Dans Getaway Car Shack

Group 4
Ryan Gubler (Himself), Michael Hanks (Database), Josh Hatch (Database), Kaiden McMillen (Django Wizard)

Project Overview

Major Design Decisions - - - - - - - - - - ->

Simple UI

Flexible Database

Django

Bootstrap

Scrum Meetings - - - - - - - - - - - - - - - - ->

Tuesdays and Thursdays in person

Mondays and Fridays on discord

Ready for Deployment - - - - - - - - - - ->

High security

Thorough testing

Requirement 1

User profiles and account security

Functionality – Req 1

From requirements document

1. User profiles and account security

1.1. The app will have unique passwords and usernames for each user. These will be used when logging in to the app.

1.2. The app will have three distinct use types, customer, employee, and manager.

1.2.1. Customers will have permission to rent cars, buy insurance, add funds, and request service calls.

1.2.2. Employees will have permission to report hours, LoJack cars, and accept service requests.

1.2.3. Managers will have permission to pay outstanding wages, hire new employees, and buy new cars.

1.3. Managers will be able to hire new employees by elevating the account status of a customer to the employee level.

Usability - Req 1

- Users, employees, and Managers have secure accounts.
- Managers have the ability to hire users to employees or managers.
- The site allows only managers and employees to log hours and complete service tickets.
- Users are allowed to reserve cars and return them, respectively.

Reliability- Req 1

We wanted accounts to be secure and non accessible from intruders.

- Django applies user security and functionality

We also wanted accounts to have balances, as well as employees and managers to have hours logged.

- Django has models available for customization.
- We used custom models that allowed a custom user to be linked to a django user, which allowed functionality.

We added decorators on each view function that makes sure someone is logged in to be able to access pages around the site.

Performance- Req 1

The code we have set up queries the database in a timely fashion when asking for a user, custom user, etc.

There have been no errors with the database being infiltrated.

Testing with Customers, Employees, and Managers has led to no problem with getting information.

```
@login_required(login_url='product:loginTest')
def account(request):
    now = date.today()
    customUser = CustomUser.objects.get(user = request.user)
    list = []
    for reservation in customUser.carreservation_set.all():
        if reservation.endDate >= now:
            list.append(reservation)
    return render(request, 'product/account.html', {'customUser': customUser,'reservations':list})
```

Sustainability- Req 1

- Because of the models we created in Django, we can add many different things to what we already have implemented.
- We could easily add one-time insurance policies, profile pics, etc. (Found in the Future Features section of requirements documentation).
- Debugging is easy because of the simplicity of the database itself and the methods we created.

```
class CustomUser(models.Model):
    user = models.OneToOneField(User,on_delete=models.CASCADE)
    balance = models.FloatField(default = 0.0)
    hours = models.FloatField(default = 0.0)

    def addFunds(self,amount):
        self.balance += amount

    def addHours(self,amount):
        self.hours += amount

    def __str__(self):
        return self.user.username
```

Users and account security scrum tasks

Sprint 2:

- Create User Login - Josh, Kaiden
- Create Custom User linked with User - Michael, Ryan

Sprint 3:

- Only manager can access add car page - Michael, Ryan
- Only manager can access pay employee page - Michael, Kaiden
- Only employee and manager can access service tickets. - Josh, Kaiden

Sprint 4:

- Customers can access add funds page and reservation page. - Josh, Ryan
- Create Unit testing for users, employees, and managers. - Michael

