CS4227 - Software Design and Architecture

Group Project -

Food Ordering System

Submitted by

Team Orange

16181492	Ciaran Carroll
18111521	Naichuan Zhang
16192206	Neale Conway
15179486	Ronan Barry

BSc in Computer Systems

University of Limerick

March 2020

Table of Contents

Requirements

Scenario

Our project idea is to make a just eat inspired web application using the Django framework. The system will both be customer and admin faced, there'll be a user log in, where he/she may view profile, create an order, add to check-out and make purchase. On the admin side the user can make changes to the menu, add stock, view deliveries and view customers. We will enforce these features in a user friendly, easy to use way.

Our goal is to create this website to include all intended functional requirements as well as reach our performance attributes such as security, extensibility, scalability and usability. We believe these non-functional requirements in particular are especially important in the development of a web application. In order to achieve these requirements we will take advantage of Django's support of the MVC programming paradigm, django's robust security features, it's ability to easily be extended and scaled and its ability to accelerate custom web application development.

Functional Requirements

Here are some functional requirements that might be applied to the food ordering system, but not limited to these:

A Customer can:

1. register (with username, email address, password, phone number, profile photo, etc.)
2. login and logout from the system
3. change the password
4. view profiles (profile photo, username, etc.)
5. change the profile photo
6. add/edit shipping address
7. view food items as a guest without login to the system

8. order food (by adding food items to online shopping cart)
9. cancel an order before payment (with an explanation why)
10. search available food items
11. make a payment after selecting a payment method
12. view order history
13. comment/rate after payment
14. report to admin
15. contact a member of the admin

An Admin can:

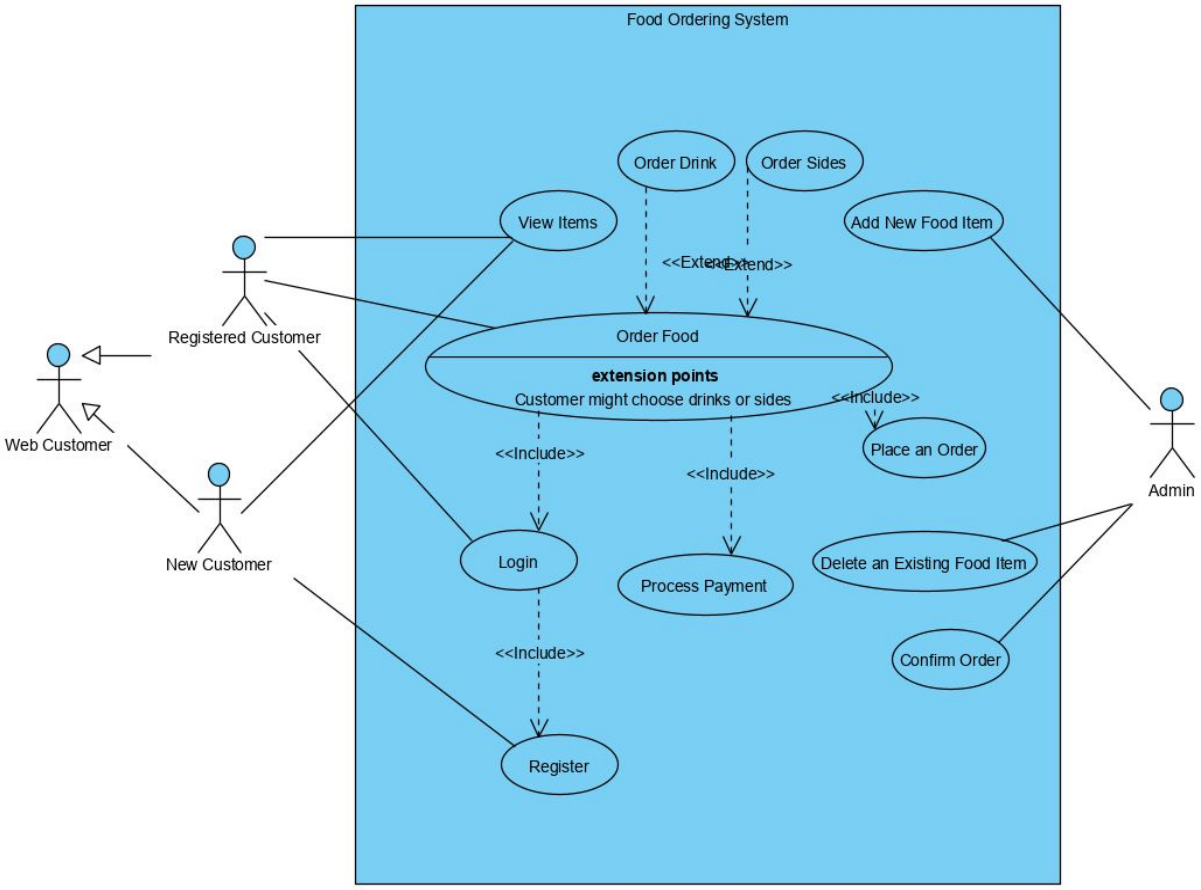
1. login and logout
2. change password
3. manage system users (delete, ban, etc.)
4. change customer's rate/comment (delete)
5. confirm an order
6. process refund (validate)
7. add new food items (price, availability, food type, etc.)
8. update existing food items (price, availability, etc.)
9. remove existing food items/types

The System should be able to:

1. record registered users with their details: the password should be encrypted to the system
2. record orders
3. record payments
4. check if the username has been registered when registering a new account
5. send an email to the user's registered mailbox, and allow the user clicking the activation link in the email to complete the account activation, the email address cannot be used twice
6. request an explanation when a customer cancels an order or payment
7. automatically cancel the order if the customer doesn't make a payment within 24 hours after placing an order
8. add loyalty points (+10) every time when a customer completes a whole ordering process, corresponding discount policy will be applied
    a. when loyalty points <= 50, "Copper" (Level 1), no discount
    b. when loyalty points > 50 and <= 100, "Silver" (Level 2), 10% discount
    c. when loyalty points > 100 and <= 150, "Gold" (Level 3), 20% discount
    d. when loyalty points > 150, "Platinum" (Level 5), 30% discount
9. display a list of different food types, each with different food items
10. display details of a food type, including an image if exists
11. display details of a food item (price, stock, image, etc.)
12. display customers' rate and/or comment

13. display customer’s order information (receipt)

Use Case Diagram



Use Case Description(Detailed)

