Improvisor Documentation

Server Documentation

Server Connection

Connect to the server using an SSH Client such as PuTTY, MTPuTTY, or Linux OpenSSH Client.
Once connected to the server, system information such as System load, Memory usage is displayed by default.

Server Configuration

The virtual Linux server that the web application is hosted on runs on Ubuntu 18.04.2 LTS.

The web application uses Python Flask and is served using Gunicorn with Nginx as a reverse proxy.

The configuration was based on this setup guide on DigitalOcean, however certain configuration files have been altered considerably to support the required technologies such as web sockets.

The server is secured with HTTPS using Certbot.

The server files are retrieved from the project Github repository. However, these are not pulled automatically and must be updated manually.

Useful Files and Directories

- The github repository is stored in **/improvisor/repo/**
- The Python virtual environment is stored in **/improvisor/venv/**
- The configuration file for the Python Flask application can be found at **/improvisor/repo/improvisor/config/defaults.py**
- Log files are stored in **/improvisor/logs/**
- Some basic server instructions can be found at **/improvisor/server_instructions.txt**
- The main Nginx configuration file can be found at **/etc/nginx/nginx.conf**
- The Improvisor Nginx socket file can be found at **/etc/nginx/sites-enabled/improvisor**
    - When the socket is enabled/disabled the file location is switched between /**sites-enabled** (enabled) and **/sites-available** (disabled).
- The Linux Improvisor Service can be found at **/etc/systemd/system/improvisor.service**
    - This is service runs at server boot and loads the web application

- The database for the website can be found at **/improvisor/repo/improvisor/data.db**

Useful Commands

Run all commands as sudo (means you don't need to use the sudo command prefix):
- *sudo -s*

Reboot the virtual Linux server:
- reboot

Restart the Improvisor service (primarily used after Github repo changes)
- *service improvisor restart*
- *service improvisor status*
- Check the status of the Improvisor service from the output of this command
- If the service is active then the web app should be live with the latest changes
- A screenshot example of an active service can be found here.

Update the server with the latest changes from the Github repository:
- *cd /improvisor/repo*
- *git pull*
- Reboot the Improvisor service (as seen above)

Update the virtual environment (if a Python pip requirement is added):
- *cd /improvisor*
- *source /improvisor/venv/bin/activate*
- *cd /improvisor/repo*
- *pip install -r requirements.txt*
- Reboot the Improvisor service (as seen above)

Reload Nginx:
- *systemctl reload nginx*

Restart Nginx:
- *systemctl restart nginx*

Additional Information

- There is currently a user file size upload limit of 32MB. This is set by Nginx in **nginx.conf**
- Gunicorn is currently configured to run with only 1 worker. This is because web sockets do not currently support multiple workers. This is set by Linux in **improvisor.service**
- The github repository name is different to the directory of the github repository on the server. The the repository is stored as '**repo**' on the server but it is the exact same repository. This is to reduce the length of file directories in configuration files and commands
- There is no live 'console' for the Python Flask application on the server like what you see when you run a local Python application. You won't be able to see print statements but certain outputs and events are logged to log files.
- The Python Flask application is split into different modules (forms, routes, sockets etc.)

