University of BRISTOL

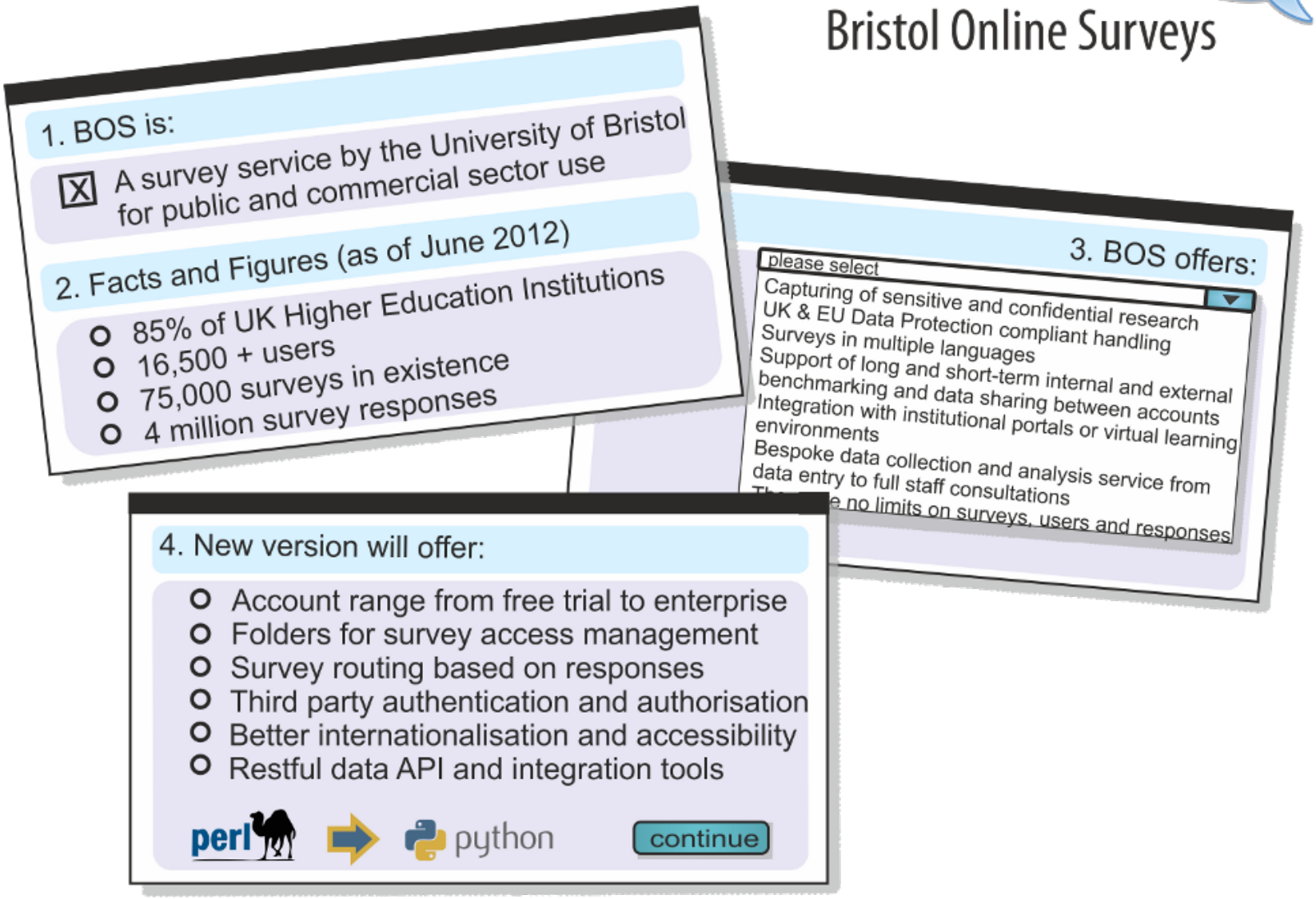
Creating federated authorisation

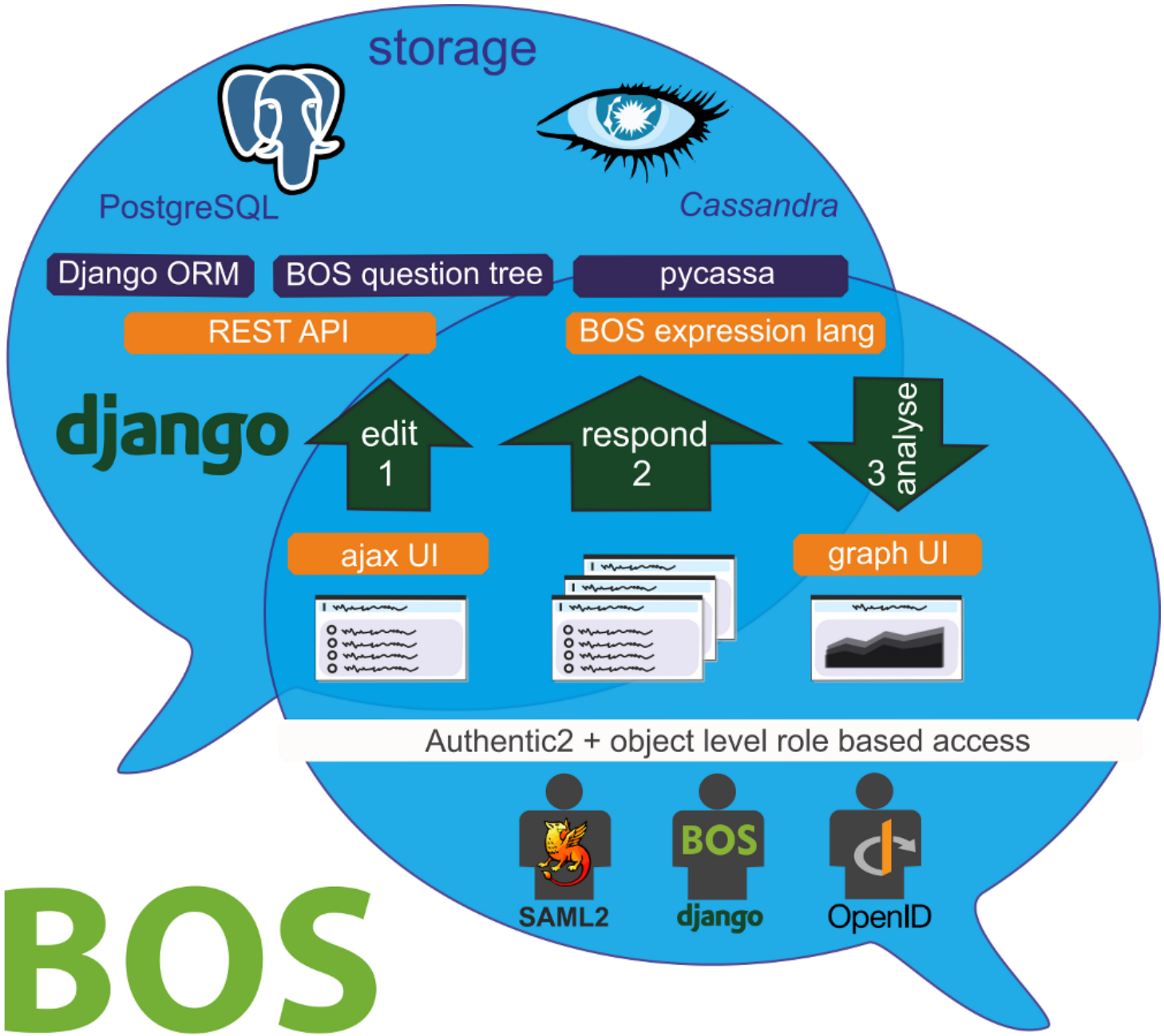
for a Django survey application

Ed Crewe

Background
- the survey application

BOS
Bristol Online Surveys





BOS

Online Surveys

Federated authorisation

What do I mean by this?

1. Users login at a third party identity provider - IdP

2. They return to the application with user attributes set by the IdP or a service access token.
   Authorisation is determined by the 3rd party.

3. The service token has an established scope = authorisation level to the service via the application

OR

4. The attributes are used to provide different entitlements = authorisation levels to the service provider - SP

Federated authorisation

What am I hoping it can do for the users?

- Familiar login page and credentials
  - moves login support out of BOS.

- People have identities across more than one institution.
  - display and manage them via a single home page.

- Devolves access control to their organisation's systems
  - reduce overhead for institutional admins.

Sorry are we assuming federated authentication too?

So taking a step back ... just because the authorisation is federated that doesn't force the authentication to be as well. *

But for BOS we want a system that federates identity providers who deliver authentication and authorisation together as a package tied to that provider and its BOS account.

So to get things clear let's cover the basics ...

* OAuth

Authentication

Login to an application assigns a user an identity

Central network sign on is the start.
Allows one login to be used across many applications.

Most widely adopted is probably **Kerberos**
(1980s MIT open source protocol based on client-server, key and ticket exchange)

SSO - (web) single sign on provides a web protocol that wraps central sign on with a web login.

Widely used open source ones are **OpenID** and central authentication service, **CAS**.

