Dan’s Auto Shop

Team: The Fast and the not-so-Furious

Presenting:

Max Edwards: Front-end Engineer and Authentication Expert
Jared Hansen: Front-end Engineer and UX Designer
Chase Miller: Back-end Engineer and Unit Test Specialist
Satchel Fausett: Back-end Engineer and Operations Manager

Project Overview - Major Design Decisions

React - It was a gamble to do the front end in React - Jared and Max were the only team members with experience in React, and that experience was very limited. However, they wanted to learn the framework, and knew that investing early in learning it would pay off later on - and it did. It was, in part, this early work in learning React that allowed us to finish a month before the deadline.

Django - The decision to use django was quite simple - every team member had previously completed several projects working in the Django framework, meaning we were all able to contribute meaningfully to the project from day one.

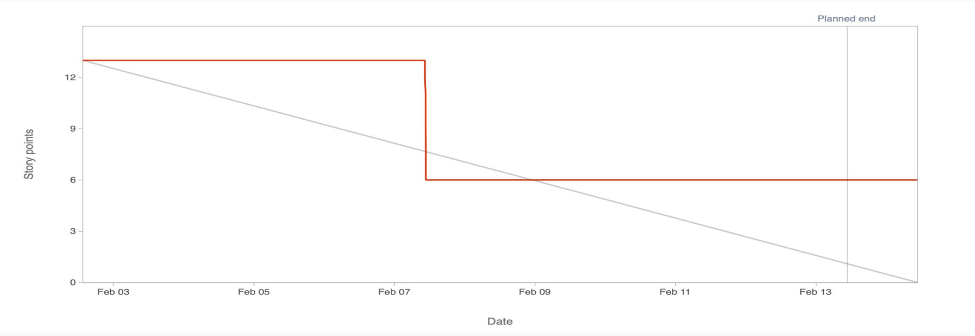
Project Overview - Agile practices

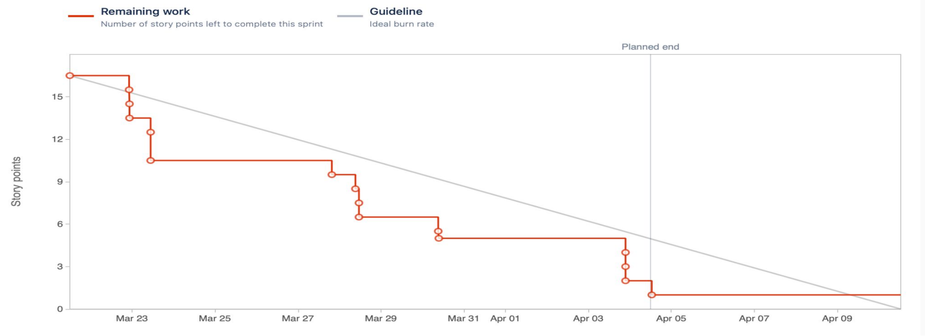
Communication - The most immediate benefit of following agile practices was enhanced communication. In the first sprint, we quickly realized we were not communicating at the necessary level for optimal productivity, and used the first sprint retrospective to correct this fault.

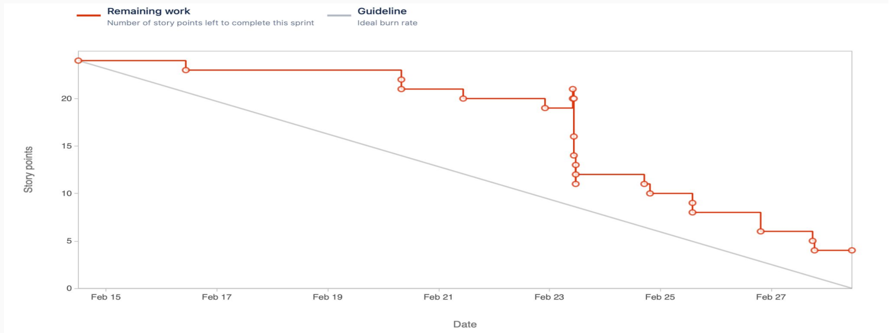
Pacing - Analyzing the project in-depth at the outset, predicting the amount of time each task would take, and dividing those tasks into sprints and then among ourselves empowered us to anticipate how much work we could complete in one sprint. Additionally, it provided us the insight we needed to get ahead and deliver a functional product ahead of time

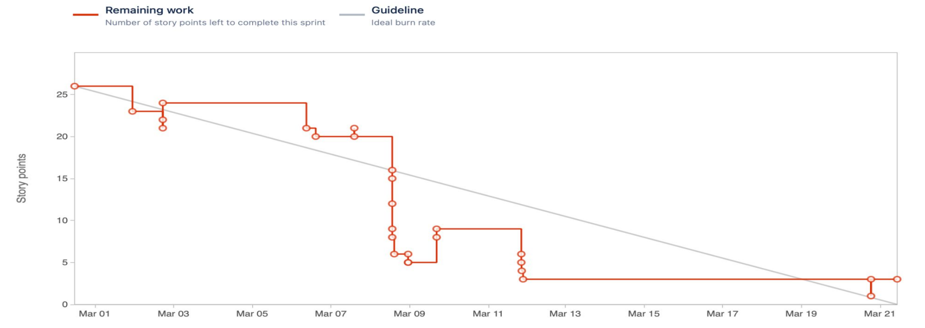
Flexibility - Following the agile process allowed us to take each sprint as it came, allowing team members to accept more or fewer tasks as they were able.

Project Overview - Burndown Chart









Project Overview - Deployment

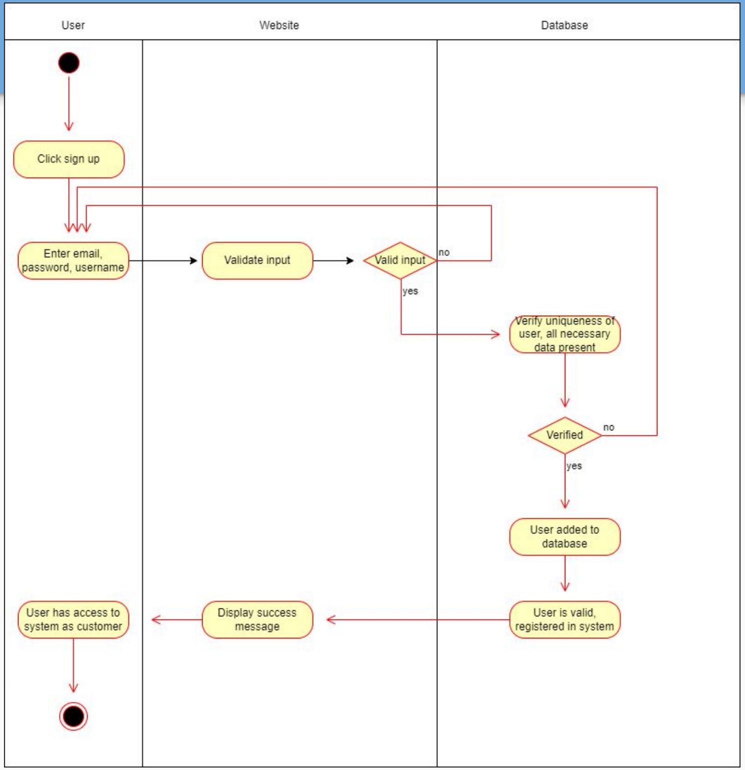
Why the project is ready to be moved to deployment:

1. As team members, we have individually and collectively gone over the requirements specification and affirmed that the requirements are met.
2. We have performed thorough and exhaustive systems testing, and are confident that we have developed functional, robust software.
3. We have implemented over 40 unit tests to ensure current and future quality of software.