Unit Testing - Requirement 1

- Tested users and their authentication
- Tested users balance and hours logged
- Example of testing is in right code snippet
- Code Snippet only tests database directly, including models and methods

```
class UserTestCase(TestCase):
    def setUp(self):
        User.objects.create(email = "michael@gmail.com", first_name = "Michael", last_name = "Hanks", password = "1234", username = "michael")
        User.objects.create(email = "test@gmail.com", first_name = "Test", last_name = "Test", password = "test", username = "test")

    def testCustomUser(self):
        user1 = User.objects.get(email = "michael@gmail.com")
        user2 = CustomUser.objects.get(user = user1)
        self.assertEqual(0, user2.balance)
        user2.addFunds(-5.0)
        user2.save()
        self.assertNotEqual(0,user2.balance)
        self.assertGreater(0,user2.balance)


    def testPassword(self):
        user1 = User.objects.get(email = "michael@gmail.com")
        user2 = User.objects.get(email = "test@gmail.com")
        self.assertNotEqual(user1.password, user2.password)
        self.assertEqual("1234", user1.password)
        self.assertEqual("test", user2.password)
```

Some more code snippets – Req 1

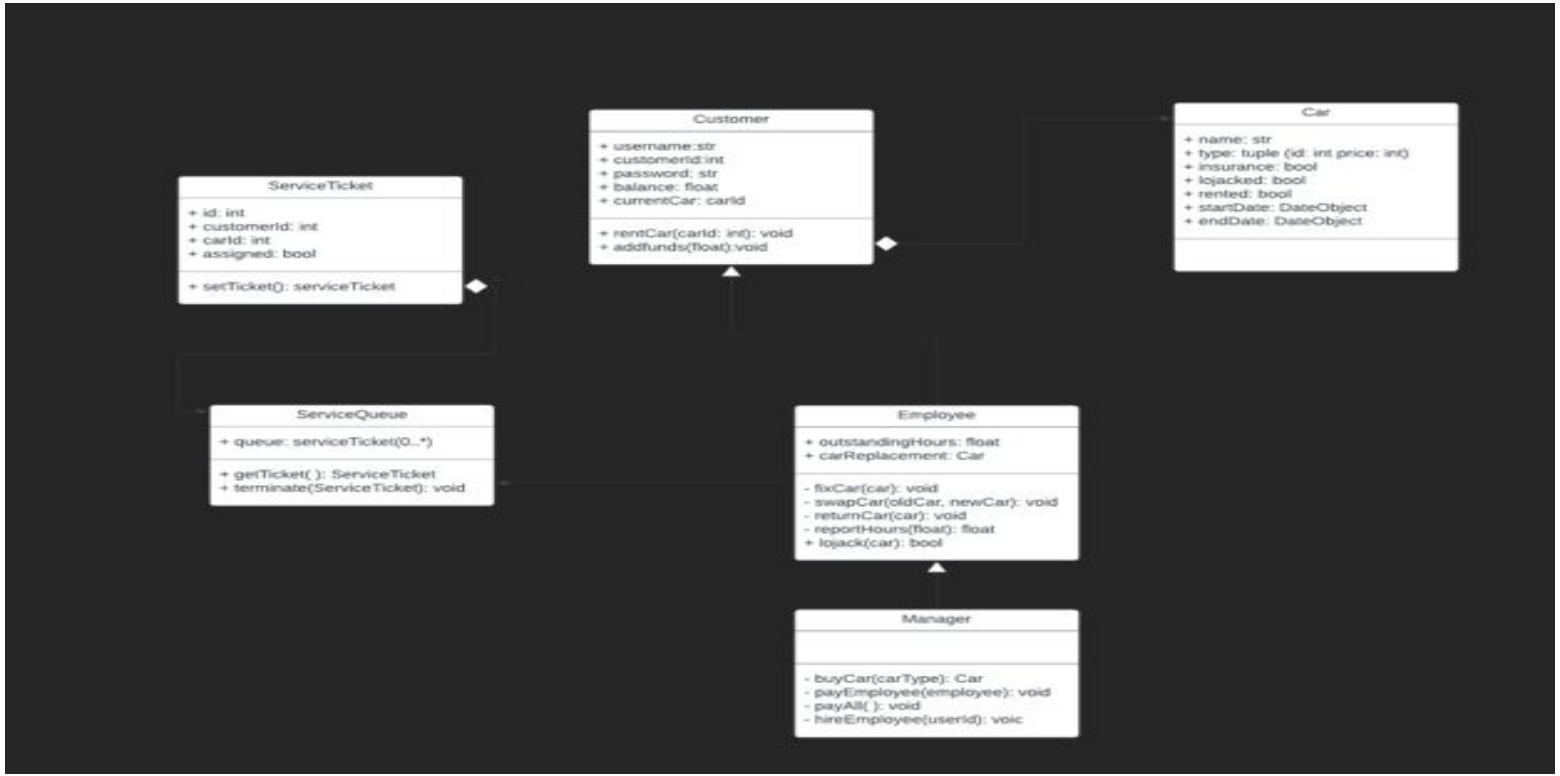
Code below: Shows that when trying to pay employees, it check if the current user is a manager or not. If not, it will redirect back to the account page, showing that the site has good security.

```
@login_required(login_url='product:loginTest')
def payEmployeePage(request):
    if request.user.has_perm('auth.Manager'):
        employees = []
        for employee in CustomUser.objects.all():
            if employee.user.has_perm('auth.Employee'):
                if employee.user.has_perm('is super user'):
                    continue
                employees.append(employee)
        return render(request, 'product/payEmployeePage.html', {'employees':employees})
    return redirect(reverse('product:account'))
```

