ChoreoNova Ticketing System

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

Jared Taylor
Spring, 2020.

Technical Project Team Members
David Crowder
Eldon Luk
Vivian Pham
Jared Taylor

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 Jared Taylor Date 5/11/2020
Jared Taylor

Approved Ahmed Ibrahim Date 4/22/2020
Dr. Ahmed Ibrahim, Department of Computer Science

Table of Contents

Abstract

The main objective for this Capstone project is to develop an online web application system that will allow ChoreoNova, a local Charlottesville fine arts company, to have an efficient and high quality ticketing system by the end of the Spring 2020 semester. The final product was created using Django, a Python based framework that incorporates HTML coding as well as other beautifying languages such as Bootstrap. The product was built using an Agile method of project work, where the entire build process was divided into 12 sprints. Each sprint, a couple of system requirements were pulled off the backlog to be worked on and completed. With the semester system that the University of Virginia functions on, functional requirements were completed in Fall 2019 while additional requirements were added on in Spring 2020. Through this project, the team learned the importance of communication in a software engineering role, particularly with non-technically savvy clients. Moreover, the team also gained a deeper appreciation of the arts. Through this project, the team hopes that they are able to help a small arts production company modernize and stay with the digital era, allowing them to continue to be able to share their stories of humanity with the American public well into the future and beyond.

List of Figures

1. Introduction

What makes humanity different from our mammalian ancestors is not only our ability to think critically but also our appreciation for the abstract. The universal means to express our understanding and comprehension of the abstract is the arts. From the beginning of our history, art has been a way for us to represent the emotions we hold dearest as mankind: love, anger, hope. Without the arts, we have no way of expressing these vital emotions, no way of expressing what makes us human.

ChoreoNova is a non-profit dance presentation company with a mission to develop, produce, and present transformative contemporary dance works that tell impassioned stories. ChoreoNova also hopes to promote personal empowerment and communal justice with all those in the communities they perform in and they help to improve individual and collective resilience amongst community members as well. Thus, their core values include storytelling, well-being, diversity, inclusiveness, cooperation, and self-supporting. ChoreoNova currently holds around three to four events per year. Each of these shows has multiple showings at different times and days, potentially at different venues as well. In recent years, ChoreoNova's performances have moved towards the utilization of more forms of media than just simply dance, integrating the arts with the sciences. Our main objective in this project is to develop a system to allow these dance presentations, as well as other ChoreoNova events, to have an efficient and high quality ticketing system.

1.1 Problem Statement

Currently, ChoreoNova's way of listing events and selling tickets is very disjointed and disconnected due to compatibility issues with the way their main webpage is built. Their current website is built on Joomla, where they have found many compatibility issues when they attempted to integrate new webpages onto their site. Their current method of selling tickets includes having all their events listed on their main website in the "Events" tab. However, users are not able to purchase tickets from that "Events" page, which is merely used to display upcoming events that ChoreoNova is holding. Instead, users must go through another link sent to them in a newsletter via email or found on ChoreoNova's Facebook page to access an EventBrite page to finally order their tickets. Overall, ChoreoNova's current band-aid solution to not being able to integrate links onto their webpage is extremely inelegant and unscalable. First off, this is a very inefficient process for users as after seeing an event they may be interested in on ChoreoNova's main webpage, they would need to somehow find this other link to the EventBrite page via email or Facebook. If this user had not signed up for the newsletter before or does not own a Facebook account, it would be virtually impossible for them to access this EventBrite page and order tickets to attend the event. For more impatient users, all this clicking and searching for the proper link to order tickets may even discourage them from purchasing tickets altogether, resulting in ChoreoNova losing business. On top of that, ChoreoNova is also handing over control of their ticket sales to a third-party company: EventBrite. This means they lose control over a lot of the revenue they generate and also are giving EventBrite valuable data that they could have kept for themselves. As such, our goal is to help ChoreoNova find a more

elegant solution to their ticket sales, integrating everything into one system throughout this school year.

1.2 Contributions

We planned to split most of the work on this ticketing system into two semesters worth of work. The first semester focused on putting together a functioning ticketing system where users can see events and purchase tickets altogether in one system. The second semester focused on adding more additional features into the system. Finally, we worked on the user interface (UI) and user experience (UX) after ensuring functionality was complete. ChoreoNova values a smooth and enjoyable experience for their customers, and therefore wants the ticketing system to reflect this.

This project is important for many reasons, as it supports the arts and the local Charlottesville community. In recent years, the rise of STEM careers has in many cases displaced studies that are sometimes referred to as “softer” studies, such as the arts. Many see STEM as more productive for society, contributing more to further advancement than something like the arts. However, society needs balance and many times, the arts provide that balance by introducing more abstract concepts, such as humanity or justice, all necessary for a full functioning society. With no humanity, society would have lost its human roots, removing us from our very own nature. The arts, especially a company like ChoreoNova, which promotes inclusivity, well-being, and cooperation help to ensure this social aspect of our species is not lost. ChoreoNova also has the added benefit of being a company that is local here to Charlottesville. It

is very important for University of Virginia students to have the opportunity to give back to the surrounding area, and to thank them for welcoming us into their home for the past four years.

