# Introduction

## Overview
1. What is Arches? .................................................. 1
2. Who is Arches for? .............................................. 2
3. Documentation Overview ....................................... 2
4. Contributing To Arches ........................................ 2

## Arches Release Process
1. Feature Releases ................................................. 3
2. Patch Releases ................................................... 3
3. Release Support ................................................. 3

## Requirements/Dependencies
1. System Requirements .......................................... 5
2. Software Dependencies ......................................... 5

## Installing Arches
1. Installation ....................................................... 9
2. General Troubleshooting ....................................... 11

## Configuring Settings
1. Quick Start ..................................................... 13
2. Arches Production Deployment ............................... 14
3. Full Explanation of the System Settings UI ............. 16

## Projects and Packages
1. Projects .......................................................... 19
2. Packages ........................................................ 20

## Designing the Database
1. Arches Database Theory ....................................... 25
2. Arches Designer ................................................. 25
3. Graph Designer ................................................. 26

## Ontologies in Arches ........................................... 31

## Creating Resources
1. Resource Manager .............................................. 33
2. Resource Editor ................................................. 34
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3 Related Resources</td>
<td>35</td>
</tr>
<tr>
<td>10 Managing Map Layers</td>
<td>37</td>
</tr>
<tr>
<td>10.1 Different Types of Layers</td>
<td>37</td>
</tr>
<tr>
<td>11 Reference Data Manager (RDM)</td>
<td>39</td>
</tr>
<tr>
<td>11.1 Concept Schemes</td>
<td>39</td>
</tr>
<tr>
<td>12 Arches Collector Manager</td>
<td>47</td>
</tr>
<tr>
<td>12.1 What Are Arches Collector Projects?</td>
<td>47</td>
</tr>
<tr>
<td>12.2 Arches Collector Manager Overview</td>
<td>47</td>
</tr>
<tr>
<td>12.3 Creating Projects in the Arches Collector Manager</td>
<td>48</td>
</tr>
<tr>
<td>12.4 Creating an Offline Basemap</td>
<td>50</td>
</tr>
<tr>
<td>13 Introduction to Arches Collector</td>
<td>51</td>
</tr>
<tr>
<td>13.1 General App Navigation</td>
<td>51</td>
</tr>
<tr>
<td>14 Arches Collector Workflow</td>
<td>53</td>
</tr>
<tr>
<td>14.1 Connect to an Arches Instance</td>
<td>53</td>
</tr>
<tr>
<td>14.2 Download a Project</td>
<td>55</td>
</tr>
<tr>
<td>14.3 Navigating a Project</td>
<td>57</td>
</tr>
<tr>
<td>14.4 Creating Records</td>
<td>60</td>
</tr>
<tr>
<td>14.5 Editing Records</td>
<td>60</td>
</tr>
<tr>
<td>14.6 Syncing Data</td>
<td>67</td>
</tr>
<tr>
<td>15 Additional Configuration</td>
<td>71</td>
</tr>
<tr>
<td>15.1 Password Validators</td>
<td>71</td>
</tr>
<tr>
<td>15.2 Time Wheel Configuration</td>
<td>72</td>
</tr>
<tr>
<td>16 Arches and Elasticsearch</td>
<td>75</td>
</tr>
<tr>
<td>16.1 Using Arches to Install Elasticsearch</td>
<td>75</td>
</tr>
<tr>
<td>16.2 Running Elasticsearch</td>
<td>75</td>
</tr>
<tr>
<td>16.3 Reindexing The Database</td>
<td>76</td>
</tr>
<tr>
<td>16.4 Using An Independent Elasticsearch Installation</td>
<td>76</td>
</tr>
<tr>
<td>16.5 Elasticsearch Index Prefix</td>
<td>77</td>
</tr>
<tr>
<td>16.6 Using Multiple Nodes</td>
<td>77</td>
</tr>
<tr>
<td>17 API</td>
<td>79</td>
</tr>
<tr>
<td>17.1 General Notes</td>
<td>79</td>
</tr>
<tr>
<td>17.2 Authentication</td>
<td>80</td>
</tr>
<tr>
<td>17.3 Concepts</td>
<td>83</td>
</tr>
<tr>
<td>17.4 Resources</td>
<td>85</td>
</tr>
<tr>
<td>17.5 Mobile Projects</td>
<td>90</td>
</tr>
<tr>
<td>18 Command Line Reference</td>
<td>93</td>
</tr>
<tr>
<td>18.1 Installation Commands</td>
<td>93</td>
</tr>
<tr>
<td>18.2 ElasticSearch Management</td>
<td>94</td>
</tr>
<tr>
<td>18.3 Import Commands</td>
<td>95</td>
</tr>
<tr>
<td>18.4 Export Commands</td>
<td>96</td>
</tr>
<tr>
<td>18.5 Other Data Management Commands</td>
<td>97</td>
</tr>
<tr>
<td>18.6 Ontology Commands</td>
<td>98</td>
</tr>
<tr>
<td>18.7 Managing Functions, DataTypes, Widgets, and Card Components</td>
<td>98</td>
</tr>
<tr>
<td>18.8 Creating Map Layers</td>
<td>99</td>
</tr>
<tr>
<td>18.9 Other Useful Django Commands</td>
<td>100</td>
</tr>
</tbody>
</table>
19 Creating a Development Environment
  19.1 Setting Everything Up ....................................................... 101

20 Creating New Functions, Widgets, Card Components, and DataTypes
  20.1 Functions ................................................................. 105
  20.2 Widgets ................................................................. 108
  20.3 DataTypes ............................................................... 111
  20.4 Card Components ....................................................... 122

21 Creating New Map Layers ..................................................... 133
  21.1 MapBox Layers .......................................................... 133
  21.2 Tileserver Vector Layers ............................................... 134
  21.3 Tileserver Mapnik Layers ............................................... 136

22 Data Model ........................................................................ 139
  22.1 Resource Model Overview ................................................ 139
  22.2 Controllers ............................................................... 140
  22.3 Graph Definition ........................................................ 141
  22.4 Ontologies ............................................................... 145
  22.5 RDM Models .............................................................. 147
  22.6 Resource Data ............................................................. 148
  22.7 UI Component Models .................................................... 152
  22.8 Data Model Graph Visualization ....................................... 155

23 Migrating Data from v3 to v4 .................................................. 157
  23.1 Preparation - Exporting v3 Data ........................................ 157

24 Resource Import/Export .......................................................... 167
  24.1 Importing Data ............................................................. 167
  24.2 Importing a CSV .......................................................... 169
  24.3 Shapefile Import .......................................................... 172
  24.4 JSON Import .............................................................. 172
  24.5 Importing Resource Relations .......................................... 178
  24.6 Exporting Arches Data .................................................... 179

25 Settings - Beyond the UI ......................................................... 183

26 Arches HIP Contents ............................................................. 185
  26.1 Resource Models .......................................................... 185
  26.2 Branches ................................................................. 185

27 Glossary ............................................................................. 189

28 Serving Arches with Apache ...................................................... 191
  28.1 Setup Apache .............................................................. 191
  28.2 Handling Static Files ..................................................... 193

HTTP Routing Table ................................................................. 195
CHAPTER 1

Overview

1.1 What is Arches?

Arches is a web-based, geospatial information system for cultural heritage inventory and management. The platform is purpose-built for the international cultural heritage field, and it is designed to record all types of immovable heritage, including archaeological sites, buildings and other historic structures, landscapes, and heritage ensembles or districts.

Arches allows administrators to create their own database schema, and manage their own thesauri, while end users can search, explore and download the resources directly. In this way Arches is not only a robust and easy to use inventory system, it is also a perfect way to publish and disseminate your organization’s cultural heritage information.

Arches is a web framework built on Dango and is designed to make it easier to build applications that need:

- **Geospatial data management** and geoprocessing like a GIS (Geographic Information System) offers, but with a much more flexible approach for modeling the geometries associated with a resource.

- the ability to **import arbitrary data schema** in the form of graphs as a means of defining the set of attributes that describe data resources

- an **Ontology** as a means of formally naming and defining data types, properties, and the relationships between the data entities that describe a resource.

- **Thesauri** to manage the controlled vocabularies needed to describe and index information in a consistent and uniform way.

Arches manages data “resources”. Resources can represent almost anything you want: physical things (such as a cultural heritage object), temporal things (such as activities or events), actors (such as a person or organization), or conceptual objects (such as an image, document, or other information carrier).

Resources are defined as directed graphs (nodes connected by edges). Nodes in the graph are used to represent the attributes (or collection of attributes) of a resource and edges define the type of relationship between attributes. In practice, a resource graph in Arches functions much like a schema does in a relational database.

Arches provides core services for creating, reading, updating, and deleting resources. Because resources are defined as graphs, Arches provides the services needed to import and parse resource graphs, as well the ability to create and interact with instance graphs (e.g. an instance of a resource graph).
To promote consistent data creation, update, and indexing workflows, Arches implements a Reference Data Manager (RDM) that can manage thesauri. The RDM allows users with the appropriate privileges to update thesaurus entries in a manner compliant with SKOS (http://www.w3.org/2004/02/skos/) and assign the concepts within a thesaurus with data entry forms.

**Arches User and Developer forum:** https://groups.google.com/d/forum/archesproject?hl=en

**Version History**

- v4.4 February, 2019: Release of *Arches Collector* mobile data collection app
- v4.0 July, 2017: Significant site redesign, addition of graph creation UI, system settings UI, internal tileserver (TileStache), updated dependencies (MapBox GL, ElasticSearch 5.2, Yarn)
- v3.0 April, 2015: Updated architecture, inclusion of the Reference Data Manager, updated dependencies (ElasticSearch, OpenLayers, Knockout)
- v2.0 March, 2014: Improved upload of digital files, assorted bug fixes
- v1.0 October, 2013: Initial Release

**License**

Arches is free software and is licensed under the terms of the GNU Affero General Public License (http://www.gnu.org/licenses/agpl-3.0.html).

### 1.2 Who is Arches for?

Arches is primarily intended for software developers who need to build flexible web applications and wish to hide the complexities of ontologies, thesauri, and geospatial data management from their users.

### 1.3 Documentation Overview

This is the official documentation for Arches. It should provide you with background information on Arches, how to install it, and a good overview of its capabilities. While you are using Arches, be aware that much of the content here is also available by clicking the “?” symbol in the top-right corner of any page.

**Improve Our Documentation!** If you find errors, have suggestions, or want to make a contribution, these docs are managed in the archesproject/arches-docs repo.

### 1.4 Contributing To Arches

Arches is open source software, which means that with your help it will continue to evolve and improve.

- **Bug Reports and Code Contribution** If you find issues with the Arches interface or code, or have the means to contribute code to fix existing issues, please begin by reading our guidelines for Contributing to Arches.

- **Translations** We are always hoping to bring Arches to new audiences around the world. Please post on the Arches forum if you are interested in contributing a translation.
Starting with version 4.1.0, the Arches team will begin making available both feature (minor) and patch (micro) releases on a regular basis.

### 2.1 Feature Releases

Feature releases will introduce significant, new features to Arches and will be announced approximately every 6 months. Feature releases may contain schema or API changes that may not be compatible with the previous feature release. Each feature release will be incremented with the pattern $a.b$, where $a$ represents the major release and $b$ represents the feature (aka minor) release. Each feature release will be placed in its own branch in git, named with its release number followed by an $x$ representing the latest patch release. (e.g. stable/a.b.x).

### 2.2 Patch Releases

Following each feature release we will resolve bugs, performance, and security issues in the most recent feature release with patch releases. A new patch release, if needed, will be announced every 1 to 3 months and will not include breaking changes with the previous patch release. Therefore, we encourage users to stay up-to-date with these releases. Patch releases will be incremented as such: $a.a.b$, $a.a.c$… with $a$ representing the feature release and $b$ and $c$ representing patch (aka micro) releases. In Git each patch release will identified in its feature release branch with a tag.

### 2.3 Release Support

We will release patches only for the latest feature release.

Feature Release Schedule:

- 4.1 - User account creation, user profile management (released Jan 2018)
  - 4.1.1 - Bug fixes (released May 2018)
• 4.2 - QA workflow (released June 2018)
• 4.3 - Relaxed graph constraint (released Oct 2018)
• 4.4 - Mobile workflow and plugins (anticipated late Jan 2018)
CHAPTER 3

Requirements/Dependencies

3.1 System Requirements

Arches works on Linux, Windows, or macOS, but some of its dependencies may be more difficult to install on certain operating systems. Most enterprise-level installations of Arches have been created on Linux servers.

To begin development or make a test installation of Arches, you will need the following:

- **2GB disk space**
  - ~1.5GB for all dependencies and 600mb for Arches.
  - In production, the amount of disk space you need will depend on the number of resources in your database, specifically uploaded images or media files.

- **4GB memory (RAM)**
  - This recommendation is based on the fact that ElasticSearch requires 2GB to run, and as per official ElasticSearch documentation no more than half of your system’s memory should be dedicated to ElasticSearch.
  - In development, it’s possible to force ElasticSearch to run with only 1GB of memory, see Running Elasticsearch.
  - In production, you may want to increase your memory, and allow ElasticSearch to use up to 32GB of memory.

3.2 Software Dependencies

- **PostgreSQL 9.6 with PostGIS 2.3** (see below)
- **GDAL > 1.11.5** and **GEOS** (see below)
- **Python 2.7** https://www.python.org/downloads/
Note: Python 2.7.9 and later comes with pip, however, with < 2.7.9, you will need to get pip from here: https://pip.pypa.io/en/latest/installing.html. We also recommend upgrading pip with this command `python -m pip install --upgrade pip`.

- **Mapnik 2.2** [http://mapnik.org/pages/downloads.html](http://mapnik.org/pages/downloads.html). *Windows users, see below.*
- **Elasticsearch 5.x**
  - To learn more about how Arches works with ElasticSearch, please read *Using Arches to Install Elasticsearch*. We recommend waiting until *after* you have Arches installed to deal with Elasticsearch.
  - CouchDB is used to sync mobile data collection projects with your main database. *If you never plan to use the Arches Collector app, you can skip this dependency.*
  - Arches will connect to CouchDB with default credentials. However, in production you must make a new CouchDB admin after installation, and then update `COUCHDB_URL = 'http://admin:admin@localhost:5984'` in *settings.py* accordingly.

The installation process for each component listed may differ based on your operating system, so please check below for more info.

### 3.2.1 Installing Dependencies on Linux

You can find fully scripted dependency installation for Ubuntu here.

### 3.2.2 Installing Dependencies on Windows

Be aware that you must install 32-bit or 64-bit versions of all dependencies based on your Windows system’s architecture. In our experience, the following installation order should work well.

**PostgreSQL 9.6 with PostGIS 2.3** - Use the EnterpriseDB installers, and use Stack Builder (included) to get PostGIS. After installation, add the following to your system’s PATH environment variable: `C:\Program Files\PostgreSQL\9.6\bin`.

**Important:** Remember to note the password that you set for the default *postgres* user, as you will need to put it in *settings.py* later.

**GDAL and GEOS** - Use the OSGeo4W installer found here: [https://trac.osgeo.org/osgeo4w/](https://trac.osgeo.org/osgeo4w/), and choose to install the GDAL package (you don’t need to install QGIS or GRASS). After installation, add the following to your system’s PATH environment variable: `C:\OSGeo4W64\bin`.


**Mapnik** - *If you are on 64-bit Windows you will not be able to install Mapnik; the Python bindings are not compatible.* Arches is fully functional without Mapnik, but some of the extra map overlay capabilities are lost. Specifically, if you are adding a Tilestache layer as a map overlay, you will not be able to use any of the Mapnik-based providers as described in this section of the Tilestache documentation.
JDK - Use the installers found here: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. Once installed, find Java on your operating system. It will be somewhere like C:\Program Files\Java\jdk*.\*.\*. Now take that full path, and add it to the JAVA_HOME system environment variable.

Python 2.7 - We recommend using 2.7.13, found here: https://www.python.org/downloads/ You must choose 32-bit or 64-bit Python depending on your operating system architecture.

pip - Python 2.7.13 comes with pip already, but we do recommend upgrading pip:

```
python -m pip install --upgrade pip
```

Elasticsearch - Take care of this after you have Arches fully installed. Use the command recommended in Using Arches to Install Elasticsearch.


3.2.3 Installing Dependencies on macOS

Please see this gist for the easiest way to install all dependencies (besides CouchDB on macOS. Running this script in full will create an installation of Arches based on the current repo, but you can use pieces of it for individual dependencies as well.

Further notes:

- **PostgreSQL 9.6 with PostGIS 2.3** - Use Postgres.app.
- **GEOS** - Use version 3.6.1 (3.6.2 has caused trouble on macOS).
4.1 Installation

Once you have all the dependencies taken care of you are ready to install Arches.

Note: If you are a developer hoping to contribute to the Arches project, please check out *Creating a Development Environment*.

4.1.1 1. Create and enter a new directory called Projects

```
mkdir Projects && cd Projects
```

4.1.2 2. Create a virtual environment called ENV

```
pip install virtualenv
virtualenv ENV
```

4.1.3 3. Activate the virtual environment

Linux and macOS:
```
source ENV/bin/activate
```

Windows:
```
ENV\Scripts\activate
```
When you activate your virtual environment, your command prompt will be prefixed with `(ENV)`. If you need to deactivate it for any reason, run `deactivate`.

### 4.1.4 4. Install Arches into your virtual environment

Use the following to get the latest stable release of Arches:

```
pip install arches --no-binary :all:
```

*Warning:* Running a pip installation without the `--no-binary :all:` argument will install `arches` in your virtual environment, but will not install all of the separate python packages that you need. This will cause you to not be able to proceed with the following steps.

### 4.1.5 5. Create a new Arches Project

Now that Arches is installed, you can make a Project. This is where you will make all of the customizations and branding that will make one installation of Arches different from the next. The name of your project should only be lowercase, and should use underscores instead of spaces or hyphens. The example below uses `my_project`. You can add `--directory path/to/dir` to change the directory your new project will be created in.

Linux and macOS:

```
arches-project create my_project
```

Windows:

```
python ENV\Scripts\arches-project create my_project
```

*Note:* At this level, “projects” are completely different from the mobile data collection “projects” that are mentioned elsewhere in this documentation.

### 4.1.6 6. Update settings_local.py

Before you continue, you may need to open `my_project\my_project\settings_local.py` and change some environment-specific settings.

- Depending on your Postgres/PostGIS installation, you may need to uncomment the `DATABASES` section and change the username and password, as well as the name of your PostGIS template database. If you will have multiple Arches projects using the same Postgres server, make sure to change the database name here as well.
- On Windows, you must specify the path to your GDAL library. Add the following line. Be sure to adjust the path as necessary for your GDAL installation, and note the *forward* slashes:

  ```python
  GDAL_LIBRARY_PATH = "C:/OSGeo4W64/bin/gdal201.dll"
  ```

- See Arches System Settings for a full rundown of how settings are handled in Arches.

*Note:* At this point you must install ElasticSearch and make sure it is running before continuing.
4.1.7 7. Setup the database

To begin working with your Project, first enter it:

```
cd my_project
```

and then (with ElasticSearch running) run:

```
python manage.py packages -o setup_db
```

**Note:** You may be prompted to enter a password for the `postgres` user, depending on your OS and how you installed Postgres/Postgis. Generally, our installation scripts set this password to `postgis`.

4.1.8 8. Run the development server

To confirm things are working, run:

```
python manage.py runserver
```

and navigate to `localhost:8000` in a browser.

The default login credentials are 'admin' for both username and password. To change this password and create new users, navigate to `localhost:8000/admin`.

4.1.9 Next steps…

Now that you have Arches installed and a Project created, you are ready to begin.

- Modify some Initial Configuration settings
- In order to allow API access to your instance, to support the use of the Arches Collector mobile app, you must register an Arches application.
- Read more about Projects and Packages
- For a quick start that will create an example database schema in your new Arches project (and allow you to begin recording data right away), you can load a sample package with the following command

```
python manage.py packages -o load_package -s https://github.com/archesproject/arches4-example-pkg/archive/master.zip -db true
```

This example package has a full set of Resource Models, Branches, and Concepts that are generally based on the original Arches HIP Contents.

**Note:** If you get an “invalid package source” error, please download the zipfile (use the url in the command) and run the same command pointing to the location of the downloaded zipfile.

4.2 General Troubleshooting

- On macOS, If you get this error
**Error:** `ValueError: --enable-zlib requested but zlib not found, aborting.`

Try running `xcode-select --install` (reference)

- Getting a connection error like this (in the dev server output or in the browser)

**Error:** `ConnectionError: ConnectionError(<urllib3.connection.HTTPConnection object at 0x0000000005C6BC50>: Failed to establish a new connection: [Errno 10061] No connection could be made because the target machine actively refused it) caused by: NewConnectionError(<urllib3.connection.HTTPConnection object at 0x0000000005C6BC50>: Failed to establish a new connection: [Errno 10061] No connection could be made because the target machine actively refused it)`

means Arches is not able to communicate with ElasticSearch. Most likely, ElasticSearch is just not running, so just start it up and reload the page. If you can confirm that it *is* running, make sure Arches is pointed to correct port.

- Postgres password authentication error

**Error:** `django.db.utils.OperationalError: FATAL: pw authentication failed for user postgres`

Most likely you have not correctly set the database credentials in your `settings.py` file. Many of our install scripts set the db user to `postgres` and password to `postgis`, so that’s what Arches looks for by default. However, if you have changed these values (particularly if you are on Windows and had to enter a password during the Postgres/PostGIS installation process), the new values must be reflected in in `settings.py` or `settings_local.py`.

**Note:** On Windows, you can avoid having to repeatedly enter the password while running commands in the console by setting the `PGPASSWORD` environment variable: `set PGPASSWORD=<your password>`. 

---

12 Chapter 4. Installing Arches
CHAPTER 5

Configuring Settings

5.1 Quick Start

Once Arches is installed, there are a few settings you must configure to make it fully operational. In the Arches interface, head to the System Settings menu (you must be logged in as the admin user, default password is admin).

The most important settings to begin with are related to the map on the Search page.

1. **Enter your Mapbox API Key.** By default, Arches uses a few basemap services from Mapbox, for which you need to provide a key. You can get a free key at mapbox.com, and for installations that do not expect exceptionally heavy traffic, this free key will be sufficient. Once you have obtained the key, copy it from Mapbox (it will start with `pk.`). Go to System Settings → Map Settings → Mapbox API and enter it there.

   **Note:** If you don’t want to use MapBox, you can avoid this step by loading in a different basemap and removing all of the default MapBox layers. More about loading different basemaps in *Creating New Map Layers*.

2. **Set the default Map Zoom and Project Extent settings.** The Map Zoom is useful for geometry editing, but note that the Search page will automatically zoom to the extent of your search results every time they are updated. The Project Area is very important as it defines the area for your hexagon bins. It may be best to open a new tab with your Search page, make a change here in the Settings, and then refresh your Search page to preview the changes you make.

3. **Change Hexagon Bin Settings.** Finally, you can change the size and precision of the search result hexagon bins. We recommend changing these settings in small increments, as making a small bin size with a large project area (for example) can be costly for your browser and may cause it to crash when loading the Search page.

After getting the Map Settings figured out, you may want to change the name of your app which can be accomplished through the System Settings and changing the index.htm file in your arches_projects folder, or create some Saved Searches to make it easier for the users to explore your database (Saved Searches).
5.2 Arches Production Deployment

5.2.1 Changing the Admin Password

The first item of business when preparing your production of Arches is to change the Admin user’s password. You cannot change the Admin user’s password in the Arches UI because the Admin account is not associated with an email. Instead you’ll need to use the Django admin page:

1. Login as admin to Arches or in the Django admin (http://[your arches domain]/admin/)
2. Navigate to the Django admin user page http://[your arches domain]/admin/auth/user/.
3. In the upper right of the page select CHANGE PASSWORD and follow the steps to update the password.

5.2.2 Configuring Captcha

Setting up your captcha will help protect your production from spam and other unwanted bots. To set up your production with captcha, first register your captcha and then add the captcha keys to your project’s settings.py. Do this by adding the following:

```
RECAPTCHA_PUBLIC_KEY = 'x'
RECAPTCHA_PRIVATE_KEY = 'x'
```

Replace the x’s with your captcha keys.

5.2.3 User Sign-up

To enable users to sign up through the Arches UI, you will have to add the following lines of code to your project’s settings.py:

```
EMAIL_USE_TLS = True
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'xxxx@xxx.com'
EMAIL_HOST_PASSWORD = 'xxxxxxx'
EMAIL_PORT = 587
```
Update the EMAIL_HOST_USER and EMAIL_HOST_PASSWORD with the correct email credentials and save the file. It is possible that this may not be enough to support your production of Arches. In that case, there’s more information on setting up an email backend on the Django site.

To configure what group new users are put into, add the following lines of code to your project’s settings.py:

```python
# group to assign users who self sign up via the web ui
USER_SIGNUP_GROUP = 'Crowdsource Editor'
```

If you would like to change which group new users are added to, replace ‘Crowdsource Editor’ with the group you would like to use.

### 5.2.4 Configure Media Directory

The media directory identifies the destination directory of uploaded files. To do so, add the following lines of code to your project’s settings.py file:

```python
# Absolute filesystem path to the directory that will hold user-uploaded files.
MEDIA_ROOT = os.path.join(ROOT_DIR)

# URL that handles the media served from MEDIA_ROOT, used for managing stored files.
# It must end in a slash if set to a non-empty value.
MEDIA_URL = '/files/'
```

Replace the `ROOT_DIR` with the pathway to the folder where you want the directory to be created. For example:

```python
ROOT_DIR = os.path.join('projects','my_project')
```

### 5.2.5 Permissions Settings

Permissions allow you to tailor the user experience by letting you control which data a user or group can view and edit. Permissions are applied to each card and by default, the guest users (aka anonymous user) has read privileges to all data. If you have data you do not want to share with all users, simply perform the following steps in the Arches UI:

1. Navigate to the ‘Resource Models’ tab of the Arches Designer
2. Select a resource model and click on the Manage button. This will trigger a drop-down menu with a few different managers; select the Permissions Manager.
3. Be sure to select a user or group within the Group Library and then select a card from the Card Library.
4. Choose which permissions to grant the user.
   - No Access will deny the user from viewing or editing data in that card.
   - Read permissions allow the user/group to see the data presented in that card, but not edit it.
   - Create/Update allows the user/group to view and edit the data in that card, as well as create new data.
   - Delete permissions allow the user to delete data from the database.
5. Click on the Apply Permissions button.

### 5.2.6 Set DEBUG = False

Most importantly, you should never run Arches in production with DEBUG = True. Open your settings.py file (or settings_local.py) and set DEBUG = False (just add that line if necessary).
Turning off the Django debug mode will:

1. Suppress the verbose Django error messages in favor of a standard 404 or 500 error page.
   
   You will now find Django error messages printed in your `arches.log` file.

   **Important:** Make sure you have `500.htm` and `404.htm` files in your project's templates directory!

2. Cause Django to stop serving static files.
   
   You must set up a real webserver, like Apache or Nginx, to serve your app. To help with this, we have created a basic tutorial on how to *Serving Arches with Apache*.

### 5.3 Full Explanation of the System Settings UI

#### 5.3.1 System Settings

*Default Application Settings*

- **Application Name** - Name of your Arches app, to be displayed in the browser title bar and elsewhere.
- **Default Data Import/Export User** - Name to associate with data that is imported into the system.

*Web Analytics*

If you have made a Google Analytics Key to track your app’s traffic, enter it here.

*Thesaurus Service Providers*

Advanced users may create more SPAQRL endpoints and register them here. These endpoints will be available in the RDM and allow you to import thesaurus entries from external sources.

#### 5.3.2 Map Settings

*Mapbox API*

Arches uses the Mapbox mapping library for map display and data creation. Arches also supports Mapbox basemaps and other services.

- **Mapbox API Key (Optional)** - By default, Arches uses some basemap web services from Mapbox. You will need to create a free API key (or “access token”) for these services to be activated. Alternatively, you could remove all of the default basemaps and add your own, non-Mapbox layers.
- **Mapbox Sprites** - Path to Mapbox sprites (use default).
- **Mapbox Glyphs** - Path to Mapbox glyphs (use default).

*Project Extent*

Draw a polygon representing your project’s extent. These bounds will serve as the default for the cache seed bounds, search result grid bounds, and map bounds in search, cards, and reports.

*Map Zoom*

You can define the zoom behavior of your maps by specifying max/min and default values. Zoom level 0 shows the whole world (and is the minimum zoom level). Most map services support a maximum of 20 or so zoom levels.

*Search Results Grid*
Arches aggregates search results and displays them as hexagons. You will need to set default parameters for the hexagon size and precision.

**Warning:** A large project area combined with a small hexagon size and/or high precision will take a very long time to load, and can crash your browser. We suggest changing these settings in small increments to find the best combination for your project.

### 5.3.3 Basic Search Settings

Set the default search results behavior. This is also where you will define the max number of resources per export operation.

### 5.3.4 Temporal Search Settings

Arches creates a Time Wheel based on the resources in your database, to allow for quick temporal visualization and queries. A few aspects of this temporal search are defined here.

- **Color Ramp** - Currently unused (saved for future implementation). The color ramp for the time wheel. For further reference, check out the d3 API reference.