Code below: When trying to lojack a car, we make sure that the request is a post request, and the user has manager permissions, to make sure the database cannot be infiltrated.

```
@login_required(login_url='product:loginTest')
def lojackCar(request):
    if request.method == "POST" and request.user.has_perm('auth.Manager'):
        reservation_id = request.POST['options']
        reservation = CarReservation.objects.get(pk = reservation_id)
        customUser = reservation.user
        reservation.lojacked = True
        ticket = ServiceTicket(customerId = customUser.user.id, carId = reservation.car.id)
        ticket.save()
        reservation.save()
        customUser.save()
        return redirect(reverse('product:overdueReservations'))
    return redirect(reverse('product:account'))
```

UML Class Diagram

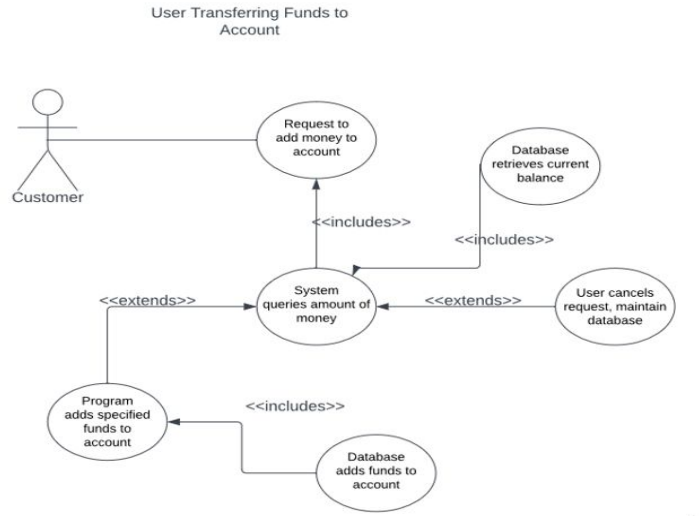


Requirement 2

Managing funds in wallet

Video Demonstration

Case 3 User Adds Money to Wallet



Participating Actor:

- Customer

Entry Conditions:

- Customer wishes to add funds to account

Exit Conditions:

- Customer has added funds to the account
- Customer decides not to add funds

Event Flow:

- Customer logs into app
- Customer navigates to account/funds page
- Customer inputs value to transfer, and submits request
- Database updates with new fund total
- Confirmation of payment is given to customer

Functionality - Req2

From requirements document

1. Funds wallet

    1.1. Each account will have a wallet that holds funds associated with it (adding funds)

        1.1.1. Customers may opt to add money into their respective wallets.

        1.1.2. Employees will have money added to their accounts when the manager pays their reported hours.

        1.1.3 The manager has access to the funds derived from rentals, insurance, and late fees.

Usability - Req 2

- Users, employees, and Managers can add funds to their account.
- Managers have the ability to purchase cars and pay employees.
- Employees can collect money from the hours they log.
- Users can rent vehicles by adding funds to their account. They also can purchase insurance, and also pay the service fee if their car gets lojacked.