2. Related Work

The two main options for a small venue to sell tickets are either to sell tickets at the event, or to use a ticketing service. Selling tickets at the venue creates problems for both the buyer and seller. The buyer will not know if there will be a ticket available for them when they arrive and also will not know where they are sitting. This will cause buyers to get to a venue very early so that they can sit in a desired section. A buyer may also be less influenced to actually go to a show if they have not already bought a ticket. The seller, on the other hand, will also not know how many people to expect, so therefore may not be able to accommodate with the appropriate number of ushers and concession staff.

An online ticketing service solves many of these problems. It allows people to ensure they have a spot at the venue and also allows the venue to know what to expect in terms of turnout ahead of time. The major flaws in ticket services such as TicketMaster and EventBrite are a lack of control and fee payments. In terms of control, demographic information can be very useful for an organization, as it shows them who to market to and what to change in their company to try and attract new audiences. However, TicketMaster does not allow merchants to see this information for free, making it impossible for merchants to have access to this information unless they pay for TicketMaster Artist Services (TicketMaster, n.d.). EventBrite has some analytic tools, but it mostly shows what types of tickets were purchased and at what time (EventBrite, n.d.). The only way they could get the exact information they want from a client would be through a survey conducted outside of these ticketing services. When using a ticketing service Choreonova doesn't own, they also cannot control the exact pipeline of how a ticket is purchased. TicketMaster and EventBrite have their own flow for how tickets are purchased, and

Choreonova wanted a buying system that minimized clicks and pages. These ticket service websites also have service charges they charge per transaction. The merchant can choose to pass these costs onto the buyer, but then they might be less inclined to purchase a ticket online due to these surcharges. The other option is to absorb the cost and keep the ticket cost the same, but that lowers overall profit.

The project aims to create a website with the core functionality of websites such as EventBrite and TicketMaster. However, this website will also be built with the addition of demographics collection and control on the buyer's experience, as well as the removal of service fees. These changes will allow ChoreoNova to regain control of the ticketing process and also keep more of the profit.

3. System Design

The main goal of the capstone project for ChoreoNova is to create a functioning ticketing system that will allow customers to purchase tickets for events. ChoreoNova put a number of requirements that needed to be met by this ticketing system. ChoreoNova's original vision was to have the system embedded on their original website. However, our group felt it was best for testing and building purposes to have the product be a completely separate website and link it straight from ChoreoNovat's own website. This allowed us to make modifications without impacting or conflicting with their other work.

In terms of functional requirements, our group has to prioritize ease for users, as that is the main obstacle facing ChoreoNova's current band-aid solution. The main goal of our project is to allow typical ChoreoNova customers, who have varying levels of technological experience, to be able to easily go on the main ChoreoNova website, click the link to buy tickets, and then purchase the tickets in a reasonable amount of time and clicks.

The front-end is the part of the website that the users can access and interact with, which is built to be logical and streamlined. The client has placed heavy emphasis on reducing the number of clicks and webpage redirects, which is the requirement that the front-end design revolves around. First, the user is redirected from the main website to the homepage of the ticketing website. From here, the user can then pick an event from this page, where there is a list and calendar view. There are also filters if the user wants to reduce the number of events to only see events of a certain type. Once the user has decided on an event, they will then click on the event and be redirected to the ticket purchasing page. Here, the user must input the quantity, location, and any special accommodations they may require. Then, the user will pay for the

tickets and finally be redirected to a confirmation page. This system will default to a simple system that a person who is not technologically savvy will be able to complete, but will also provide the tools, such as calendars and filters, that allow more advanced users to have a more refined experience.

The product needs to also have an easy-to-use back-end system for our client or any other ChoreoNova employees to access the website as an admin and be able to modify objects, view statistics, and to look at user demographics so they can plan for their future events. Having an admin back-end system will give the flexibility for ChoreoNova to expand with more employees and more shows if they choose to do so in the future. This will lay the groundwork for years, if not decades, of future development. The back-end for this system will hold all the information that ChoreoNova will need in regards to ticketing, with the most important information including the event that tickets are being sold for, how many tickets are left for that event, the price of tickets, a seating chart if relevant, and special requests that specific customers may have.

Our team is using Django, a web framework that uses Python as the programming language and has an Model-View-Controller (MVC) structure, which is a three-tiered structure that keeps data, presentation, and logic separate from each other. Using this framework allows our ticketing system to be able to store tickets and events as models, display the desired information as a view on the rendered webpage, and control how a user interacts with the system.

As a team, we decided to license our code under a GNU General Public License. We feel that, given the scope of our project, it made the most sense to use a public licensing system for our developed code. Another major reason why we decided to license under GNU is because the

license was free, which is most optimal for both our group and our client so that they would not have to worry about paying for a license after the project is complete.

3.1 System Requirements

System requirements are the various functionalities that the system we are building must contain, per our customer's wishes. Gathering system requirements allows us to begin the process of building a system by planning what needs to be done. By splitting the list of system requirements into minimum requirements, desired requirements, and optional requirements, we know which features should be prioritized. The minimum requirements are as follows:

- As a user, I should be able to buy a ticket to the event I selected so that I have a verified method of entrance.
- As a user, I should be able to input any special requests so that the company knows how to accommodate my needs.
- As the owner, I need a way to easily accept payments for each ticket transaction.
- As an admin, I should be able to create an event and include the appropriate details so that users can see what events are occurring.
- As an admin, I should be able to track ticket sales so that I can see the number of remaining tickets for an event.
- As an admin, I should be able to see which tickets have a special request so that I can properly accommodate their needs.