Use Case	Generate an Order
Goal in Context	Customer selects a type of food
Scope & Level	System
Preconditions	1. User is logged in

	2. The order contains at least one food item 3. The system has sufficient stock of selected items
Postconditions	1. Stock of the order item is decremented from database 2. Payment has to be done within 24 hours 3. The order is added to the user's personal info
Success End Conditions	User receives order, we have processed the order and waiting for payment to be finished
Failed End Conditions	1. The order is not sent out successfully 2. The user cancels the order
Primary, Secondary Actors	User, Admin
Trigger	User initiates an order online
Description	1. User opens the website 2. User logs in 3. User selects food types 4. User selects sides 5. User selects drinks 6. User views and checks the items in order 7. User confirms the order
Extensions	3a. Selected food type is unavailable, a message will be sent 4a. Selected sides are unavailable, a message will be sent 5a. Selected drink is unavailable, a message will be sent 7a. User removes the items and cancels the order
Priority	Top
Variations	None
Open Issues	What if we only have part of the order?
Due Date	Release 1.0

Use Case	Add a Food Item
Goal in Context	The admin introduces a new food item under an existing food type, along with its details into the system
Scope & Level	System

Preconditions	1. The User is logged in as an admin 2. The food item does not exist in the database
Postconditions	1. New food item is added to database with its details (price, image, stock, etc.)
Success End Conditions	Admin has added the new food item into the database with its details. Users are available to view the food item's details by default
Failed End Conditions	New food item is not added to the database. User cannot view this food item on the webpage
Primary, Secondary Actors	Admin, User
Trigger	Admin tries to add a new food item
Description	1. The user is logged in as an admin 2. System shows a screen to allow the admin add a food item 3. Admin enters details of the food item 4. Admin confirms the details 5. The new food item is added to database
Extensions	5a. The food item failed to be added to database, an error message will be sent out to the admin, ask the admin to retry
Priority	Top
Variations	None
Open Issues	None
Due Date	Release 1.0

Use Case Descriptions

Register

Actor Action	**System Response**
1. User selects register	2. System moves to register form
3. User enters new account details	4. system checks for valid data and saves account to database
Alternative Course	
4a. User enters incorrect details and is shown an invalid details message. 4b. User is asked to enter details again	

Login

Actor Action	**System Response**
1. User selects login	2. System moves to login form
3. User enters account details	4. system checks for corrects password and account info
	5. User is logged in
Alternative Course	
4a. User enters incorrect details and is shown an invalid details message. 4b. User is asked to enter details again	

Add New Menu Item

Actor Action	**System Response**
1. User selects AddMenuItem, must be an admin account	2. System moves to add menu item form
3. User enters new menu item details	4. system checks for valid input
	5. system saves new menu item to database
Alternative Course	
4a. User enters incorrect details and is shown an invalid details message.	

View Previous Orders

Actor Action	**System Response**
1. User selects view previous orders	2. System moves to previous orders form
	3. system checks database for previous orders linked to currently logged in account
	4. system displays data to user
Alternative Course	
4a. User may not have any previous orders	

Check Cart

Actor Action	**System Response**
1. User selects Check Cart button	2. System moves to cart form
	3.system displays current order within cart
Alternative Course	
4a. Cart may be empty	

Non-Functional Requirements

Usability

Usability is a key NFR for any website. A user must be able to easily navigate and easily perform the key goals of the website. The user must also be able to leave the website for a period of time and readily be able resume activity. The website must also be enjoyable to re-attract users and not contain bugs.

Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system. If we want our company to be ambitious and achieve growth, we must ensure our system is able to handle any increase in work and be comfortable accommodating growth as increased popularity of the application brings a higher workload.

Security

Web application security is crucial because attacks against internet-exposed web apps are the top cause of data breaches. To ensure we or our customers don’t suffer from malicious attacks we must ensure our site is adequately secure. Sites with payment features are some of the most highly targeted applications for hackers to attack, due to the possible accessibility of users' payment details.

Extensibility

Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension. Over time we may want to add additional features to our system. It is important that the addition of new features does not affect the already implemented features causing additional time and therefore cost.

Tactics to Support Quality Attributes

Usability

We will ensure application usability is at a sufficient level by performing frequent ad hoc testing on the application. This will ensure that any unwanted bugs will be found. We will perform ad hoc testing on other users' areas of the application to test the ease of use of other developers' work.

Scalability

To account for this during implementation we will ensure proper use of collections. Collections in Python are containers that are used to store collections of data, for example, list, dict, set, tuple etc. These collections are highly valuable in that they are highly scalable. We will also follow the Django framework as MVT promotes loose coupling.

Security

We will avail of the many security features Django has. Some of Django's security features include their SQL queries being constructed using query parameterization, this prevents injection attacks. Using Django templates protects you against the majority of XSS attacks. Django also has built-in protection against most types of cross site request forgery attacks.

Extensibility

Ensuring a loosely coupled system promotes extensibility. We should be able to add new features without affecting previously existing features or having to modify existing features. Following the model view template architecture will also ensure will allow the system to be independently extensibility when a

Architectural and Design Patterns

Builder Design Pattern

The Builder Design Pattern is a creational design pattern which can be applied to construct complex objects that require multiple steps of initialization or assignment. It can be used as a proxy to complete the object's build process. It allows the same build process can be used to create multiple different representations and is easy to extend and use with classes. In the project, the Builder is used to facilitate user with different attributes. When a new user is registered, the User may or may not have profile photo at the beginning, but all the users need to have at least username and password, so Builder can make User have different representations with different attributes.

```python
class AbstractUserBuilder(ABC):
    """the builder interface"""
    @abstractmethod
    def set_username(self, username):
        raise NotImplementedError

    @abstractmethod
    def set_password(self, password):
        raise NotImplementedError

    @abstractmethod
    def set_email(self, email):
        raise NotImplementedError

    @abstractmethod
    def set_icon(self, icon):
        raise NotImplementedError

    @abstractmethod
    def save(self):
        raise NotImplementedError

    @abstractmethod
    def reset(self):
        raise NotImplementedError

    @abstractmethod
    def get_user(self):
        raise NotImplementedError
```

```python
class UserBuilder(AbstractUserBuilder, ABC):
    """the concrete builder"""
    def __init__(self):
        super(UserBuilder, self).__init__()
        self.__user = User()

    def set_username(self, username):
        self.__user.username = username
        return self

    def set_password(self, password):
        self.__user.password = password
        return self

    def set_email(self, email):
        self.__user.email = email
        return self

    def set_icon(self, icon):
        self.__user.icon = icon
        return self

    def save(self):
        self.__user.save()
        return self
```

```python
class UserDirector:
    """the director"""
    def __init__(self, builder):
        super(UserDirector, self).__init__()
        self.__builder = builder

    def construct(self, username, password, email, icon):
        return self.__builder.reset()\
            .set_username(username)\
            .set_password(password)\
            .set_email(email)\
            .set_icon(icon)\
            .save().get_user()
```