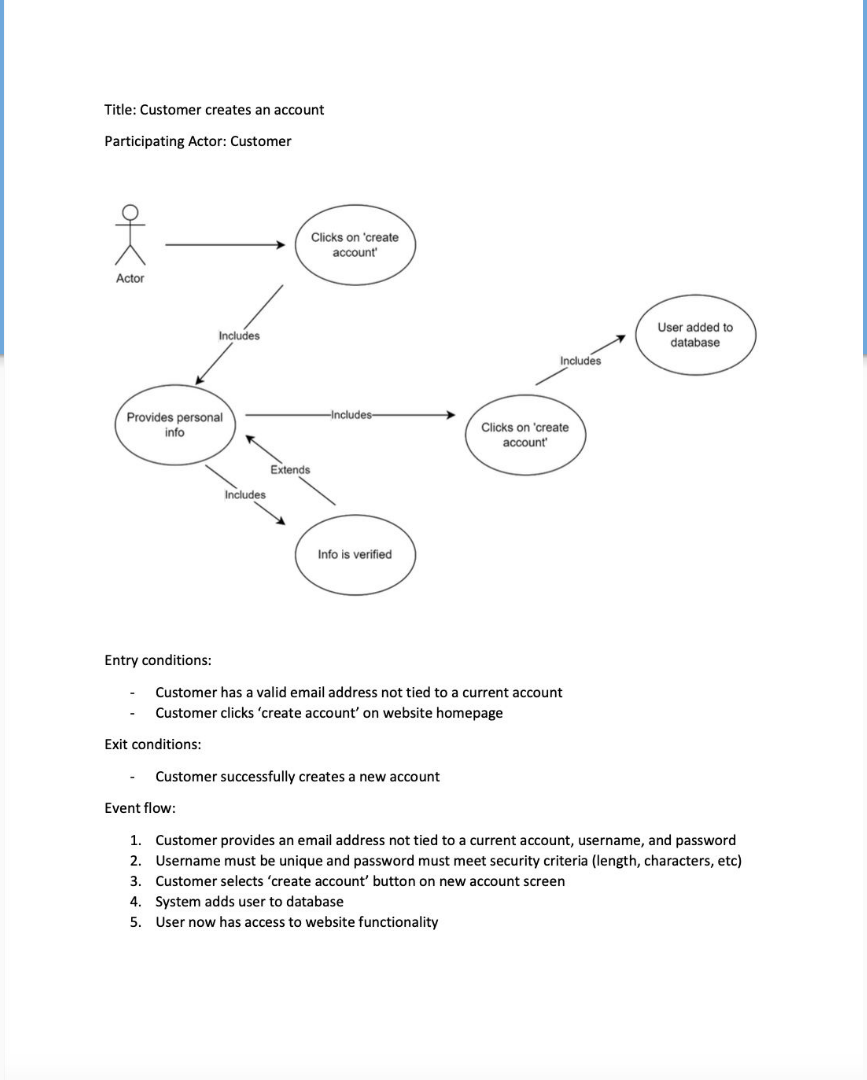
Requirement 1

- 4.7 Customer creates an account has money, email, phone, etc.
- Functionality - User may create an account with their personal info and create a password.
- Usability - clear, easy to use user interface that is a simple form to fill out
- Reliability - The interface works. A user should be able to create an account as long as their email is unique.
- Performance - Fast, doesn't take long to load them into their new account page
- Supportability - Tested both functionally and with unit tests on the backend. It should be easy to create a user and find them in the database.

Diagrams



User
Website
Database
Click sign up
Enter email, password, username
Validate input
Valid input
no
yes
Verify uniqueness of user, all necessary data present
Verified
no
yes
User added to database
User is valid, registered in system
Display success message
User has access to system as customer



Title: Customer creates an account

Participating Actor: Customer

Actor
Clicks on 'create account'
Includes
Provides personal info
Includes
Clicks on 'create account'
Includes
User added to database
Extends
Includes
Info is verified

Entry conditions:

- Customer has a valid email address not tied to a current account
- Customer clicks 'create account' on website homepage

Exit conditions:

- Customer successfully creates a new account

Event flow:

1. Customer provides an email address not tied to a current account, username, and password
2. Username must be unique and password must meet security criteria (length, characters, etc)
3. Customer selects 'create account' button on new account screen
4. System adds user to database
5. User now has access to website functionality

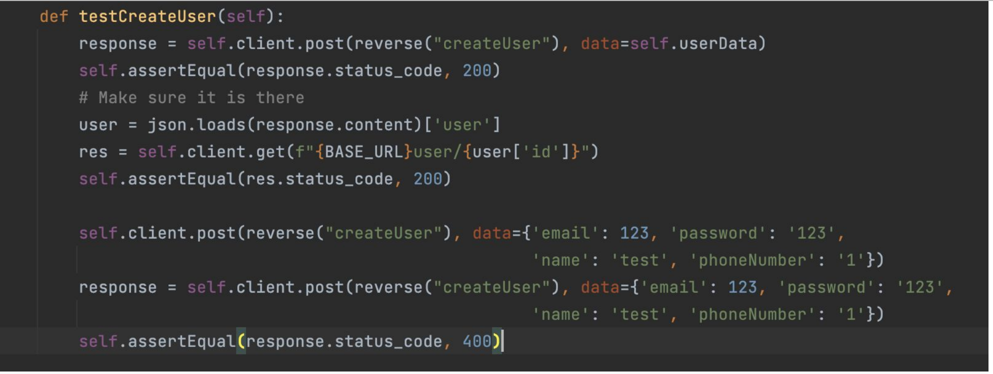
Scrum Tasks Requirement 1

- Issue 18 Backend: create user and vehicle class receiver POST. - Chase Miller, Max Edwards
- Issue 41 Frontend: create an account interface. - Max Edwards
- Issue 52: Create unit tests for authentication, create user, update user, delete user, and get user. - Max Edwards, Chase Miller

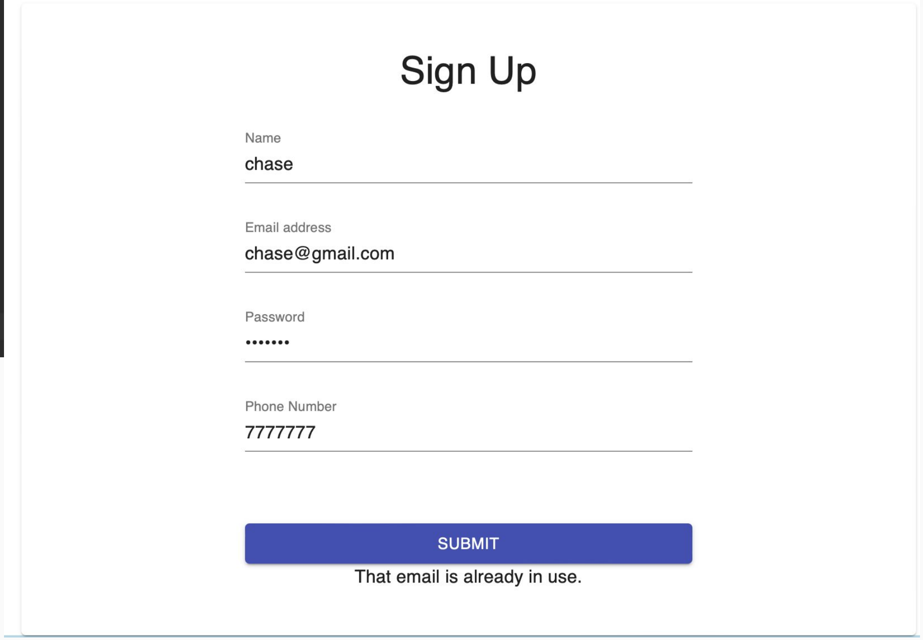
Tests Requirement 1

- We tested the requirement of 4.7 Customer creates an account has money, email, phone, etc. Through functional testing and unit testing.
- The unit testing was done through the django server testing environment.
- We wrote a test that sent a create user call to our database with both good data, and bad data. The correct response codes were then returned by the django server.
- Functional testing, a user may create an account as long as their email is unique. We created an account then tried to create a new account with the same email.

Tests Requirement 1



```
def testCreateUser(self):
    response = self.client.post(reverse("createUser"), data=self.userData)
    self.assertEqual(response.status_code, 200)
    # Make sure it is there
    user = json.loads(response.content)['user']
    res = self.client.get(f"{BASE_URL}user/{user['id']}")
    self.assertEqual(res.status_code, 200)

    self.client.post(reverse("createUser"), data={'email': 123, 'password': '123',
                                                 'name': 'test', 'phoneNumber': '1'})
    response = self.client.post(reverse("createUser"), data={'email': 123, 'password': '123',
                                                 'name': 'test', 'phoneNumber': '1'})
    self.assertEqual(response.status_code, 400)
```



Sign Up

Name
chase

Email address
chase@gmail.com

Password
•••••••

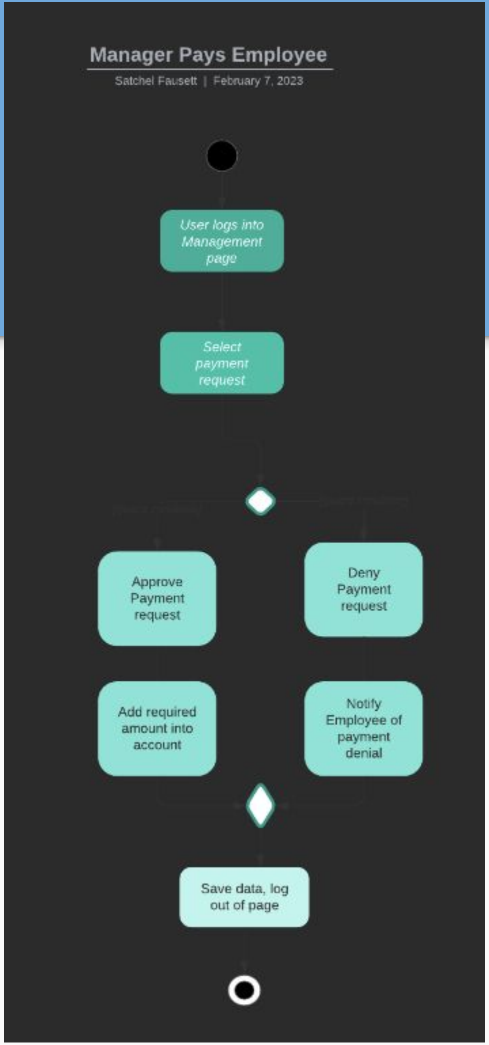
Phone Number
7777777

That email is already in use.

