SolidWorks API Project Manager: Server Documentation

CCET 4610 Spring 2023

Developed by: Weston Shakespear

I. Introduction

There are 2 main classes in addition to the main program for the server. The main.py file contains all the functions for api access over the network. These functions take the supplied request data and format it a bit in order to pass to the fileManager class, responsible for saving and recalling files, and then the database class which is responsible for logging all the file information to the main project database and to each individual project's database.

II. main.py

File upload:

```python
@app.route("/file_upload", methods=["POST"])
def file_upload():
    print("uploaddddd")
    request_data = request.get_json()

    json_data = json.loads(json.dumps(request_data))
    up_content = json_data["content"]
    up_checksum = json_data["checksum"]
    up_filepath = json_data["filePath"]
    up_filename = json_data["fileName"]
    up_project = json_data["project"]
    up_relations = json_data["relations"]

    print("relations received----")
    print(up_relations)

    up_remotedir = json_data["remoteDir"]
    up_remotedir = up_remotedir.replace("\\", "/")

    up_user = json_data["user"]


    up_resource = json_data['resource']
    print("preview: ", up_resource)

    if not (fileMan.getStringChecksum(up_content.encode("utf-8")) == up_checksum):
        return "retry"



    if up_resource == "True":

        print("resource sent")
        if not fileMan.saveB64Resource(up_content, up_project + "/resources/", up_filename, overwrite=True):
            return "error saving resource"

        return "good"

    check = db.checkUpload(up_project, up_remotedir, up_filename)

    if check == -1:
        return "bad project"

    else:

        if check == 0:
            # new file
            print("new file")
            filename = up_filename + ".1"
            if not fileMan.saveB64(up_content, up_remotedir, filename):
                return "error saving new file"
            db.addFileEntry(up_filename, up_remotedir, up_checksum, up_user, up_relations)

        else:
            # update file
            version = str(int(check) + 1)
            filename = up_filename + "." + version
            if not fileMan.saveB64(up_content, up_remotedir, filename):
                print("broekn")
                return "error saving updated file"

            print(db.updateFileEntry(up_filename, up_remotedir, up_checksum, up_user, up_relations, version))
            print("update file", version)

    return "good"
```

When the desktop application uploads a file to the server this is the function that gets called with that request information. The file being uploaded is accompanied by the project,

directory, and relation information which needs to be extracted from the request. This metadata is used to place the received file in the correct location and allows for the project database to be updated.

The uploaded file is sent with a checksum, a number that has been specifically calculated off of the binary data from the file. This checksum is checked using the fileManager class method checkUpload on line 104. This method checks for the type of file and then checks that the checksum matches. The returned value is used for the following logic.

This function then tests to find out if this is a new file, an updated version of a file, or a resource file for the previews. Depending on which case this is the needed methods are called from the fileManager and database classes.

File download:

```
132  @app.route("/file_download", methods=["GET"])
133  def file_download():
134      argsImmutable = request.args
135      args = argsImmutable.to_dict()
136
137      dw_filename = args["fileName"]
138      dw_project = args["project"]
139
140      print(dw_filename)
141      print(dw_project)
142
143      filePath = db.getLatestPath(dw_project, dw_filename)
144
145      print("path: ", filePath)
146
147      data = fileMan.getB64(filePath)
148
149
150      return json.dumps(data)
```

When a file needs to be downloaded it's a pretty simple ordeal. The client only needs to supply the filename and the project in JSON format. That is deserialized on lines 137 and 138. Line 143 calls the database class method which returns the filename for the most recent version of the file. This filename is given to the fileManager class method getB64 which then encodes the files and allows for this to be given to the client.

File Previews:

```
154  @app.route("/preview/<project>/<file>")
155  def preview(project, file):
156      print(project, file)
157
158      filename = data_location + project + '/resources/' + file
159      print(filename)
160      return send_file(filename + ".BMP", mimetype='image/gif')
```

When This function is in charge of supplying the preview image to the client program. It was important to host the previews so that no files needed to be downloaded in order to view the preview. The URL is split on line 154 with a special Flask call and the file is recalled from the disk and sent back to the client as a hosted image.

III. fileManager.py

Saving the encoded file:

```
59      def saveB64(self, input_data, path, name, overwrite=False):
60
61
62          # decode the data
63          out_bytes = self.decodeString(input_data)
64          save_path = self.buildSavePath(path + "/", name)
65
66          self.log.debug("Attempting to save " + name + " in " + save_path)
67
68          if (os.path.exists(save_path) and (not overwrite)):
69              self.log.error(name + " already exists");
70          elif not(os.path.exists(self.data_path + path.split("/")[0])):
71              print("Project doesn't exist");
72          else:
73              print()
74              self.createNeededDirs(save_path)
75
76              try:
77                  with open(save_path, "wb") as file:
78                      file.write(out_bytes)
79                  self.log.debug("Saving " + path + " [SUCCESS]")
80                  return True
81
82              except Exception as e:
83                  self.log.debug("Saving " + path + " [ERROR]" + e)
84          return False
```

The saveB64 method is used to save a string of bytes locally as a file. This first decodes the string into a list of bytes and attempts to save it to proper location with the version appeneded to the filename. Whether or not to overwrite the current file can be supplied and this will be used in the future to force an overwrite from the client program.