Command Design Pattern

The Command Design Pattern is a behavioral pattern of the object. It encapsulates a request or operation as an object and allows us to parameterize the client with different requests. In the project, we used the Command design pattern to manage ordering processes, such as creating an order and cancelling an order. We declared a Command interface with an Undoable interface to support the implementation of a concrete command with undo operation. OrderCommand is a concrete command we have in the project, which

implements an undoable operation. When we want to use OrderCommand, order_service() method should be invoked first, a concrete service and an order object are then passed to the OrderCommand as class variables, and the method will automatically determines the currently passed service and concatenates the corresponding Execute and Undo methods for the OrderCommand based on that service, then the execute() method is invoked. In addition, to make it easy to save and manage a series of Command records, a CommandManager helper class has been provided.

```python
class Command(metaclass=ABCMeta):
    """Command Interface"""

    @abstractmethod
    def execute(self):
        raise NotImplementedError("You should implement this.")
```

```python
class Undoable(metaclass=ABCMeta):
    """the interface for defining undoable operation"""

    @abstractmethod
    def undo(self):
        raise NotImplementedError("You should implement this.")
```

```python
class OrderCommand(Command, Undoable):
    """the concrete command"""

    def __init__(self, order: Order, execute: Execute, undo: Undo):
        self.__order = order
        self.__execute = execute
        self.__undo = undo
        self.__orders: List[Order] = list()

    @classmethod
    def order_service(cls, service, order) -> 'OrderCommand':
        if isinstance(service, AbstractOrderOrderService):
            return cls._for_create_order(service, order)
        elif isinstance(service, AbstractCancelOrderService):
            return cls._for_cancel_order(service, order)
        else:
            raise NotImplementedError("No available service type found!")

    @classmethod
    def _for_create_order(cls, service: AbstractOrderOrderService, order: Order) -> 'OrderCommand':
        exec_func: Execute = lambda o: service.create_order(o)
        undo_func: Undo = lambda o: service.undo(o)
        return cls(order, exec_func, undo_func)

    @classmethod
    def _for_cancel_order(cls, service: AbstractCancelOrderService, order: Order) -> 'OrderCommand':
        exec_func: Execute = lambda o: service.cancel_order(o)
        undo_func: Undo = lambda o: service.undo(o)
        return cls(order, exec_func, undo_func)

    def execute(self) -> Order:
        order: Order = self.__execute(self.__order)
        self.__orders.append(order)
        return order

    def undo(self):
        order = self.__orders.pop()
        self.__undo(order)

    def __len__(self):
        return len(self.__orders)
```

```python
class CommandManager:
    """manage commands"""
    def __init__(self):
        self.__undo_commands: List[Union[Command, Undoable]] = list()

    def execute(self, command: Command):
        """execute the command"""
        result = command.execute()
        self.__undo_commands.append(command)
        return result

    def clear(self):
        """clear up all saved commands"""
        self.__undo_commands.clear()

    def undo(self, times: int = 1):
        """undo the previous [times] command(s)"""
        for i in range(times):
            command = self.__undo_commands.pop()
            command.undo()

    @property
    def undo_commands_size(self):
        """get number of undo commands"""
        return len(self.__undo_commands)
```

Memento Design Pattern

The Memento pattern can be used to store the current internal state and restore the internal state to the previous state by saving the state outside of the object. In our project, memento is mainly applied to handle exceptions, that is, backup the necessary data or objects to prevent data loss when an exception occurs. We have developed the Memento class which stores a state, the Originator sees a wide interface, it can access all the necessary data to restore itself to its previous state, while the Caretaker can only pass the memento to other object (Sarcar 2018). In our Memento's implementation, an originator object is passed into caretaker as a parameter so that it can request the memento from originator. At the same time, a memento list is stored in caretaker to store the saved memento, and also for the convenience to roll back to the previous state. The Caretaker class encapsulates two methods, save() and undo(), which can be used to save the state inside a memento and revert back to previous state, respectively.

```python
class Memento:
    """the memento"""

    def __init__(self, state: Any) -> None:
        self.__state = state

    def get_state(self) -> Any:
        """Originator uses this method when restoring its state"""
        return self.__state
```

```python
class Originator:
    """
    the originator:
    sees wide interface - can access all the data necessary to restore itself to its previous state
    """

    def __init__(self, state: Any) -> None:
        self.__state = state

    def save_memento(self) -> Memento:
        """save the current state inside a memento"""
        return Memento(self.__state)

    def restore_memento(self, memento: Memento) -> None:
        """restore the Originator's state from a memento"""
        self.__state = memento.get_state()

    def set_state(self, state: Any) -> None:
        """set state"""
        self.__state = state

    def get_state(self) -> Any:
        """get state"""
        return self.__state
```

```python
class Caretaker:
    """
    the caretaker:
    sees narrow interface to the memento - can only pass the memento to the other objects
    """

    def __init__(self, originator: Originator) -> None:
        """caretaker can request a memento from originator"""
        self.__mementos = []
        self.__originator = originator

    def save(self) -> None:
        """save state to memento and add to list"""
        self.__mementos.append(
            self.__originator.save_memento())

    def undo(self) -> None:
        """revert back to the previous state from list"""
        if not len(self.__mementos):
            return
        memento = self.__mementos.pop()
        try:
            self.__originator.restore_memento(memento)
        except Exception as e:
            print(e)
```

A second memento was added as part of the delivery functionality. The purpose of this memento was to allow a system admin to perform the undo function if the user cancels an order. The delivery state will then revert back to the state it was previously in.

Factory Design Pattern

Factory Method is a Creational Design Pattern that allows an interface or a class to create an object, but let subclasses decide which class or object to instantiate. Using the Factory method, we have the best ways to create an object. Here, objects are created without exposing the logic to the client and for creating the new type of the object, the client uses the same common interface.

Strategy Design Pattern

The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them. In this application, the strategy method is used as a way to make payments. The user selects either paypal or card and the payment detail are saved to the corresponding model.