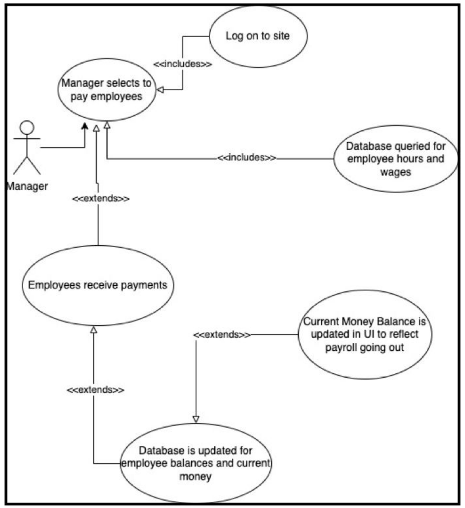
Requirement 2

- 2.6 Supervisor pays employees
- Functionality - Supervisor or manager may view their employees and pay them according to the hours that they have worked.
- Usability - clear, easy to use user interface that is a simple layout that allows the manager to exchange funds from the company wallet to the desired employee.
- Reliability - The interface works. Only a manager should have access to this page. The funds should leave and go into the correct accounts.
- Performance - Fast, doesn't take long to load the new funds.
- Supportability - Tested both functionally and with unit tests on the backend.

Diagrams


Manager Pays Employee
Satchel Fausett | February 7, 2023
User logs into Management page
Select payment request
Approve Payment request
Deny Payment request
Add required amount into account
Notify Employee of payment denial
Save data, log out of page

Title: Supervisor Pays Employees


Log on to site
<<includes>>
Manager selects to pay employees
Manager
Database queried for employee hours and wages
<<includes>>
<<extends>>
Employees receive payments
Current Money Balance is updated in UI to reflect payroll going out
<<extends>>
<<extends>>
Database is updated for employee balances and current money

Participating Actor: Supervisor

Entry Conditions:
- Supervisor signs in
- Supervisor wants to pay his employees

Exit Conditions:
- Supervisor pays employees
- Supervisor decides not to pay and leaves the page
- Supervisor has insufficient funds to pay employees

Event Flow:
1. Supervisor logs in to system
2. Supervisor selects the “Payments” page
3. Supervisor views all employees, their hours, and the amount of money for each
4. Supervisor selected individually or “select all” button
5. Supervisor pays and the money leaves their account balance
6. System displays confirmation of payments

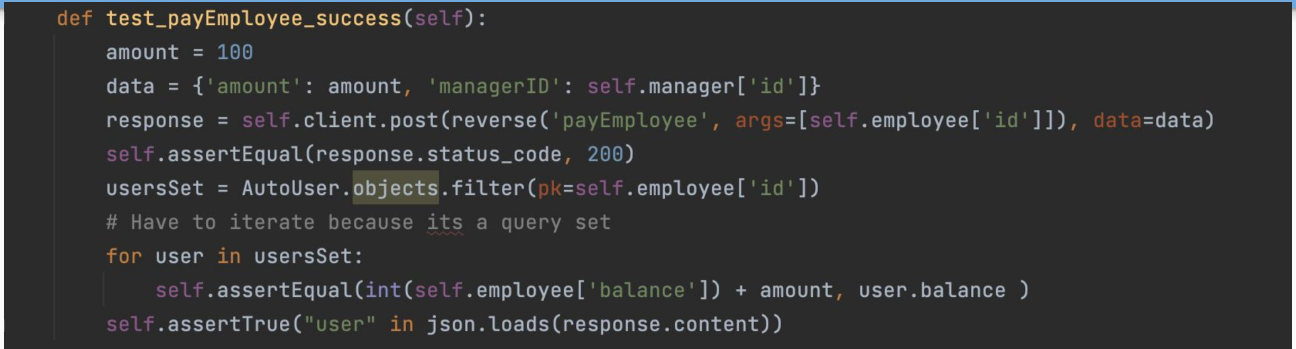
Scrum Tasks Requirement 2

- 15: remove or add money to account
- 20: Add subtract money from specific person id
- 22: employees registering time
- 33: Component for manager to pay employees
-

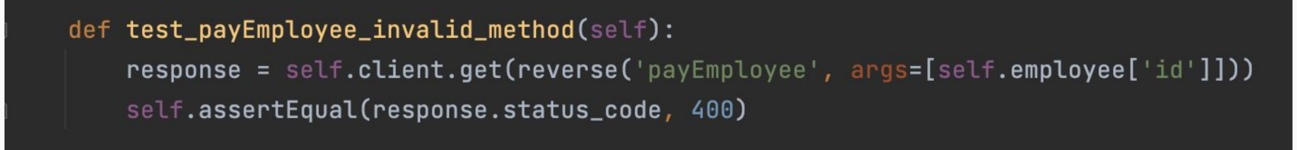
Tests Requirement 2

- We tested the requirement of 2.6 Supervisor pays employees. Through functional testing and unit testing.
- The unit testing was done through the django server testing environment.
- We tested calls to the django server that were correct, missing data, invalid url, invalid employee id. The correct status codes were all returned.
- Functionally we went into the website and payed the employees for the hours that they have logged.

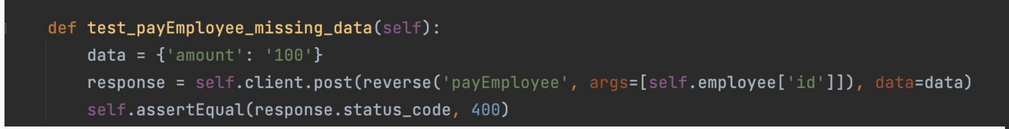
Tests Requirement 2



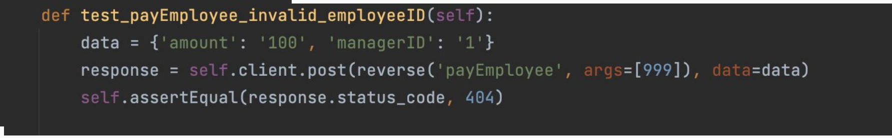
```python
def test_payEmployee_success(self):
    amount = 100
    data = {'amount': amount, 'managerID': self.manager['id']}
    response = self.client.post(reverse('payEmployee', args=[self.employee['id']]), data=data)
    self.assertEqual(response.status_code, 200)
    usersSet = AutoUser.objects.filter(pk=self.employee['id'])
    # Have to iterate because its a query set
    for user in usersSet:
        self.assertEqual(int(self.employee['balance']) + amount, user.balance )
    self.assertTrue("user" in json.loads(response.content))
```



```python
def test_payEmployee_invalid_method(self):
    response = self.client.get(reverse('payEmployee', args=[self.employee['id']]))
    self.assertEqual(response.status_code, 400)
```

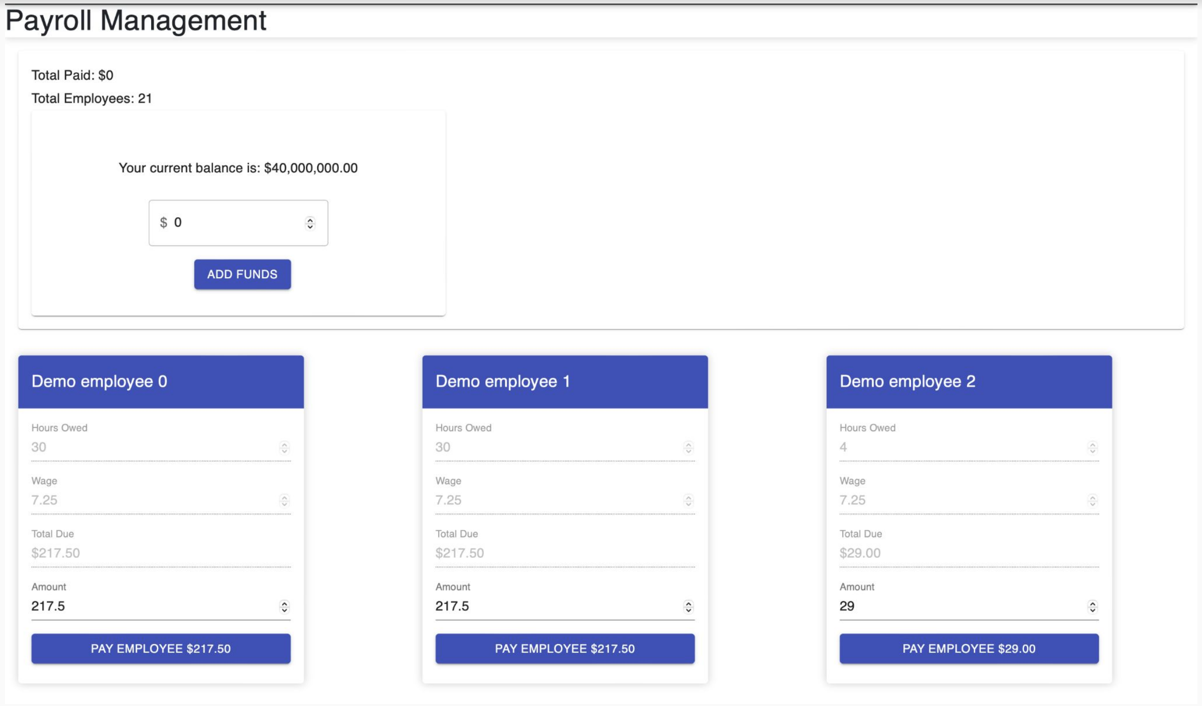


```python
def test_payEmployee_missing_data(self):
    data = {'amount': '100'}
    response = self.client.post(reverse('payEmployee', args=[self.employee['id']]), data=data)
    self.assertEqual(response.status_code, 400)
```



```python
def test_payEmployee_invalid_employeeID(self):
    data = {'amount': '100', 'managerID': '1'}
    response = self.client.post(reverse('payEmployee', args=[999]), data=data)
    self.assertEqual(response.status_code, 404)
```