Authorisation

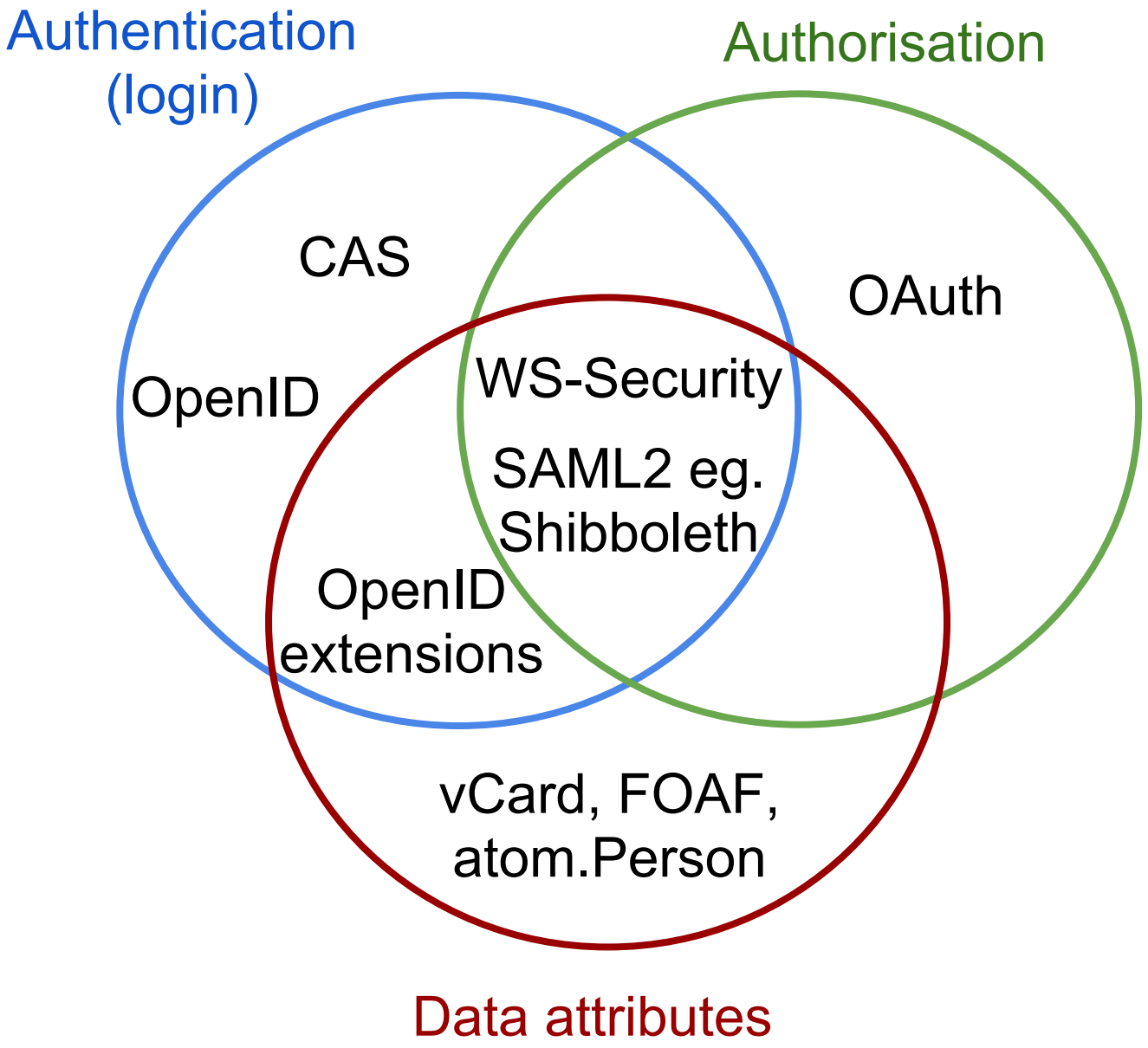
Grants an identity access

1. **OAuth** app proxies a user's identity to a service provider

2. **SAML2** a set of standards with many implementations from simple SSO to full federation management.
   - Google SAML
   - SAML 2 Kerberos Web Browser SSO
   - Liberty alliance SSO (Lasso)
   - Active Directory ADFS2 (federates with WS-Security)
   - Shibboleth ... etc.

3. **OpenId** doesn't do authorisation yet, but has data attributes ... with possible authorisation features in future.

comparison of web identity exchange protocols



Security

Establish trust relationship & encrypt communication

OpenId and **CAS** do not require key exchange between IdP and SP, they also rely on the transport layer being encrypted.

OAuth does require key exchange for the trust relationship.

SAML2 requires key exchange for signing messages, and uses XML encryption to secure messages on top of SSL.

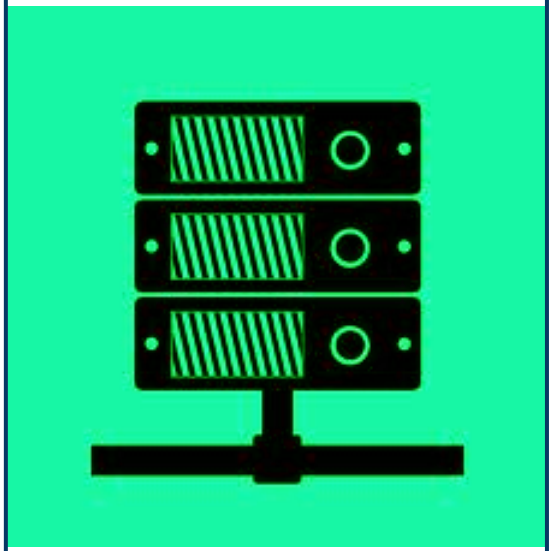
WS-Security is similar, a Windows / IBM encryption standard that uses secure SOAP messaging* for authorisation federation

* OpenID, CAS, SAML2 & OAuth use either SOAP or REST

authorisation process

user requests access

application - service provider



login

redirected to idp with callback url

redirects to callback with signed handle

handle used to request authorisation

SAML2 attributes sent (or OAuth token set)

identity provider

Back to BOS

survey system users

- The UK has a Shibboleth federation of all Universities, so we need to 'Shibbolize' BOS - both for login and authorisation where possible
- Some of these users may still wish to use a local login
- There are also non-University users who should be able to use their favourite OpenID provider
- Users may have multiple accounts / institutions.

... so what does django or its plugins give us

django.contrib.auth

1. Basic users, groups and class level permissions.
   A user profile convention for extending data attributes.

2. Group is just id and name

3. User has a small set of fixed
   data attributes along with password
   and 'roles' all together in one table.

django.contrib.auth future

There has been much debate over improvements and hence little progress for a few years - so recently a BDFLs' decision was made.

Move to a fully customisable User model specified in settings with mixins for data, permissions & authentication.

Hence, developers can pick 'n' mix with their custom User model.

Work has commenced on this for the next django version (1.5)

django authorisation eggs

1. **django-guardian** - tight integration with contrib.auth and django admin for object level permissions, but not roles.

2. **django-rules / rulez** - flexible rules based object authorisation so can be made to act like RBAC (rulez fork = memory only, for speed)

3. **django-permissions** (part of LFS) - uses contrib.auth users and groups then its own permission & role tables to deliver full object RBAC

django authentication* eggs

1. **django-social-auth** - most popular and easy to install, pure contrib.auth with OAuth and OpenID

2. **Authentic2** - SAML2, CAS, OAuth, OpenID federation django app - uses Lasso C-library for speed.

3. **django-shibboleth** - thin wrapper of Apache Shibboleth - Shibboleth only (just for one SAML2 implementation)

4. **pySAML2** - SAML2 only, requires repoze (ZODB) and wsgi server.