Reading and encoding local files:

```
104     def getB64(self, path):
105
106         if (os.path.exists(path)):
107             with open(path, "rb") as file:
108                 data_bytes = file.read()
109
110             dataB64 = base64.b64encode(data_bytes)
111             checksum = self.getStringChecksum(dataB64)
112             print(checksum)
113
114             data = {
115                 "content": dataB64.decode(),
116                 "checksum": checksum
117             }
118             return data
119
120         else:
121             return "null"
```

The getB64 method is used to send a file to the client program. The file is first opened on line 107 and then read as bytes on the following line. The file is then encoded into a string and

then the checksum is calculated. The file string and checksum are then packaged using JSON and returned to the main program.

IV. database.py

Project database creation:

```
130     def createProjectDB(self, project):
131         path = self.data_location + project + '/' + 'data.db'
132 
133         with sq.connect(path) as conn:
134             files_query = '''CREATE TABLE files (filename TEXT,
135                                                  owner TEXT,
136                                                  created TEXT,
137                                                  version TEXT,
138                                                  relations TEXT,
139                                                  last_user TEXT,
140                                                  last_modified TEXT,
141                                                  path TEXT,
142                                                  checksum TEXT)'''
143             conn.execute(files_query)
```

When a new project is created this method creates a new database on lines 131 and 133 and then creates a new table called files on line 134.

Reading project information:

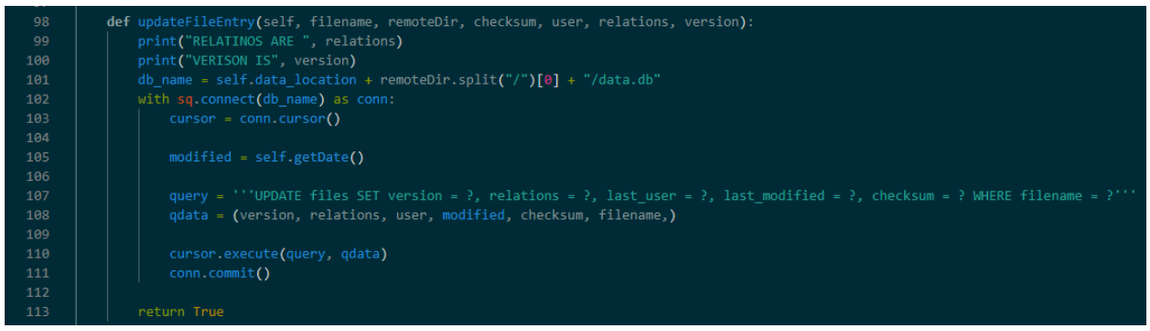
```
199     def readProjects(self):
200         with sq.connect(self.db_location) as conn:
201             cur = conn.cursor()
202 
203             query = '''SELECT * FROM projects'''
204             cur.execute(query)
205 
206             self.tree = {}
207 
208             for entry in cur.fetchall():
209                 items = list(entry)
210                 row = {
211                     'name': items[0],
212                     'owner': items[1],
213                     'created': items[2],
214                     'head': items[3]
215                 }
216                 self.tree[row['name']] = {}
217                 # self.projects[row['name']] = row;
218                 project_db = self.data_location + row['head'];
219 
220                 # self.files[row['name']] = self.readFiles(project_db)
221                 self.readFiles(project_db, row['name'])
222 
223 
224     def readFiles(self, project, pname):
225         db_path = project + "/data.db"
226         with sq.connect(db_path) as conn:
227             cur = conn.cursor()
228 
229             query = '''SELECT * FROM files'''
230             cur.execute(query)
231 
232             for entry in cur.fetchall():
233                 items = list(entry)
234 
235                 row = {
236                     'filename': items[0],
237                     'owner': items[1],
238                     'created': items[2],
239                     'version': items[3],
240                     'relations': items[4],
241                     'last_user': items[5],
242                     'last_modified': items[6],
243                     'path': items[7],
244                     'checksum': items[8]
245                 }
246                 self.tree[pname][row['filename']] = row
247 
```

WEvery time the client program asks for a list of the current projects and files this first method ”readProjects” is called. This gets all the rows from the project database on line 203 and then iterates through each one of them, giving the project name and database to the second method defined on line 224.

The “readFiles” method does pretty much the same thing as the “readProjects” loop, except that it opens the database for each project and gets all of those rows. This loop iterates over those and

adds all the information to the database classes' own internal variable “tree” which hold the current file information for all projects.

Updating file entries:



```
98  def updateFileEntry(self, filename, remoteDir, checksum, user, relations, version):
99      print("RELATINOS ARE ", relations)
100     print("VERISON IS", version)
101     db_name = self.data_location + remoteDir.split("/")[0] + "/data.db"
102     with sq.connect(db_name) as conn:
103         cursor = conn.cursor()
104
105         modified = self.getDate()
106
107         query = '''UPDATE files SET version = ?, relations = ?, last_user = ?, last_modified = ?, checksum = ? WHERE filename = ?'''
108         qdata = (version, relations, user, modified, checksum, filename,)
109
110         cursor.execute(query, qdata)
111         conn.commit()
112
113     return True
```

While this is some pretty simple SQL code, it does show how some of the database management is done. The query is defined on line 107 with the variables supplied on line 108 and then given to the database. This query will replace the version, relations, last_user, last_modified, and checksum value on the row in the database that matches the supplied filename.