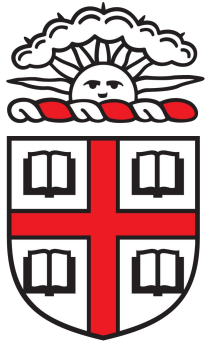


BROWN

VIVO connect share discover

A new (editing) frontend for VIVO

Hector Correa / hector_correa@brown.edu
Steven McCauley / steven_mccauley@brown.edu
Brown University Library

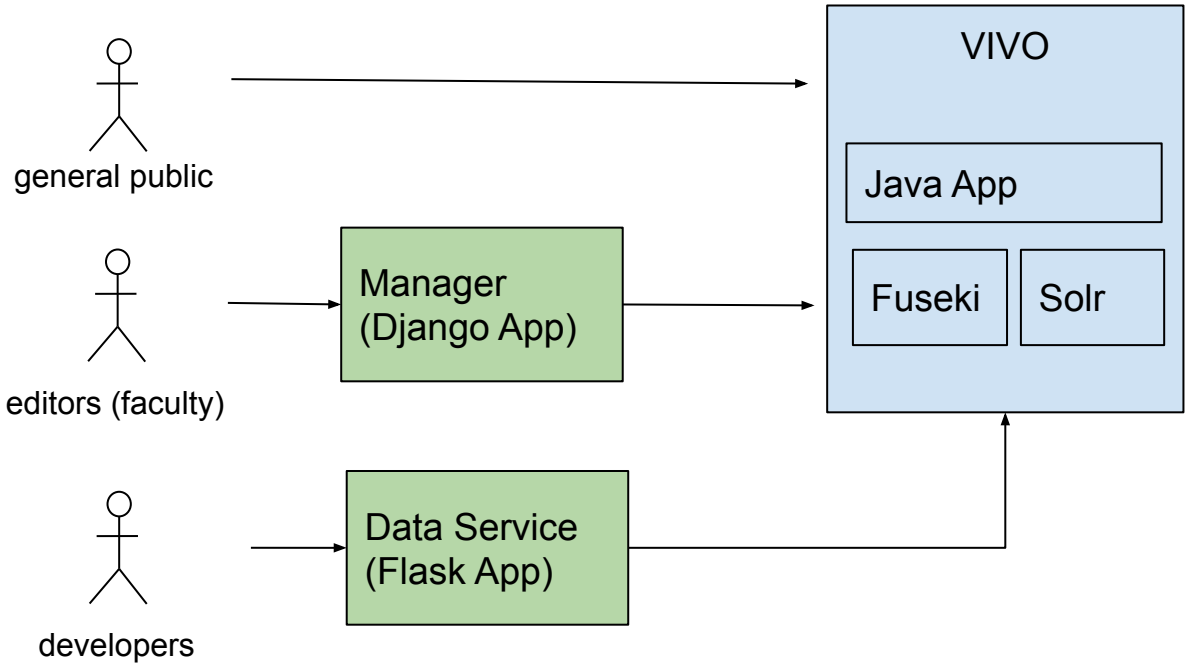
September/2019 - Podgorica, Montenegro

VIVO at Brown

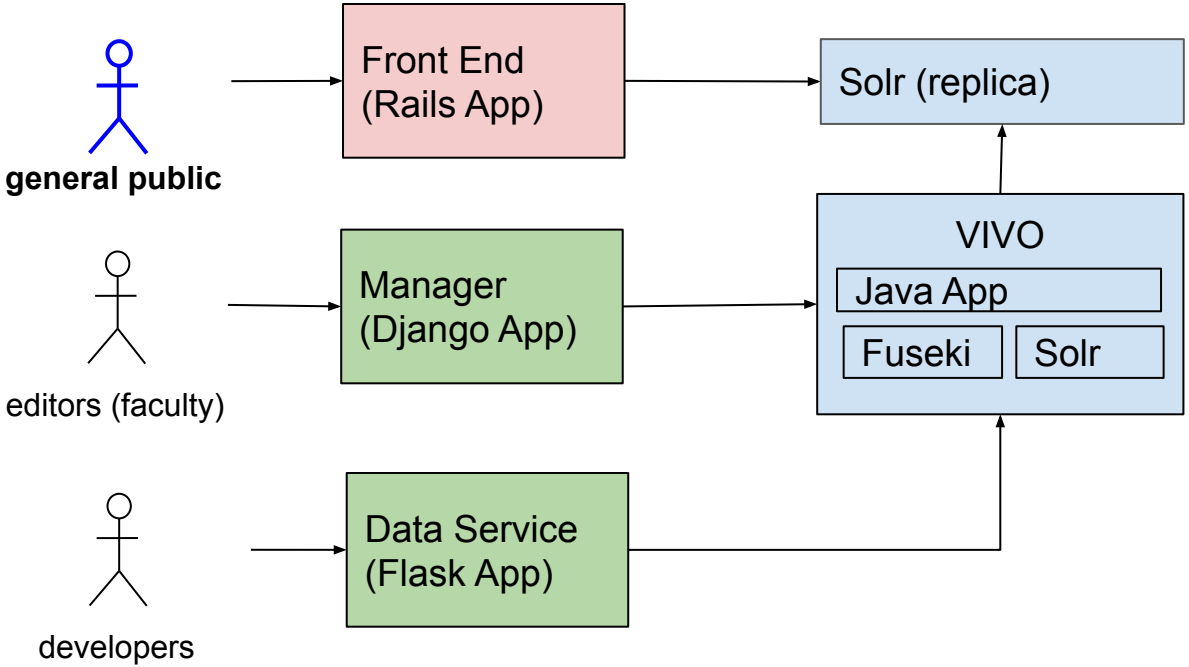
Background

- In production since 2014:
  - VIVO for profile display and browsing
  - Custom editor interface as standalone application
  - 4000+ researchers
- In 2017 we released a new *read-only* frontend (slides)
- In 2018 we added visualizations to new frontend (slides)
- In 2019
  - More visualizations and reports (CSV, Excel)
  - **New *editing* frontend (in progress)**