After the minimum requirements were completed and we had a functioning ticketing system, we worked on desired requirements. These included:

- As a user, I should be able to select which seat I want in the venue
- As a user, I should be able to print or receive a QR code after I purchase my ticket so that I can present it to get into the event.
- As a user, I want to be able to look at a calendar of events to make it easier to find out which performances I can attend.
- As an admin, I should easily be able to see how many tickets are left.
- As an admin, I should be able to edit the event throughout the sale period so that I can make updates as needed.

Finally, we listed an optional requirement that would possibly be worked on after the completion of the rest of the requirements.

- As a user, I should be able to organize events by type so that I know which shows I would be interested in.

3.2 Wireframes

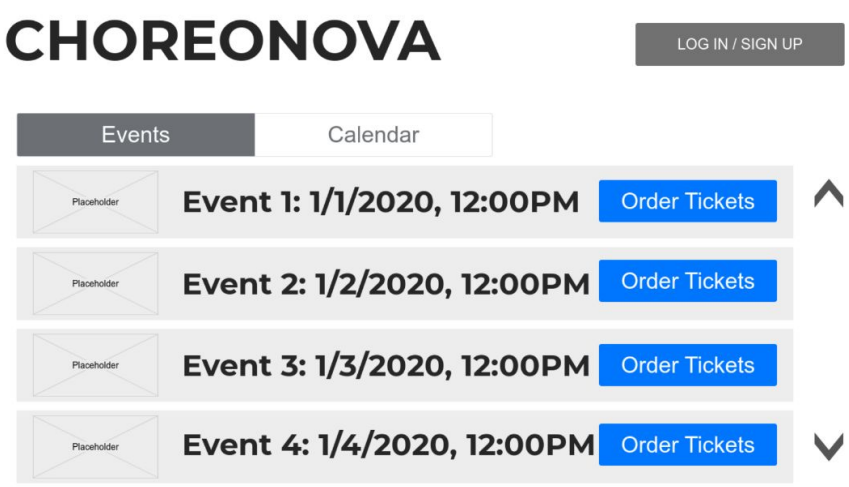

Figure 1. Wireframes (Event / Main page view)

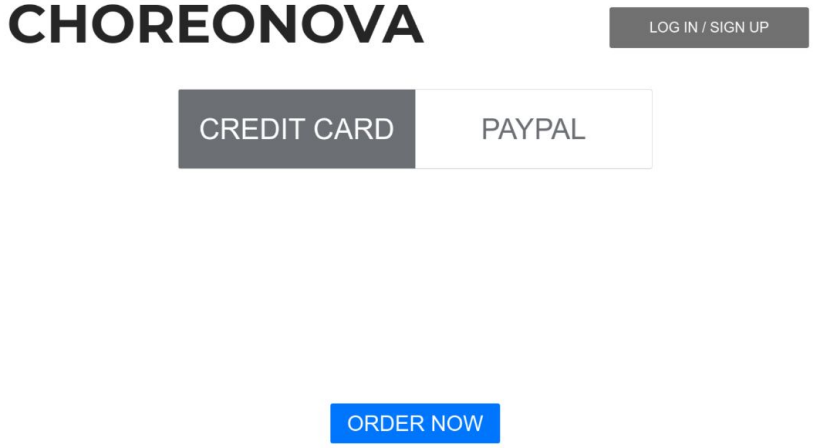

Figure 2. Wireframes (Payment view)

Figure 3. Wireframes (Confirmation view)

Figures 1, 2, and 3 display three of our screens from our wireframe creation. During the planning process for our ticketing system, we developed seven screens to show a general idea of how the system would work and how the screens would transition into each other. While many planning decisions changed from early Fall 2019 to Spring 2020, these were excellent jumping off points for actually creating our design.

The most faithful screen to the final product is Figure 1. This screen shows the main page that is launched when going to http://choreonova.herokuapp.com. We wanted to visualize how events would be displayed, and we felt that a system that showed the events both in an overall list and as a calendar would be the best display options for users.

The other two screens, as seen in Figure 2 and Figure 3, were implemented in the final version as well, albeit with a few differences. Figure 2 shows the original planned payment view, with an option between PayPal and credit card. In the final version, we decided to only give the

option to use PayPal for the customer's simplicity and security's sake. Figure 3 shows the confirmation screen.

In general, these wireframes purposefully lacked detail to give us more freedom. Wireframes allowed us to design the product from a high level while still restricting us to try to follow the transitions.