- **Time wheel configuration** - Currently unused (saved for future implementation). You can, however, modify the time wheel configuration using the advanced settings, *Time Wheel Configuration*.

### 5.3.5 Saved Searches

Arches allows you save a search and present it as convenience for your users. Saved Searches appear as search options in the main Search page. Creating a Saved Search is a three-step process.

1. **Specify Search Criteria** - Go to the Search page and enter all the criteria you would like to use to configure your Saved Search. You may notice that with the addition of each new search filter (either by using the term filter, map filtering tools, or temporal filters) the URL for the page will change.

2. **Copy the URL** - In your browser address bar, copy the entire URL. This will be a long string that defines each of the search filters created in step 1.

3. **Create the Saved Search** - Finally, head back to this page and fill out the settings that you see at left. You can also upload an image that will be shown along with your Search Search.

### 5.3.6 Maintaining UI-Defined Settings

Because these settings are stored in the database, as opposed to a settings.py file, if you drop and recreate your database, you will lose them and need to re-enter them by hand. To avoid this, you should run this command after you have finished configuring settings through the UI:

```
python manage.py packages -o save_system_settings [-d arches/db/system_settings]
```

A file named “System_Settings.json” will be saved to the directory indicated. If no directory is indicated the file will be saved to settings.SYSTEM_SETTINGS_LOCAL_PATH, which is my_project/my_project/system_settings/ by default. This same path is used to import settings when a new package is loaded into your project.
6.1 Projects

Arches Projects facilitate all of the customizations that you will need to make one installation of Arches different from the next. You can update html to modify web page branding, and add functions, datatypes, and widgets to introduce new functionality. A project sits outside of your virtual environment, and can thus be transferred to any other system where Arches is installed.

To create a project, see step 5 of the installation guide.

The general structure of a new Arches project is:

```
my_project/
│  manage.py
│  my_project/
│      settings.py
│      datatypes/
│      functions/
│      media/
│      templates/
│      widgets/
```

*Not all files are shown*

**Important:** At this level, “projects” are completely different from the mobile data collection “projects” that are mentioned elsewhere in this documentation.

6.1.1 settings.py

Many project-specific settings are defined here. You should use `settings_local.py` to store variables that you may want to keep out of the public eye (db passwords, API keys, etc.).
6.1.2 templates

This directory holds HTML templates that you can modify to customize the branding and general appearance of your project.

6.1.3 datatypes, functions, and widgets

These directories will store the custom datatypes, functions, and widgets that you can create for the project. Developers interested in pursuing these customizations should start with this customization documentation.

6.2 Packages

A package is an external collection of arches data (resource models, business data, concepts, collections) and customization files (widgets, datatypes, functions, system settings) that you can load into an arches project.

6.2.1 Loading a Package

To load a package simply run the load_package command using your *project’s manage.py file:

```
python manage.py packages -o load_package -s https://github.com/package/archive/branch.zip -db true
```

- **-db** *true* to run setup_db to rebuild your database. default = ‘false’
- **-ow** *overwrite* to overwrite concepts and collections. default = ‘ignore’
- **-st** *stage* to stage concepts and collections. default = ‘stage’
- **-s** a path to a zipfile located on github or locally
- **-o** operation name
- **-y** accept defaults (will overwrite existing branches and system settings with those in the package)

If you do not pass the `-db True` to the load_package command, your database will not be recreated. If you already have resource models and branches with the same id as those you are importing, you will be prompted to confirm whether you would like to keep or overwrite each model or branch.

**Note:** It is important to note that you cannot load a package directly into core Arches. Packages must be loaded into a project.

If you are a developer running the latest arches you probably want to create a project with a new Arches installation. This ensures that the `arches_project create` command uses the latest project templates.

1. Uninstall arches from your virtualenv

   `pip uninstall arches`

2. Navigate into arches root folder delete the `build` directory

3. Reinstall arches
python setup.py install
python setup.py develop

4. Navigate to where you want to create your new project and run:

```
arches-project create mynewproject
```

**Note:** You can use the option `[-d|--directory] <directory_name>` to change the directory your new project will be created in.

5. Finally run the `load_package` command using the project’s manage.py file.

```
python manage.py packages -o load_package -s https://github.com/package/archive/branch.zip -db true
```

### 6.2.2 Creating a New Package

If you want to create additional projects with the same data or share your data with others that need to create similar projects, you probably want to create a package.

The `create_package` command will help you get started by generating the folder structure of a new package and loading the resource models of your current project into your new package.

1. To create new package simply run the `create_package` command. The following example would create a package called `mypackge`.

```
python manage.py packages -o create_package -d /Full/path/to/mypackage
```

   `-d` full path to the package directory you would like to create

   `-o` operation name

2. Below is a list of directories created by the `create_package` command and a brief description of what belongs in each. Be sure not to place files that you do not want loaded into these directories. If, for example, you have draft business data that is not ready for loading, just add a new directory and stage your files there. Directories other than what is listed below will be ignored by the loader.

   - **business_data** Resource instance .csv and corresponding .mapping files, each sharing the same base name.
   - **business_data/files** Files to be added to the uploaded files directory
   - **business_data/relations** Resource relationship files (.relations)
   - **business_data/resource_views** sql views of flattened resource models
   - **extensions/function** Each function in this directory should have its own directory with a template (.htm), viewmodel (.js) and module (.py). Each file must share the same base name.
   - **extensions/datatypes** Each datatype in this directory should have its own directory with a template (.htm), viewmodel (.js) and module (.py). Each file must share the same base name.
   - **extensions/widgets** Each widget in this directory should have its own folder with a template (.htm), viewmodel (.js) and configuration file (.json). Each file must share the same base name.
   - **graphs/branches** arches.json files representing branches
   - **graphs/resource_models** arches.json files representing resource models
map_layers/mapbox_styles/overlays* Each overlay should have a directory with a mapbox style as exported from mapbox including a style.json file, license.txt file and an icons directory.

map_layers/mapbox_styles/basemaps* Each basemap should have a directory with a mapbox style as exported from mapbox including a style.json file, license.txt file and an icons directory.

map_layers/tile_server/overlays* Each overlay should have a directory with a .vrt file and .xml to style and configure the layer. Each file must share the same base name.

map_layers/tile_server/basemaps* Each overlay should have a directory with a .vrt file and .xml to style and configure the layer. Each file must share the same base name.

preliminary_sql sql files containing database operations necessary for your project.

reference_data/concepts SKOS concepts .xml files

reference_data/collections SKOS collection .xml files

system_settings The system settings file for your project

* map layer configuration By default mapbox-style layers will be loaded with the name property found in the layer’s style.json file. The default name for tile server layers will be the basename of the layer’s xml file. For both mapbox-style and tile server layers the default icon-class will be fa fa-globe. To customize the name and icon-class, simply add a meta.json file to the layer’s directory with the following object:

```json
{
    "name": "example name",
    "icon": "fa example-class"
}
```

3. It is not necessary to populate every directory with data. Only add those files that you would like to share.

Once you’ve added the necessary files to your package, simply compress it as a zip file or push it to a github repository and it’s ready to be loaded.

6.2.3 Configuring a Package

Two different files are used to define custom settings for your package.

• package_settings.py The django settings relevant to your project not managed in system settings. For example, you may want to include your time wheel configuration and your analysis SRID settings in this file so that users do not have add these settings manually to their own settings file after loading your package. This file is copied into your project when the package is loaded.

• package_config.json This file allows you to configure other parts of the data loading process. For example, the order in which the business data files are loaded. Contents of this file may look like

```json
{
    "permitted_resource_relationships": [],
    "business_data_load_order": [
        "a_LHD_Investigative_Activities_HM.csv",
        "LHD_Actors.csv",
        "LHD_Administration.csv",
        "LHD_Bibliographic_Sources.csv",
        "LHD_Heritage_Architectural_Landmark.csv",
        "LHD_Heritage_Asset_Areas_PC.csv",
        "LHD_Heritage_Asset_Artefacts_HM.csv",
        "LHD_Organizations.csv",
        "Lincoln_Heritage_Asset_Monument.csv"
    ]
}
```

(continues on next page)
6.2.4 Updating an Existing Package

If you make changes to the resource models in your project you may want to update your package with those changes. You can do that with the `update_package` command:

```
python manage.py packages -o update_package -d /Full/path/to/mypackage
```

- `-d` full path to the package directory you would like to update
- `-o` operation name
- `-y` accept defaults (will overwrite existing resource models with those from your project)

Bear in mind that this command will not update a package directly on Github. It will however update a package in a local directory that you have cloned from an existing package on Github or created yourself with the `create_package` command.
CHAPTER 7

Designing the Database

7.1 Arches Database Theory

Let’s begin with a brief primer on some of the core concepts upon which Arches is constructed.

**Resources** - Resources are what we call database records. If you are using Arches to create an inventory of historic buildings, each one of those buildings will be recorded as a “resource”. This terminology is used throughout the app.


**Branches** - Branches are tools for transport of complex node structures from one Resource Model to another. This allows you to avoid manually recreating the same “branches” in multiple Resource Models.

**Note:** Both Resource Models and Branches are sometimes referred to generically as “graphs”. This is because their underlying architecture is a graph. However, as you’ll see, they play completely different roles in Arches.

**Important:** The Arches Designer is used for altering the record-keeping structure of your database; it does not alter the physical Data Model.

7.2 Arches Designer

The Arches Designer is where you export, import, duplicate, modify, and create your Resource Models and Branches. Any user who is part of the Graph Editor group will have access to the Arches Designer.
If you don’t see any Resource Models listed in your Arches Designer, you may want to consider loading a package. Alternatively, you can directly import individual Resource Model files through Add ...

To edit a Resource Model, click on it or click Manage ... > Manage Graph and you’ll be brought to the Graph Designer.

7.3 Graph Designer

Almost all aspects of Resource Model and Branch design are handled in the Graph Designer. The exception is Functions, which are handled in the separate Function Manager.

The Graph Designer comprises three tabs, the Graph Tab, Cards Tab, and Permissions Tab. Each tab is used to configure a different aspect of the Resource Model: In the Graph Tab you design the node structure, in the Cards Tab you configure the user interface (card) for each nodegroup, and in the Permissions Tab you are able to assign detailed permission levels to each card. The general workflow for using the Graph Designer is to proceed through the tabs in that same order.

7.3.1 Graph Tab

The Graph Tab is where you build the actual graph, a structured set of nodes and nodegroups, which is the core of a Resource Model or Branch. As noted above, sometimes Resource Models and Branches are generically referred to as “graphs”, and this may seem confusing at first, but you’ll come to see that it is an appropriate nickname.

In practice, constructing the graph means adding nodes (or existing Branches) to the Graph Tree, which appears on the left side of the page when the Graph Tab is activated. When you add a new node, you set many different settings for that node, like datatype, in the main panel of the page.
During the graph construction process, you are able to create a new Branch from any portion of your graph. This is useful if you have completed a large section of the graph, and want to reuse it later in another Resource Model.

**Note:** If you are building a graph that uses an ontology, the ontology rules will automatically be enforced during this graph construction process.

Along the way, you can use the preview button to display the graph in a more graph-like manner. This view will be familiar to users of Arches going back to version 3.0.

### 7.3.2 Cards Tab

Once you have added nodes to the graph, you can switch to the Cards Tab to begin refining the user interface. As you can see, the graph tree is replaced with a “card tree”, which is very similar to what users will see when they begin creating a resource using this Resource Model.

The top of the card tree is the root of the Resource Model, and you’ll select it to configure the public-facing resource report. Below this, you’ll see a list of cards in the Resource Model, some of which may be nested within others. There will be a card in the card tree for every nodegroup in the graph tree. Finally, within each card you’ll see one or more widgets. These correspond to nodes in the graph that collect business data. In the image above, the Appellation widget is selected.

When you select a card or a widget, you will see the Card Manager or Widget Manager appear on the right-hand side of the page. This is where you will update settings like labels, placeholder text, tooltips, etc. The middle of the page shows a preview of how a data entry user will experience the card.

**Tip:** While working with the Cards Tab, you may need to go back and change a node in the Graph Tab. Be aware that though you may expect node changes in the Graph Tab to cascade to widget configurations in the Cards Tab, this does not always happen. Be sure to double-check your work!
Fig. 2: Screenshot of the Graph Tab in the Graph Designer, showing the graph in preview mode.

Fig. 3: Screenshot of the Cards Tab in the Graph Designer, showing an “Actor” Resource Model.
7.3.3 Permissions Tab

Arches allows you to define permissions at the card level, so in the Permissions Tab you’ll see the card tree, just as in the Cards tab. However, you will only be able to select entire cards, not individual nodes.

Fig. 4: Screenshot of the Permissions Tab in the Graph Designer, showing an “Actor” Resource Model.

Once you have selected one or more cards, you can select a user or user group and then assign one of the following permissions levels:

- **Delete**  Allows users to delete instances of this nodegroup. Note, this is not the same as being allowed to delete an entire resource, permissions for which are not handled here.

- **No Access**  Disallows users from seeing or editing instances of this nodegroup. Use this permission level to hide sensitive data from non-authenticated users (the public).

- **Read**  Allows users to see this nodegroup’s card. If disallowed, the card/nodegroup will be hidden from the map and resource reports.

- **Create/Update**  Allows users to create or edit instances of this nodegroup. This provides the ability to let users edit some information about a resource, while be restricted from editing other information.
Arches data is modeled with graphs. A graph is a collection of nodes, structured like branches, all emanating from the root node, which represents the resource itself. If you are modeling a building resource, you may have a root node called “Building” with a node attached to it called “Name”. You can imagine that complex and thoroughly documented resources will have many, many nodes.

An ontology is a set of rules that categorizes these nodes into classes, and dictates which classes can be connected to each other. It’s a “rulebook” for graph construction. Generally, this is known as a CRM (Conceptual Reference Model), and Arches comes preloaded with the CIDOC CRM v6.2, an ontology created by ICOM specifically to describe cultural heritage data. To learn more about the CIDOC CRM, visit cidoc-crm.org or view a full list of classes and properties.

When creating Resource Models and Branches, users have the option of enforcing an ontology throughout the graph, or creating a graph with no ontology. If an ontology is chosen, the Graph Designer will enforce all of the applicable node class (CRM Entities) and edge (CRM Properties) rules during use of the Graph Designer. Importantly, if a Resource Model uses an ontology one can only add Branches to it that have been made with the same ontology.
Creating Resources

9.1 Resource Manager

You may create new Resources only if you have access to the Resource Manager page. From there, you will begin by choosing which Resource Model you would like to use. Note that a Resource Model must have its status set to active for it to appear in the Resource Manager.

Fig. 1: Your Resource Manager page may look different than this image, depending on what Resource Models you have set up in your database.
9.2 Resource Editor

The Resource Editor is used to create new or edit existing Resources. On the left-hand side of the page you will see this Resource’s “card tree”, which shows all of the data entry cards that you can edit. Think of “creating data” as “adding cards”.

To begin, select a card, enter data, and click Add. Some cards may allow multiple instances, in which case you will be able to add as many of the same type as you want.

Fig. 2: Simple data entry in Arches.

Once you have saved data for a resource, you can see a full summary by selecting the top card. This is the resource report.

In some cases, cards will be nested within other cards, as in the example of adding a geo-location below.

Fig. 3: Created nested data in Arches.

9.2.1 Provisional Edits

If you are a member of the Resource Editor group, all of your edits—either creating new resources or editing existing ones—will be considered “Provisional”. A member of the Resource Reviewer group can then approve your edits, making them “Authoritative”.

1. Resource Editor makes an edit:

   ![Your changes have been submitted for review]

2. For Resource Reviewers, search results indicate provisional data:

   ![Results: 1](image)

   Resource Editors only see provisional data while using the resource editor.

3. Resource Reviewer will be prompted to Q/A the edit:

   ![Saved Description](image)

4. Accept or Decline:
5. Approved edits are immediately visible:

Tip: A Resource Reviewer can also use the “Q/A Type” search filter (see images above) to only find resources with (or without) provisional edits.

9.3 Related Resources

From the Resource Editor you can also access the Related Resources Editor, which is used to create a relationship between this resource and another in your database. To do so, open the editor, find the resource, and click Add. Your Resource Model will need to be configured to allow relations with the target Resource Model. If relations are not allowed, resources in the dropdown menu will not be selectable.

After a relation has been created, you can further refine its properties, such as what type of relation it is, how long it lasted, etc. While viewing the relation in grid mode, begin by selecting the relation in the table. You will see the “Delete Selected” button appear. Next click “relation properties”, enter the information, and don’t forget to “Save” when finished.

Fig. 4: Creating a relationship between two resource in Arches, and adding properties to that relationship.

Note: Creating a relationship between two resources using the related resource editor is fundamentally different from creating a resource instance node in graph. Creating a relationship is good for making a visual “web” of resource
relationships. Using a resource instance node in a Resource Model’s graph allows you to “embed” one resource inside of another.
10.1 Different Types of Layers

Arches allows a great deal of customization for the layers on the search map. The contents of the following section will be useful when using the Map Layer Manager to customize your layers.

10.1.1 Resource Layers

Resource Layers display the resource layers in your database. One Resource Layer is created for each node with a geospatial datatype (for example, geojson-feature-collection). You are able to customize the appearance and visibility of each Resource Layer in the following ways.

Styling

Define the way features will look on the map. The example map has demonstration features that give you a preview of the changes you make. You can choose to use Advanced Editing to create a more nuanced style. Note that changes made in Advanced Editing will not be reflected if you switch back to basic editing. For styling reference, checkout the MapBox Style Specification.

Clustering

Arches uses “clustering” to better display resources at low zoom levels (zoomed out). You are able to control the clustering settings for each resource layer individually.

- Cluster Distance - distance (in pixels) within which resources will be clustered
- Cluster Max Zoom - zoom level after which clustering will stop being used
- Cluster Min Points - minimum number of points needed to create a cluster

Caching

Caching tiles will improve the speed of map rendering by storing tiles locally as they are creating. This eliminates the need for new tile generation when viewing a portion of the map that has already been viewed. However, caching is not a simple matter, and it is disabled by default. Caching is only advisable if you know what you are doing.


10.1.2 Basemaps and Overlays

A Basemap will always be present in your map. Arches comes with a few default basemaps, but advanced users can configure and add more.

Overlays are the best way to incorporate map layers from external sources. On the search map, a user is able to activate as many overlays as desired simultaneously. Users can also change the transparency of overlays. New overlays can be added in the same manner as new basemaps.

Adding New Basemaps or Overlays

If you are a developer interested in creating new map layers (which could be new visualizations of resources or new basemaps and overlays), please see *Creating New Map Layers*.

10.1.3 Styling

Note that depending on the type of layer, there are different styling options. For styling reference, checkout the MapBox Style Specification.

10.1.4 Settings

- Layer name - Enter a name to identify this basemap.
- Default search map - For basemaps, you can designate one to be the default. For overlays, you can choose whether a layer appears on the in the search map by default. Note that in the search map itself you can change the order of overlays.
- Layer icon - Associate an icon with this layer
The Arches Reference Data Management (RDM) tool is a core Arches module which enables the creation and maintenance of controlled vocabularies for use in drop-downs and controlled fields within the various Arches Resource forms.

The use of the RDM is restricted to the Reference Data Manager, the person responsible for maintaining the controlled vocabularies. It allows for the creation, update, amendment and deletion of concept schemes (controlled vocabularies). In addition, the RDM enables you to export your schemes as SKOS-Compliant XML files as well as the import of external thesauri. For more information on SKOS see http://www.w3.org/2004/02/skos/.

11.1 Concept Schemes

A concept scheme can be viewed as an aggregation of one or more concepts and the semantic relationships (links) between those concepts.

Each controlled vocabulary within the Arches RDM, whether it is a simple wordlist or a polyhierarchical thesaurus, is defined as a concept scheme. [More detail about concept schemes needed here]

11.1.1 Getting started

In this section you will learn about:

- **Adding a new concept scheme**
  - Adding a label to a scheme
  - Adding a note to a scheme

- **Building the scheme**
  - Adding a Top Concept to a scheme
  - Importing a Top Concept from an external scheme
  - Adding a child concept
- Importing a child concept
- Adding an additional Parent Concept (polyhierarchy)
- Browsing the scheme using the graph interface
- Adding a Related Concept
- Adding an image to a concept
- Searching for a concept
- Deleting a concept

- Importing a scheme
- Exporting a scheme
- Deleting a scheme

Adding a new concept scheme

1. In the left hand panel select Add Scheme from the Tools dropdown. The Add Concept Scheme pop-up will appear.
2. Insert the title [Test Scheme] of the new concept scheme in the ConceptScheme Name field.
3. Add a brief description of the Concept Scheme in the **Scope Note** field.

4. Select the language of the ConceptScheme by clicking in the ‘**Language**’ field. This currently defaults to **en-(US) (English)**

5. Click the **Save Changes** button. The new concept scheme will appear in the **ConceptSchemes** panel.

**Adding a label to a scheme**

It is possible to add multiple labels to a scheme. This is useful as some schemes may have been referred to by different names previously.

1. Select the **Test Scheme** from the left hand **RDM** panel. The **Test Scheme** will appear in the right hand panel.

2. Select **Add Label**. The **Add Concept Label** pop-up will appear.

3. Click in the field marked ‘**prefLabel**’. The list of label types will appear.

4. Select the label type.

5. Select the language of the label by clicking in the **Language** field. This currently defaults to **en-(US) (English)**

6. Click the **Save** button. The new label will appear in the **Labels** panel.

**Adding a note to a scheme**

It is possible to add multiple notes to a scheme. This allows the reference data manager to add more information regarding the scheme including the scope of what it covers, it’s definition, changes to the history of the scheme, and how it should be used.

1. Select **Add Note**. The **Add Concept Note** pop-up will appear.

2. Enter the text for the new note in the ‘**Note Editor**’ field.

3. Click in the field marked ‘**scopeNote**’. The list of Note types will appear.

4. Select the relevant Note type.

5. **Note:** Only one note of each type is allowed.

6. Select the language of the Note by clicking in the ‘**Language**’ field. This currently defaults to **en-(US) (English)**

7. Click the **Save** button. The new Note will appear in the **Notes** panel.

**Building the Scheme**

Having created the new scheme you should now add the Top Concepts. These will form the framework for the vocabulary and act as the parents for more detailed concepts. This multi-level construction is known as the hierarchy. In a simple wordlist there will be only one level of concepts but in a complex thesaurus the hierarchy can be many levels deep.

**Adding a Top Concept to a scheme**

1. In the Right hand panel select **Add Top Concept** from the **Manage** dropdown. The **Add Concept** pop-up will appear.
2. Enter the text for the label in the ‘Label’ field.
3. Enter the definition of the concept in the ‘Note’ field. The list of Note types will appear.
4. Select the language of the Note by clicking in the ‘Language’ field. This currently defaults to en-(US) (English).
5. Select hasTopConcept from the ‘Relation from Parent’ field.
6. Click the Save Changes button. The new concept will appear in the Broader/Narrower Concepts panel.

It may be that other concept schemes similar to the one you are developing may already exist. If this is the case it is possible to import concepts along with their attributes from an external source. By default the RDM can import concepts from the Getty Art and Architecture Thesaurus.

**Importing a Top Concept from an external scheme**

1. In the Right hand panel select Import Top Concept from SPARQL from the Manage dropdown. The Import Concept pop-up will appear.
2. Select Getty AAT from the list of Schemes available.
3. In the ‘Search for a concept’ field type the text of a concept, eg. houses. A selection of concepts matching the text will appear.
4. Select the appropriate concept. The Concept Identifier field will be populated with the URI of the concept.
5. Click the Import button. The new concept will appear in the Broader/Narrower Concepts panel.
6. Click on the concept. The Concept details panel will appear and the Notes panel will be populated with the external concept’s scopeNote

**Adding a child concept**

1. Select the concept, which will act as the parent for the new child concept, by clicking on it. The Concept details panel will appear.
2. In the Right hand panel select Add Child from the Manage dropdown. The Add Concept pop-up will appear.
3. Enter the text for the label in the ‘Label’ field.
4. Enter the definition of the concept in the ‘Note’ field. The list of Note types will appear.
5. Select the language of the Note by clicking in the ‘Language’ field. This currently defaults to en-(US) (English).
6. Select narrower from the ‘Relation from Parent’ field.
7. Click the Save Changes button. The new concept will appear in the Broader/Narrower Concepts panel.
8. Click on the new concept. The Concept details panel will appear and the Notes panel will be populated with the concept’s scopeNote.

**Importing a child concept**

1. Select the concept, which will act as the parent for the new child concept, by clicking on it. The Concept details panel will appear.
2. In the Right hand panel select Import Child from SPARQL from the Manage dropdown. The Import Concept pop-up will appear.
3. Select Getty AAT from the list of Schemes available.
4. In the ‘Search for a concept’ field type the text of a concept, eg. castle. A selection of concepts matching the text will appear.

5. Select the appropriate concept. The Concept Identifier field will be populated with the URI of the concept.


7. Click on the concept. The Concept details panel will appear and the Notes panel will be populated with the external concept’s scopeNote

**Adding an additional Parent Concept (polyhierarchy)**

Some concepts may have more than one parent for example a castle is a type of fortification but it is also a domestic building. This situation where there are more than one possible parent concepts is called polyhierarchy.

1. Select the concept, which you want to add a parent concept to, by clicking on it.

2. Select Manage Parents from the Manage dropdown. The New Parent Concept pop-up will appear.

3. In the ‘Search for a concept’ field type the text of the parent concept you are going to add, eg. domestic buildings. A selection of concepts matching the text will appear.

4. Select the appropriate concept.

**Browsing the Scheme using the graph interface**

For any Concept the Broader/Narrower Concepts panel defaults to the tree view and shows a concept’s immediate broader (parent) and narrower (child) concepts. The scheme may also be browsed using the graph interface.

1. In the Broader/Narrower Concepts panel click Show graph. The graph view will appear centred on the concept you have chosen.

2. Navigate the graph by clicking on the ‘nodes’ (the circles). Clicking on a node will bring up a dialog box with the concept label and a ‘x’ symbol.

3. Click on the label to jump to the details for a concept

**Adding a Related Concept**

As part of a thesaurus it is possible to relate concepts which are not hierarchically related but may be of interest to a user. This ‘Associative’ relationship can be made by relating one concept to many others.

1. In the Right hand panel click on Add Related Concept in the Related Concepts panel. The Manage Related Concepts pop-up will appear.

2. Enter the text for the related concept in the ‘Select a concept’ field. A selection of concepts matching the text will appear.

3. Select the appropriate concept.

4. Click in the Relation type field. The Relation Type dropdown will appear.

5. Select ‘Related’.

6. Click the Save button. The related term is added to the concept.
Adding an image to a concept

Searching for a concept

Deleting a concept

Deleting a concept is simple in Arches but care should be taken that the concept has not been used in any recording forms. If a concept has been used a warning message will appear informing the Reference Data Manager that all instances of the concept in use must be replaced with an alternative concept before the concept can be deleted. If the Reference Data Manager is certain the concept has not been used then the concept may be deleted using either of the following methods.

Method 1: Tree view

1. Identify the concept’s parent concept and bring up its details.
2. In the *Broader/Narrower Concepts* panel make sure the tree view is visible (This is the default view).
3. Click on the ‘x’ symbol next to the concept to be deleted. The *Delete a Concept* pop-up will appear.

   **Note:** A warning message ‘By deleting this concept, you will also be deleting the following concepts as well. This operation cannot be undone.’ will appear. If you do not want to delete the concept click the *No* button.

4. Click the *Yes* button. The concept is deleted (along with any of its children).

Method 2: Graph view

1. Identify the concept’s parent concept and bring up its details.
2. In the *Broader/Narrower Concepts* panel make sure the graph view is visible by clicking on *Show graph*.
3. Click on the node for the concept to be deleted. A dialog box with the concept label and a ‘x’ symbol will appear.
4. Click on the ‘x’ symbol next to the concept to be deleted. The *Delete a Concept* pop-up will appear.

   **Note:** A warning message ‘By deleting this concept, you will also be deleting the following concepts as well. This operation cannot be undone.’ will appear. If you do not want to delete the concept click the *No* button.

5. Click the *Yes* button. The concept is deleted (along with any of its children) and the node will disappear.

Importing a scheme

1. In the left hand panel select *Import Scheme* from the *Tools* dropdown. The *Import New Concept Scheme* pop-up will appear.
2. Click the *Choose File* button. The *Windows Explorer* panel will appear.
3. Navigate to the file to be uploaded.
Note: This file should be a SKOS file in any format parseable by Python’s RDFLib. Examples include RDF/XML, N3, NTriples, N-Quads, Turtle, TriX, RDFa and Microdata.

4. Click Open. You will be returned to the Import New Concept Scheme pop-up and the name of the file will have populated the form.

5. Click Upload File. The number of Concept Schemes will have increased by 1 and the imported concept scheme will appear in the ConceptSchemes panel.