VIVO at Brown: 2014

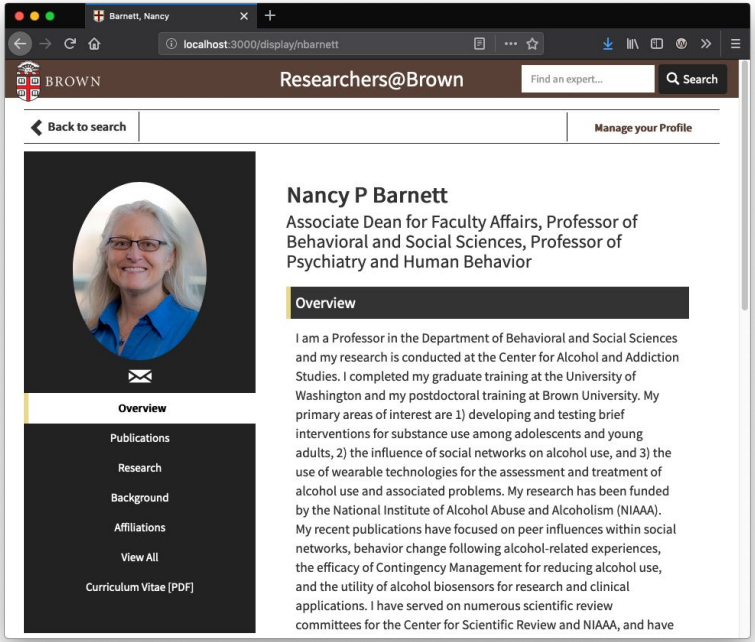


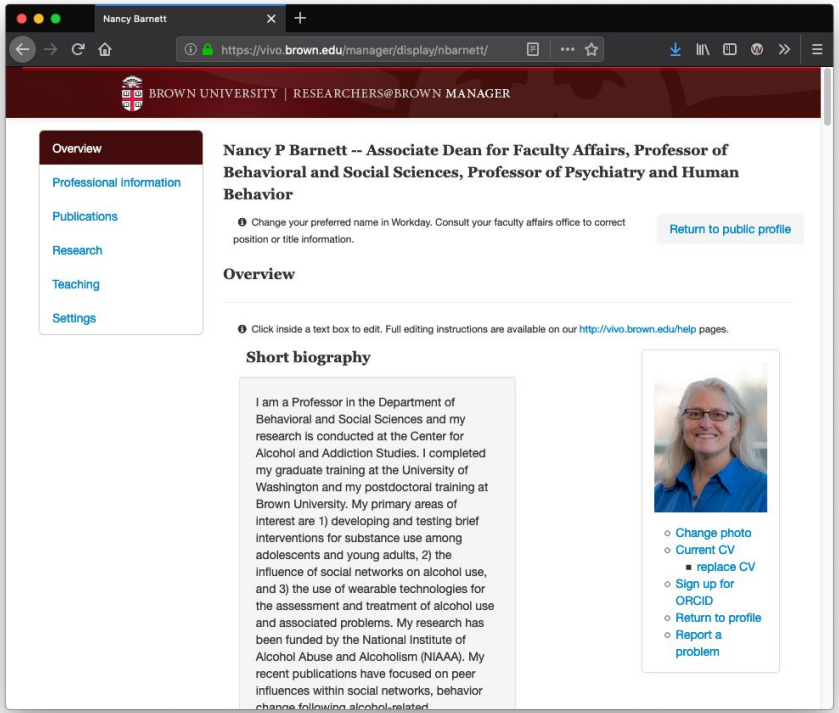
VIVO at Brown: 2017



Demo of current setup
(two apps)

Current setup (two apps)

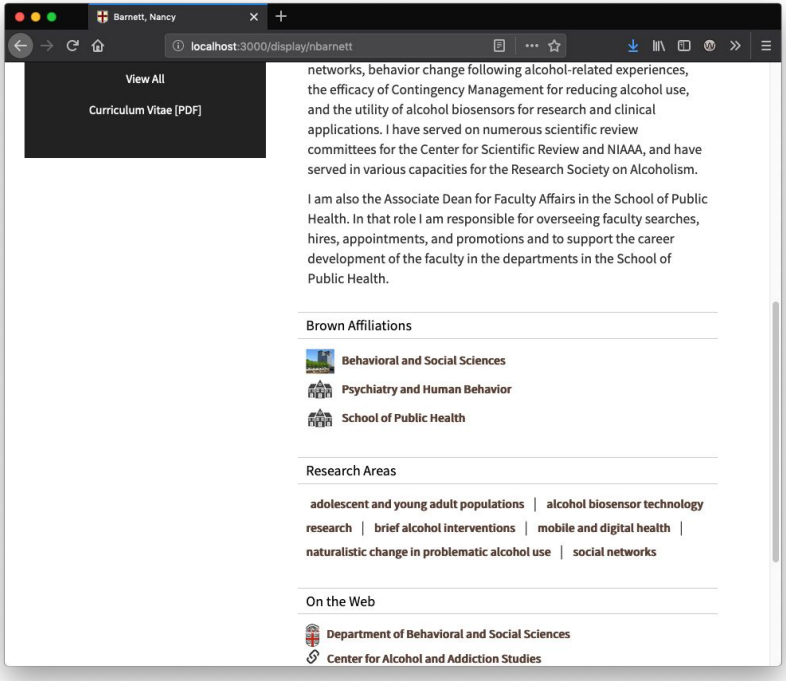


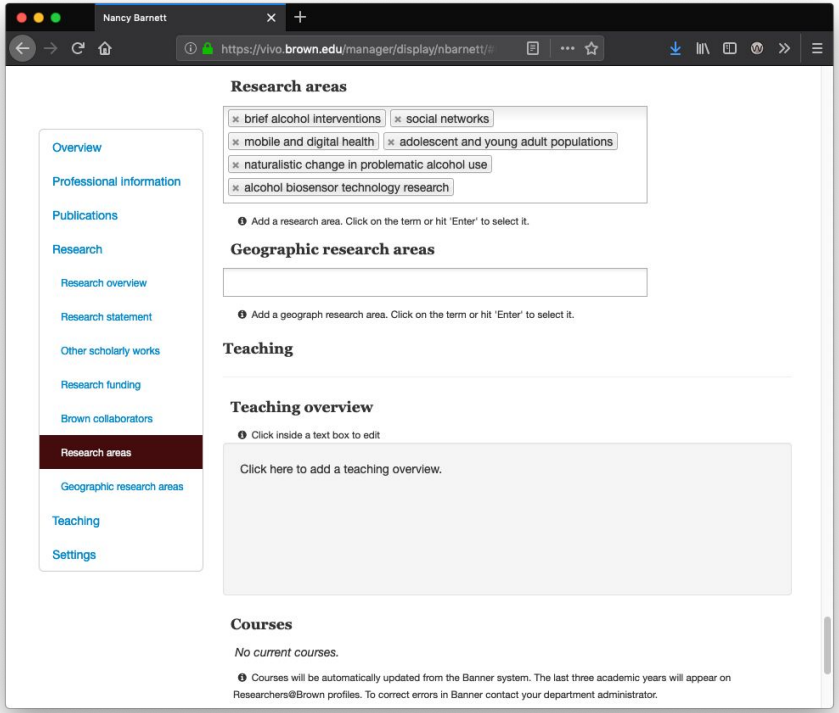


left: viewing frontend

right: current editor frontend

Current setup (two apps)

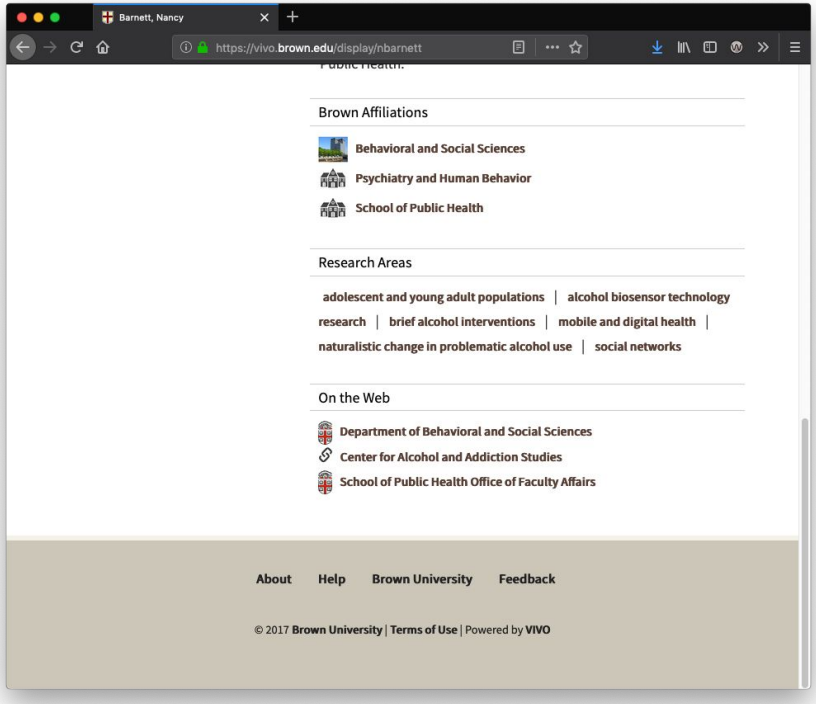


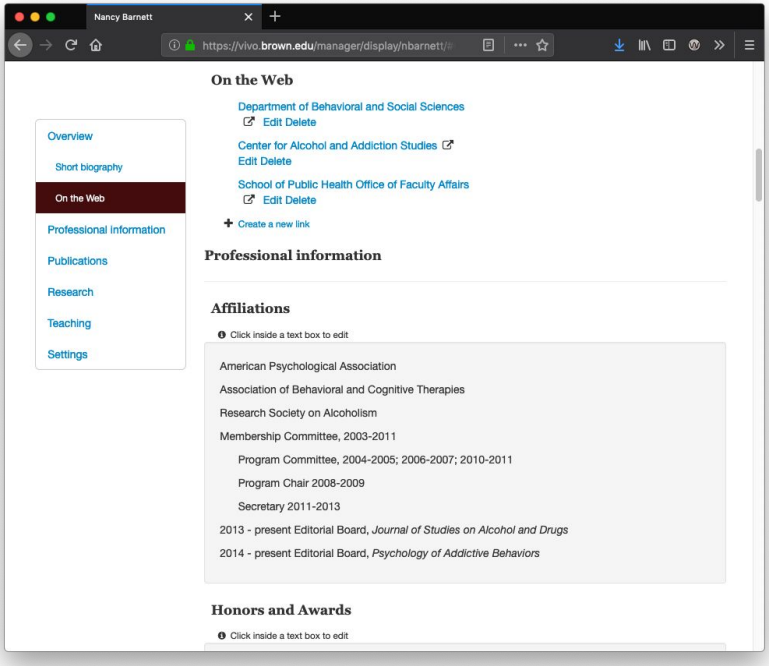


left: viewing frontend

right: current editor frontend

Current setup (two apps)