Composite Design Pattern

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task. The Composite was used in our project to allow us to treat all items in an order as a single instance which we could use to get a total price or to create OrderItem Objects in our database.

```python
class ItemComponent(ABC):
    """Composite Interface"""

    @abstractmethod
    def get_price(self):
        raise NotImplementedError("You should implem

    @abstractmethod
    def create_order_item(self, order: Order):
        raise NotImplementedError("You should implem
```

```python
class ItemComposite(ItemComponent):
    """Composite"""

    def __init__(self):
        self._items: List[ItemComponent] = list()

    def get_price(self):
        price: Decimal = Decimal('0.0')
        for item in self._items:
            price += item.get_price()
        return price

    def create_order_item(self, order: Order):
        for item in self._items:
            item.create_order_item(order)

    def add(self, item: ItemComponent):
        self._items.append(item)

    def remove(self, item: ItemComponent):
        self._items.remove(item)
```

```python
class ItemLeaf(ItemComponent):
    """Composite Leaf"""

    def __init__(self, item: Item, amount: int):
        self._item = item
        self._amount = amount

    def get_price(self):
        return self._item.price * self._amount

    def create_order_item(self, order: Order):
        order_item = OrderItem(item=self._item, order=order, amount=self._amount)
        order_item.save()
```

Visitor Design Pattern

Visitor design pattern is one of the behavioral design patterns. It is used when we have to perform an operation on a group of similar kinds of Objects. With the help of the visitor pattern, we can move the operational logic from the objects to another class. Our visitor pattern allowed us to create different kinds of visitors which could return the fields of

Visitable models in different formats such as string or json. The singledispatch functool is used on the visit method which creates a generic function which can be overridden by different methods which take different model Objects as a parameter. The register method attaches these methods to the visit function which allows us to hide everything except the generic visit method from the different models.

```python
class Visitable(Model):
    @abstractmethod
    def accept(self, visitor: AbstractVisitor):
        raise NotImplementedError

    class Meta:
        abstract = True
```

```python
class AbstractVisitor(ABC):
    def __init__(self) -> None:
        super().__init__()

        from order.models import Order, Cart, Item

        self.visit = singledispatch(self.visit)
        self.visit.register(Order, self._visit_order)
        self.visit.register(Cart, self._visit_cart)
        self.visit.register(Item, self._visit_item)

    @abstractmethod
    def visit(self, element: Visitable):
        raise NotImplementedError

    @abstractmethod
    def _visit_order(self, element: Order):
        raise NotImplementedError
```

```python
class Cart(Visitable):
    id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    item = models.ForeignKey(Item, on_delete=models.CASCADE, null=True)
    num = models.IntegerField(default=1)
    is_selected = models.BooleanField(default=True)

    def accept(self, visitor):
        return visitor.visit(self)
```

```python
class StringVisitor(AbstractVisitor):
    def __init__(self) -> None:
        super().__init__()

    def visit(self, element: Visitable):
        super().visit(element)

    def _visit_order(self, element: Order):
        string = "Order ID: " + str(element.id) + " / "
        string += "User ID: " + str(element.user.id) + " / "
        string += "Total Price:  " + str(element.total_price) + " / "
        string += "Timestamp: " + str(element.timestamp) + " / "
        string += "State: " + str(element.state)
        return string
```

REST Architectural Pattern

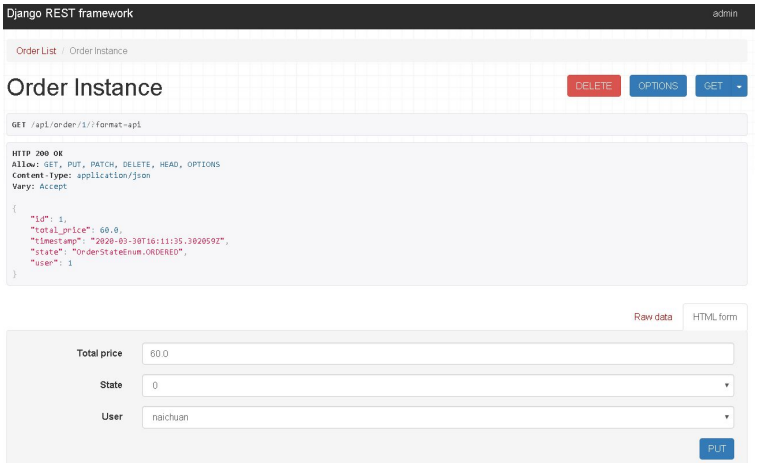
Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data. In order for an API to be RESTful, it has to follow a set of constraints. The API must have a Uniform interface, Client — server separation, be Stateless, a Layered system, be Cacheable, Code-on-demand.

The REST architectural pattern was implemented by integrating the Django REST framework which includes Serialization that supports both ORM (Django Models) and non-ORM data sources along with ViewSets and Routers. Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. A ViewSet class is simply a type of class-based View, that does not provide any method handlers such as .get() or .post(), and instead provides actions such as .list() and .create(). REST framework adds support for automatic URL routing to Django, and provides you with a simple, quick and consistent way of wiring your view logic to a set of URLs. The Framework provides both JSON and HTML interfaces for API access.

```python
# Convert complex data (e.g. Models) into Python datatypes
class OrderSerializer(serializers.ModelSerializer):
    class Meta:
        model = Order
        fields = '__all__'
```

```python
# Register views in ViewSet
router = routers.DefaultRouter()
router.register('api/order', OrderViewSet, 'order')
router.register('api/cart', CartViewSet, 'cart')
router.register('api/item', ItemViewSet, 'item')
router.register('api/orderitem', OrderItemViewSet, 'orderitem')

urlpatterns = router.urls
```

```python
# Controller for request
class OrderViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    permission_classes = [
        permissions.IsAuthenticatedOrReadOnly,
    ]
    serializer_class = OrderSerializer
```



Interceptor Architectural Pattern

The Interceptor Architectural Pattern is based on creating interception points during the standard execution of a project. It allows the framework to be extended transparently and easily define additional services which were not made available during the original development time. The main components of this pattern are the Context , Framework , Dispatcher , Interceptor. During execution events will occur and context objects which can be intercepted. When an event occurs, the framework will notify its dispatchers of the event. In turn, the dispatchers will notify their registered interceptors.