* SAML2 ones do some authorisation too

What we chose to use

Authentic2

- Has a UI (django admin) config of SAML2, OpenID, OAuth, CAS federation. With federation policies creation.
- Perfect for handling a large number of IdPs

django-permissions

- Acts as a roles based separation layer from permissions.
- Does so on an object level basis. Treats users and groups as common principals in a standard roles based access control manner.
- Permissions are separate from the default contrib.auth ones

Requirements part 1

1. map one or more authentication identities to a single user

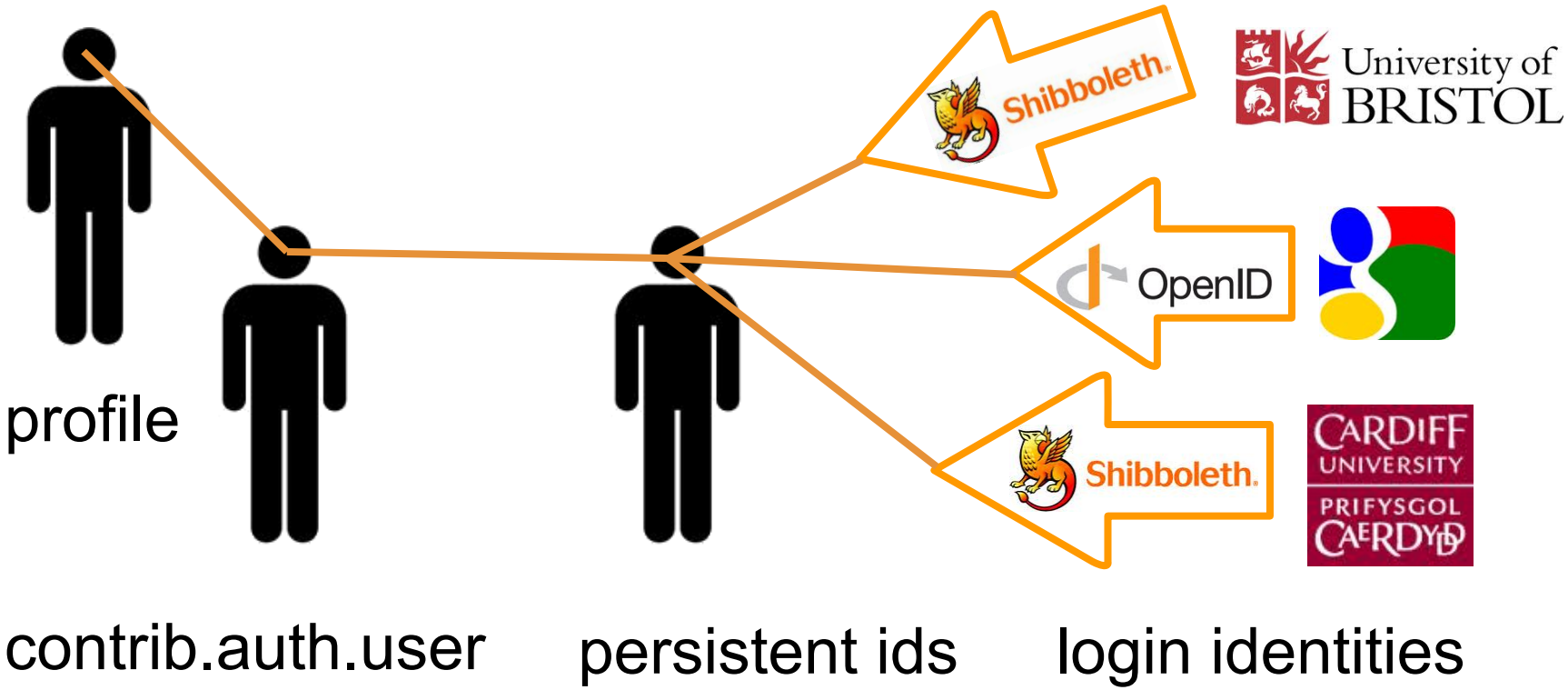
2. easily configure a federation for the survey service of remote authentication and authorisation providers (IdPs)

3. apply policies for federated authorisation

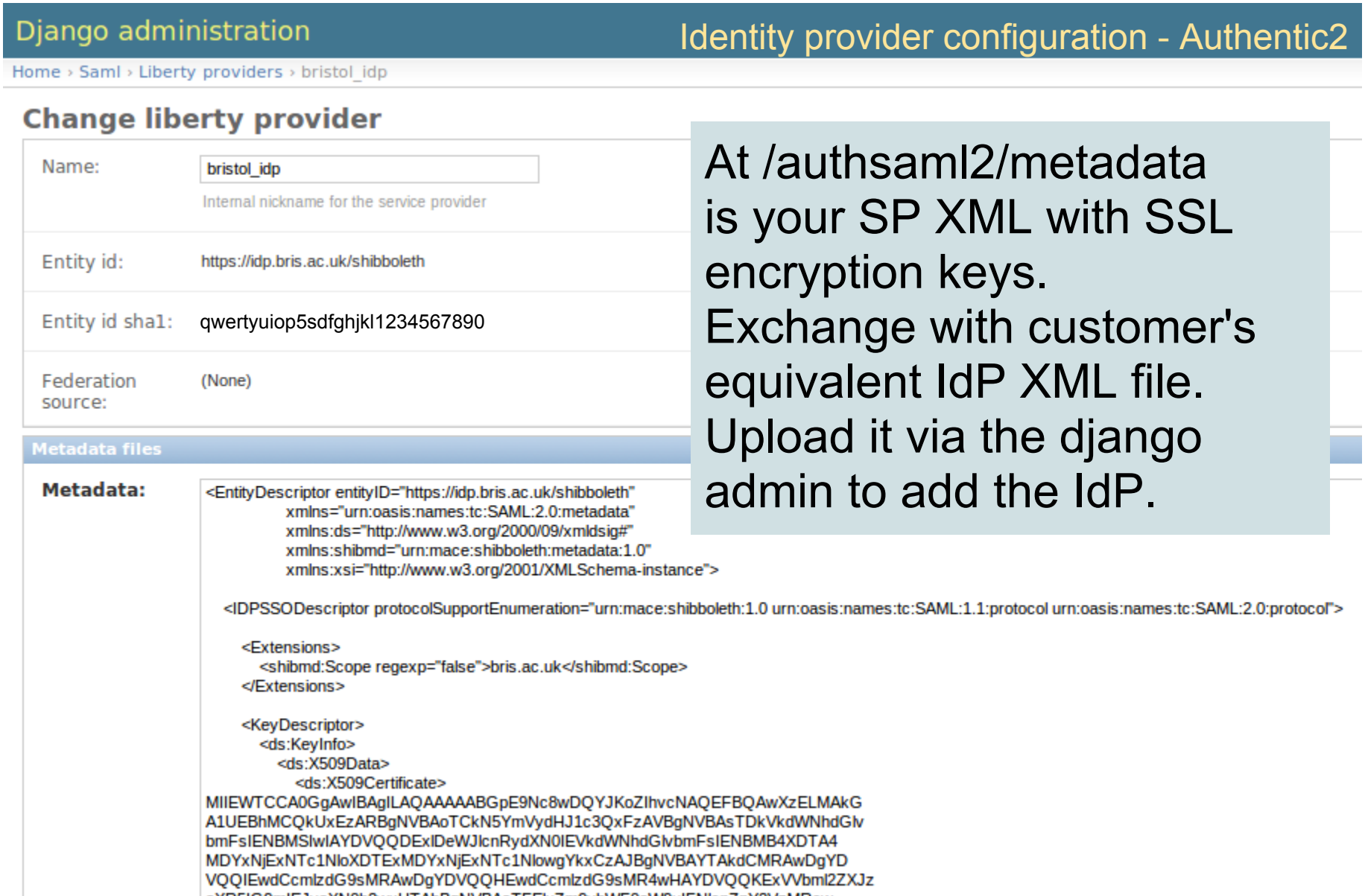
4. use of role based access control (RBAC) for allocating object permissions to groups