Exporting a scheme

1. In the left hand panel select Export Scheme from the Tools dropdown. The Export Scheme pop-up will appear.
2. Click in the Select Scheme to Export field. The Concept Scheme dropdown menu will appear.
3. Select the scheme to be exported and click on it. The scheme name will populate the field.
4. Click the Export button. A new browser tab will appear containing a SKOS-compliant XML export of the scheme.
5. Right click on the file and select Save as... from the pop-up menu. The Save as panel will appear.
6. Choose where you want to save the file by navigating through the folder structure.
7. Give the file a relevant name and click the Save button. The file will be saved to the selected location.
Deleting a scheme

1. In the left hand panel select **Delete Scheme** from the **Tools** dropdown. The **Delete Scheme** pop-up will appear.

   **Note:** A warning message stating ‘You won’t be able to undo this operation! Are you sure you want to permanently delete this entire scheme from Arches?’ will appear. If you do not want to delete the scheme click the **Close** button.

2. Click in the **Select Scheme to Delete** field. The **Concept Scheme** dropdown menu will appear.

3. Select the scheme to be deleted and click on it. The scheme name will populate the field.

4. Click the **Delete** button.

   **Note:** The warning message will appear again along with a list of all of the concepts to be deleted. If you do not want to delete the scheme click the **Close** button.

5. Click the **Delete** button to confirm deletion. The scheme is deleted along with all its concepts.
12.1 What Are Arches Collector Projects?

Using the Arches Collector mobile app to crowdsource data is dependent on the creation of Projects, by a database administrator, in the Arches Collector Manager. Each Project is a discrete collection effort comprising the following:

- Specific users or groups of users who are allowed to participate
- Start and end dates to define when users are allowed to participate
- A selection of which Resource Models will be used, and more precisely, which cards from within those Resource Models will be used
- A set of existing resources from your database, to which users may add more data or just use as reference (based on card-level permissions)
- An optional MBTiles file to provide a basemap for users working outside of wifi/cell network range

To facilitate a wide array of simultaneous data collection efforts, an administrator can create multiple projects, all targeting different users and a different type of data collection.

See also:
To learn more about how data collection users will participate in Projects, please see Arches Collector Workflow.

12.2 Arches Collector Manager Overview

Important: To access the Arches Collector Manager, you must be in the Mobile Project Administrator group or have superuser status.

To get to the Arches Collector Manager, use the icon in the left-hand nav bar, or go directly to /collector_manager. Before you have created any projects, the interface will look like this:
As you begin to add more and more projects, the interface will resemble the Arches Designer, with Projects taking the place of Resource Models:

12.3 Creating Projects in the Arches Collector Manager

Use the + New Project button to begin creating a new project, and you’ll be brought to the summary page:
Now, using the navigation tree in the left-hand panel, you will set all of your project parameters. Detailed information about each parameter is available in the in-app help:

When you have filled out all the parameters, you will see in the project summary that all Project Requirements have been met. You are now ready to activate the Project.

**Warning:** Once a Project has been activated for the first time, its parameters are saved to an intermediate database, and they cannot be changed. Therefore, it’s wise to double-check everything before the initial activation.

Great! Now that you have activated your project, any data collection user that you assigned to it will be able to download it through the Arches Collector app and begin collecting data.

**Important:** If Arches Collector users are unable to connect to your Arches instance, you may need to register an Arches application.

After you have activated the project and data collectors have used it to add resources to your database, the project summary page will tell your how long the project has been active, how many resources have been collected, how many have connected to it, and more.
12.4 Creating an Offline Basemap

Projects allow for the inclusion of an MBTiles file to serve as a basemap while data collectors are outside of wifi/cell network range. This file will be downloaded and stored on an Arches Collector users’ device the first time that “offline maps” are activated in a project, not upon the initial download. Therefore, users should activate this map in the project before going into the field.

You can also obtain high quality and inexpensive tiles from sites such as https://openmaptiles.com. To create an MBTiles file with your own geospatial data, you can use a free account at Mapbox.com, or desktop software like TileMill.

Note: Large files will require longer downloads and more storage space on mobile devices.
Introduction to Arches Collector

Download links for the app are coming soon!

Arches Collector is a mobile app that allows you and your team to collect heritage data in the field, and sync it back to your main Arches instance. Database administrators facilitate this process by designing “projects” in the Arches instance, which define who can participate, what kind of data will be collected, where the project will take place, and for how long. Approved users can then connect to the Arches instance, download the project, and collect data, and sync it back to the database.

13.1 General App Navigation

Arches Collector is set up in a three-tiered hierarchical structure. Menu options throughout the app are context-sensitive based on the level within which you are working.

> **Arches Instance Level** - You will connect to one or more Arches Instances.

>> **Project Level** - Once connected to an Arches instance, you’ll download and enter a project from that instance.

>>> **Resource Level** - Within a project, you’ll be working with individual resources to create and edit data.
14.1 Connect to an Arches Instance

Begin by connecting to an existing Arches instance. This is a deployment of Arches that you can also view on the web. You’ll use credentials provided to you by the database administrator.
Fig. 2: Be sure to include http:// or https:// in the URL for the Arches instance.
You can create and maintain connections to as many Arches instances as you want.

![Arches Instances](image)

Fig. 3: This user has only connected to one instance, but more connections will be shown here once they are made.

### 14.2 Download a Project

Upon entering an Arches instance, you will see a list of projects. These are all of the mobile data collection projects to which you have been granted access. Choose a one, and download it to begin working.

Once you have downloaded a project, you can enter it and begin collecting data.

#### Inactive Projects

A project marked as “Inactive” means that no new users can download it. If you have already downloaded a project that is now marked inactive, you can continue to work with it as before.

#### 14.2.1 Download the Offline Map

In some cases, you may expect to collect data outside of wifi/cellular network range, so the database administrator will configure this project with an offline map. **This map will be downloaded the first time you activate the “Use offline maps” switch.**

**Important:** Make sure you download the offline map *before* heading out into the field for the first time! We recom-
Fig. 4: This user has access to three different projects in this instance, but has only downloaded the one at the top of the list.
mend doing this while connected to wifi, as the offline map file could be large.

From the list of projects, open the settings menu for a project and flip the “Use offline maps” switch.

14.2.2 Managing Projects

You can keep your project screen clean by leaving projects if you will no longer be participating (you can rejoin later), or by deleting projects from your device. All unsynced data in a project will be lost upon deletion.

14.3 Navigating a Project

There are four main screens within a project, and you can switch between them by swiping left and right, or using the nav bar along the bottom of the screen.

**NEW** - Choose a resource model to create a new resource.

From here you will create new resources.

**REVIEW** - View all of the resources in your project. This includes:

- New resources you have created
- Existing Resources that were downloaded as part of the project
- New resources created and synced by other members of this project (you must sync your own project to get these resources)
Fig. 5: This user is viewing a project called “CDB Historic Survey - Part 1”. The project was configured with only one resource model, named “Heritage Asset - Monument”.

Arches Documentation, Release 4.4.2
Fig. 6: This user has quite a few resources in the review screen. The top one has been edited (in this case, created) locally, so it has a white background. Those with grey backgrounds have not been edited locally, and were likely part of the initial project download.
From here you can select a resource to edit or view its report.

**MAP** - A map view showing all of the resources in your project that have location data. The basemap is defined in the project configuration. See *Download the Offline Map* if you expect to be collecting data beyond wifi/cellular network range.

![Map view](image)

**Fig. 7:** Selected, a resource’s name and description (if present) will be shown in a pop-up. These attributes are defined by the configuration of the resource model’s “Define Resource Descriptors” function.

From here you can select a resource to edit.

**SUMMARY** - A project-level summary.

### 14.4 Creating Records

New resources are made from the **NEW** screen, where you will choose a resource model to use for the new resource. In the following example, a new “Heritage Asset - Monument” resource will be created.

Beginning to create a new resource, you’ll see all of the cards (or “records”) that can be added to the resource.

While the example above shows text and dropdown entries, other types of data will have different input methods.

### 14.5 Editing Records

From the **REVIEW** screen of a project, you can select resources to edit them. In this example “The Eliza Jane” will be selected.
Description
An effort to inventory historic monuments around the city of New Orleans

2019/03/27 16:28
Date/Time of Last Sync

6 Records download  1 Records Edited

14.5. Editing Records
Fig. 8: In the images above, the “Monument Asset Name” card is selected in order to create a new name for the resource.
Fig. 9: An Asset Name and Name Type are entered. Because all changes are automatically saved, the display name is updated as soon as this name is entered. Note that this happens because the Define Resource Descriptors function has been properly configured in the Resource Model by the database administrator.
Fig. 10: Use the in-app “back” button to return to the resource overview. There is now one Monument Asset Name record.
Fig. 11: Creating location records happens through a map interface. Select the shapetype, and tap the map to create geometries. With lines and polygons, double-tap to finish the geometry.
Fig. 12: Note that this user is shown the option to delete The Eliza Jane. Once the project has been synced, deletion will no longer be an option. You will also never be allowed to delete resources that were initially downloaded with the project.
When you’ve selected a resource you’ll be brought to its RESOURCE REPORT: a summary of all the records that have been added to it. As shown below, you can edit any existing record (as long as you have the proper permissions).

Fig. 13: Use the Edit icon to edit or add a record. You can also use the EDITOR tab to switch to the full resource record view.

14.6 Syncing Data

When you have finished collecting data, either creating new resources or editing existing ones, you will sync your local edits to the web Arches deployment. In fact, you can do this at any time during your data collection, and as often as you want.

When you sync your the project on your mobile device, this is what happens:

- All local changes will be pushed to the Arches instance.
- All local will be made available to all other participants in the same project.
- You will download any data changes that other project participants have synced.
- You will **not** download changes that have been made to resources directly in the Arches instance since the project was first activated.

Additionally, some changes occur to the resources you have created in your project:

- You will no longer be able to delete entire resources that you created locally.
- You will no longer be able to delete resource records that you created locally.
Fig. 14: While working in a project, use the icon in the upper right to open the menu from which you can sync, using the “Refresh survey data” button.

You can sync a project from within the project itself:

Or you can sync from the main project list page:
Fig. 15: Use “Refresh all records in this project” from the project settings menu.
Additional Configuration

15.1 Password Validators

By default, Arches requires that passwords meet the following criteria:

- Have at least one numeric and one alphabetic character
- Contain at least one special character
- Have a minimum length of 9 characters
- Have at least one upper and one lower case character

Admins can change these requirements by configuring the `AUTH_PASSWORD_VALIDATORS` setting in their projects' `settings_local.py` file. Below is the default validator setting:

```python
AUTH_PASSWORD_VALIDATORS = [
    
    
    {'NAME': 'arches.app.utils.password_validation.NumericPasswordValidator',
    #Passwords cannot be entirely numeric,
    },
    
    {'NAME': 'arches.app.utils.password_validation.SpecialCharacterValidator',
    #Passwords must contain special characters
    'OPTIONS': {
        'special_characters': (',', '@', '#', '!', '(','*','&','^','%','$'),
    },
    },
    
    {'NAME': 'arches.app.utils.password_validation.HasNumericCharacterValidator',
    #Passwords must contain 1 or more numbers
    },
    
    {'NAME': 'arches.app.utils.password_validation.HasUpperAndLowerCaseValidator',
    #Passwords must contain upper and lower characters
    }
    ]
```
To remove a password validator in Arches, you can simply remove a validator from the list of AUTH_PASSWORD_VALIDATORS.

To modify the list of required special characters, simply edit the list of characters in the special_characters option in the SpecialCharacterValidator validator.

To change the minimum length of a password, change the min_length property in the MinLengthValidator validator.

Advanced users can override or add new validators by creating their own validation classes as explained in Django’s password validation documentation.

## 15.2 Time Wheel Configuration

By default Arches will bin your data in the search page time wheel based on your data’s temporal distribution. This enables Arches to bin your data efficiently. If your data spans over 1000 years, the bins will be by millennium, half-millennium and century. If your data spans less than a thousand years, your data will be binned by millennium, century, and decade.

You may decide, however, that the bins do not reflect your data very well, and in that case you can manually define your time wheel configuration by editing the TIMEWHEEL_DATE_TIERS setting.

Here is an example of a custom time wheel:

```
TIMEWHEEL_DATE_TIERS = {
    "name": "Millennium",
    "interval": 1000,
    "root": True,
    "child": {
        "name": "Century",
        "interval": 100,
        "range": {"min": 1500, "max": 2000},
        "child": {
            "name": "Decade",
            "interval": 10,
            "range": {"min": 1750, "max": 2000}
        }
    }
}
```

Each tier, (‘Millennium’, ‘Century’, ‘Decade’ are each tiers) will be reflected as ring in the time wheel. Properties:

- “name” - The name that will appear in the description of the selected period
- “interval” - The number of years in each bin. For example, if your data spans 3000 years, and your interval is 1000, you will get three bins in that tier.
- “root” - This applies only to the root of the config and should not be modified.
• “child” - Adding a child will add an additional tier to your time wheel. You can nest as deeply as you like, but the higher the resolution of your time wheel, the longer it will take to generate the wheel.

• “range” - A range is optional, but including one will restrict the bins to only those within the range.

If you do need to represent decades or years in your time wheel and this impacts performance, you can cache the time wheel for users that may load the search page frequently. To do so, you just need to activate caching for your project. If you have Memcached running at the following location 127.0.0.1:11211 then the time wheel will automatically be cached for the ‘anonymous’ user. If not you can update the CACHES setting of your project:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': os.path.join(APP_ROOT, 'tmp', 'djangocache'),
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

This will cache the time wheel to your project’s directory. There are other ways to define your cache that you may want to use. You can read more about those options in Django’s cache documentation.

By default the time wheel will only be cached for ‘anonymous’ user for 24 hours. To add other users or to change the cache duration, you will need to modify this setting:

```
'CACHE_BY_USER = {'anonymous': 3600 * 24}
```

The CACHE_BY_USER keys are user names and their corresponding value is the duration (in seconds) of the cache for that user. For example, if I wanted to cache the time wheel for the admin user for 5 minutes, I would change the CACHE_BY_USER setting to:

```
'CACHE_BY_USER = {'anonymous': 3600 * 24, 'admin': 300}
```
16.1 Using Arches to Install Elasticsearch

The easiest way to install Elasticsearch is to use a command that comes with Arches. Once you have installed Arches (either with pip or from the Arches repo), activate your virtual environment, enter your app/project root directory (the one that contains manage.py), run

```
python manage.py es install
```

Elasticsearch will be installed in the root folder. You can specify an alternate destination for the installation by using the -d argument. For example

```
python manage.py es install -d C:\Projects
```

will result in a new directory C:\Projects\elasticsearch-5.2.1 from which you’ll be able to run Elasticsearch.

16.2 Running Elasticsearch

To start Elasticsearch from a command line, run

**Linux/macOS**

```
path/to/elasticsearch-5.2.1/bin/elasticsearch
```

**Note:** To run the process in the background, add -d.

**Windows**

```
path\to\elasticsearch-5.2.1\bin\elasticsearch
```
Note: To run the process in a new terminal you can double-click the `elasticsearch.bat` file found in `elasticsearch-5.2.1\bin`. To properly set up Elasticsearch as a background service on Windows, check out this documentation.

To make sure Elasticsearch is running correctly, use

```
curl localhost:9200
```

on any operating system. You should get a JSON response that includes “You Know, For Search…”

For more information about running the service (and all things Elasticsearch), please visit the official Elasticsearch documentation.

Important:
1. By default, Elasticsearch 5.2.1 uses 2GB of memory (RAM). For basic development purposes, we have found it to run well enough on 1GB. Use `ES_JAVA_OPTS="-Xms1g -Xmx1g" ./bin/elasticsearch -d` to set the memory allotment on startup (read more). You can use the same command to give more memory to Elasticsearch in a production setting.
2. Some users have reported getting this error right after starting Elasticsearch: `org.elasticsearch.bootstrap.StartupException: java.lang.IllegalStateException: No match found`. Upgrading to 5.2.2 seems to fix this problem, as described in this issue.

16.3 Reindexing The Database

At some point, you may need to reindex the entire database. This can be helpful if a bulk load failed halfway through, or if you need to point your database at a different Elasticsearch installation. In the command line run:

```
python manage.py es index_database
```

Be warned that this process can take a long time if you have a lot of resources in your database. Also, if you are in DEBUG mode it can cause your server to run out of memory (see reindex the database).

If the `es index_database` operation doesn’t solve your issue, you can try this series of commands:

```
python manage.py es delete_indexes
python manage.py es setup_indexes
python manage.py es index_database
```

16.4 Using An Independent Elasticsearch Installation

You may want to use an existing installation of Elasticsearch, integrate with something like Amazon Elasticsearch Service, or perhaps create a new instance using the normal Elasticsearch installation process. In any case, you will need to locate the config file `elasticsearch.yml` and add the following two lines to the end of it:

```
script.inline: true
script.indexed: true
```
You may need to update `settings.py` or `settings_local.py` in your project so Arches can find Elasticsearch in this new location. Add or modify the following lines as necessary:

```python
ELASTICSEARCH_HTTP_PORT = XXXX  # new port number
ELASTICSEARCH_HOSTS = [{'host': 'localhost', 'port': ELASTICSEARCH_HTTP_PORT}]
```

If you have Elasticsearch running on a different host, update the 'host' directive accordingly and make sure the external host will allow connections to your IP through whatever port is in use.

### 16.5 Elasticsearch Index Prefix

In `settings.py` you’ll find the variable `ELASTICSEARCH_PREFIX`. By default it is the name of your project, and this prefix is prepended to all of the Elasticsearch indices that your project creates and uses (e.g. `myproject_resource`, `myproject_resource_relations` and `myproject_strings`). This allows multiple projects to use the same Elasticsearch service without sharing any data.

If you want multiple Arches projects to share data, you could set the prefix to be the same in each project and point them to the same Elasticsearch service. This could get very messy...

### 16.6 Using Multiple Nodes

If you are watching the console output after you start up Elasticsearch, you may notice `Cluster health status changed from [RED] to [YELLOW]`. “Yellow” status is an indicator that you only have one node running in the cluster, and Elasticsearch will not be able to make replicas. To fix this (go from “Yellow” to “Green”), you’ll need to install another Elasticsearch instance and configure its node as part of your existing cluster. Here’s some background and a stack overflow question with straightforward instructions for adding a node.

If you can use AWS and Ubuntu Xenial, then you can to begin with an AMI we have available. The AMI is in the US West (Oregon) region, its id and name are `ami-56e44f2e` and `es-5.2.1-two-nodes`, respectively. If you are going to use both nodes (which are configured to work together in the same cluster), use at least a t2.large instance and run `source /home/ubuntu/esstart.sh`. To use a single instance, you’ll need at least a t2.medium instance and run `/usr/share/elasticsearch/node1/bin/elasticsearch -d`. Here’s more info specifically about Elasticsearch on AWS.
17.1 General Notes

Arches allows any parameters to be passed in via custom HTTP headers OR via the querystring. All requests to secure services require users to pass a “Bearer” token in the authentication header.

To use a HTTP header to pass in a parameter use the form:

```
HTTP-X-ARCHES-{upper case parameter name}.
```

So, for example, these are equivalent requests

```

curl http://localhost:8000/mobileprojects?format=json-ld
```

If both a custom header and querystring with the same name are provided, then the querystring parameter takes precedence.

In the following example “html” will be used as the value for the “format” parameter.

```
˓→format=html
```

**Note:** Querystring parameters are case sensitive. Behind the scenes, custom header parameters are converted to lower case querystring parameters.

In the following example there are 3 different parameters (“format”, “FORMAT”, and “Format”) with 3 different values (“html”, “json”, and “xml”) respectively.

```
http://localhost:8000/mobileprojects?format=html&FORMAT=json&Format=xml
```
17.2 Authentication

Most Arches API endpoints require an OAuth access token.

OAuth 2.0 is a simple and secure authentication mechanism. It allows applications to acquire an access token for Arches via a quick redirect to the Arches site. Once an application has an access token, it can access a user’s resources on Arches. Authentication with OAuth can be accomplished in the following steps:

17.2.1 Registering an application with Arches

To allow others to connect to your Arches instance, including Arches Collector users, you must create an OAuth client id and add it to your settings.

1. In a browser go to

   http://<yourdomain:port>/o/applications/

2. Create a new application

3. Fill out the form with a Name of your choosing, and set Client type and Authorization grant type as shown in the image below.
4. Copy the Client id and submit the form (you can access this id at any time).

5. In your Arches project’s settings.py or settings_local.py file, set or add this variable

   ```python
   MOBILE_OAUTH_CLIENT_ID = "<your new Client id>"
   ```

6. That’s all! Arches Collector users will now be able to connect to your Arches instance using their own credentials.

**Important:**

- Only make one application, though you are technically allowed to make more.
- An application is “owned” by whichever user created it, and will not be visible to other users.
POST /o/token
gets an OAuth token given a username, password, and client id

Note: You should only make this call once and store the returned token securely. You should not make this call per request or at any other high-frequency interval. This token is to be used with clients registered with the “Resource Owner Password Credentials Grant” type see Registering an application with Arches for more information on registering an application

For additional information see https://tools.ietf.org/html/rfc6749#section-4.3

Form Parameters

• **username** – a users username (or email)
• **password** – a users password
• **grant_type** – “password”
• **client_id** – the registered applications client id, see Registering an application with Arches

Status Codes

• 401 Unauthorized – there’s no user or the user has been deactivated, or the client id is invalid

Example request:

curl -X POST http://localhost:8000/o/token/ -d "username=admin&password=admin&
  →grant_type=password&client_id=onFiQSbPfg2psUcl2fBvaaEHA58MKHavl3iuSaRf"

Example response:

HTTP/1.1 200 OK
Content-Type: application/json

{
  "access_token": "TS3pE2bEXRCakRls4IGKCVVa02v6FE",
  "token_type": "Bearer",
  "expires_in": 36000,
  "refresh_token": "y3rzXKf8dXdb25ayMMVIIigTkqEKr0",
  "scope": "read write"
}

returned when an invalid username or password is supplied

HTTP/1.1 401 Unauthorized
Content-Type: application/json

{"error_description": "Invalid credentials given.", "error": "invalid_grant"}

returned when an invalid client id is supplied, or the registered client is not “public” or the grant type used to register the client isn’t “Resource Owner Password Credentials Grant”

HTTP/1.1 401 Unauthorized
Content-Type: application/json

{"error": "invalid_client"}
17.3 Concepts

GET /rdm/concepts/{uuid:concept instance id}
gets a single rdm concept instance

Query Parameters

- **format** – {"json"}
- **indent** – number of spaces to indent json output
- **includesubconcepts** – option to include sub concepts in the return
- **includeparentconcepts** – option to include parent concepts in the return
- **includerelatedconcepts** – option to include related concepts in the return
- **depthlimit** – limit the number of subconcept layers to return if includesubconcepts is true
- **lang** – show subconcep results with specified language first

Request Headers

- **Authorization** – oAuth token for user authentication, see /o/token

Example request:

```bash
curl -H "Authorization: Bearer zo41Q1IMgAW30xOroiCUxjv3yci8Os" -X GET http://localhost:8000/rdm/concepts/5e04c83e-1ae3-42e8-ae31-4f7c25f737a5?format=json&indent=4
```

Example json response:

```
HTTP/1.0 200 OK
Content-Type: application/json

{
    "hassubconcepts": true,
    "id": "5e04c83e-1ae3-42e8-ae31-4f7c25f737a5",
    "legacyoid": "http://www.archesproject.org/5e04c83e-1ae3-42e8-ae31-4f7c25f737a5",
    "nodetype": "Concept",
    "parentconcepts": [],
    "hassubconcepts": true,
    "id": "7b8e4771-2680-4004-9743-40ea78e8c2a9",
    "legacyoid": "http://www.archesproject.org/7b8e4771-2680-4004-9743-40ea78e8c2a9",
    "nodetype": "ConceptScheme",
    "parentconcepts": [],
    "relatedconcepts": [],
    "relationshiptype": "hasTopConcept",
    "subconcepts": [],
    "values": [
        "category": "label",
        "conceptid": "7b8e4771-2680-4004-9743-40ea78e8c2a9",
        "id": "b18048a9-4814-43f0-bb88-99fa22a42fbe",
        "language": "en-US",
    ]
}
```
"type": "prefLabel",
"value": "DISCO"
],

"category": "note",
"conceptid": "7b8e4771-2680-4004-9743-40ea78e8c2a9",
"id": "16ea8772-d5dd-481d-91a7-c09703718138",
"language": "en-US",
"type": "scopeNote",
"value": "Concept scheme for managing Data Integration for Conservation Science thesauri",

"category": "identifiers",
"conceptid": "7b8e4771-2680-4004-9743-40ea78e8c2a9",
"id": "9eaa8a10-e9f2-4ce3-ac8b-c4904097b4c9",
"language": "en-US",
"type": "identifier",
"value": "http://www.archesproject.org/7b8e4771-2680-4004-9743-40ea78e8c2a9"
]
])

"relatedconcepts": [],
"relationships": "",
"subconcepts": [{
  "hassubconcepts": false,
  "id": "0788acb1-9968-43e8-80f7-37b37e155f95",
  "legacyoid": "http://www.archesproject.org/0788acb1-9968-43e8-80f7-37b37e155f95",
  "nodetype": "Concept",
  "parentconcepts": [],
  "hassubconcepts": false,
  "id": "5e04c83e-1ae3-42e8-ae31-4f7c25f737a5",
  "legacyoid": "http://www.archesproject.org/5e04c83e-1ae3-42e8-ae31-4f7c25f737a5",
  "nodetype": "Concept",
  "parentconcepts": [],
  "relatedconcepts": [],
  "relationships": "narrower",
  "subconcepts": [],
  "values": []
}]
"relatedconcepts": [],
"relationships": "narrower",
"subconcepts": [],
"values": [{
  "category": "label",
  "conceptid": "0788acb1-9968-43e8-80f7-37b37e155f95",
  "id": "dd5c6d39-7bc4-438e-abe2-544b8ae06864",
  "language": "en-US",
  "type": "prefLabel",
  "value": "Artist"
}, {
  "category": "identifiers",
  "conceptid": "0788acb1-9968-43e8-80f7-37b37e155f95",
  "id": "5f355975-29a7-4a53-8260-4093d63c1967",
  "language": "en-US",
  "type": "identifier",
  "value": "http://www.archesproject.org/0788acb1-9968-43e8-80f7-37b37e155f95"
}]
(continues on next page)
17.4 Resources

GET /resources/
gets a paged list of resource instance ids in json-ld format

Query Parameters

• page – number specifying the page of results to return

Example request:

curl -X GET http://localhost:8000/resources/
curl -X GET http://localhost:8000/resources/?page=2

Example response:

HTTP/1.0 200 OK
Content-Type: application/json

{  
  "@context": "https://www.w3.org/ns/ldp/",  
  "@id": "",  
  "@type": "ldp:BasicContainer",  
  "ldp:contains": [    
    "http://localhost:8000/resources/00000000-0000-0000-0000-000000000100",    
    "http://localhost:8000/resources/00000000-0000-0000-0000-000000000101",    
    "http://localhost:8000/resources/000ee2fe-4568-457b-960c-3e1ec3f53e10",    
    "http://localhost:8000/resources/000fa53f-0f06-4648-a960-c42b8accd235",    
    "http://localhost:8000/resources/001b6c4b-f906-4df2-9fcd-b9fda95eed95",    
    "http://localhost:8000/resources/0032990e-f8d6-4a7b-8032-d90d3c764b40",    
    "http://localhost:8000/resources/003619ca-5fa7-4e75-b3b7-a62f40fe9419",    
    "http://localhost:8000/resources/00366caa-3c00-4909-851d-0d650e62f820",    
    "http://localhost:8000/resources/003874d7-8e73-4323-bddf-b893651e22c1"  
  ]
}
GET /resources/{uuid:resource instance id}
gets a single resource instance

Query Parameters

- **format** – {“xml”, “json”, “json-ld”}
- **indent** – number of spaces to indent json output

Request Headers

- **Authorization** – OAuth token for user authentication, see /o/token
- **Accept** – optional alternative to “format”, {“application/xml”, “application/json”, “application/ld+json”}

Example request:

```bash
```

Example json response:

```json
HTTP/1.0 200 OK
Content-Type: application/json

{
   "business_data": {
      "resources": [
         {
            "tiles": [
               {
                  "data": {
                     "e4b37f8a-343a-11e8-ab89-dca90488358a": "203 Boultham Park Road",
                     "e4b4b7f5-343a-11e8-a681-dca90488358a": "null,
                     "provisionaledits": "null,
                     "parenttile_id": "null,
                     "nodegroup_id": "e4b37f8a-343a-11e8-ab89-dca90488358a",
                     "sortorder": 0,
                     "resourceinstance_id": "99131129-7451-435d-aab9-33eb9031e6d1",
                     "tileid": "b72225a9-4e3d-47ee-8d94-52316469bc3f"
                  }
               },
               {
                  "data": {
                     "e4b37f8a-343a-11e8-ab89-dca90488358a": "null,
                     "provisionaledits": "null,
                     "parenttile_id": "null,
                     "nodegroup_id": "e4b37f8a-343a-11e8-ab89-dca90488358a",
                     "sortorder": 0,
                     "resourceinstance_id": "99131129-7451-435d-aab9-33eb9031e6d1",
                     "tileid": "b72225a9-4e3d-47ee-8d94-52316469bc3f"
                  }
               }
            ]
         }
      }
   }
}
```
PUT /resources/{uuid:resource instance id}
updates a single resource instance