When creating and passing a context object there are two ways you can do it. The per event interception means that a new context object will be created per event, but this however has higher overheads due to repeated creation and deletion of the object. Per registration means that instead of a new context object being created each time, one will be assigned during an interceptor's registration and updated during the application's execution. We have used a per-event context creation approach in our project even though there are downsides to this method.

```python
from .food_dispatcher import *

class InterceptorFramework:

    foodDispatcher = food_dispatcher()

    def __init__(self):
        self.foodDispatcher = food_dispatcher()

    def register_food_interceptor(self,interceptor):

        return self.foodDispatcher.register_food_interceptor(interceptor)

    def remove_food_interceptor(self,interceptor):

        return self.foodDispatcher.remove_food_interceptor(interceptor)

    def on_log_event(self,context):

        self.foodDispatcher.on_log_event(context)
```

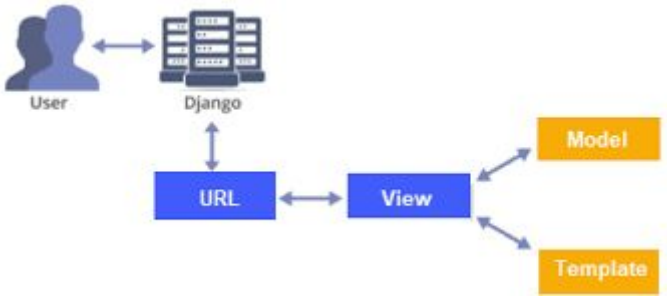
```python
from .Context import *

class food_dispatcher:

    interceptor_list = []

    def register_food_interceptor(self,interceptor):

        return self.interceptor_list.append(interceptor)

    def remove_food_interceptor(self,interceptor):

        return self.interceptor_list.remove(interceptor)

    def on_log_event(self,context):

        for x in self.interceptor_list:
            x.on_log_event(context)
```

```python
from .Context import *
from logger import *
import re

class food_interceptor:

    def on_log_event(self,context):

        pattern = re.compile(r'[a-zA-z]\s+')
        data = context.get_name()

        if(re.search(pattern, data)):
            nothing = None
        else:
            logging.info('Invalid format for new menu item entered : ' + data)
```

```python
class Context(object):
    name = ""
    type = ""
    stock = ""
    price = ""

    def get_name(self):
        return self.name

    def get_type(self):
        return self.type

    def get_stock(self):
        return self.stock

    def get_price(self):
        return self.price

class NewMenuItemContext(Context):

    name = ""
    type = ""
    stock = ""
    price = ""

    def __init__(self,name,type,stock,price):

        self.name = name
        self.type = type
```

System Architecture

Django is based on MVT (Model-View-Template) architecture. MVT is an architectural pattern for developing a web application. MVT Structure has the following three parts. The Model is going to act as the interface of your data. It is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database. The View is the user interface — what you see in your browser when you render a website. It is represented by HTML/CSS/Javascript files. A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

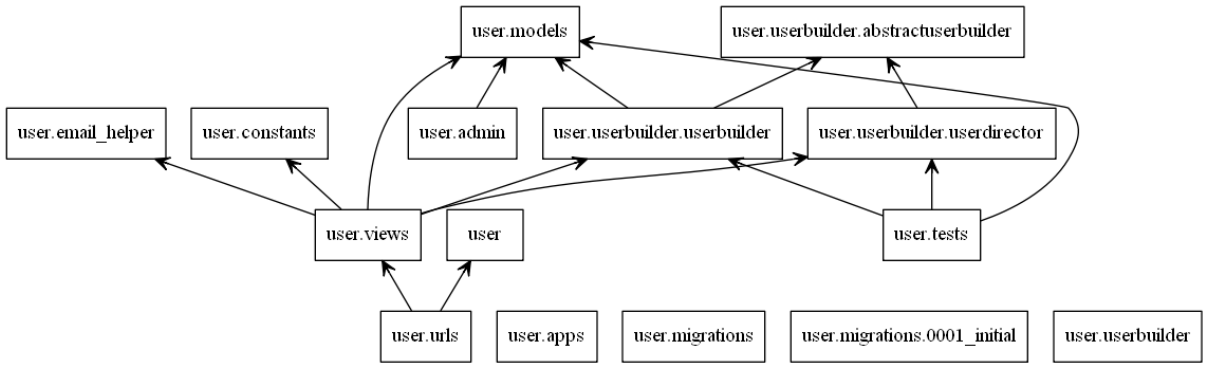


How the architecture works is as follows. A user requests a resource from Django. Django works as a controller and checks the available resources in the urls.py file. If the URL maps to a resource, a view is called that interacts with the model and template. Django then sends the template back to the user as a response.

