File - /home/tcassar/projects/settle/src/__init__.py

```python
# coding=utf-8
import logging

logging.basicConfig(filename='./log')
```

File - /home/tcassar/projects/settle/src/client/cli.py

```python
# coding=utf-8

import click

import src.client.client as client


@click.group()
def settle():
    ...


# |--------|
# |  USER  |
# |--------|


@click.option("--pub_key", prompt="Path to RSA key", type=click.Path())
@click.option("--password", prompt=True, hide_input=True, confirmation_prompt=True)
@click.option("--email", prompt=True)
@click.option("--name", prompt="Full Name")
@settle.command()
def register(
    name,
    email,
    password,
    pub_key,
):
    client.register(name, email, password, pub_key)


@click.argument("email")
@settle.command()
def whois(email):
    client.whois(email)


@click.option("-g", "--groups", flag_value="groups", default=False)
```

```
39 @click.option("-t", "--transactions", flag_value="
   transactions", default=False)
40 @click.option("--email", prompt=True)
41 @settle.command()
42 def show(transactions, groups, email):
43     """Shows all of your open transactions / groups
   along with IDs"""
44     client.show(transactions, groups, email)
45
46
47 @click.option("--password", prompt=True, hide_input=
   True)
48 @click.option("--email", prompt=True)
49 @click.argument("key_path")
50 @click.argument("transaction_id")
51 @settle.command()
52 def sign(transaction_id, key_path, email, password):
53     """Signs a transaction"""
54     client.sign(transaction_id, key_path, email,
   password)
55
56
57 @click.option("-g", "--group", default=0)
58 @click.option("-t", "--transaction", default=0)
59 @settle.command()
60 def verify(group, transaction):
61     """Will verify a transaction if given a
   transaction ID or an entire group if given a group
   ID"""
62     client.verify(group, transaction)
63
64
65 @click.option("--password", prompt=True, hide_input=
   True, confirmation_prompt=True)
66 @click.option("--name", prompt=True)
67 @settle.command(name="new-group")
68 def new_group(name, password):
69     client.new_group(name, password)
70
71
72 @click.option(
```

```
 73     "--group_password",
 74     prompt="Group Password",
 75     hide_input=True,
 76 )
 77 @click.option(
 78     "--password",
 79     prompt="Your password",
 80     hide_input=True,
 81 )
 82 @click.option("--email", prompt=True)
 83 @click.argument("group_id")
 84 @settle.command()
 85 def join(email, password, group_id, group_password
    ):
 86     """Joins a group given an ID"""
 87     client.join(email, password, group_id,
    group_password)
 88
 89
 90 @click.option("--password", prompt="Group Password"
    , hide_input=True)
 91 @click.argument("group_id")
 92 @settle.command()
 93 def simplify(group_id, password):
 94     """Simplifies debt of a group"""
 95     client.simplify(group_id, password)
 96
 97
 98 @click.option("--password", prompt=True, hide_input
    =True)
 99 @click.option("--email", prompt="Your email")
100 @click.option("--group", "-g", prompt=True)
101 @click.option("--reference", prompt=True)
102 @click.option("--amount", prompt="Amount (in GBP)")
103 @click.option("--dest_email", prompt="Email of
    payee")
104 @settle.command(name="new-transaction")
105 def new_transaction(email, password, dest_email,
    amount, group, reference):
106     """Generates a new transaction"""
107     client.new_transaction(email, password,
```

```
107 dest_email, amount, group, reference)
108
109
110 @click.option("--email", prompt=True)
111 @click.option("--group_id", prompt="Group ID")
112 @settle.command(name="show-group")
113 def show_group(email, group_id):
114     client.group_debt(group_id, email)
115
116
117 @click.option("--password", prompt=True, hide_input
    =True)
118 @click.option('--email', prompt=True)
119 @click.argument('transaction')
120 @settle.command()
121 def tick(email, transaction, password):
122     """Ticks off a transaction as settled up in the
     real world"""
123     if transaction is None:
124         click.secho('Please provide a transaction
    with the -t flag (--help for help)', fg='red')
125         return
126
127     client.tick(email, password, transaction)
```

File - /home/tcassar/projects/settle/src/client/client.py

```python
# coding=utf-8
import copy
import sys

import click
import json
import requests

import src.client.cli_helpers as helpers
import src.crypto.keys as keys
import src.server.models as models
import src.server.schemas as schemas
import src.transactions.transaction as trn
from src.client.cli_helpers import show_transactions

trap = helpers.trap

###############
#  ANCILLARY  #
###############


@trap
def register(
    name,
    email,
    password,
    pub_key,
):
    """Register to settle, using email, passwd, and an RSA public key"""

    # extract and store modulus and pub exp as bytes, ready for db
    ldr = keys.RSAKeyLoader()
    try:
        ldr.load(pub_key)
        ldr.parse()
        pub_key: keys.RSAPublicKey = keys.RSAPublicKey(ldr)
    except keys.RSAParserError as rsa_err:
```

```
39          click.secho(
40              f"Failed to create account - issue with
   given RSA key;\n{rsa_err}",
41              fg="red",
42              bold=True,
43          )
44          return
45
46      password: str = helpers.hash_password(password)
47
48      usr = models.User(
49          name,
50          email,
51          hex(pub_key.n),
52          hex(pub_key.e),
53          password,
54      )
55
56      # build json repr of object
57      schema = schemas.UserSchema()
58      usr_as_json = schema.dump(usr)
59
60      response = requests.post(helpers.url("user"),
   json=usr_as_json)
61      try:
62          helpers.validate_response(response)
63      except helpers.InvalidResponseError:
64          print(response)
65          click.secho(f"Failed to create account under
    email {email}", fg="yellow")
66          return
67
68      click.secho(f"Account created successfully", fg=
   "green")
69      click.echo(f"{usr}")
70
71
72  @trap
73  def whois(email):
74      """gives your name, email, public key numbers"""
75      usr_response: requests.Response = requests.get(
```

```
 75 helpers.url(f"user/{email}"))
 76     helpers.validate_response(usr_response)
 77
 78     # build a user from received data
 79
 80     schema = schemas.UserSchema()
 81     usr = schema.load(usr_response.json())
 82     click.secho(f"\nFound user with email {email}:\
    n", fg="green")
 83     click.secho(str(usr))
 84
 85
 86 @trap
 87 def show(transactions, groups, email):
 88     """Shows all of your open transactions / groups
     along with IDs"""
 89
 90     # note: flags are None or True for some
    godforsaken reason
 91
 92     # show both if no flags
 93     if not transactions and not groups:
 94         transactions = groups = True
 95
 96     if groups:
 97         click.secho("Groups that you are a member
    of:\n", fg="blue")
 98
 99         # receive list of groups JSON;
100         try:
101             groups_data = requests.get(helpers.url(
    f"group/{email}"))
102         except helpers.ResourceNotFoundError as ire
    :
103             raise helpers.ResourceNotFoundError(
104                 f"Problem fetching your group...\n{
    ire}"
105             )
106
107         group_objs: list[models.Group] = []
108         for group in groups_data.json()["groups"]:
```

```
109             group_objs.append(
110                 models.Group(group["id"], group["
    name"], group["password"])
111             )
112 
113         groups = models.GroupList(group_objs)
114 
115         click.echo(str(groups))
116         # type: ignore
117 
118     if transactions:
119 
120         # transaction schema
121 
122         # receive list of transactions
123         try:
124             transactions_data = requests.get(
    helpers.url(f"/transaction/{email}"))
125             helpers.validate_response(
    transactions_data)
126         except helpers.ResourceNotFoundError as ire
    :
127             raise helpers.ResourceNotFoundError(
128                 f"Problem with fetching your
    transactions...\n{ire}"
129             )
130         show_transactions(transactions_data)
131 
132 
133 @trap
134 def join(email, password, group_id, group_password
    ):
135 
136     # 1: verify user
137     helpers.auth_usr(email, password)
138 
139     # 2: verify group
140     helpers.auth_group(group_id, group_password)
141 
142     # 3: post to groups
143     group = requests.post(helpers.url(f"group/{
```

```
143 group_id}/{email}"))
144     helpers.validate_response(group)
145     click.secho(f"Successfully joined group {
    group_id}", fg="green")
146 
147 
148 @trap
149 def new_group(name, password):
150     schema = schemas.GroupSchema()
151     # note: 0 is placeholder, will be overwritten
    by db
152     group = models.Group(0, name, helpers.
    hash_password(password))
153 
154     as_json = schema.dump(group)
155     response = requests.post(helpers.url("group"),
    json=as_json)
156 
157     try:
158         helpers.validate_response(response)
159     except helpers.InvalidResponseError as ire:
160         raise helpers.InvalidResponseError(f"Couldn
    't create new group...\n{ire}")
161 
162     click.secho(response.text, fg="green")
163     click.secho("You can join this group with `
    settle join`")
164 
165 
166 ###############
167 # FUNCTIONAL #
168 ###############
169 
170 ###############
171 # FUNCTIONAL #
172 ###############
173 
174 
175 @trap
176 def new_transaction(email, password, dest_email,
    amount, group, reference):
```

```
177     # 1. verify src credentials
178     helpers.auth_usr(email, password)
179 
180     # 2. get users as models.User objects
    destination exists
181     src = helpers.get_user(email)
182     dest = helpers.get_user(dest_email)
183 
184     # convert amount into pence
185     amount = float(amount)
186     amount *= 100
187     if type(amount) == float:
188         amount //= 1
189         amount = int(amount)
190 
191     # build key objects
192     src_key_ldr = keys.RSAKeyLoaderFromNumbers()
193     dest_key_ldr = keys.RSAKeyLoaderFromNumbers()
194 
195     src_key_ldr.load(n=int(src.modulus, 16), e=int(
    src.pub_exp, 16))
196     dest_key_ldr.load(n=int(dest.modulus, 16), e=
    int(dest.pub_exp, 16))
197 
198     src_key = keys.RSAPublicKey(src_key_ldr)
199     dest_key = keys.RSAPublicKey(dest_key_ldr)
200 
201     # build transaction
202 
203     transaction = trn.Transaction(
204         src=src.id,
205         dest=dest.id,
206         amount=amount,
207         src_pub=src_key,
208         dest_pub=dest_key,
209         reference=reference,
210         group=group,
211     )
212 
213     # build schema
214     trn_schema = schemas.TransactionSchema()
```

```
215
216     # post to server
217     response = requests.post(
218         helpers.url("transaction"), json=trn_schema
    .dump(transaction)
219     )
220
221     try:
222         helpers.validate_response(response)
223     except helpers.InvalidResponseError as ire:
224         raise helpers.InvalidResponseError(f"Failed
     to add transaction\n{ire}")
225
226     click.secho(f"Transaction generated with ID={
    response.json()}", fg="green")
227     click.echo(f"Sign with `settle sign {response.
    json()}`")
228
229
230 @trap
231 def simplify(group_id, password):
232     """Will settle the group; can be done by anyone
     at anytime;
233     everyone signs newly generated transactions if
    new transactions are generated"""
234
235     helpers.auth_group(group_id, password)
236
237     response = requests.post(helpers.url(f"/
    simplify/{group_id}"))
238
239     try:
240         helpers.validate_response(response)
241
242     except helpers.ResourceNotFoundError as ire:
243         raise helpers.ResourceNotFoundError(f"
    Problem settling group... \n{ire}")
244
245     except helpers.NoChanges as nc:
246         raise nc
247
```

```
248
249 @trap
250 def sign(transaction_id, key_path, email, password
    ):
251     """Signs a transaction given an ID and a path
    to key"""
252
253     # auth user
254     helpers.auth_usr(email, password)
255
256     # load private key
257
258     ldr = keys.RSAKeyLoader()
259     try:
260         ldr.load(key_path)
261         ldr.parse()
262         key: keys.RSAPrivateKey = keys.
    RSAPrivateKey(ldr)
263     except keys.RSAParserError as rsa_err:
264         click.secho(
265             f"Failed to sign transaction - issue
    with given RSA key;\n{rsa_err}",
266             fg="red",
267             bold=True,
268         )
269         return
270
271     # pull signable transaction
272     response = requests.get(helpers.url(f"
    transaction/signable/{transaction_id}"))
273
274     try:
275         helpers.validate_response(response)
276     except helpers.InvalidResponseError as ire:
277         raise helpers.InvalidResponseError(f"Failed
     to fetch group data\n{ire}")
278
279     transaction: trn.Transaction = schemas.
    TransactionSchema().make_transaction(
280         response.json()
281     )
```

```
282
283     # determine origin
284     usr_response: requests.Response = requests.get(
    helpers.url(f"user/{email}"))
285     helpers.validate_response(usr_response)
286
287     # build a user from received data
288     usr = schemas.UserSchema().make_user(
    usr_response.json())
289
290     # validate key in provided path against key
    from db
291     key_as_str = f"n={key.n},\ne={key.e}"
292
293     if usr.id == transaction.src:
294         origin = "src"
295         if transaction.src_pub.strip() !=
    key_as_str.strip():
296             raise helpers.AuthError(
297                 "Private key provided does not
    match the listing in the db"
298             )
299         else:
300             click.secho("\tKey validated against
    server")
301
302     elif usr.id == transaction.dest:
303         origin = "dest"
304         if transaction.dest_pub.strip() !=
    key_as_str.strip():
305             raise helpers.AuthError(
306                 "Private key provided does not
    match the listing in the db"
307             )
308         else:
309             click.secho("\tKey validated against
    server", fg="green")
310
311     else:
312         raise helpers.ResourceNotFoundError(
313             "Email provided doesn't match any of
```

```
313 the users in the transaction"
314         )
315 
316     try:
317         # convert keys of sigs to ints to enable
    overwrite protection
318         as_strs = copy.deepcopy(transaction.
    signatures)
319         transaction.signatures = {}
320         for s_key, s_val in as_strs.items():
321             if type(s_key) is str:
322                 transaction.signatures[int(s_key
    )] = s_val
323 
324         transaction.sign(key, origin=origin)
325         click.secho("\tsuccessfully signed
    transaction", fg="green")
326     except trn.TransactionError as te:
327         raise helpers.AuthError(f"Failed to sign
    transaction: {transaction.ID}\n{te}")
328 
329     # convert signature to hex
330     # todo: add sig_as_hex to transaction obj
331     sig_hex = hex(int.from_bytes(transaction.
    signatures[usr.id], sys.byteorder))
332 
333     # patch signature
334     schema = schemas.SignatureSchema()
335     signature = models.Signature(transaction_id,
    sig_hex, origin)
336     rep = requests.patch(helpers.url("transaction/
    auth/"), json=schema.dump(signature))
337 
338     if rep.status_code == 201:
339         click.secho("\tSuccessfully appended
    signature in database!", fg="green")
340 
341 
342 @trap
343 def verify(groups, transactions: int):
344     """Will show you the transactions of a group,
```

```
344 or verify a transaction passed in by ID"""
345
346     if not groups and not transactions:
347         click.secho("Please provide a groups ID or
    transaction ID", fg="yellow")
348         return
349
350     if transactions:
351         response = requests.get(helpers.url(f"
    transaction/auth/{transactions}"))
352
353         try:
354             helpers.validate_response(response)
355         except helpers.InvalidResponseError as ire:
356             raise helpers.InvalidResponseError(f"
    Error in getting transaction, {ire}")
357
358         schema = schemas.PrettyTransactionSchema()
359         schema.make_pretty_transaction(response.
    json()).secho()
360
361     if groups:
362         try:
363             transactions_data = requests.get(
    helpers.url(f"group/debt/{groups}"))
364             helpers.validate_response(
    transactions_data)
365         except helpers.ResourceNotFoundError as ire
    :
366             raise helpers.ResourceNotFoundError(
367                 f"Problem with fetching your
    transactions...\n{ire}"
368             )
369
370         pretty_schema = schemas.PrettyListSchema()
371         # print(transactions_data.json())
372         pretty_list: models.PrettyList =
    pretty_schema.make_pretty_list(
373             transactions_data.json()
374         )
375         trn_schema = schemas.
```

```
375 PrettyTransactionSchema()
376
377         for trn in pretty_list.src_list +
    pretty_list.dest_list:
378             trn_schema.make_pretty_transaction(trn
    ).secho()
379
380
381 @trap
382 def tick(email: str, password: str, t_id: int):
383     # auth user
384     helpers.auth_usr(email, password)
385
386     rep = requests.patch(helpers.url(f"transaction/
    settle/{t_id}"), json=json.dumps({'email': email},
    indent=4))
387     helpers.validate_response(rep)
388     click.secho(f'Transaction {t_id} marked as
    settled!', fg='green')
389
390
```

