

REVOLUTIONIZING FOOD DELIVERY SERVICES WITH HOME COOKED MEALS

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia


In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Guillermo Saavedra-Diaz
Spring, 2020

Technical Project Team Members

Steven Morrison
Habib Karaky
Isabel Kershner
Guillermo Saavedra
Shivani Saboo
Jack Short
Ankith Yennu

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signature  Date 05/01/2020
Guillermo Saavedra-Diaz

Approved  Date 5/1/2020
Dr. Ahmed Ibrahim, Department of Computer Science

Table of Content

Abstract	3
List of Figures	4
1. Introduction	6
1.1 Problem Statement	7
1.2 Contributions	7
2. Related Work	9
3. System Design	11
3.1 System Requirements	12
3.2 Wireframes	15
3.3 Sample Code	16
3.4 Sample Tests	25
3.5 Code Coverage	26
3.6 Installation Instructions	27
4. Results	33
5. Conclusions	34
6. Future Work	36

Abstract

Food delivery services take advantage of consumers' desire for convenience, allowing them to stay at home and not worry about cooking for themselves. Consumers have increasingly ordered food from delivery services and avoided preparing food from scratch, even though home-cooked meals provide a more nutritious and well-rounded diet. The goal of the Capstone project is to develop a food delivery service, called HomeEats, that is hosted as a web application specifically for delivering home-cooked meals. HomeEats was developed using the Python Django Framework and the site is hosted on an AWS server for production purposes. The capstone team followed the scrum agile methodology to develop the web application.

This application will give at-home chefs the ability to create a HomeEats cook account and post a list of home-cooked meals as a menu. Customers can make a HomeEats customer account to view and order meals from the application. This web application combines the consumer's desire for convenience while also providing passionate at-home chefs the opportunity to cook.

The HomeEats site administrator has the ability to oversee customer and cooks' activity, such as customer's orders, cook's dishes, and revenue reports. HomeEats provides safety features such as mandatory administrator approval for all menu items and customer reviews of dishes. HomeEats also requires all menu items posted by a chef to be approved by the administrator before the dish can be ordered by a customer.

The customer review feature allows customers to write a review on a previously ordered dish and give a numerical rating on a scale from 0 to 5. Reviews will help to inform other customers and site administrators about the quality of the food. When chefs post a dish, they

must list all of the ingredients and any possible allergies. Dishes are associated with cuisine types which can help customers to more easily find the chefs that they would like to order from. Customers can contact and message cooks about their order, which is especially helpful for customers who have allergies or dietary restrictions. During the checkout process, customers will also have the ability to tip the cook and the checkout page precomputes possible tipping percentages for easy tipping. The HomeEats web application has been designed to help customers obtain home-cooked meals easily and to ensure the safety of its patrons through the use of a review system, administrator monitoring, and approval systems.

List of Figures

Figure 1. Login Screen	15
Figure 2. Customer Checkout Screen	15
Figure 3. Customer Account Creation Screen	15
Figure 4. Cook Order Screen	16
Figure 5. Dish Details Screen	16

1. Introduction

HomeEats seeks to deliver meals to those that do not have time to cook but still want the quality of home-cooked food. We want to be able to offer quality food from at home cooks that differs from the typical meal eaten out. Our team developed a web application called HomeEats, which serves as a food delivery service, similar to UberEats and Grubhub, but tailor-made for home-cooked meals. HomeEats allows consumers like you to finally have the ability to access fresh, home-cooked meals without having to go buy ingredients or prepare the dish themselves.

Unlike other food delivery platforms, this convenience does not come at a sacrifice of food quality or fresh ingredients. To use the platform, consumers create an account on the site, enter their location and instantly view a large selection of dishes being cooked by amateur cooks in the area. They can view all the ingredients in the dish, the type of cuisine it originates from, estimated preparation time, and background on the chef, including reviews from previous customers. Once they select the dish they want, customers purchase the dish directly online, at which point the chef will be notified that an order has been placed and begin cooking. Home chefs will be able to specify when they are online and available to cook, how many orders they can take at a time, and in the case of a bulk order being placed in advance, they will have a few hours to choose whether or not to accept the order.

Just like companies like Uber and AirBnB revolutionized the market by introducing the sharing-economy style to gaps in our society, we believe HomeEats can have a similar effect on the food-delivery and production markets. Not only does it provide people with a healthy alternative to quick and convenient dining, we believe it will increase the number of people who actually use food-delivery apps as a regular source of food. As an added bonus, it prevents food

wastage and gives cooks a productive and systematic way to share their dishes either for hobby or for extra money.

1.1 Problem Statement/Contributions

The problem that this project is trying to solve is the lack of access to a market of home-cooked meals for both the suppliers and the consumers. This project recognizes that there are two groups of people, people who want to sell their home-cooked meals and consumers who want to purchase home-cooked meals, that are not currently connected, and both will benefit when provided a platform to connect. Our system is a platform to connect these two demographics. Not only does it connect both demographics, but it also provides an extra source of income for at home cooks, a healthier alternative to more traditional food delivery and a nutritional benefit to the consumer.

We were able to create a web-based application which served as a functional marketplace for cooks to sell their home-cooked meals on demand to consumers. The application allows a cook to post their meal as being available for purchase and apply the appropriate tags to said meal. A consumer is then able to purchase a meal by searching and/or entering their personal preferences for what they would like to eat. The application also provides the cook with delivery estimates, and the ability to notify the consumer of delivery updates. We also implemented a messaging feature within the application to allow consumers and cooks to communicate about the meals regarding preferences. We also implemented the ability for the cooks to view reports of their sales, so they know exactly how much money they are making from each sale, and how

much the platform is taking as a fee. Our application also has many admin features, such as the ability to approve cooks and view the revenue a cook is producing. Since our project was designed to be completed in this time we were able to successfully fulfill the problem statement by creating a marketplace where both demographics can have their needs met.