3.3 Sample Code

```python
#model that distingushes event with multiple dates/showing
class EventDate(models.Model):
    event_time = models.DateTimeField(default=datetime.now, blank=True)
    event_name = models.CharField(max_length=200, default="")
    event_timezone = models.CharField(max_length=5, default="EST")
    tickets_left = models.IntegerField(default = 0)
    event_location = models.CharField(max_length=200, default="")

    def __str__(self):
        return self.event_time.strftime("%d-%b-%Y (%H:%M:%S.%f)")


#model for individual events
class Event(models.Model):
    event_name = models.CharField(max_length=200)

    #event types
    P = 'Performances'
    LD = "Lectures/Demos"
    S = "Screenings"
    W = "Workshops"
    O = "Other"
    TYPE_CHOICES=(
        (P, "Performances"),
        (LD, "Lectures/Demos"),
        (S, "Screenings"),
        (W, "Workshops"),
        (O, "Other")
    )
    event_type = models.CharField(
        max_length=100,
        choices=TYPE_CHOICES,
        default=P,
    )
    description = models.CharField(max_length=1000, default="")
    price = models.IntegerField(default=0)
    #a field that lists all the different scheduled dates associated with this event
    dates = models.ManyToManyField(EventDate)

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

Figure 4. EventDate and Event Models

Figure 4 displays the EventDate and Event model structure that is used to store information about each event. The Event model is the model for the overarching event,

containing information that would be the same for all showings of the same event. The Event model holds a ManyToManyField that can hold all the associated EventDate objects, each representing an individual showing. Fields that are pertinent to each individual showing, such as location and number of tickets, are contained within the EventDate model instead.

Figure 5 displays the Ticket model, which is the model used to store relevant information regarding each batch purchase of tickets. Thus, each Ticket object contains information regarding the event it is for, the time the event is at, seating category, number of tickets associated with the purchase, the price it was bought at, and any special accommodations needed. Finally, there is also a create method that is used to create the Ticket object after payment confirmation, which allows for the Ticket objects to be created without a relevant form.

```python
class Ticket(models.Model):
    #email associated with ticket
    email = models.CharField(max_length=200, default="")
    #number of tickets associated with specific purchase
    number_of_tickets = models.BigIntegerField(default=0)
    #various seating categories
    GA = 'General Admission'
    VIP = "VIP"
    H = "Handicap"
    TYPE_CHOICES=(
        (GA, "General Admission"),
        (VIP, "VIP"),
        (H, "Handicap"),
    )
    seating_category = models.CharField(
        max_length=100,
        choices=TYPE_CHOICES,
        default=GA,
    )
    #special requests
    special_accommodations = models.TextField(max_length=1000, default="", blank=True)
    #event_name to deduct the number of tickets bought
    event_name = models.CharField(max_length=200, default="")
    #date and time of showing
    event_time = models.DateTimeField(default = datetime.today, blank=True)
    #price of ticket passed from event
    ticket_price = models.IntegerField(default=0, blank=True)

    #ticket_id = models.BigIntegerField(default=0)
    #seat if reserved
    #ticket_seat = models.CharField(max_length=200, default="")

    def __str__(self):
        return self.email

    #method to create Ticket object after payment confirmation
    @classmethod
    def create(cls, ids, email, number_of_tix, cat, acc, name, date, price):
        ticket = cls(id = ids, email=email, number_of_tickets=number_of_tix, seating_category=cat, special_accommodations=acc, event_name = name, event_time = date, ticket_price = price)
        return ticket
```

Figure 5. Ticket Model

Figure 6 displays the Seat model, which is the model used to store information about individual seats for different events. Each seat contains information about whether or not the seat

was already reserved, what event the seat is attached to, the name of the seat (including section and number), and other extraneous information about each seat. Generally, many events are seated by first-come first-served, so having a seat object designed for an event-by-event basis is sufficient for the scope of the website.

```
class Seat(models.Model):
    event_name = models.CharField(max_length=200)

    event_time = models.DateTimeField(default = datetime.today, blank=True)

    #When someone searches for a seat, it checks if it's still available
    seat_name = models.CharField(max_length=20)
    seat_special_description = models.TextField(max_length=500, default="")

    seat_number = models.CharField(max_length=20, default = "")
    seat_section = models.CharField(max_length=20, default="")

    seat_claimed = models.BooleanField(default=False)

    seat_category = models.CharField(max_length=200, default="General Admission")
```

Figure 6. Seat model

Figure 7 displays an excerpt of the view used in the page to order tickets. Some of the code included in this excerpt includes code that automatically pulls important data about the event onto the form, which is a PurchaseForm. The excerpt also includes some of the code we wrote to validate certain fields in the forms, namely the amount of tickets customers can purchase and the email. As warning messages, we used the messages features that came with Django and basically had a warning print every time an error was encountered, such as purchasing more tickets than were available or an invalid email. The warning message would be

displayed at the top of the refreshed form, forcing the customer to fix the errors before proceeding. The PurchaseForm class can be seen in Figure 8.

```python
#defines where fields inputted in the PurhcaseForm will go to or displays empty PurchaseForm
def order(request):
    #checks to see if form uses the more secure POST method
    if request.method == 'POST':
        purchaseForm = PurchaseForm(request.POST)

        # add ticket_price to form
        purchaseForm.ticket_price = request.POST['ticket_price']
        #pass event_name to hold the name of the event the ticket purchase is for
        purchaseForm.event_name = request.POST['event_name']
        #pass event_date to know which showing the ticket purchase is for
        purchaseForm.event_time = request.POST['time']
        timestamp = mktime_tz(parsedate_tz(purchaseForm.event_time))
        utc_dt = datetime(1970, 1, 1) + timedelta(seconds=timestamp)
        #find the right showing based off of date
        event = EventDate.objects.get(event_time = utc_dt)