File - /home/tcassar/projects/settle/src/client/__init__.py

```python
# coding=utf-8
import logging

logging.basicConfig(level=logging.CRITICAL)

```

File - /home/tcassar/projects/settle/src/client/cli_helpers.py

```python
# coding=utf-8
import click
import requests

from src.crypto import hashes as hasher
from src.server import models as models, schemas as schemas

SERVER = "http://127.0.0.1:5000/"


class AuthError(Exception):
    ...


class ResourceNotFoundError(Exception):
    """Resource could not be found on the server (404)"""


class InvalidResponseError(Exception):
    """Requested action was understood but not allowed (403 / 409)"""


class ServerError(Exception):
    ...


class NoChanges(Exception):
    ...


def hash_password(password) -> str:
    return str(hasher.Hasher(password.encode(encoding="utf8")).digest().h)


def url(query: str) -> str:
    return SERVER + query

```

```
38
39 def validate_response(response: requests.Response
   ) -> None:
40     """Raises InvalidResponseError if response is
   invalid"""
41     e = response.text
42     if response.status_code == 404:
43         raise ResourceNotFoundError(
44             f"ERROR: Could not find requested
   resource on server, {e}"
45         )
46     elif response.status_code == 409:
47         raise ResourceNotFoundError(
48             f"The resource that you are trying to
   create already exists, {e}"
49         )
50     elif response.status_code == 403:
51         raise ResourceNotFoundError(f"This action
   was not allowed by the server, {e}")
52     elif response.status_code // 100 == 5:
53         raise ServerError(f"Error {response.
   status_code}: {response.json()['message']}")
54     elif response.status_code == 202:
55         raise NoChanges(response.text)
56
57
58 def _auth(resource: str, password: str):
59     usr_response: requests.Response = requests.get(
   url(resource))
60     try:
61         validate_response(usr_response)
62     except ResourceNotFoundError:
63         raise ResourceNotFoundError(
64             f"No {resource.split('/')[0]} with
   identifier {resource.split('/')[1]} found"
65         )
66
67     # build a user from received data
68
69     rep = usr_response.json()
70
```

```
 71     if rep["password"] != hash_password(password):
 72         raise AuthError("Password Incorrect")
 73
 74
 75 def auth_usr(email: str, password: str):
 76     """Authorises user, raises AuthError if fails
    """
 77     return _auth(f"user/{email}", password)
 78
 79
 80 def auth_group(group_id: int, password: str):
 81     return _auth(f"group/{group_id}", password)
 82
 83
 84 def get_user(email: str) -> models.User:
 85     usr_rep = requests.get(url(f"user/{email}"))
 86
 87     try:
 88         validate_response(usr_rep)
 89     except ResourceNotFoundError:
 90         raise ResourceNotFoundError(f"No user
    associated with email {email}")
 91
 92     usr = usr_rep.json()
 93
 94     return models.User(
 95         usr["name"], usr["email"], usr["modulus"],
    usr["pub_exp"], "", usr["id"]
 96     )
 97
 98
 99 def trap(func) -> object:
100     """
101     Decorator to handle errors on these functions
102     """
103
104     def inner(*args, **kwargs):
105         try:
106             func(*args, **kwargs)
107
108         except AuthError as ae:
```

```
109             click.secho("Authorisation Error;
    aborting...", fg="red")
110             click.secho(ae, fg="red")
111
112         except ResourceNotFoundError as nre:
113             click.secho(
114                 f"{nre}",
115                 fg="yellow",
116             )
117
118         except ServerError as se:
119             click.secho(se, fg="red")
120
121         except NoChanges as nc:
122             click.secho(f"No changes were made\n{nc
    }", fg="yellow")
123
124         except requests.exceptions.ConnectionError:
125             click.secho('ERROR: Could not connect
    to server', fg='red')
126
127     return inner
128
129
130 def show_transactions(transactions_data: requests.
    Response):
131     try:
132         unverified_running = 0.00
133         verified_running = 0.00
134         for pretty in transactions_data.json()["
    src_list"]:
135
136             click.secho(
137                 f'\nYou owe {pretty["other"]} £{
    round(pretty["amount"] / 100, 2):.2f}',
138                 fg="yellow",
139             )
140
141             click.secho(
142                 f'\nReference: {pretty["time"]}'
143                 + f'\nAgreed upon at {pretty["
```

```
143 reference"]}'
144                     + f'ID: {pretty["id"]}'
145             )
146 
147             if pretty["verified"] == 1:
148                 click.secho("Verified", fg="green")
149                 verified_running += pretty["amount"
    ]
150             else:
151                 click.secho("Unverified", fg="red")
152                 unverified_running += pretty["
    amount"]
153 
154         for pretty in transactions_data.json()["
    dest_list"]:
155             click.secho(
156                 f'\n{pretty["other"]} owes you £{
    round(pretty["amount"] / 100, 2):.2f}',
157                 fg="yellow",
158             )
159 
160             click.secho(
161                 f'\nReference: {pretty["time"]}'
162                 + f'\nAgreed upon at {pretty["
    reference"]}'
163                 + f'ID: {pretty["id"]}'
164             )
165 
166             if pretty["verified"] == 1:
167                 click.secho("Verified", fg="green")
168                 verified_running -= pretty["amount"
    ]
169             else:
170                 click.secho("Unverified", fg="red")
171                 unverified_running -= pretty["
    amount"]
172 
173         click.echo("----------\n")
174 
175         unverified_running = round(
    unverified_running / 100, 2)
```

```
176         verified_running = round(verified_running
     / 100, 2)
177 
178         if verified_running > 0:
179             click.secho(f"You owe a total of £{
    verified_running:.2f}", fg="red")
180         elif verified_running < 0:
181             click.secho(
182                 f"You are owed a total of £{
    verified_running * -1 :02}", fg="blue"
183             )
184         else:
185             click.secho(f"You owe and are owed
    nothing; all debts settled", fg="green")
186 
187         if unverified_running > 0:
188             click.secho(
189                 f"Your unverified totals => you owe
     £{unverified_running:.2f}",
190                 fg="yellow",
191             )
192         elif unverified_running < 0:
193             click.secho(
194                 f"Your unverified totals => you are
     owed £{unverified_running * -1 :02}",
195                 fg="yellow",
196             )
197         else:
198             click.secho(f"Your unverified totals
     => all debts settled", fg="yellow")
199 
200     except TypeError as te:
201         if transactions_data.json() is None:
202             click.secho("No open transactions", fg=
    "green")
203             click.echo(te)
204 
205 
206 def show_group(transaction_data: requests.Response
    ):
207     schema = schemas.PrettyListSchema()
```

```
208     pretty_list: models.PrettyList = schema.
    make_pretty_list(transaction_data.json())
209     print(pretty_list)
210
```

```python
# coding=utf-8
"""
RSA Sign/Verify classes
"""

from __future__ import annotations

import sys
from abc import ABC
from typing import TYPE_CHECKING

from src.crypto import keys

if TYPE_CHECKING:
    pass


class DecryptionError(Exception):
    """Failed to decrypt"""


class SigningError(Exception):
    ...


class RSA(ABC):
    """RSA methods; ABC so is never instantiated; just used for namespacing"""

    @staticmethod
    def check_private_key(key) -> None:
        if type(key) == keys.RSAPublicKey:
            raise DecryptionError("Cannot decrypt with a public key")

    @staticmethod
    def int_to_bytes(n: int) -> bytes:
        return int.to_bytes(n, length=n.bit_length(), byteorder=sys.byteorder).rstrip(
            b"\x00"
        )
```

```
39
40      @staticmethod
41      def bytes_to_str(b: bytes) -> str:
42          return b.decode("utf8").replace("\x00", "")
43
44      @staticmethod
45      def encrypt(message: bytes, publicKey: keys.
    RSAPublicKey) -> bytes:
46          message = int.from_bytes(message, sys.
    byteorder)
47          cipher = pow(message, publicKey.e, publicKey
    .n)
48
49          return RSA.int_to_bytes(cipher)
50
51      @staticmethod
52      def naive_decrypt(ciphertext: bytes, privateKey
    : keys.RSAPrivateKey) -> bytes:
53          # TODO: CRT decryption
54
55          RSA.check_private_key(privateKey)
56
57          ciphertext = int.from_bytes(ciphertext, sys.
    byteorder)
58          plaintext = pow(ciphertext, privateKey.d,
    privateKey.n)
59
60          return RSA.int_to_bytes(plaintext)
61
62      @staticmethod
63      def sign(msg: bytes, key: keys.RSAPrivateKey
    ) -> bytes:
64          # check we have a private key
65
66          # extract integer if sign given bytes
67
68          RSA.check_private_key(key)
69
70          msg = int.from_bytes(msg, sys.byteorder)
71          cipher = pow(msg, key.d, key.n)  # type:
    ignore
```

```
72
73          return RSA.int_to_bytes(cipher)
74
75      @staticmethod
76      def inv_sig(sig: bytes | int, key: keys.
    RSAPublicKey) -> bytes:
77          """Will produce what was originally fed
    into sign() using public key
78          used in verifying; if verified, should
    generate hash of obj"""
79          if type(sig) is bytes:
80              sig: int = int.from_bytes(sig, sys.
    byteorder)
81
82          de_sig = pow(sig, key.e, key.n)
83
84          return RSA.int_to_bytes(de_sig)
85
```

File - /home/tcassar/projects/settle/src/crypto/keys.py

```python
# coding=utf-8
"""Interface to all RSA encrypt / decrypt functions
"""

import os
import os.path
import re
import subprocess
from dataclasses import dataclass, field


class RSAKeyLoaderFromNumbers:
    ...


class RSAKeyError(Exception):
    """Problem with RSA Key"""


class RSAParserError(Exception):
    """Error in parsing file containing key"""


class RSAPublicKeyError(Exception):
    ...


@dataclass
class RSAKeyLoader:
    """Will load a key from a private key file"""

    # initialise what we need, don't pass in anything at instantiation
    lookup: dict[str, int] = field(default_factory=lambda: {})
    key: None | str = None

    def load(self, path_to_private_key: str) -> str:
        """
        Loads PEM private key from file in file path;
```

```
38          """
39
40          # will only attempt to load if file exists
41          # if openssl rsa doesn't like file report as
     incorrect format
42
43          try:
44              if not os.path.exists(
    path_to_private_key):
45                  raise RSAParserError(
46                      f"File not found at current path
    : \n{path_to_private_key}"
47                  )
48              key = str(
49                  subprocess.check_output(
50                      f"openssl rsa -noout -text < {
    path_to_private_key}", shell=True
51                  )
52              )
53          except subprocess.CalledProcessError:
54              raise RSAParserError("File not in PEM
    private key format")
55
56          # strip into long easily parsable str; =>
    remove spaces, newlines, colons
57          key = key.replace(" ", "").replace("\\n", ""
    ).replace(":", "")
58
59          # converted from bytes so remove b' ' with
    slice
60          key = key[2:-1]
61
62          # split after title section, discard info
63          # pub exp is given in bin in text format;
    breaks splitting header with )
64          key = re.sub(r"\(0x[0|1]*\)", "", key)  #
    matches of the form (0x[0|1]*)
65          head, key = key.split(")")
66
67          if head != "RSAPrivate-Key(2048bit,2primes":
68              raise RSAParserError("File not in
```

```
68 correct format")
69
70         self.key = key
71         return key
72
73     def parse(self, keys: str | None = None) ->
   None:
74         """Given loaded SSL info, parses and
   populates n, d, e, p, q"""
75
76         def hexstr_to_int(hexstr: str) -> int:
77             # deals with exception pub exp which is
    always a decimal
78             return int(hexstr) if re.fullmatch("[0-
   9]+", hexstr) else int(hexstr, 16)
79
80         # use local key if key not given
81         if keys is None:
82             keys: str = self.key
83             if self.key is None:
84                 raise RSAParserError("Key has not
   been loaded")
85
86         # set up delimiters; tells us to split when
    parsing
87         delimiters = ["n", "e", "d", "p", "q", "
   exp1", "exp2", "crt_coef"]
88
89         # split into a list along delimiters
90         keys = re.sub(
91             "modulus|publicExponent|privateExponent
   |prime1|prime2|exponent1|exponent2|coefficient",
92             " ",
93             keys,
94         )
95         keys: map = map(hexstr_to_int, keys.split
   ())
96
97         # combine to dict
98         self.lookup = {label: key for label, key in
    zip(delimiters, keys)}
```

```
 99
100
101 @dataclass
102 class RSAPublicKey:
103     def __init__(self, loader: RSAKeyLoader |
    RSAKeyLoaderFromNumbers):
104         self.lookup = loader.lookup
105
106     def __str__(self):
107         return f"n={self.n},\ne={self.e}\n"
108
109     def __getattr__(self, item: str) -> int:
110         """Redefine getattr so that will only give
    n and e"""
111         if item == "n" or item == "e":
112             return self.lookup[item]
113         else:
114             raise RSAPublicKeyError("Requested
    attribute not part of the Public Key")
115
116     def _exists(self, item) -> bool:
117         """Returns an attribute if it exists else
    raise an RSAKeyError"""
118         if self.lookup is not None:
119             try:
120                 return not not self.lookup[item]
121             except KeyError:
122                 raise RSAKeyError(
123                     "Requested attribute not found
    ; have you parsed a key?"
124                 )
125
126
127 @dataclass
128 class RSAKeyLoaderFromNumbers:  # type: ignore
129     lookup: dict[str, int] = field(default_factory=
    lambda: {})
130
131     def load(self, n: int, e: int, d: int = 0) ->
    None:
132         self.lookup["n"] = n
```

```
133         self.lookup["e"] = e
134         if d:
135             self.lookup["d"] = d
136 
137     def pub_key(self) -> RSAPublicKey:
138         return RSAPublicKey(self)
139 
140     def priv_key(self):
141         return RSAPrivateKey(self)
142 
143 
144 class RSAPrivateKey(RSAPublicKey):
145     def __str__(self):
146         return f"n={self.n},\ne={self.e},\nd={self.
    d}"
147 
148     def __getattr__(self, item: str) -> int:
149         """
150         Accepted items:
151             n
152             e
153             d
154             p
155             q
156             exp1
157             exp2
158             crt_coef
159         """
160         if self._exists(item):
161             return self.lookup[item]
162         else:
163             raise RSAKeyError
164 
165 
166 class TestPubKey(RSAPublicKey):
167     def __init__(self, n, e):
168         self.n = n
169         self.e = e
170 
```