Tests Requirement 2



Payroll Management

Total Paid: $0

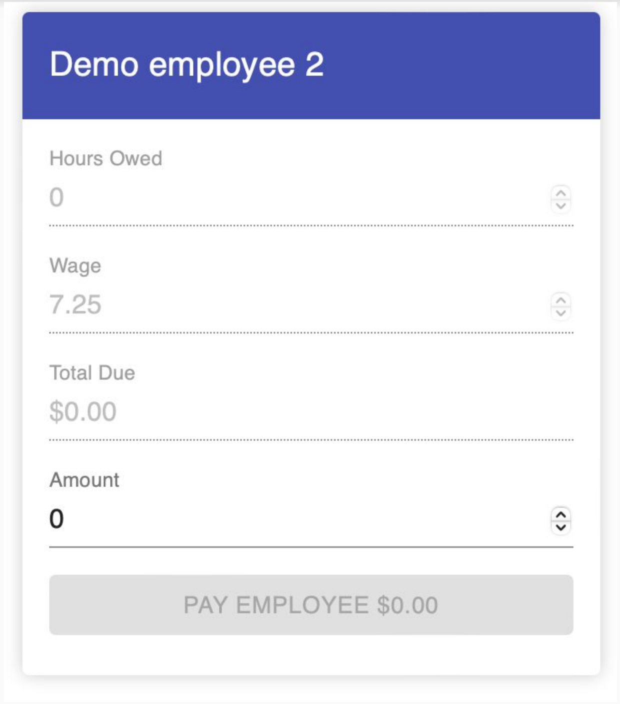
Total Employees: 21

Your current balance is: $40,000,000.00

$ 0

ADD FUNDS

Demo employee 0	Demo employee 1	Demo employee 2
Hours Owed: 30	Hours Owed: 30	Hours Owed: 4
Wage: 7.25	Wage: 7.25	Wage: 7.25
Total Due: $217.50	Total Due: $217.50	Total Due: $29.00
Amount: 217.5	Amount: 217.5	Amount: 29
PAY EMPLOYEE $217.50	PAY EMPLOYEE $217.50	PAY EMPLOYEE $29.00



Demo employee 2

Hours Owed
0

Wage
7.25

Total Due
$0.00

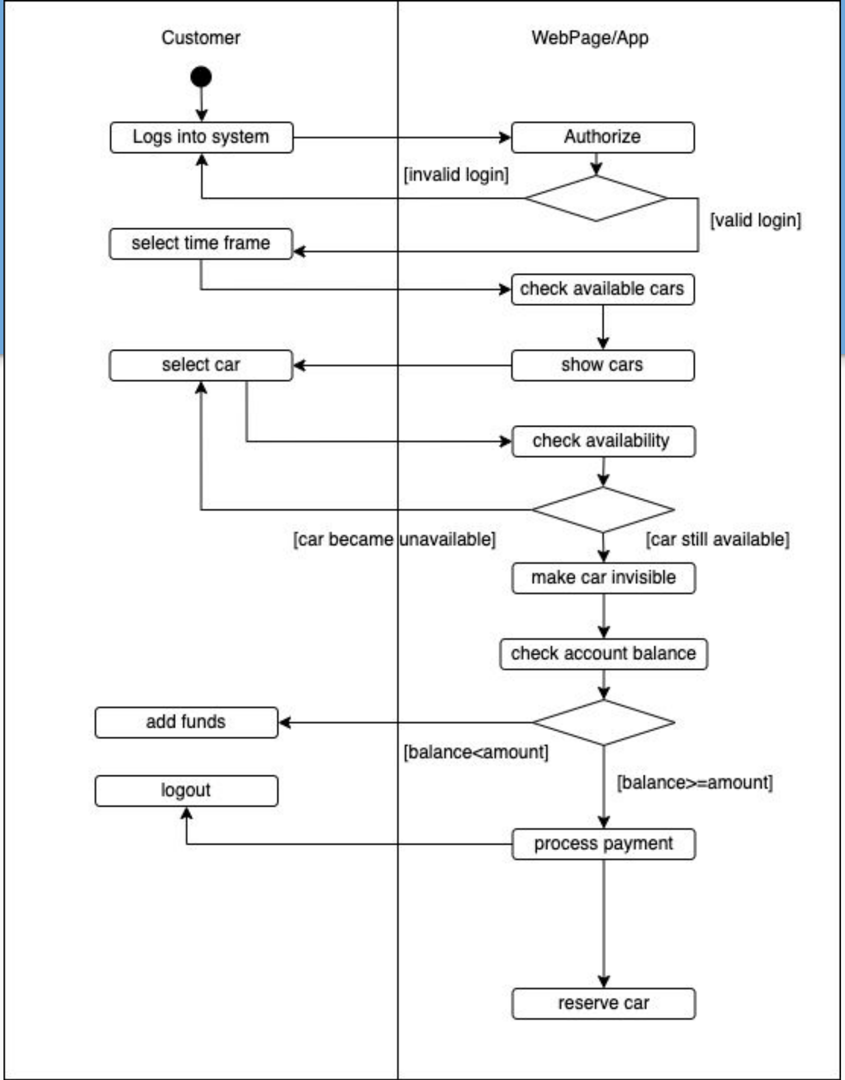
Amount
0

PAY EMPLOYEE $0.00

Requirement 3

- 1.1 reservation system for vehicles
- Functionality - User reserves a car and becomes unavailable once selected then will be reserved. After reserved money leaves user account
- Usability - easy to use interface with a calendar selector for the user to choose their dates.
- Reliability - The interface works. A user should not be able to reserve a car that is already reserved or in process of reservation.
- Performance - Fast, doesn’t take long to reserve a car.
- Supportability - Tested both functionally and with unit tests on the backend.

Diagrams


Customer
WebPage/App
Logs into system
Authorize
[invalid login]
[valid login]
select time frame
check available cars
select car
show cars
check availability
[car became unavailable]
[car still available]
make car invisible
check account balance
add funds
[balance<amount]
[balance>=amount]
logout
process payment
reserve car

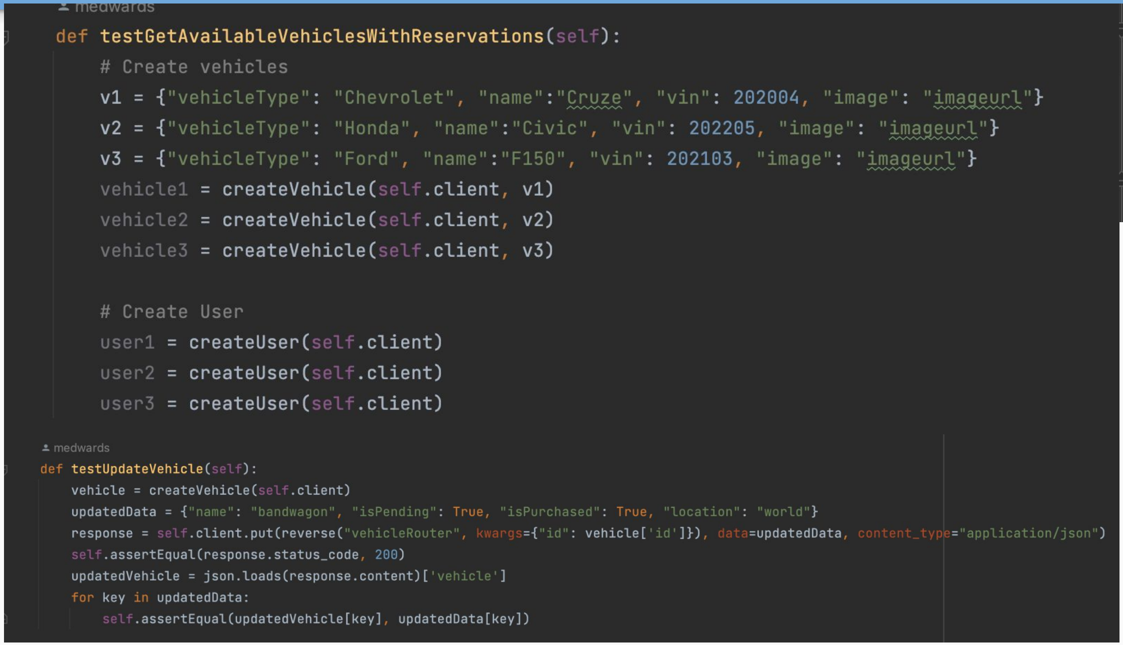
Scrum Tasks Requirement 3

- 14: check on time reservation and availability
- 25: calculate fee based on return date
- 29: component for checking out a car
- 30: component for viewing a calendar
- 40: rent a car interface