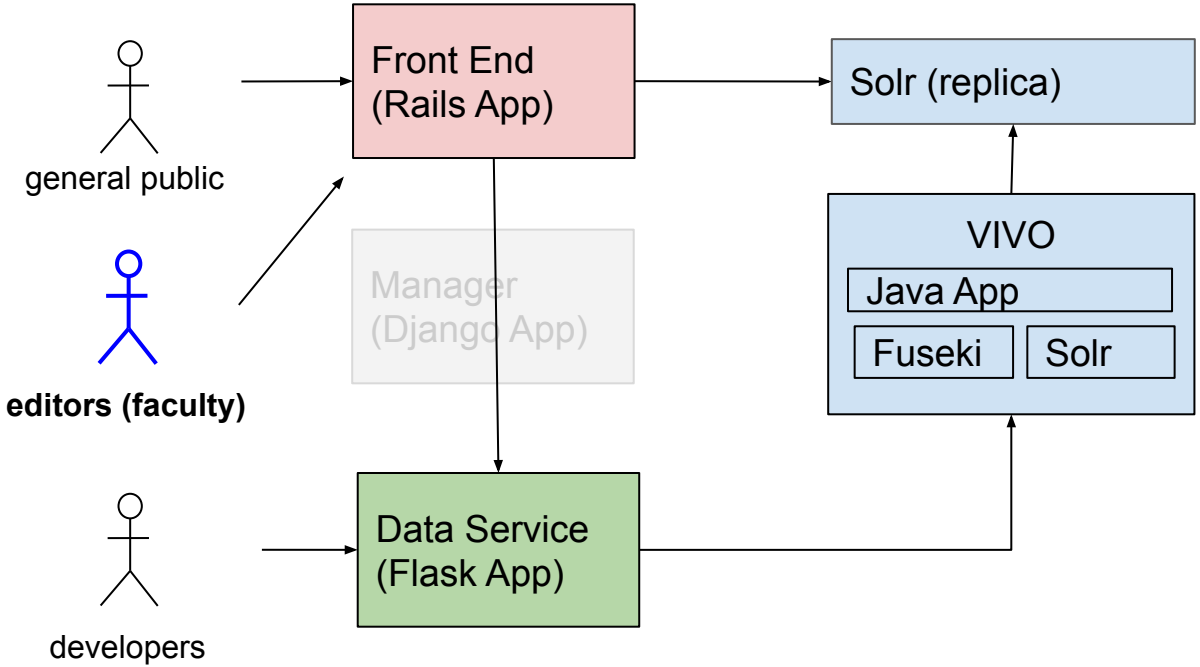
left: viewing frontend

right: current editor frontend

Problems with current setup (frontend)

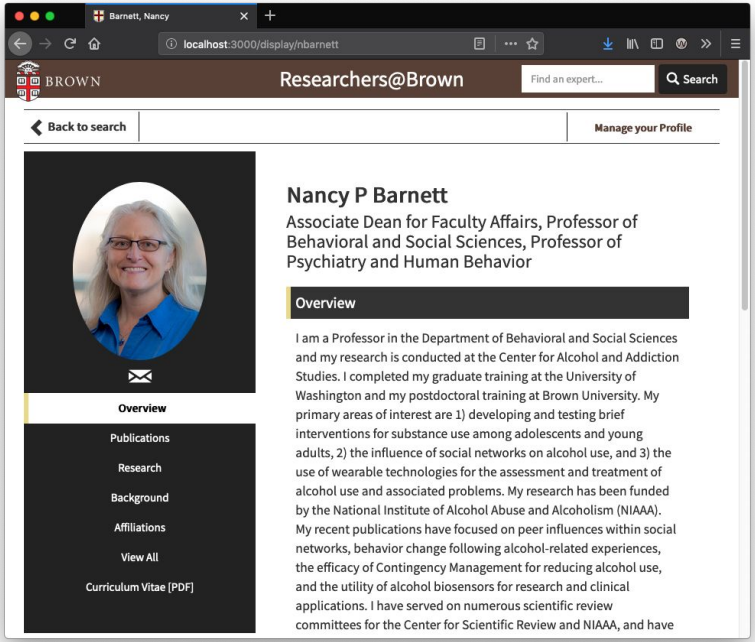
- Mismatch between profile display and profile editing
  - Two different user experiences
  - Two different codebases
- Django application is difficult to manage
  - Misuse of MVC framework
  - Multiple points of interaction with external systems: VIVO triple store, external APIs

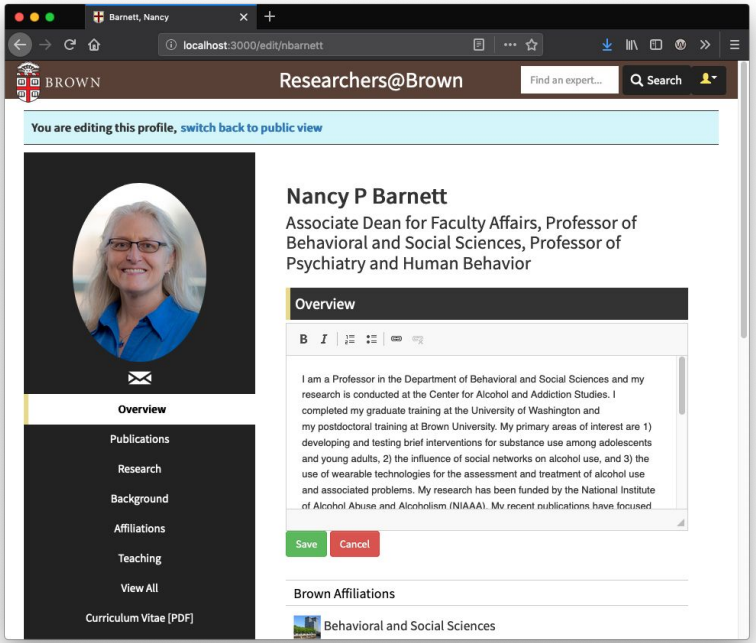
VIVO at Brown: 2019 (later this year)



Demo of new setup
(one app)

New setup (one app)