File - /home/tcassar/projects/settle/src/crypto/hashes.py

```python
# coding=utf-8

"""
Simple RSA implementation for signing / verifying digital signatures.
Will have side channel vulnerabilities due to the way that Python stores numbers
Hashing is done through hashlib library
"""

import hashlib
import sys
from dataclasses import dataclass


class HashError(Exception):
    """Hash provided is not in valid format"""


class HasherError(Exception):
    ...


@dataclass
class Hash:
    def __init__(self, h: bytes):
        self.validate(h)
        self.h: bytes = h

    def int_digest(self) -> int:
        # int representation of bytes, not actual value
        return int.from_bytes(self.h, byteorder=sys.byteorder)

    @staticmethod
    def validate(passed_hash):
        try:
            assert len(passed_hash) == 32  # 256 / 8
            assert type(passed_hash) == bytes
        except AssertionError:
```

```
38              raise HashError("Hash was not in the
   form of a hash")
39
40
41  class Hasher:
42      """Interface to hashing; for consistency always
   use SHA256,"""
43
44      @staticmethod
45      def _validate(inp):
46          """Raises HasherError if input isn't bytes
   """
47          if type(inp) is not bytes:
48              raise HasherError(
49                  f"Type passed into _hasher was not
   bytes; received {type(inp)}"
50              )
51
52      def __init__(self, initial: bytes = b"") -> None
   :
53          self._validate(initial)
54          self._hasher = hashlib.sha3_256(initial)
55
56      def update(self, msg: bytes) -> None:
57          """Updates _hasher with given message after
   checking that message was in fact bytes"""
58          self._validate(msg)
59          self._hasher.update(msg)
60
61      def digest(self) -> Hash:
62          """Digests current hash; digests always
   returns a Hash object"""
63          return Hash(self._hasher.digest())
64
65      def int_digest(self) -> int:
66          return Hash(self._hasher.digest()).
   int_digest()
67
```

```
1 # coding=utf-8
2
```

```
 1 -----BEGIN PUBLIC KEY-----
 2 MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqzzsGK6N
   9oxQLtcc4zds
 3 ygDF6Cruv5qlBHlO0rSzQAk+OBy+
   LchMJ02mQKLSw3moxHhogTlJpKecTusBS/PD
 4 /VNYz2gjBLe4Dba4jPyT59Ah40bAHI1z5C/PqH82b51TXO+
   4RqC7MprCdSeeNXQc
 5 L5V+1Ob30GIoyKkw4lCVRrnCclWF5uOE1YnDFjCsdKTW+
   LcwYOQuR7vq2PHpbeWO
 6 m8dJz1Y2zuXTxc27ztqGPcriFrtm7DkV/3Li9g6WsT1AQO0OOF+
   SaiW866+u2zNc
 7 9jST2gKzKSYMktcYPHnusblnRzlCpLpB8hPHEkc2ApdZRvoSc+Mq
   +9kwjUMyXEy/
 8 qwIDAQAB
 9 -----END PUBLIC KEY-----
10
```

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAssGOMnKA
/0q+PtM3sCJR
hWOjNocYURePnuE82Ahch1wIAYTYNsoKPy0ft3HJiTHNviWC/
G6iIJBNTvcznuZz
0UwtkVe8Pq/iZuK976s/Yg+
1SIrTqETTUbwNhmWEL4kjCzI4VEmrjgdnKnZKHDxo
cSnwZUXDv55HNTpAi5CGNwFVQnaKxORDp3nhMY5vnSb0s2vpAEXt
KjKeJydhWLAh
fJP5qEDxD4qOaJQJyZbXsvT+tmm4QnuF+
ptGwDmG3xAZA9Lbtk6xeOplUMzQOKnN
nm9CUBEGoOYAVyGWv9yml/
ucPIaCDdjWOkR0RuMqgqLeiROGjs3BAomkk548Wzdu
FwIDAQAB
-----END PUBLIC KEY-----

```

```
 1 -----BEGIN PUBLIC KEY-----
 2 MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwmwTyC9d
   846+33clYUSk
 3 cd+x9i/3j292+vO0wUp1WWA/
   ceJvVbtkyieVAPRmW9o8ow12e67blp0e5TKtksjR
 4 OI7AnVVT2zJgV4Xbfj/Jqu6x5CNdjVA4vwRnYVp+
   hEQv907A2VJZgXTfras0kI6Z
 5 sryHRpGHUs+gjddWfAap/f9dbtSeDi7dPSW+
   Nr6vzwd53N5VadqKd2N2x2CMEUAP
 6 cwYwOn6dGCyojN4E5Mo1/rJ1QEn6y9Hvuqb8OwpudPxw/phKhax+
   VIyDPaH6QMxR
 7 LnZvpepc4HnQrzXmie99Bhb/
   JfAQv34hoNgJ5HnoUzO2n0xTCxelBG7rIWUs/VXG
 8 BQIDAQAB
 9 -----END PUBLIC KEY-----
10
```

```
 1 -----BEGIN RSA PRIVATE KEY-----
 2 MIIEowIBAAKCAQEAqzzsGK6N9oxQLtcc4zdsygDF6Cruv5qlBHlO
   0rSzQAk+OBy+
 3 LchMJ02mQKLSw3moxHhogTlJpKecTusBS/PD/
   VNYz2gjBLe4Dba4jPyT59Ah40bA
 4 HI1z5C/PqH82b51TXO+4RqC7MprCdSeeNXQcL5V+
   1Ob30GIoyKkw4lCVRrnCclWF
 5 5uOE1YnDFjCsdKTW+
   LcwYOQuR7vq2PHpbeWOm8dJz1Y2zuXTxc27ztqGPcriFrtm
 6 7DkV/3Li9g6WsT1AQO0OOF+SaiW866+
   u2zNc9jST2gKzKSYMktcYPHnusblnRzlC
 7 pLpB8hPHEkc2ApdZRvoSc+Mq+9kwjUMyXEy/
   qwIDAQABAoIBAANaLvkQucDA4HT6
 8 Sxt7o0qVF0rDRGdF3MMoqM1hMj0nsSsiDOsSh8MhNwb+
   6Qdgo1gtT0ZwW6u4iEvX
 9 N/
   BHtmeIMS3mSQE3o4fJMvW3oCGipncvlGU6s7Ec6oDO9L7copwoKU
   Bgtyl3dCUd
10 AHDldAPOdmwulV1j/o5nGYjksPdwdVB5w/
   xnt79FfcAqiYQkShLPUSB9vNGvYc3a
11 xLPupa28dW06ae22/XMTvIdMD73AAMfwTXPwIEkYiQ8L/
   rALGgKxiXDvEHoGNUjj
12 8oRJJOI6fflqzWDunf1crs/exPI+wyxCQBsnp9W8R/
   iJjfzgGMgarW7ifz2VeAjl
13 DUhGPlECgYEA4bwY49tNu9D8/wsZHHqEVuMJZWkyQszW/
   9VF2jns8O7feaYkg++H
14 Tax1zIrLjZeyXheEd0ePUCXwp+A+
   puJfMQDUygyd39GJq5JKOQtjLGNfVslcIbws
15 7cveAAKHPgtZ/315DceZp0T/
   etmSIPKfyKfRwXkdT9gc48BJVJCyYbkCgYEAwjJU
16 fBmcife7c7M9iXTgznsptw14l5eWNGUhEtq12ah9B+
   SWO1O2fBvmert17B2qqrfD
17 7WZ+
   Ou5LukVKTZSlYbK3d1tYECi8IX3dTEM16tzI07rldiSLMYmVGB2y
   rUFEla0i
18 GqToI3os5hjgeb2LFnfCDG2ElXyV3cnEKCfhroMCgYEAjlphU+
   gryEKOGYR1LIly
19 xZxdg49oe8nT1/g2Gd2nt3Tch0moA2/
   dYrVcgEYTAdLlUAMCrXn6Pa//aM64k+Nz
20 5mJAzr5QHSJ18DXMCtdMjmSIbiGDsV6K8tc6w8TUZuMfuUf2PCNg
   maSgfGeSiKaY
```

```
21 7yAt3hWzz3NZHKNZWzP42jkCgYA/
   YcntTzamWTLXSnMVQA53lf9BfaYUZCdkJnWq
22 /
   7NGvRVB1DusVWBEFZ8eA70zOWOQoLXT8BXXChuxShg8Rf3Ma1YyI
   KAXdhQhIkFu
23 OKmKZFEvudpWuwqrOmptGpRMs/
   a8m2t8IsKZgbDR0ODDGzggNyoggEY7vBP19XqO
24 WKXyHQKBgEtL9XgELGf+huemb5xcnkhPEPpfhhGLcJdYa0bK/
   zvSvHb5KPW/Nq9O
25 qTTDc1roXW9sXJ+
   Iaz3hUZkCHBLaqFmeqLsXTw9dGIsANjVyxQFiArqLg88H/OP/
26 1zSh3LNMotIryeDXUPhpfaFJurZU3oGC0vqdPBiAFe3KLSsmn3G5
27 -----END RSA PRIVATE KEY-----
28
```

```
 1 -----BEGIN RSA PRIVATE KEY-----
 2 MIIEpAIBAAKCAQEAssGOMnKA/0q+
   PtM3sCJRhWOjNocYURePnuE82Ahch1wIAYTY
 3 NsoKPy0ft3HJiTHNviWC/G6iIJBNTvcznuZz0UwtkVe8Pq/
   iZuK976s/Yg+1SIrT
 4 qETTUbwNhmWEL4kjCzI4VEmrjgdnKnZKHDxocSnwZUXDv55HNTpA
   i5CGNwFVQnaK
 5 xORDp3nhMY5vnSb0s2vpAEXtKjKeJydhWLAhfJP5qEDxD4qOaJQJ
   yZbXsvT+tmm4
 6 QnuF+
   ptGwDmG3xAZA9Lbtk6xeOplUMzQOKnNnm9CUBEGoOYAVyGWv9yml
   /ucPIaC
 7 DdjWOkR0RuMqgqLeiROGjs3BAomkk548WzduFwIDAQABAoIBAAYk
   eIprPNuI6fim
 8 sWcwBrUas2UQR1miAaFoA/s4blWIaA4FTpizYJ7DIi/
   fjRlVyMgb6eBavN95le58
 9 uB1hteEfSQGRjjppONJoQDaYH9lyyLvxaUyC8PACEkToAB7SC446
   74yV63s57eWb
10 ovhNaxlYY5ZXS16uRawikE07biE9r7WVRQIrKesbwMFvDD3T8Vk8
   3Q6m9dAF3uFm
11 En7++v/sEJM1/
   vBFd2QtCGPT9AajqI8XDrvfJVEdIfd8iUP243uwSjgCsGZoJyPq
12 X/beLBK+
   BLtrJB0obw9UUukD23NjVaAd6rg13smasCvd9RN9ak2V1ivRudjM
   kG9S
13 JloBKwECgYEA7JXiTcWXieqvkW9nCdD943Lxj/
   nZSksYj3fpouEEtOgrNNruu6pw
14 igJmNYUA/KUIN1N5pxyOT4x4+
   Wgr8HqcUz1yqQsv9OFtGpVtTWXs8KlBguzFS2/O
15 1zgJayBe/
   7FjT1ZW06wdKw5d8a5ZYaPW0LDsrgDOYlYhTCYht2WQ5ncCgYEAw
   WzP
16 Djnv9JdQdy7Z6DLnOn753Akmjvrf+sKvbsW2HDiqMKZg+
   DTbTlRlsZzCbv6AMe4f
17 Ji+3bA5d+
   Scdaq8BtDP8ZniLXW71KJ8CT0zPoGC02vYuOnnIBFEuprJGH/
   BIKCFZ
18 pYAhowvnN7vDbNrgT+P1T3/pzydb7ZWpWGF4fWECgYB+
   FqO1r0lER7+qUyUnJisp
19 vWjgwtf7sGo2jEIIfMR1Zgcg9E2n1v6DjyPKAki0XYAzfFmppp5x
   5vvScKJ6V95Y
```

```
20 vrcNQNkeOcZcFKdcSKYu3QX9MRM1UTF7onHili4LOA8liW5dAa3J
   9K480B7y2s7y
21 ClkZo3RbFGxKmUf+5jKDoQKBgQCal/
   i1iiD7AKVTXKLVLzoYSrww3S9wP5sctNW6
22 V1NCIxgDYjdGqhhN9q6A0qWkSMz1GzjSMHkNaD47kMO2LMHT4Wju
   DZJ1zuUq3kve
23 Tan0qNZj2zb/ja4LpUb//
   KSHimhhiqY33L0Fr9HRfcSoEBgQQkYikD2zT+Xtchyz
24 gtPmgQKBgQDrs7LmPSk5/
   DvcIrQQTwFR6BXSR7aizJjgbk72taip94KO63Q7RJF3
25 1TWkjDRJdnh5ZPgF+FC5tn23wBK6hj3rF8bNYMJWLAEVoH2/
   Ks0ZSSmQjRPEJIPK
26 GrnOOue1Irs4fl065ptjZtw8HgKitE9sle1vIgwpUKQXyFrfXAit
   kg==
27 -----END RSA PRIVATE KEY-----
28
```