Query Parameters

- **format** – {"xml", "json", "json-ld"}
• **indent** – number of spaces to indent json output

**Request Headers**

• **Authorization** – OAuth token for user authentication, see /o/token

• **Accept** – optional alternative to “format”, {“application/xml”, “application/json”, “application/id+json”}

**Example request:**

```bash
curl -H "Authorization: Bearer {token}" -X PUT -d {data in json-ld format} http://localhost:8000/resources/{resource instance id}
```

```bash
curl -H "Authorization: Bearer zo41Q1IMgAW30xOroiCUXjv3yici80s" -X PUT -d '{
  "@id": "http://localhost:8000/resource/47a1830c-74ec-11e8-bff6-14109fd34195",
  "@type": [
    "http://www.cidoc-crm.org/cidoc-crm/E18_Physical_Thing",
    "http://localhost:8000/graph/ab74af76-fa0e-11e6-9e3e-026d961c88e6"
  ],
  "http://www.cidoc-crm.org/cidoc-crm/P140i_was_attributed_by": [{
    "@id": "http://localhost:8000/node/1f7b4c8f-9932-47e4-9ec5-0284c77d893c/aspect/node/677f236e-09cc-11e7-8ff7-6c4008b05c4c",
    "@type": "http://www.cidoc-crm.org/cidoc-crm/E15_Identifier_Assignment",
    "http://www.cidoc-crm.org/cidoc-crm/P1_is_identified_by": [{
      "@id": "http://localhost:8000/tile/4a489c153683/node/677f39a8-09cc-11e7-834a-6c4008b05c4c",
      "@type": "http://www.cidoc-crm.org/cidoc-crm/E41_Appellation",
      "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {
        "@id": "http://localhost:8000/tile/4a489c153683/node/677f39a8-09cc-11e7-834a-6c4008b05c4c",
        "@type": "http://www.cidoc-crm.org/cidoc-crm/E55_Type",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "ecb20ae9-4a57-4011-83bf-1c936e2d6b6a"
      },
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "Claudio"
    },
    {
      "@id": "http://localhost:8000/tile/4a489c153683/node/677f39a8-09cc-11e7-834a-6c4008b05c4c",
      "@type": "http://www.cidoc-crm.org/cidoc-crm/E41_Appellation",
      "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {
        "@id": "http://localhost:8000/tile/4a489c153683/node/677f39a8-09cc-11e7-834a-6c4008b05c4c",
        "@type": "http://www.cidoc-crm.org/cidoc-crm/E55_Type",
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "81dd62d2-6701-4195-b74b-8057456bbab4"
      },
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "Alejandro"
    }
  ],
  "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {
    "@id": "http://localhost:8000/tile/2206a5bbf509/node/677f303d-09cc-11e7-9aa6-6c4008b05c4c",
    "@type": "http://www.cidoc-crm.org/cidoc-crm/E41_Appellation",
    "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {
      "@id": "http://localhost:8000/tile/2206a5bbf509/node/677f303d-09cc-11e7-9aa6-6c4008b05c4c",
      "@type": "http://www.cidoc-crm.org/cidoc-crm/E55_Type",
      "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "e4699732-efee-46c0-87e1-3f0a930a43db"
    }
  }
}]
```

(continues on next page)
Example json response:

```
HTTP/1.0 200 OK
Content-Type: application/json

{
    "@id": "http://localhost:8000/resource/47a1830c-74ec-11e8-bff6-14109fd34195",
    "@type": ["http://www.cidoc-crm.org/cidoc-crm/E18_Physical_Thing",
              "http://localhost:8000/graph/ab74af76-fa0e-11e6-9e3e-026d961c88e6"],
    "http://www.cidoc-crm.org/cidoc-crm/P140i_was_attributed_by": {
        "@id": "http://localhost:8000/tile/1f7b4c8f-9932-47e4-9ec5-0284c77d893c/node/67ff39e9-1e7f-8bff-6c4008b05c4c",
        "@type": "http://www.cidoc-crm.org/cidoc-crm/E15_Identifier_Assignment",
        "http://www.cidoc-crm.org/cidoc-crm/P1_is_identified_by": [ {
            "@id": "http://localhost:8000/tile/6efb8ac0-623c-47cb-9846-4a489c153683/node/67ff303d-09cc-11e7-9a66-6c4008b05c4c",
            "@type": "http://www.cidoc-crm.org/cidoc-crm/E41_Appellation",
            "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {  
                "@id": "http://localhost:8000/tile/6efb8ac0-623c-47cb-9846-4a489c153683/node/67ff303d-09cc-11e7-834a-6c4008b05c4c",
                "@type": "http://www.cidoc-crm.org/cidoc-crm/E55_Type",
                "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "ecb20ae9-a457-4011-83bf-1c936e2d6b6a"
            },
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "Claudio"
        },
        {
            "@id": "http://localhost:8000/tile/b53f2aaa-348b-4b73-9ff9-195090038c8b/node/67ff39e9-1e7f-8bff-6c4008b05c4c",
            "@type": "http://www.cidoc-crm.org/cidoc-crm/E41_Appellation",
            "http://www.cidoc-crm.org/cidoc-crm/P2_has_type": {  
                "@id": "http://localhost:8000/tile/b53f2aaa-348b-4b73-9ff9-195090038c8b/node/67ff39e9-1e7f-834a-6c4008b05c4c",
                "@type": "http://www.cidoc-crm.org/cidoc-crm/E55_Type",
                "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "81dd62d2-6701-4195-b74b-8057456bba4b"
            },
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#value": "Alejandro"
        }
    }
}
```

(continues on next page)
DELETE /resources/{uuid:resource instance id}  
deletes a single resource instance

Request Headers
  • Authorization – OAuth token for user authentication, see /o/token

Example request:
```
curl -H "Authorization: Bearer zo41Q1IMgAW30xOroiCUxjv3yci80s" -X DELETE http://localhost:8000/resources/00131129-7451-435d-aab9-33eb9031e6d1
```

Example response:
```
HTTP/1.0 200 OK
```

17.5 Mobile Projects

GET /mobileprojects  
get a list of mobile data collection projects that a user has been invited to participate in

Example request:
```
curl -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN_\h82PHVTCA9vdoHrcZxH-x5mbllly1537t3rGzcM" -X GET http://localhost:8000/mobileprojects
```

Example response:
```
HTTP/1.0 200 OK
Content-Type: application/json

[
  {
    "active": true,
    "bounds": "MULTIPOLYGON EMPTY",
    "cards": [],
    "createdby_id": 1,
    "datadownloadconfig": {
      "count": 1000,
      "custom": null,
      "download": false,
      "resources": []
    },
    "description": "A description of this project."
    "enddate": "2018-03-16",
] (continues on next page)
"groups": [
  6
],
"id": "e3d95999-2323-11e8-894b-14109fd34195",
"lasteditedby_id": 1,
"name": "Forbidden Project",
"startdate": "2018-03-04",
"tilecache": "",
"users": [
  1
]
}

Request Headers

- Authorization – JWT (JSON web token) for user authentication, see /auth/get_token
• Installation Commands
• ElasticSearch Management
• Import Commands
• Export Commands
• Managing Functions, DataTypes, Widgets, and Card Components
• Other Useful Django Commands

This page serves as a quick reference guide for working with Arches through a command prompt. Along with default Django commands, a good deal of Arches operations have been added to manage.py. In a command prompt, [activate your virtual environment](Dev-Installation#4-activate-the-virtual-environment), then run the following commands from your root app directory (the one that contains manage.py).

_All file or directory path parameters (_s_, _c_, _d_) should be absolute paths._

### 18.1 Installation Commands

#### 18.1.1 installing from a local repo clone

```
pip install -e .
```

- **-e**: This argument with the value . indicates to pip that it should link the local directory with the virtual environment.

Installs Arches into your virtual environment from a local clone of the archesproject/arches repo, or your own fork of that repo. To do this properly, create a new virtual environment and activate it, clone the repo you want, enter that repo’s root directory, and then run the command. Also, this command must be followed by:

```
pip install -r arches/install/requirements.txt
```
in order to properly install all of Arches’ python requirements. Make sure to use \ instead of / on Windows.

### 18.1.2 creating an Arches project

```
arches-project create <name_of_project> [(-d|--directory) <directory_name>]
```

- `-d`, `--directory` (Optional) The name of the directory you’d like your new project located in.

### 18.1.3 creating (or recreating) the database

```
python manage.py packages -o setup_db
```

- `-o` packages operation, in this case `setup_db`

Deletes and recreates the database, as defined by `settings.DATABASES['default']`. Likewise, this command will remove all existing data.

### 18.1.4 loading a package into a project

```
python manage.py packages -o load_package -s source_of_package [-db true]
```

- `-db` Add this argument with the value `true` to force the destruction and recreation of your database before loading the package.

The source (`-s`) of a package can be either a path to a local directory, the location of a local zipfile containing a package, or the url to a github repo archive that contains a package. For example, loading the v4 sample package from where it resides in github would just be:

```
python manage.py packages -o load_package -s https://github.com/archesproject/arches4→example-pkg/archive/master.zip
```

### 18.2 ElasticSearch Management

#### 18.2.1 install elasticsearch

```
python manage.py es install [-d path_to_destination_directory]
```

- `-d` (optional) Path to directory in which ElasticSearch should be installed. Default is the root of your project.

Installs ElasticSearch into the indicated location. If no destination is provided, ElasticSearch is placed in the app root directory. Read [[[Installing and Running Elasticsearch]]] for more.

#### 18.2.2 run elasticsearch

```
path/to/elasticsearch-5.2.1/bin/elasticsearch
```
18.2.3 reindex the database

```
python manage.py es index_database
```

**Important:** Memory usage will continuously increase during indexing, if you do not have DEBUG = False. This is because Django will store all db queries in memory, and a lot of them happen during indexing. Be wary of this when indexing a large database, or on servers with small memory provisions.

18.3 Import Commands

18.3.1 Import Resource Models or Branches in archesjson format

```
python manage.py packages -o import_graphs [-s path_to_json_directory_or_file]
```

- **-s** Path to the source file you are importing. If not specified, the command will look to settings.RESOURCE_GRAPH_LOCATIONS for directory paths

18.3.2 Import reference data in skos/rdf format

```
python manage.py packages -o import_reference_data -s 'path_to_rdf_file' [-ow { 'overwrite'|'ignore'}] [-st {'stage'|'keep'}]
```

18.3.3 Import business data

```
python manage.py packages -o import_business_data -s 'path_to_source_file' [-c 'path_to_mapping_file'] [-ow {'overwrite'|'append'}] [--create_concepts {'create'|'append'}] [--bulk_load]
```

- **-c** The path to the mapping file. The mapping file tells Arches how to map the columns from your csv file to the nodes in your resource graph. This option is required if there is not a mapping file named the same as the business data file and in the same directory with extension `.mapping` instead of `.csv` or `.json`.

- **-ow** Determines how resources with duplicate ResourceIDs will be handled: append adds more tile data to an existing resource; overwrite replaces any existing resource with the imported data. This option only applies to CSV import. JSON import always overwrites.

- **--bulk, --bulk_load** Bulk load values into the database. By setting this flag the system will use Django’s `bulk_create` operation. The model’s `save()` method will not be called, and the `pre_save` and `post_save` signals will not be sent.

- **--create_concepts** Creates or appends concepts and collections to your rdm according to the option you select. `create` will create concepts and collections and associate them to the mapped nodes. `append` will append concepts to the existing collections assigned to the mapped nodes and create collections for nodes that do not have an assigned collection.

See Importing a CSV for CSV formatting requirements
### 18.3.4 Import resource to resource relations

```bash
django-admin manage.py packages -o import_business_data_relations -s 'path_to_relations_file'
```

See *Importing Resource Relations*

### 18.4 Export Commands

#### 18.4.1 export branch or resource model schema

```bash
django-admin manage.py packages -o export_graphs -d 'path_to_destination_directory' -g uuid/branches/resource_models/all
```

- `-o` packages operation, in this case `export_graphs`
- `-d` Absolute path to destination directory
- `-g` UUID of specific graph, or `branches` for all branches, `resource_models` for all resource models, or `all` for everything.

Exports Resource Models and/or Branches. Note that sometimes (as in this case) Resource Models and Branches are generically called “graphs”.

#### 18.4.2 export business data to csv or json

```bash
django-admin manage.py packages -o export_business_data -d 'path_to_destination_directory' -f 'csv' or 'json' [-c 'path_to_mapping_file' -g 'resource_model_uuid' -single_file]
```

- `-o` packages operation, in this case `export_business_data`
- `-d` Absolute path to destination directory
- `-f` Export format, must be `csv` or `json`
- `-c` (required for `csv`) Absolute path to the mapping file you would like to use for your `csv` export.
- `-single_file` (optional for `csv`) Use this parameter if you’d like to export your grouped data to the same `csv` file as the rest of your data.
- `-g` (required for `json`, optional for `csv`) The resource model UUID whose instances you would like to export.

Exports business data to `csv` or `json` depending on the `-f` parameter specified. For `csv` export a mapping file is required. The exporter will export all resources of the type indicated in the `resource_model_id` property of the mapping file and the `-g` parameter will be ignored. For `json` export no mapping file is required, instead a resource model uuid should be passed into the `-g` command.

Note that in a Windows command prompt, you may need to replace `’` with `"`.

#### 18.4.3 export business data to shapefile
python manage.py export shp -t 'name_of_db_view' -d 'output_directory'

- **-t** A resource instance database view
- **-d** The destination directory for point, line, and polygon shapefiles, created when the command is run.

### 18.4.4 business data export examples

```python
python manage.py packages -o export_business_data -f 'csv' -c 'path_to_mapping_file'
```

Exports all business data of the resource model indicated in the mapping file. Two files are created. The first file contains one row per resource (if you resources all have the same geometry type this file can be used to create a shape file in QGIS or other program). The second file contains the grouped attributes of your resources (for instance, alternate names, additional classifications, etc.).

```python
python manage.py packages -o export_business_data -f 'json' -g 'resource_model_id'
```

- **-f** ‘json’ or ‘csv’

Exports all business data of the passed in resource_model_id to the specified file format. Take a look at the RESOURCE_FORMATERS dictionary in Arches’ settings.py for some other interesting options.

### 18.5 Other Data Management Commands

```python
python manage.py resources remove_resources [-g graph_id][-y]
```

- **-g** A Graph UUID to remove all the resource instances of.
- **-y** Forces this command to run without interactive confirmation.

Removes all resources from your database, but leaves the all resources models, branches, thesauri, and collections intact.

```python
python manage.py packages -o create_mapping_file -d 'path_to_destination_directory' -g 'comma separated graph uuids'
```

- **-d** Path to directory to place the output in.
- **-g** One or more graph UUIDs to create a mapping for.

This mimics the ‘Create Mapping File’ command from the Arches Designer UI.

```python
python manage.py packages -o import_mapping_file -s 'path_to_mapping_file'
```

Imports a mapping file for a particular resource model. This will be used as the export mapping file for a resource by default (e.g. for search export).
18.6 Ontology Commands

18.6.1 load an ontology or ontology extension

```
python manage.py load_ontology [-s <path to ontology>] [-x <path to extension>] [-r/--reload] [-vn <version number>]
```

- `-s`  Path to new ontology xml file to load  
- `-x`  Path to ontology extension xml file to load. You’ll be prompted to choose which existing ontology you would like to extend.  
- `-vn`  If loading a new ontology or adding an extension, you must supply a version number.  
- `-r`  Reload ontology. You’ll be prompted to choose which ontology to reload.

18.7 Managing Functions, DataTypes, Widgets, and Card Components

To learn how to build new Functions, DataTypes, or Widgets, please see ref: Creating New Functions, Widgets, Card Components, and DataTypes.  

Note that when importing Widgets and associated DataTypes, Widgets must be registered first.

18.7.1 function commands

list registered functions

```
python manage.py fn list
```

Lists all currently registered functions.

registering functions

```
python manage.py fn register --source path/to/your/function.py
```

Register a newly created function. These .py files should sit in your projects functions directory.

unregistering functions

```
python manage.py fn unregister -n 'Sample Function'
```

Unregister a function. Use the function name that is returned by fn list.

18.7.2 datatype commands

list registered datatypes

```
python manage.py datatype list
```

Lists all currently registered datatypes.

registering and updating datatypes
Registers a new datatype, in this example as defined in `wkt_point.py`.

```
python manage.py datatype register --source /Users/me/Documents/projects/mynewproject/
  → mynewproject/datatypes/wkt_point.py
```

Updates a datatype, necessary anytime changes are made to your datatype’s properties.
```
python manage.py datatype update --source /Users/me/Documents/projects/mynewproject/
  → mynewproject/datatypes/wkt_point.py
```

--source Location of the .py file that defines the datatype.

**unregister a datatype**
```
python manage.py datatype unregister -d 'wkt-point'
```

Unregisters a datatype, in this example a datatype named `wkt-point`.
```
-d Name of datatype to unregister. Use the datatype name that is returned by
datatype list.
```

### 18.7.3 widget commands

All widget-related commands are identical to those for datatypes, just substitute `widget` for `datatype`. Also note that where datatypes are defined in .py files, widgets are defined in .json files.

**18.7.4 card component commands**

All component-related commands are identical to those for widgets, just substitute `card_component` for `widget`. JSON files are used to register Card Components.

### 18.8 Creating Map Layers

See *Creating New Map Layers* for file format requirements and other in-depth information.

#### 18.8.1 MapBox

```
python manage.py packages -o add_mapbox_layer -j /path/to/mapbox_style.json -n "New
  → MapBox Layer" [{-b|--is_basemap}] [{-i|--layer_icon} 'icon_class']
```

- `j` The path to the Mapbox JSON file
- `n` The name of the Mapbox layer

**18.8.2 Tileserver**

```
python manage.py packages -o add_tileserver_layer -t /path/to/tileserver_config.json -
  → n "New [Mapnik] Tileserver Layer" [-m /path/to/mapnik_config.xml] [{-b|--is_basemap}
  →] [{-i|--layer_icon} 'icon_class']
```

- `j` The path to the Tileserver JSON file
The name of the Tileserver layer
(Optional) The path to the Mapnik XML file

```
python manage.py packages -o delete_tile_server_layer -n <layer-name>
```

Deletes the Tileserver layer with the specified name.

### 18.8.3 Common Options

- `-i, --layer_icon` (Optional) The FontAwesome icon class to use for a Tileserver layer. Default: `fa fa-globe`
- `-b, --is_basemap` (Optional) Add this option to make the layer a basemap.

### 18.9 Other Useful Django Commands

#### 18.9.1 Run the django webserver

```
python manage.py runserver
```

Run the Django dev server. Add `0.0.0.0:8000` to explicitly set the host and port, which may be necessary when using remote servers, like an AWS EC2 instance. More about `runserver`.

#### 18.9.2 collect static files

```
python manage.py collectstatic
```

Collects all static files and places them in a single directory. Generally only necessary in production. Also allows all static files to be hosted on another server).

Django’s full `manage.py` commands are documented here.
Creating a Development Environment

The following is our recommendation for creating an Arches environment that works well for developers. The first thing to consider is the general structure that will be in place, presumably all in the same directory:

- **/ENV** - A virtual environment, we'll name it ENV. This should be built from Python 2.7.x, and you should make sure pip is up-to-date. `python -m pip install --upgrade pip`

- **/arches** - The local clone of your fork of the archesproject/arches repo, which gets installed into your virtual environment. Often, this part of the code is referred to as “core Arches”.

- **/my_project** - The location of your Arches project. This is the app in which you will be making the majority of your front-end customizations (new images, new template contents, etc.).

- **/my_package** - The location of your Arches package. Packages contain all of the custom database definitions that you will use. Resource Models, Branches, Concepts, Datatypes, Widgets, Functions, business data (resources for initial load)… these are all stored in a package. Packages are loaded into Projects via the command line.

### 19.1 Setting Everything Up

1. Make sure all dependencies are installed. You may also have external services in mind for some of the Arches dependencies, like a remote Postgres/Postgis or ElasticSearch installation. In that case you'll need credentials eventually, but not immediately. You'll also need git installed locally.

2. Make a new virtual environment built from Python 2.7.x and activate it.

   ```bash
   virtualenv ENV
   source ENV/bin/activate
   ```

   or, to activate on Windows:

   ```bash
   ENV\Scripts\activate
   ```

3. Clone the archesproject/arches repo. We recommend that you clone your own fork of the repo, but you can also clone the original repo if you don’t plan to contribute code.
4. Link your local clone of the repo to ENV. This is instead of using pip install arches which would install the pypi Arches distribution directly into ENV. When you install the local clone, any code changes you make inside of /arches (like checking out a new git branch) will be immediately reflected in your runtime environment.

```bash
(ENV)$ git clone https://github.com/archesproject/arches.git
(ENV)$ cd arches
(ENV)$ pip install -e .
(ENV)$ pip install -r arches/install/requirements.txt
(ENV)$ pip install -r arches/install/requirements_dev.txt
(ENV)$ cd ..
```

5. Create the new project you will be working on. In this project you’ll make all of the necessary changes to your templates, define all of your settings.py variables, run manage.py, etc. This is also where all your yarn components will be installed.

```bash
(ENV)$ arches-project create my_project
```

or on Windows:

```bash
(ENV)\> ENV\Scripts\python.exe ENV\Scripts\arches-project create my_project
```

**Note:** You can use the option `[-d|--directory] <directory_name>` to change the directory your new project will be created in.

6. You can now enter your new project and create a package

```bash
(ENV)$ cd my_project
(ENV)$ python manage.py packages -o create_package -d ../my_package
```

on Windows remember to change the forward slashes to backward slashes.

You now have your development environment ready. It should look like:

```bash
/Projects
 /env          # virtual environment, activate this but don't modify things inside of it
    /arches    # clone of your core Arches fork (git repo)
    /my_project # the project, where you store settings and from which you run manage.
    /my_package # the package, where you should export your resource models, etc. so you can share them (could be a git repo)
```

Before proceeding, you should either alter the settings.py file in your project to give it your postgres/postgis credentials `DATABASES['default'],` or add a settings_local.py file and put the credentials there to keep them out of version control (and, if you are on windows, you need `GDAL_LIBRARY_PATH = path/to/gdalxxx.dll`). Then get ElasticSearch running and you can run `python manage.py packages -o setup_db`. Now you can run `python manage.py runserver` and view Arches in a browser.

**Note:** Whenever changes occur in the Arches code base, you may want to update your local clone. If you are not going to make a new project from the updated code, you should run `yarn install` and `python manage.py`
migrate to fully update your local install. These commands should be run from within /arches.
Creating New Functions, Widgets, Card Components, and DataTypes

Functions, Widgets, Card Components, and DataTypes have overlapping functionality and architecture. These constructs are used to add extra functionality to Arches upon data entry or data visualization.

At a high level, if you’re trying to add a custom interface for entering business data, you’ll use a Widget. If you need to enter and display a type of data unknown to Arches, you’ll need a new DataType to go with your Widget. If you need sophisticated side effects or to perform additional computation based on the data entered, it’s time for a Function.

**Note:** To develop these components, you’ll need familiarity with Django templates, intermediate skills in the Python programming language, and some basic familiarity with the Knockout.js front-end framework.

### 20.1 Functions

Functions are the most powerful extension to Arches. Functions associated with a Resource are called during various CRUD operations, and have access to any server-side model. Proficient Python/Django developers will find few limitations extending an Arches Project with Functions.

Function must be created, registered, and then associated with a Resource Model.

#### 20.1.1 Creating a Function

A Function comprises three separate files, which should be seen as front-end/back-end complements. On the front-end, you will need a component made from a Django HTML template and JavaScript pair, which should share the same basename.

In your Project, these files must be placed like so:

```
/myproject/myproject/media/js/views/components/functions/
spatial_join.js  /myproject/myproject/templates/views/components/
functions/spatial_join.htm
```
The third file is a Python file which contains a dictionary telling Arches some important details about your Function, as well as its main logic.

```
/myproject/myproject/functions/spatial_join.py
```

**Note:** As in the example above, its advisable that all of your files share the same basename. (If your Function is moved into a Package, this is necessary.) A new Project should have an example function in it whose files you can copy to begin this process.

### 20.1.2 Defining the Function’s Details

The first step in creating a function is defining the details that are in the top of your Function’s .py file.

```python
details = {
    'name': 'Sample Function',
    'type': 'node',
    'description': 'Just a sample demonstrating node group selection',
    'defaultconfig': {'selected_nodegroup': ''},
    'classname': 'SampleFunction',
    'component': 'views/components/functions/sample-function'
}
```

- **name** Required Name is used to unregister a function, and shows up in the `fn list` command.
- **type** Required As of version 4.2, this should always be set to `node`
- **description** Optional Add a description of what your Function does.
- **defaultconfig** Required A JSON object with any configuration needed to serve your function’s logic
- **classname** Required The name of the python class that holds this Function’s logic.
- **component** Required Canonical path to html/js component.

**More about the defaultconfig field**

Any configuration information you need your Function to access can be stored here. If your function needs to calculate something based on the value of an existing Node, you can refer to it here. Or, if you want your Function to e-mail an administrator whenever a specific node is changed, both the Node ID and the email address to be used are good candidates for storage in the `defaultconfig` dictionary.

The `defaultconfig` field serves both as a default, and as your user-defined schema for your function’s configuration data. Your front-end component for the function will likely collect some of this configuration data from the user and store it in the `config` attribute of the pertinent `FunctionXGraph`.

### 20.1.3 Writing your Function Logic

In your Function’s Python code, you have access to all your server-side models. You’re basically able to extend Arches in any way you please. You may want to review the [Data Model](https://docs.archesproject.org) documentation.

### 20.1.4 Function Hooks

Your function needs to extend the `BaseFunction` class. Depending on what you are trying to do, you will need to implement the `get`, `save`, `delete`, `on_import`, and/or `after_function_save` methods.
class MyFunction(BaseFunction):
    def get(self):
        raise NotImplementedError
    def save(self, tile, request):
        raise NotImplementedError
    def delete(self, tile, request):
        raise NotImplementedError
    def on_import(self, tile):
        raise NotImplementedError
    def after_function_save(self, functionxgraph, request):
        raise NotImplementedError

Note: Not all of these methods are called in the current Arches software. You can also leave any of them unimplemented, and the `BaseFunction` class will raise a `NotImplementedError` for you. Arches is designed to gracefully ignore these exceptions for functions.

A detailed description of current functionality is below.

**save and delete**

The `Tile` object will look up all its Graph’s associated Functions upon being saved. Before writing to the database, it calls each function’s `save` method, passing itself along with the Django `Request` object. This is likely where the bulk of your function’s logic will reside.

The `Tile` object similarly calls each of its graph’s functions’ `delete` methods with the same parameters. Here, you can execute any cleanup or other desired side effects of a Tile’s deletion. Your `delete` implementation will have the same signature as `save`.

**after_function_save**

The Graph view passes a `FunctionXGraph` object to `after_function_save`, along with the request.

The `FunctionXGraph` object has a `config` attribute which stores that instance’s version of the `defaultconfig` dictionary. This is a good opportunity, for example, to programmatically manipulate the Function’s configuration based on the Graph or any other server-side object.

You can also write any general logic that you’d like to fire upon the assignment of a Function to a Resource.

**on_import**

The import module calls `on_import` if the file format is a JSON-formatted Arches file, and passes an associated Tile object.

CSV imports do not call this hook.
20.1.5 The UI Component

Having implemented your function’s logic, it’s time to develop the front-end components required to associate it with Resources and provide any configuration data.

The component you develop here will be rendered in the Resource Manager when you associate the function with a Resource, and this is where you’ll put any forms or other UI artifacts used to configure the Function.

Developing your Function’s UI component is very similar to developing Widgets. More specific guidelines are in progress, but for now, refer to the sample code in your project’s templates/views/components/functions/ directory, and gain a little more insight from the templates/views/components/widgets/ directory. The complementary JavaScript examples will be located in media/js/views/components/functions/ and media/js/views/components/widgets directories.

20.1.6 Registering Functions

First, list the names of functions you already have registered:

(ENV)$ python manage.py fn list

Now you can register your new function with

(ENV)$ python manage.py fn register-source <path to your function's .py file>

For example:

(ENV)$ python manage.py fn register --source /Documents/projects/mynewproject/ →mynewproject/functions/sample_function.py

Now navigate to the Function Manager in the Arches Designer to confirm that your new function is there and functional. If it’s not, you may want to unregister your function, make additional changes, and re-register it. To unregister your function, simply run

(ENV)$ python manage.py fn unregister --name 'Sample Function'

All commands are listed in Command Line Reference - Function Commands.

20.2 Widgets

Widgets allow you to customize how data of a certain DataType is entered into Arches, and further customize how that data is presented in Reports. You might have several Widgets for a given DataType, depending on how you want the Report to look or to match the context of a certain Resource.

Widgets are primarily a UI artifact, though they are closely tied to their underlying DataType.

To develop a custom Widget, you’ll need to write three separate files, and place them in the appropriate directories. For the appearance and behavior of the Widget, you’ll need a component made of a Django template and JavaScript file placed like so:

project_name/templates/views/components/widgets/sample-widget.htm
project_name/media/js/views/components/widgets/sample-widget.js

To register and configure the Widget, you’ll need a JSON configuration file:

project_name/widgets/sample-widget.json
20.2.1 Configuring your Widget

To start, here is a sample Widget JSON file:

```json
{
   "name": "sample-widget",
   "component": "views/components/widgets/sample-widget",
   "defaultconfig": {
      "x_placeholder": "Longitude",
      "y_placeholder": "Latitude"
   },
   "helptext": null,
   "datatype": "sample-datatype"
}
```

The most important field here is the `datatype` field. This controls where your Widget will appear in the Arches Resource Designer. Nodes each have a DataType, and Widgets matching that DataType will be available when you’re designing your Cards. The value must match an existing DataType within Arches.

You can also populate the `defaultconfig` field with any configuration data you wish, to be used in your Widget’s front-end component.

20.2.2 Designing Your Widget

Your Widget’s template needs to include three Django template “blocks” for rendering the Widget in different contexts within Arches. These blocks are called `form`, `config_form`, and `report`. As you might guess from their names, `form` is rendered when your Widget appears on a Card for business data entry, `config_form` is rendered when you configure the Widget on a card when designing a Resource, and `report` controls how data from your Widget is presented in a Report.

Here is an example:

```html
{% extends "views/components/widgets/base.htm" %}
{% load i18n %}

{% block form %}
<div class="row widget-wrapper">
   <div class="form-group">
      <label class="control-label widget-input-label" for="" data-bind="text:label">
         →
      </label>
      <div class="col-xs-12">
         <input type="number" data-bind="textInput: x_value, attr: {placeholder: x_placeholder}" class="form-control input-lg widget-input" style="margin-bottom: 5px">
      </div>
      <div class="col-xs-12">
         <input type="number" data-bind="textInput: y_value, attr: {placeholder: y_placeholder}" class="form-control input-lg widget-input" style="margin-bottom: 5px">
      </div>
      <div class="col-xs-12">
         <input type="text" data-bind="textInput: srid" class="form-control input-lg widget-input" style="margin-bottom: 5px">
      </div>
      <div class="col-xs-12">
         <label class="control-label widget-input-label" for="">Preview</label>
         <input disabled type="text" data-bind="textInput: preview" class="form-control input-lg widget-input">
      </div>
   </div>
</div>
```

(continues on next page)
To pull it all together, you’ll need to write a complementary JavaScript file. The Arches UI uses Knockout.js, and the best way to develop your Widget in a compatible way is to write a Knockout component with a ViewModel corresponding to your Widget’s view (the Django template).

Here is an example, continuing with our sample-widget:

```javascript
define(['ko', 'underscore', 'viewmodels/widget'], function (ko, _, WidgetViewModel) {

  /*
   * registers a text-widget component for use in forms
   * @function external:"ko.components".text-widget
   * @param {object} params
   * @param {string} params.value - the value being managed
   * @param {function} params.config - observable containing config object
   * @param {string} params.config().label - label to use alongside the text input
   * @param {string} params.config().placeholder - default text to show in the text
   * @param {object} params.configKeys - a list of keys to use
   * @param {object} params.configKeys.x - x-coordinate
   * @param {object} params.configKeys.y - y-coordinate
   * input
   */

  return ko.components.register('sample-widget', {
    viewModel: function (params) {
      params.configKeys = ['x_placeholder', 'y_placeholder'];
      WidgetViewModel.apply(this, [params]);
      var self = this;
      if (this.value()) {
        var coords = this.value().split('POINT(')[1].replace(')', '').split(',').map(Number);
        var srid = this.value().split(';')[0].split('=').map(Number);
        this.x_value = ko.observable(coords[0]);
        this.y_value = ko.observable(coords[1]);
      }
    }
  });
});
```

(continues on next page)
20.2.3 Registering your Widget

After placing your Django template and JavaScript files in their respective directories, you are now ready to register your Widget:

```python
python manage.py widget register --source /Documents/projects/mynewproject/
    mynewproject/widgets/sample-widget.json
```

The command will confirm your Widget has been registered, and you can also see it with:

```python
python manage.py widget list
```

If you make an update to your Widget, you can load the changes to Arches with:

```python
python manage.py widget update --source /Documents/projects/mynewproject/mynewproject/
    widgets/sample-widget.json
```

All the Widget commands are detailed in *Command Line Reference - Widget Commands*.

20.3 DataTypes

A DataType defines a type of business data. DataTypes are associated with Nodes and Widgets. When you are designing your Cards, the Widgets with the same DataType as the Node you are collecting data for will be available. In your Branches, each Node with a DataType will honor the DataType configuration you specify when you create it.

The simplest (non-configurable, non-searchable) DataTypes consist of a single Python file. If you want to provide Node-specific configuration to your DataType (such as whether to expose a Node with that DataType to Advanced Search or how the data is rendered), you’ll also develop a UI component comprising a Django template and JavaScript file.

In your Project, these files must be placed accordingly:

Optional Configuration Component:
To begin, let’s examine the sample-datatype included with Arches:

```python
from arches.app.datatypes.base import BaseDataType
from arches.app.models import models
from arches.app.models.system_settings import settings

sample_widget = models.Widget.objects.get(name='sample-widget')

details = {
    'datatype': 'sample-datatype',
    'iconclass': 'fa fa-file-code-o',
    'modulename': 'datatypes.py',
    'classname': 'SampleDataType',
    'defaultwidget': sample_widget,
    'defaultconfig': {'placeholder_text': ''},
    'configcomponent': 'views/components/datatypes/sample-datatype',
    'configname': 'sample-datatype-config',
    'isgeometric': False
}

class SampleDataType(BaseDataType):
    def validate(self, value, row_number=None, source=None):
        errors = []
        try:
            value.upper()
        except:
            errors.append({'type': 'ERROR', 'message': 'datatype: {0} value: {1} {2} --> {3} - {4}. {5}'.format(self.datatype_model.datatype, value, row_number, source, this is not a string', 'This data was not imported.'))}
        return errors

    def append_to_document(self, document, nodevalue, nodeid, tile):
        document['strings'].append({'string': nodevalue, 'nodegroup_id': tile.nodegroup_id})

    def transform_export_values(self, value, *args, **kwargs):
        if value != None:
            return value.encode('utf8')

    def get_search_terms(self, nodevalue, nodeid=None):
        terms = []
        if nodevalue is not None:
            if settings.WORDS_PER_SEARCH_TERM == None or (len(nodevalue.split(' ')) <= settings.WORDS_PER_SEARCH_TERM):
                terms.append(nodevalue)
        return terms

    def append_search_filters(self, value, node, query, request):
        try:
            if value['val'] != '':
```
(continues on next page)
20.3.1 Writing Your DataType

Your DataType needs, at minimum, to implement the validate method. You’re also likely to implement the transform_import_values or transform_export_values methods. Depending on whether your DataType is spatial, you may need to implement some other methods as well. If you want to expose Nodes of your DataType to Advanced Search, you’ll also need to implement the append_search_filters method.

You can get a pretty good idea of what methods you need to implement by looking at the BaseDataType class in the Arches source code located at arches/app/datatypes/base.py and below:

```python
import json
from django.core.urlresolvers import reverse
from arches.app.models import models

class BaseDataType(object):

    def __init__(self, model=None):
        self.datatype_model = model

    def validate(self, value, row_number=None, source=None):
        return []

    def append_to_document(self, self, document, nodevalue, nodeid, tile):
        """
        Assigns a given node value to the corresponding key in a document in
        preparation to index the document
        """
        pass

    def after_update_all(self):
        """
        Refreshes mv_geojson_geoms materialized view after save.
        """
        pass

    def transform_import_values(self, self, value, nodeid):
        """
        Transforms values from probably string/wkt representation to specified
        datatype in arches
        """
        return value

    def transform_export_values(self, self, value, *args, **kwargs):
        """
```

(continues on next page)
Transforms values from probably string/wkt representation to specified
datatype in arches

```python
return value
```

def get_bounds(self, tile, node):
    ""
    Gets the bounds of a geometry if the datatype is spatial
    ""
    return None

def get_layer_config(self, node=None):
    ""
    Gets the layer config to generate a map layer (use if spatial)
    ""
    return None

def should_cache(self, node=None):
    ""
    Tells the system if the tileserver should cache for a given node
    ""
    return False

def should_manage_cache(self, node=None):
    ""
    Tells the system if the tileserver should clear cache on edits for a
given node
    ""
    return False

def get_map_layer(self, node=None):
    ""
    Gets the array of map layers to add to the map for a given node
    should be a dictionary including (as in map_layers table):
    nodeid, name, layerdefinitions, isoverlay, icon
    ""
    return None

def clean(self, tile, nodeid):
    ""
    Converts '' values to null when saving a tile.
    ""
    if tile.data[nodeid] == '':
        tile.data[nodeid] = None

def get_map_source(self, node=None, preview=False):
    ""
    Gets the map source definition to add to the map for a given node
    should be a dictionary including (as in map_sources table):
    name, source (json)
    ""
    tileserver_url = reverse('tileserver')
    if node is None:
        return None
    source_config = {
        "type": "vector",
        "tiles": ["%s/z/%x/%y.pbf" % (tileserver_url, node.nodeid)]
    }
count = None

if preview == True:
    count = models.TileModel.objects.filter(data__has_key=str(node.nodeid)).
    →count()

    if count == 0:
        source_config = {
            "type": "geojson",
            "data": {
                "type": "FeatureCollection",
                "features": [
                    {
                        "type": "Feature",
                        "properties": {
                            "total": 1
                        },
                        "geometry": {
                            "type": "Point",
                            "coordinates": [
                                -122.4810791015625,
                                37.93553306183642
                            ]
                        }
                    },
                    {
                        "type": "Feature",
                        "properties": {
                            "total": 100
                        },
                        "geometry": {
                            "type": "Point",
                            "coordinates": [
                                -58.30078125,
                                -18.075412438417395
                            ]
                        }
                    },
                    {
                        "type": "Feature",
                        "properties": {
                            "total": 1
                        },
                        "geometry": {
                            "type": "LineString",
                            "coordinates": [
                                [ -179.82421875,
                                  44.213709909702054
                                ],
                                [ -154.16015625,
                                  32.69486597787505
                                ],
                                [ -171.5625,
                                  18.812717856407776
                                ]
                            ]
                        }
                    }
                ]
            }
        }
    }
[[-145.72265625, 2.986927393334876], [-158.37890625, -30.145127183376115]]

},

"type": "Feature",
"properties": {
"total": 1
},
"geometry": {
"type": "Polygon",
"coordinates": [
[[-50.9765625, 22.59372606392931], [-23.37890625, 22.59372606392931], [-23.37890625, 42.94033923363181], [-50.9765625, 42.94033923363181], [-50.9765625, 22.59372606392931]]
}

},

"type": "Feature",
"properties": {
"total": 1
},
"geometry": {
"type": "Polygon",
"coordinates": [
[-27.59765625, -14.434680215297268]]
}

(continues on next page)
[ -24.43359375,  
-32.10118973232094 
],  
[  
0.87890625,  
-31.653381399663985 
],  
[  
2.28515625,  
-12.554563528593656 
],  
[  
-14.23828125,  
-0.3515602939922709 
],  
[  
-27.59765625,  
-14.434680215297268 
] }  
] }  
] }  
}  
}  
}  
return {
    "nodeid": node.nodeid,
    "name": "resources-$s" % node.nodeid,
    "source": json.dumps(source_config),
    "count": count
}  

def get_pref_label(self, nodevalue):
    
    Gets the prefLabel of a concept value
    
    return None

def get_display_value(self, tile, node):
    
    Returns a list of concept values for a given node
    
    return unicode(tile.data[str(node.nodeid)])

def get_search_terms(self, nodevalue, nodeid=None):
    
    Returns a nodevalue if it qualifies as a search term
    
    return []

def append_search_filters(self, value, node, query, request):
    
    Allows for modification of an elasticsearch bool query for use in advanced search

(continues on next page)
the validate method

Here, you write logic that the Tile model will use to accept or reject a Node’s data before saving. This is the core implementation of what your DataType is and is not.

The validate method returns an array of errors. If the array is empty, the data is considered valid. You can populate the errors array with any number of dictionaries with a type key and a message key. The value for type will generally be ERROR, but you can provide other kinds of messages.

the append_search_filters method

In this method, you’ll create an ElasticSearch query Nodes matching this datatype based on input from the user in the Advanced Search screen. (You design this input form in your DataType’s front-end component.)

Arches has its own ElasticSearch query DSL builder class. You’ll want to review that code for an idea of what to do. The search view passes your DataType a Bool() query from this class, which you call directly. You can invoke its must, filter, should, or must-not methods and pass complex queries you build with the DSL builder’s Match class or similar. You’ll execute this search directly in your append_search_filters method.

In-depth documentation of this part is planned, but for now, look at the core datatypes located in Arches’ source code for examples of the approaches you can take here.

Note: If you’re an accomplished Django developer, it should also be possible to use Elastic’s own Python DSL builder in your Project to build the complex search logic you’ll pass to Arches’ Bool() search, but this has not been tested.

20.3.2 Configuring your DataType

You’ll need to populate the details dictionary to configure your new DataType.

def handle_request(self, current_tile, request, node):
    """
    Updates files
    """
    pass

datatype Required The name of your datatype. The convention in Arches is to use kebab-case here.

iconclass Required The FontAwesome icon class your DataType should use. Browse them here.
**modulename Required** This should always be set to `datatypes.py` unless you’ve developed your own Python module to hold your many DataTypes, in which case you’ll know what to put here.

**classname Required** The name of the Python class implementing your datatype, located in your DataType’s Python file below these details.

**defaultwidget Required** The default Widget to be used for this DataType.

**defaultconfig Optional** You can provide user-defined default configuration here.

**configcomponent Optional** If you develop a configuration component, put the fully-qualified name of the view here. Example: `views/components/datatypes/sample-datatype`

**configname Optional** The name of the Knockout component you have registered in your UI component’s JavaScript file.

**isgeometric Required** Used by the Arches UI to determine whether to create a Map Layer based on the DataType, and also for caching. If you’re developing such a DataType, set this to True.

**Important:** `configcomponent` and `configname` are required together.

### 20.3.3 Developing the Configuration Component

Your component JavaScript file should register a Knockout component with your DataType’s `configname`. This component should be an object with two keys: `viewModel`, and `template`.

The value for `viewModel` should be a function where you put the logic for your template. You’ll be setting up Knockout observable and computed values tied to any form elements you’ve developed to collect Advanced Search or Node-level configuration information from the user.

The value for `template` should be another object with the key `require`, and the value should be `text! datatype-config-templates/<your-datatype-name>`. Arches will know what to do with this – it comes from the value you supplied in your Python file’s `details` dictionary for `configcomponent`.

Pulling it all together, here’s the JavaScript portion of Arches’ `date` DataType.

```javascript
define(['knockout'], function (ko) {
  var name = 'date-datatype-config';
  ko.components.register(name, {
    viewModel: function (params) {
      var self = this;
      this.search = params.search;
      if (this.search) {
        var filter = params.filterValue();
        this.viewMode = 'days';
        this.op = ko.observable(filter.op || '');
        this.searchValue = ko.observable(filter.val || '');
        this.filterValue = ko.computed(function () {
          return {
            op: self.op(),
            val: self.searchValue()
          }
        }).extend({ throttle: 750 });
        params.filterValue(this.filterValue);
        this.filterValue.subscribe(function (val) {
          params.filterValue(val);
        });
      }
    }
  });
});
```

(continues on next page)
Advanced Search Rendering

If you’re supporting Advanced Search functionality for Nodes with your DataType, your Django template will include a search block, conditionally rendered by Knockout.js if the search view is active. Here’s the one from the boolean datatype:

<!-- ko if: $data.search -->

`<div class="col-sm-12">
  <select class="resources" data-bind="value: searchValue, chosen: {width: '100%', disable_search_threshold: 15}, options: [{id:'t', name:trueLabel}, {id:'f', name:falseLabel}], optionsText: 'name', optionsValue: 'id'>
    <option value="eq"> = </option>
    <option value="gt"> > </option>
    <option value="lt"> < </option>
    <option value="gte"> >= </option>
    <option value="lte"> <= </option>
  </select>
</div>`

Note the <!-- ko if: $data.search --> directive opening and closing the search block. This is not an HTML comment – it’s Knockout.js-flavored markup for the conditional rendering.

Arches’ built-in date DataType does not use the Django template block directive, but only implements advanced search, and contains a more sophisticated example of the component logic needed:

`<!-- ko if: $data.search -->

<% load i18n %>

<%-- ko if: $data.search -->

<% div class="col-md-4 col-lg-3" %>
  <select class="resources" tabindex="-1" style="display: none;" data-bind="value: op, chosen: {width: '100%', disable_search_threshold: 15}"
    op, chosen: {width: '100%', disable_search_threshold: 15} %> 
    <option value="eq"> = </option>
    <option value="gt"> > </option>
    <option value="lt"> < </option>
    <option value="gte"> >= </option>
    <option value="lte"> <= </option>
  </select>
</div>

<% div class="col-md-8 col-lg-9" %>
  <input type="" placeholder="" class="form-control input-md widget-input" data-bind="value: searchValue, datepicker: {format: 'YYYY-MM-DD', viewMode: viewMode, minDate: false, maxDate: false}"
    viewMode: viewMode, minDate: false, maxDate: false">
</div>

<!-- /ko -->`
Node-specific Configuration

This section of your template should be enclosed in Knockout-flavored markup something like: <!-- ko if: $data.graph -->, and in your Knockout function you should follow the convention and end up with something like if (this.graph) {

Here, you put form elements corresponding to any configuration you’ve implemented in your DataType. These should correspond to keys in your DataType’s defaultconfig.

Arches’ boolean DataType has the following defaultconfig:

```javascript
{'falseLabel': 'No', 'trueLabel': 'Yes'}
```

You can see the corresponding data bindings in the Django template:

```html
<!-- ko if: $data.graph -->
<div class="control-label">
    {% trans "Label 'True'" %}
</div>
<div class="col-xs-12 pad-no crud-widget-container">
    <input type="" id="" class="form-control input-md widget-input" data-bind="value: →trueLabel, valueUpdate: 'keyup'">
</div>
<div class="control-label">
    {% trans "Label 'False'" %}
</div>
<div class="col-xs-12 pad-no crud-widget-container">
    <input type="" id="" class="form-control input-md widget-input" data-bind="value: →falseLabel, valueUpdate: 'keyup'">
</div>
<!-- /ko -->
```

And finally, here is the boolean DataType’s JavaScript file in its entirety:

```javascript
define(['knockout'], function (ko) {
    var name = 'boolean-datatype-config';
    ko.components.register(name, {
        viewModel: function (params) {
            var self = this;
            this.search = params.search;
            this.graph = params.graph;

            this.trueLabel = params.config ? params.config.trueLabel : params.node.
            →config.trueLabel;
            this.falseLabel = params.config ? params.config.falseLabel : params.node.
            →config.falseLabel;

            if (this.search) {
                var filter = params.filterValue();
                this.searchValue = ko.observable(filter.val || '');
                this.filterValue = ko.computed(function () {
                    return {
                        val: self.searchValue()
                    }
                });
                params.filterValue(this.filterValue());

            }
        });
    });
}),
```

(continues on next page)
20.3.4 Registering your DataType

These commands are identical to working with Widgets, but you use the word `datatype` instead. Please refer to
`Command Line Reference - Widget Commands`.

20.4 Card Components

Beginning in Arches 4.3, Cards are rendered using Card Components, allowing them to be composed and nested
arbitrarily in various contexts within the Arches UI. Arches comes with a default Card Component that should suit
most needs, but you can also create and register custom Card Components to extend the front-end behavior of Arches.

Developing Card Components is very similar to developing Widgets. A Card Component consists of a Django tem-
plate and Knockout.js JavaScript file. To register your component, you’ll also need a JSON file specifying its initial
configuration.

To develop your new card, you’ll place files like so in your project:

```plaintext
project_name/templates/views/components/cards/my-new-card.htm
project_name/media/js/views/components/cards/my-new-card.js
```

To register and configure the Component, you’ll need a JSON configuration file:

```plaintext
project_name/cards/my-new-card.json
```

20.4.1 Creating a Card Component

The default template and Knockout files illustrate everything a Card Component needs, and you’ll be extending
this functionality. Your template will provide conditional markup for various contexts (‘editor-tree’, ‘designer-tree’,
‘permissions-tree’, ‘form’, and ‘report’), render all the card’s Widgets, and display other information.

Here’s the template for the default Card Component:

```html
{% load i18n %}
<!-- ko foreach: { data: $data, as: 'self' } -->
<!-- ko if: state === 'editor-tree' -->
<li role="treeitem card-treeitem" class="jstree-node" data-bind="css: { 'jstree-open': card.tiles().length > 0 && card.expanded(), 'jstree-closed': card.tiles().length > 0 && !card.expanded(), 'jstree-leaf': card.tiles().length === 0 }, scrollTo: card.scrollTo, container: '.resource-editor-tree'">
  <i class="jstree-icon jstree-ocl" role="presentation" data-bind="click: function() { card.expanded(!card.expanded()) }"></i>
  <a class="jstree-icon jstree-ocl" role="presentation" data-bind="click: function() { card.canAdd() ? card.selected(true) : card.tiles()[0].selected(true) }"></a>
</li>
<!-- ko foreach: { data: $data, as: 'self' } -->
<!-- ko if: state === 'editor-tree' -->
```

(continues on next page)
20.4. Card Components
| 124 Chapter 20. Creating New Functions, Widgets, Card Components, and DataTypes |

(continued from previous page)

```html
<!-- ko component: {
  name: self.form.cardComponentLookup[card.model.component_id()].componentname,
  params: {
    state: 'editor-tree',
    card: card,
    tile: null,
    loading: self.loading,
    form: self.form
  }
} -->

```
Chapter 20. Creating New Functions, Widgets, Card Components, and DataTypes
<div class="new-provisional-edit-card-container">
  <!-- ko if: reviewer && provisionalTileViewModel.provisionaledits().length > 0 && !provisionalTileViewModel.tileIsFullyProvisional() -->
  <!--ko if: !provisionalTileViewModel.selectedProvisionalEdit()-->
  <div class='new-provisional-edits-list'>
    <div class='new-provisional-edits-header'>
      <div class='new-provisional-edits-title'>
        {% trans 'Provisional Edits' %}
      </div>
      <div class='btn btn-shim btn-danger btn-labeled btn-xs fa fa-trash new-provisional-edits-delete-all' style="padding: 3px;" data-bind="click: function(){provisionalTileViewModel.deleteAllProvisionalEdits()}">
          {% trans 'Delete all edits' %}
      </div>
    </div>
    <!-- /ko -->
  </div>
  <!-- /ko -->
  <!--ko foreach: { data: provisionalTileViewModel.provisionaledits(), as: 'pe' } -->
    <div class='new-provisional-edit-entry' data-bind="css: {'selected': pe === $parent.provisionalTileViewModel.selectedProvisionalEdit()}, click: function(){ $parent.provisionalTileViewModel.rejectProvisionalEdit(pe)}">
      <div class='title'>
        <div class='field'>
          <div class='field'>
            <a href='' class='field fa fa-times-circle new-delete-provisional-edit' data-bind="click : function(){$parent.provisionalTileViewModel.rejectProvisionalEdit(pe)}"></a>
          </div>
        </div>
        <div class='field timestamp'>
          <span data-bind="text : pe.displaydate">@</span> <span data-bind="text : pe.displaytimestamp"></span>
        </div>
      </div>
    </div>
  <!-- /ko -->
</div>
(continues on next page)
<h4 data-bind="text: card.model.name"></h4><h5 data-bind="text: card.model.instructions"></h5>

<-- ko if: card.widgets().length > 0 -->
<form class="widgets" style="margin-bottom: 20px;">
  <div data-bind="foreach: {data: card.widgets, as: 'widget'}">
    <div data-bind="component: {
      name: self.form.widgetLookup[widget.widget_id()].name,
      params: {
        formData: self.tile.formData,
        tile: self.tile,
        form: self.form,
        config: widget.configJSON,
        label: widget.label(),
        value: self.tile.data[widget.node_id()],
        node: self.form.nodeLookup[widget.node_id()],
        expanded: self.expanded,
        graph: self.form.graph,
        type: "resource-editor"
      },
      css: { "active": widget.selected, "hover": widget.hovered,
        "widget-preview": self.preview
      },
      click: function(data, e) { if (!widget.selected() && self.preview)
        widget.selected(true); },
      event: { mouseover: function(){ if (self.preview) widget.hovered(true)
        }, mouseout: function(){ if (self.preview) widget.hovered(null) } } }">
  </div>
</form>

<!-- /ko -->

<!-- ko if: card.widgets().length === 0 -->
<ul class="card-summary-section" data-bind="css: {disabled: !tile.tileid}">
  <!-- ko foreach: { data: tile.cards, as: 'card' } -->
  <li class="card-summary">
    <a href="javascript:void(0)" data-bind="click: function () {
      if (card.parent.tileid) {
        card.canAdd() ? card.selected(true) : card.tiles()[0].
        selected(true);
      } }">
      <!-- ko if: card.widgets().length > 0 -->
      (continues on next page)
      </a>
    </li>
  </ul>
</li>
</ul>

</div>
</form>

</li>
</ul>

(continues on next page)
<span data-bind="text: card.widgets()[0].label || card.model.name" class="tile-summary-label"></span>

<div style="display: inline;" data-bind="component: {
  name: self.form.widgetLookup[card.widgets()[0].widget_id()].name,
  params: {
    tile: tile,
    node: self.form.nodeLookup[card.widgets()[0].node_id()],
    config: self.form.widgetLookup[card.widgets()[0].widget_id()].config,
    label: self.form.widgetLookup[card.widgets()[0].widget_id()].label,
    value: tile.data[card.widgets()[0].node_id()]
  },
  type: 'resource-editor',
  state: 'display_value'
}"
><div style="display: inline;" data-bind="component: {
  name: self.form.widgetLookup[card.widgets()[0].widget_id()].name,
  params: {
    tile: tile,
    node: self.form.nodeLookup[card.widgets()[0].node_id()],
    config: self.form.widgetLookup[card.widgets()[0].widget_id()].config,
    label: self.form.widgetLookup[card.widgets()[0].widget_id()].label,
    value: tile.data[card.widgets()[0].node_id()]
  },
  type: 'resource-editor',
  state: 'display_value'
}"
></div>
</div>

<!-- /ko -->

<!-- ko if: card.widgets().length === 0 -->

<span data-bind="text: card.model.name"></span>

<!-- /ko -->
</a>
</li>
</ul>

<!-- /ko -->
</ul>
</li>
</ul>

<!-- /ko -->
<div class="install-buttons">

<!-- ko if: tile.tileid -->

<button class="btn btn-shim btn-warning btn-labeled btn-lg fa fa-trash" data-bind="click: function () { self.form.deleteTile(tile); }">{% trans 'Delete this record' %}</button>

<!-- /ko -->

<!-- ko if: tile.dirty() -->

<!-- ko if: provisionalTileViewModel && !provisionalTileViewModel.tileIsFullyProvisional() -->

<button class="btn btn-shim btn-danger btn-labeled btn-lg fa fa-times" data-bind="click: tile.reset">{% trans 'Cancel edit' %}</button>

<!-- /ko -->

<!-- ko if: tile.tileid -->

<button class="btn btn-shim btn-success btn-labeled btn-lg fa fa-plus" data-bind="click: function () { self.form.saveTile(tile); }">{% trans 'Save edit' %}</button>

<!-- /ko -->

<!-- /ko -->

<!-- ko if: !tile.tileid -->

<button class="btn btn-shim btn-success btn-labeled btn-lg fa fa-plus" data-bind="click: function () { self.form.saveTile(tile); }">{% trans 'Add' %}</button>

<!-- /ko -->
</div>
</div>
</div>

20.4. Card Components 129
<!-- ko if: state === 'report' -->
<div class="rp-card-section">
  <span class="rp-tile-title" data-bind="text: card.model.get('name')"></span>
  <!-- ko foreach: { data: card.tiles, as: 'tile' } -->
    <div class="rp-card-section">
      <!-- ko if: card.model.get('widgets')().length > 0 -->
        <div class="rp-report-tile" data-bind="attr: { id: tile.tileid }">
          <dl class="dl-horizontal">
            <!-- ko foreach: { data: card.model.get('widgets'), as: 'widget' } -->
              <!-- ko component: { name: widget.widgetLookup[widget.get("widget_id")()], params: { config: configJSON, label: widget.get("label"()), node: widget.node, value: tile.data[widget.node.nodeid], state: "report" } } -->
            <!-- /ko -->
          </dl>
        </div>
      <!-- /ko -->
    </div>
  <!-- /ko -->
</div>
</div>  
<!-- ko if: card.tiles().length === 0 -->
<div class="row rp-report-tile rp-no-data">
  <!-- ko ifnot: card.model.get('cardid') -->
    {%
      trans "Sorry, you don't have access to this information" %}
  <!-- /ko -->
  <!-- ko if: card.model.get('cardid') -->
    {%
      trans "No data added yet for" %} "<span data-bind="text: card.model.get('name')"></span>"
  <!-- /ko -->
</div>
<!-- /ko -->
</div>
<!-- /ko -->

(continues on next page)
And here's the Knockout file:

