT auth_level TO vc;
GRANT ROLE
vc=#
```

Setup

```
vc=# CREATE TABLE auth_only();
CREATE
vc=# REVOKE SELECT ON auth_only FROM
PUBLIC, vc;
REVOKE
vc=# SET ROLE TO vc;
SET
vc=>
```

sudo for Databases

```
vc=> SELECT * FROM auth_only;
ERROR:  permission denied for relation
auth_only
vc=> SET ROLE auth_level;
SET
vc=> SELECT * FROM auth_only;
--
(0 rows)
```

It’s How We Role

- Basic roles

It’s How We Role

- Basic roles
  - Defines your permissions tree

It’s How We Role

- Basic Roles
  - Defines your permissions tree
- Privileged Roles
  - The over-arching Container roles
  - User, Moderator, Admin, etc.

It’s How We Role

- Basic Roles
  - Defines your permissions tree
- Privileged Roles
  - The over-arching permissions definitions
  - User, Moderator, Admin, etc.
- Entry Point

Entry point has LOGIN granted.

Your users.can do *this*

- users.can

A Quick Check

```
IF users.can(user_id, ‘type.read’) THEN
  RETURN QUERY
     SELECT * FROM type;
ELSE
  exceptable.permission_denied(‘User
lacks type.read permissions’);*
END IF;

* Exceptable function
```

Wait.. I don’t know you.

- users.can
- users.validate

User Legitimacy

```
CREATE OR REPLACE FUNCTION users.validate (
    in_username text,
    in_password text
) RETURNS int

SELECT users.validate('vc','password');
```

The Forge of Users

- users.can
- users.validate
- users.create

Creating Users

```
CREATE OR REPLACE FUNCTION users.create (
    in_username text,
    in_role text,
    in_password text
) RETURNS int


SELECT
users.create('user','auth',md5('password'
|| 'your_salt'));
```

Basic User Get

- users.can
- users.validate
- users.create
- users.get

Collecting Users

```
CREATE OR REPLACE FUNCTION users.get (
    in_user_id int
) RETURNS users.user

CREATE TYPE users.user AS (
    id int,
    username text,
    role text,
    groups text[] -- Used for app checks
);
```

This is a fairly limited user model - a real model would include the personal information &c that's needed.
It does serve as a great example on what is really *required* for vertically challenged to work.

Standing on Pylons

Wherein

- Exceptions
- Permissions checks
- Repoze.Who

Exceptions

- Repoze.who cares about 401s and 403s

Exceptions

- Repoze.who cares about 401s and 403s
- Trap VC errors in the Application
  - Emit an appropriate 401 or 403

Exceptions

- Repoze.who cares about 401s and 403s
- Trap VC errors in the Application
  - Emit an appropriate 401 or 403
- Exceptions are handled ONCE

Once

```
class BaseController(WSGIController):
  def __call__(self, environ, start_response):
    """Invoke the Controller"""
    # WSGIController.__call__ dispatches to the Controller
method
    # the request will be routed to.
    try:
      return WSGIController.__call__(self, environ,
start_response)
    except PermissionsError, e:
       abort(403)
    except NoSuchUserError, e:
       abort(401)
```

I have @needs too!

- Easily Decorate Pylons Views - @needs(‘type.read’)

I have @needs too!

- Easily Decorate Pylons Views - @needs(‘type.read’)
- Using DB permissions in the App layer

What are our Roles?

```
CREATE OR REPLACE FUNCTION users.get (
    in_user_id int
) RETURNS users.user

CREATE TYPE users.user AS (
    id int,
    username text,
    role text,
    groups text[] -- Used for app checks
);
```

This is why we return the groups our user is able to view.

I have @needs too!

- Easily Decorate Pylons Views - @needs(‘type.read’)
- Using DB permissions in the App layer
- A single DB query - Not a query per controller call.

I have @needs too!

- Easily Decorate Pylons Views - @needs('type.read')
- Using DB permissions in the App layer
- A single DB query - Not a query per object.
- Why? Multiple tiers are good.

Pylons in Repose

- The Application Cycle

Pylons in Repose

- The Application Cycle
- A Basic Repoze Configuration

A Basic Configuration

```
[plugin:form]
  use =
    repoze.who.plugins.form:make_redirecting_plugin
  login_form_url = /account/login
  login_handler_path = /account/dologin
  logout_handler_path = /account/logout
  rememberer_name = auth_tkt
[plugin:auth_tkt]
  use = repoze.who.plugins.auth_tkt:make_plugin
  secret = ticket_secret
[mdproviders]
  plugins = vc.repoze:MetadataProvider;browser
```

Pylons in Repose

- The Application Cycle
- A Basic Repoze Configuration
- The Permissions Swap

The Permissions Swap

```
from vc.model import user
class MetadataProvider(object):
  def add_metadata(self, environ, identity):
    if userid:
      try:
        u = user.user(userid)
        u.become(u.role)
      except PermissionError, e:
        if “model” in identity:
          del identity[“model”]
          return None
      identity[“model”] = u
```

And the Procedure

```
CREATE FUNCTION users.become (
    in_user_id int, in_role text
) RETURNS VOID AS $$
    BEGIN
        IF users.can($1, $2) THEN
            SET ROLE TO quote_literal($2);
        ELSE
            exceptable.permission_denied('User
cannot become specified permission level');
        END IF;
    END;
$$ LANGUAGE PLPGSQL;
```

Pylons in Repose

- The Application Cycle
- A Basic Repoze Configuration
- The Permissions Swap
- Anonymous Access

Anonymous Access

- A Wrapping Identifier
- Test For an Identity

Wrap the Identifier

```
class AnonymousWrapper(object):
  def identify(self, environ):
   i = environ['repoze.who.plugins']
['auth_tkt'].identify(environ)
   if i:
     # Not anonymous
     return i
   else:
     # Is anonymous
     # return a really basic anonymousness.
     return {"repoze.who.userid":
             anonymous_user_id} # custom UID
```

Anonymous Access

- A Wrapping Identifier
- Test For an Identity
- Return a User, or Anonymous
- Anonymous users are otherwise identical

An Anonymous User

```
SELECT users.create(
  'anonymous', -- anonymous user
  'anonymous', -- Basic, read-only role
   NULL); -- No password
```

Advantages, Disadvantages, Vulnerabilities

Wherein

- Advantages
- Disadvantages
- Vulnerabilities

Advantages

- Same roles in your Procedures, as your Application code
  - Great for consistency
- Can Implement Row-level Permissions

Row-Level Example

```
PERFORM id FROM table WHERE id = i_id AND
owner_id = user_id;
IF NOT FOUND THEN
    -- Some Procedural code.
ELSE
  exceptable.permission_denied('User does
  not own specified record.');
END IF;
```

Disadvantages

- No PG-backed row-level authorization

Disadvantages

- No PG-backed row-level authorization
- DDL permissions take work to implement

Simple, but Timeconsuming

```
vc=# CREATE TABLE auth_only();
CREATE
vc=# REVOKE SELECT ON auth_only FROM
PUBLIC, vc;
REVOKE
```

For. Every. Object.

Disadvantages

- No direct row-level authorization
- DDL permissions take work to implement
- No ad-hoc Users

Disadvantages

- No direct row-level authorization
- DDL permissions take work to implement
- No ad-hoc Users
- Custom permissions require new entry points

Vulnerabilities

- Any possible PG escalation attack
- SET ROLE could switch to an Admin user
- User errors a perennial favourite

And thus

Questions?

Get it

https://projects.commandprompt.com/public/verticallychallenged

Thank you!