```
 1 -----BEGIN RSA PRIVATE KEY-----
 2 MIIEpQIBAAKCAQEAwmwTyC9d846+33clYUSkcd+x9i/3j292+
   vO0wUp1WWA/ceJv
 3 VbtkyieVAPRmW9o8ow12e67blp0e5TKtksjROI7AnVVT2zJgV4Xb
   fj/Jqu6x5CNd
 4 jVA4vwRnYVp+hEQv907A2VJZgXTfras0kI6ZsryHRpGHUs+
   gjddWfAap/f9dbtSe
 5 Di7dPSW+
   Nr6vzwd53N5VadqKd2N2x2CMEUAPcwYwOn6dGCyojN4E5Mo1/
   rJ1QEn6
 6 y9Hvuqb8OwpudPxw/phKhax+
   VIyDPaH6QMxRLnZvpepc4HnQrzXmie99Bhb/JfAQ
 7 v34hoNgJ5HnoUzO2n0xTCxelBG7rIWUs/
   VXGBQIDAQABAoIBAQCKPl8ykLu1W2LN
 8 cuLZbv2fGvhnnWPiUdfASVnVtyQKES4LxH5173GTbOG+
   dAn0dhF3vzLob0Ukq0Q0
 9 pPxWywQ5uweq1+lwizmGAmA4PRMgv+
   kt1xfGS6yN5ou75aTgV7vjldP6s1uBeaEy
10 0/
   HkeAFB3lwyCXi3oXsAYIXWWsa1G4n2lEj5FlhWLnuJ2KSEA6PejF
   RopB4GzA7m
11 xYyZre4wKN7oTiSke+
   Pm2xYVMzhwxDG0PQdYZQtyle0ZU8n2E8NxAGrh4KkSNVcP
12 9yknoVYz6Lq85lABziMuhmHnnEGs3r2TRkp29aWJ1IQYjX7/
   UFWEBdqrIm9ePkdp
13 vX+
   S6lABAoGBAOOjqqGrAp21sOf2LXh2lsiq5pnGBI2KqLXiO5XfeJM
   HOvN9MWt/
14 tMgDtTeD7gbYC2pXIKpFuyVQjP10Ol/
   R6Hvm3BzaKwEAtXbsfdmzTxqfs7HI8SXR
15 pwKWqr3m6oFOE+
   EuNBhcbfucmhhB3Wg5lseSDHcRxTLIuvjl1bhZzoqBAoGBANqk
16 +32nhDmou11kXMAs7ipv49oc6T6n2cs1Gy/
   Fbqq1lq4TgDDVgoKwznYB1habFy+E
17 e3qCPcu8JVXuOso+
   L0j21yQ1WlC0OlIfObMgqLCpZjzMqa9upDE304ogJqQJDQk2
18 g/
   LG01fQJtafe3QeoHeTZ9wObuk99Qv8guEOtVGFAoGBAMS6pzY8kn
   A1m4N8L2PS
19 KRGXEN+
   IUFr6oD9AGvq1PIDkWMAhS9p5bYUGH1CxBb6Ia6UlUyeUR95BtPc
```

```
19 9wU0a
20 HW8m3sdYjJ27PRhf3YuM+SorJqLY4/8pJsqH51ti+
   vtwvKF4yrDbAHnYpxuboHr4
21 eiRTRc6JXwE6lMuR5ZgClQsBAoGAI8puAJuzYVzljtwm8q5oLjoy
   qjmhVMhVNpZy
22 5NcEzpz7FXPLwDKzMoGOynJygTDSEs01CVDYnMkns3FT3ldflioR
   /bNeHWfjRB4o
23 a9IkywZv3fQCst0Bs6zXy/yHVsLEh4WNA+jYH7/
   LG8bvhoqc6fYPQk56iWPDATtM
24 kVq/A6ECgYEAuXIqXKiiU+xElmI/3F0aMQDk5z+
   Rodrc8hsUM7BehsmrjerHPK1i
25 qzel3EYEzxdt4vLhgHBgpaA1I91duYQcWENDBUZ134wNx7Sf5s4Q
   i/hrlo4obVA6
26 lEwGgQuHE6XQWH10mouAlp4s4snfRreXurfOMYgMEVlGXk0sDAsn
   xm4=
27 -----END RSA PRIVATE KEY-----
28
```

File - /home/tcassar/projects/settle/src/server/setup.py

```
 1 # coding=utf-8
 2 from setuptools import setup, find_packages  # type
   : ignore
 3
 4 setup(
 5     name="settle-server",
 6     version="0.1.0",
 7     py_modules=["endpoint"],
 8     entry_points="""
 9     [console_scripts]
10     settle-server=endpoint:settle_server
11     """,
12 )
13
```

File - /home/tcassar/projects/settle/src/server/models.py

```python
# coding=utf-8
from dataclasses import dataclass

import click


@dataclass
class User:
    name: str
    email: str
    modulus: str
    pub_exp: str
    password: str = "default"
    id: int = 0

    def __str__(self):
        return f"""\nName:\t{self.name}
Email:\t{self.email}
Modulus:\t{self.modulus},
Public Exponent:\t{self.pub_exp}
                """


@dataclass
class Group:
    id: int
    name: str
    password: str

    def __repr__(self):
        return f"Group(name={self.name}, id={self.id})"

    def __str__(self):
        return f"Group:\n\tName: {self.name}, ID = {self.id}"


@dataclass
class GroupLink:
    id: int
```

```
40      group_id: int
41      usr_id: int
42
43
44  @dataclass
45  class GroupList:
46      groups: list[Group]
47
48      def __str__(self):
49          out = ""
50          for group in self.groups:
51              out += f"{str(group)}\n"
52
53          return out
54
55
56  @dataclass
57  class PrettyTransaction:
58      id: int
59      group: int
60      amount: int
61      time: str
62      reference: str
63      other: str
64      verified: bool
65
66      def secho(self):
67          click.secho("\n----")
68          click.secho(f"Transaction ID = {self.id}\n
    Group: {self.group}", bold=True)
69          click.secho(f"{self.other}, £{(int(self.
    amount) / 100):.2f}", fg="blue")
70          click.secho(self.reference)
71          click.secho(f"at {self.time}")
72
73          if self.verified:
74              click.secho("Verified: True\n", fg="
    green")
75
76          else:
77              click.secho("Verified: False\n", fg="red
```

```
 77 ", blink=True, bold=True)
 78
 79
 80 @dataclass
 81 class PrettyList:
 82     src_list: list[PrettyTransaction]
 83     dest_list: list[PrettyTransaction]
 84
 85     def __bool__(self):
 86         return True if (self.src_list or self.
    dest_list) else False
 87
 88     def __repr__(self):
 89         return f"{self.src_list}"
 90
 91     def secho(self):
 92         for trn in self.src_list:
 93             trn.secho()
 94         for trn in self.dest_list:
 95             trn.secho()
 96
 97
 98 @dataclass
 99 class Signature:
100     transaction_id: int
101     signature: str  # store as hex
102     origin: str  # src or dest
103
```

File - /home/tcassar/projects/settle/src/server/schemas.py

```python
# coding=utf-8

from marshmallow import Schema, fields, post_load

import models  # type: ignore
import src.transactions.ledger
import src.transactions.transaction


class UserSchema(Schema):
    id = fields.Int()
    name = fields.Str()
    email = fields.Email()
    modulus = fields.Str()
    pub_exp = fields.Str()
    password = fields.Str()

    @post_load
    def make_user(self, data, **kwargs):
        return models.User(**data)


class GroupSchema(Schema):
    id = fields.Int()
    name = fields.Str()
    password = fields.Str()

    @post_load
    def make_group(self, data, **kwargs):
        return models.Group(**data)


class GroupLinkSchema(Schema):
    id = fields.Int()
    group_id = fields.Int()
    usr_id = fields.Int()

    @post_load
    def make_group_link(self, data, **kwargs):
        return models.GroupLink(**data)

```

```
42
43 class GroupListSchema(Schema):
44     groups = fields.List(fields.Nested(GroupSchema
   ()))
45
46     @post_load
47     def make_group_list(self, data, **kwargs):
48         return models.GroupList(**data)
49
50
51 # Transaction schemas
52
53
54 class TransactionSchema(Schema):
55     src = fields.Int()
56     dest = fields.Int()
57     amount = fields.Int()
58     src_pub = fields.Str()
59     dest_pub = fields.Str()
60     ID = fields.Int()
61     reference = fields.Str()
62     time = fields.Str()
63     signatures = fields.Dict()
64     group = fields.Int()
65
66     @post_load
67     def make_transaction(self, data, **kwargs):
68         return src.transactions.transaction.
   Transaction(**data)
69
70
71 class PrettyTransactionSchema(Schema):
72     id = fields.Int()
73     other = fields.Str()
74     group = fields.Int()
75     amount = fields.Int()
76     time = fields.Str()
77     reference = fields.Str()
78     verified = fields.Bool()
79
80     @post_load
```

```
 81     def make_pretty_transaction(self, data, **
    kwargs):
 82         return models.PrettyTransaction(**data)
 83
 84
 85 class PrettyListSchema(Schema):
 86     src_list = fields.List(fields.Nested(
    PrettyTransactionSchema()))
 87     dest_list = fields.List(fields.Nested(
    PrettyTransactionSchema()))
 88
 89     @post_load
 90     def make_pretty_list(self, data, **kwargs):
 91         return models.PrettyList(**data)
 92
 93
 94 class SignatureSchema(Schema):
 95     transaction_id = fields.Int()
 96     signature = fields.Str()  # store as hex
 97     origin = fields.Str()  # src or dest
 98
 99     @post_load
100     def make_signature(self, data, **kwargs):
101         return models.Signature(**data)
102
```

File - /home/tcassar/projects/settle/src/server/endpoint.py

```python
# coding=utf-8
import os

import click
from flask import Flask, g
from flask_restful import Resource, Api, abort  # type: ignore

from src.server.processes import get_db
from src.server.resources import (
    Group,
    User,
    UserGroupBridge,
    PrettyTransaction,
    TransactionSigVerif,
    Simplifier,
    GroupDebt,
    SignableTransaction,
)

app = Flask(__name__)
api = Api(app)


@app.teardown_appcontext
def close_connection(exception):
    """Closes db if sudden error"""
    db = getattr(g, "_database", None)
    if db is not None:
        db.close()


@click.group()
def settle_server():
    ...


@click.option("-d", "--debug", is_flag=True, default=False)
@click.option("-h", "--host", default="127.0.0.1")
@settle_server.command()
```

```
40 def start(host, debug):
41     os.chdir("/home/tcassar/projects/settle")
42     app.run(debug=debug, host=host)
43     db = get_db()
44
45
46 api.add_resource(Group, "/group/<int:id>", "/group")
47
48 api.add_resource(PrettyTransaction, "/transaction",
   "/transaction/<string:email>")
49
50 api.add_resource(User, "/user/<string:email>", "/
   user")
51
52 api.add_resource(
53     UserGroupBridge, "/group/<int:id>/<string:email
   >", "/group/<string:email>"
54 )
55
56 api.add_resource(
57     TransactionSigVerif, "/transaction/auth/<int:id
   >", "/transaction/auth/"
58 )
59
60 api.add_resource(Simplifier, "/simplify/<int:gid>")
61
62 api.add_resource(GroupDebt, "/group/debt/<int:id>")
63
64 api.add_resource(SignableTransaction, "/transaction/
   settle/<int:t_id>", "/transaction/signable/<int:id>"
   )
65
```

File - /home/tcassar/projects/settle/src/server/processes.py

```python
# coding=utf-8
import sqlite3

from flask import g

import src.server.models as models
import src.transactions.transaction
from src.crypto import keys as keys

DATABASE = "/home/tcassar/projects/settle/settle_db.sqlite"


class ResourceNotFoundError(Exception):
    ...


def get_db():
    """Returns current database connection"""
    db = getattr(g, "_database", None)
    if db is None:
        # connect
        db = g._database = sqlite3.connect(DATABASE)

    return db


def build_args(data_from_cursor: list | tuple) -> list:
    """Given a ROW OF PARAMETERS FROM DB will return list as args"""
    if args := data_from_cursor:
        args = [item for item in args]
        return args
    else:
        raise ResourceNotFoundError(
            "Database error: failed to build schema of object as nothing was retrieved"
        )


```

```
38 def build_pretty_transactions(
39     src_sql: str, dest_sql: str, cursor: sqlite3.
   Cursor, args: list
40 ) -> models.PrettyList:
41     """Returns pretty list of unchecked transactions
   """
42     # TODO: add transaction verification
43 
44     if type(args) is not list:
45         args = list(args)
46 
47     src_transactions: list[models.PrettyTransaction
   ] = []
48     dest_transactions: list[models.PrettyTransaction
   ] = []
49 
50     src_data = cursor.execute(src_sql, args)
51 
52     for row in src_data.fetchall():
53         pretty = models.PrettyTransaction(*
   build_args(row), False)
54         print(f"Checking id {pretty.id}")
55         verify_pretty(pretty, cursor)
56         src_transactions.append(pretty)
57 
58     dest_data = cursor.execute(dest_sql, args)
59 
60     for row in dest_data.fetchall():
61         pretty = models.PrettyTransaction(*
   build_args(row), False)
62         print(f"Checking id {pretty.id}")
63         verify_pretty(pretty, cursor)
64         dest_transactions.append(pretty)
65 
66     return models.PrettyList(src_transactions,
   dest_transactions)
67 
68 
69 def get_verified_transaction_by_id(
70     id: int, cursor: sqlite3.Cursor
71 ) -> src.transactions.transaction.Transaction:
```

```
 72     """Gets transaction object from id
 73     raises ResourceNotFoundError if nothing is
    returned"""
 74
 75     sql = """SELECT src_id, dest_id, k.n, k.e, k2.n
    , k2.e, amount, transactions.id, reference,
    time_of_creation, src_sig, dest_sig, g.id FROM
    transactions
 76 JOIN keys k on transactions.src_key = k.id
 77 JOIN keys k2 on transactions.dest_key = k2.id
 78 JOIN pairs p on transactions.pair_id = p.id
 79 JOIN groups g on transactions.group_id = g.id
 80 WHERE transactions.id = ?;
 81     """
 82
 83     raw_transaction_data = cursor.execute(sql, [id
    ]).fetchone()
 84     if not raw_transaction_data:
 85         raise ResourceNotFoundError(f"No
    transaction with ID={id}")
 86
 87     else:
 88         src_key_data = raw_transaction_data[2:4]
 89         dest_key_data = raw_transaction_data[4:6]
 90         transaction_data = list(
    raw_transaction_data[:2]) + list(
 91             raw_transaction_data[6:]
 92         )
 93
 94         (
 95             src_id,
 96             dest_id,
 97             amount,
 98             t_id,
 99             ref,
100             time,
101             src_sig,
102             dest_sig,
103             group,
104         ) = transaction_data
105
```