```javascript
define(['ko', 'bindings/scrollTo'], function(ko) {
  var viewModel = function(params) {
    this.state = params.state || 'form';
    this.preview = params.preview;
    this.loading = params.loading || ko.observable(false);
    this.card = params.card;
    this.tile = params.tile;
    if (this.preview) {
      if (!this.card.newTile) {
        this.card.newTile = this.card.getNewTile();
      }
      this.tile = this.card.newTile;
    }
    this.form = params.form;
    this.provisionalTileViewModel = params.provisionalTileViewModel;
    this.reviewer = params.reviewer;
    this.expanded = ko.observable(true);
    this.beforeMove = function(e) {
      e.cancelDrop = (e.sourceParent !== e.targetParent);
    }
  };
  return ko.components.register('default-card', {
    viewModel: viewModel,
    template: {
      require: 'text!templates/views/components/cards/default.htm'
    }
  });
});
```

### 20.4.2 Registering your Card Component

To register your Component, you’ll need a JSON configuration file looking a lot like this sample:

```json
{
  "name": "My New Card",
  "componentid": "ee17d6c-0c32-4536-8a01-392df734de1c",
  "component": "/views/components/cards/my-new-card",
  "componentname": "my-new-card",
  "description": "An awesome new card that does wonderful things."
}
```

- **componentid** Optional A UUID4 for your Component. Feel free to generate one in advance if that fits your workflow; if not, Arches will generate one for you and print it to STDOUT when you register the Component.

- **name** Required The name of your new Card Component, visible in the drop-down list of card components in the Arches Designer.

- **description** Required A brief description of your component.
component Required The path to the component view you have developed. Example: views/components/cards/sample-datatype

componentname Required Set this to the last part of component above.

defaultconfig Required You can provide user-defined default configuration here. Make it a JSON dictionary of keys and values. An empty dictionary is acceptable.

Commands

To register your Card Component, use this command:

```
python manage.py card_component register --source /Documents/projects/mynewproject/\n  →mynewproject/cards/new-card-component.json
```

The command will confirm your Component has been registered, and you can also see it with:

```
python manage.py card_component list
```

If you make an update to your Card Component, you can load the changes to Arches with:

```
python manage.py card_component update --source /Documents/projects/mynewproject/\n  →mynewproject/cards/new-card-component.json
```

All the Card Component commands are detailed in Command Line Reference - Card Component Commands.
Creating New Map Layers

A developer can add new layers to the map by registering them through the command line interface.

New map layers can come from many different geospatial sources – from shapefiles to GeoTIFFs to external Web Map Services to reconfigurations of the actual resource data stored within Arches.

New map layers can be created with two general definitions, as MapBox layers or tileserver layers, each with its own wide range of options.

For working examples, please see our arches4-geo-examples repo.

**Note:** By default, new map layers are designated as Overlays. To designate the layer as a Basemap, just add `-b` to the load commands shown below.

### 21.1 MapBox Layers

```python
python manage.py packages -o add_mapbox_layer -j /path/to/mapbox_style.json -n "New MapBox Layer"
```

Arches allows you to make direct references to styles or layers that have been previously defined in MapBox Studio. You can make entirely new basemap renderings, save them in your MapBox account, then download the style definition and use it here. Read more about MapBox Styles.

Additionally, you can take a MapBox JSON file and place any mapbox.js layer definition in the `layers` section, as long as you define its source in the `sources` section.

**Note:** One thing to be aware of when trying to cascade a WMS through a MapBox layer is that mapbox.js is much pickier about CORS than other js mapping libraries like Leaflet. To use an external WMS or tileset, you may be better off using a tileserver layer as described below. You can find WMS examples in the arches4-geo-examples repo.
21.2 Tileserver Vector Layers

To add a new tileserver layer, you need a .json file that contains a TileStache-compliant layer definition. Within this file, you can use any of the many different data provider classes from Tilestache. The .json file that you load into Arches for a tileserver layer should have three sections:

```json
{
    "type": "vector",  ## This value should be "raster" or "vector".
    "layers": [        ## This is a mapbox.js layer definition which defines the style
        ## of the layer and links the source name with the layer name.
    {
        "id": "rivers",  ## This is the tileserver configuration that will be used by
        "type": "line",   ## TileStache. Refer to TileStache docs and place the entire
        "source": "rivers",  ## "provider" section into this "config" section.
        "source-layer": "rivers",
        "layout": {
            "visibility": "visible"
        },
        "paint": {
            "line-width": 2,
            "line-color": "rgb(37, 58, 241)"
        }
    }
    ],
    "config": {
        "provider": {
            "class": "TileStache.Goodies.VecTiles:Provider",
            "kwargs": {
                "dbinfo": {
                    "host": "localhost",
                    "user": "postgres",
                    "password": "postgis",
                    "database": "arches",
                    "port": "5432"
                },
                "simplify": 0.5,
                "queries": [
                    "select gid as __id__, name, st_asgeojson(geom) as geojson, st___transform(geom, 900913) as __geometry__ from rivers"
                ]
            }
        }
    }
}
```

Here’s a full example of a tilestache file that makes a layer from data in PostGIS (a table called “rivers”):

```json
{
    "type": "vector",
    "layers": [
        {
            "id": "rivers",
            "type": "line",
            "source": "rivers",
            "source-layer": "rivers",
            "layout": {
                "visibility": "visible"
            },
            "paint": {
                "line-width": 2,
                "line-color": "rgb(37, 58, 241)"
            }
        }
    ],
    "config": {
        "provider": {
            "class": "TileStache.Goodies.VecTiles:Provider",
            "kwargs": {
                "dbinfo": {
                    "host": "localhost",
                    "user": "postgres",
                    "password": "postgis",
                    "database": "arches",
                    "port": "5432"
                },
                "simplify": 0.5,
                "queries": [
                    "select gid as __id__, name, st_asgeojson(geom) as geojson, st___transform(geom, 900913) as __geometry__ from rivers"
                ]
            }
        }
    }
}
```
21.2.1 Making Selectable Vector Layers

In Arches, it’s possible to add a vector layer whose features may be “selectable”. This is especially useful during drawing operations. For example, a building footprint dataset could be added as a selectable vector layer, and while creating new building resources you would select and “transfer” these geometries from the overlay to the new Arches resource.

1. First, the data source for the layer may be geojson or vector tiles. This could be a tile server layer serving vector features from PostGIS, for example.

2. Add a property to your vector features called “geojson”.

3. Populate this property with either the entire geojson geometry for the feature, or a url that will return a json response containing the entire geojson geometry for the feature. This is necessary to handle the fact that certain geometries may extend across multiple vector tiles.

4. Add the overlay as you would any tileserver layer (see above).

You will now be able to add this layer to the map and select its features by clicking on them.

21.2.2 Adding Click and Hover Styles

In addition to making overlay features selectable, you can define styles for their hover and click states.

1. To do so, each feature in your overlay needs a unique _featureid. If you’re overlay served from PostGIS, you can define this property in the layer config’s queries array like so:

   ```json
   "queries": [
      "select gid as __id__, gid as _featureid, site_name, feature_info_content, st_asgeojson(geom) as geojson, st_transform(geom, 900913) as __geometry__ from example_layer"
   ]
   ```

2. Next you will need to ensure your source-layer is properly defined. In the source layer the source-layer property must match the id property and cannot contain spaces or periods. This layer will be hidden when the hover or click layer is revealed, so this should be a fill layer if your click or hover layers contain a fill.

3. Define the hover and click layers. These each must have a _featureid filter their ids must be suffixed with either a -click or -hover. For example:

   ```json
   {
      "layout": {
         "visibility": "visible"
      },
      "source": "example_layer",
      "filter": [
         "all",
         [
            "==",
            "__id__",
            "__featureid"
         ]
      ]
   }
   ```

(continues on next page)
4. If you are loading your layers from a package, each layer must have an accompanying `meta.json` file with a name defined. This will ensure that the `source-layer` property is saved to the layer as you intend. If you do not have a `meta.json` file, the source-layer name will be the map layer’s file name, and will probably not work properly. See the example package for an example:

https://github.com/archesproject/arches4-example-pkg/tree/master/map_layers/tile_server/overlays/vector_example

### 21.2.3 Customizing Map Popup Content

You can display custom HTML in the search map popup when a user hovers or clicks on a feature in a vector layer.

1. First, the data source for the layer may be geojson or vector tiles. This could be a tile server layer serving vector features from PostGIS, for example.

2. Add a property to your vector features called “feature_info_content”.

3. Populate this property with either an html element or a url from which to load html. If you use a url, you will need to update the ‘ALLOWED_POPUP_HOSTS’ to include the host from which you want to request HTML.

4. Add the overlay as you would any tileserver layer (see above).

You will now be able to add this layer to the map see the markup defined in the ‘feature_info_content’ in the search map popup.

### 21.3 Tileserver Mapnik Layers

```bash
code
python manage.py packages -o add_tileserver_layer -m /path/to/mapnik_config.xml -n "New Mapnik Tileserver Layer"
```

Mapnik is the provider that TileStache uses to serve rasters, and is very commonly used in Arches. Arches allows you to upload a Mapnik XML file to configure a new tileserver layer, instead of creating the full JSON file. This is the easiest way to make layers from GeoTiffs and shapefiles. A basic example of a Mapnik XML file is shown below (it points to a geotiff named hillshade.tif). For more about creating these XML files, see the [Mapnik XML reference](https://mapnik.org/documentation/).
<Map background-color="transparent">
  <Layer name="Hillshade">
    <StyleName>raster</StyleName>
    <Datasource>
      <Parameter name="type">gdal</Parameter>
      <Parameter name="file">hillshade.tif</Parameter>
      <Parameter name="nodata">0</Parameter>
    </Datasource>
  </Layer>
  <Style name="raster">
    <Rule name="rule 1">
      <RasterSymbolizer opacity=".7" scaling="bilinear" mode="normal" />
    </Rule>
  </Style>
</Map>
22.1 Resource Model Overview

Resources in an Arches database are separated into distinct Resource Models designed to represent a kind of physical real-world resource, such as a historic artifact or event. In the technical sense, the term Resource Model refers collectively to the following user-facing elements in Arches:

1. A Graph data structure representing a physical real-world resource, such as a building, a public figure, a website, an archaeological site, or a historic document.

2. A set of Cards to collect and display data associated with instances of this Resource Model.

The relationships among these components and their dependencies are visualized below:
The Arches logical model has been developed to support this modular construction, and the relevant models are described below as they pertain to the graph, UI components, and the resource data itself (not illustrated above).

Note: In the UI you will see a distinction between “Resource Models” and “Branches”, but underneath these are both made from instances of the Graph model. The primary difference between the two is the isresource property, which is set to True for a Resource Model.

Branches are used for records that might appear in multiple Resource Models, such as a person or place. Branches can be included as children of any Ontology-permitted Node in a Resource Model.

22.2 Controllers

Arches platform code defines base classes for some of its core data models, and uses proxy models to implement their controllers. In smaller classes, “controller” code is included with the data model class. This documentation primarily discusses the models, but controller behavior is discussed where relevant to how the models are used, and all models are referred to by their more succinct “controller” name.
22.3 Graph Definition

A Graph is a collection of NodeGroups, Nodes, and Edges which connect the Nodes.

Note: This definition does not include UI models and attributes, which are discussed below.

In the Arches data model, Nodes represent their graph data structure namesakes, sometimes called vertices. A Node does the work of defining the Graph data structure in conjunction with one or more Edges, and sometimes collecting data.

NodeGroups are an Arches feature used to represent a group of one or more Nodes that collect data. NodeGroups can be nested, creating a metadata structure which is used to display the graph in the UI and collect related information together.

A NodeGroup exists for every Node that collects data, and both contains and shares its UUID with that node. NodeGroups with more than one member Node are used to collect composite or semantically-related information. For example, a NodeGroup for a Node named Name.E1 may contain a Name Type.E55 Node. This way, a Graph with this NodeGroup may store Names with multiple “types”, always collecting the information together.

NodeGroups are used to create Cards, and this is done based on the cardinality property. Therefore, not every NodeGroup will be used to create a Card, which allows NodeGroups to exist within other NodeGroups. The parentnodegroup property is used to record this nesting.

A user-defined Function may be registered and then associated with a Graph in order to extend the behavior of Arches. For more information, see here.

22.3.1 GraphModel

```python
class GraphModel(models.Model):
    graphid = models.UUIDField(primary_key=True, default=uuid.uuid1)
    name = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    deploymentfile = models.TextField(blank=True, null=True)
    author = models.TextField(blank=True, null=True)
    deploymentdate = models.DateTimeField(blank=True, null=True)
    version = models.TextField(blank=True, null=True)
    isresource = models.BooleanField()
    isactive = models.BooleanField()
    iconclass = models.TextField(blank=True, null=True)
    color = models.TextField(blank=True, null=True)
```

(continues on next page)
subtitle = models.TextField(blank=True, null=True)
ontology = models.ForeignKey('Ontology', db_column='ontologyid', related_name='graphs', null=True, blank=True)
functions = models.ManyToManyField(to='Function', through='FunctionXGraph')
jsonldcontext = models.TextField(blank=True, null=True)
template = models.ForeignKey('ReportTemplate', db_column='templateid', null=True, blank=True)
    default='50000000-0000-0000-0000-000000000001'
config = JSONField(db_column='config', default={})

@property
def disable_instance_creation(self):
    if not self.isresource:
        return _('Only resource models may be edited - branches are not editable')
    if not self.isactive:
        return _('Set resource model status to Active in Graph Designer')
    return False

def is_editable(self):
    result = True
    if self.isresource:
        resource_instances = ResourceInstance.objects.filter(graph_id=self.graphid).count()
        result = False if resource_instances > 0 else True
    if settings.OVERRIDE_RESOURCE_MODEL_LOCK == True:
        result = True
    return result

class Meta:
    managed = True
db_table = 'graphs'

22.3.2 Node

class Node(models.Model):
    """
    Name is unique across all resources because it ties a node to values within tiles.
    Recommend prepending resource class to node name.
    """
    nodeid = models.UUIDField(primary_key=True, default=uuid.uuid1)
    name = models.TextField()
    istopnode = models.BooleanField()
    ontologyclass = models.TextField(blank=True, null=True)
    datatype = models.TextField()
    nodelocation = models.ForeignKey(NodeGroup, db_column='nodegroupid', blank=True, null=True)
    graph = models.ForeignKey(GraphModel, db_column='graphid', blank=True, null=True)
    config = JSONField(blank=True, null=True, db_column='config')
    issearchable = models.BooleanField(default=True)
isrequired = models.BooleanField(default=False)
sortorder = models.IntegerField(blank=True, null=True, default=0)

def get_child_nodes_and_edges(self):
    """
    gather up the child nodes and edges of this node
    returns a tuple of nodes and edges
    """
    nodes = []
    edges = []
    for edge in Edge.objects.filter(domainnode=self):
        nodes.append(edge.rangenode)
        edges.append(edge)
        child_nodes, child_edges = edge.rangenode.get_child_nodes_and_edges()
        nodes.extend(child_nodes)
        edges.extend(child_edges)
    return (nodes, edges)

@property
    def is_collector(self):
        return str(self.nodeid) == str(self.nodegroup_id) and self.nodegroup is not None

def get_relatable_resources(self):
    relatable_resource_ids = [
        r2r.resourceclassfrom for r2r in Resource2ResourceConstraint.objects.filter(
            resourceclassto_id=self.nodeid)
    ]
    relatable_resource_ids = relatable_resource_ids + \
    [r2r.resourceclassto for r2r in Resource2ResourceConstraint.objects.filter(
        resourceclassfrom_id=self.nodeid)
    ]
    return relatable_resource_ids

def set_relatable_resources(self, new_ids):
    old_ids = [res.nodeid for res in self.get_relatable_resources()]
    for old_id in old_ids:
        if old_id not in new_ids:
            Resource2ResourceConstraint.objects.filter(Q(resourceclassto_id=self.nodeid) | Q(
                resourceclassfrom_id=self.nodeid), Q(resourceclassto_id=old_id) | Q(resourceclassfrom_id=old_id)).delete()
    for new_id in new_ids:
        if new_id not in old_ids:
            new_r2r = Resource2ResourceConstraint.objects.create(
                resourceclassfrom_id=self.nodeid, resourceclassto_id=new_id)
            new_r2r.save()

class Meta:
    managed = True
    db_table = 'nodes'
22.3.3 NodeGroup

class NodeGroup(models.Model):
    nodegroupid = models.UUIDField(primary_key=True, default=uuid.uuid1)
    legacygroupid = models.TextField(blank=True, null=True)
    cardinality = models.TextField(blank=True, default='1')
    parentnodegroup = models.ForeignKey('self', db_column='parentnodegroupid',
                                        blank=True, null=True)  # Allows nodegroups within nodegroups

    class Meta:
        managed = True
        db_table = 'node_groups'
        default_permissions = ()
        permissions = (
            ('read_nodegroup', 'Read'),
            ('write_nodegroup', 'Create/Update'),
            ('delete_nodegroup', 'Delete'),
            ('no_access_to_nodegroup', 'No Access'),
        )

22.3.4 Edge

class Edge(models.Model):
    edgeid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field
    # type is a guess.
    name = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    ontologyproperty = models.TextField(blank=True, null=True)
    domainnode = models.ForeignKey('Node', db_column='domainnodeid', related_name='edge_domains')
    rangenode = models.ForeignKey('Node', db_column='rangenodeid', related_name='edge_ranges')
    graph = models.ForeignKey('GraphModel', db_column='graphid', blank=True, null=True)

    class Meta:
        managed = True
        db_table = 'edges'
        unique_together = (('rangenode', 'domainnode'),)

22.3.5 Function

class Function(models.Model):
    functionid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field
    # type is a guess.
    name = models.TextField(blank=True, null=True)
    functiontype = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    defaultconfig = JSONField(blank=True, null=True)
    modulename = models.TextField(blank=True, null=True)
    classname = models.TextField(blank=True, null=True)

(continues on next page)
22.4 Ontologies

An ontology standardizes a set of valid CRM (Conceptual Reference Model) classes for Node instances, as well as a set of relationships that will define Edge instances. Most importantly, an ontology enforces which Edges can be used to connect which Nodes. If a pre-loaded ontology is designated for a Graph instance, every NodeGroup within that Graph must conform to that ontology. You may also create an “ontology-less” graph, which will not define specific CRM classes for the Nodes and Edges.

These rules are stored as OntologyClass instances, which are stored as JSON. These JSON objects consist of dictionaries with two properties, down and up, each of which contains another two properties ontology_property and ontology_classes (down assumes a known domain class, while up assumes a known range class).

```json
{
    "down": {
        "ontology_property": "P1_is_identified_by",
        "ontology_classes": [
            "E51_Contact_Point",
            "E75_Conceptual_Object_Appellation",
            "E42_Identifier",
            "E45_Address",
            "E41_Appellation"
        ]
    }
}
```
Aches comes preloaded with the CIDOC CRM, an ontology created by ICOM (International Council of Museums) to model cultural heritage documentation. However, a developer may create and load an entirely new ontology.

### 22.4.1 Ontology

```python
class Ontology(models.Model):
    ontologyid = models.UUIDField(default=uuid.uuid1, primary_key=True)
    name = models.TextField()
    version = models.TextField()
    path = models.FileField(storage=get_ontology_storage_system())
    parentontology = models.ForeignKey('Ontology', db_column='parentontologyid',
                                       related_name='extensions', null=True,
                                       blank=True)

class Meta:
    managed = True
db_table = 'ontologies'
```

### 22.4.2 OntologyClass

```python
class OntologyClass(models.Model):
    
    the target JSONField has this schema:

    values are dictionaries with 2 properties, 'down' and 'up' and within each of
    those another 2 properties, 'ontology_property' and 'ontology_classes'

    "down" assumes a known domain class, while "up" assumes a known range class

    .. code-block:: python

        "down": {
            "ontology_property": "P1_identifies",
            "ontology_classes": [
                "E51_CONTACT_POINT",
                "E75_CONCEPTUAL_OBJECT_APPELLATION",
                "E42_IDENTIFIER",
            ]
        }
```

(continues on next page)
22.5 RDM Models

The RDM (Reference Data Manager) stores all of the vocabularies used in your Arches installation. Whether they are simple wordlists or a polyhierarchical thesauri, these vocabularies are stored as “concept schemes” and can be viewed as an aggregation of one or more concepts and the semantic relationships (links) between those concepts.

In the data model, a concept scheme consists of a set of Concept instances, each paired with a Value. In our running name/name_type example, the Name Type.E55 Node would be linked to a Concept (Name Type.E55) which would have two child Concepts. Thus, where the user sees a dropdown containing “Primary” and “Alternate”, these are actually the Values of Name Type.E55’s two descendent Concepts. The parent/child relationships between Concepts are stored as Relation instances.

22.5.1 Concept

class Concept(models.Model):
    conceptid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field
    # type is a guess.
    nodetype = models.ForeignKey('DNodeType', db_column='nodetype')
    legacyoid = models.TextField(unique=True)

class Meta:
    managed = True
    db_table = 'concepts'
22.5.2 Relation

```python
class Relation(models.Model):
    conceptfrom = models.ForeignKey(Concept, db_column='conceptidfrom', related_name='relation_concepts_from')
    conceptto = models.ForeignKey(Concept, db_column='conceptidto', related_name='relation_concepts_to')
    relationtype = models.ForeignKey(DRelationType, db_column='relationtype')
    relationid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.

class Meta:
    managed = True
    db_table = 'relations'
    unique_together = (('conceptfrom', 'conceptto', 'relationtype'),)
```

22.5.3 Value

```python
class Value(models.Model):
    valueid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
    concept = models.ForeignKey('Concept', db_column='conceptid')
    valuetype = models.ForeignKey(DValueType, db_column='valuetype')
    value = models.TextField()
    language = models.ForeignKey(DLanguage, db_column='languageid', blank=True, null=True)

class Meta:
    managed = True
    db_table = 'values'
```

22.6 Resource Data

Three models are used to store Arches business data:

- **ResourceInstance** - one per resource in the database
- **Tile** - stores all business data
- **ResourceXResource** - records relationships between resource instances

Creating a new resource in the database instantiates a new ResourceInstance, which belongs to one resource model and has a unique resourceinstanceid. A resource instance may also have its own security/permissions properties in order to allow a fine-grained level of user-based permissions.

Once data have been captured, they are stored as Tiles in the database. Each Tile stores one instance of all of the attributes of a given NodeGroup for a resource instance, as referenced by the resourceinstanceid. This business data is stored as a JSON object, which is a dictionary with n number of keys/value pairs that represent a Node’s id nodeid and that Node’s value.

in theory:

```
{
    "nodeid": "node value",
    ...
}
```
in practice:

```json
{
    "20000000-0000-0000-0000-000000000002": "John",
    "20000000-0000-0000-0000-000000000004": "Primary"
}
```

(In keeping with our running example, the keys in the second example would refer to a Name.E1 node and an Name Type.E55 node, respectively.)

Arches also allows for the creation of relationships between resource instances, and these are stored as instances of the `ResourceXResource` model. The `resourceinstanceidfrom` and `resourceinstanceidto` fields create the relationship, and `relationshiptype` qualifies the relationship. The latter must correspond to the appropriate top node in the RDM. This constrains the list of available types of relationships available between resource instances.

### 22.6.1 ResourceInstance

```python
class ResourceInstance(models.Model):
    resourceinstanceid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
    graph = models.ForeignKey(GraphModel, db_column='graphid')
    legacyid = models.TextField(blank=True, unique=True, null=True)
    createdtime = models.DateTimeField(auto_now_add=True)

    class Meta:
        managed = True
        db_table = 'resource_instances'
```

### 22.6.2 TileModel

```python
class TileModel(models.Model):  # Tile
    ""
    the data JSONField has this schema:
    ""
    values are dictionaries with n number of keys that represent nodeid's and values
    the value of that node instance
    ""
    .. code-block:: python

    
    {
        nodeid: node value,
        nodeid: node value,
        ...
    }
    
    {
        "20000000-0000-0000-0000-000000000002": "John",
        "20000000-0000-0000-0000-000000000003": "Smith",
        "20000000-0000-0000-0000-000000000004": "Primary"
    }
```

(continues on next page)
the provisionaledits JSONField has this schema:

values are dictionaries with \( n \) number of keys that represent nodeid's and values→ the value of that node instance

.. code-block:: python

```python
{
  userid: {
    value: node value,
    status: "review", "approved", or "rejected"
    action: "create", "update", or "delete"
    reviewer: reviewer's user id,
    timestamp: time of last provisional change,
    reviewtimestamp: time of review
  }
}
```

```python
{
  1: {
    "value": {
      "20000000-0000-0000-0000-000000000002": "Jack",
      "20000000-0000-0000-0000-000000000003": "Smith",
      "20000000-0000-0000-0000-000000000004": "Primary"
    },
    "status": "rejected",
    "action": "update",
    "reviewer": 8,
    "timestamp": "20180101T1500",
    "reviewtimestamp": "20180102T0800"
  },
  15: {
    "value": {
      "20000000-0000-0000-0000-000000000002": "John",
      "20000000-0000-0000-0000-000000000003": "Smith",
      "20000000-0000-0000-0000-000000000004": "Secondary"
    },
    "status": "review",
    "action": "update"
  }
}
```

```python
tileid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
resourceinstance = models.ForeignKey(ResourceInstance, db_column='resourceinstanceid')
parenttile = models.ForeignKey('self', db_column='parenttileid', blank=True, null=True)
data = JSONField(blank=True, null=True, db_column='tiledata')  # This field type is a guess.
nodegroup = models.ForeignKey(NodeGroup, db_column='nodegroupid')
sortorder = models.IntegerField(blank=True, null=True, default=0)
provisionaledits = JSONField(blank=True, null=True, db_column='provisionaledits')  # This field type is a guess.
```
class Meta:
    managed = True
db_table = 'tiles'

def save(self, *args, **kwargs):
    if (self.sortorder is None or (self.provisionaledits is not None and self.data == {})):
        sortorder_max = TileModel.objects.filter(
            nodegroup_id=self.nodegroup_id, resourceinstance_id=self.resourceinstance_id).
        resourceinstance_id.aggregate(Max('sortorder'))['sortorder__max']
        self.sortorder = sortorder_max + 1 if sortorder_max is not None else 0
    super(TileModel, self).save(*args, **kwargs)  # Call the "real" save() method.

22.6.3 ResourceXResource

class ResourceXResource(models.Model):
    resourcexid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
    resourceinstancefrom = models.ForeignKey('ResourceInstance', db_column='resourceinstancefrom', blank=True, null=True,
                                              related_name='resxres_resource_instance_ids_from')
    resourceinstanceidto = models.ForeignKey('ResourceInstance', db_column='resourceinstanceidto', blank=True, null=True,
                                              related_name='resxres_resource_instance_ids_to')
    notes = models.TextField(blank=True, null=True)
    relationshiptype = models.TextField(blank=True, null=True)
    datestarted = models.DateField(blank=True, null=True)
    dateended = models.DateField(blank=True, null=True)
    created = models.DateTimeField()
    modified = models.DateTimeField()

def delete(self):
    from arches.app.search.search_engine_factory import SearchEngineFactory
    se = SearchEngineFactory().create()
    se.delete(index='resource_relations', doc_type='all', id=self.resourcexid)
    super(ResourceXResource, self).delete()

def save(self):
    from arches.app.search.search_engine_factory import SearchEngineFactory
    se = SearchEngineFactory()
    if not self.created:
        self.created = datetime.datetime.now()
    self.modified = datetime.datetime.now()
    document = model_to_dict(self)
    se.index_data(index='resource_relations', doc_type='all', idfield='resourcexid')
    super(ResourceXResource, self).save()}
22.6.4 Edit Log

A change in a Tile’s contents, which is the result of any resource edits, is recorded as an instance of the EditLog model.