Reliability - Req 2

We wanted adding funds to be simple and user friendly.

- The custom input bar only accepts an integer to help account for bad input that a user might give.
- We have funds in the model so that the funds for the users, employees, and managers will keep track of the money in the accounts accurately.
- The tests we have help ensure the accuracy of funds moving around in accounts.

Performance - Req 2

The code we have set up queries the database in a timely fashion when asking for current funds, and also when updating funds.

We are clean from any known errors. The math behind the funds addition and subtraction has been working correctly.

Testing the user, employee, and manager accounts has shown that the movement and flow of funds has been accurately portrayed.

```python
@login_required(login_url='product:loginTest')
def payEmployeePage(request):
    if request.user.has_perm('auth.Manager'):
        employees = []
        for employee in CustomUser.objects.all():
            if employee.user.has_perm('auth.Employee'):
                if employee.user.has_perm('is super user'):
                    continue
                employees.append(employee)
        return render(request, 'product/payEmployeePage.html', {'employees':employees})
    return redirect(reverse('product:account'))

@login_required(login_url='product:loginTest')
def payAll(request):
    if request.method == "POST" and request.user.has_perm('auth.Manager'):
        manager = CustomUser.objects.get(user = request.user)
        for user in CustomUser.objects.all():
            user1 = user.user
            if user1.has_perm('auth.Employee'):
                hours = user.hours
                user.addFunds(float(hours*15.0))
                user.addHours(float(hours*-1))
                manager.addFunds(float(hours*-15.0))
                user.save()
                manager.save()
        return redirect(reverse('product:account'))
    return redirect(reverse('product:account'))
```

```python
@login_required(login_url='product:loginTest')
def addFunds(request):
    customUser = CustomUser.objects.get(user = request.user)
    if request.method == 'POST':
        if "10" in request.POST:
            customUser.addFunds(10.0)
            customUser.save()
            return redirect(reverse("product:addFunds"))
        elif "25" in request.POST:
            customUser.addFunds(25.0)
            customUser.save()
            return redirect(reverse("product:addFunds"))
        elif "50" in request.POST:
            customUser.addFunds(50.0)
            customUser.save()
            return redirect(reverse("product:addFunds"))
        elif "100" in request.POST:
            customUser.addFunds(100.0)
            customUser.save()
            return redirect(reverse("product:addFunds"))
        elif "custom" in request.POST:
            customUser.addFunds(float(request.POST["custom"]))
            customUser.save()
            return redirect(reverse("product:addFunds"))

    return render(request, 'product/addFunds.html', {'customUser': customUser})
```

Sustainability - Req 2

- Because of the models created in Django, we can add functionality of various things.
- We could add transferring funds to bank accounts. (Found in the Future Features section of requirements documentation)
- Debugging is easy since simple math operations are used when purchasing and earning.

```
class CarTestCase(TestCase):
    def setUp(self):
        Car.objects.create(name="Mustang",price=100.0)
        Car.objects.create(name="Rio",price=50.0)
        rio = Car.objects.get(name = "Rio")


    def test_car_price(self):
        mustang = Car.objects.get(name = "Mustang")
        rio = Car.objects.get(name="Rio")
        self.assertEqual(mustang.price,100.0)
        self.assertEqual(rio.price,50.0)
        self.assertNotEqual(rio.price,100.0)
```

Requirement 3

Car Inventory/Reservations

Functionality

Functionality, from Functional Requirements doc.