```
106         print(
107             f"amount: {amount}\n"
108             f"tran id: {t_id}\n"
109             f"ref: {ref}\n"
110             f"time: {time}\n"
111             f"src_sig: {src_sig}\n"
112             f"dest_sig: {dest_sig}\n"
113             f"group: {group}\n"
114         )
115 
116         # build keys
117         # convert hex -> ints
118         src_key_data = map(lambda x: int(x, 16), src_key_data)
119         dest_key_data = map(lambda x: int(x, 16), dest_key_data)
120 
121         # load to key loaders
122         src_ldr = keys.RSAKeyLoaderFromNumbers()
123         dest_ldr = keys.RSAKeyLoaderFromNumbers()
124 
125         src_ldr.load(*src_key_data)  # type: ignore
126         dest_ldr.load(*dest_key_data)
127 
128         # add keys to transaction data in the right place so that they can be unpacked as positional arguments
129         transaction_data.insert(3, dest_ldr.pub_key())
130         transaction_data.insert(3, src_ldr.pub_key())
131 
132         # build signatures dict for transaction
133 
134         signatures = {}
135 
136         for sig, notary in zip([src_sig, dest_sig], [src_id, dest_id]):
137             if sig:
138                 signatures[notary] = int(sig, 16)
139 
```

```
140         # remove signatures from transaction data,
    replace w dict
141         transaction_data.remove(src_sig)
142         transaction_data.remove(dest_sig)
143 
144         transaction_data.insert(-1, signatures)
145 
146         # return transaction
147         return src.transactions.transaction.
    Transaction(*transaction_data)
148 
149 
150 def user_exists(email: str, cursor: sqlite3.Cursor
    ) -> bool:
151     return not not cursor.execute(
152         """SELECT COUNT(*) FROM users WHERE email
     = ?""", [email]
153     ).fetchone()[0]
154 
155 
156 def group_exists(id: int, cursor: sqlite3.Cursor
    ) -> bool:
157     return not not cursor.execute(
158         """SELECT COUNT(*) FROM groups WHERE id
     = ?""", [id]
159     ).fetchone()[0]
160 
161 
162 def transaction_to_pretty(emails, transaction,
    verified):
163     pretty = models.PrettyTransaction(
164         transaction.ID,
165         transaction.group,
166         transaction.amount,
167         transaction.time,
168         transaction.reference,
169         f"{emails[0]} -> {emails[1]}",
170         verified,
171     )
172     return pretty
173 
```

```
174
175 def verify_pretty(
176     pretty: models.PrettyTransaction, cursor:
    sqlite3.Cursor
177 ) -> models.PrettyTransaction:
178     """Update the verification status of pretty
    depending on signatures"""
179     try:
180         t = get_verified_transaction_by_id(pretty.
    id, cursor)
181         t.verify()
182         pretty.verified = True
183     except src.transactions.transaction.
    VerificationError:
184         pretty.verified = False
185         print(f"Signature of {pretty.id} invalid")
186
187     return pretty
188
189
190 def push_transaction(
191     transaction: src.transactions.transaction.
    Transaction, cursor: sqlite3.Cursor
192 ):
193
194     # check pair exists; append if not
195     pair_id: int = cursor.execute(
196         """SELECT pairs.id FROM pairs WHERE src_id
     = ? and dest_id = ?""",
197         [
198             transaction.src,
199             transaction.dest,
200         ],
201     ).fetchone()[0]
202
203     if not pair_id:
204         cursor.execute(
205             """INSERT INTO pairs (src_id, dest_id)
    VALUES (?, ?)""",
206             [
207                 transaction.src,
```

```
208                 transaction.dest,
209             ],
210         )
211         get_db().commit()
212 
213     # get key ids of users
214     key_ids = cursor.execute(
215         """SELECT keys.id FROM keys
216                         JOIN users u on keys.id = u
    .key_id
217                         JOIN users u2 on keys.id =
    u2.key_id
218                         WHERE u.id = ? OR u2.id
     = ?""",
219         [transaction.src, transaction.dest],
220     ).fetchall()
221 
222     sql = """INSERT INTO transactions
223     (pair_id, group_id, amount, src_key, dest_key,
    reference, time_of_creation)
224             VALUES (?, ?, ?, ?, ?, ?, ?)"""
225 
226     print(key_ids[0][0], key_ids[1][0])
227 
228     # append unsigned transactions
229     cursor.execute(
230         sql,
231         [
232             pair_id,
233             transaction.group,
234             transaction.amount,
235             key_ids[0][0],
236             key_ids[1][0],
237             transaction.reference,
238             transaction.time
239         ],
240     )
241 
242     get_db().commit()
243 
```

File - /home/tcassar/projects/settle/src/server/resources.py

```python
# coding=utf-8
import json

from flask import request
from flask_restful import Resource, abort  # type: ignore

import src.transactions.transaction as transactions
import src.transactions.ledger as ledgers
from src.server import models as models, schemas as schemas, processes as processes


# Resources


class Group(Resource):
    def get(self, id: int):
        cursor = processes.get_db().cursor()
        # check group exists and
        get_group = """SELECT id, name, password FROM groups
                    WHERE groups.id = ?"""
        try:
            group_data = cursor.execute(get_group, [id])
            group_data = group_data.fetchall()
            # create group object
            group = models.Group(*processes.build_args(*group_data))
        except IndexError:
            abort(404, message="Group ID does not exist")
            group = ""  # type: ignore
        except TypeError as te:
            abort(404, message=f"Group data invalid ; {te}")
            group = ""  # type: ignore
        # create group schema
        schema = schemas.GroupSchema()
        return schema.dump(group), 200
```

```
35
36      def post(self):
37
38          # print(request.json)
39          schema = schemas.GroupSchema()
40          group = schema.load(request.json)
41
42          cursor = processes.get_db().cursor()
43
44          cursor.execute(
45              """INSERT INTO groups (name, password)
    VALUES (?, ?)""",
46              [group.name, group.password],
47          )
48
49          processes.get_db().commit()
50
51          return f"Created group ID={cursor.lastrowid}
     named {group.name}", 201
52
53
54  class User(Resource):
55      def get(self, email: str):
56          # query db for all users
57          cursor = processes.get_db().cursor()
58
59          query = """
60                    SELECT users.name, users.email,
    keys.n, keys.e, users.password, users.id
61                    FROM users, keys
62                    WHERE email = ? AND keys.id = users
    .key_id;
63                  """
64
65          # only return one usr, so unpack first into
    usr_data
66          try:
67              usr_data: list = cursor.execute(query, [
    email]).fetchone()
68          except IndexError:
69              # returned blank info
```

```
70              return "User data not found", 404
71
72          # use data to build user class
73          # use schema to convert to json
74
75          try:
76              # make sure the requested user exists
77              usr = models.User(*[item for item in
    usr_data])
78              schema = schemas.UserSchema()
79          except TypeError:
80              # didn't have required arguments to
    build usr
81              return "User data invalid", 404
82
83          return schema.dump(usr), 200
84
85      def post(self):
86
87          cursor = processes.get_db().cursor()
88
89          # now is a user object
90          schema = schemas.UserSchema()
91          usr = schema.load(request.json)
92
93          query = """SELECT users.id FROM users WHERE
     email = ?
94          """
95
96          exists = cursor.execute(query, [usr.email])
97          if exists.fetchall():
98              abort(409, message="User already exists
    ")
99
100         # add user to db
101
102         keys_query = """INSERT INTO keys (n, e)
103                         VALUES (?, ?)"""
104
105         users_query = """INSERT INTO users (NAME,
    EMAIL, PASSWORD, KEY_ID)
```

```
106                             VALUES (?, ?, ?, ?)"""
107
108         cursor.execute(keys_query, [usr.modulus,
    usr.pub_exp])
109
110         key_id = cursor.lastrowid
111         cursor.execute(users_query, [usr.name, usr.
    email, usr.password, key_id])
112         processes.get_db().commit()
113
114         return (schema.dump(usr),)
115
116
117 class UserGroupBridge(Resource):
118     """For handling users connections to groups
119     POST will add user to group
120     GET will get all groups associated with user"""
121
122     def post(self, id: int, email: str):
123         # assumes these things already exist as
    they have been validated by client already
124         uid_sql = """SELECT id FROM users WHERE
    email = ?"""
125
126         glink_sql = """INSERT INTO group_link (
    group_id, usr_id)
127                             VALUES (?, ?)"""  # group
    id then user id
128
129         cursor = processes.get_db().cursor()
130         uid = cursor.execute(uid_sql, [email]).
    fetchone()[0]
131
132         cursor.execute(glink_sql, [id, uid])
133
134         processes.get_db().commit()
135
136         glink_data = cursor.execute(
137             """SELECT * from group_link WHERE id
     = ?""", [cursor.lastrowid]
138         )
```

```
139
140         try:
141             print("making group")
142             glink = models.GroupLink(*processes.
    build_args(glink_data.fetchone()))
143         except processes.ResourceNotFoundError as
    re:
144             return 404, f"{re}, failed"
145
146         schema = schemas.GroupLinkSchema()
147
148         return schema.dump(glink), 201
149
150     def get(self, email: str):
151         """Return all groups that a user is part of
    """
152
153         sql = """SELECT g.id, g.name, g.password
    FROM users
154         JOIN group_link gl on users.id = gl.usr_id
155         JOIN groups g on g.id = gl.group_id
156         WHERE users.id
157                   = (
158             SELECT users.id FROM users
159             WHERE email = ?
160             );
161         """
162
163         cursor = processes.get_db().cursor()
164         group_data = cursor.execute(sql, [email])
165         groups: list[models.Group] = []
166
167         for row in group_data.fetchall():
168             groups.append(models.Group(*processes.
    build_args(row)))
169
170         groups_obj = models.GroupList(groups)
171         groups_schema = schemas.GroupListSchema()
172
173         return groups_schema.dump(groups_obj), 200
174
```

```
175
176 class PrettyTransaction(Resource):
177     def get(self, email: str):
178         """Gets a user's unsigned transactions"""
179
180         cursor = processes.get_db().cursor()
181
182         if not processes.user_exists(email, cursor
    ):
183             print("not found")
184             return f"User by email {email} not
    found", 404
185
186         src_sql = """SELECT transactions.id,
    group_id, amount, reference, time_of_creation, u2.
    email FROM transactions
187                 INNER JOIN pairs p on p.id =
    transactions.pair_id
188                 INNER JOIN users u on u.id = p.
    src_id
189                 INNER JOIN users u2 on u2.id = p.
    dest_id
190                 WHERE transactions.src_settled = 0
    AND u.email = ?
191         """
192
193         dest_sql = """
194         SELECT transactions.id, group_id, amount,
    reference, time_of_creation, u2.email FROM
    transactions
195                 INNER JOIN pairs p on p.id =
    transactions.pair_id
196                 INNER JOIN users u on u.id = p.
    dest_id
197                 INNER JOIN users u2 on u2.id = p.
    src_id
198                 WHERE transactions.src_settled = 0
    AND u.email = ?"""
199
200         pretty_list = processes.
    build_pretty_transactions(
```

```
201             src_sql, dest_sql, cursor, [email]
202         )
203 
204         pretty_list_schema = schemas.
    PrettyListSchema()
205         if not pretty_list:
206             return "No open transactions", 200
207         else:
208             return pretty_list_schema.dump(
    pretty_list), 200
209 
210     def post(self):
211         cursor = processes.get_db().cursor()
212 
213         # note: IDs are ints
214 
215         # load transaction object into schema from
    request
216         trn_json = request.json
217         trn_schema = schemas.TransactionSchema()
218         transaction = trn_schema.make_transaction(
    trn_json)
219 
220         # check that both people involvled in
    transaction are members of transaction group
221         sql = """SELECT count(*) FROM groups
222                  INNER JOIN group_link gl on groups
    .id = gl.group_id
223                  INNER JOIN users u
224                  INNER JOIN users u2
225                  WHERE group_id = ? and u.id = ?
    and u2.id = ?
226           """
227 
228         users_in_group = cursor.execute(
229             sql, [transaction.group, transaction.
    src, transaction.dest]
230         )
231         if users_in_group.fetchone()[0] == 0:
232             return f"Users are not both members of
    group {transaction.group}", 403
```

```
233
234         insert_to_pairs = """INSERT INTO pairs (
    src_id, dest_id)
235                              VALUES (?, ?) ON
    CONFLICT DO NOTHING """
236
237         cursor.execute(insert_to_pairs, [
    transaction.src, transaction.dest])
238         processes.get_db().commit()
239
240         # get relevant info
241         pair_id = cursor.execute(
242             """SELECT pairs.id FROM pairs
243                       WHERE src_id = ? AND dest_id
     = ?""",
244             [transaction.src, transaction.dest],
245         ).fetchone()[0]
246
247         key_id_query = """SELECT keys.id FROM keys
248                           JOIN users u on keys.id
     = u.key_id
249                           WHERE u.id = ?"""
250
251         src_key_id = cursor.execute(key_id_query, [
    transaction.src]).fetchone()[0]
252         dest_key_id = cursor.execute(key_id_query
    , [transaction.dest]).fetchone()[0]
253
254         cursor.execute(
255             """INSERT INTO transactions
256         (pair_id, group_id, amount, src_key,
    dest_key, reference, time_of_creation)
257         VALUES (?, ?, ?, ?, ?, ?, ?)""",
258             [
259                 pair_id,
260                 transaction.group,
261                 transaction.amount,
262                 src_key_id,
263                 dest_key_id,
264                 transaction.reference,
265                 transaction.time,
```

```
266                 ],
267             )
268 
269             processes.get_db().commit()
270 
271             return cursor.lastrowid, 201
272 
273 
274 class TransactionSigVerif(Resource):
275     def get(self, id):
276         """Verify a transaction, returning pretty
    copy of verified transaction"""
277 
278         try:
279             transaction = processes.
    get_verified_transaction_by_id(
280                 id, processes.get_db().cursor()
281             )
282         except processes.ResourceNotFoundError as
    rnfe:
283             return str(rnfe), 404
284 
285         try:
286             transaction.verify()
287             verified = True
288         except transactions.VerificationError:
289             verified = False
290 
291         # get emails involved in transaction, check
     if user is src or dest
292         emails = (
293             processes.get_db()
294             .cursor()
295             .execute(
296                 """
297                     SELECT u.email, u2.email FROM
    transactions
298                     JOIN pairs p on transactions.
    pair_id = p.id
299                     JOIN users u on p.src_id = u.id
300                     JOIN users u2 on dest_id = u2.
```