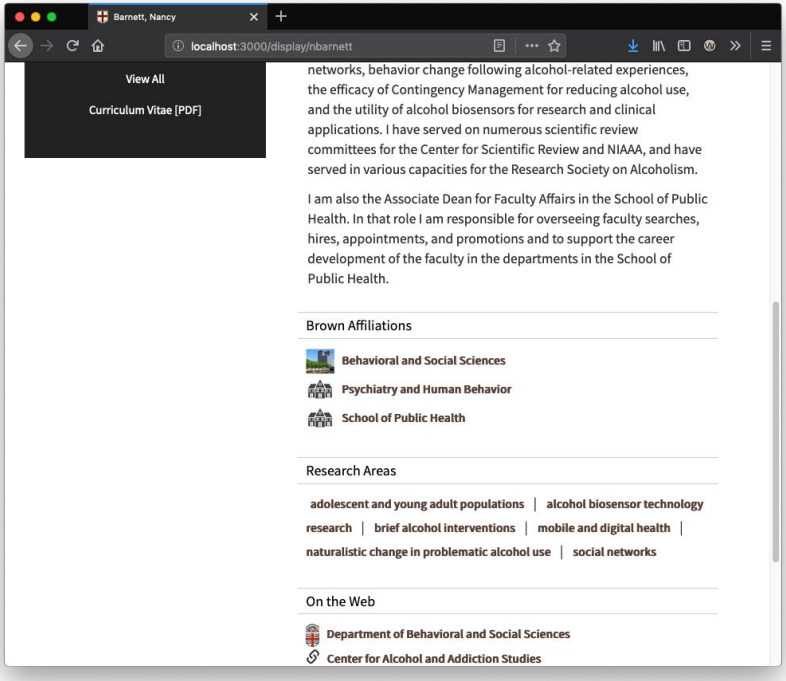


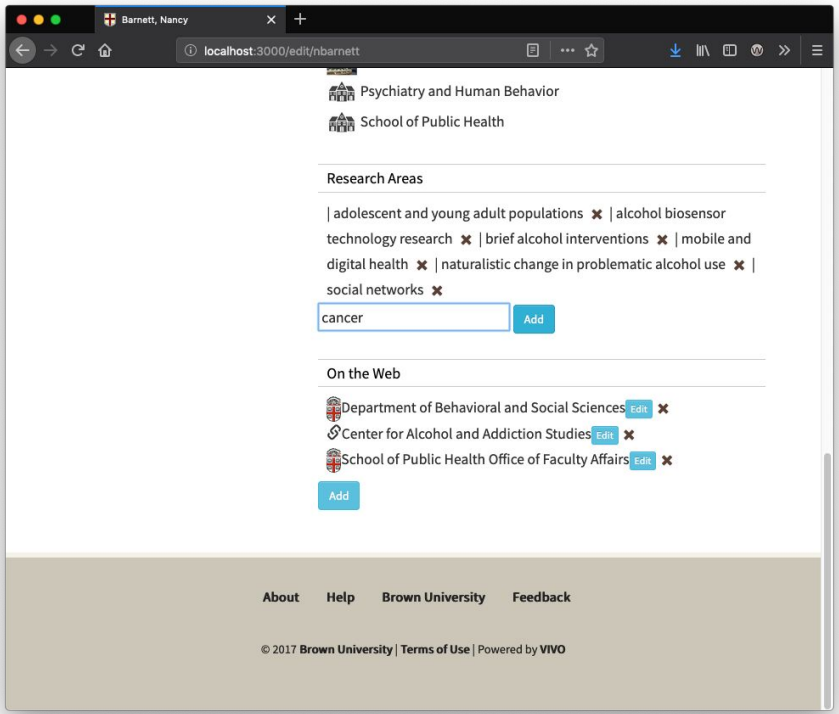


left: viewing frontend

right: new editor frontend

New setup (one app)

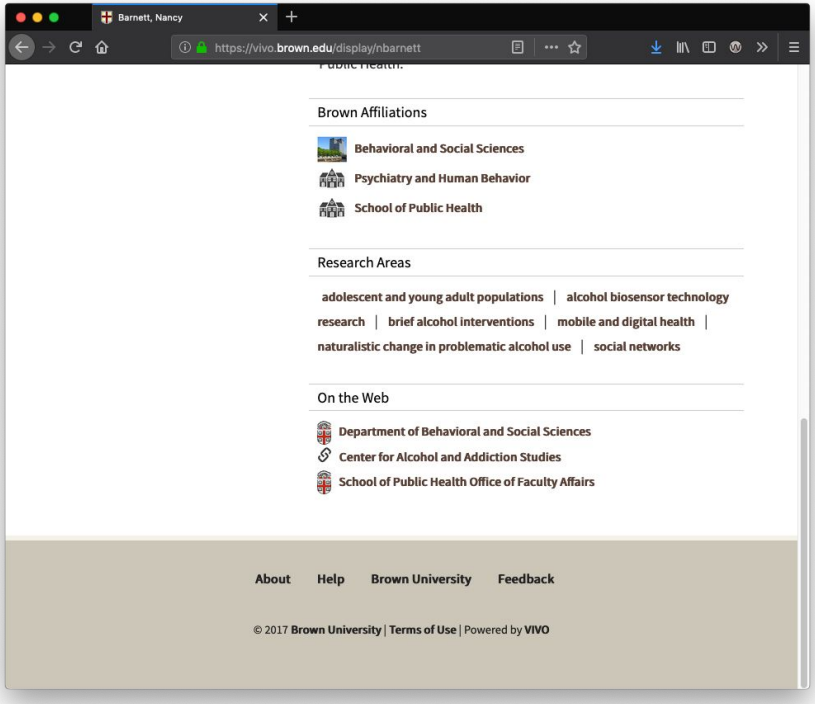


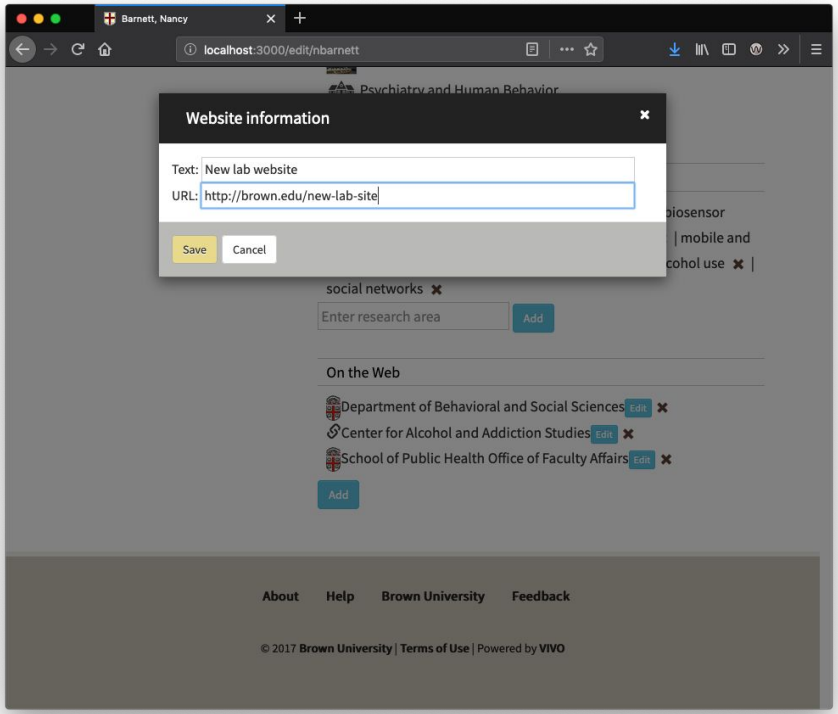


left: viewing frontend

right: new editor frontend

New setup (one app)





left: viewing frontend

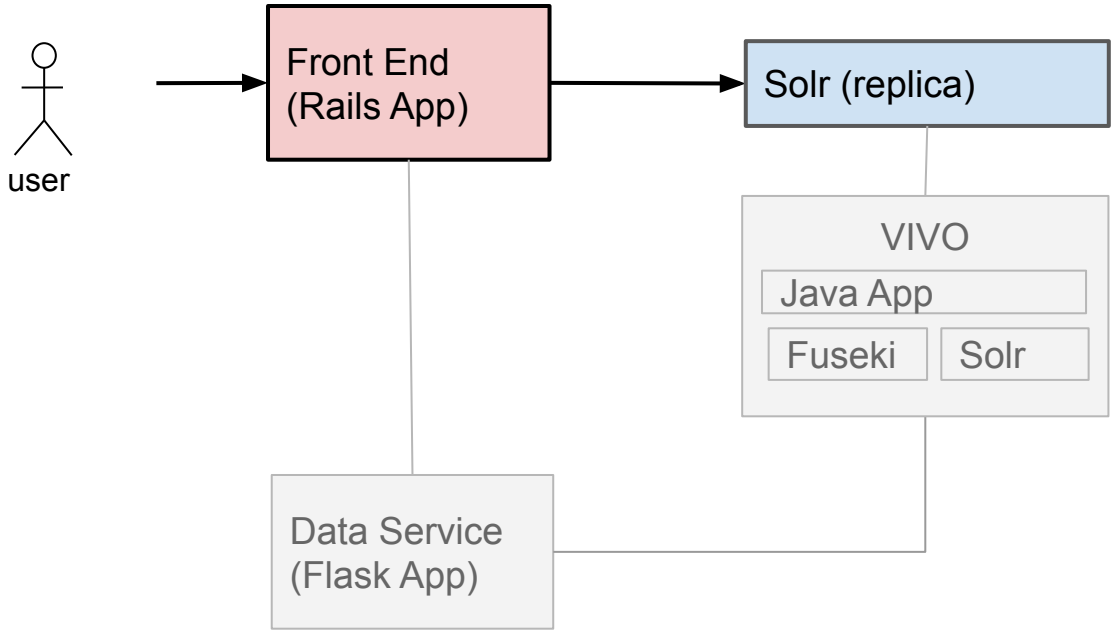
right: new editor frontend

Goals (frontend)