2. Related Work

There are many other systems that do food delivery including but not limited to UberEats, GrubHub, DoorDash, Seamless, Postmates, and more. They all do more or less the same thing of delivering food from restaurants to customers. All work via mobile application and aim for marketing advantages over each other by creating exclusive servicing deals with particular restaurants. For example, if one food-delivery service has exclusive rights to a popular restaurant then any customer who wants food from that restaurant delivered must use that particular application. These popular food delivery platforms also outsource their delivery process to third-party drivers that also interact with the platforms via a mobile application. The main way that these third-party drivers generate an income is through a base pay and tips.

While these are popular and successful systems, they do not address a very large gap in the market: home-made food. People opt for food-delivery services either because they cannot themselves cook or do not have the time to cook and convenience takes priority. Unfortunately, restaurant food can oftentimes be unhealthy especially for many consecutive meals in a row. These delivery apps have done a great job of trying to incorporate different dietary restrictions (vegan, gluten-free, etc) and providing services to healthier restaurants (such as salad places) and continue to pursue these goals. That being said, none of them provide the health benefits of home-made food with natural ingredients and no preservatives. This is one strength of our system. Another gap in current systems is that not all people want restaurant food. People are busy and require convenient and quick food but that does not mean they necessarily want food from a restaurant - it just is usually the only alternative. Our system fills this gap as well by allowing people an alternative to enjoy handmade food from homes - meals that are more

intimate, unique, and give a sense of home that may be missing from people's otherwise busy lives. This is also centralizing the process to only one person, the cook, who takes care of everything from cooking the food to delivering the food to the customer. This approach creates a more personalized experience for the customers, compared to other food delivery platforms that break up the process to multiple people. For these reasons, we believe our system is a necessary component to the current market and has the potential to be widely successful.

The system can exist as its own application but can also be something bought and auctioned for by any of the aforementioned existing systems. It would be quite simple and logical to have one application from which people can order food whether from a restaurant or home. The way our system is designed very closely mirrors the restaurant food delivery application design and would be easy to integrate into existing models.

3. System Design

Our application was built using primarily Django, which is a popular framework based in the Python programming language often used for web applications of this scale. HTML, Javascript, and CSS (Cascading Style Sheets) were also used in order to style the user interface and provide extra functionality to the web application. Our application also connects to a SQL Database which stores all the information on the site from cooks and customer user accounts to dishes, reviews, and purchases. The Stripe API (Application User Interface) was integrated into the application to facilitate processing credit card and debit card transactions when food purchases are made. In order for our team to simultaneously contribute to the project, all of our code was hosted in Github, an online software development platform. The project was completed over the course of the Fall and Spring semesters of the 2019-2020 school year through our CS Practicum class.

Our team met with our client bi-weekly at the end of each sprint cycle. For our purpose, a sprint cycle is a two-week development period where each team member is tasked with a feature to implement or work on. A Trello board was used in order to keep track of completed tasks and future goals through the semesters. The purpose of the sprint cycle is to allow for continuous planning, focused development, and short-term goals that the team can set and strive to reach. For example, a sample sprint goal may be implementing the customer landing page where they can browse through dishes. The following sprint goal may be adding filters to the dishes to allow for improved site navigation. These chronological short-term goals led to the long-term goal of launching a fully functional site, allowing customers to order meals directly from cooks in their area, and give full administrative control to our client. Administrative features include approving

cooks when they sign up, removing customers, and viewing food ordering reports. HomeEats fits into the health and social dimensions of the food delivery app market by providing the same convenience and ease of use without sacrificing food quality.

3.1 System Requirements

Gathering system requirements are very important because it provides a solid foundation for the system and gives the project team a clear roadmap of the development cycle and how to prioritize tasks based on time and importance. For our application there are three main user categories: the cook, the admin, and the customer. Below are our requirements split up by category as well as necessity.

Minimum:

Admin Users:

- As an admin, I should be able to approve any cook account before it is created, so that I can guarantee the customers a reliable experience
- As an admin, I should be able to make sure any updates to a cook's personal account information are reviewed, to maintain their accuracy
- As an admin, I should be able to review reports and reviews, and potentially ban users or cooks, to keep the site safe from bad actors
- As an admin, I should be able to view cooks' online time and offline time per week.
- As an admin, I should be able to view the orders that a cook has received.
- As an admin, I should be able to view all accepted orders with the total amount paid including the amount going to the cook and HomeEats.

- As an admin, I should be able to view cooks' cancelled order history with reasons.
- As an admin, I should be able to view revenue reports that can be adjusted to a specific timeframe (week, month, quarter, semi-annual, annual, custom).
- As an admin, I should be able to set which reasons a cook can give for cancelling an order.
- As a system administrator I should be able to ensure that a payment option is selected prior to an order being processed, so that I can ensure meals are paid for before a cook begins to prepare the dish.

Cook Users:

- As a cook, I should be able to apply as a cook with my First Name, Last Name, Address, and Kitchen License
- As a cook, I should not be able to login to my account unless my application has been approved
- As a cook, I should be able to set what plates are available as soon as I log in, so that I can quickly get online and start receiving orders
- As a cook, I should be required to enter/edit my name, email, phone and address, so that I can be contacted in many ways
- As a cook, I should be able to add a new dish to my list of available dishes
- As a cook, I should be able to set a picture, ingredients, price, time to deliver, name, and type of food