```
300 id
301                         WHERE transactions.id = ?
302                     """,
303                     [id],
304                 )
305                 .fetchone()
306             )
307 
308             # build pretty transaction to send back to
    the user
309 
310             pretty = processes.transaction_to_pretty(
    emails, transaction, verified)
311 
312             schema = schemas.PrettyTransactionSchema()
313 
314             return schema.dump(pretty), 200
315 
316         def patch(self):
317             """Append a signature to a transaction"""
318 
319             schema = schemas.SignatureSchema()
320             sig = schema.make_signature(request.json)
321             if request.json is None:
322                 return "Invalid Request", 404
323 
324             if sig.origin == "dest":
325                 sql = """ UPDATE transactions
326                           SET dest_sig = ?
327                           WHERE id = ?"""
328             elif sig.origin == "src":
329                 sql = """UPDATE transactions
330                 SET src_sig = ?
331                 WHERE id = ?"""
332             else:
333                 return (
334                     "Signature does not originate from
    one of the parties in this transaction",
335                     403,
336                 )
337 
```

```
338         cursor = processes.get_db()
339         cursor.execute(sql, [sig.signature, sig.
    transaction_id])
340 
341         processes.get_db().commit()
342 
343         return "Successfully added signature to
    transaction", 201
344 
345 
346 class Simplifier(Resource):
347     def post(self, gid: int):
348         """Actually settle the group, return ledger
     schema, 201 if succeeded"""
349 
350         cursor = processes.get_db()
351 
352         # build full transactions of group
353         # get list of IDs required
354 
355         print(gid)
356 
357         ids = cursor.execute(
358             """SELECT transactions.id from
    transactions
359                                     WHERE group_id = ?
    AND src_settled = 0 AND dest_settled = 0""",
360             [gid],
361         ).fetchall()
362 
363         unfiltered_transactions: list[transactions.
    Transaction] = []
364         for id in ids:
365             unfiltered_transactions.append(
    processes.get_verified_transaction_by_id(*id,
    cursor))  # type: ignore
366 
367         # build ledger
368         ledger = ledgers.Ledger()
369         for transaction in unfiltered_transactions:
370             ledger.append(transaction)
```

```
371
372         # simplify debt system
373         try:
374             ledger.simplify_ledger()
375         except ledgers.NoFutherSimplifications:
376             return (
377                 "No changes made to debt structure
     - heuristic did not find anywhere to simplify",
378                 202,
379             )
380         except ledgers.VerificationError:
381             return "Couldn't simplify group -
    unverified transactions in group", 403
382
383         # mark old transactions as settled
384
385         # otherwise, group debt is now simplified
386         # thus, mark off all old transactions and
    push in new ones
387         cursor.execute(
388             """UPDATE transactions
389         SET src_settled = 1, dest_settled = 1
390         WHERE group_id = ? """,
391             [gid],
392         )
393
394         # push transactions to db
395         for transaction in ledger.ledger:
396             transaction.group = gid
397             processes.push_transaction(transaction
    , cursor)
398
399         return 'success', 201
400
401
402 class GroupDebt(Resource):
403     def get(self, id):
404         """Return open transactions in a group"""
405
406         cursor = processes.get_db()
407
```

```
408         sql = """SELECT transactions.id, group_id,
    amount, time_of_creation, reference, u.email, u2.
    email
409                     FROM transactions
410                     INNER JOIN pairs p on p.id =
    transactions.pair_id
411                     INNER JOIN users u on u.id = p.
    src_id
412                     INNER JOIN users u2 on u2.id =
    p.dest_id
413                     WHERE group_id = ? AND (
    transactions.src_settled = 0 OR transactions.
    dest_settled = 0);
414             """
415
416         trns: list[models.PrettyTransaction] = []
417         for row in cursor.execute(sql, [id]).
    fetchall():
418             row = list(row)
419             dest = row.pop()
420             src = row.pop()
421             row.append(f"{src} -> {dest}")
422
423             pretty = models.PrettyTransaction(*
    processes.build_args(row), False)
424             pretty = processes.verify_pretty(pretty
    , cursor)
425
426             trns.append(pretty)
427
428         schema = schemas.PrettyListSchema()
429         return schema.dump(models.PrettyList(trns
    , [])), 200
430
431
432 class SignableTransaction(Resource):
433     def get(self, id):
434         """Returns a fully signable transaction
    object"""
435
436         try:
```

```
437             transaction = processes.
    get_verified_transaction_by_id(
438                 id, processes.get_db().cursor()
439             )
440         except processes.ResourceNotFoundError as
    rnfe:
441             return str(rnfe), 404
442 
443         schema = schemas.TransactionSchema()
444         return schema.dump(transaction), 200
445 
446     def patch(self, t_id):
447         cursor = processes.get_db()
448         email: dict = json.loads(request.json)['
    email']
449 
450         # determine if user is src or dest
451         emails = cursor.execute("""SELECT u.email,
    u2.email FROM transactions
452         JOIN pairs p on transactions.pair_id = p.id
453         JOIN users u on p.src_id = u.id
454         JOIN users u2 on p.dest_id = u2.id
455         WHERE transactions.id = ? """, [t_id]).
    fetchone()
456 
457         if email == emails[0]:
458             # usr is src thus append src_settled
459             sql = """UPDATE transactions SET
    src_settled = 1 WHERE id = ?"""
460         elif email == emails[1]:
461             sql = """UPDATE transactions SET
    dest_settled = 1 WHERE id = ?"""
462         else:
463             return f'Email provided is not involved
     in transaction {t_id}', 403
464 
465         cursor.execute(sql, [t_id])
466         cursor.commit()
467 
```

```python
# coding=utf-8
import logging
import sys
from dataclasses import dataclass
from typing import Callable

from ordered_set import OrderedSet

# initialise logger
import src.simplify.graph_objects
from src.simplify import base_graph as graphs

logging.basicConfig(stream=sys.stdout, encoding="utf-8", level=logging.DEBUG)


# typing
disc_map = dict[
    src.simplify.graph_objects.Vertex, src.simplify.graph_objects.Vertex | bool
]
"""disc_map used to track which nodes have been discovered; {node: discovered?}"""


prev_map = dict[
    src.simplify.graph_objects.Vertex, src.simplify.graph_objects.Vertex | None
]
"""prev_map used to track a path through a graph; stored as {node: comes_from_node}.
If node has no links (i.e. is start node), value is None"""


class SearchError(Exception):
    def __init__(self, text, node: src.simplify.graph_objects.Vertex):
        super(SearchError, self).__init__(text, node)
        self.node = node
```

```
34
35
36 def void(
37     current: src.simplify.graph_objects.Vertex,
38     neighbour: src.simplify.graph_objects.Vertex,
39 ) -> None:
40     """Placeholder for a plain bfs; allows adding
   functionality such as a maxflow along each edge
   during a BFS"""
41     pass
42
43
44 def str_map(generic_map: disc_map | prev_map) -> str
   :
45     """""""
46     str_rep = ""
47     for node, prev in generic_map.items():
48         str_rep += f"{node}: {prev},\n".upper()
49     return str_rep
50
51
52 @dataclass(init=False)
53 class BFSQueue:
54     def __init__(self, *args: src.simplify.
   graph_objects.Vertex):
55         self.data: OrderedSet[src.simplify.
   graph_objects.Vertex] = OrderedSet(args)
56
57     def __str__(self):
58         str_ = ""
59         for datum in self.data:
60             str_ += str(datum).upper()
61         return str_
62
63     def enqueue(self, *args: src.simplify.
   graph_objects.Vertex):
64         for v in args:
65             graphs.GenericDigraph.sanitize(v)
66             self.data.append(v)
67
68     def dequeue(self):
```

```
 69         return self.data.pop(0)
 70
 71     def is_empty(self) -> bool:
 72         return not len(self.data)
 73
 74
 75 class Path:
 76     """Namespace for graph operations to do with walking through graph"""
 77
 78     """Shortest path code: returns list[Vertex], hops along a path"""
 79
 80     @staticmethod
 81     def build_bfs_structs(
 82         graph: graphs.GenericDigraph,
 83         src: None | src.simplify.graph_objects.Vertex = None,
 84     ) -> tuple[BFSQueue, disc_map, prev_map]:
 85         """Helper function to initialise prev_map, disc_map and bfs queue;
 86         if source is passed then queue initialised with src"""
 87         queue = BFSQueue()
 88         disc: disc_map = {node: False for node in graph.nodes()}
 89         prev: prev_map = {node: None for node in graph.nodes()}
 90
 91         if src:
 92             queue.enqueue(src)
 93         else:
 94             # get 'first' item from graph
 95             # n = len(graph.graph)
 96             # start = list(graph.graph.keys())[random.randint(0, n - 1)]
 97             # queue.enqueue(start)
 98             start = next(iter(graph.graph))
 99             logging.debug(f"starting from {start}")
100
101             # queue.enqueue(next(iter(graph.graph
```

```
101 )))
102 
103         return queue, disc, prev
104 
105     @staticmethod
106     def shortest_path(
107         graph: graphs.GenericDigraph,
108         source: src.simplify.graph_objects.Vertex,
109         sink: src.simplify.graph_objects.Vertex,
110         neighbours: Callable,
111     ) -> list[src.simplify.graph_objects.Vertex]:
112         """
113         Uses a recursive implementation of BFS to
    find path between nodes
114         Accepts graph, source node, sink node,
    returns list of nodes, which is the path from src
    to sink
115         """
116 
117         # create queue, discovered list, previous
    list
118         queue, discovered, prev = Path.
    build_bfs_structs(graph, source)
119 
120         # recursive call
121         previous = Path.BFS(
122             graph=graph,
123             queue=queue,
124             discovered=discovered,
125             target=sink,
126             previous=prev,
127             neighbours=neighbours,
128         )
129 
130         return Path._build_path(previous, sink)
131 
132     @staticmethod
133     def _build_path(
134         previous: prev_map, sink: src.simplify.
    graph_objects.Vertex
135     ) -> list[src.simplify.graph_objects.Vertex]:
```

```
136         """Given a mapping of previous nodes,
    reconstructs a path to sink"""
137         path: list[src.simplify.graph_objects.
    Vertex] = [sink]
138 
139         while current := previous[path[0]]:
140             path.insert(0, current)
141 
142         if path[0] == sink:
143             path = []
144 
145         return path
146 
147     @staticmethod
148     def BFS(
149         *,
150         graph: graphs.GenericDigraph,
151         queue: BFSQueue,
152         discovered: disc_map,
153         target: src.simplify.graph_objects.Vertex
     | None,
154         previous: prev_map,
155         neighbours: Callable,
156         do_to_neighbour: Callable = void,
157     ) -> prev_map:
158         """BFS Search as part of finding the
    shortest path through unweighted graph from src ->
    target.
159         Target = None => walk through entire graph
    , terminate at empty queue
160         Returns 'previous' list so that path can be
     rebuilt.
161         Can pass in a function `do_to_neighbour` to
     do to all nodes during the BFS
162 
163         neighbour function need to return edge
164         """
165         # breakpoint: f"q: {queue}\n discovered: {
    str_map(discovered)}\nprev: {str_map(prev)}"
166 
167         # will only happen if no path to node
```

```
168         if queue.is_empty():
169             return previous
170 
171         else:
172             # discover next node in queue
173             current = queue.dequeue()
174             discovered[current] = True
175 
176             # check we haven't been fed a
    standalone node (i.e. no forward or backwards links
    )
177             if not graph.connected(current):
178                 if not queue:
179                     raise SearchError("Cannot
    traverse a non connected node", current)
180 
181             # if discovered target node return prev
182             if current == target:
183                 return previous
184 
185             else:
186                 # otherwise, continue on
187                 # enqueue neighbours, keep track of
     whose neighbours they are given not already
    discovered
188                 # do passed in function to
    neighbouring nodes
189                 for neighbour in neighbours(current
    ):
190                     if not discovered[neighbour.
    node]:
191                         previous[neighbour.node] =
    current
192                         queue.enqueue(neighbour.
    node)
193 
194                     do_to_neighbour(current,
    neighbour.node)
195 
196                 # recursive call on new state
197                 return Path.BFS(
```

```
198                         graph=graph,
199                         queue=queue,
200                         discovered=discovered,
201                         target=target,
202                         previous=previous,
203                         neighbours=neighbours,
204                         do_to_neighbour=do_to_neighbour
    ,
205                     )
206
```

```python
# coding=utf-8

"""
Set up graph object to be used in condensing debt settling
"""

import copy

import graphviz

from src.simplify.graph_objects import Vertex, Edge


class GraphError(Exception):
    ...


class Default:
    def __gt__(self, other):
        return True


class GenericDigraph:
    def __init__(self, vertices: list[Vertex]) -> None:
        """
        Sets up a graph given a list of vertices
        """
        # initialise with values being empty
        # build dict checking each type as we go

        self.graph: dict[Vertex, list[Edge]] = {
            vertex: [] if self.sanitize(vertex) else None  # type: ignore
            for vertex in vertices
        }

        self._backwards_graph: dict[Vertex, list[Edge]] = copy.deepcopy(self.graph)

```

```
38      def __str__(self):
39          """Pretty print graph"""
40          out = ""
41          for node, adj_list in self.graph.items():
42              pretty_nodes = ""
43              for edge in adj_list:
44                  pretty_nodes += f"{str(edge).upper()
    }"
45              out += f"{str(node).upper()} -> {
    pretty_nodes}\n"
46
47          return out
48
49      def __len__(self):
50          return len(self.graph)
51
52      def __getitem__(self, item):
53          return self.graph[item]
54
55      def __eq__(self, other):
56          return self.graph == other
57
58      def to_dot(self, *, n: int = 0, title="graph"):
59          """prints dot representation of graph"""
60          dot_source = ""
61          for src, adj_list in self.graph.items():
62              for edge in adj_list:
63                  dot_source += f"{str(src)} -> {str(
    edge.node)} {edge.to_dot()}\n"
64              if not adj_list:
65                  dot_source += f"{src}\n"
66
67          dot = graphviz.Source(f"digraph {{ {
    dot_source} }}")
68          dot.format = "svg"
69          dot.render(f"./graph_renders/{title}{n}")
70
71      def nodes(self) -> list[Vertex]:
72          """Returns nodes in the graph"""
73          return list(self.graph.keys())
74
```