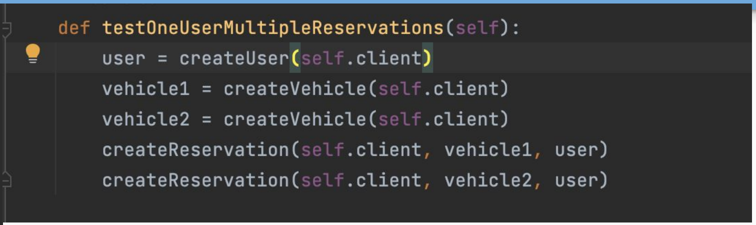
Tests Requirement 3

- Both unit testing and functional testing was done.
- Unit tests were written on the django server to test the api functionality.
- The unit tests check for getting available vehicles, user with multiple reservations, creating reservations with good and bad data, deleting reservations.

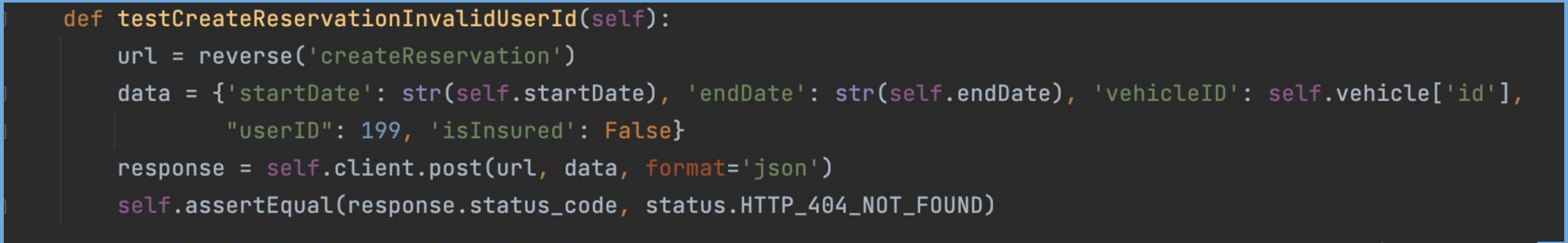
Tests Requirement 3



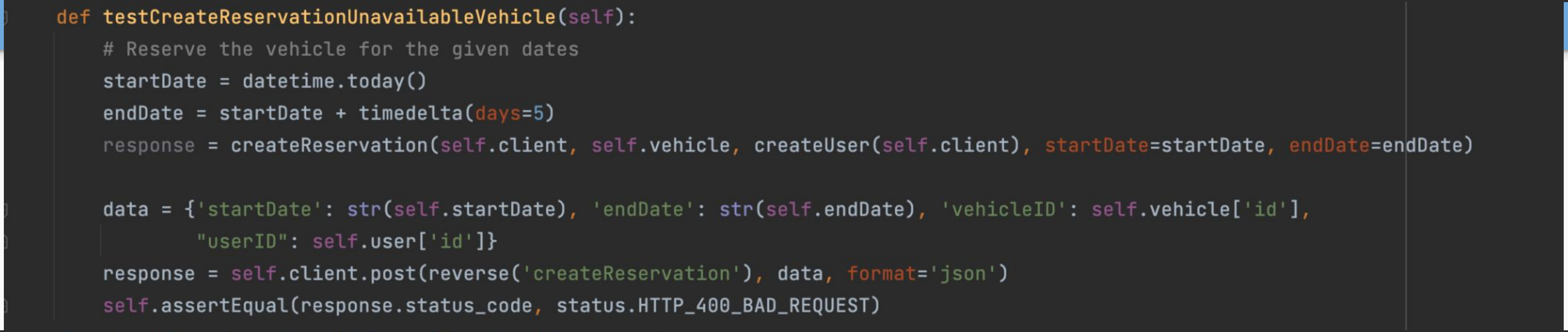
```
medwards
def testGetAvailableVehiclesWithReservations(self):
    # Create vehicles
    v1 = {"vehicleType": "Chevrolet", "name":"Cruze", "vin": 202004, "image": "imageurl"}
    v2 = {"vehicleType": "Honda", "name":"Civic", "vin": 202205, "image": "imageurl"}
    v3 = {"vehicleType": "Ford", "name":"F150", "vin": 202103, "image": "imageurl"}
    vehicle1 = createVehicle(self.client, v1)
    vehicle2 = createVehicle(self.client, v2)
    vehicle3 = createVehicle(self.client, v3)

    # Create User
    user1 = createUser(self.client)
    user2 = createUser(self.client)
    user3 = createUser(self.client)

medwards
def testUpdateVehicle(self):
    vehicle = createVehicle(self.client)
    updatedData = {"name": "bandwagon", "isPending": True, "isPurchased": True, "location": "world"}
    response = self.client.put(reverse("vehicleRouter", kwargs={"id": vehicle['id']}), data=updatedData, content_type="application/json")
    self.assertEqual(response.status_code, 200)
    updatedVehicle = json.loads(response.content)['vehicle']
    for key in updatedData:
        self.assertEqual(updatedVehicle[key], updatedData[key])
```



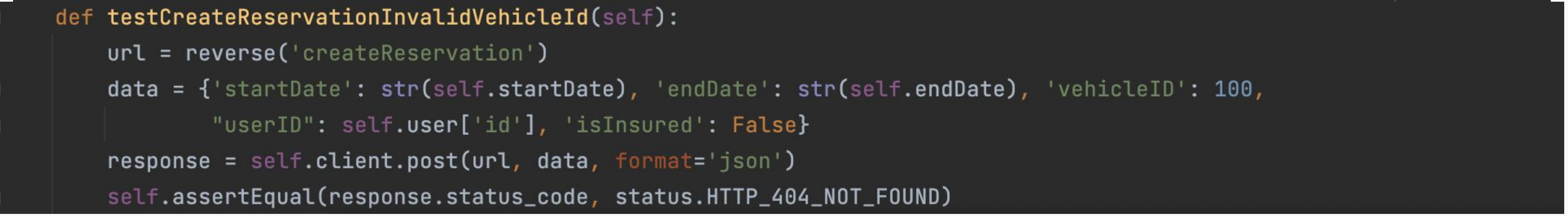
```
def testOneUserMultipleReservations(self):
    user = createUser(self.client)
    vehicle1 = createVehicle(self.client)
    vehicle2 = createVehicle(self.client)
    createReservation(self.client, vehicle1, user)
    createReservation(self.client, vehicle2, user)
```