- 2.1 The app will have a database of cars currently under ownership of Dans Getaway
  - 2.1.1 The cars will either be reserved, or free to reserve. The period of reservation will be at minimum one day.
  - 2.1.2 The app database will keep track of whether a car is currently insured, and whether it is currently LoJacked.
- 2.2 Cars will come in one of three types of vehicles, these will cost $100, $50, or $10 per day of rental.
- 2.3 The manager has the ability to purchase new cars to be added to the inventory.

Usability

- We expected that all four tiers of user access would need to interface with our car inventory.
  - Users would need to find available cars, and sort them by availability and price range.
  - Employees would need to be able to find and complete service tickets that had been attached to certain cars.
  - Managers would need to be able to view the car inventory, as well as add new cars to the inventory
- We wanted the UI to be intuitive, as well as consistent throughout the website. We accomplished this by relying mostly on simple html forms. This was helpful in the development process because it was easy to implement, and almost all users would come to the app with experience using identical UI elements.

Reliability

- Due to the relative simplicity of of our application we had a goal of an essentially bug free database of cars

- By using the built in Django database SQlite, we had to do very little to ensure that our get and post requests had the effects that we expected them to, this allowed for our car inventory to be developed relatively bug free.

- While bug testing was an important part of our development process, we did find that due to time constraints we did not have the opportunity to test the site as much as we wanted.
    - Ideally we would have a trial run using professional testers.

Performance

- For this project we used the Django framework. Because Django is written in python, it will always have performance limitations when compared to other frameworks, likely if we were going to produce this app commercially we would be using a faster technology.

- None of our code related to the car inventory relies on complicated algorithms, allowing us to be fairly certain that none of our functions causes unexpected slowdown during runtime.

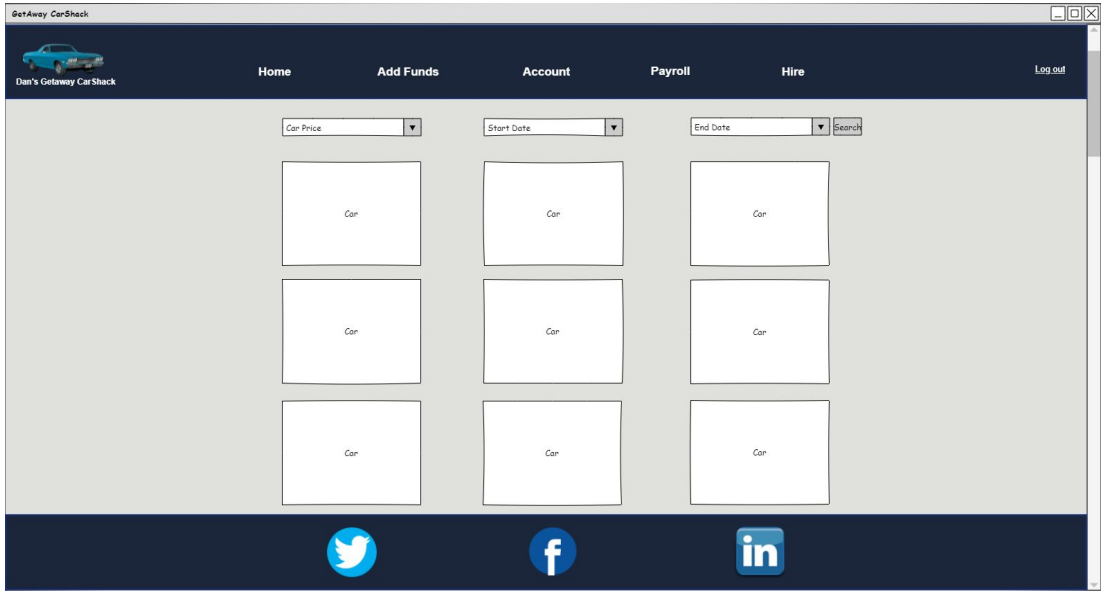
- Because the nature of this car rental app is designed for management of a specific lot and not a large corporation, we can be certain that the number of cars actually in the database will be relatively small. This ensures that the file size of our program will be manageable for almost any computer running it.