- Better user experience
  - A single frontend for viewing and editing makes sense
  - Benefits for us as developers but more importantly to users
- Easier updates
  - Decoupling user interface, CRUD operations, and other services

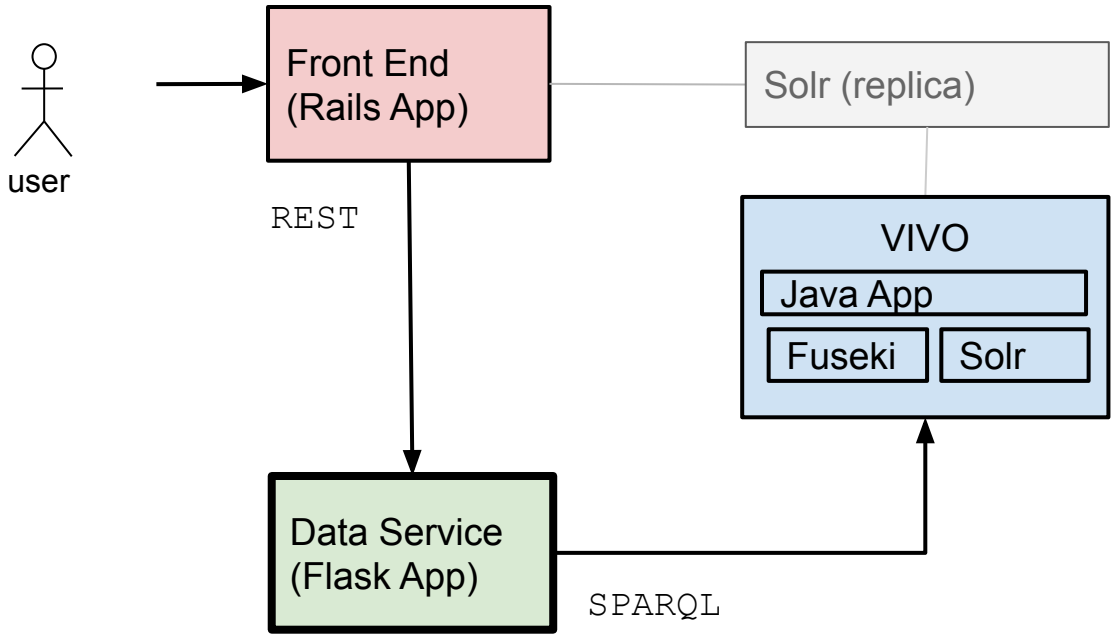
General flow when viewing data

`https://vivo.brown.edu/`**`search`**
`https://vivo.brown.edu/`**`display`**`/user-id`



General flow when editing data

`https://vivo.brown.edu/`**`edit`**`/user-id`



Backend

Backend: current state

Problems

- Multiple points of interaction with triple store
- Custom scripts for RDF generation and SPARQL queries
- Hard-to-debug data anomalies: multiple rdfs:labels, typed and untyped data, unexpected inferencing

Goals

- Single point of interaction: web API for data updates
- High-level API for scripting boilerplate RDF/SPARQL

Tools for scripting RDF (and other graphs)

Prior Work: a brief sample

- VIVO community: Harvester, VIVO Pump
  - https://wiki.duraspace.org/display/VIVO/VIVO+Harvester
  - https://wiki.duraspace.org/display/VIVO/VIVO+Pump
- rdflib (and Resource module)
  - https://rdflib.readthedocs.io/en/stable/index.html
  - https://rdflib.readthedocs.io/en/stable/apidocs/rdflib.html#module-rdflib.resource
- SuRF
  - https://pythonhosted.org/SuRF/
- Neo4j-OGM
  - https://neo4j.com/docs/ogm-manual/current/

Desired function: transactional edits on individual resources

- Common pattern in MVC web frameworks: the Model and ORM

Object-Relational Mapper: Model + CRUD workflow

```
from app import models, db

@route('/edit/<userID>/overview')
def update_overview(userID):
    faculty = models.Faculty.query \
                  .filter_by(id=userID).first()
    faculty.overview = "My new overview"
    db.session.add(faculty)
    db.session.commit()
    return {'overview': faculty.overview}
```

```
from app import db

class Faculty(Model):
    __table__ = 'faculty'

    id = db.Column(db.Integer, key=True)
    overview = db.Column(db.String)
    statement = db.Column(db.String, length=500)
    research_areas = db.Column(db.Integer,
        relationship='ResearchArea')
    weblinks = db.Column(db.Integer,
        relationship='WebLink')
```

TABLE 'faculty'

id	overview	statement	research_area	weblinks
1	"My old overview"	"My statement"	17	13

*Based on general pattern using Flask-SQLAlchemy: https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart/

Designing an ORM for RDF ("R" is for "Resource")

- Lack of ORM-style interface for RDF triplestores is both a source of bugs, and an impediment to adoption and development
- RDF and SPARQL operations map neatly onto basic CRUD operations
- Schemaless, stateless design of RDF triplestores eliminates much of the complexity contained in ORM
- Implementing the basic functions of an ORM for an RDF triplestore is an achievable goal, and would be a great benefit to RDF development

RDF and ORM workflow: transactions and commits

```
def update_overview(userID):
    …
    …
    db.session.commit()
```

```
msg = '''
    PREFIX
    DELETE DATA {
      GRAPH                                {

      }
    }
    INSERT DATA {
      GRAPH                                {

      }
    }
'''
http.post('http://localhost:8080/vivo/api/sparqlUpdate', data={'update': msg})
```

- SPARQL request is platform-agnostic: no need for middleware to communicate with storage layer
- HTTP request establishes transaction scope
- Scripting INSERT/DELETE graph only requires knowing which triples to put where (and associated named graphs)
- Using HTTP for transactions carries performance questions, especially for bulk updates
- https://wiki.duraspace.org/display/VIVODOC19x/SPARQL+Update+API

RDF and ORM workflow: transactions and commits

```
def update_overview(userID):
    …
    db.session.add(faculty)
    db.session.commit()
```

```
msg = '''
    PREFIX brown <http://vivo.brown.edu/profile/>
    DELETE DATA {
      GRAPH <http://vivo.brown.edu/data> {
        <http://vivo.brown.edu/individual/steve> brown:overview "My old overview"^^xsd:string .
      }
    }
    INSERT DATA {
      GRAPH <http://vivo.brown.edu/data> {
        <http://vivo.brown.edu/individual/steve> brown:overview "My new overview"^^xsd:string .
      }
    }
'''
http.POST('http://localhost:8080/vivo/api/sparqlUpdate', data={'update': msg})
```

- SPARQL request is platform-agnostic: no need for middleware to communicate with storage layer
- HTTP request establishes transaction scope
- Scripting INSERT/DELETE graph only requires knowing which triples to put where (and associated named graphs)
- Using HTTP for transactions carries performance questions, especially for bulk updates
- https://wiki.duraspace.org/display/VIVODOC19x/SPARQL+Update+API

RDF and ORM workflow: Resource definition

```
from app import db

class Faculty(Model):
    __table__ = 'faculty'

    id = db.Column(db.Integer, key=True)
    overview = db.Column(db.String)
    statement = db.Column(db.String, length=500)
    research_areas = db.Column(db.Integer,
        relationship='ResearchArea')
    weblinks = db.Column(db.Integer,
        relationship='WebLink')
```

```
from app import db

class Faculty(Resource):
    graph = 'http://vivo.brown.edu/data'
    id = db.URI
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks'...
```

id	overview	statement	...
1	"My old overview"	"My statement"	...