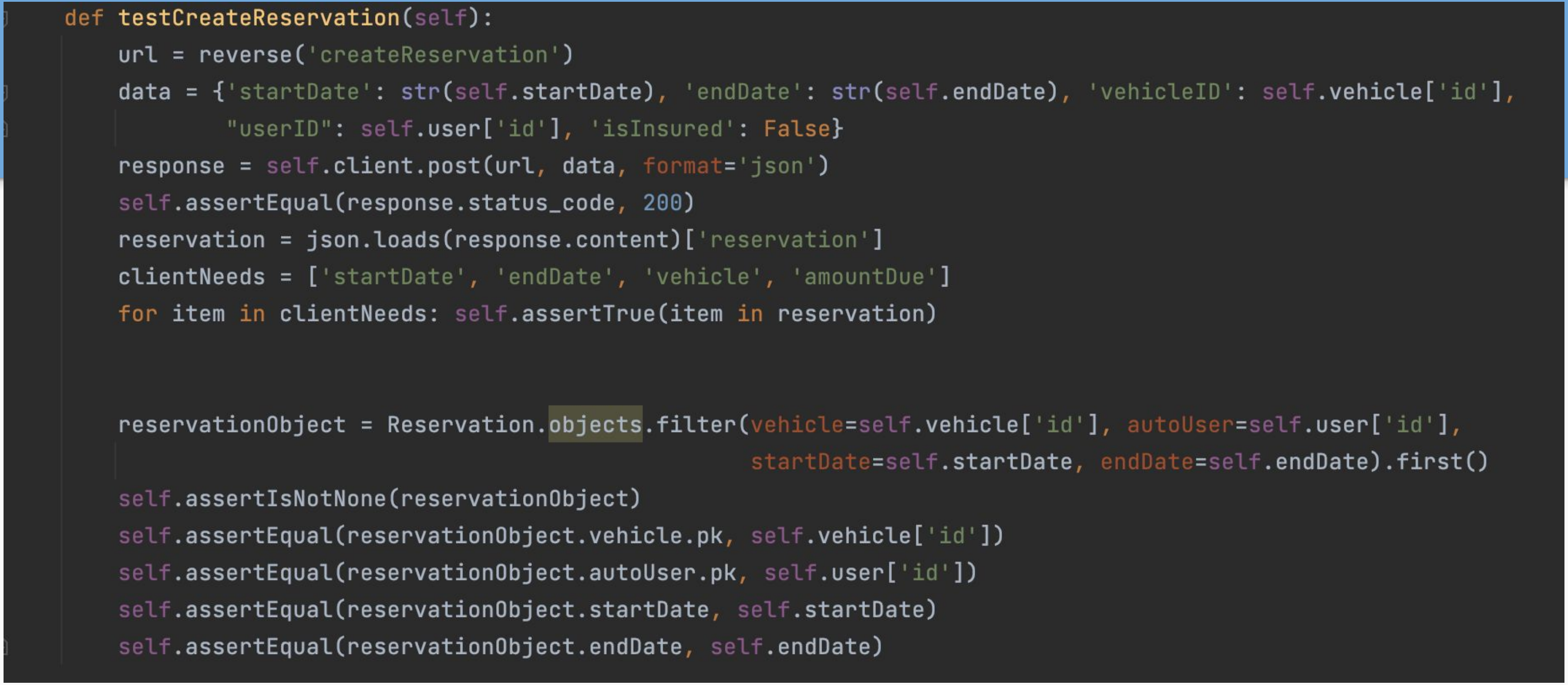
```python
    def testCreateReservationInvalidUserId(self):
        url = reverse('createReservation')
        data = {'startDate': str(self.startDate), 'endDate': str(self.endDate), 'vehicleID': self.vehicle['id'],
                "userID": 199, 'isInsured': False}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```



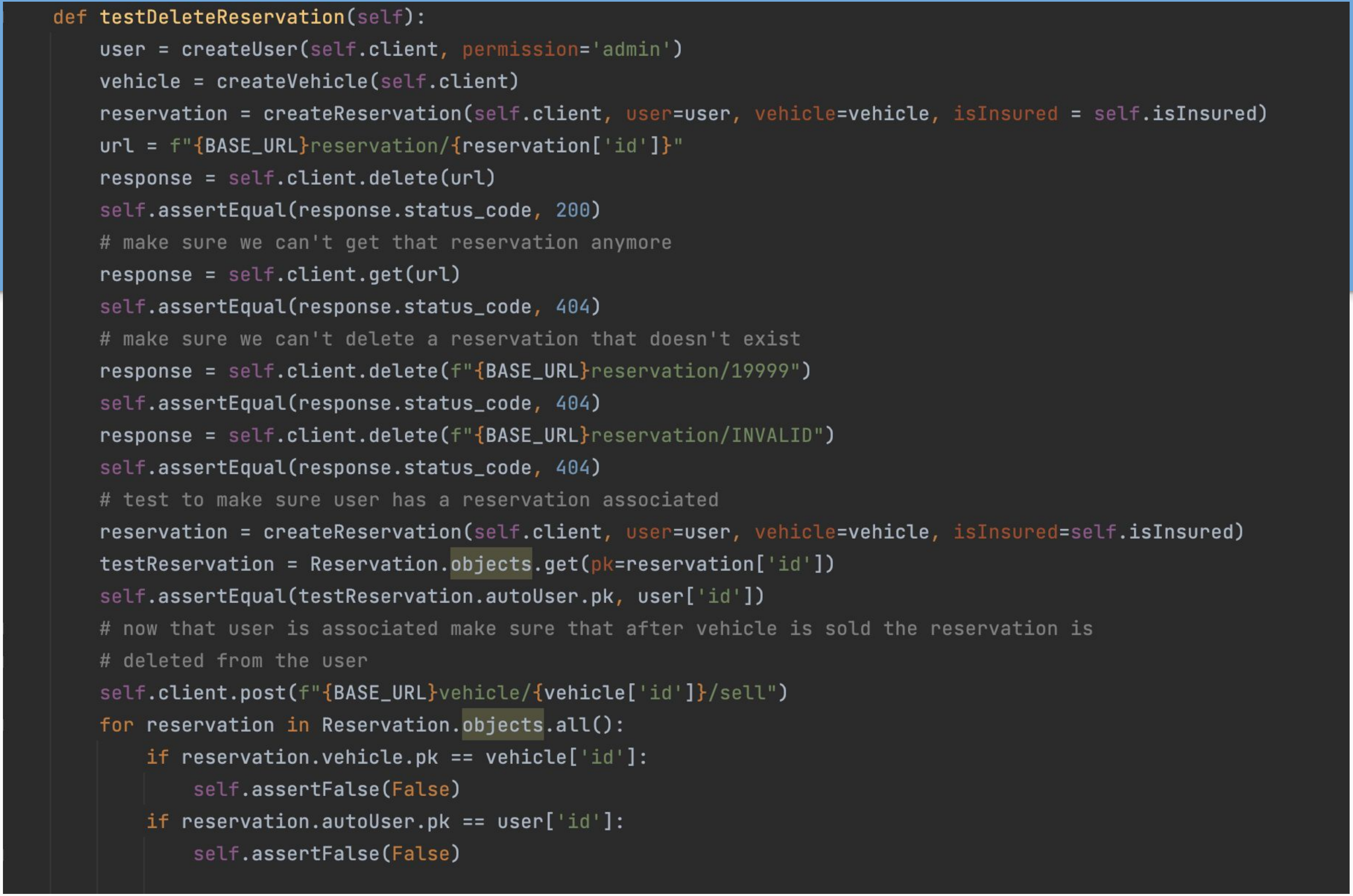
```python
    def testCreateReservationUnavailableVehicle(self):
        # Reserve the vehicle for the given dates
        startDate = datetime.today()
        endDate = startDate + timedelta(days=5)
        response = createReservation(self.client, self.vehicle, createUser(self.client), startDate=startDate, endDate=endDate)

        data = {'startDate': str(self.startDate), 'endDate': str(self.endDate), 'vehicleID': self.vehicle['id'],
                "userID": self.user['id']}
        response = self.client.post(reverse('createReservation'), data, format='json')
        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```



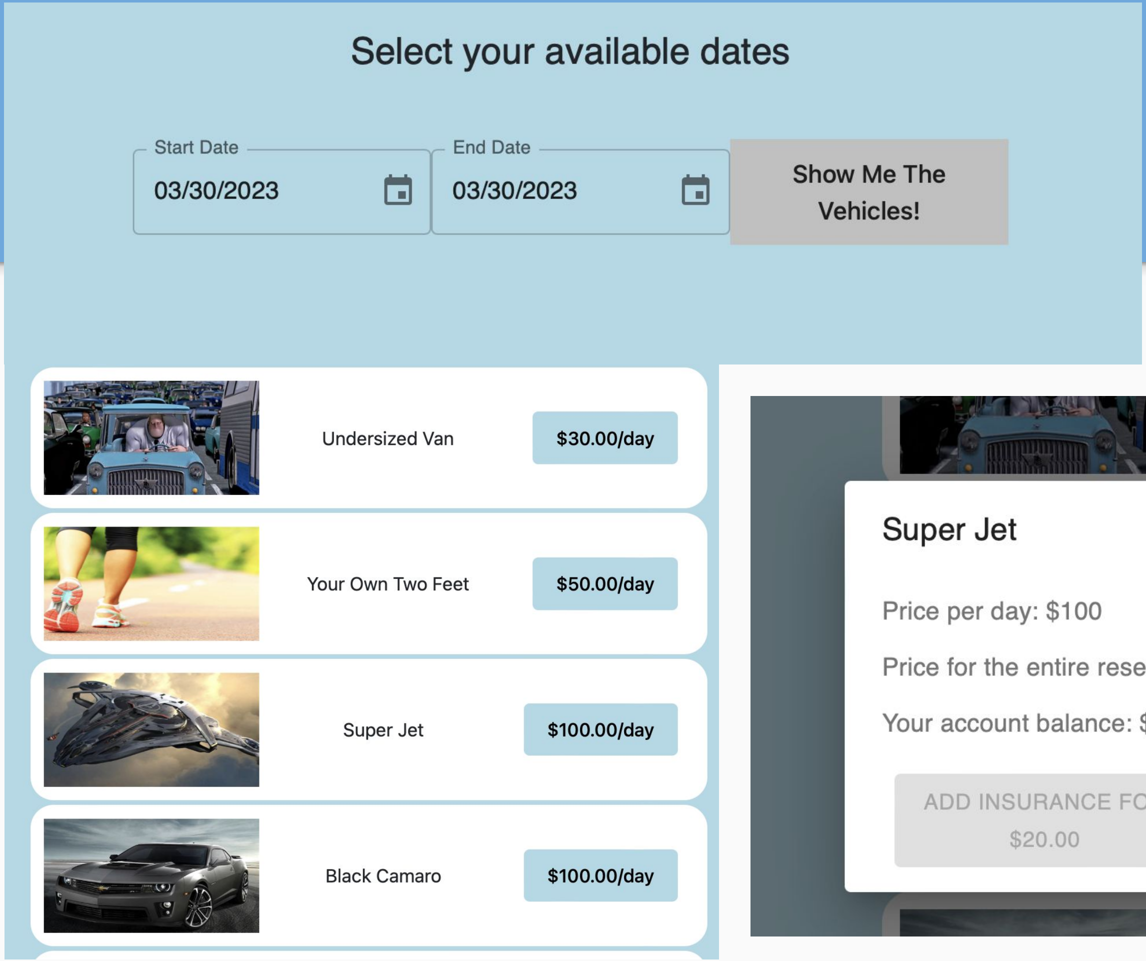
```python
    def testCreateReservationInvalidVehicleId(self):
        url = reverse('createReservation')
        data = {'startDate': str(self.startDate), 'endDate': str(self.endDate), 'vehicleID': 100,
                "userID": self.user['id'], 'isInsured': False}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```