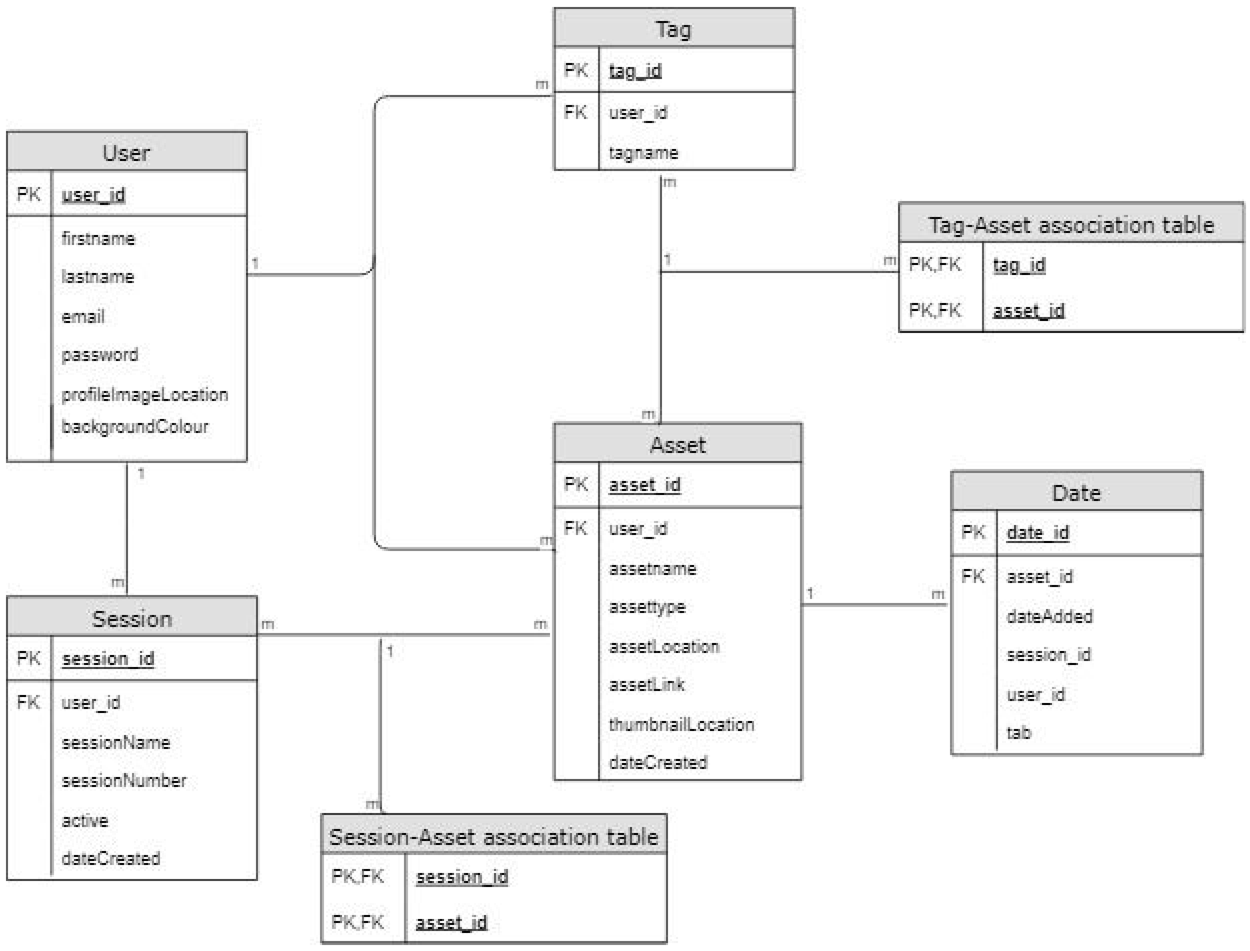
Database Documentation

The structure of the database is represented by the Entity-Relationship model below. This model was implemented using Python 3 and Flask SQLAlchemy.

This database documentation is intended to explain how to use Improvisor's database API within the context of the functionality we have added. It will not explain how to use SQLAlchemy or Flask SQLAlchemy, documentation for these libraries can be found at the following URLs.

https://docs.sqlalchemy.org/en/latest/orm/tutorial.html - SQLAlchemy

http://flask-sqlalchemy.pocoo.org/2.3/ - Flask SQLAlchemy



User Model

Basic Use - A UserModel object contains all details about an individual user of Improvisor. The arguments required to create a new user are firstname, lastname, email and password. Once an object is created, it must be added to the database in order for it to persist across sessions, this can be done using the save_to_db function. Whenever a new user is created, a directory needs to be created in the filesystem for that user.

```
from improvisor.routes import addDirectory

newUser = UserModel("Michael", "Scott", "dunder@mifflin.com", "ILovePam")
newUser.save_to_db()
addDirectory(newUser.id)
```

Optional arguments include: profileImageLocation and backgroundColour

profileImageLocation is where the user's profile display picture is stored. It is not recommended that this is manually set on object instantiation as there are existing routes that update this value safely.

backgroundColour is the colour of the background shown on the presenter view, this is intended to be changed if there is a particular asset that loses visibility when shown against the default background. This can be changed on instantiation of UserModel or alternatively, there is an endpoint that will change this field.