Supportability

- Reservations are represented as model instances in our database.
  - This allows for easy addition of new data fields that can be used to create new functionality.
    - For example we could attach the parking lot a return is supposed to take place.
- The above reasoning also applies to our car models, for example we could create a field that holds date regarding car maintenance, oil changes, how old the tires are, and the like.
- Because of the modularity of the code expanding upon existing features of the site should be relatively easy.
- One possible concern moving forward is that the views file in our project is already quite cluttered, we could possibly remedy this by splitting this file up into multiple directories, or possibly utilize multiple apps inside the same project.

Low Fidelity Modeling

You'll note that the original modeling for the reservation page looks quite different than our final product. Some of this is due to the inclusion of the picaday calendar in our app.



Unit Testing

- Test for pricing
- Test for reservations
- Test for proper naming

```python
class CarTestCase(TestCase):
    def setUp(self):
        Car.objects.create(name="Mustang",price=100.0)
        Car.objects.create(name="Rio",price=50.0)
        rio = Car.objects.get(name = "Rio")


    def test_car_price(self):
        mustang = Car.objects.get(name = "Mustang")
        rio = Car.objects.get(name="Rio")
        self.assertEqual(mustang.price,100.0)
        self.assertEqual(rio.price,50.0)
        self.assertNotEqual(rio.price,100.0)

    def test_car_reservation(self):
        mustang = Car.objects.get(name = "Mustang")
        rio = Car.objects.get(name="Rio")
        self.assertEqual(0,len(mustang.carreservation_set.all()))
        self.assertEqual(0,len(rio.carreservation_set.all()))
        self.assertNotEqual(1,len(rio.carreservation_set.all()))


    def test_car_name(self):
        mustang = Car.objects.get(name = "Mustang")
        rio = Car.objects.get(name="Rio")
        self.assertEqual("Mustang",mustang.name)
        self.assertEqual("Rio",rio.name)
        self.assertNotEqual("Rio",mustang.name)
```

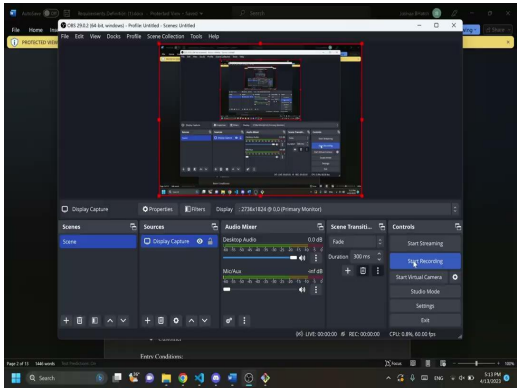
Work Itemization

- Create user view for returning car rentals
  - Michael Hanks
- Create backend view for inventory page
  - Michael Hanks
- Create javascript functionality for retrieving available cars from database.
  - Michael Hanks
- Create ticket method
  - Joshua Hatch
- Create Django function to return a JSON object with available cars, given the start and end date
  - Michael Hanks
- Create picaday calendar
  - Ryan Gubler
- Create unit test for car models
  - Michael Hanks
- Create Django view function for logging out user/employee/manager
  - Michael Hanks
- Create service ticket model
  - Joshua Hatch
- Create car model
  - Michael Hanks
- Create high fidelity modeling
  - Joshua Hatch
- Create manager view for lojacking overdue rentals
  - Michael Hanks
- Create high fidelity prototype for relevant pages
  - Ryan Gubler