```python
def testCreateReservation(self):
    url = reverse('createReservation')
    data = {'startDate': str(self.startDate), 'endDate': str(self.endDate), 'vehicleID': self.vehicle['id'],
            "userID": self.user['id'], 'isInsured': False}
    response = self.client.post(url, data, format='json')
    self.assertEqual(response.status_code, 200)
    reservation = json.loads(response.content)['reservation']
    clientNeeds = ['startDate', 'endDate', 'vehicle', 'amountDue']
    for item in clientNeeds: self.assertTrue(item in reservation)


    reservationObject = Reservation.objects.filter(vehicle=self.vehicle['id'], autoUser=self.user['id'],
                                                   startDate=self.startDate, endDate=self.endDate).first()
    self.assertIsNotNone(reservationObject)
    self.assertEqual(reservationObject.vehicle.pk, self.vehicle['id'])
    self.assertEqual(reservationObject.autoUser.pk, self.user['id'])
    self.assertEqual(reservationObject.startDate, self.startDate)
    self.assertEqual(reservationObject.endDate, self.endDate)
```



```python
    def testDeleteReservation(self):
        user = createUser(self.client, permission='admin')
        vehicle = createVehicle(self.client)
        reservation = createReservation(self.client, user=user, vehicle=vehicle, isInsured = self.isInsured)
        url = f"{BASE_URL}reservation/{reservation['id']}"
        response = self.client.delete(url)
        self.assertEqual(response.status_code, 200)
        # make sure we can't get that reservation anymore
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)
        # make sure we can't delete a reservation that doesn't exist
        response = self.client.delete(f"{BASE_URL}reservation/19999")
        self.assertEqual(response.status_code, 404)
        response = self.client.delete(f"{BASE_URL}reservation/INVALID")
        self.assertEqual(response.status_code, 404)
        # test to make sure user has a reservation associated
        reservation = createReservation(self.client, user=user, vehicle=vehicle, isInsured=self.isInsured)
        testReservation = Reservation.objects.get(pk=reservation['id'])
        self.assertEqual(testReservation.autoUser.pk, user['id'])