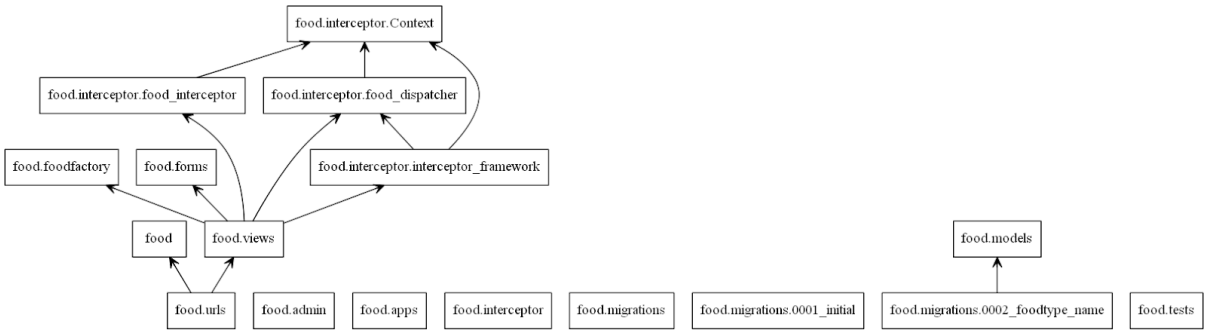
Structural and Behavioural Diagrams

Package Diagrams

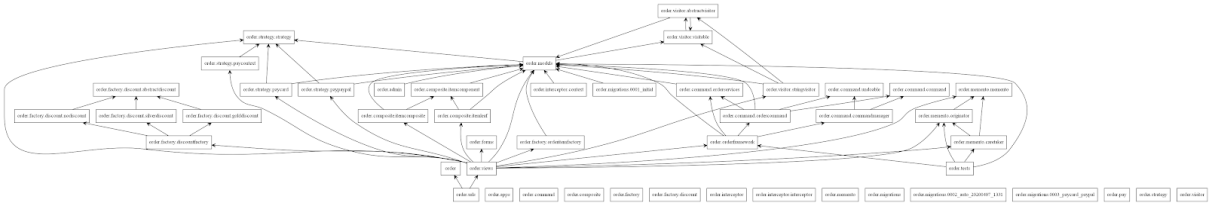
User



Food

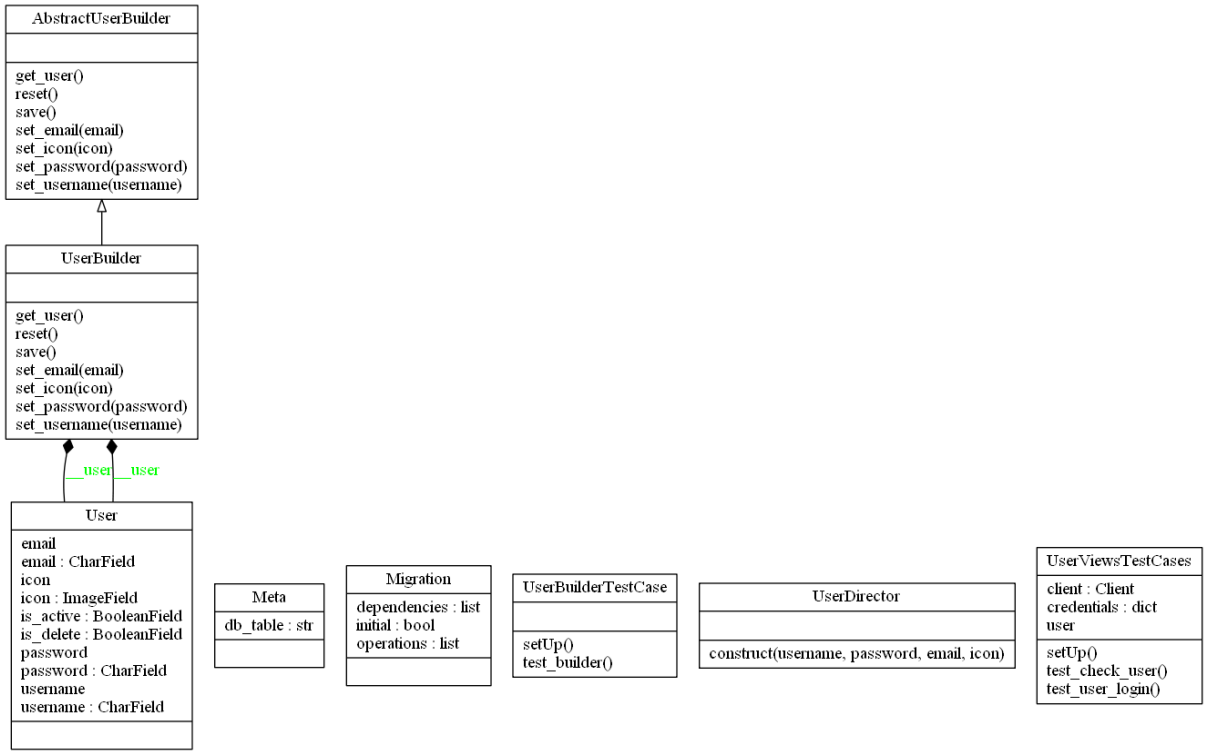


Order

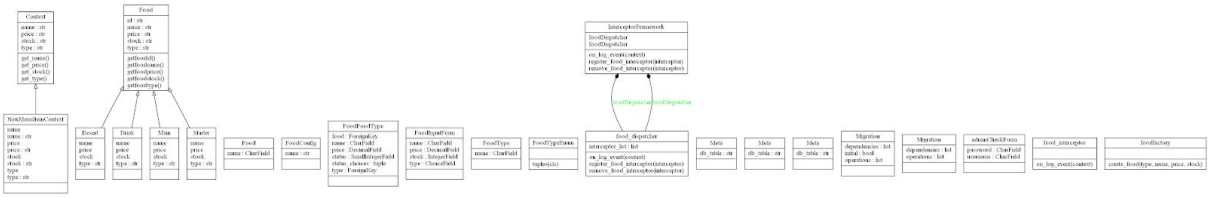


Class Diagrams

User



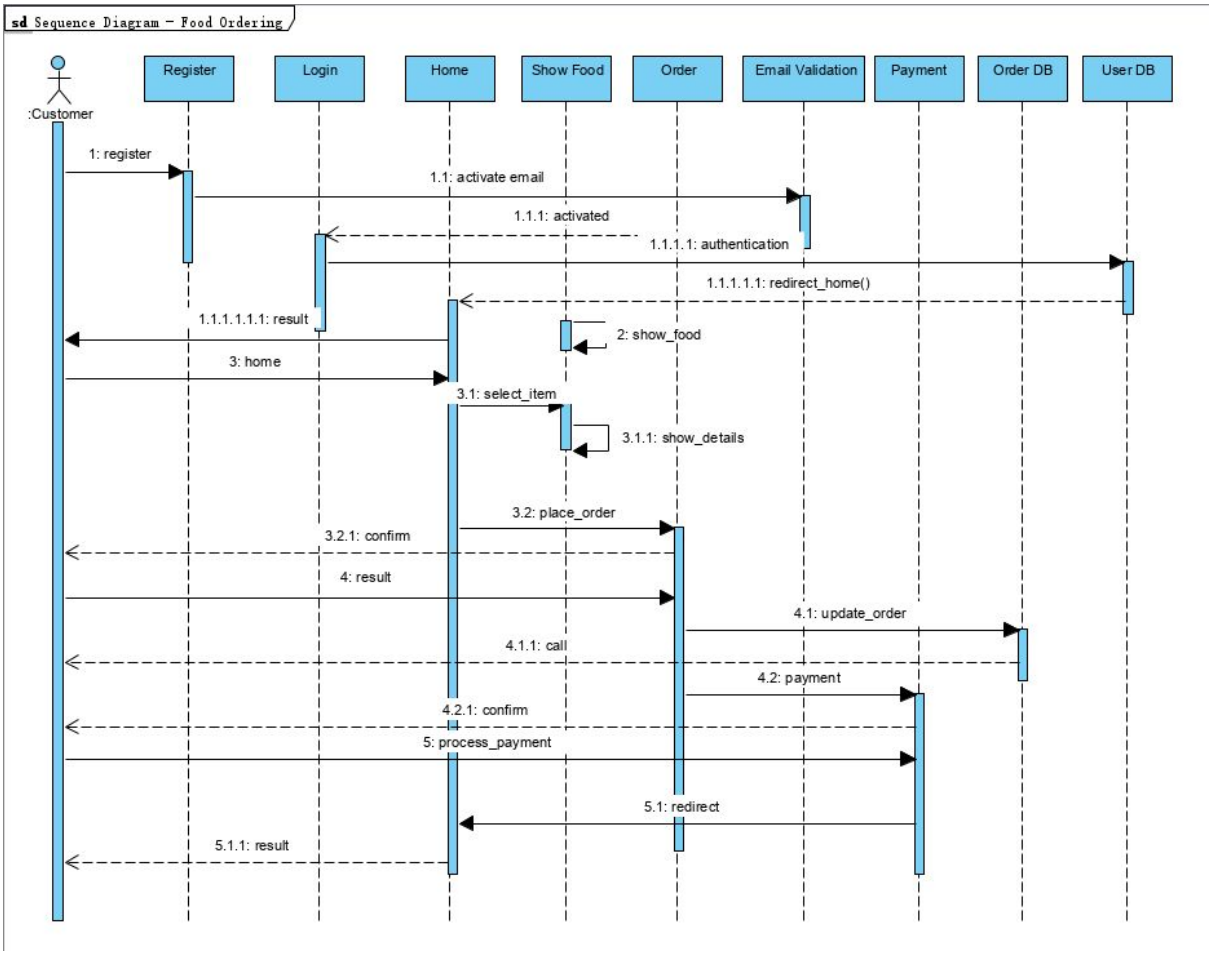
Food



Order

Sequence Diagram