<http://.../steve>	brown:overview	"My old overview"
<http://.../steve>	brown:statement	"My statement"
<http://.../steve>	...	...

- Mapping graph data to dictionary- or object-like structures is pretty standard: https://www.python.org/doc/essays/graphs/
- Relational DB table row corresponds to subset of triplestore : set of triples with shared subject URI
- Row ID is Subject, Columns are Properties, and Column values are Objects

Resource loading: dynamic attributes

```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

OPTIONAL clauses

```
msg = '''
  PREFIX brown <http://vivo.brown.edu/profile/>

  CONSTRUCT {
    ?uri brown:overview ?overview .
    ?uri …
  }
  WHERE {
    ?uri rdf:type brown:Faculty .
    OPTIONAL { ?uri brown:overview ?overview .}
    OPTIONAL { …
  }
```

- In triplestore, there are no placeholders or empty Column cells: triples either exist, or do not exist
- RDF is extremely flexible, but unpredictable: no way to know ahead of time what triple statements exist for a particular Resource
- Previously handled using OPTIONAL and UNION clauses, which are functional but suboptimal

Resource loading: dynamic attributes

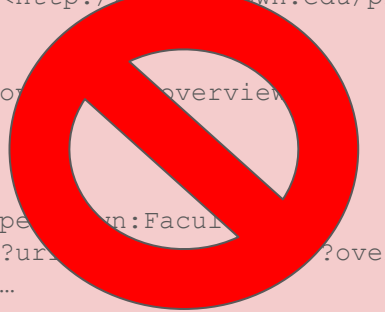
```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

OPTIONAL clauses

```
msg = '''
  PREFIX brown <http://…brown.edu/profile/>

  CONSTRUCT {
    ?uri brown:o…overvie…
    ?uri …
  }
  WHERE {
    ?uri rdf:type …n:Facul…
    OPTIONAL { ?uri …?overview .}
    OPTIONAL { …
  }
```



- In triplestore, there are no placeholders or empty Column cells: triples either exist, or do not exist
- RDF is extremely flexible, but unpredictable: no way to know ahead of time what triple statements exist for a particular Resource
- Previously handled using OPTIONAL and UNION clauses, which are functional but suboptimal

Resource loading: dynamic attributes

```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

UNION clauses

```
msg = '''
  PREFIX brown <http://vivo.brown.edu/profile/>

  CONSTRUCT {
    ?uri brown:overview ?overview .
    ?uri …
  }
  WHERE {
    ?uri rdf:type brown:Faculty .
    } UNION {
        ?uri rdf:type brown:Faculty
        ?uri brown:overview ?overview .
    } UNION { ...
  }
```

- In triplestore, there are no placeholders or empty Column cells: triples either exist, or do not exist
- RDF is extremely flexible, but unpredictable: no way to know ahead of time what triple statements exist for a particular Resource
- Previously handled using OPTIONAL and UNION clauses, which are functional but suboptimal

Resource loading: dynamic attributes

```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

UNION clauses

```
msg = '''
  PREFIX brown <http://…brown.edu/profile/>

  CONSTRUCT {
    ?uri brown:o…overvie
    ?uri …
  }
  WHERE {
    ?uri rdf:type …n:Facul…
    } UNION {
        ?uri rdf:type …ity
        ?uri brown:overview ?overview .
    } UNION { ...
  }
```

- In triplestore, there are no placeholders or empty Column cells: triples either exist, or do not exist
- RDF is extremely flexible, but unpredictable: no way to know ahead of time what triple statements exist for a particular Resource
- Previously handled using OPTIONAL and UNION clauses, which are functional but suboptimal

Resource loading: dynamic attributes

```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

```
def update_overview(userID):
    faculty = models.Faculty.query...
```

```
msg = "DESCRIBE <http://.../steve>"
data = http.post(
    'http://localhost:8080/vivo/api/sparqlQuery',
     data={'query': msg})
```

<http://.../steve>	brown:overview	...
<http://.../steve>	obo:71003401	...
<http://.../steve>	brown:statement	...
<http://.../steve>	obo:04300002	...

- DESCRIBE query returns full subset of triples with shared subject URI
- Attributes of Resource object act as filters on subset, returning only those triples with mapped properties
- DESCRIBE is performant, complete, and makes no assumptions about state of triples

Resource loading: dynamic attributes

```
from app import db

class Faculty(Resource):
    …
    overview = db.Property(uri='brown:overview',
        datatype=db.String)
    statement = db.Property(uri='brown:statement',
        datatype=db.String, length=500)
    research_areas = db.Property(uri='brown:topics',
        relationship='ResearchArea')
    weblinks = db.Property(uri='brown:weblinks',
        relationship='WebLink')
```

```
def update_overview(userID):
    faculty = models.Faculty.query...
```

```
msg = "DESCRIBE <http://.../steve>"
data = http.post(
    'http://localhost:8080/vivo/api/sparqlQuery',
     data={'query': msg})
faculty = models.Faculty.load(data)
```

<http://.../steve>	brown:overview	...
<http://.../steve>	obo:71003401	...
<http://.../steve>	brown:statement	...
<http://.../steve>	obo:04300002	...

- DESCRIBE query returns full subset of triples with shared subject URI
- Attributes of Resource object act as filters on subset, returning only those triples with mapped properties
- DESCRIBE is performant, complete, and makes no assumptions about state of triples

RDF and ORM workflow: the rest

```
faculty = models.Faculty.query \
              .filter_by(id='steve').first()
faculty.overview = "My new overview"
db.session.add(faculty)
db.session.commit()
```

Query filtering

```
faculty = models.Faculty.query.filter_by(id='steve').first()
```

```
msg = '''
    DESCRIBE ?uri
    WHERE {
        ?uri rdf:type brown:Faculty.
        ?uri brown:id "steve"^^xsd:string .
    }
'''
```

Attribute update

```
faculty.overview = "My new overview"
```

```
remove = [(<http://vivo.brown.edu/individual/steve>,<brown:overview>,"\\"My old overview\\"^^xsd:string")]
add = [(<http://vivo.brown.edu/individual/steve>,<brown:overview>,"\\"My new overview\\"^^xsd:string")]
```

Designing an ORM for RDF ("R" is for "Resource")

- We can build an interface to triplestores that encourages consistency while maintaining the flexibility of RDF
- Basic functions not covered here: resource creation, deletion, chaining, listing
- Advanced functions: resource loading strategies, ontology/model reflection, complex queries, triplestore middleware
- Current goals: build off prior work, see basic functions through to completion before getting sidetracked on more advanced features

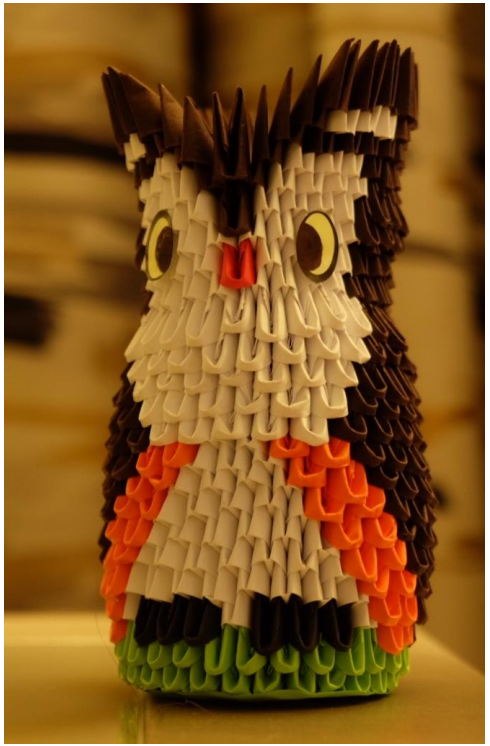
In closing...

Takeaways

- We are using VIVO
  - To store the data
  - To leverage semantic web, linked data, yada-yada-yada
  - Ontology management and other admin functions via the native VIVO app
- Lots of code outside of VIVO
  - Ruby and JavaScript (frontend)
  - Python and SPARQL (backend)
- Keep components separated
  - Display, triplestore access, searching, graph caching
- Fast release cycles
  - Frontend can be deployed at will
  - Backend can be updated at will (as long as the API does not change)

Thanks!

- Live site: https://vivo.brown.edu/ (read-only version)
- Source code
  - Frontend: https://github.com/Brown-University-Library/vivo-on-rails
  - Backend: https://github.com/Brown-University-Library/rab-trax
- Slides: https://tinyurl.com/vivo-2019-brown



[the end]

[backup slides]

Future work

- Renovating newly uncoupled services
  - Publication manager, administrative interface, report generation, email service...
  - API interaction: autocomplete, PubMed, FAST, etc.
- Data migration
  - Cleaning up untyped and otherwise malformed data
  - All triples in named graphs
- Automated processes
  - Bulk updates
- Improving performance
  - Caching, minimizing network calls

Queries: current state