        #checks to see that form is valid and submits the data, redirecting to payment screen
        if purchaseForm.is_valid():
            #do not let anyone purchase a ticket if there are no tickets left
            if event.tickets_left == 0:
                messages.warning(request, f'There are no tickets left for this event. Please try again later')
                return render(request, 'tickets/order.html', {'events': request.GET['event'], 'form': purchaseForm, 'price': request.GET['event_price'], 'time': request.GET['

            #prints an error if the number of tickets ordered is < 1 and refreshes the page
            if int(request.POST['number_of_tickets']) < 1:
                messages.warning(request, f'You must purchase at least 1 ticket!')
                return render(request, 'tickets/order.html', {'events': request.GET['event'], 'form': purchaseForm, 'price': request.GET['event_price'], 'time': request.GET['

            # #prints an error if the number of tickets ordered is > 10 and refreshes the page
            # if int(request.POST['number_of_tickets']) > 10:
            #     return render(request, 'tickets/order.html', {'events': request.GET['event'], 'form': purchaseForm, 'price': request.GET['event_price']})

            #prints an error if the number of tickets ordered is more than what is left and refreshes the page
            if int(request.POST['number_of_tickets']) > event.tickets_left:
                messages.warning(request, f'Please purchase %s or less tickets, that is all we have left!!' % event.tickets_left)
                return render(request, 'tickets/order.html', {'events': request.GET['event'], 'form': purchaseForm, 'price': request.GET['event_price'], 'time': request.GET['

            # email input validation
            try:
                validators.validate_email(request.POST['email'])
            # if error, print warning and refresh page
            except ValidationError:
                messages.warning(request, f'Please enter a valid email!')
```

Figure 7. View to order tickets

```python
#form to purchase tickets stores puchrase information into Tickets model
class PurchaseForm(forms.ModelForm):
    class Meta:
        model = Ticket
        fields = ('email', 'number_of_tickets', 'seating_category', 'special_accommodations', 'ticket_price', 'event_name', 'event_time')
        labels = {
                'number_of_tickets': 'Quantity',
                'special_accommodations': 'Special Accommodation Request'
                }
```

Figure 8. PurchaseForm class

Figures 9 and 10 show snippets of the order_seats view. This view is responsible for handling seat reservation after tickets are ordered and paid for. Due to the fact that this view takes place after payment is complete, having the ability to claim seats on selection would not cause any problems because people would have already paid for these seats even if they reserve them and do not show up. There are several POST options from the order-seats HTML page. The

first is “Reset” (as seen in Figure 10) which will unclaim every seat that users have selected and empty out their list of seats (which is stored as a session variable for each individual user). The option “Remove” removes an individual selected seat from a user’s seat list. The last option is simply ordering the seat itself, which checks to make sure that the user is not going above their number of tickets and also makes sure that the number of seat objects associated with the event has not already reached zero. In Figure 9, it shows the case where the page was accessed not via a POST request (which would be the case when opening the page from payment confirmation), where it sets every variable up for use on the page.

```
    else:
        if int(request.session.get('seats_left')) > 0:
            request.session['seat_list'].append(request.POST['seat_input'])


            #TODO: Change "Dance for Charlottesville" to GET var
            seat_to_change = Seat.objects.get(event_name=request.session.get('event'),
                                              seat_name=request.POST['seat_input'])
            seat_to_change.seat_claimed = True
            seat_to_change.save()

            request.session['seats_left'] = int(request.session.get('ticket_number')) - len(request.session.get('seat_list'))
        #TODO: elif an error message to show up on screen
        return render(request, 'tickets/order_seats.html',
                      {'events': request.session.get('event'), 'price': 3,
                       'num_tickets': request.session.get('seats_left'), 'seat_list':request.session.get('seat_list'), 'rem_seats':remaining_seats})

#When loaded from a non-POST request (aka, when loaded up from payment_done)
else:
    #Check to see if visited_seats exists / is false (which basically means that if it doesn't exist then on first load specify these vars
    if not request.session.get('visited_seats'):
        request.session['seat_list'] = []
        request.session['seats_left'] = request.session.get('ticket_number')

#Generate the seat list
    for seat in Seat.objects.all():
        if seat.event_name == request.session.get('event'):
            if not seat.seat_name in request.session.get('seat_list'):
                if not seat.seat_claimed:
                    remaining_seats.append(seat.seat_name)

    remaining_seats.sort()
```

Figure 9. Order-seats view and process (part 1)

```
def order_seats(request):

    #Create remaining seats list to keep track of what seats for the event are left to purchase
    remaining_seats = []

    if request.method == 'POST':

        request.session['visited_seats'] = True

    #Go through the seats objects to find eligible seats and get rid of seats that are claimed / not from the event
        for seat in Seat.objects.all():
            if seat.event_name == request.session.get('event'):
                if not seat.seat_name in request.session.get('seat_list'):
                    if seat.seat_name != request.POST['seat_input']:
                        if not seat.seat_claimed:
                            remaining_seats.append(seat.seat_name)

    #Sort the seats alphabetically so it doesn't matter when they were added
        remaining_seats.sort()

        #Check if the POST request set "reset" to a, meaning that they want to empty out their reserved seats and unclaim them
        if request.POST['reset'] == 'a':