For Food Ordering as a New Customer



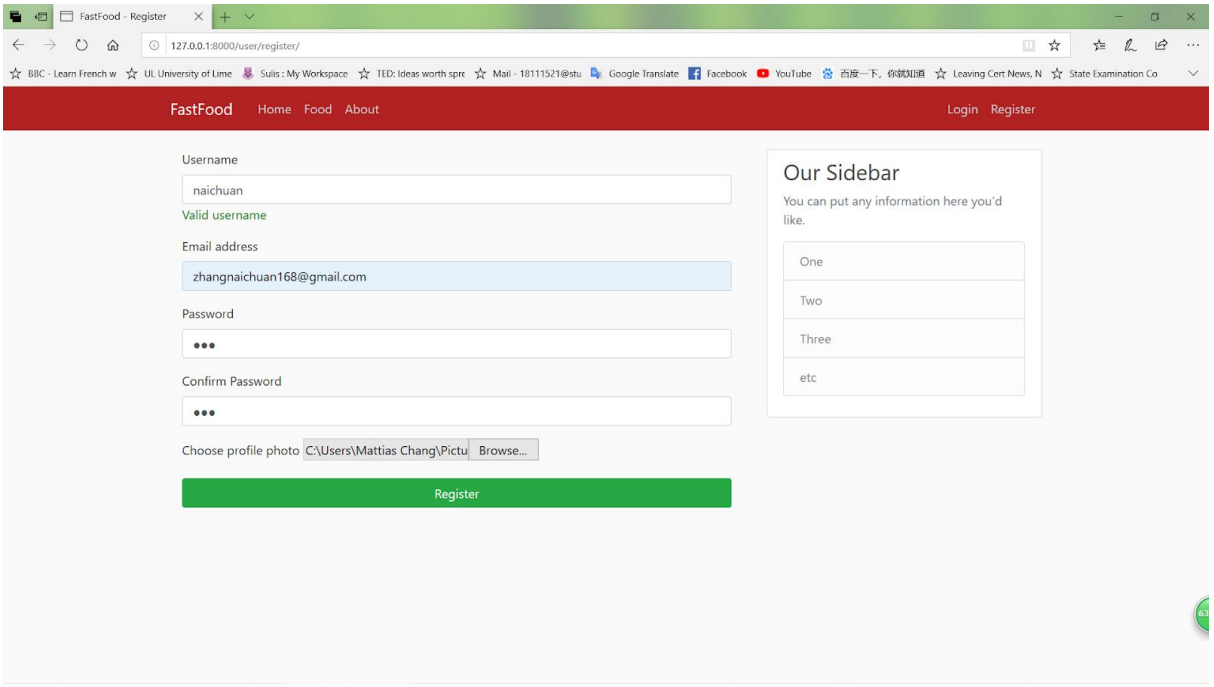
Class Diagrams(Attached Files)

three diagrams have been attached as png files.

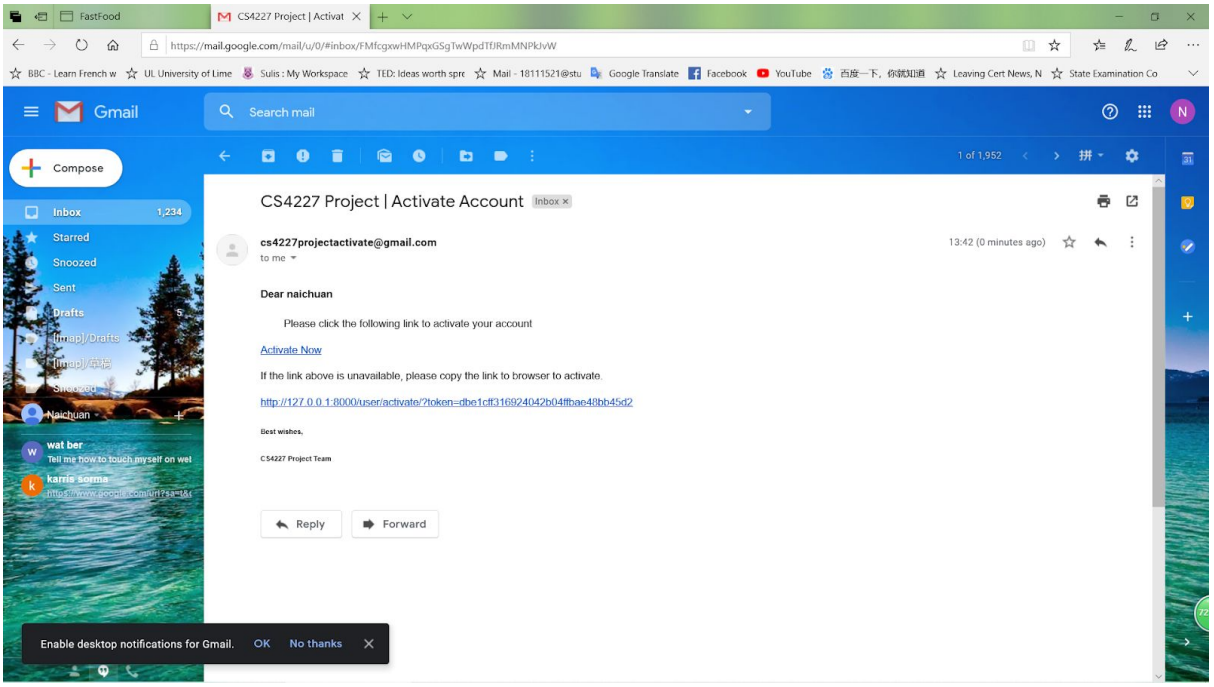
- Python Class UML(most detailed was too large to export)
- Django Model Dependency UML
- Javascript Module Dependency UML