Class methods - find_by_email, find_by_id

find_by_email takes an email as an argument and returns the corresponding UserModel object. This is not case sensitive.

```
existingUser = UserModel.find_by_email("dunder@mifflin.com")
```

find_by_id works in the same way except takes an integer representing a UserModel object id as input

Important Routes - /signup, /user_settings

/signup: This creates a new user object with the default profile picture and background colour. It is recommended that new users are created through this endpoint as doing so encrypts the password before saving to the database.

/user_settings: This is where the details of a user are modified including the background colour argument.

Relationships - SessionModel, TagModel, AssetModel

To retrieve a list of SessionModel objects representing all the sessions created by the user:

```
existingUser.sessions
```

To retrieve the active session of a user:

```
existingUser.activeSession
```

To retrieve a list of TagModel objects representing tags used by the user across all assets (no duplicates)

```
existingUser.tags
```

To retrieve a list of AssetModel objects representing all assets in the user’s asset pool

```
existingUser.assets
```

Tag Model

Basic Use **-** A TagModel represents details about a tag, a single tag object may be referenced by many assets. The arguments required to create a new tag are tagname and user_id. Tagname is not case sensitive and when the object is created it must be saved to the database. This method of tag creation is not recommended because tag objects with the same tagname will be duplicated in the database.

```
newTag = TagModel("paper", existingUser.id)
newTag.save_to_db()
```

Class Methods **-** find_by_tagName, add_tag

find_by_tagName takes the name of a tag and returns a tag object if the current user that is logged in has created that tag. This is not case sensitive.

```
TagModel.find_by_tagName("paper")
```

add_tag takes the name of a tag and creates a new tag in the database for the current user that is logged in. It is recommended that new tag objects are created this way as this prevents tags with the same name being duplicated. save_to_db() is called in this function so there is no need to call it after using add_tag

```
newTag = TagModel.add_tag("company")
```

Important Routes **-** /assets/new, /assets/<id>/update

/assets/new: Tag objects are created as part of this route when a new asset object is being created. This is dependant on tags being specified when an asset is created by the user

/assets/<id>/update: Tag objects are created as part of this route when an asset object's details are being updated. This is dependant on tags being specified when an asset is being updated by a user. This route will also remove tags from the database if this is what the user has selected to do.

Relationships - UserModel, AssetModel

To retrieve the user object that the tag belongs to

```
companyTag = TagModel.find_by_tagName("company")
userObject = companyTag.user
```

To retrieve a list of AssetModel objects representing all of the assets in the current user's asset pool that are associated with this tag

```
companyTag = TagModel.find_by_tagName("company")
assetsWithTag = companyTag.assets
```

Other **-** remove_from_db

remove_from_db will remove a tag from the database and all assets that contained this tag will no longer be associated with it.

```
companyTag = TagModel.find_by_tagName("company")
companyTag.remove_from_db()
```

Asset Model

Basic Use - An AssetModel object represents details about an asset, it is specific to a user and cannot be accessed by other users. An asset can be used in many different user sessions as well as exist multiple times in a single session. Therefore, it can have many DateModel objects associated with it. The arguments required to create a new asset are assetname, user_id and assettype. An assettype can be either a file or a link

```
existingUser = UserModel.find_by_email("dunder@mifflin.com")
newAsset = AssetModel("door", existingUser.id, "file")
```

At this point, the asset exists in the database but has no associated thumbnail or asset file resource. It is recommended to use the upload function in routes.py to solve this

```
from improvisor.routes import upload

existingAsset = AssetModel.find_by_assetName("door")
upload(existingAsset[0], "path/to/fileResource", "path/to/fileThumbnail")
```

Class Methods - delete_by_assetId, find_by_assetId, find_by_assetName

delete_by_assetId takes an integer representing an asset object's id and removes this asset from this database. Any sessions (current or previous) containing this asset will have the asset removed. This method will also remove asset files from file system.

find_by_assetId takes an integer representing an asset object's id and returns that asset object. This is the best way to find an asset since id is a unique identifier

find_by_assetName takes a string representing an asset object's name and return all assets belonging to the currently logged-in user that match that name

Important Routes - /assets/new, /assets/<id>/update

/assets/new: Asset objects which can be a link or a file are created in this route. The upload function is called from this route and the directory on the filesystem is created for the asset's files. This route will also add any specified tags to the asset.