            #TODO: Unclaim the seats
            for k in request.session['seat_list']:
                seat_to_change = Seat.objects.get(event_name=request.session.get('event'),
                                                  seat_name=k)
                seat_to_change.seat_claimed = False
                seat_to_change.save()
```

Figure 10. Order-seats view and process (part 2)

Figure 11 is an excerpt of the PayPal payment process and what happens when the payment is successful. The top of the excerpt contains the code that sends information about the purchase to PayPal Checkout, which includes the price, an invoice number that is a randomly generated integer to differentiate different purchases, etc. Once the customer is transferred over to PayPal Checkout, there would be two different possible outcomes: one for a successful payment (payment_done) and another for payment failure (payment_cancelled). In Figure 11, payment_done can be seen. The main task that payment_done attempts to achieve is to generate the Ticket object, which cannot be generated earlier with the submission of PurchaseForm in the order stage as at that point, we are unsure of whether or not the customer would go through with buying and reserving the ticket. Thus, all the relevant information from the original

PurchaseForm must be pulled to payment_done where we can finally create the Ticket object using a predefined create method. payment_done is also where we can safely deduct the number of tickets purchased from the associated EventDate object as now we are certain the tickets are now purchased.

```
    order_id = request.session.get('order_id')
    #order = get_object_or_404(Ticket, id=order_id)
    host = request.get_host()

    # retrieves price stored in order view
    price = request.session.get('total')

    paypal_dict = {
        'business': settings.PAYPAL_RECEIVER_EMAIL,
        'amount': price,
        'item_name': "ticket",
        'invoice': str(order_id),
        'currency_code': 'USD',
        'notify_url': 'http://{}{}'.format(host,
                                           reverse('paypal-ipn')),
        'return_url': 'http://{}{}'.format(host,
                                           reverse('payment_done')),
        'cancel_return': 'http://{}{}'.format(host,
                                              reverse('payment_cancelled')),
    }

    form = PayPalPaymentsForm(initial=paypal_dict)
    return render(request, 'tickets/process_payment.html', {'order': order, 'form': form})

 returns once payment successful
csrf_exempt
ef payment_done(request):
    #deduct the number of tickets bought from the event since payment was successful
    eventname = request.session.get('event')
    date = request.session.get('date')
    #find the correct showing based off of date
    timestamp = mktime_tz(parsedate_tz(date))
    utc_dt = datetime(1970, 1, 1) + timedelta(seconds=timestamp)
    event = EventDate.objects.get(event_time = utc_dt)
    tickets = request.session.get('ticket_number')
    event.tickets_left = event.tickets_left - int(tickets)
    event.save()
    # retrieves price stored in order view
    price = request.session.get('total')
    #retrieve ticket id stored from above
    order_id = request.session.get('order_id')
    #restore purchase email
    email = request.session.get('email')
    #restore purchase seating category
    cat = request.session.get('cat')
    #restore special accomodations from purchase
    acc = request.session.get('acc')
    #create the ticket to database now that payment is confirmed
    ticket = Ticket.create(order_id, email, tickets, cat, acc, eventname, utc_dt, price)
    ticket.save()
```

Figure 11. Payment view and process

3.4 Sample Tests

Throughout the entire process of developing the web application, we wrote tests to ensure the features we were creating worked as intended. Testing is important because it allows us to catch any bugs in the code and fix them before the final application is deployed. During every sprint, we have created unit tests that correspond to the features we chose to develop.

Figure 12 shows a basic unit test for the views that we created through Django. This test checks to see if the view created redirects to the correct page that we wanted.

```
def test_index(self):
    resolver = resolve('/order/')
    self.assertEqual(resolver.view_name, 'order')
```

Figure 12. View Test

Figure 13 is a test for the Seat model that is used in our web application. This test creates a Seat object, saves it, then checks the fields inside of the Seat created to see if it was correctly saved.

```
def test_special(self):
    s = Seat(event_name="Test", seat_name="G9", seat_special_description="Only available for people over 60 years old")
    s.save()
    self.assertEqual(s.seat_special_description, "Only available for people over 60 years old")
```

Figure 13. Seat Model Test

Figure 14 tests the form we created through Django. This test creates an array of all the fields the form is asking for and then inputs it into the Django form. It then checks if the form is valid or not, based on the information the form contains.

```
class FormsTest(TestCase):
    # checks if regular form is valid
    def test_forms(self):
        form_data = {'email': 'john.doe@gmail.com', 'number_of_tickets':'5', 'seating_category': 'General Admission',
                     'special_accommodations':'hi', 'ticket_price': 20, 'event_name':'hi', 'event_time':"2003-02-13 04:04"}
        form = PurchaseForm(data=form_data)
        self.assertTrue(form.is_valid())
```

Figure 14. Form Test

Figure 15 is a test for the EventDate model. It creates an EventDate object with a missing attribute then saves it. Since the model requires a time field, Django throws an exception when trying to save the object. The test checks to see if an exception was thrown.

```
def BadTest(self):
    e = EventDate(event_time="", event_name = "Vivian's Dance", event_timezone = "PST", tickets_left=50,
                  event_location = "Sprint Pavilion")
    e.save()
    self.assertRaise(Exception)
```

Figure 15. EventDate Model Test

Figure 16 is a test for the Event model. Since the Event model contains a ManyToMany field of EventDates, this test creates an Event as well as a corresponding EventDate, then checks to see if the EventDate created is a part of the Event object.