- As a cook, I should be able to set which dishes I'm willing to make, so that I don't have to make dishes that I don't have their ingredients.
- As a cook, I should be able to report customers and their reviews, in order to protect my reputation from unfounded criticism and ban disrespectful or malicious customers
- As a cook, I should be able to make a separate account if I want to order through the site, so that I don't get confused between things I've ordered and things I have to cook
- As a cook, I should be able to set my own delivery range centered at my address, so that I am not pressured to deliver outside of my comfort zone
- As a cook, I should be able to tag food as vegan, allergy, etc. so that customers can choose foods which are suitable for them
- As a cook, I should be able to set a certain mileage I'm willing to travel so that I can have quick and efficient delivery service
- As a cook, I should be able to set a limit on how many meals I can make in a specified time frame, so that I don't get overbooked
- As a cook, I should be able to set when I am open and closed, so that customers can't attempt to order food from me when I am not available
- As a cook, I should be able to accept or reject meal orders so that I have control over what meals and how many meals I am making
- As a cook, I should be able to set an estimated cooking and delivery time, so that customers are aware of an approximate waiting time

Customer/Diner Users:

- As a customer, I should be able to see the ingredients in the dishes I plan to order
- As a customer, I should be able to see a picture of the dish I plan to order
- As a customer, I should be able to see the cost of the dish I plan to order
- As a customer, I should be able to see the estimated time of cooking for the dish I plan to order
- As a customer, I should be able to view the type of food I plan to order (e.g. Chinese, Thai, Indian, Mexican, etc.)
- As a customer I should be able to sort the dishes by price
- As a customer I should be able to sort the dishes by rating
- As a customer I should be able to sort the dishes by the distance where the cooks are from me
- As a customer I should be able to sort the dishes by the type of food I plan to order (e.g. Chinese, Thai, Indian, Mexican, etc.)
- As a customer I should be able to rate the food I purchase on a scale of 0-5 stars
- As a customer, I should be able to favorite a cook or a dish, so that I can easily find the cook or dish again
- As a customer, I should be able to review the dishes that I order, so that other customers are aware of the quality of that dish
- As a customer, I should be able to see a delivery status that indicates started cooking, on the way, and delivered so that I know when to expect my food

- As a customer, I should be able to set multiple addresses so that my food can be delivered to a location, even if I am not yet there
- As a customer, I should be able to see an average rating for each dish if the data is available
- As a customer, I should be able to cancel an order that has not started cooking yet so that I don't waste food and money if I change my mind
- As a customer I want to be able to tip the chef preparing my dish so that I can reward and encourage my favorite chefs.

Desired:

Cook Users:

- As a cook, I should be able to set a limit on how many meals I can make in a specified time frame, so that I don't get overbooked

Customer/Diner Users:

- As a customer, I should be able to order at least 3 hours in advance

Optional:

Customer/Diner Users:

- As a customer, I want personal information to be anonymous when messaging the cook, so that my information is kept private
- As a customer I want to be able to message my cook to customize the order to my liking.

3.1 Wireframes

Wireframes are crucial to software development because it forces the developers to tangibly visualize the system and allows the client to see the potential designs for the system prior to the start of implementation. This visualization fosters an extremely important and necessary communication between the client and development team that allows them to take steps to improve the product to better fit the customer needs and requirements in a productive manner.

Below are the wireframes we made at the beginning of our project last semester. Though our design changed significantly since then, the wireframes illustrate the base design we envisioned and how the bi-weekly customer meetings led us to evolving the design of the system into what it is now.

A wireframe for a login screen. It features a white header with the word "LOGIN" in black. Below the header is a light blue background containing two white input fields labeled "USERNAME" and "PASSWORD". At the bottom, there are two blue buttons: "LOGIN" on the left and "Forgot Password?" on the right.

Figure 1 - Login Screen

A wireframe for a customer checkout screen. It has a white header with a browser address bar showing "http://HomeEds/Details/Checkout". Below the header is a light blue background. On the left, there's a "Back to Food" button and a list of items: "Current Order - Fettuccini Alfredo", "Quantity - 1", "Price - \$13.99", "Tax - \$1.00", "Delivery Fee - \$2.00", and "Total - \$16.99". On the right, there's a "Checkout" button and three payment options: "Cash", "Credit/Debit", and "Venmo". A note says: "Pay the total amount in Cash to the delivery driver when he/she comes to your door. Please remember this does not include tip."

Figure 2 - Customer checkout screen with their itemized order and payment options.

A wireframe for a customer account creation screen. It has a white header with the text "Account Creation - Customer". Below the header is a light blue background containing several white input fields: "First Name", "Last Name", "Email", "Phone Number", "Street", "City", "State" (with a dropdown arrow), "ZipCode", "Username", and "Password". At the bottom, there is a blue button labeled "CREATE!".

Figure 3 - Customer Account Creation screen with necessary personal information requested.



Figure 4 - Cook Order screen which shows them the number of orders for each dish they currently have, allows them to accept or reject pending order requests.

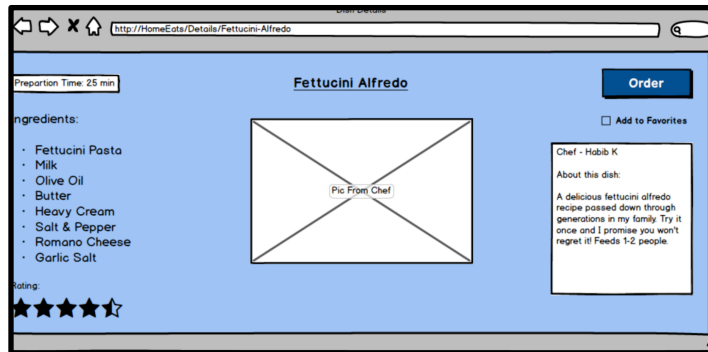


Figure 5 - Dish Details screen with information about the Chef and important details about the Dish such as ingredients, previous picture of it, the rating, and the preparation time.