```
 75     @staticmethod
 76     def edge_from_nodes(node: Vertex, list_: list[
    Edge]) -> Edge:
 77         """Checks if node is in a list of edges,
    will return relevant edge if found
 78         usage:"""
 79         for edge in list_:
 80             if edge.node == node:
 81                 return edge
 82             else:
 83                 continue
 84 
 85         raise GraphError("Node not in list")
 86 
 87     @staticmethod
 88     def nodes_from_edges(edges: list[Edge]) -> list
    [Vertex]:
 89         nodes = []
 90         for edge in edges:
 91             nodes.append(edge.node)
 92         return nodes
 93 
 94     @staticmethod
 95     def sanitize(v: Vertex, *args) -> bool:
 96         """Raises GraphGenError if args are not
    Vertex"""
 97         if args is not None:
 98             tests = [v, *args]
 99         else:
100             tests = [v]
101         for test in tests:
102             if type(test) is not Vertex:
103                 raise GraphError(f"{test} if of
    type {type(test)} not Vertex ")
104 
105         return True
106 
107     def is_node(self, v: Vertex) -> bool:
108         return v in self.graph
109 
110     def add_node(self, v: Vertex) -> None:
```

```
111         self.graph[v] = []
112         self._backwards_graph[v] = []
113 
114     def pop_node(self, v: Vertex) -> dict[Vertex,
    list[Edge]]:
115         """Pops node, returns key/value pair of
    node and previous connections"""
116         # look at _backwards_graph to find
    associations
117         for edge in self._backwards_graph[v]:
118             # removing B from A and C; A's pass
119             pointing_node_neighbours = self.
    _backwards_graph[
120                 edge.node
121             ]  # [Edge(A), Edge(C)]
122             pointing_edge = self.edge_from_nodes(v
    , self.graph[edge.node])
123             self.graph[edge.node].remove(
    pointing_edge)  # type: ignore
124 
125         return {v: self.graph.pop(v)}
126 
127     def is_edge(self, s: Vertex, t: Vertex) -> int:
128         """Checks for an edge between nodes (
    directional: s->t !=> t->s)"""
129         try:
130             self.edge_from_nodes(t, self.graph[s])
131             return 1
132         except GraphError:
133             return 0
134 
135     def pop_edge(self, s: Vertex, t: Vertex) ->
    Edge:
136         self.sanitize(s, t)
137         edge = self.edge_from_nodes(t, self.graph[s
    ])
138 
139         if edge is None:
140             raise GraphError("Cannot pop edge that
    doesnt exist")
141 
```

```
142         self.graph[s].remove(edge)
143         return edge
144 
145     def neighbours(self, node: Vertex) -> list[Edge
    ]:
146         self.sanitize(node)
147         return self[node]
148 
149     def connected(self, node: Vertex) -> bool:
150         """Returns true if node has connections to
    graph"""
151         forwards = self[node]
152         backwards = self._backwards_graph[node]
153         return True if forwards or backwards else
    False
154 
155 
156 class Digraph(GenericDigraph):
157     def add_edge(self, s: Vertex, *args: Vertex
    ) -> None:
158         self.sanitize(s, *args)
159         for target in args:
160             self.graph[s].append(Edge(target))
161             self._backwards_graph[target].append(
    Edge(s))
162 
```

```python
# coding=utf-8
from dataclasses import dataclass

from src.simplify.base_graph import GenericDigraph, GraphError
from src.simplify.graph_objects import Vertex

"""Flow Graph"""


class FlowEdgeError(Exception):
    ...


class EdgeCapacityZero(Exception):
    ...


@dataclass(init=False, repr=False, eq=False)
class FlowEdge:
    def __init__(self, src: Vertex, node: Vertex, capacity: int, flow: int = 0):
        self.src: Vertex = src
        self.node: Vertex = node  # where is edge pointing
        self.capacity: int = capacity  # non -ve;
        self.flow: int = flow  # always initialised to 0
        self.residual: bool = not capacity

    def __str__(self):
        return (
            f"{self.node} [{self.flow}/{self.capacity}], "
            if not self.residual
            else f"R: {self.node} [{self.flow}/{self.capacity}],"
        )

    def __repr__(self):
        return self.__str__()
```

```
36
37      def __eq__(self, other):
38          # TODO: write this properly
39          return str(self) == str(other)
40
41      def __lt__(self, other):
42          return self.capacity < other.capacity
43
44      def to_dot(self):
45          base = f'[label="  {self.flow}/{self.
    capacity}  "]'
46          return base[:-1] + ", color=red]" if self.
    residual else base
47
48      def unused_capacity(self):
49          """Returns unused capacity of the edge"""
50          return self.capacity - self.flow
51
52      def push_flow(self, flow: int):
53          """Pushes flow down an edge; raises error if
     too much"""
54          current = self.flow
55          if (new := flow + current) > self.capacity:
56              print(new, self.unused_capacity())
57              # raise error
58              raise FlowEdgeError(
59                  f"Tried to add {flow} units of flow"
60                  f" to an edge with {self.
    unused_capacity()} units of flow available, "
61                  f"totalling {new}/{self.capacity}
    units"
62              )
63          else:
64              self.flow = new
65
66      def adjust_edge(self):
67          """Changes edge values so that capacity =
    unused_capacity, flow = 0"""
68          if self.residual:
69              # reset residual edge
70              self.flow = 0
```

```
71
72          else:
73              # make capacity = to what was unused
    capacity, unless unused capacity 0;
74              # if unused capacity 0, raise
    EdgeCapacityZero
75              if new := self.unused_capacity():
76                  self.capacity = new
77                  self.flow = 0
78              else:
79                  raise EdgeCapacityZero(self, new)
80
81
82  class FlowGraph(GenericDigraph):
83      def __init__(self, vertices: list[Vertex]):
84
85          super().__init__(vertices)
86
87          # map to keep track of people's net debts
    ; +ve if owes group, -ve if owed by group
88          self.net_debt: dict[Vertex, int] = {node: 0
     for node in vertices}
89
90      def __bool__(self):
91          """Returns true if not empty"""
92          return not ({node: [] for node in self.
    nodes()} == self.graph)
93
94      def __eq__(self, other):
95          return str(self) == str(other)
96
97      @staticmethod
98      def edge_from_nodes(node: Vertex, list_: list[
    FlowEdge]) -> FlowEdge:  # type: ignore
99          """gets edge from a list of edges (i.e. an
    adjacency list) by node;
100         doesn't discriminate against residual edges
    """
101
102         return GenericDigraph.edge_from_nodes(node
    , list_)  # type: ignore
```

```
103
104     def get_edge(self, src: Vertex, dest: Vertex
    ) -> FlowEdge:
105         """Gets edge in graph between two nodes,
    residual edges included"""
106         for node in [src, dest]:
107             self.sanitize(node)
108
109         return GenericDigraph.edge_from_nodes(dest
    , self[src])  # type: ignore
110
111     def is_edge(self, s: Vertex, t: Vertex, *,
    residual=False) -> bool:
112         """Checks for edge in a graph; residual =
    True allows broadening to include residual edges"""
113         try:
114             edge = self.edge_from_nodes(t, self[s])
115             if residual:
116                 return True
117             else:
118                 return False if edge.residual else
    True
119
120         except GraphError:
121             return False
122
123     def add_edge(self, src: Vertex, *edges: tuple[
    Vertex, int], update_debt=True):
124         """Add a FlowEdge to graph, and also add a
    residual edge"""
125         for dest, capacity in edges:
126             # make sure nodes in graph
127             self.sanitize(src, dest)
128
129             # no existing edge between two nodes
130             if not (self.is_edge(src, dest) or self
    .is_edge(dest, src)):
131                 # add normal edge
132                 self[src].append(FlowEdge(src, dest
    , capacity))
133                 # add residual edge
```

```
134                 self[dest].append(FlowEdge(dest,
    src, 0))
135
136                 self[src].sort(reverse=True)
137
138             # edge going in direction of edge being
     added
139             elif self.is_edge(src, dest):
140                 self.get_edge(src, dest).capacity
     += capacity
141
142             elif self.is_edge(dest, src, residual=
    False):
143                 new_cap = self.get_edge(dest, src).
    capacity - capacity
144                 # if new capacity is 0 pop edge
145
146                 if new_cap > 0:
147                     self.get_edge(dest, src).
    capacity = new_cap
148                 if new_cap < 0:
149                     self.pop_edge(dest, src)
150                     self.add_edge(src, (dest,
    new_cap * -1))
151                 if new_cap == 0:
152                     self.pop_edge(dest, src)
153
154
155             if update_debt:
156                 # handle net_debt;
157                 self.net_debt[src] += capacity
158                 self.net_debt[dest] -= capacity
159
160     def pop_edge(self, src: Vertex, dest: Vertex
    , *, update_debt=False):
161         """removes REAL edges, and deletes residual
     counterpart"""
162
163         # normal
164         fwd_edge = self.edge_from_nodes(dest, self[
    src])
```

```
165         self[src].remove(fwd_edge)
166         # residual
167         res = self.edge_from_nodes(src, self[dest])
168         if res.node == src:
169             # assert self[dest][0] == self[dest][1]
170             self[dest].remove(res)
171 
172         else:
173             raise ValueError(
174                 f"Tried to pop residual edge;
    intended {dest} -> {src}, got {dest} -> {res.node}"
175             )
176 
177         if update_debt:
178             # handle net_debt;
179             self.net_debt[src] -= fwd_edge.capacity
180             self.net_debt[dest] += fwd_edge.
    capacity
181 
182     def flow_neighbours(self, node: Vertex):
183         """returns neighbours of a node including
    those related by residual edge
184         edges are valid iff they have unused
    capacity"""
185 
186         # build list of edges from adj list with
    unused capacity
187         return [edge for edge in self[node] if edge
    .unused_capacity()]
188 
189     def adjust_edges(self):
190         """Run edge adjust on each edge, if new
    capacity = 0 and residual = false,
191         delete ede + residual counterpart"""
192 
193         edge: FlowEdge
194 
195         for n, node in enumerate(self.nodes()):
196             for edge in self[node]:
197                 try:
198                     edge.adjust_edge()
```

```
199                 except EdgeCapacityZero:
200                     # edge no longer has a place in
     my graph
201                     # delete edge + residual; will
    condition only ever raised on fwd edges
202                     self.pop_edge(node, edge.node)
203
204     #
205     # def nodes(self):
206     #     """Returns nodes in order of incoming
    edges (smallest first)"""
207     #     def incoming_func(node):
208     #         return len([edge for edge in self[
    node] if edge.residual])
209     #
210     #     incoming = [(node, incoming_func(node))
    for node in self.graph.keys()]
211     #     incoming.sort(key=lambda row: row[1],
    reverse=True)
212     #     return [node for node, _ in incoming
213
```

```python
# coding=utf-8

from dataclasses import dataclass


"""All objects to be used in various graphs"""


class FlowError(Exception):
    ...


@dataclass
class Vertex:
    """Representation of a vertex; carries data and ID"""

    ID: int  # IDs should be unique
    label: str = ""  # optional label

    def _key(self) -> tuple:
        """Returns immutable repr of object for hashing"""
        return self.ID, self.label

    def __str__(self):
        return self.label if self.label else str(self.ID)

    def __hash__(self):
        """for adding to lists / dicts"""
        return hash(bytes(f"{self._key()}".encode("utf8")))


@dataclass
class Edge:
    node: Vertex

    def __str__(self):
        return str(self.node)
```

```
38
39      def to_dot(self):
40          """str repr for dot"""
41          return ""
42
43
44  @dataclass
45  class WeightedEdge(Edge):
46      weight: int
47
48      def __str__(self):
49          return f"{self.node} [{self.weight}], "
50
51      def to_dot(self):
52          return f"[label={self.weight}"
53
54
55  @dataclass
56  class FlowEdge(Edge):
57      capacity: int = 0
58      flow: int = 0
59      residual: bool = False  # class as residual edge
     if capacity is 0
60
61      def __str__(self):
62          return (
63              f"{self.node} [{self.flow}/{self.
    capacity}], " if not self.residual else ""
64          )
65
66      def to_dot(self):
67          base = f'[label="{self.flow}/{self.capacity}
    "]'
68          return base[:-1] + ", color=red]" if self.
    residual else base
69
70      def unused_capacity(self) -> int:
71          return self.capacity - self.flow
72
73      def push_flow(self, flow):
74          if self.flow + flow > self.capacity or type(
```

```
flow) is not int:
            raise FlowError(
                f"Cannot send {flow} units down path of capacity {self.capacity}"
            )
        self.flow += flow

```

```python
# coding=utf-8

import copy

from src.simplify import path as path
from src.simplify.flow_graph import FlowGraph, FlowEdge
from src.simplify.graph_objects import Vertex


class SettleError(Exception):
    ...


class NoOptimisations(Exception):
    """No optimisations to graph; already in simplest form"""


class MaxFlow:
    @staticmethod
    def edmonds_karp(graph: FlowGraph, src: Vertex, sink: Vertex) -> int:

        max_flow = 0

        while aug_path := MaxFlow.augmenting_path(graph, src, sink):
            bottleneck = MaxFlow.bottleneck(graph, aug_path)
            max_flow += bottleneck

            MaxFlow.augment_flow(graph, aug_path, bottleneck)
            # graph.to_dot()

        return max_flow

    @staticmethod
    def augmenting_path(graph: FlowGraph, src: Vertex, sink: Vertex) -> list[Vertex]:
```

```
35          """find the shortest path from src -> sink
    using BFS"""
36          return path.Path.shortest_path(
37              graph, src, sink, neighbours=graph.
    flow_neighbours
38          )
39
40      @staticmethod
41      def bottleneck(graph: FlowGraph, node_path: list
    [Vertex]) -> int:
42          """Returns bottleneck of a path"""
43          aug_path = MaxFlow.nodes_to_path(graph,
    node_path)
44          remaining = [edge.unused_capacity() for edge
     in aug_path]
45          return min(remaining)
46
47      @staticmethod
48      def augment_flow(graph: FlowGraph, node_path:
    list[Vertex], flow: int) -> None:
49          """Adds flow to normal edges of path,
    subtracts from residual
50          deals with pushing to residual and hence
    subtracting from normal"""
51          # normal edges
52          aug_path = MaxFlow.nodes_to_path(graph,
    node_path)
53          for edge in aug_path:
54              edge.push_flow(flow)
55
56          # flip node path to give residual
57          node_path.reverse()
58          # get edges from reversed path
59          res_path = MaxFlow.nodes_to_path(graph,
    node_path)
60
61          # push flow down res path
62          for edge in res_path:
63              edge.push_flow(flow * -1)
64
65      @staticmethod
```