Authentic2 does requirement 1

Authentic2 is designed to map one or more identities to a user. A django user can be auto-created via a Shibboleth login, or OpenID etc. and they can add other login identities to themselves.

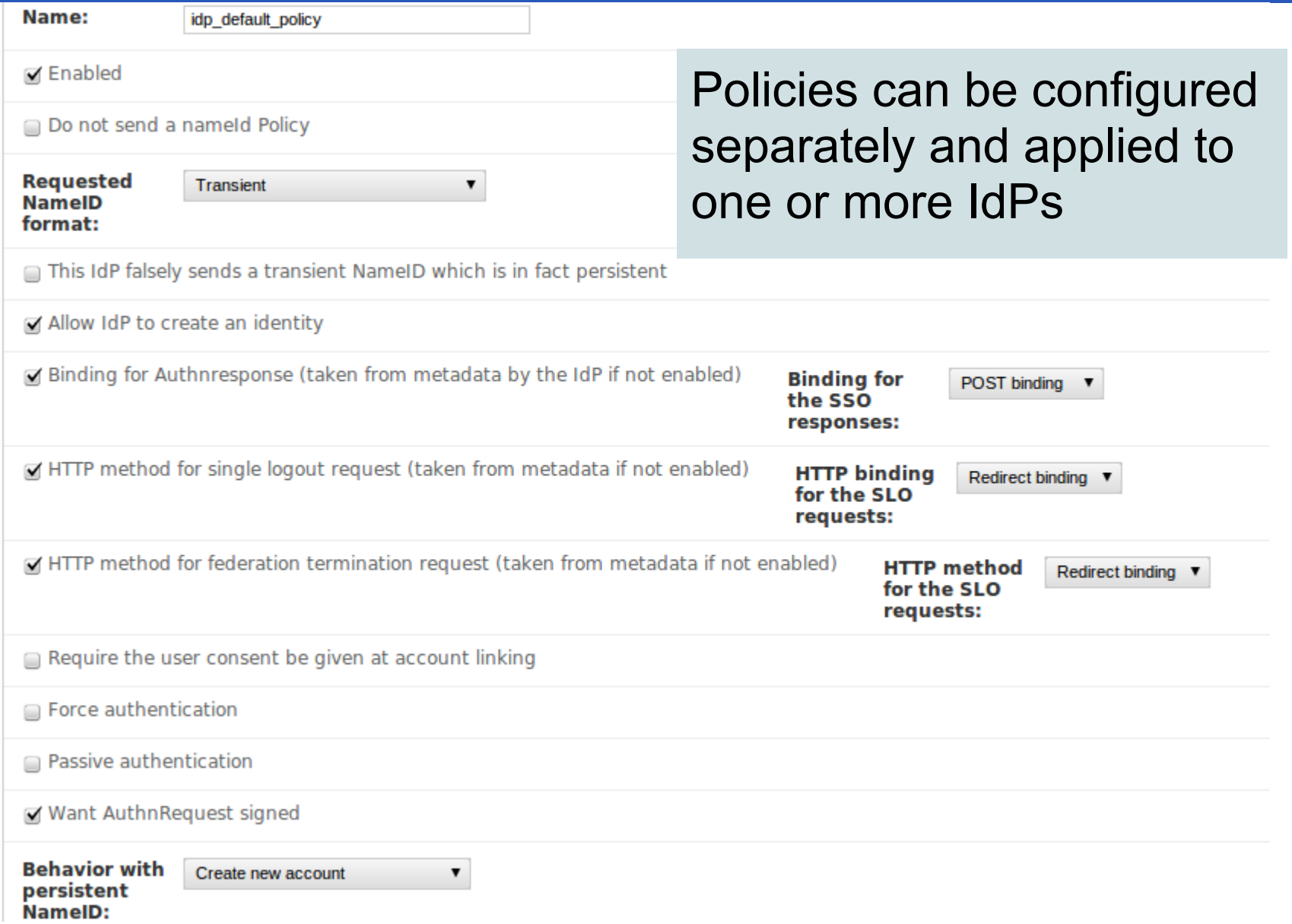


Authentic2 easy IdP creation -> 2



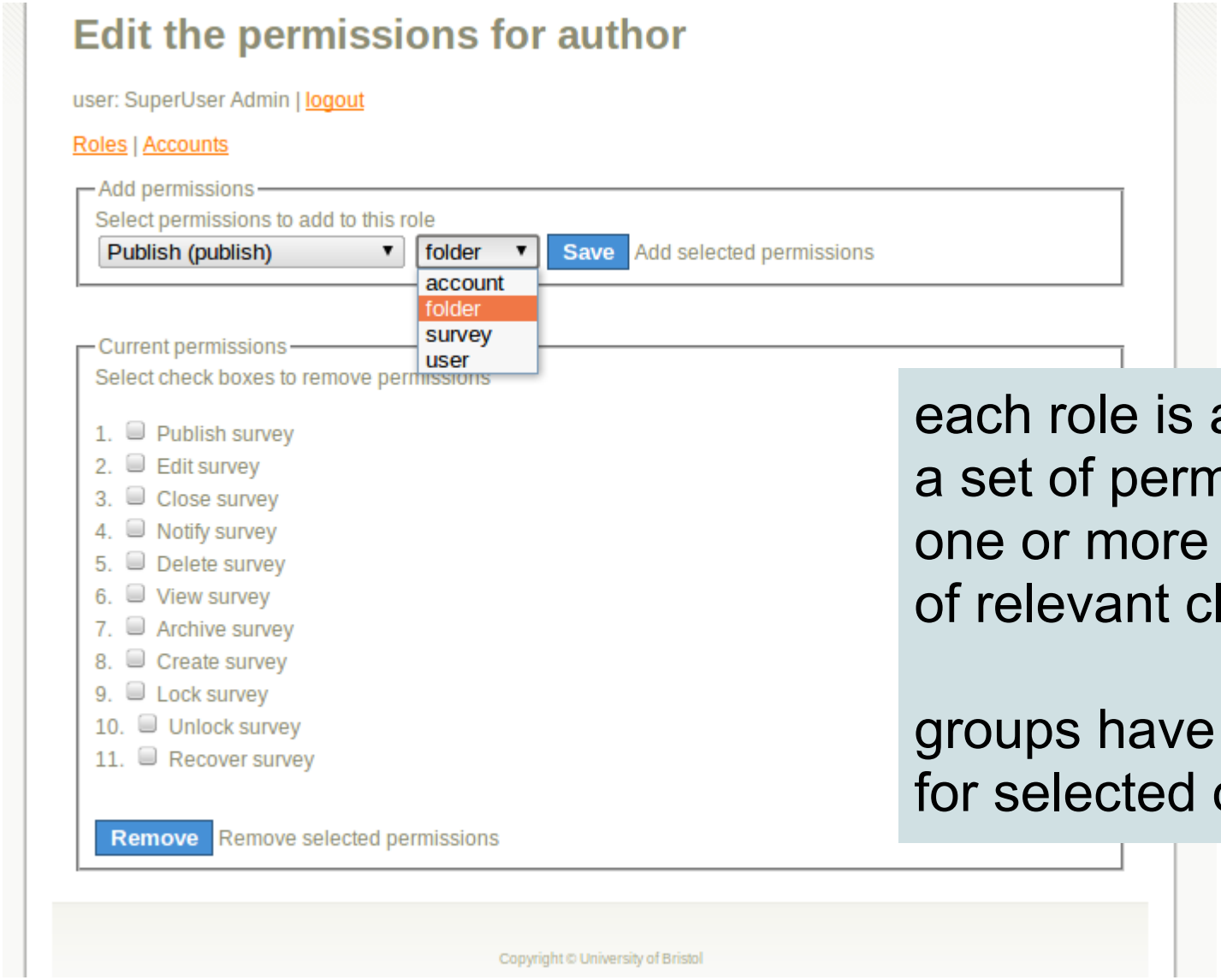
At /authsaml2/metadata is your SP XML with SSL encryption keys. Exchange with customer's equivalent IdP XML file. Upload it via the django admin to add the IdP.

Authentic2 easy policy config -> 3



Policies can be configured separately and applied to one or more IdPs

django-permissions object roles -> 4



each role is allocated a set of permissions on one or more of a subset of relevant classes

groups have these roles for selected objects

What was modified

django-permissions

Roles are allocated as a set of permissions per object - but we only need a single role definition per class.

Modify to make local role allocation group only, and automate it.

So user role allocation is remote only, via entitlements.

Authentic2

Shibboleth IdPs require the login ID to be transient.

Modify transient ID policies to check for persistent ID attributes.

If found switch to use them to create a persistent user mapping.

Requirements part 2

5. derive (temporary) authorisation via IdP set attributes

6. mix local and remote authorisation allocation

7. query remote allocation for admin purposes

8. create an easy UI for users to manage their federated accounts and for account admins to manage their users.