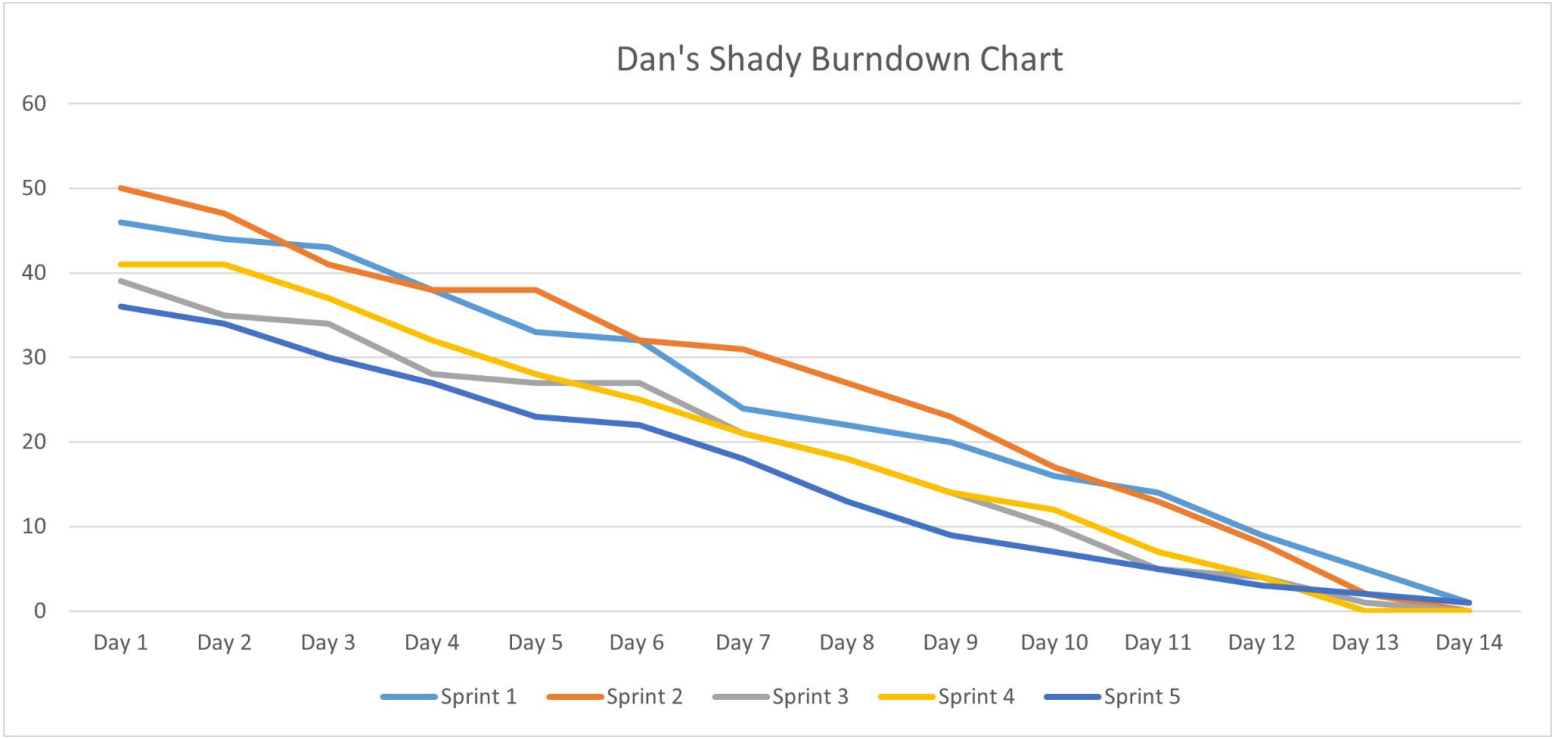
Videos...

Video 1 - Use Case 5 - Manager Purchasing Car

Video 2 - Use Case 1 - User Rents Car



Five Color Burndown Chart



Resources

- https://docs.djangoproject.com/en/4.1/topics/auth/default/ - Django documentation for users and authentication
- https://stackoverflow.com - General purpose advice/help regarding almost every aspect of the project.
- https://getbootstrap.com/ - CSS framework used for an easier to implement and more professional looking front end.
- https://github.com/ - Powerful version control, online repository, and team management tool.
- https://discord.com/ - Used for team communication, and roughly half of team stand up meetings.

Questions?