```
66      def nodes_to_path(graph: FlowGraph, nodes: list
    [Vertex]) -> list[FlowEdge]:
67          graph.sanitize(*nodes)
68
69          edges: list[FlowEdge] = []
70          for src, neighbour in zip(nodes, nodes[1
    :]):
71              edges.append(graph.get_edge(src,
    neighbour))
72
73          return edges
74
75
76  class Simplify:
77      @staticmethod
78      def simplify_debt(debt: FlowGraph) -> FlowGraph
    :
79          """
80          for edge(u, v) in graph:
81              if new := maxflow(u, v):
82                  clean.add_edge(u, (v, new))
83                  messy.adjust_edges()
84          """
85
86          clean = FlowGraph(debt.nodes())
87
88          d_cache = copy.deepcopy(debt)
89
90          # iterate through edges in graph:
91          while not not debt:
92              edge: FlowEdge  # type: ignore
93              for (node, adj_list) in debt.graph.
    items():
94
95                  for edge in adj_list:  # type:
    ignore
96                      if not edge.residual:
97                          if flow := MaxFlow.
    edmonds_karp(debt, node, edge.node):
98                              clean.add_edge(node, (
    edge.node, flow))
```

```
 99                             debt.adjust_edges()
100
101         if clean == d_cache:
102             clean.to_dot(n=1)
103             raise NoOptimisations
104
105         if d_cache.net_debt != clean.net_debt:
106             raise SettleError("Settling failed;
    debt was skewed")
107
108         return clean
109
```

```python
# coding=utf-8

from src.simplify.base_graph import GenericDigraph, GraphError
from src.simplify.graph_objects import Vertex, WeightedEdge


class WeightedDigraph(GenericDigraph):
    def add_edge(self, source: Vertex, *edges: tuple[Vertex, int]) -> None:

        # sanitize source
        self.sanitize(source)
        for node, weight in edges:
            if self.is_edge(source, node):
                existing: WeightedEdge
                existing = self.edge_from_nodes(node, self[source])  # type: ignore
                existing.weight += weight
            else:
                self.sanitize(node)
                self.graph[source].append(WeightedEdge(node, weight))
                self._backwards_graph[node].append(WeightedEdge(source, weight * -1))

    def flow_through(self, node: Vertex) -> int:
        """Returns sum of weights out of node
        +ve => net flow out of node
        -ve => net flow into node"""
        flow = 0
        edge: WeightedEdge

        for graph in [self.graph, self._backwards_graph]:
            for edge in graph[node]:  # type: ignore
                flow += edge.weight

        # return -ve, as backwards edges have -ve value and forwards have +ve
```

```
34          return flow
35
36      def net_debts(self) -> dict[Vertex, int]:
37          """Returns a map of everyone with net money
    owed (-ve if they need to pay)"""
38          return {node: self.flow_through(node) for
    node in self.nodes()}
39
40      def is_edge(self, s: Vertex, t: Vertex) -> int:
41          try:
42              edge: WeightedEdge
43              edge = self.edge_from_nodes(t, self.
    graph[s])  # type: ignore
44              return edge.weight
45          except GraphError:
46              return 0
47
```

```python
# coding=utf-8

import csv
import os
from dataclasses import dataclass, field

import src.simplify.flow_algorithms
import src.simplify.flow_graph as flow
from src.crypto import keys
from src.simplify.flow_algorithms import Simplify, SettleError
from src.simplify.graph_objects import Vertex
from src.transactions.transaction import Transaction, VerificationError


class LedgerBuildError(Exception):
    """Error building ledger"""


class NoFutherSimplifications(Exception):
    ...


@dataclass
class Ledger:
    """Multiple transactions contained to one group (assumed from building);
    built from a stream of transaction objects"""

    # ledger, big list of transactions;
    # TODO: maybe make ledger generator
    ledger: list[Transaction] = field(default_factory=lambda: [])
    nodes: list[Vertex] = field(default_factory=lambda: [])
    key_map: dict[int, keys.RSAPublicKey] = field(default_factory=lambda: {})

    def __bool__(self):
        """False if ledger empty"""
```

```
36          return not not self.ledger
37
38      def __eq__(self, other):
39          return self.ledger == other.ledger
40
41      def append(self, transaction: Transaction) ->
    list[Transaction]:
42          """Nice syntax for adding transactions to
    ledger"""
43
44          if type(transaction) is not Transaction:
45              raise LedgerBuildError(
46                  f"cannot append type {transaction}
    to ledger; must be transaction"
47              )
48          else:
49              self.ledger.append(transaction)
50              self.key_map[transaction.src] =
    transaction.src_pub
51              self.key_map[transaction.dest] =
    transaction.dest_pub
52
53          return self.ledger
54
55      def _verify_transactions(self) -> None:
56          """Verifies the keys of all the transactions
     in the group.
57          Raises error if a faulty transaction is
    found"""
58
59          for trn in self.ledger:
60              trn.verify()
61
62      def _as_flow(self) -> flow.FlowGraph:
63          """Returns ledger as a flow graph"""
64          # Extract IDs involved -> nodes
65          nodes: set[Vertex] = set()
66          for trn in self.ledger:
67              nodes.add(Vertex(trn.src))
68              nodes.add(Vertex(trn.dest))
69
```

```
 70         self.nodes = list(nodes)
 71         self.nodes.sort(key=lambda node: node.ID)
 72 
 73         # build flow graph with nodes
 74         as_flow = flow.FlowGraph(self.nodes)
 75 
 76         # verify transaction, add to graph
 77         for trn in self.ledger:
 78             trn.verify()
 79             as_flow.add_edge(Vertex(trn.src), (
    Vertex(trn.dest), trn.amount))
 80 
 81         return as_flow
 82 
 83     def _flow_to_transactions(self, fg: flow.
    FlowGraph) -> list[Transaction]:
 84         """For each edge, make a transaction"""
 85 
 86         new_trns: list[Transaction] = []
 87 
 88         edge: flow.FlowEdge
 89 
 90         # go through every outgoing transaction by
    person
 91         for person, adj_list in fg.graph.items():
 92             for edge in adj_list:  # type: ignore
 93 
 94                 # don't add residual edges to
    ledger
 95                 if edge.residual:
 96                     continue
 97 
 98                 else:
 99                     # generate UNSIGNED
    transactions
100                     # TODO: let db handle id; keep
    it initialised to 0
101                     edge: flow.FlowEdge  # type:
    ignore
102                     trn = Transaction(
103                         edge.src.ID,
```

```
104                             edge.node.ID,
105                             edge.capacity,
106                             self.key_map[edge.src.ID],
107                             self.key_map[edge.node.ID],
108                         )
109
110                         new_trns.append(trn)
111
112         return new_trns
113
114     def simplify_ledger(self):
115         # build ledger as a flow graph
116         fg = self._as_flow()
117         fg.to_dot(title="pre_settle")
118         try:
119             simplified_fg = Simplify.simplify_debt(
    fg)
120
121             # settle, update ledger
122             simplified_fg.to_dot(title="settled")
123             self.ledger = self.
    _flow_to_transactions(simplified_fg)
124
125         except SettleError:
126             # no changes made to graph, keep ledger
     as is, with sigs.
127             # print(
128             #     "Graph already at few
    transactions per person; no optimisations found"
129             # )
130             raise SettleError('Failed to settle
    graph... aborted')
131
132         except src.simplify.flow_algorithms.
    NoOptimisations:
133             raise NoFutherSimplifications(
134                 "Graph already at few transactions
    per person; no optimisations found"
135             )
136
137         except VerificationError as ve:
```

```
138             # let verification error propagate up
139             raise ve
140 
141 
142 class LedgerLoader:
143     @staticmethod
144     def load_from_csv(path: str) -> list[Ledger]:
145         """Load from a csv, in transaction format
    """
146 
147         print("loading from csv")
148 
149         def get_field(str_: str) -> int:
150             return header.index(str_)
151 
152         def build_trn() -> tuple[
153             int, int, int, keys.RSAPublicKey, keys.
    RSAPublicKey, int
154         ]:
155 
156             ldr = keys.RSAKeyLoaderFromNumbers()
157             ldr.load(n=int(row[get_field("src_n"
    )]), e=int(row[get_field("src_e")]))  # type:
    ignore
158             src_pub: keys.RSAPublicKey = keys.
    RSAPublicKey(ldr)
159 
160             ldr2 = keys.RSAKeyLoaderFromNumbers()
161             ldr2.load(n=int(row[get_field("dest_n"
    )]), e=int(row[get_field("dest_e")]))  # type:
    ignore
162             dest_pub: keys.RSAPublicKey = keys.
    RSAPublicKey(ldr2)
163 
164             print(src_pub, dest_pub, "---", sep="\n
    ")
165 
166             return (
167                 int(row[get_field("src")]),
168                 int(row[get_field("dest")]),
169                 int(row[get_field("amount")]),
```

```
170                 src_pub,
171                 dest_pub,
172                 int(row[get_field("ID")]),
173             )
174 
175         if not os.path.exists(path):
176             raise FileNotFoundError(
177                 f"File not found at current path: \
    n{os.getcwd()};\nsearched for {path}"
178             )
179 
180         transactions: list[list[Transaction]] = []
181 
182         # generate transaction objects, store as
    list of groups of transactions
183         with open(path) as csvfile:
184             transaction_reader = csv.reader(csvfile
    , delimiter=",")
185             for row in transaction_reader:
186                 # use header to build index of
    where things are
187                 if row[0] == "ID":
188                     header: list[str] = row
189                     continue
190 
191                 # build transaction, keep groups
    intact
192 
193                 trn = Transaction(*build_trn())
194 
195                 try:
196                     transactions[int(row[get_field(
    "group")])].append(trn)
197 
198                 except IndexError:
199                     # make position at group if not
     made yet (assuming consecutive 0 indexed group
    numbers
200                     transactions.append([trn])
201 
202         ledgers: list[Ledger] = []
```

```
203
204         for group in transactions:
205             ledger = Ledger()
206             for trn in group:
207                 ledger.append(trn)
208             ledgers.append(ledger)
209
210         return ledgers
211
```

```python
# coding=utf-8

"""
Handles transaction object
"""

import datetime

# coding=utf-8
import sys
from abc import ABC, abstractmethod
from dataclasses import dataclass, field

from src.crypto import keys, hashes, rsa


class TransactionError(Exception):
    ...


class VerificationError(Exception):
    ...


class Signable(ABC):
    """Base class for objects that can be signed; needs a hash implementation, cannot be added into ABC for reasons"""

    @abstractmethod
    def sign(self, key: keys.RSAPrivateKey, *, origin: str) -> None:
        ...

    @abstractmethod
    def hash(self) -> bytes:
        ...


@dataclass
class Transaction(Signable):
```

```
39      src: int
40      dest: int
41      amount: int
42      src_pub: keys.RSAPublicKey
43      dest_pub: keys.RSAPublicKey
44      ID: int = 0
45      reference: str = ""
46      time: datetime.datetime = datetime.datetime.now
    ()
47      signatures: dict[int, bytes] = field(
    default_factory=lambda: {})
48      group: int = 0
49
50      def hash(self) -> bytes:
51          # standard way to produce hash using SHA256
    interface from crypto lib
52          return (
53              hashes.Hasher(
54                  f"{self.src, self.dest, self.amount
    , self.reference, self.time}".encode(
55                      "utf8", sys.byteorder
56                  )
57              )
58              .digest()
59              .h
60          )
61
62      # note: IMPORTANT: CANNOT USE DUNDER HASH AS
    PYTHON DOES WEIRD HASH COMPRESSION
63      def __hash__(self):
64          raise NotImplementedError("__hash__ cannot
    be used, used .hash() method")
65
66      def sign(self, key: keys.RSAPrivateKey, *,
    origin: str) -> None:
67          def overwrite_err(where: str) -> None:
68              """Helper to raise overwrite error"""
69              raise TransactionError(f"Cannot
    overwrite signature of {where}")
70
71          def generate_signature(obj: Transaction) ->
```

```
 71 bytes:
 72             """Helper to generate a signature of
    the object"""
 73             sig = rsa.RSA.sign(obj.hash(), key)
 74             return sig
 75 
 76         # raise error if not given a priv key
 77         if (keytype := type(key)) is not keys.
    RSAPrivateKey:
 78             raise TransactionError(f"Was given a {
    keytype}, not an RSAPrivateKey")
 79 
 80         # initialise sigs dict if it doesn't
    already exist
 81         if not self.signatures:
 82             self.signatures = {self.src: b"", self.
    dest: b""}
 83         else:
 84             # check that, if only one signature
    exists, the other is created as null
 85             try:
 86                 self.signatures[self.src]
 87             except KeyError:
 88                 self.signatures[self.src] = b""
 89 
 90             try:
 91                 self.signatures[self.dest]
 92             except KeyError:
 93                 self.signatures[self.dest] = b""
 94 
 95         # accept origin as src or dest;
 96         # check for overwrite, raise error if case
 97         # append with key of ID to signatures dict
 98         if origin == "src":
 99             if not self.signatures[self.src]:  #
    check for overwrites
100                 self.signatures[self.src] =
    generate_signature(self)
101             else:
102                 overwrite_err("src")  # handle
    overwrite err
```

```
103
104         elif origin == "dest":
105             if not self.signatures[self.dest]:
106                 self.signatures[self.dest] =
    generate_signature(self)
107             else:
108                 overwrite_err("dest")
109
110         else:
111             # if parameter wasn't src or dest,
    raise error
112             raise ValueError(f"{origin!r} not a
    valid parameter; use 'src' or 'dest'")
113
114     def verify(self) -> None:
115         """Raise verification error if invalid sig
    """
116
117         # ensure that transaction has its two
    signatures:
118         try:
119             for sig in [self.src, self.dest]:
120                 _ = self.signatures[sig]
121         except KeyError:
122             raise VerificationError(f"Has not been
    signed by USRID {sig}")
123
124         # build list of keys to verif
125         usr_keys = [self.src_pub, self.dest_pub]
126
127         if not usr_keys:
128             raise VerificationError("No valid keys
    were passed in")
129
130         hash_from_obj: bytes = self.hash()
131
132         # note: if signature is a hex string
    convert to bytes
133         for s_key, val in self.signatures.items():
134             if type(val) is str:
135                 self.signatures[s_key] = int(val,
```

```
135 16).to_bytes(256, sys.byteorder)
136
137         for key, origin in zip(usr_keys, [self.src
    , self.dest]):
138             hash_from_sig = rsa.RSA.inv_sig(self.
    signatures[origin], key)
139             if hash_from_sig != hash_from_obj:
140                 raise VerificationError(f"{self.src
    } sig invalid")
141
```