```
def TestEventDateFields(cls):
    e = Event(event_name="Cool", event_type="Performances")
    e.save()
    ed = e.EventDate.create(event_time="2003-02-13 04:04", event_name = "Cool", event_timezone = "PST",
                            tickets_left=50, event_location = "Sprint Pavilion")
    ed.save()
    self.assertEqual(e.EventDate.get(pk=ed.pk), ed)
```

Figure 16. Event Test

3.5 Code Coverage

Since our project was built using the Django framework, which is Python based, we chose to use the standard coverage package that comes with Python, Coverage.py. To use this

package, first install coverage using the command “pip install coverage”. After that, there are a couple of configurations that can be added to the file “.coveragerc” to customize the generated report. For a Django project, make sure the source is set to be the entire Django project directory. To see the coverage report, simply run “coverage report” or “coverage html” to see a nicer HTML format of the coverage report. The statistics from the coverage report can be seen in Figure 17.

Coverage report: 83%

Module ↓	statements	missing	excluded	coverage
choreonova/__init__.py	0	0	0	100%
choreonova/settings.py	25	2	0	92%
choreonova/urls.py	4	0	0	100%
tests.py	304	12	0	96%
tickets/__init__.py	0	0	0	100%
tickets/admin.py	8	0	0	100%
tickets/apps.py	3	0	0	100%
tickets/forms.py	9	0	0	100%
tickets/migrations/0001_initial.py	5	0	0	100%
tickets/migrations/0002_event_event_location.py	4	0	0	100%
tickets/migrations/0003_auto_20191106_1107.py	4	0	0	100%
tickets/migrations/0004_event_event_type.py	4	0	0	100%
tickets/migrations/0005_auto_20191109_0128.py	4	0	0	100%
tickets/migrations/0006_auto_20191109_0131.py	4	0	0	100%
tickets/migrations/0007_auto_20191109_2049.py	4	0	0	100%
tickets/migrations/0008_seat_seat_category.py	4	0	0	100%
tickets/migrations/0009_auto_20191122_2358.py	4	0	0	100%
tickets/migrations/0010_auto_20191124_0039.py	4	0	0	100%
tickets/migrations/0011_auto_20191124_0108.py	4	0	0	100%
tickets/migrations/0012_seat_seat_claimed.py	4	0	0	100%
tickets/migrations/0013_auto_20191124_2307.py	4	0	0	100%
tickets/migrations/0014_auto_20191125_0013.py	4	0	0	100%
tickets/migrations/0015_auto_20191208_2305.py	4	0	0	100%
tickets/migrations/0016_auto_20191208_2306.py	4	0	0	100%
tickets/migrations/0017_auto_20191209_0001.py	4	0	0	100%
tickets/migrations/0017_ticket_event_name.py	4	0	0	100%
tickets/migrations/0018_merge_20191209_0116.py	4	0	0	100%
tickets/migrations/0019_auto_20191209_1417.py	4	0	0	100%
tickets/migrations/__init__.py	0	0	0	100%
tickets/models.py	66	3	0	95%
tickets/tests.py	1	0	0	100%
tickets/views.py	137	92	0	33%
Total	**638**	**109**	**0**	**83%**

Figure 17. Coverage Report

3.6 Installation Instructions

These installation instructions are meant to install the product from scratch on the Heroku hosting service. It grabs the code from a Github repository then deploys it to Heroku. The first step is to navigate to signup.heroku.com. Create a new account and authenticate the account by going to the email and clicking the confirmation link that Heroku sent.

The application's code is currently in a private repository on Github. To have access to this code, a Github account will have to be made or active, and the admin of the repository will need the account information to grant permission. If access is not granted to https://github.com/uva-cp-1920/choreonova, the project will be inaccessible. If a new Github account is needed, it can be created at https://github.com/join?source=header-home.

To deploy a Heroku app, Git and Heroku CLI must be installed. Depending on what type of Operating System the host computer has, these installation instructions will be slightly different. To install Git, navigate to https://git-scm.com/book/en/v2/Getting-Started-Installing-Git and follow the installation instructions for the proper OS. To install Heroku CLI, navigate to https://devcenter.heroku.com/articles/heroku-cli.

At this point, open the terminal or command prompt and enter the following command:

```
git clone https://github.com/uva-cp-1920/choreonova.git
```

A copy of the repository should be generated and can be accessed using:

```
cd choreonova
```

The repository has all necessary files to deploy to Heroku, so all that is needed now is to create the Heroku app. This can be done by entering:

```
heroku create
```

After this, push this repository to Heroku. This is done by entering the following command:

```
git push heroku master:master
```

After this process is complete, the website is accessible by going to the Heroku app and clicking “Open App”, or, alternatively, from the command line by entering:

```
heroku open
```

If the server requires more resources, a Dyno can be added by navigating to the app on heroku.com, going to the “Resources” tab, and then clicking “Change Dyno Type”. Otherwise, the project will run on the free tier of Heroku hosting.

4. Results

The ticketing system that we created fulfills the initial requirements received by the customer. The main issue of ChoreoNova's previous method of selling tickets was that it was on an external website. The ticketing system that we developed allows customers to go through and purchase a ticket to their desired event using around six mouse clicks, whereas their original system of purchasing a ticket required way more mouse clicks, as users would have to search for the EventBrite link and go through that entirely separate system.

To use the ticketing system we created, customers will click on the link listed on ChoreoNova's main webpage, bringing them to the ChoreoNova ticketing website. From there, they will be able to browse through the events that the company is offering. Once they have found an event they would like to purchase or reserve a ticket for, the customer can hit a button for that event, bringing them to the next page where they can enter their information, such as e-mail, quantity of tickets, and any special accommodation requests. After filling in the necessary information, the customer is then directed to reserve their seats for the event. Finally, they are directed to the PayPal payment page. When the payment has gone through as successful, the customer is redirected to the ChoreoNova ticketing web page, where they are shown a confirmation page with their ticket and a demographics collection screen. Additionally, an email containing a QR code and other information about their ticket will be sent to the customer's provided email address.

The other stakeholders of the system, the ChoreoNova staff, can access the backend using an admin account. In the backend, they will be able to create new events that will be displayed on the main page for users to purchase tickets. Administrators can also create new seat objects,

which will be displayed when the user reserves seats during the purchasing process. They will be able to look through the record of all tickets that have been purchased, see demographic information collected, as well as see any special accommodation requests so they can adjust the venue as necessary.

5. Conclusions

There is a lot the development team can take away from this project. The most important takeaway is the importance of effective communication. Early on, we had some trouble communicating our ideas with our customer. It can be a bit of a challenge to explain extremely technical topics to a customer who has very limited technical knowledge. Moreover, many times, the customer tended to overestimate our abilities and asked for a lot more than what is possible with only four undergraduate students. These are very common issues that will arise in the software engineering industry. Many people tend to not see the blood and sweat that goes into building a software as they only see the final product. They may feel that a software is magically built, and as such, software engineers can easily build something in a short amount of time, which is definitely not the case for anyone who has ever attempted coding. Ultimately, we resolved this issue by sitting down with our customer and explaining the situation to her carefully, ensuring that she understands clearly where we are coming from.

Working on this project has shown us the importance of e-commerce, especially as it pertains to smaller businesses. With the limited funds that many small businesses or arts centers have, it can be very difficult for them to afford some of the software systems offered out there by the bigger companies such as Amazon or Google. By providing free coding services to many of these smaller, non-profit businesses, we can help give them a leg up in survival and keeping up with the modern times. Doing so allows for a further reach of the arts that ChoreoNova wants to provide, letting them share their hard work and mission to more people. Digitalizing also allows them to be able to continue to share their art well into the future, helping them become more self-sufficient by having their own ticketing system. The arts are an important facet to human

expression and changes with societal moods. We are all very proud to have been able to contribute to sharing the arts with more people, in Charlottesville and beyond, one ticket at a time.

6. Future Work

Due to a lack of time, our ticketing system, while complete and functional, has some potential components that can be added in the future if given more time, resources, and funding. The following features and ideas can be implemented in the future:

1. Having a fully functioning user login and signup system was a planned feature from the original semester: As we got closer to the end of the project, we came to the conclusion that the feature was too far out of the pipeline of the functionality and was deemed not feasible for the scale of this application. A login / signup system could be implemented if given more time and resources, however, and the system could keep track of user statistics, ticket history, and other demographic information that is currently stored in its own simple model that ChoreoNova could use as part of their show planning strategy.
2. Having a more robust calendar system that actually had a month-by-month calendar in a traditional design: The primary reason why this did not come to be was because of the fact that the number of shows done per year by ChoreoNova, as of Spring 2020, was not enough to reasonably need a detailed calendar system without it looking mostly empty. In the future if ChoreoNova does expand to more than 3-5 shows per year, having an HTML style calendar could be a very aesthetic feature to try to implement.
3. Developing a more convenient way to add seats to the database: The scope of what functionality seat reservation actually needed was fairly small in comparison to the project as a whole. Unfortunately, as a result, actually changing the administrative page to have a faster way to add seat objects, as well as providing a way to directly upload theater images to the front end of the site, were not tasks that we were able to complete

within the group size and time allotted. However, since our client claimed that seat reservation was situational (depending on the event), we felt that our current system was sufficient.

The items described above are simply optional features that we felt could be implemented in the future. Given our experience with customer management and product development, our team would be equipped to handle these features if given a theoretical opportunity to reset the development process.

In the future, because of these experiences, we as a group feel confident that we can tackle larger projects and communicate more efficiently and effectively with our client. Our client meetings, especially in the later sprints, proved to have problems with actually having our client keep a more focused scope for her project, and we had to learn how to negotiate on what features were the most important for a project of her size. Even though we have performed to the best of our current abilities given the circumstances, we each individually feel more able to better manage these types of problems in the future.

7. References

Adam Johnson. (n.d.). Retrieved from
https://adamj.eu/tech/2019/04/30/getting-a-django-application-to-100-percent-coverage/

Coverage.py. (n.d.). Retrieved from https://coverage.readthedocs.io/en/latest/

EventBrite (n.d.). Retrieved from
https://www.eventbrite.com/blog/academy/event-data-analytics/

TicketMaster (n.d.). Retrieved from
http://pages.tmclient.ticketmaster.com/page.aspx?QS=5c591a8916642e73763a937a7205ee1dcdc8ffd94f7dc4333a1af9702427839a