9. identity lifespan management

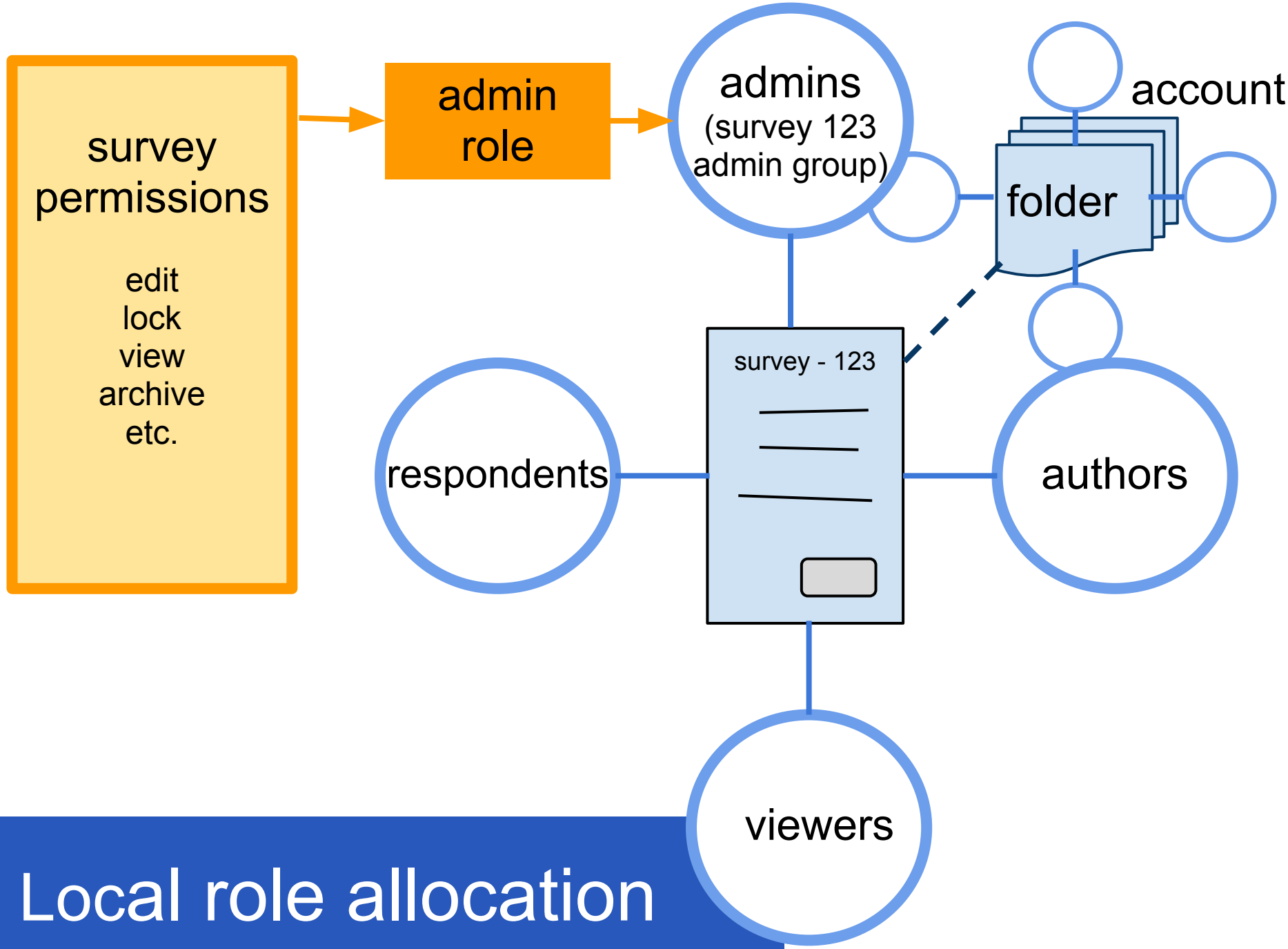
How do we tackle requirements 5 & 6

Roles act as a separation layer from permissions and are either assigned locally to groups or remotely via entitlement attributes to users.

Roles = superuser, administrator, author, viewer, respondent

Groups are auto-created and assigned roles via signals linked to the creation of objects requiring permission allocation

Permissions checking is done via a standard view class or function decorator. The decorator first tests remote then local authorisation.



Local role allocation

Remote role allocation

Remote survey allocation uses entitlement attributes to allocate the same roles to objects for a user.

bos:account:role:object:(id or codename):(ex/include:list)

e.g. bos:cardiff:author:survey:how_are_you

the account name is checked against the institution's issuer Shibboleth url.

Entitlements retrieved from the user session are temporary allocations dependent on the SAML login.

Permission checking

Protect code with decorators that check request.user either for django's class or function based views

```
@class_permissions('edit', 'lock')
@func_permissions(['edit', 'lock'], objects)
```

Wrappers for calls to

```
check_permission(user, object, permission)
```

which uses get_entitlements then get_roles to do the check

Also utility method for reverse lookup

```
objects = objs_with_permission(user, klass, permissions)
```

A look at the code ...

So what happens when somebody goes to edit a survey. Django url dispatch does its thing and routes the call to the edit survey view with survey id etc. passed in the kwargs, and request.user available ...

```
@class_permissions('edit', 'lock')
class EditSurvey(UpdateView):
    """ A view for updating a survey """
    template_name = "edit/survey.html"
    model = Survey
    form_class = SurveyForm

    def get_context_data(self, **kwargs):
        context = super(EditSurvey, self).get_context_data(**kwargs)
        context['mode'] = 'edit_post'
        ...
```

```
class class_permissions(object):
    """ Tests the objects associated with class views - against permissions """
    perms = []
    request = None

    def __init__(self, *args):
        self.perms = args

    def __call__(self, View):
        """ Main decorator method """

        def _wrap(request=None, *args, **kwargs):
            """ First decorate with dispatch_set_request with request.user from
                login_required can then test permissions in get_context_data.
            """
            setter = getattr(View, 'dispatch', None)
            if setter:
                decorated = method_decorator(dispatch_set_request( self))(setter)
                setattr(View, setter.__name__,
                        method_decorator(login_required)(decorated))
            getter = getattr(View, 'get_context_data', None)
            if getter:
                setattr(View, getter.__name__,
                        method_decorator(decklass_permissions( self))(getter))
            return View
        return _wrap()
```

class permissions

```
def decklass_permissions(decklass):
    """ Function decorator for decorator class to check user for permissions.
        decklass is the decorated class, view_func is get_context_data
        and applies to different generic view classes
    """

    def decorator(view_func):
        @wraps(view_func, assigned=available_attrs(view_func))
        def _wrapped_view(**kwargs):
            context = view_func(**kwargs)
            obj_list = context.get( 'object_list', [])
            if not obj_list:
                obj = context.get( 'subobject',
                                   context.get( 'object', None))
                if obj:
                    obj_list = [obj, ]
            check_permissions(decklass.request, decklass.perms,
                              decklass.request.user, obj_list)
            return context
        return _wrapped_view
    return decorator
```