/assets/update: Given a specific asset object id, this route will change details of an asset object, these include adding tags to the asset, removing tags from the asset, changing the asset's name and updating the asset's resources.

Relationships - UserModel, TagModel, SessionModel, DateModel

To retrieve the user object that the asset belongs to

```
existingAsset = AssetModel.find_by_assetName("door")
existingAsset[0].user
```

To retrieve a list of TagModel objects representing tags that are associated with the asset object

```
existingAsset = AssetModel.find_by_assetId(1)
existingAsset.tags
```

Whenever an asset is added to a session, a DateModel object is created and associated with the asset that was added. To get a list of all DateModel objects associated with the asset

```
existingAsset = AssetModel.find_by_assetId(1)
existingAsset.sessionDates
```

To retrieve a list of all DateModel objects belonging to a particular session

```
existingUser = UserModel.find_by_email("dunder@mifflin.com")
activeSession = existingUser.activeSession
dates = get_dates_for_session(activeSession.sessionNumber)
```

Once the dates for the session are retrieved it is possible to access each asset that is in the session (including assets used more than once)

```
for date in dates:
      print(date.asset.assetname)
```

It is possible to access all sessions that a particular asset is used in

```
existingAsset = AssetModel.find_by_assetId(1)
sessionsWithAsset = existingAsset.get_user_session_appearances()
```

Session Model

Basic Use **-** A SessionModel object represents a user's presentation session. One session will always be an "active" session meaning the session that is currently in use/being added to. The other session objects belonging to a user will be old sessions that can not be edited but can be analysed and exported as a pdf file. A session requires no arguments to be created but will still need to be saved to the database.

```
newSession = SessionModel()
newSession.save_to_db()
```

Class Methods **-** find_all_sessions, find_active_session, find_by_sessionNumber, find_by_sessionId

find_all_sessions will return a list of all sessions that the user has on their account

find_active_session will return the active session

find_by_sessionNumber takes an integer representing the number of a session and returns the session object belonging to the user matching that number

find_by_sessionId takes an integer representing the ID of a session and returns the session object belonging to the user matching that ID.

The difference between session number and session ID: The session ID is the unique identifier of a session across all users, the session number is unique only to the individual user account.

Relationships - UserModel, AssetModel

To retrieve the user object that the session is associated with

```
existingSession = SessionModel.find_by_sessionNumber(1)
sessionOwner = existingSession.user
```

To retrieve a list of AssetModel objects representing the assets that are associated with the session (this list does not contain duplications of assets, when a session has assets presented more than once)

```
existingSession = SessionModel.find_by_sessionNumber(1)
assetObjects = existingSession.assets
```

Other - remove_from_db, add_asset

remove_from_db: removing a session from the database also requires that all the DateModel objects contained in each Asset object that was referenced in the session are deleted. This function deletes these Date objects before deleting the Session object.

add_asset: when an AssetModel object is presented in the session, a DateModel object must be created to mark represent that instance of the Asset within the session. To do this, add_asset takes an AssetObject as argument and the particular tab of the session it is getting added to.

```
existingSession = SessionModel.find_by_sessionNumber(1)
existingAsset = AssetModel.find_by_assetName("door")
existingSession.add_asset(existingAsset, 1)
```

Date Model

Basic Use - An DateModel object captures the details about the addition of an AssetModel object to a specific session and tab. An Asset object may be referenced multiple times in any one session, a unique Data object is created for each individual reference. The DateModel object will contain the time that the presentation showed an asset as well as the Asset object that was shown. The arguments required to create a Date object are asset_id, session_id, user_id and tab. It will also need to be saved to the database.

```
existingAsset = AssetModel.find_by_assetName("door")
existingUser = UserModel.find_by_email("dunder@mifflin.com")
existingSession = existingUser.activeSession
date = DateModel(existingAsset.id, existingSession.id, existingUser.id,
tab=1)
```

Class Methods - find_by_sessionNum

find_by_sessionNum takes an integer representing a Session object's id and returns all DateModel objects that exist for that session

Relationships - AssetModel

To find retrieve the AssetModel object associated with the date object

```
assetObj = date.asset
```