```
class TrainingEditView(FormView, AuthRequiredView, VIVOMixin):
    template_name = 'profile/edit_training.html'
    form_class = TrainingForm
    success_url = '/'

    def build_query(self):
        return """
        PREFIX owl:   <http://www.w3.org/2002/07/owl#>
        CONSTRUCT {
            ?uri ?p ?o .
            ?o rdfs:label ?label
        }
        WHERE {
            { ?uri ?p ?o . }
            UNION {
                ?uri ?p ?o .
                ?p a owl:ObjectProperty .
                ?o rdfs:label ?label
            }
        }
        """
```

```
class EditPublicationsView(FormView, AuthRequiredView, VIVOMixin):
    template_name = 'profile/publications/form.html'
    form_class = DynamicPubForm
    success_url = '/'

    def build_query(self):
        return """
        PREFIX owl:   <http://www.w3.org/2002/07/owl#>
        CONSTRUCT {
            ?uri ?p ?o .
            ?o rdfs:label ?label
        }
        WHERE {
            { ?uri ?p ?o . }
            UNION {
                ?uri ?p ?o .
                ?p a owl:ObjectProperty .
                ?o rdfs:label ?label
            }
        }
        """
```

Issues

- Repeated code
- Query is bound to Django View: can’t isolate, validate, debug query results

Queries: current state

```
def get_credentials(self, local_name):
    query = """
    select ?res ?credText ?credNum
        (year(?start) as ?credStart) (year(?end) as ?credEnd)
        ?accText ?specText
    where {{
        d:{0} bprofile:hasCredential ?res .
        ?res rdfs:label ?credText .
        OPTIONAL {{ ?res bprofile:credentialNumber ?credNum }}
        OPTIONAL {{ ?res bprofile:startDate ?start }}
        OPTIONAL {{ ?res bprofile:endDate ?end }}
        OPTIONAL {{ ?res bprofile:credentialGrantedBy ?acc .
                    ?acc rdfs:label ?accText }}
        OPTIONAL {{ ?res bprofile:hasSpecialty ?spec .
                    ?spec rdfs:label ?specText }}
    }}
    ORDER BY DESC(?credStart)
    """.format(local_name)
    results = self.query(query)
```

Issues

- OPTIONAL statements
- Returning multiple object classes from same query
- Data processing in query
    - sorting, date formatting

Queries: current state

```
def get_matching_concept(self, term, profile_uri):
    q = u"""
    SELECT ?concept
    WHERE
    {{
        {{
            <{0}> <{2}> ?concept .
            ?ra rdfs:label "{1}" .
        }}
        UNION
        {{
            <{0}> <{2}> ?concept .
            ?ra rdfs:label "{1}"^^xsd:string .
        }}
    }}
    """.format(profile_uri, term, self.prop)
    results = vstore.query(q)
```

Issues

- UNION statements
- Typed/untyped data
- Data handling in query

Data editing: current state

```
def write_course_rdf(courseRows):
    statements = []
    for courseRow in courseRows:
        statements.extend([
            (courseRow['courseURI'], RDF.type, VIVO['Course']),
            (courseRow['courseURI'], RDF.type, OWL['Thing']),
            (courseRow['courseURI'], VITRO.mostSpecificType, VIVO['Course']),
            (courseRow['courseURI'], RDFS.label, Literal(courseRow['courseLabel'])),
            (courseRow['courseURI'], VIVO['dateTimeInterval'], courseRow['termURI']),
            (courseRow['teacherURI'], BLOCAL['teacherFor'], courseRow['courseURI']),
            (courseRow['courseURI'], BLOCAL['hasTeacher'], courseRow['teacherURI']),
            ])
    for stmt in statements:
        g.add(stmt)
    print(g.serialize(destination=outFile, format='n3'))
```

Issues

- Manual generation of URI
- Inverse properties: operating on multiple objects

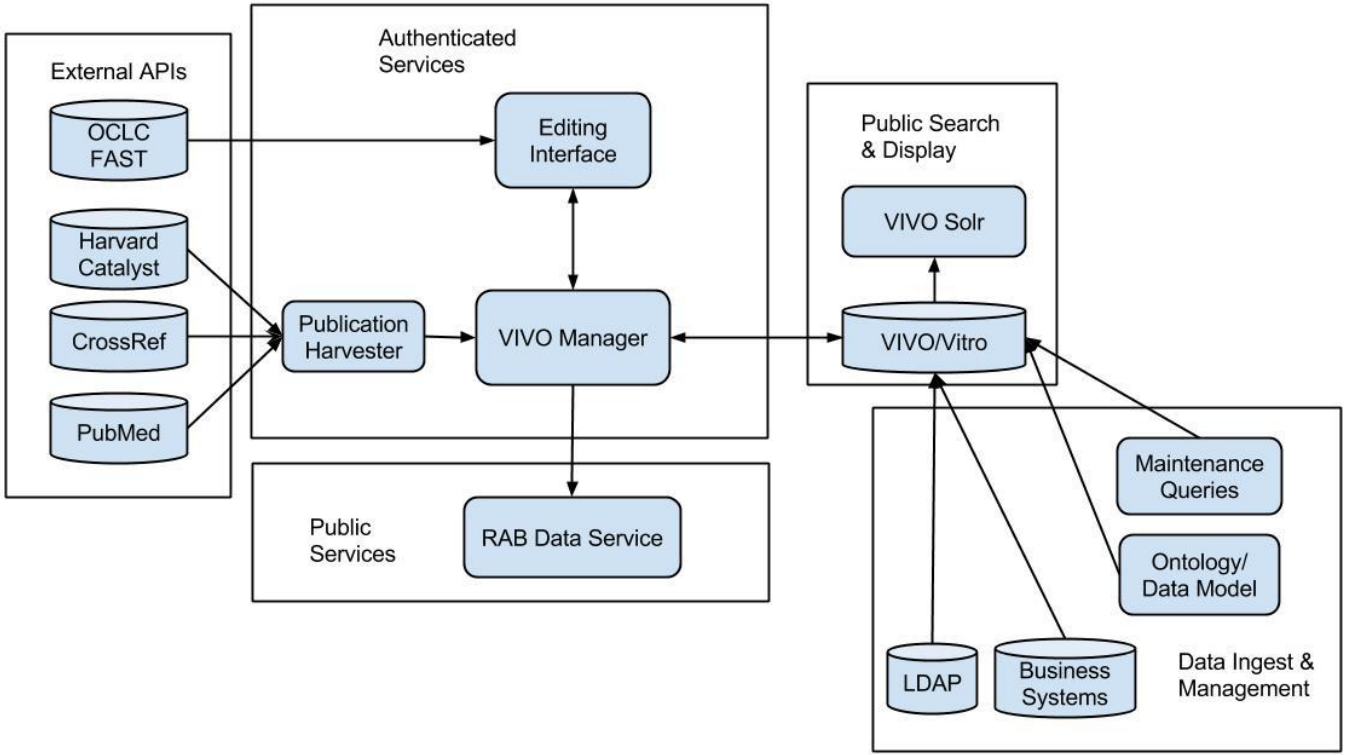
Data editing: current state