decorator class core function decorator

```
def check_permissions(request, perms, user=None, obj_list=[]):
    """ Checks permissions against list of objects
        If used to decorate a function then these objects must be passed in
        the kwargs as object or object_list
        Also includes superuser test
        Retrieve role for object via saml2 entitlements first,
        then local permissions.
    """
    if not user:
        raise Denied("""No user has been passed in kwargs or context
                        to test %s permissions"""  % str(perms))
    if 'superuser' in perms and not user.is_superuser:
        raise Denied("You do not have superuser permissions" )
    if not obj_list:
        raise Denied("""There is no object supplied in the request
                        to test %s permissions"""  % str(perms))

    ents = get_entitlements(request)
    for codename in perms:
        for obj in obj_list:
            if not has_class_permission(obj, user, codename, ents):
                raise Denied("""User '%s' doesn't have permission
                                '%s' for object '%s' (%s)"""
                             % (user, codename, obj, obj.__class__.__name__))
```

check permissions function

```
def get_entitlements(request):
    """ Grab the entitlements from the session
        Format received - bos:account:role:object_type:id/default_short_name/all
                returned - [account][all][object_type] = roles list
                       or  [account][object_type][id] = roles list
        where object_type = lowercase class name eg. survey
    """
    attributes = request.session.get( 'attributes', {})
    entitlements = attributes.get( 'eduPersonEntitlement', '')
    issuer = attributes.get( '__issuer', '')
    accounts = ()
    acct_ents = {}
    if issuer:
        accounts = issuer_accounts(issuer)
        for acct in accounts:
            acct_ents[acct] = { 'all':{}}
    if accounts and entitlements:
        entitlements = entitlements[ 0].lower().split(' ')
        entitlements = [ e.replace(EPREFIX,  '') for e \
                         in entitlements if e.startswith(EPREFIX) ]
        entitlements = [e.split(DIVIDER)  for e in entitlements]
        # clean up and check data formats
        ...
```

get entitlements

```
def has_class_permission(obj, user, codename, ents):
    """ Check permissions via roles """
    def get_roles_ents(obj, user):
        """ Add roles a user has for an object together
            from both entitlements and local allocation (by groups)
        """
        if ents is None:
            roles = []
        else:
            roles = role_entitlements(ents, obj)
        roles.extend(get_roles(user, obj))
        return roles
    ct = get_content_type(obj)
    if check_parent_class_permissions(ct, codename, obj):
        return True
    return check_class_permissions(ct, codename, get_roles_ents(obj, user))

def check_class_permissions(ct, codename, roles):
    """ Checks whether the content class has the permission for the role """
    p = ObjectPermission.objects.filter(content_type=ct, content_id= 0,
                        role__in=roles, permission__codename = codename)
    if p.count() > 0:
        return True
    return False
```

check class permissions

```
def role_entitlements(ents, obj):
    """ Translate the Shibboleth entitlements to roles for the object
        for temporary assignment to user during has_permission check
        First establish account for object then check entitlements
    """
    roles = []
    account = ''
    if ents:
        objtype = type(obj).__name__.lower()
        if hasattr(obj, 'account_id'):
            account = obj.account_id.default_short_name
        elif type(obj) == ACCOUNT_TYPE:
            account = obj.default_short_name
        if account:
            a_ents = ents.get(account, {})
            if a_ents:
                if a_ents['all'].has_key(objtype):
                    roles = a_ents[ 'all'][objtype]
                if a_ents.has_key(objtype):
                    if a_ents[objtype].has_key(obj.id):
                        roles.extend(a_ents[objtype][obj.id])
    if roles:
        return list(Role.objects.filter(name__in=roles))
    else:
        return []
```

role entitlements

#TODO: requirements 7,8 & 9

7. query remote allocation for admin purposes
8. create an easy UI for users to manage their federated accounts and for account admins to manage their users
9. identity lifespan management

Shibboleth native install has a utitlity which allows direct querying of a user's attributes by supplying a persistent ID to the IdP (NativeSPAccountChecking)
Need to use this or build an equivalent based on Authentic2

This would allow both for the display of remote authorisation allocations and lifespan tests.
The remote and local data can be combined in the admin UI

Conclusion

University of BRISTOL

Python and Django have a good range of tools and add ons to deliver complex authentication and authorisation, greatly reducing the work needed.
(thanks also to Authentic2 and django-permissions)

For a service of the scale of BOS a fully featured authorisation federation tool is a core building block for allowing smooth integration into client's systems.

Questions
or
suggestions?

personal - http://www.edcrewe.com
work - http://www.bris.ac.uk/ilrt/
mail - ed.crewe@bristol.ac.uk
twitter - @edcrewe

Abstract

schedule https://ep2012.europython.eu/p3/schedule/ep2012/ links to https://ep2012.europython.eu/conference/talks/creating-federated-authorisation-for-a-django-survey-system

This talk is about the development of the user system for an online national national national survey application.

The goal of the talk would be to impart some knowledge of the current state of open authorisation standards and how the python web application tools that are available for them may be applied in practise. The prerequisites are some background in web development and perhaps authorisation systems - experience of Django is not necessary but may be useful.

The introduction will give background regarding the application, for context, e.g. 3 million survey responses in a Perl web application being rewritten in Django with Cassandra and Postgres data storage. The need to add external access control via Shibboleth (SAML) and OpenID.
This will be followed by a summary of the features and differences between the three main open standards for third party access control, SAML, OAuth and OpenId.

Then I will move on to the issues involved:

- **mapping one or more authentication identities to a single user**
- **how authorisation can be derived via attributes, to automate group membership**
- **the use of role based access control for allocating object permissions to groups**
- **identity lifespan management**
- **mixing local and remote authorisation allocation, etc.**

Next will be an explanation of what django.contrib.auth has, its likely future (a rewrite is currently under discussion), and a review of the various authentication and authorisation add on eggs available for Django that could help deliver elements of these requirements.

This section will end with what we chose to use and the issues that this involved.
Finally some python code! So a look at some of the more generically useful implementation code, e.g. development of standard object permission decorators for Django class views.

Concluding with where we are now and lessons learned.