3.3 Sample Code

Models.py

```
'''
Columns in the customer database table. Customer model is a type of user account, as
specified by the one-to-one field with the user model.
'''
class Customer(models.Model):
    #False = account has not been banned; True = account has been banned
    banned = models.BooleanField(default=False)

    phone_number = models.CharField(null=False,max_length=10, default="")

    # Each user account can have maximum one relationship with a customer,
    vice-versa
    user = models.OneToOneField(User, on_delete=models.CASCADE)

    #User can have many favorite dish objects
    favorites = models.ManyToManyField(Dish, blank=True)

    def __str__(self):
        return self.user.first_name + " " + self.user.last_name + " (" +
            str(self.id) + ") "
```

```
'''
Columns in the cook database table. The cook is a type of user, as specified by
the one-to-one field with the User model
'''
class Cook(models.Model):
    # False = account has not been banned; True = account has been banned
    banned = models.BooleanField(default=False)

    # False = account has not been approved; True = account has been approved
    approved = models.BooleanField(default=False)

    # False = cook is not currently accepting orders; True = cook is currently
    accepting orders
    online = models.BooleanField(default=False)

    # Kitchen license identification code
    kitchen_license = models.CharField(max_length=30)

    government_id = models.ImageField(default="", upload_to='cook_government_ids')

    phone_number = models.CharField(max_length=30, default="")

    # Each user account can have maximum one relationship with a cook, vice-versa
    user = models.OneToOneField(User, on_delete=models.CASCADE)

    delivery_distance_miles = models.IntegerField(default=30)

    delivery_fee = models.DecimalField(default=0, decimal_places=2, max_digits=6)

    def __str__(self):
        return self.user.first_name + " " + self.user.last_name + " (" +
            str(self.id) + ")"
```

```
'''
Columns of the dish database table. Cooks have the ability to generate new dishes in
the database table.
'''
class Dish(models.Model):
    # title is the name of the dish
    title = models.CharField(default="", max_length=30)
```

```

# False = dish is available for order; True = dish is unavailable for order
cook_disabled = models.BooleanField(default=False)

# Specifies what kind of food the dish is, e.g. Italian
cuisine = models.ForeignKey(Cuisine, on_delete=models.CASCADE)

description = models.CharField(default="", max_length=200)

# List of ingredients that compose the dish, stored as an array in the database table
ingredients = ArrayField(models.CharField(max_length=30, blank=True), default=list)

dish_image = models.ImageField(default="", upload_to='dishes')

cook_time = models.IntegerField(default=0)

price = models.DecimalField(default=0, decimal_places=2, max_digits=6)

# Specifies the cook that makes the dish
cook = models.ForeignKey(Cook, on_delete=models.CASCADE)

# Average of all numeric ratings from customers
rating = models.IntegerField(default=0)

# False = dish is not vegan; True = dish is vegan
vegan = models.BooleanField(default=False)
allergies = models.CharField(default="", max_length=200)

def __str__(self):
    return self.title + " (" + str(self.id) + ")"

class Meta:
    verbose_name_plural = "Dishes"

```

The models defined here are for the Customer, Cook and Dish. These models hold the majority of our data as they define all the attributes necessary for keeping track of which cook makes what dishes and the information we need to provide to the customer regarding the dish. The

Cook and Customer models show info regarding two types of users that will interact with our system.

Views.py

```
'''
View of the customer creation form with form validation.
'''

def customercreate(request):
    if request.method == 'POST':
        form = forms.CustomerCreateForm(request.POST)
        address_form = forms.AddressCreateForm(request.POST)

        if form.is_valid() and address_form.is_valid():
            data = form.cleaned_data
            address_data = address_form.cleaned_data

            if verify_address(address_data['street'], address_data['town'],
address_data['state']): #Checks that the address is a valid location
                '''
                Creating new user and customer objects to save and link together
                '''
                user = User.objects.create_user(username=data['email'], email=data['email'],
password=data['password'], first_name=data['first_name'], last_name=data['last_name'])
                customer = models.Customer.objects.create(phone_number=data['phone_number'],
user_id=user.id)
                user.is_customer = True
                user.save()
                customer.save()

                '''
                Get customer id of customer object just made, then linking and saving it to
customer's address object
                '''
                customer = models.Customer.objects.get(user_id=user.id)
                address = models.Address.objects.create(customer=customer,
street_name=address_data['street'], city=address_data['town'],
state=address_data['state'], zipcode=address_data['zipcode'],
current_customer_address=True)
```

```

address.save()

'''
Creating a shopping cart object for customer to store orders in
'''
shopping_cart = models.ShoppingCart.objects.create(customer=customer)
shopping_cart.save()
messages.add_message(request, messages.SUCCESS, 'Your account has been
successfully created, login to start ordering now!')

# Confirmation email of account creations
send_mail(
    'Welcome to HomeEats',
    'Your HomeEats account has been created and you can start ordering now!',
    'capstonecustomer2020@gmail.com',
    [user.email],
    fail_silently=False,
)
return HttpResponseRedirect(reverse('login'))
'''

If any of the customer information is incorrect, keep form filled out but display
error messages
'''
else:
    form = forms.CustomerCreateForm()
    address_form = forms.AddressCreateForm()
    return render(request, 'customer_create.html', {'form': form, 'address_form':
address_form})
@login_required
@cook_required
def delete_dish(request, dish_id):
    dish = Dish.objects.get(id=dish_id)
    cuisine = Cuisine.objects.get(id=dish.cuisine_id)
    cook = Cook.objects.get(user_id=request.user.id)
    if (dish.cook == cook): #only allow a cook to delete his/her own dish
        dish.delete()
    objs = Dish.objects.filter(cook=cook, cuisine=cuisine)
    if not objs: #check if they are deleting the only dish left in the cuisine
        cuisine.cooks.remove(cook) #if so, remove them from the cuisine
        return HttpResponseRedirect(reverse('cook_manage')) #cuisine doesn't exist so
redirect to cook/manage
    return HttpResponseRedirect(reverse('cook_cuisine_dishes', args=[cuisine.id]))
#redirect to cuisine because it exists