        # now that user is associated make sure that after vehicle is sold the reservation is
        # deleted from the user
        self.client.post(f"{BASE_URL}vehicle/{vehicle['id']}/sell")
        for reservation in Reservation.objects.all():
            if reservation.vehicle.pk == vehicle['id']:
                self.assertFalse(False)
            if reservation.autoUser.pk == user['id']:
                self.assertFalse(False)
```



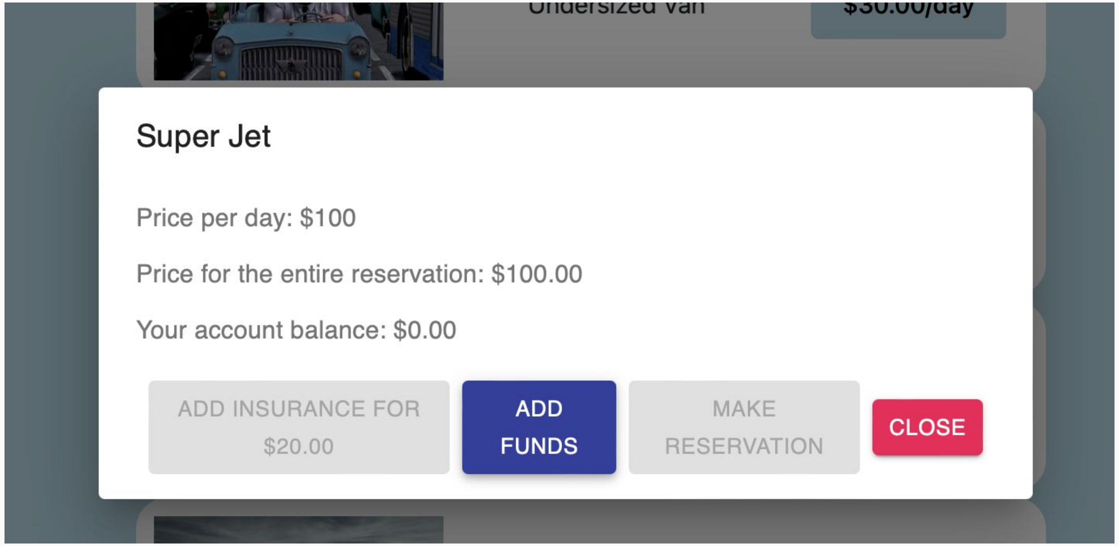
Select your available dates

Start Date
03/30/2023

End Date
03/30/2023

Show Me The Vehicles!

Undersized Van	$30.00/day
Your Own Two Feet	$50.00/day
Super Jet	$100.00/day
Black Camaro	$100.00/day



Super Jet

Price per day: $100

Price for the entire reservation: $100.00

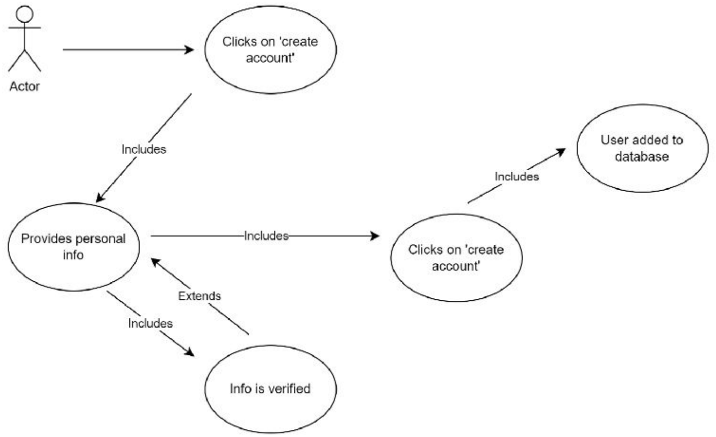
Your account balance: $0.00

Screen Capture 1

Use Case: Customer creates an account

Title: Customer creates an account

Participating Actor: Customer



Entry conditions:

- Customer has a valid email address not tied to a current account
- Customer clicks 'create account' on website homepage

Exit conditions:

- Customer successfully creates a new account

Event flow:

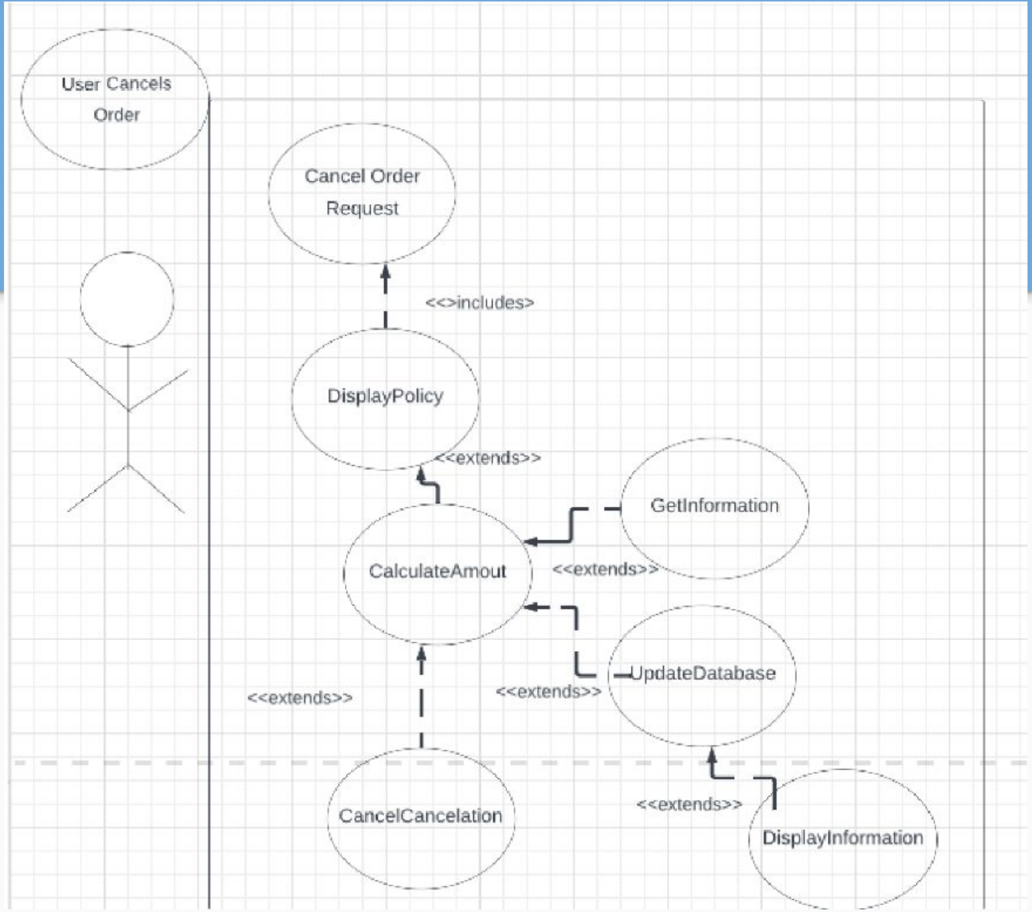
1. Customer provides an email address not tied to a current account, username, and password
2. Username must be unique and password must meet security criteria (length, characters, etc)
3. Customer selects 'create account' button on new account screen
4. System adds user to database
5. User now has access to website functionality

Screen Capture 1

User Signs Up

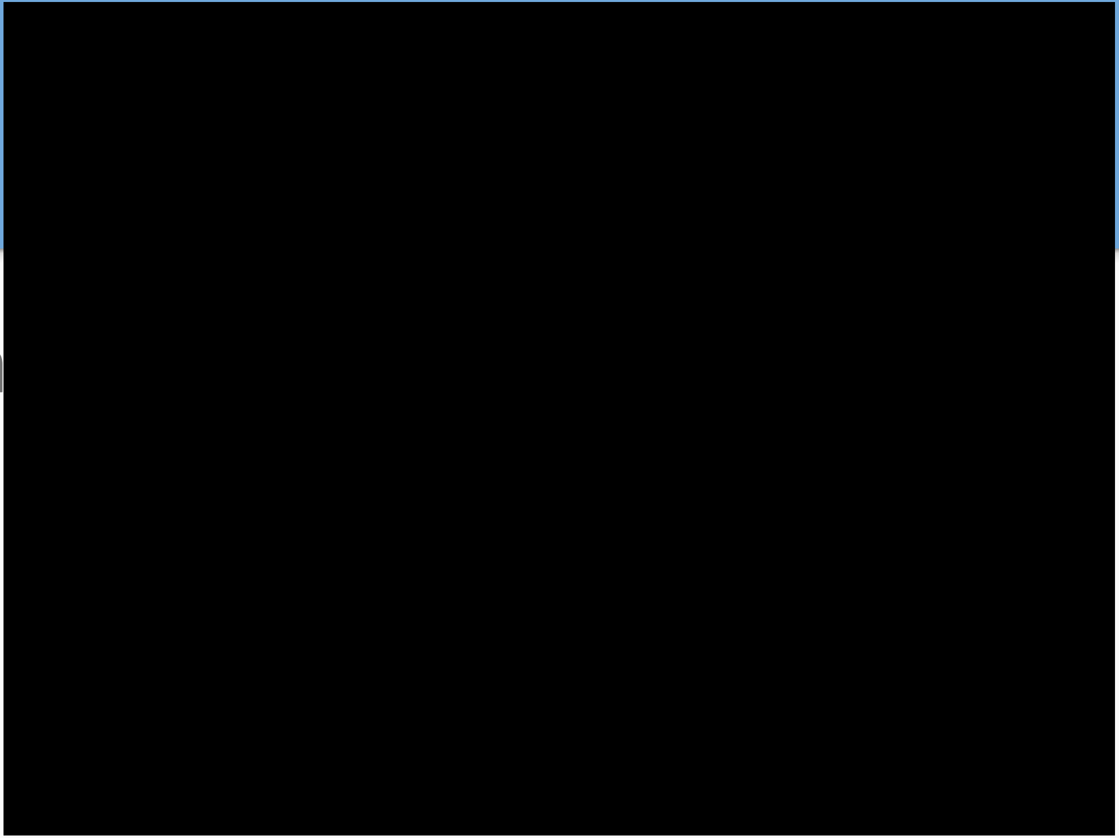
Screen Capture 2

Use Case: Customer cancels a reservation



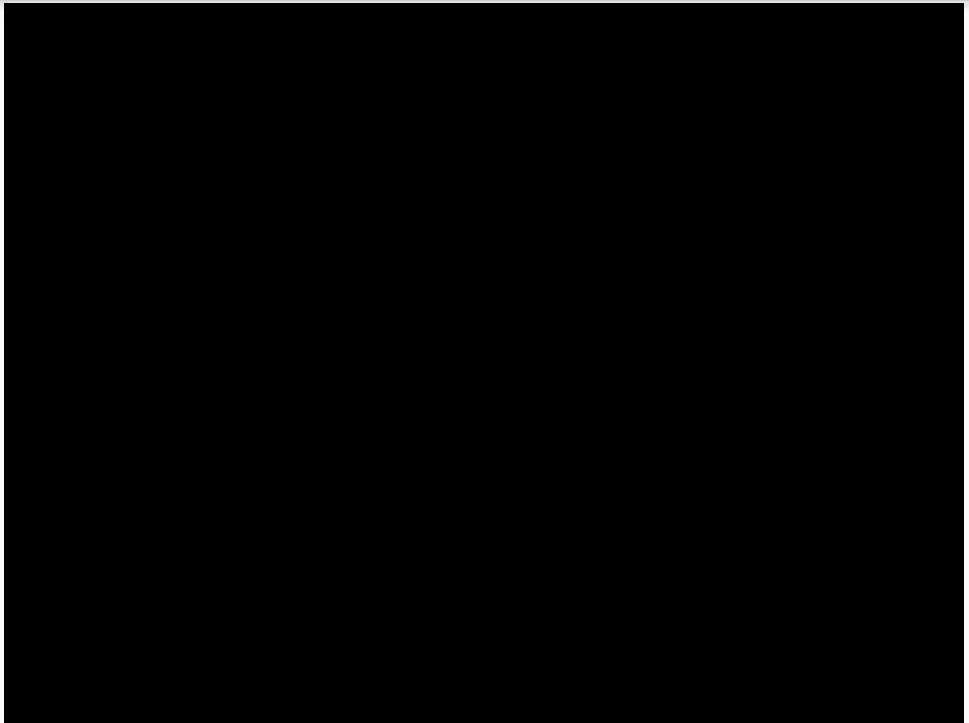
Screen Capture 2

Cancel Reservation



Screen Capture 3

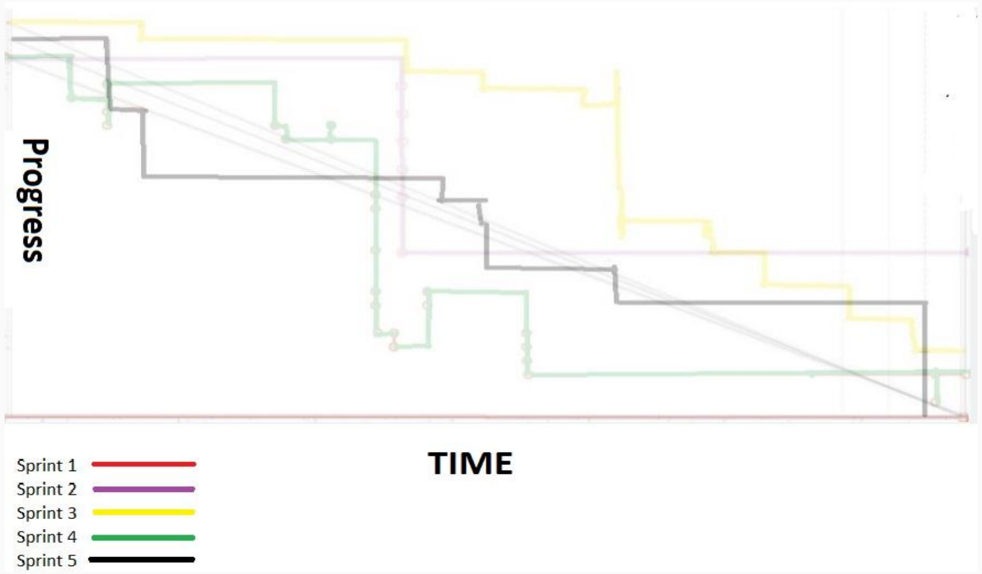
Use Case: Creating User



Screen Capture 4

Use Case: Logging into system

Burndown Chart



Resources

Django tutorial: https://docs.djangoproject.com/en/4.1/intro/tutorial01/

React tutorial: https://www.codecademy.com/catalog

Diagram Software: https://www.lucidchart.com/pages/

Documentation: https://legacy.reactjs.org/docs/getting-started.html

https://docs.djangoproject.com/en/4.1/

Questions