```python
class EditLog(models.Model):
    editlogid = models.UUIDField(primary_key=True, default=uuid.uuid1)
    resourcedisplayname = models.TextField(blank=True, null=True)
    resourceclassid = models.TextField(blank=True, null=True)
    resourceinstanceid = models.TextField(blank=True, null=True)
    nodegroupid = models.TextField(blank=True, null=True)
    tileinstanceid = models.TextField(blank=True, null=True)
    edittype = models.TextField(blank=True, null=True)
    newvalue = JSONField(blank=True, null=True, db_column='newvalue')
    oldvalue = JSONField(blank=True, null=True, db_column='oldvalue')
    newprovisionalvalue = JSONField(blank=True, null=True, db_column='newprovisionalvalue')
    oldprovisionalvalue = JSONField(blank=True, null=True, db_column='oldprovisionalvalue')
    timestamp = models.DateTimeField(blank=True, null=True)
    userid = models.TextField(blank=True, null=True)
    user_firstname = models.TextField(blank=True, null=True)
    user_lastname = models.TextField(blank=True, null=True)
    user_email = models.TextField(blank=True, null=True)
    user_username = models.TextField(blank=True, null=True)
    provisional_userid = models.TextField(blank=True, null=True)
    provisional_user_username = models.TextField(blank=True, null=True)
    provisional_edittype = models.TextField(blank=True, null=True)
    note = models.TextField(blank=True, null=True)

class Meta:
    managed = True
    db_table = 'edit_log'
```

22.7 UI Component Models

A number of models exist specifically to support the resource model UI. The purpose of this is to create direct relationships between the resource graph and the data entry cards that are used to create resource instances. Generally, the process works like this:

1. A resource graph is an organized collection of NodeGroups which define what information will be gathered for a given resource model.

2. A resource’s Cards and are tied to specific NodeGroups and define which input Widgets will be used to gather values for each Node in that NodeGroup. Card Components are used to render the cards in various contexts in the Arches UI.
Cards are UI representations of a NodeGroup, and they encapsulate the Widgets that facilitate data entry for each Node in a given NodeGroup instance.

While a Card will only handle data entry for a single NodeGroup (which may have many Nodes or NodeGroups), a single NodeGroup can be handled by more than one Card.

Throughout the Arches UI, Card Components are used to render Cards in both read-only and data entry contexts.

**Note:** Beginning in Arches 4.3, Card Components provide functionality formerly provided by Forms, Menus, and Reports.

### 22.7.1 CardModel

```python
class CardModel(models.Model):
    cardid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
    name = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    instructions = models.TextField(blank=True, null=True)
    cssclass = models.TextField(blank=True, null=True)
    helpenabled = models.BooleanField(default=False)
    helptitle = models.TextField(blank=True, null=True)
    helptext = models.TextField(blank=True, null=True)
    nodegroup = models.ForeignKey('NodeGroup', db_column='nodegroupid')
    graph = models.ForeignKey('GraphModel', db_column='graphid')
    active = models.BooleanField(default=True)
    visible = models.BooleanField(default=True)
    sortorder = models.IntegerField(blank=True, null=True, default=None)
    component = models.ForeignKey('CardComponent', db_column='componentid',
                                 default=uuid.UUID('f05e4d3a-53c1-11e8-b0ea-784f435179ea'), on_delete=models.SET_DEFAULT)
    config = JSONField(blank=True, null=True, db_column='config')

    def is_editable(self):
        result = True
        tiles = TileModel.objects.filter(nodegroup=self.nodegroup).count()
        result = False if tiles > 0 else True
```

(continues on next page)
if settings.OVERRIDE_RESOURCE_MODEL_LOCK == True:
    result = True
    return result

class Meta:
    managed = True
db_table = 'cards'

## 22.7.2 Card Component

A Card Component renders a Card.

class CardComponent(models.Model):
    componentid = models.UUIDField(primary_key=True, default=uuid.uuid1)
    name = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    component = models.TextField()
    componentname = models.TextField()
    defaultconfig = JSONField(blank=True, null=True, db_column='defaultconfig')

    @property
    def defaultconfig_json(self):
        json_string = json.dumps(self.defaultconfig)
        return json_string

class Meta:
    managed = True
db_table = 'card_components'

Field description:

- **name**: A name to be displayed in the UI for this component
- **description**: A description to be displayed in the UI for this component
- **component**: A require path for the JS module representing this component
- **componentname**: A Knockout.js component name used by this component (for rendering via knockout’s component binding handler)
- **defaultconfig**: A default JSON configuration object to be used by cards that implement this component

## 22.7.3 Widget

class Widget(models.Model):
    widgetid = models.UUIDField(primary_key=True, default=uuid.uuid1)  # This field type is a guess.
    name = models.TextField(unique=True)
    component = models.TextField(unique=True)
    defaultconfig = JSONField(blank=True, null=True, db_column='defaultconfig')
    helptext = models.TextField(blank=True, null=True)
    datatype = models.TextField()

    @property
    def defaultconfig_json(self):
22.7.4 DDataType

Used to validate data entered into widgets

class DDataType(models.Model):
    datatype = models.TextField(primary_key=True)
    iconclass = models.TextField()
    modulename = models.TextField(blank=True, null=True)
    classname = models.TextField(blank=True, null=True)
    defaultwidget = models.ForeignKey(db_column='defaultwidget', to='models.Widget',
                                       null=True)
    defaultconfig = JSONField(blank=True, null=True, db_column='defaultconfig')
    configcomponent = models.TextField(blank=True, null=True)
    configname = models.TextField(blank=True, null=True)
    issearchable = models.NullBooleanField(default=False)
    isgeometric = models.BooleanField()

    class Meta:
        managed = True
        db_table = 'd_data_types'

22.8 Data Model Graph Visualization

Fig. 1: Data model showing only Arches models.

Fig. 2: Full data model.
CHAPTER 23

Migrating Data from v3 to v4

**Terminology Note**

In v3 we had “resource graphs”, while in v4 we call these “Resource Models”. Conceptually they are the same. We’ll be referring to them here as “v3 graphs” and “v4 graphs”/“Resource Models”, respectively.

Upgrading your Arches installation is a complex process, as a significant backend redesign was implemented in v4. We have developed the following documentation (and the code to support it) to guide you through the process. You will be performing a combination of shell commands and basic file manipulation.

Before migrating data, you’ll need to install Arches 4 and make a new project. You can name your project whatever you want, but throughout this documentation we’ll refer to it as `my_project`. You can customize the templates and images in your project any time (before or after migrating the data). We recommend adding a Mapbox key so you can use the map for visual checks during the migration.

**See also:**

Refer to *Installation* and *Projects and Packages*.

Before moving on, you should be able to run the Django devserver from your project, and view it in a browser at `http://localhost:8000`.

Once you are ready, you can move on to exporting all of your legacy data from v3.

### 23.1 Preparation - Exporting v3 Data

You must extract some content from your v3 deployment before beginning the migration process.

#### 23.1.1 Export the v3 business data

In your v3 command line run:
python manage.py packages -o export_resources -d v3fullexport.json

You will get a console update during the process, which could take a few minutes. The result will be two files:
- v3fullexport.json – these are the resources
- v3fullexport.relations – these are the resource-to-resource relationships

Place these files somewhere easy to access.

23.1.2 Export the v3 reference data

In a browser, go to your v3 RDM and export the “Arches” scheme.

1. Click **Tools** and choose **Export Scheme**.
2. In the dialog, choose the “Arches” scheme.
3. Click **Export** and you’ll be taken to a browser page showing the contents of the scheme in XML format.
4. Right-click anywhere on the page and choose **Save as...** or **Save page as...**
5. Name your file “v3reference_data.xml”, and place it somewhere easy to access.

**Warning:** You are only able to migrate one scheme. If your v3 dropdown lists are composed of concepts from two different schemes (i.e. you added another scheme alongside “Arches”, added concepts to it, and then added those concepts to dropdown lists) you’ll need to manually consolidate these schemes into one before exporting.

**Dropdown Lists themselves are not migrated, they are recreated in v4 based on Top Concepts.**

23.1.3 Acquire the v3 resource graph _nodes.csv files

You should be able to find these in your original `source_data/resource_graphs` directory, whose contents should be a _edges.csv and _nodes.csv for every resource graph in your database. We only want the _nodes.csv files.

You can also export these directly from v3 with this command:

```python
python manage.py packages -o export_graphs -d path/to/temp/directory
```

Place these files somewhere easy to access.

**Arches-HIP Users**

Disregard this step, your _nodes.csv files will be provided for you.

23.1.4 Transfer all v3 uploaded media files

You must move all of the media files that have been uploaded to your v3 deployment to your v4 project.

By default, the directory in your new v4 project should be called `my_project/my_project/uploadedfiles`. If this directory doesn’t exist, create it, and move all of the v3 media into it.

**AWS S3 and Azure Users**
You can potentially use the same storage bucket, and just point your v4 project at it. Just make sure your content is in a folder called uploadedfiles. In theory this should work, but we haven’t tested it.

Proceed to…

**Arches-HIP Workflow**

1. Download the prepared v4 HIP package.

   The package is located here: github.com/legiongis/arches-v4-hip-pkg.
   You can download it from Github, then unzip, rename, and place it in your project, or use this command to clone it locally:
   ```
   git clone https://github.com/legiongis/arches-v4-hip-pkg pkg
   ```
   The result should be a new package within your project named pkg:
   ```
   my_project/
   └── manage.py
   my_project/
   ├── pkg/
   ```
   Now go into your project’s `my_project/my_project/settings.py` file and add this new line somewhere after the `APP_ROOT` line:
   ```python
   PACKAGE_DIR = os.path.join(os.path.dirname(APP_ROOT),'pkg')
   ```
   **Note:** You can actually name your new package whatever you want, and place it wherever you want, as long as `PACKAGE_DIR` holds the path to it. You can even leave out this setting entirely if you pass `--target path/to/package` to all of the v3 commands below.

2. Move your v3 data into the package.

   Move `v3fullexport.json` and `v3fullexport.relations` from **v3 Preparation Step 1** into `v3data/business_data`.
   Move `v3reference_data.xml` from **v3 Preparation Step 2** into `v3data/reference_data`.
   Your package should now look like this:
   ```
pkg/
  └── v3data/
        └── business_data/
                └── v3fullexport.json
                └── v3fullexport.relations
        └── graph_data/
        └── reference_data/
                └── v3reference_data.xml
        └── rm_configs.json
   ```
3. Convert your v3 reference data.

Run:

```bash
python manage.py v3 convert-v3-skos
```

New v4 reference data files will be created as shown below.

```
pkg/
  └ reference_data/
      └ collections/
          └ collections.xml
      └ concepts/
          └ thesaurus.xml
      └ v3topconcept_lookup.json # already existed
```

You can also add the `-i/--import` flag to automatically load the reference data into your database.

**Tip:** At this time you may as well load your package. This will prepare your database with the Resource Models and reference data, and get you ready for the final two steps:

```bash
python manage.py packages -o load_package -s "full/path/to/my_project/pkg/"
```

4. Write the v4 resource JSON.

Now you are ready to convert and import your v3 data:

```bash
python manage.py v3 write-v4-json
```

This command will create new v4 resource JSON files in `pkg/business_data`, one per Resource Model. You'll be provided with easy copy/paste commands to load the files if you want, or you can add `-i/--import` to the command to load the resources directly.

To help you debug any errors you encounter, and generally give you more control over this command, we've provided a number of optional arguments.

- `-i, --import` Directly imports the resources after the JSON file is created.
- `-m, --resource-models` List the names of resource models to process, by default all are used.
- `-n, --number` Limits the number of resources to load.
- `--exclude` List of resource ids (uuids) to exclude from the write process.
- `--verbose` Enables verbose printing during the process. Not recommended for large operations.

To give an example:

```bash
python manage.py v3 write-v4-json -m "Activity" -n 100 -i --exclude 08b68d46-c202-458a-bf11-bc7a1dd5b2ef
```

will write only the first 100 “Activity” resources to v4 JSON (even if there are more Resource Models in your package), excluding a single resource whose id is `08b68d46-c202-458a-bf11-bc7a1dd5b2ef`, and will then immediately import these resources into your database.
Tip: During this process, it may be useful to use:

```
python manage.py resources -o remove_resources -y
```

to erase all existing resources in your database and start from scratch.

5. Write the v4 resource relations file.

Once you have all of your resources loaded in your database, you can import the resource-to-resource relations from v3. Use:

```
python manage.py v3 write-v4-relations
```

to write the file, and add \-i/-\--import to directly import them. You will likely get errors if you try to import them but have not loaded all of your resources.

6. Load the entire package.

Though you may have been loading the individual pieces of the package along the way, the final step should be a full reload of the package. The following command will erase your whole database and load it with the reference data, Resource Models, and resources you have just created.

```
python manage.py packages -o load_package -s "/full/path/to/my_project/pkg" --db true
```

Full Workflow

Experienced developers should be able to use some of these steps individually to accomplish discrete tasks, but we generally recommend following this workflow as a whole.

Note: All of the commands below must be run from within your v4 project.

1. Create a new package

```
python manage.py packages -o create_package -d pkg
```

The result should be a new package within your project named pkg:

```
my_project/
  manage.py
  my_project/
  pkg/
```

Now go into your project's `my_project/my_project/settings.py` file and add this new line somewhere after the `APP_ROOT` line:

```
PACKAGE_DIR = os.path.join(os.path.dirname(APP_ROOT), 'pkg')
```
Note: You can actually name your new package whatever you want, and place it wherever you want, as long as PACKAGE_DIR holds the path to it. You can even leave out this setting entirely if you pass --target path/to/package to all of the v3 commands below.

2. Prepare your package.

```python manage.py v3 start-migration```

This will create some new directories and content in your package:

```
pkg/
   | reference_data/
   |   | v3topconcept_lookup.json
   | v3data/
   |   | business_data/
   |   | graph_data/
   |   | reference_data/
```

3. Move your v3 data into the package.

Move `v3fullexport.json/v3fullexport.relations` from v3 Preparation Step 2 into v3data/business_data.

Move `v3reference_data.xml` from v3 Preparation Step 2 into v3data/reference_data.

Move the _nodes.csv files from v3 Preparation Step 3 into v3data/graph_data.

Your package should now look like this:

```
pkg/
   | v3data/
   |   | business_data/
   |   |  v3fullexport.json
   |   | v3fullexport.relations
   |   | graph_data/
   |   |  RESOURCE_GRAPH.Exx_nodes.csv
   |   |  etc.
   |   | reference_data/
   |   |  v3reference_data.xml
   |   |  rm_configs.json
```


Run:

```python manage.py v3 convert-v3-skos --import```

New v4 reference data files will be created as shown below, and the --import flag will automatically load them into your database.

Now that the v3 reference data has been loaded, you are ready to create the v4 Resource Models. This migration process does not attempt to create them based on your old v3 graphs. There are a number of reasons for this, but most simply, v4 graphs have different constraints and support different datatypes and structures than those in v3. In other words, your v4 database will be better off with graphs that have been created natively, not translated from v3.

Generally, we would expect the v4 graphs to look like their v3 analogs, but we have built in quite a bit of wiggle room:

- The graph names can differ
- The node names can differ
- The graph structure can differ

However, there must still be a one-to-one relationship between v3 and v4 graphs.

When it comes to node datatypes, the translation from v3 to v4 is pretty straight-forward.

<table>
<thead>
<tr>
<th>v3 business table</th>
<th>v4 datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>strings</td>
<td>string</td>
</tr>
<tr>
<td>dates</td>
<td>date or edtf</td>
</tr>
<tr>
<td>geometries</td>
<td>geojson-feature-collection</td>
</tr>
<tr>
<td>domains</td>
<td>concept - if single value per v3 branch</td>
</tr>
<tr>
<td>domains</td>
<td>concept-list - if multiple values per v3 branch were allowed</td>
</tr>
</tbody>
</table>

Important: When you set a v4 node to concept or concept-list, you will need to select which collection to use. This is why it’s best to have migrated and loaded your RDM scheme (step 4 above) before making the Resource Models.

See also:

Refer to *Designing the Database* for help on this task. Within the Arches Designer itself, click for detailed help on each page.

Once you have built all of the Resource Models, export them into your package. You can do this one-by-one from the Arches Designer interface, or use:

```
python manage.py packages -o export_graphs -d pkg/graphs/resource_models -g "all"
```
By the end of this step, you should have one JSON file per Resource Model in pkg/graphs/resource_models.

6. Generate and populate the node lookup files.

Begin by running:

```
python manage.py v3 generate-rm-configs
```

which will create v3data/rm_configs.json. This file will be used to link the name of your v4 Resource Models with the names of their corresponding v3 graphs, as well as point to the files that link each node. Initially its content will look like:

```
{
   "Activity": {
      "v3_entitytypeid": "<fill out manually>",
      "v3_nodes_csv": "run 'python manage.py v3 generate-lookups",
      "v3_v4_node_lookup": "run 'python manage.py v3 generate-lookups"
   }
}
```

where "Activity" is the name of a v4 Resource Model. As the file says, you must now fill out the v3_entitytypeid value for all items. Typically, this will look something like "ACTIVITY.E7"—upper-case with a CRM class appended to it.

Now, also as the file says, run:

```
python manage.py v3 generate-lookups
```

and you’ll see the rest of the values get filled out.

There will now be more CSV files in the v3data/graph_data directory. There is one per v3 graph, and they are used to match the names of v3 node names (column one), with v4 node names (column two). All of the v3 nodes will be listed for you, but you have to fill out the v4 node names manually, using your new Resource Models for reference. A portion of a filled out file could look like:

```
Table 2: ACTIVITY.E7_v4_lookup.csv

<table>
<thead>
<tr>
<th>v3_node</th>
<th>v4_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVITY_TYPE.E55</td>
<td>Activity Type</td>
</tr>
<tr>
<td>ADDRESS_TYPE.E55</td>
<td>Address Type</td>
</tr>
<tr>
<td>etc...</td>
<td>etc...</td>
</tr>
</tbody>
</table>
```

Finally, you can use:

```
python manage.py v3 test-lookups
```

to check your work. Once this test passes, you can move on.
7. Write the v4 resource JSON.

Now you are ready to convert and import your v3 data:

```
python manage.py v3 write-v4-json
```

This command will create new v4 resource JSON files in `pkg/business_data`, one per Resource Model. You’ll be provided with easy copy/paste commands to load the files if you want, or you can add `-i/--import` to the command to load the resources directly.

To help you debug any errors you encounter, and generally give you more control over this command, we’ve provided a number of optional arguments.

- `-i, --import` Directly imports the resources after the JSON file is created.
- `-m, --resource-models` List the names of resource models to process, by default all are used.
- `-n, --number` Limits the number of resources to load.
- `--exclude` List of resource ids (uuids) to exclude from the write process.
- `--verbose` Enables verbose printing during the process. Not recommended for large operations.

To give an example:

```
python manage.py v3 write-v4-json -m "Activity" -n 100 -i --exclude 08b68d46-c202-458a-bf11-bc7a1dd5b2ef
```

will write only the first 100 “Activity” resources to v4 JSON (even if there are more Resource Models in your package), excluding a single resource whose id is `08b68d46-c202-458a-bf11-bc7a1dd5b2ef`, and will then immediately import these resources into your database.

**Tip:** During this process, it may be useful to use:

```
python manage.py resources -o remove_resources -y
```

to erase all existing resources in your database and start from scratch.

8. Write the v4 resource relations file.

Once you have all of your resources loaded in your database, you can import the resource-to-resource relations from v3. Use:

```
python manage.py v3 write-v4-relations
```

to write the file, and add `-i/--import` to directly import them. You will likely get errors if you try to import them but have not loaded all of your resources.

9. Load the entire package.

Though you may have been loading the individual pieces of the package along the way, the final step should be a full reload of the package. The following command will erase your whole database and load it with the reference data, Resource Models, and resources you have just created.
```bash
python manage.py packages -o load_package -s "full/path/to/my_project/pkg" --db true
```

After you have all the v3 data exported, you are ready to follow the appropriate workflow for your deployment.

**Which workflow should I use?**

If your v3 deployment of Arches was based on Arches-HIP, and you did not modify any of the graphs (but you did customize the RDM content), you can use the Arches-HIP Workflow. If you did modify some of the graph nodes (change their names, etc.), you can probably still use that workflow, but you’ll need to edit the `v3data/graph_data/...nodes_lookup.csv` file for that graph.
Currently, all data import and export operations happen through the Arches command line interface.

24.1 Importing Data

Arches provides methods for importing data in a few different formats. Generally, you are placing the values you want to import into a structured file. The form that each value takes, depends on the data type of its target node.

Be aware that the graph-based structure of Resource Models in Arches means that your data must be carefully prepared before import, to ensure that branches, groupings, and cardinality is maintained. The method for doing this is determined by which file format you decide to use. Additionally, the data type of the target node for each value in your file will dictate that value’s format.

24.1.1 Import Value Formats

- **string** In CSV, must be quoted only if the value contains a comma.

  Examples: Smith Cottage "Behold, the Forevertron." <p>This is a rich text description that contains HTML tags.</p>

- **number** Integers or floats; never use quotes or comma separators.

  Examples: 42 -46517322.464453

- **date** YYYY-MM-DD, no quotes.

  Examples: 1305-10-31 1986-02-02

- **edtf** Must be a valid Extended Date Time Format string.

  Examples: "2010-10" "-y10000"

- **geojson-feature-collection** In CSV, must be WKT.


In JSON, include the entire definition of a GeoJSON Feature Collection (the properties and id attributes can be empty). Use geojson.io and geojsonlint.com for testing.

Example - JSON:

```
"features": [
  {
    "geometry": {
      "coordinates": [-82.53973, 29.658642],
      "type": "Point"
    },
    "id": "<arbitrary id>",
    "properties": {},
    "type": "Feature"
  },
  {
    "type": "FeatureCollection"
  }
]
```

- **concept**

In CSV/SHP, if the values in your concept collection are *unique* you can use the label (prefLabel) for a concept. If not, you will get an error during import and you must use UUIDs instead of labels (if this happens, see Concepts File below). If a prefLabel has a comma in it, it must be triple-quoted: ""Shingles, original"".

**Examples - CSV/SHP:** Slate ""Shingles, original""

In JSON, you must use a concept’s UUID.

- **concept-list**

In CSV/SHP, must be a single-quoted list of prefLabels (or UUIDs if necessary). If a prefLabel contains a comma, then that prefLabel must have double-quotes: "Slate,""Shingles, original"", Thatch".

**Examples - CSV/SHP:** "Slate, Thatch" Brick

In JSON, a list of UUIDs must be used. If only one value is present, it must still be placed within brackets.

**Examples - JSON:** ["d11630fa-c5a4-49b8-832c-5976e0044bca", "651c59b0-ff30-11e8-9975-94659cf754d0", "cdcc206d-f80d-4cc3-8685-40e8949158f8"]

- **domain-value**

A string that matches a valid domain value for this node, single-quoted if it contains a comma.

- **domain-value-list**

A single-quoted list of strings that match valid domain values for this node. Follow quoting guidelines for concept-list if any of the values contain commas.

- **file-list**

In CSV/SHP, must be a relative path to the file that should be uploaded.

In JSON, a full definition of the file-list data looks like this, though some of these attributes, like lastModified and size can be omitted:
The file should already exist in the proper uploadedfiles directory before the JSON file is imported.

24.2 Importing a CSV

One method of bulk loading data into Arches is to create a CSV (comma separated values) file. We recommend using MS Excel or Open Office for this task. More advanced users will likely find a custom scripting effort to be worthwhile.

Note: Your CSV should be encoded into UTF-8. These steps will help you if you are using MS Excel.

The workflow for creating a CSV should be something like this:

1. Identify which Resource Model you are loading data into
2. Download the mapping file and concepts file for that resource model
3. Modify the mapping file to reference your CSV
4. Populate the CSV with your data
5. Import the CSV using the Import business data command.

24.2.1 CSV File Requirements

Each row in the CSV can contain the attribute values of one and only one resource.

The first column in the CSV must be named ResourceID. ResourceID is a user-generated unique ID for each individual resource. If ResourceID is a valid UUID, Arches will adopt it internally as the new resource’s identifier. If ResourceID is not a valid UUID Arches will create a new UUID and use that as the resource’s identifier. Subsequent columns can have any name.

ResourceIDs must be unique among all resources imported, not just within each csv, for this reason we suggest using UUIDs.

<table>
<thead>
<tr>
<th>Resource ID</th>
<th>attribute 1</th>
<th>attribute 2</th>
<th>attribute 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
<tr>
<td>2</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
<tr>
<td>3</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
</tbody>
</table>
Simple CSV with three resources, each with three different attributes.

Or, in a raw format (if you open the file in a text editor), the CSV should look like this:

| Resource ID, attribute 1, attribute 2, attribute 3 |
|---------|---------|---------|
| 1, attr. 1 value, attr. 2 value, attr. 3 value |
| 2, attr. 1 value, attr. 2 value, attr. 3 value |
| 3, attr. 1 value, attr. 2 value, attr. 3 value |

Multiple lines may be used to add multiple attributes to a single resource. You must make sure these lines are contiguous, and every line must have a ResourceID. Other cells are optional.

<table>
<thead>
<tr>
<th>Resource ID</th>
<th>attribute 1</th>
<th>attribute 2</th>
<th>attribute 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
<tr>
<td>2</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
<tr>
<td>2</td>
<td>attr. 2 additional value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>attr. 1 value</td>
<td>attr. 2 value</td>
<td>attr. 3 value</td>
</tr>
</tbody>
</table>

CSV with three resources, one of which has two values for attribute 2.

Depending on your Resource Model’s graph structure, some attributes will be handled as “groups”. For example, Name and Name Type attributes would be a group. Attributes that are grouped must be on the same row. However, a single row can have many different groups of attributes in it, but there may be only one of each group type per row. (e.g. you cannot have two names and two name types in one row).

<table>
<thead>
<tr>
<th>Resource ID</th>
<th>name</th>
<th>name_type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yucca House</td>
<td>Primary</td>
<td>“this house, built in...”</td>
</tr>
<tr>
<td>2</td>
<td>Big House</td>
<td>Primary</td>
<td>originally a small cabin</td>
</tr>
<tr>
<td>2</td>
<td>Old Main Building</td>
<td>Historic</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Writer’s Cabin</td>
<td>Primary</td>
<td>housed resident authors</td>
</tr>
</tbody>
</table>

CSV with three resources, one of which has two groups of name and name_type attributes. Note that “Primary” and “Historic” are the prefLabels for two different concepts in the RDM.

You must have values for any required nodes in your resource models.

Note: If you are using MS Excel to create your CSV files, double-quotes will automatically be added to any cell value that contains a comma.

24.2.2 Mapping File

All CSV files must be accompanied by a mapping file. This is a JSON-structured file that indicates which node in a Resource Model’s graph each column in the CSV file should map to. The mapping file should contain the source column name populated in the file_field_name property for all nodes in a graph the user wishes to map to. The mapping file should be named exactly the same as the CSV file but with the extension ‘.mapping’, and should be in the same directory as the CSV.

To create a mapping file for a Resource Model in your database, go to the Arches Designer landing page. Find the Resource Model into which you plan to load resources, and choose Export Mapping File from the Manage menu.

Unzip the download, and you’ll find a .mapping file as well as a _concepts.json file (see Concepts File). The contents of the mapping file will look something like this:
The mapping file contains cursory information about the resource model (name and resource model id) and a listing of the nodes that compose that resource model. Each node contains attributes to help you import your business data (not all attributes are used on import, some are there simply to assist you). The concept_export_value attribute is only present for nodes with datatypes of concept, concept-list, domain, and domain-list - this attribute is not used for import. It is recommended that you not delete any attributes from the mapping file. If you do not wish to map to a specific node simply set the file_field_name attribute to "".

You will now need to enter the column name from your CSV into the file_field_name in appropriate node in the mapping file. For example, if your CSV has a column named “activity_type” and you want the values in this column to populate “Activity Type” nodes in Arches, you would add that name to the mapping file like so:

```
{
  ...,
  {
    "arches_nodeid": "bbc5cf1f-fa16-1e6-9e3e-026d961c88e6",
    "arches_node_name": "Activity Type",
    "file_field_name": "activity_type", <-- place column name here
    "data_type": "concept",
    "concept_export_value": "label",
    "export": false
  },
  ...
}
```

To map more than one column to a single node, simply copy and paste that node within the mapping file.

### 24.2.3 Concepts File

When populating concept nodes from a CSV you should generally use the prefLabel for that concept. However, in rare instances there may be two or more concepts in your collection that have identical prefLabels (this is allowed in Arches). In this case you will need to replace the prefLabel in your CSV with the UUID for the Value that represents that prefLabel.
To aid with the process, a “concepts file” is created every time you download a mapping file, which lists the valueids and corresponding labels for all of the concepts in all of the concept collections associated with any of the Resource Model’s nodes. For example:

```
"Name Type": {
    "ecb20ae9-a457-4011-83bf-1c936e2d6b6a": "Historic",
    "81dd62d2-6701-4195-b74b-8057456bba4b": "Primary"
},
```

You would then need to use 81dd62d2-6701-4195-b74b-8057456bba4b instead of Primary in your CSV.

### 24.3 Shapefile Import

```
python manage.py packages -o import_business_data -s 'path_to_shapefile' -c 'path_to_mapping_file' [-ow {'overwrite'|'append'}]
```

Uploading a shapefile to Arches is very similar to uploading a CSV file with a few exceptions. The same rules apply to rich text, concept data, grouped data, and contiguousness. And, like CSV import, shapefile import requires a mapping file. Note that in this mapping file, the node you wish to map the geometry to must have a `file_field_name` value of ‘geom’.  

Other Requirements:

- The shapefile must contain a field with a unique identifier for each resource named ‘ResourceID’.
- The shapefile must be in WGS 84 (EPSG:4326) decimal degrees.
- The shapefile must consist of at least a .shp, .dbf, .shx, and .prj file. It may be zipped or unzipped.
- Dates in a shapefile can be in ESRI Shapefile date format, Arches will convert them to the appropriate date format. They can also be strings stored in YYYY-MM-DD format.

**Note:** More complex geometries may encounter a `mapping_parser_exception` error. This error occurs when a geometry is not valid in elasticsearch. To resolve this, first make sure your geometry is valid using ArcMap, QGIS, or PostGIS. Next, you can modify the precision of your geometry to 5 decimals or you can simplify your geometry using the QGIS simplify geometry geoprocessing tool, or the PostGIS `st_snaptogrid` function.

### 24.4 JSON Import