```

```
return HttpResponseRedirect(reverse('cook_manage'))
```

```
'''
If the heart-shaped like button on the dish card is empty, the customer can press the
button to
save the dish to the arraylist of favorite dishes.

Else the heart-shaped like button on the dish card is filled red, the customer can
press the button
to remove the dish from the arraylist of favorite dishes.
'''
@login_required
@customer_required
def toggle_favorite(request):
    if request.method == "POST":
        dish = Dish.objects.get(id=request.POST["dish_id"])
        customer = Customer.objects.get(user_id=request.user.id)
        if dish in customer.favorites.all():
            customer.favorites.remove(dish)
            customer.save()
            data = {
                'status': dish.title + ' favorite removed'
            }
        else:
            customer.favorites.add(dish)
            customer.save()
            data = {
                'status': dish.title + ' favorite added'
            }

    return JsonResponse(data)
```

The controllers defined here are `customercreate`, `delete_dish` and `toggle_favorite`. We have an example of a controller that allows a customer to create an account on our site that will further allow them to place orders from our cooks. Delete dish is an example of how our cooks can interact with the system by deleting dishes they no longer want to cook for customers.

`Toggle_favorite` shows an example of our customers interacting with the system as they can

make a dish they have ordered or are planning to order a favorite dish that they can easily access later through our favorites feature.

Forms.py

```
'''
Information the customer needs to enter to create an account
'''
class CustomerCreateForm(forms.ModelForm):
    error_css_class = 'error'
    first_name = forms.CharField(label='First Name', required=True,
    error_messages={'required': 'Please enter your first name.'},)
    last_name = forms.CharField(label='Last Name',
    required=True, error_messages={'required': 'Please enter your last name.'})
    password = forms.CharField(widget=forms.PasswordInput())
    email = forms.EmailField(required=True,)
    phone_number = forms.CharField(label='Phone Number')

    class Meta:
        model = Customer
        fields = ('first_name', 'last_name', 'password', 'phone_number',)

    '''
    Validates phone number field
    '''
    def clean_phone_number(self):
        data = self.cleaned_data['phone_number']

        if not data.isdigit(): # Checks that the phone number only contains digits
            raise forms.ValidationError('Enter a valid phone number, e.g. 0123456789')

        elif len(data) != 10: #Checks that the phone number only contains 10
characters
            raise forms.ValidationError('Enter a valid phone number, e.g. 0123456789')
        return data

    '''
    Validates the email field
    '''
    def clean_email(self):
        email = self.cleaned_data.get('email')
```



```

        if User.objects.filter(username=email).exists(): # Checks that another account
with the entered email does not exist
            raise forms.ValidationError("An account with this email already exists, go
to login page or use a different email")
        return email

```

```

class DishCreateForm(forms.Form):
    title = forms.CharField(required=True,)
    dish_image = forms.ImageField()

    # Displays all of the cuisines in the cuisine database table as a drop down
selection
    cuisine = forms.ModelChoiceField(queryset=Cuisine.objects.all(),empty_label='Select
a cuisine')

    # Allows the cook to enter the ingredients into a text field and splits the text at
commas as saves ingredients into an array
    ingredients = SimpleArrayField(forms.CharField())
    description =
forms.CharField(required=True,widget=forms.Textarea(attrs={'style':'width=50%;','rows'
:3}))
    cook_time = forms.IntegerField(required=True, label='Cook time (in
minutes)',min_value=1)
    price = forms.FloatField(required=True,min_value=0.00,
widget=forms.NumberInput(attrs={'step':0.01}))
    vegan = forms.BooleanField(required=False,initial=False)
    allergies = forms.CharField(required=False)

    class Meta:
        model = Dish
        fields = ['title', 'cuisine','description', 'dish_image',
'ingredients','price','cook time','vegan','allergies']

```

```

'''
Filters dish results based on search criteria and can sort dishes based on rating and
price
'''

class DishSearchForm(forms.Form):
    search = forms.CharField(label="Search",max_length=30, required=False,
widget=forms.TextInput(attrs={'placeholder':'Search','class':'form-control mr-sm-2'}))
    SORT_CHOICES = (
        ('rating', 'Sort: Rating'),

```

```

        ('price', 'Price: Low to High'),
        ('reverse_price', 'Price: High to Low'),
    )
    cuisine_types = Cuisine.objects.all()
    cuisines = [ ('none', 'Cuisine: All')]

    for cuisine in cuisine_types:
        cuisines.append((cuisine.id, 'Cuisine: '+cuisine.name))
    sort = forms.ChoiceField(
        choices=SORT_CHOICES,
        widget=forms.Select(attrs={'onchange':'submitForm()','class':'custom-select','style':'font-size:10pt;margin-right:10px'}),
        required=False)
    cuisine = forms.ChoiceField(
        choices=cuisines,
        widget=forms.Select(attrs={'onchange':'submitForm()','class':'custom-select','style':'font-size:10pt;margin-right:10px',}),
        required=False)

```

Here we have a few forms that are used to validate data as our users interact with our system.

The Dish Search form keeps track of the filters entered by the user on the homepage to determine what dish listings to provide to the user.