```
def process(rows):
    for n, row in enumerate(rows):
        edu_id = hashlib.sha1(edu_hash.encode('utf-8')).hexdigest()
        edu_uri = make_uri("degree" + edu_id)
        add_graph += [
            (edu_uri, RDF.type, VIVO.EducationalTraining),
            (edu_uri, RDFS.label, Literal(degree)),
            (edu_uri, VIVO.educationalTrainingOf, person_uri),
        ]
        if org_uri:
            add_graph.add((edu_uri, VIVO.trainingAtOrganization, org_uri))
        if year:
            add_graph.add(
                (edu_uri, BLOCAL.degreeDate, Literal(year))
            )
    return add_graph
```

Issues

- Manual generation of URI
- Inverse properties: relying on inferencing

Data ecosystem: current state



Resource-oriented SPARQL

```
from app import db
from app.models import FacultyProfile
```

```
db.session.query(FacultyProfile).all()
```

```
DESCRIBE ?uri
WHERE {
        ?uri <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://vivo.brown.edu/ontology/profile#Profile> .
}
```

- DESCRIBE queries leverage built-in indexes for efficiency
- Post-processing of resource attributes
- https://www.w3.org/TR/rdf-sparql-query/#describe

Resource-oriented SPARQL

```
db.session.query(FacultyProfile).get("http://vivo.brown.edu/individual/sdmccaul")
```

```
DESCRIBE <http://vivo.brown.edu/individual/sdmccaul>
WHERE {
        <http://vivo.brown.edu/individual/sdmccaul> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://vivo.brown.edu/ontology/profile#Profile> .
}
```

```
db.session.query(FacultyProfile).filter_by(short_id="sdmccaul", name="Steven McCauley").first()
```

```
DESCRIBE ?uri
WHERE {
        ?uri <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://vivo.brown.edu/ontology/profile#Profile> .
        ?uri <http://vivo.brown.edu/ontology/vivo-brown/shortId> "sdmccaul"^^xsd:string .
        ?uri <http://vivo.brown.edu/ontology/profile#fullName> "Steven McCauley"^^xsd:string .
}
```

Object-Relational Mapper: CRUD workflow

```
from app import models, db

def update_overview(facultyID):
    data = request.json()
    faculty = models.Faculty.query.filter_by(id=facultyID).first()
    faculty.overview = data.get('overview')
    db.session.add(faculty)
    db.session.commit()
    return {'overview': faculty.overview}
```

Problems with current setup (backend)

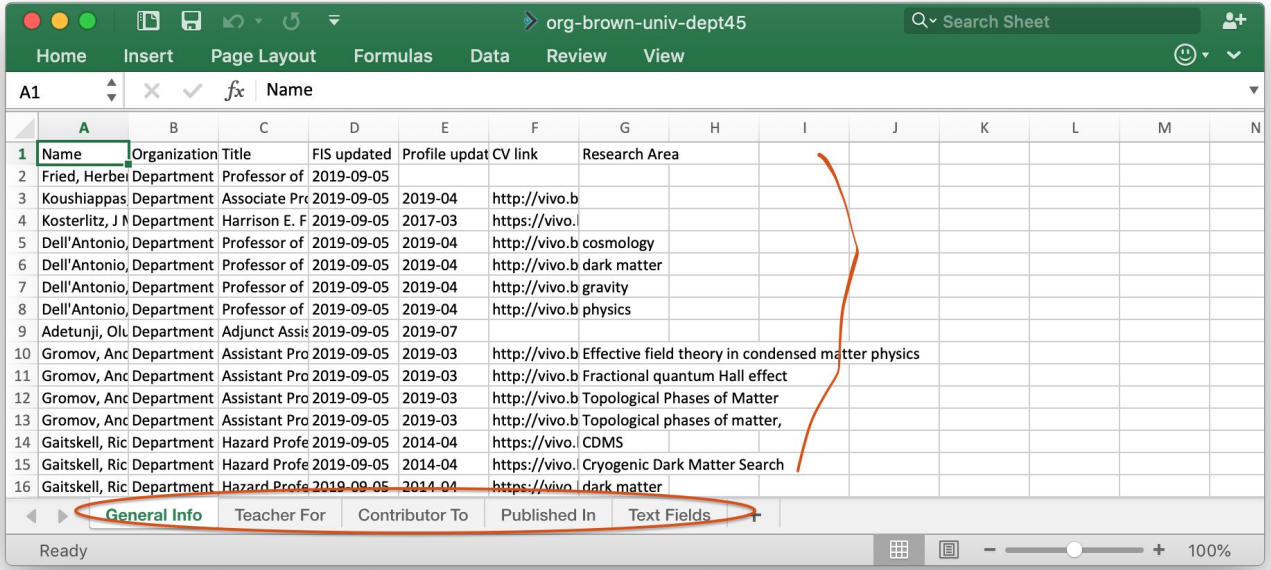
- Django application is difficult to manage
  - Misuse of MVC framework
  - Multiple points of interaction with external systems: VIVO triple store, external APIs
  - Tightly bound with other services: publication harvesting, administrative interface and reports
- Interaction with triple store is unsatisfactory
  - Burrows into VIVO's built-in "primitiveRdfEdit" endpoint, which is designed for built-in forms
- Uncontrolled RDF/SPARQL
  - Mostly produced by hand: irregular and hard to debug
  - Scattered throughout codebase

Visualizations

- General Workflow
  - See slides 2018
- Researcher level samples
  - Coauthor network
  - Coauthor treemap
  - Collaboration network
- Organization level samples
  - Collaborator network
  - Publication history
  - Research areas
- Team level samples
  - (work in progress)

Reports

- We love linked data, but users love spreadsheets
- Created reports for users that output to Excel
- Single Excel file with all the information for a group of researchers



	A	B	C	D	E	F	G
1	Name	Organization	Title	FIS updated	Profile updat	CV link	Research Area
2	Fried, Herbe	Department	Professor of	2019-09-05			
3	Koushiappas	Department	Associate Pr	2019-09-05	2019-04	http://vivo.b	
4	Kosterlitz, J M	Department	Harrison E. F	2019-09-05	2017-03	https://vivo.	
5	Dell'Antonio,	Department	Professor of	2019-09-05	2019-04	http://vivo.b	cosmology
6	Dell'Antonio,	Department	Professor of	2019-09-05	2019-04	http://vivo.b	dark matter
7	Dell'Antonio,	Department	Professor of	2019-09-05	2019-04	http://vivo.b	gravity
8	Dell'Antonio,	Department	Professor of	2019-09-05	2019-04	http://vivo.b	physics
9	Adetunji, Olu	Department	Adjunct Assis	2019-09-05	2019-07		
10	Gromov, Anc	Department	Assistant Pro	2019-09-05	2019-03	http://vivo.b	Effective field theory in condensed matter physics
11	Gromov, Anc	Department	Assistant Pro	2019-09-05	2019-03	http://vivo.b	Fractional quantum Hall effect
12	Gromov, Anc	Department	Assistant Pro	2019-09-05	2019-03	http://vivo.b	Topological Phases of Matter
13	Gromov, Anc	Department	Assistant Pro	2019-09-05	2019-03	http://vivo.b	Topological phases of matter,
14	Gaitskell, Ric	Department	Hazard Profe	2019-09-05	2014-04	https://vivo.	CDMS
15	Gaitskell, Ric	Department	Hazard Profe	2019-09-05	2014-04	https://vivo.	Cryogenic Dark Matter Search
16	Gaitskell, Ric	Department	Hazard Profe	2019-09-05	2014-04	https://vivo.	dark matter

RDF and SPARQL: enforcing consistency

Hand-coding leads to data anomalies and other development headaches

- Multiple values for rdfs:label
- Untyped data and typed data, often for the same property
- Inconsistent inverse properties and inferencing
- Boilerplate queries with subtle variations

RDF development is hampered by the lack of standard tools

- Other DB options offer APIs and libraries for interacting with storage layer
- ORMs hide