```
python manage.py packages -o import_business_data -s 'path_to_json' [-ow {'overwrite'|'append'}]
```

JSON import of business data is primarily intended for transferring business data between arches instances. Because of this it’s not especially user-friendly to create or interpret the JSON data format, but doing so is not impossible. 

First, there are at least two ways you can familiarize yourself with the format. The system settings in an Arches package is stored in this json format, you can open one of those up and take a look. Perhaps a better way in your case is to create some business data via the ui in your instance of arches and export it to the json format using the business data export command defined here [Export Commands](#). This can act as a template json for data creation. For the rest of this section it may be helpful to have one of these files open to make it easier to follow along.

General structure of the entire file:
The json format is primarily a representation of the tiles table in the arches postgres database with some information about the resource instance(s) included. Within the business_data object of the json are two objects, the tiles object and the resourceinstance object. Let’s start with the resource instance object.

Structure of the resourceinstance object:

```
{
  "graph_id": uuid,
  "resourceinstanceid": uuid,
  "legacyid": uuid or text
}
```

- `graph_id` - the id of the resource model for which this data was created
- `resourceinstanceid` - the unique identifier of this resource instance within Arches (this will need to be unique for every resource in Arches)
- `legacyid` - an identifier that was used for this resource before its inclusion in Arches. This can be the same as the resourceinstanceid (this is the case when you provide a UUID to the ResourceID column in a CSV) or it can be another id. Either way it has to be unique among every resource in Arches.

The tiles object is a list of tiles that compose a resource instance. The tiles object is a bit more complicated than the resourceinstance object, and the structure can vary depending on the cardinality of your nodes. The following cardinality examples will be covered below:

1. 1 card
2. n cards
3. 1 parent card with 1 child card
4. 1 parent card with n child cards
5. n parent cards with 1 child card
6. n parent cards with n child cards

But first a description of the general structure of a single tile:

```
{
  "tileid": "<uuid>",
  "resourceinstance_id": "<uuid>",
  "nodegroupid": "<uuid>",
  "sortorder": 0,
  "parenttile_id": "<uuid>" or null,
  "data": { . . . }
}
```

- `tileid` - unique identifier of the tile this is the primary key in the tiles table and must be a unique uuid
- **resourceinstance_id** - the uuid corresponding to the instance this tile belongs to (this should be the same as the resourceinstance_id from the resourceinstance object.
- **nodegroup_id** - the node group for which the nodes within the data array participate
- **sortorder** - the sort order of this data in the form/report relative to other tiles (only applicable if cardinality is n)
- **parenttile_id** - unique identifier of the parenttile of this tile (will be null if this is a parent tile or the tile has no parent)
- **data** - json structure of a node group including the nodeid and data populating that node. For example:

```json
{
  "data": {
    "<uuid for building name node>": "Smith Cottage"
  }
}
```

The tile object is tied to a resource model in two ways: 1) through the nodegroup_id 2) in the data object where nodeids are used as keys for the business data itself.

Now for a detailed look at the actual contents of tiles. Note that below we are using simplified values for tileid, like "A" and "B", to clearly illustrate parent/child relationships. In reality these must be valid UUIDs.

### 24.4.1 1 card

1: There is one and only one instance of this nodegroup/card in a resource:

```json
[
  {
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
    "data": {
      "nodeid": "some data",
      "nodeid": "some other data"
    }
  }
]
```

This structure represents a tile for a nodegroup (consisting of two nodes) with no parents collecting data with a cardinality of 1.

### 24.4.2 n cards

n: There are multiple instances of this nodegroup/card in a resource:

```json
[
  {
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
```

(continues on next page)
24.4.3 1 parent card with 1 child card

1-1: One and only one parent nodegroup/card contains one and only one child nodegroup/card:

```

[{
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
    "data": {}
},
{
    "tileid": "X",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "A",
    "data": {
        "nodeid": "data",
        "nodeid": "other data"
    }
}]
```

24.4.4 1 parent card with n child cards

1-n: One and only one parent nodegroup/card containing multiple instances of child nodegroups/cards:

```

[{
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "<uuid from resource model>",
    "data": {}
}]
```

24.4. JSON Import
24.4.5 n parent cards with 1 child card

n-1: Many parent nodegroups/cards each with one child nodegroup/card:

```json
[
  {
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodelogroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
    "data": {}
  },
  {
    "tileid": "X",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodelogroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "A",
    "data": {
      "nodeid": "data",
      "nodeid": "other data"
    }
  },
  {
    "tileid": "Y",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodelogroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "A",
    "data": {
      "nodeid": "more data",
      "nodeid": "more other data"
    }
  }
]
```
24.4.6 n parent cards with n child cards

n-n: Many parent nodegroups/cards containing many child nodegroups/cards:

```
[
  {
    "tileid": "A",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
    "data": {}
  },
  {
    "tileid": "X",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "A",
    "data": {
      "nodeid": "data",
      "nodeid": "other data"
    }
  },
  {
    "tileid": "B",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": null,
    "data": {}
  },
  {
    "tileid": "Y",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "B",
    "data": {
      "nodeid": "more data",
      "nodeid": "more other data"
    }
  }
]
```
"parenttile_id": "B",
"data": {
    "nodeid": "more data",
    "nodeid": "more other data"
}
},
{
    "tileid": "Z",
    "resourceinstance_id": "<uuid from resourceinstance.resourceinstanceid>",
    "nodegroupid": "<uuid from resource model>",
    "sortorder": 0,
    "parenttile_id": "B",
    "data": {
        "nodeid": "even more data",
        "nodeid": "even more other data"
    }
}
]

24.5 Importing Resource Relations

It is possible to batch import Resource Relations (also referred to as “resource-to-resource relationships”). To do so, create a .relations file (a CSV-formatted file with a .relations extension). The header of the file should be as follows:

| resourceinstanceidfrom, resourceinstanceidto, relationshiptype, datestarted, dateended, notes |

In each row, resourceinstanceidfrom and resourceinstanceidto must either be an Arches ID (the UUID assigned to a new resource when it is first created) or a Legacy ID (an identifier from a legacy database that was used as a ResourceID in a JSON or CSV import file).

You can find the UUID value for your desired relationshiptype in the concept.json file downloaded with your resource model mapping file.

datestarted, dateended and notes are optional fields. Dates should be formatted YYYY-MM-DD.

Once constructed you can import the .relations file with the following command:

```bash
python manage.py packages -o import_business_data_relations -s 'path_to_relations_file'
```

All the resources referenced in the .relations CSV need to already be in your database. So make sure to run this command after you have imported all the business data referenced in the .relations file.

**Note:** You can also create relationships between resources using the resource-instance data type. When you are making the graph for a new resource model, you can set one of the nodes to hold a resource instance. This is not the same as creating Resource Relations as described above.
24.6 Exporting Arches Data

All file-based business exports must happen through the command line interface. The output format can either be JSON (the best way to do a full dump of your Arches database) or CSV (a more curated way to export a specific subset of data). To use Arches data in other systems or export shapefiles, users will have to begin by creating a new resource database view (see below).

24.6.1 Writing Business Data Files

The output format can either be JSON (the best way to do a full dump of your Arches database) or CSV (a more curated way to export a specific subset of data).

To export JSON, use:

```bash
python manage.py packages -o export_business_data -d 'path_to_destination_directory' -f 'json' -g 'resource_model_uuid'
```

Note that you’ll have to provide the UUID for the Resource Model whose resources you want to export. The easiest way to find this UUID is by looking at the browser url while editing the Resource Model in the Arches Designer UI.

To export CSV, use:

```bash
python manage.py packages -o export_business_data -d 'path_to_destination_directory' -f 'csv' -c 'path_to_mapping_file'
```

When exporting to CSV, you need to use a Mapping File, which will determine the content of your CSV (which nodes are exported, etc.). Add the --single-file argument to export your grouped data to the same CSV file as the rest of your data.

More about these export commands can be found in Export Commands.

24.6.2 Resource Database Views

To export to spatial formats such as shapefile, it is necessary to flatten the graph structure of your resources. One way to do this is to create a database view of your resource models. Arches does not do this automatically because there are many ways to design a flattened table depending on your needs.

You can add any number of database views representing a given resource model either for export, or to connect directly to a GIS client such as QGIS or ArcGIS. When writing a view to support shapefile export be sure that your view does not violate any shapefile restrictions. For example, shapefile field names are limited to 10 characters with no special characters and text fields cannot store more than 255 characters.

If you plan to use the arches export command to export your view as a shapefile, you also need to be sure that your view contains 2 fields: geom with the geometry representing your resource instance’s location and geom_type with the postgis geometry type of your geom column.

To write your view, you should start by getting a mapping file for your resource. You can do that by going to the Arches Designer page and then in the manage dropdown of your resource model select Create Mapping File. A zip file will be downloaded and within that file you will find your .mapping file. This file lists all the ids that you will need to design your view.

Below is an example of a simple resource model view. If a resource instance has a tile with geojson saved to it, that tile will be represented as a record in the view along with the corresponding nodeid and tileid. A unique id (gid) is assigned to each row. If a node has more than one geometry, the geometries are combined into a multipart geometry. If a node has more than one geometry of different types, a record will be created for each type. The UUID (ab74af76-fa0e-11e6-9e3e-026d961c88e6) in the last line of this example is the id of the view’s resource model.
1. When creating your own view, you will need to replace this UUID with your own resource model’s id. You can find this UUID in your mapping file assigned to the property: `resource_model_id`.

```sql
CREATE OR REPLACE VIEW vw_monuments_simple AS
 WITH mv AS (SELECT tileid, resourceinstanceid, nodeid, ST_Union(geom) AS geom,
    ST_GeometryType(geom) AS geom_type
 FROM mv_geojson_geoms
 GROUP BY tileid, nodeid, resourceinstanceid, ST_GeometryType(geom))
 SELECT row_number() OVER () AS gid,
    mv.resourceinstanceid,
    mv.tileid,
    mv.nodeid,
    ST_GeometryType(geom) AS geom_type,
    geom
 FROM mv
 WHERE (SELECT graphid FROM resource_instances WHERE mv.resourceinstanceid = resourceinstanceid) = 'ab74af76-fa0e-11e6-9e3e-026d961c88e6'
```

2. Here is a more complete example which includes columns with tile data:

```sql
CREATE OR REPLACE VIEW vw_monuments AS
 WITH mv AS (SELECT tileid, resourceinstanceid, nodeid, ST_Union(geom) AS geom,
    ST_GeometryType(geom) AS geom_type
 FROM mv_geojson_geoms
 GROUP BY tileid, nodeid, resourceinstanceid, ST_GeometryType(geom))
 SELECT row_number() OVER () AS gid,
    mv.resourceinstanceid,
    mv.tileid,
    mv.nodeid,
    ST_GeometryType(geom) AS geom_type,
    name_tile.tiledata ->> '677f303d-09cc-11e7-9aa6-6c4008b05c4c' AS name,
    (SELECT value FROM values WHERE cast(name_tile.tiledata ->> '677f39a8-09cc-11e7-834a-6c4008b05c4c' AS uuid) = valueid ) AS nametype,
    (SELECT value FROM values WHERE cast(component.tiledata -> 'ab74afec-fa0e-11e6-9e3e-026d961c88e6')::uuid) item_id LEFT JOIN values v ON v.
    valueid=item_id, '') AS const_tech,
    record.tiledata ->> '677f2c0f-09cc-11e7-b412-6c4008b05c4c' AS record_type,
    geom
 FROM mv
 LEFT JOIN tiles name_tile
 ON mv.resourceinstanceid = name_tile.resourceinstanceid
 AND name_tile.tiledata ->>'677f39a8-09cc-11e7-834a-6c4008b05c4c' != ' '
 LEFT JOIN tiles component
 ON name_tile.resourceinstanceid = component.resourceinstanceid
 AND component.tiledata ->>'ab74afec-fa0e-11e6-9e3e-026d961c88e6' != ' '
 LEFT JOIN tiles record
 ON name_tile.resourceinstanceid = record.resourceinstanceid
 AND record.tiledata ->>'677f2c0f-09cc-11e7-b412-6c4008b05c4c' != ' '
 WHERE (SELECT graphid FROM resource_instances WHERE mv.resourceinstanceid = resourceinstanceid) = 'ab74af76-fa0e-11e6-9e3e-026d961c88e6'
```
3. You will notice that for each node added as a column in the table, we perform a LEFT JOIN to the tiles table and the nodeid from which we want data. Here is an example joining to the tile containing the record node which has a nodeid of 677f2c0f-09cc-11e7-b412-6c4008b05c4c.

```
LEFT JOIN tiles record
  ON name_tile.resourceinstanceid = record.resourceinstanceid
  AND record.tiledata->>'677f2c0f-09cc-11e7-b412-6c4008b05c4c' != ''
```

4. We can then define a field be referencing that tile:

```
(SELECT value FROM values WHERE cast(record.tiledata ->> '677f2c0f-09cc-
˓→11e7-b412-6c4008b05c4c' AS uuid) = valueid ) AS record_type
```

5. How you define your fields depends largely on what the node datatype is:

A node with a string datatype:

```
name_tile.tiledata -> '677f303d-09cc-11e7-9aa6-6c4008b05c4c' AS name
```

A node with a concept value id. The following returns the concept values label:

```
(SELECT value FROM values WHERE cast(name_tile.tiledata -> '677f39a8-
˓→09cc-11e7-834a-6c4008b05c4c' AS uuid) = valueid ) AS nametype
```

A node with a concept-list. The following returns a concatenated string of concept value labels:

```
array_to_string((SELECT array_agg(v.value) FROM unnest(ARRAY(SELECT jsonb_˓→array_elements_text(component.tiledata -> 'ab74afec-fa0e-11e6-9e3e-
˓→026d961c88e6'))::uuid[]) item_id LEFT JOIN values v ON v.valueid=item_˓→id), ',') AS const_tech
```
In reality, many more settings are used than are exposed in the UI. The best place to see all available settings is in the main arches repo.

Additionally, settings can be defined in many different places. Here is the full inheritance pattern for a typical Arches project:

- **arches/settings.py** If you installed Arches through pypi (pip install arches) this file will be deep in your virtual environment, and you shouldn’t touch it.
  - values here can be superseded by...

- **my_project/my_project/settings.py** Settings here define backend information specific to your app. For example, this is where you would add new references to template context processors.
  - values here can be superseded by...

- **my_project/my_project/package_settings.py** Settings here define backend information specific to the package loaded to your app. You do not need to create or modify this file as it will be loaded when you load a package. However, you may want to edit this file if your intent is to design or modify a package.
  - values here can be superseded by...

- **my_project/my_project/settings_local.py (optional)** Typically kept out of version control, a settings_local.py file is used for 1) sensitive information like db credentials or keys and 2) environment-specific settings, like paths needed for production configuration.
  - values here can be superseded by...

- **System Settings Manager** Settings exposed to the UI are the end of the inheritance chain. In fact, these settings are stored as a resource in the database, and the contents of this resource is defined in the System Settings Graph. Nodes in this graph with a name that matches a previously defined setting (i.e. in the files above) will override that value with whatever has been entered through the UI.

If you’re a developer, you’ll notice that the codebase uses:
from arches.app.models.system_settings import settings

in favor of:

from django.conf import settings

This is to ensure that UI settings are implemented properly. If you are using settings outside of a UI context you will need to follow the import statement with settings.update_from_db().
An easy way to get started with Arches is to load the arches4-example-pkg into your new project. The database schema of this package is generally based on Arches-HIP, which was developed alongside Arches v3, and it comprises 6 Resource Models and 36 Branches.

### 26.1 Resource Models

- **Activity Resource Model** This resource model describes activities relating to heritage resources and heritage resource groups.
- **Actor Resource Model** This resource model describes actor resources such as individual people and groups of people.
- **Heritage Resource Group Resource Model** This resource model describes heritage resource groups which are groupings of historically significant resources. Those historically significant resource may more may not themselves be instances of Heritage Resource.
- **Heritage Resource Model** This resource model describes heritage resources, which includes monuments, buildings, structures, etc.
- **Historical Event Resource Model** Resource Model for the Historical Event Resource (E5), which is used to describe significant historical events.
- **Information Resource Model** This resource model defines information resources, such as images, reports, and publications.

### 26.2 Branches

- **Activity Phase** Describes the time span and type of an Activity Resource.
- **Actor Phase** Phase Type Assignment for the Actor resource model. Connects to Actor E39 via P41i
• **Appellation**  Describes an appellation assigned to an Actor Resource.
  
  Relates to resource model via P131

• **Beginning of Existence**  Describes the type and time span of the beginning of a resource’s existence.
  
  Relates to the Actor resource model E39 via P92i. Relates to Historic Event E5 and Activity E7 via p116i.

• **Component**  Physical thing on a heritage resource. Connect to Heritage Resource E18 via P46

• **Condition Assessment**  Describes the conditions, threats, and disturbances affecting a Heritage Resource or Heritage Resource Group. Additional information may include a management recommendation, a condition image, the data the condition was assessed and a description of the condition.
  
  This branch meets technical business rules for CRM compliance but CRM experts have said that this is an inappropriate implementation of E62.
  
  Connects to all resources with P140i

• **End of Existence**  Describes the type and time span of the end of a resource’s existence.
  
  Relates to the Actor E39 Resource Model via P93i

• **Evaluation**  This branch evaluates instances of Heritage Resource E18 and Heritage Resource Group E27, and is based on the City of Los Angeles’ SurveyLA survey methodology for evaluation of significance and eligibility for designation as a heritage resource or heritage resource group. Evaluation consists of defining a historic context (Evaluation Criteria) which triggers specific eligibility requirements (Eligibility Requirements) and results in the selection of one or more status codes (Status) and the writing of a reasons statement (Reasons). The branch also contains nodes to record the date of Evaluation, the dates defining the Period of Significance for evaluation.
  
  Relates to E18 and E27 resource models via P140i.

• **Event Phase Type**  Describes a Historical Event Type within a given timespan/phase.
  
  Connects to the Historical Event resource model via P10

• **Existence**  Start/End dates for resources. Assumed to be typed and within calendar time.
  
  Connects to Activity (E7) and Historical Event (E5) resource models via P114 Connects to Heritage Resource (E18) and Heritage Resource Group (E27) resource models via P12i Connects to Actor (E39) resource model via P11i

• **External Identifier**  Used to hold identifiers necessary to link a given resource to records in an external (perhaps legacy) system.

• **Heritage Resource Group Phase Type**  Phase Type Assignments for Heritage Resource Group E27
  
  Connects via P92i

• **Heritage Resource Phase Type**  Phase Type Assignments for Heritage Resource E18.
  
  Connects via P92i

• **Information Carrier**  An object or substance used to record and accumulate data

• **Information Resource Copyright**  Used to define legal privileges associated with an Information Resource.

• **Information Resource Creation Event**  Describes the creators, contributors, type, and date of a creation and/or update event of an information resource.
• **Keyword** An informative word used for resource information retrieval that indicates the content of a resource.
  Relates to resources via P2.

• **Language** This branch describes the language of an Information Resource.

• **Measurement** The Measurement branch measures instances of a Heritage Resource E18 or a Heritage Resource Group E27
  Relates to those resource models via P39i

• **Modification Event** Describes the modification of a Heritage Resource.
  Connects to Heritage Resource (E18) via P12i

• **Name** The name of a resource.
  Applies to Heritage Resource, Heritage Resource Group, Activity, and Historical Event. Actors receive their "names" from the Appellation branch.

• **Place 1** Describes the physical location of a heritage resource or heritage resource group. Includes extra nodes for cadastral information.
  Relates to Heritage Resource E18 with P53 Relates to Heritage Resource Group E27 with P89

• **Place 2** For Actor resources: Describes the former and current residences of an Actor resource. Relates to Actor E39 with P74 For Activity and Historic Event: Describes the location of where an activity or historic event took place. Relates to Activity E7 with P7. Relates to Historical Event E5 with P7

• **Place 3** Describes the location of an information resource. Relates to Information Resource E73 with P67

• **Publication Event** This branch describes the publication of an Information Resource.
  Connects to Information Resource (E73) with -P128

• **Resource Type Classification** Describes the type of resource.
  Applies to: Heritage Group, Heritage

• **Resource Update Event** This is intended as a sub-branch to Resource Creation Event within the Information Resource Resource Model. Stores the date when an update to the information resource occurred.

• **Right** Captures information about special status, protection, or privileges afforded under law.
  Applies to: Heritage Resource, Heritage Resource Model

• **Temporal Coverage** Defines time span for which the Information Resource is relevant. Applies to: Information Resource

• **Title** Describes the title of an Information Resource.
Glossary of Arches v4 Terminology

- **Arches Designer** - A user interface for facilitating database design, i.e. the creation of Resource Models. The Arches Designer consists of many different tools, such as the Graph Designer and Card Manager, each of which helps build a different facet of Resource Model creation.

- **Basemaps** - Underlying map layers which include, by default, aerial imagery, streetmaps, or terrain. You can also add your own basemaps through the same process as adding overlays.

- **Branch** - Branches are building blocks that aid in the creation of resource models. When you add a branch to a resource model, its contents are copied into the resource model. This allows you to further customize the resource model while leaving the original branch unaltered.

- **Card** - Cards are used to configure the data entry representation of a branch’s graph; they define how information will be collected for each nodegroup. In some cases a complex branch may have multiple cards, which will be aggregated into a card container. Cards contain widgets, and determine how the widgets are ordered in the user interface.

- **Concept** - A vocabulary term that is used throughout the Arches database to define resource. A concept has a preferred label (“house”) and may have any number of alternative labels (“domicile”, “townhouse”). When searching your database, a search for “domicile” would automatically use the “house” concept.

- **Concept Collections** - Concepts are grouped into collections. An example would be the concepts “Eastlake”, “Italianate”, and “Queen Anne”, all of which would be grouped in an “Architectural Style” concept collection.

- **Datatype** - A defined type of business data, such as a number or a date. Each node has a datatype.

- **Resource Model** - Resource Models are top-level categories for resources in your database. When creating new resources, a data entry user must decide which resource model to use, thereby defining what information is collected for the resource. The entire Arches Designer exists to facilitate the creation and customization of resource models.

- **Graph** - A network of nodes, connected by edges, that defines the set of attributes for either a Branch or a Resource Model. If an ontology is enforced on the graph, each node will belong to an ontological class and only certain types of edges may be used to connect them.
• **Menu** - Menus are groupings of cards associated with a given resource model. They allow for an organized, thematic approach to data entry.

• **Node** - The smallest unit of a graph, a node will have a name and datatype. If the graph participates in an ontology, the node must also have a CRM class, and a defined relationship (edge) between it and the node upstream of it.

• **Nodegroup** - Within graphs, nodes are aggregated into nodegroups. An example of a nodegroup would be Name and Name Type. Edit permissions are enforced at the nodegroup level.

• **Ontology** - A set of rules the govern the way nodes are defined and connected in a graph. Arches comes pre-loaded with the CIDOC CRM, an ontology developed by ICOM to model cultural heritage.

• **Overlays** - Static map layers that can be added to Arches. These could be historic maps, administrative boundaries, or existing map services published elsewhere.

• **Reference Data Manager (RDM)** - User interface for managing all of the concepts in your Arches database.

• **Resource** - An entry in your Arches database. Each resource is created using a resource model.

• **Resource Layers** - Map layers that are created from your Arches database. There is one resource layer for each node with datatype ‘geojson-feature-collection’ that is stored across all resource models.

• **Resource Relationships** - Arches provides the ability to relate one resource to another by creating resource relationships. Resource relationships are directional and will have an associated concept, such as “contains / is contained in”.

• **Resource Report** - A resource’s report shows all of the saved information for a resource. Templates for reports are associated with each resource models.

• **Time Wheel** - A graphical interface used to support advanced time-based visualization and search of your database.

• **Widget** - An input element designed to manage form input of a specific datatype. Each widget represents one node, and widgets for all nodes in a nodegroup are contained in a single card.
When you are putting your Arches project in production, you’ll need to serve it with a webserver. The following is a guide using Apache as an example, split into two sections:

- Setup Apache
- Handling Static Files

28.1 Setup Apache

During development, it’s easiest to use the Django webserver to view your Arches installation. However, once you are ready to put the project into production, you’ll have to use a more efficient webserver like Apache or nginx.

We have the most experience using Apache, which is very easy to install and configure. The following instructions work for Ubuntu 14.04, minor changes may be necessary for a different OS.

1. Get apache2 and mod_wsgi

   ```bash
   $ sudo apt-get install apache2
   $ sudo apt-get install libapache2-mod-wsgi
   ```

2. Create the Python process

   In order to properly configure Apache, we must:
• Create a python daemon process
• Set the path to your project’s wsgi.py file and reference to the python daemon process created above
• Give Apache access to the main project directory

All of these tasks are handled by adding a block of code to Apache’s ../sites-enabled/000-default.conf file. Use this command to open the file

```bash
$ sudo nano /etc/apache2/sites-enabled/000-default.conf
```

and paste the following code into the `<VirtualHost *:80>` stanza, changing directory and file paths where necessary:

```
WSGDaemonProcess arches python-path=/home/ubuntu/Projects/my_project:/<br>˓→home/ubuntu/Projects/ENV/lib/python2.7/site-packages<br>WSGIScriptAlias / /home/ubuntu/Projects/my_project/my_project/wsgi.py<br>˓→process-group=arches
```

```
<Directory /home/ubuntu/Projects/my_project>
  Options Indexes FollowSymLinks
  AllowOverride None
  Require all granted
</Directory>
```

Use `ctrl+x` to save the file.

You may find it helpful to read the [Official Django Documentation](https://docs.djangoproject.com) on serving Django apps with Apache and mod_wscgi.

3. Set file system permissions for Apache

Now we must give Apache write-permission in a few locations. We’ll do that by first changing the permissions of the necessary files and directories, and second by setting the Apache user as the group.

```bash
$ sudo chmod 775 /home/ubuntu/Projects/my_project/my_project
$ sudo chgrp www-data /home/ubuntu/Projects/my_project/my_project
$ sudo chmod 664 /home/ubuntu/Projects/my_project/my_project/arches.log
$ sudo chgrp www-data /home/ubuntu/Projects/my_project/my_project/arches.log
```

**Note:** On Ubuntu the Apache user is `www-data` (used in the example below), but on CentOS it is `httpd`.

These commands should give Apache sufficient permissions to create and modify the arches/uploadedfiles directory (where user uploads are stored by default) and the arches/tileserver directory where Tilestache caches tiles that it renders.

**Note:** Please post to the [Arches forum](https://www.archesproject.org/community) if you find that more permissions need to be modified, or that these directions can be simplified further.

4. Restart Apache.
   
   **Ubuntu**
You should now be able to view your app from any web browser by navigating directly to your IP address (you don’t need to run the Django dev server now).

Important: With Apache serving Arches, any changes to a .py file will not be reflected until you restart Apache:

If you are still in development and just want to use Apache instead of the Django server (and keep getting the lengthy Django error messages instead of a 500 page), you can stop here. Otherwise, you’ll need to continue on to handle your app’s static files.

### 28.2 Handling Static Files

There are two cases in which you need to follow these directions to handle static files (js, css, and images):

- You are going to set `DEBUG = False`, at which point Django will no longer serve them, or
- You are leaving `DEBUG = True` but are developing and serving your project from a non-root location, say `www.example.com/arches4` instead of `www.example.com`.

1. Create static files directory

   This directory can be placed anywhere. In the example below we are putting it inside of your project.

   ```
   $ mkdir /home/ubuntu/Projects/my_project/my_project/static
   ```

2. Configure your Arches project settings

   Now open your `settings.py` (or `settings_local.py`) file, and add these lines to it.

   ```
   STATIC_ROOT = os.path.join(PACKAGE_ROOT, 'static')
   STATIC_URL = '/static/
   ```

   This will point Django to your new static directory, and also tell it how to create a URL that points to that directory.

3. Collect the static files

   With your virtual environment activated, enter your project’s top directory and run this command:

   ```
   $ python manage.py collectstatic
   ```

   Watch as all of your static files (including those that come standard with Django) are copied to the new directory. Now we are ready to tell Apache where to find them.

   Important: With Apache serving your app you must run `python manage.py collectstatic` any time you make any changes to static files.

4. Configure Apache settings
Use

```bash
$ sudo nano /etc/apache2/sites-enabled/000-default.conf
```

to edit the default Apache configuration file. Find your `<VirtualHost *:80>` stanza with some familiar code in it. Below the original code you added, paste this block, changing paths as necessary.

```conf
Alias /static/ /home/ubuntu/Projects/my_project/my_project/static/

<Directory /home/ubuntu/Projects/my_project/my_project/static>
  Options Indexes FollowSymLinks
  AllowOverride None
  Require all granted
</Directory>
```

The `Alias` line tells Apache where to look when Django sends it the `/static/` URL, and the subsequent block allows Apache access to your newly created static directory.

4. Restart Apache.

```bash
$ sudo service apache2 restart
```
HTTP Routing Table

/mobileprojects
GET /mobileprojects, 90

/o
POST /o/token, 81

/rdm
GET /rdm/concepts/{uuid:concept instance id}, 83

/resources
GET /resources/, 85
GET /resources/{uuid:resource instance id}, 86
PUT /resources/{uuid:resource instance id}, 87
DELETE /resources/{uuid:resource instance id}, 90