3.4 Sample Tests

Testing is very important to ensure the integrity of any web application. Without comprehensive tests, an application can have bugs that ruin the reliability and potentially the safety of the entire system. Maintaining 100% testing coverage of all code is critical so that the application is guaranteed functional, and future changes to the code do not have unintended consequences that go unnoticed.

```

def test_add_and_remove(self):
    self.client.login(username='anki@anki.com', password='ankith')
    self.client.post(reverse('addtocart'), {'dish_id': 1})

```

```
response = self.client.post(reverse('removefromcart'), {'dish_id': 1})
self.assertEqual(response.status_code, 200)
```

test_add_and_remove ensures that adding and removing dishes from the cart are working properly. The test logs in as a customer, adds a dish to the cart, and then removes that dish from the cart. We assert that removing the dish from the cart returns a success status code which affirms that the dish was removed.

```
def test_change_status_from_cooking_to_delivery(self):
    self.client.login(username='ramsey@ramsey.com',password='ramseyramsey')
    response = self.client.get(reverse('cook_home'))
    self.client.get(reverse('cooking_to_delivery', args=[2]))
    order = Order.objects.get(id=2)
    self.assertEqual(order.status, 'o')
```

test_change_status_from_cooking_to_delivery checks that the view for changing the status from Cooking to Out For Delivery is working correctly. The test logs in as a cook, goes to the cook home page, calls the view to change the status, and then checks that the status is now Out For Delivery.

3.5 Code Coverage

Our team used coverage.py to test our code coverage. With pip already installed, coverage.py was installed using the command: **pip install coverage**

Next, coverage.py was run with the following command:

```
coverage run --source='.' manage.py test homeeats_app
```

We then generated a coverage report with: **coverage html**

Coverage report: 100%

<i>Module</i> ↓	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
homeeats\settings.py	31	0	0	100%
homeeats\urls.py	6	0	0	100%
homeeats_app\admin.py	51	0	0	100%
homeeats_app\decorators.py	10	0	0	100%
homeeats_app\forms.py	154	0	0	100%
homeeats_app\models.py	154	0	0	100%
homeeats_app\urls\cook_urls.py	3	0	0	100%
homeeats_app\urls\customer_urls.py	3	0	0	100%
homeeats_app\urls\main_urls.py	4	0	0	100%
homeeats_app\views\admin_views.py	108	0	0	100%
homeeats_app\views\cook_views.py	353	0	0	100%
homeeats_app\views\customer_views.py	475	0	0	100%
homeeats_app\views\main_views.py	149	0	0	100%
Total	1501	0	0	100%

coverage.py v5.0.3, created at 2020-03-29 23:16

3.6 Installation Instructions

AWS Database Setup

1. Create AWS account if necessary and launch AWS console
2. Go to RDS (Relational Database Service)
3. Click on “Create Database” and you will be taken to a database creation module
4. Choose “Standard Create” for method and “PostgreSQL” for the engine type
5. Use the free tier template if this is meant for development or a production template if meant for production
6. Give your database a name, username and password in the Settings section, make sure to write them down because you will need them later
7. Under connectivity, hit the “Additional connectivity configuration” dropdown, and select **Yes** for publicly accessible.
8. Keep everything else the same as default and select “Create Database”
9. In the console, click on your new database so you are on a page that looks similar to this:

The screenshot shows the Amazon RDS console for instance 'dev-homeeats1'. The left sidebar contains navigation links like Dashboard, Databases, Query Editor, etc. The main panel shows the instance details under the 'Summary' tab, including DB identifier, CPU usage (3.50%), Info (Available), Class (db.t2.micro), Role (Instance), Current activity (0 Sessions), Engine (PostgreSQL), and Region & AZ (us-east-1f). Below this, the 'Connectivity & security' tab is active, showing details for Endpoint & port, Networking (Availability zone, VPC, Subnet group), and Security (VPC security groups, Public accessibility).

10. Scroll down to security group rules and add rules necessary so that your security group rules look like this:

The screenshot shows the 'Security group rules' section for the 'default (sg-0fa8f47e)' security group. It displays two rules: one for CIDR/IP - Inbound and another for CIDR/IP - Outbound, both with a rule of 0.0.0.0/0. Below this, the 'Replication' section is visible, showing one rule.

Installing Python3 on Linux

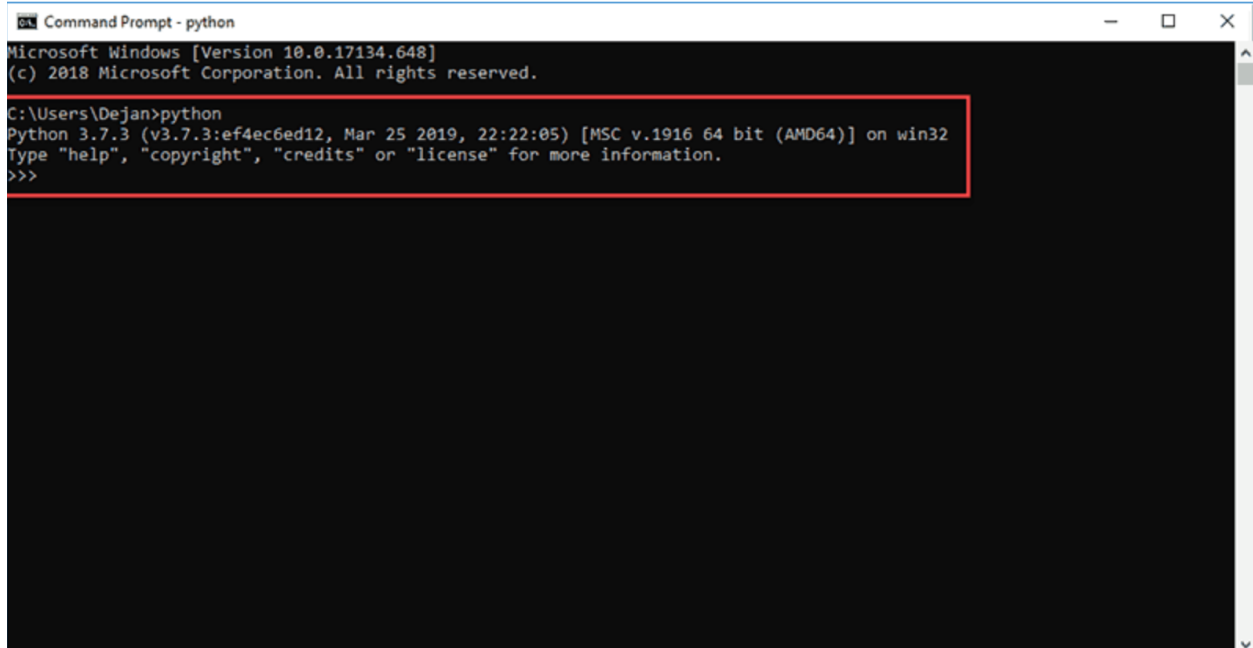
1. Open the Terminal, type “python3 --version” to see if you have Python3 installed. If python3 has been installed, then there is no need to reinstall python3.
2. If you are using Ubuntu 16.10 or newer, then you can easily install Python 3.6 with the following commands:
 - a. `sudo apt-get update`
 - b. `sudo apt-get install python3.6`
3. If you’re using another version of Ubuntu
 - a. `sudo apt-get install software-properties-common`
 - b. `$ sudo add-apt-repository ppa:deadsnakes/ppa`
 - c. `$ sudo apt-get update`
 - d. `$ sudo apt-get install python3.6`
4. Check if pip3 has been installed by entering “command -v pip3” into the command line

Installing Python 3 on a Mac

1. Open the Terminal, type “python3 --version”. If python3 has been installed, then there is no need to reinstall python3.
2. Install Xcode developer tools by typing, “xcode-select --install”
3. Install HomeBrew by typing and entering, “ruby -e “\$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)”
4. Then type and enter: export PATH="/usr/local/opt/python/libexec/bin:\$PATH"
5. Install Python3 and Pip by entering, brew install python
6. Check to make sure Python3 was installed by entering “python3 --version”

Installing Python3 on Windows

1. Go to <https://www.python.org/downloads/windows/>
2. Download the Windows x86 executable installer if your Windows installation is a 32-bit system. Download the Windows x86-64 executable installer if your Windows installation is a 64-bit system.
3. Run the Python Installer once the installer has finished downloading.
4. Make sure you select the Install launcher for all users and Add Python 3.7 to PATH checkboxes.
5. Select Install Now
6. Open the Command Prompt by opening the windows Start menu and typing “cmd”. Then, open the Command Prompt application.
7. Verify that Python3 was installed. Using the Command Prompt, navigate to the directory in which Python was installed on the system. Double-click python.exe. The output should be similar to what you can see below:



```
Command Prompt - python
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Dejan>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

8. Verify Pip was installed: enter “pip -V” into the console. If pip was installed correctly, the version number and location will be outputted to the console.

Clone the HomeEats repository

1. Open the Command Prompt application on Windows or the Terminal application on Mac.
2. Move to the directory where you would like the HomeEats repository to be installed.
3. Enter into the command line, git clone <https://github.com/uva-cp-1920/HomeEats.git>

Database Configuration in HomeEats Repo

1. Go to the AWS console and to the RDS service

- Click on the RDS instance you created to open a page looking like this:

The screenshot shows the AWS RDS console page for an instance named 'dev-homeeats1'. The page has a breadcrumb trail 'RDS > Databases > dev-homeeats1'. At the top right are 'Modify' and 'Actions' buttons. Below is a 'Summary' section with a table of instance details:

Summary			
DB identifier dev-homeeats1	CPU 3.50%	Info Available	Class db.t2.micro
Role Instance	Current activity 0 Sessions	Engine PostgreSQL	Region & AZ us-east-1f

Below the summary is a tabbed interface with 'Connectivity & security' selected. It shows three columns of configuration:

Connectivity & security		
Endpoint & port Endpoint dev-homeeats1.cetolplhgphy.us-east-1.rds.amazonaws.com Port 5432	Networking Availability zone us-east-1f VPC vpc-356b0c4c Subnet group default-vpc-356b0c4c Subnets subnet-39930f15	Security VPC security groups default (sg-0fa8f47e) (active) Public accessibility Yes Certificate authority rds-ca-2015 Certificate authority date

- Go to the settings.py file in your HomeEats Repo and copy/paste the below to replace what is currently in the DATABASES variable.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'homeeats',
        'USER': 'postgres',
        'PASSWORD': 'homeeats123',
        'HOST': 'dev-homeeats1.cetolplhgphy.us-east-1.rds.amazonaws.com',
        'PORT': '5432',
    }
}
```

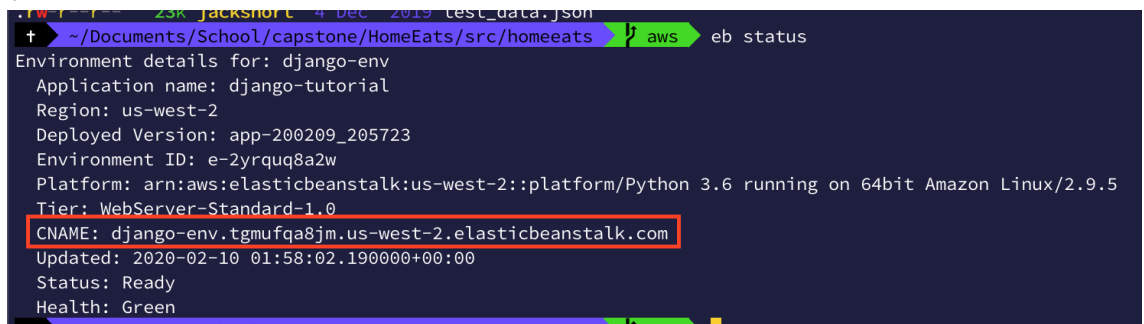
- Replace the NAME, USER and PASSWORD with the credentials you wrote down from the first part of these installation instructions when you provisioned your RDS instance
- Replace HOST and PORT with the Endpoint & Port shown in the screenshot from step 2

Install HomeEats dependencies

- On the command line, from the HomeEats root folder, “cd src/homeeats”
- Type “ls” to ensure that you are in the right location. You should see requirements.txt listed as a file in that directory.
- Then enter into the command line, pip3 install --upgrade -r requirements.txt.

Deploying to AWS:

1. In a terminal install awsebcli
 - a. `pip3 install awsebcli`
2. CD into the HomeEats root folder
3. Initialize EB Cli repository
 - a. `$ eb init -p python-3.6 <APPLICATION NAME>`
4. Create an environment and deploy the application to it
 - a. `$ eb create <ENVIRONMENT NAME>`
5. Find the domain name of the new environment
 - a. `$ eb status`



```
23k jackshort 4 Dec 2019 test_data.json
+ ~/Documents/School/capstone/HomeEats/src/homeeats ➤ aws ➤ eb status
Environment details for: django-env
Application name: django-tutorial
Region: us-west-2
Deployed Version: app-200209_205723
Environment ID: e-2yrquq8a2w
Platform: arn:aws:elasticbeanstalk:us-west-2::platform/Python 3.6 running on 64bit Amazon Linux/2.9.5
Tier: WebServer-Standard-1.0
CNAME: django-env.tgmufqa8jm.us-west-2.elasticbeanstalk.com
Updated: 2020-02-10 01:58:02.190000+00:00
Status: Ready
Health: Green
```

6. Edit the settings.py file located at <HOMEEATS ROOT>/src/homeeats/homeeats/settings.py
 - a. Add your applications CNAME to the ALLOWED_HOSTS array
 - b. For example: `ALLOWED_HOSTS = ['128.143.67.97', 'localhost', '127.0.0.1', 'django-env.tgmufqa8jm.us-west-2.elasticbeanstalk.com']`
7. Save the file then deploy the application
 - a. `$ eb deploy`
8. When the deployment process completes you can view your application with
 - a. `$ eb open`

4. Results

Our system was able to address all the problems we planned on tackling. Customers create an account on the site, and are immediately able to see dishes available near them, filter their selection and the checkout through the web application app. Our initial testing results have determined that it takes approximately 5 minutes for a customer to create an account, order their first dish and complete the checkout/payment process. This is quite efficient and on par with the time it takes to set-up and begin using our competitors' applications. However, since our platform is unique in the market due to us providing home-cooked meals, our customers are not sacrificing any additional time and are ending up with much healthier food alternatives.

From the cook's perspective, it also takes less than 5 minutes to apply for an account, not including the time it takes the admin to approve the cook, which is estimated as within 24 hours. After approval it will take less than 5 minutes for the cook to put their first dish up on the site and begin taking orders. Connecting amateur cooks to nearby consumers was one of our primary objectives. Similarly, for the Admin users the features are instantaneous and they have 24 hours to approve cooks which, in it of itself, takes less than 5 minutes. Overall, the system addresses all of our initial requirements and allows for the smooth usage of all three of our stakeholders - customers, cooks, and admin users.

5. Conclusions

Our system has the potential to revolutionize the food delivery industry. Food delivery apps are currently used for convenience. They allow people to continue working and accomplish other tasks, without having to allocate time to get groceries, cook, or clean. The small sacrifice that is made in terms of food quality is made up for in terms of productivity. However, HomeEats completely eliminates that sacrifice. It is now a win-win situation, people are able to continue being productive and finishing their tasks, while having freshly prepared home cooked meals delivered straight to them. Furthermore, our network of amateur chefs can most likely produce higher quality food that the average consumer may be incapable of making. The system benefits people who cannot cook, people who are too busy to cook, and people who want healthier convenient options in food.

There are an enormous number of people that this system can benefit. College students living in dorms without proper kitchens can have access to fresh meals that they are missing. Parents who are running late at work do not have to pick up frozen meals or fast food to feed their families. Anyone simply not feeling like cooking or going to get groceries does not have to compromise their health and well-being by eating junk food. On the other hand, from the chef's perspective, this is a wonderful opportunity to grow your brand and become a popular figure in the local food community. It is also an opportunity to experiment with cooking new dishes and seeing how customers respond (in the form of reviews). Even if the cooks in the system are not aspiring chefs, this is a great opportunity for people with extra time/extra food/cooking abilities to make extra money on the side. It also helps promote a healthy, non-screen activity at home for those cooks as society continues to search for healthier alternatives than hours on a laptop/

mobile/tv screen. Finally, it also helps reduce food wastage! As the world looks for ways to reduce wastage across the board, it is important to understand how much food gets wasted by households every week! This system allows people to use their extra food supplies to make another dish that could be of value to someone else in their community and saves them from having to just throw it out.

6. Future Work

Future work on this system should revolve around applications in the real world. Rather than for simply convenience, food delivery applications have the ability to transport food in times of need. Currently, in the spring of 2020, we are facing a global pandemic with the coronavirus. People everywhere are told to practice social distancing and avoid going out. For those people who do not know how to cook, are out of groceries, or are feeling sick, this is a perfect opportunity to have freshly prepared food delivered straight to your residence. With businesses temporarily shutting down, this also allows chefs and drivers to continue to bring in revenue to help them pay the bills and provide for their own families.

Additionally, though this application can sustain itself and certainly has the potential to be a stand-alone system for users across the United States, it also has the potential to be part of existing food delivery applications. From a customer viewpoint, it makes sense for customers to have *one* application where they can order food from either restaurants or from homes. Future research could include the system design for combining the two types of food delivery into one system or even redesigning the current system so it can be purchasable to integrate into *any* existing food-delivery applications currently on the market. Research could also include analysis on whether it is better or worse to have the two food delivery systems in one application instead of as separate entities.