GUI Screenshots

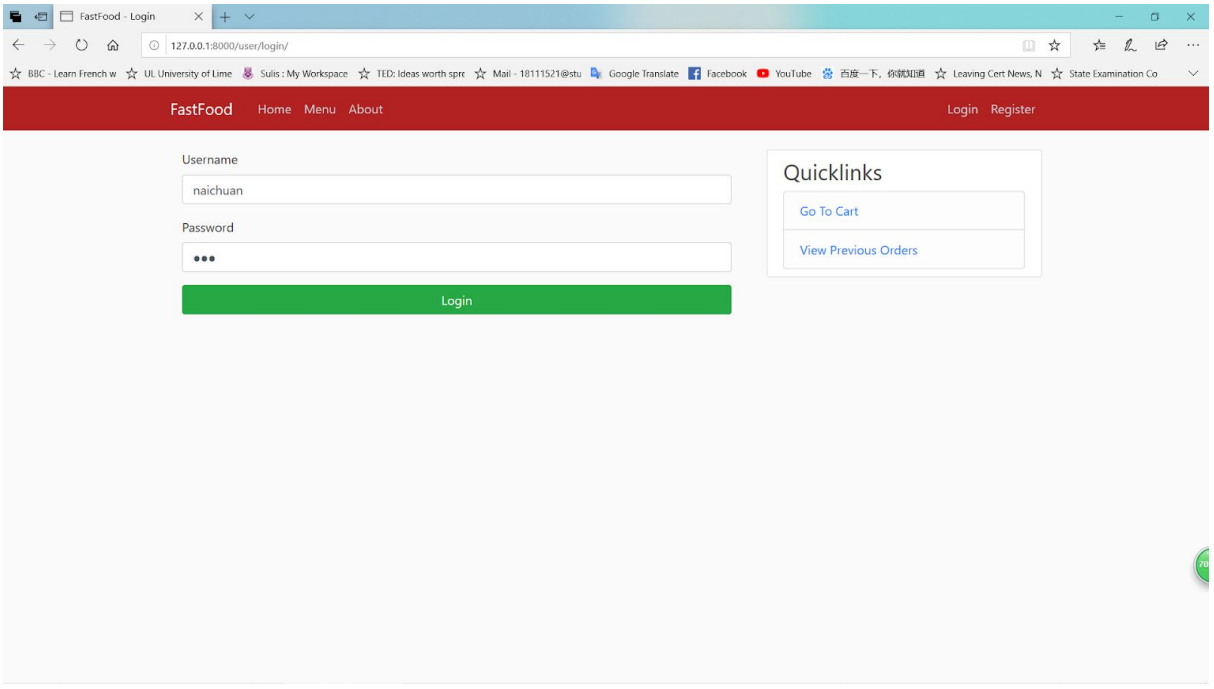
Registration



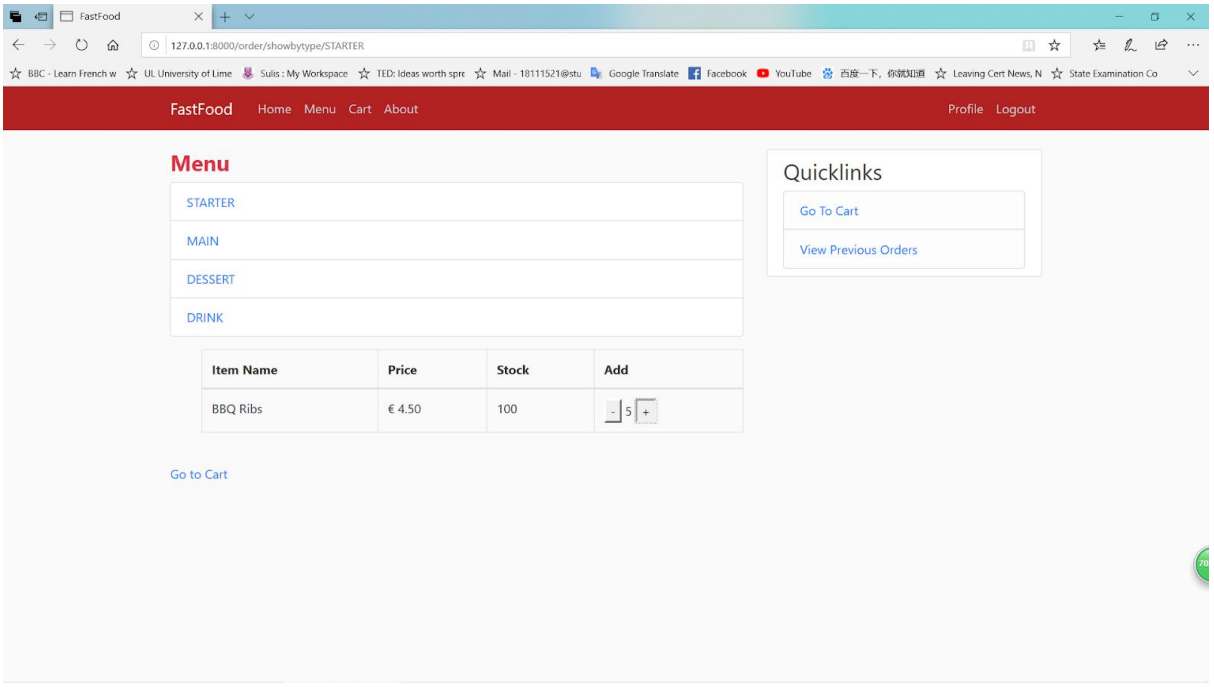
Email Activation



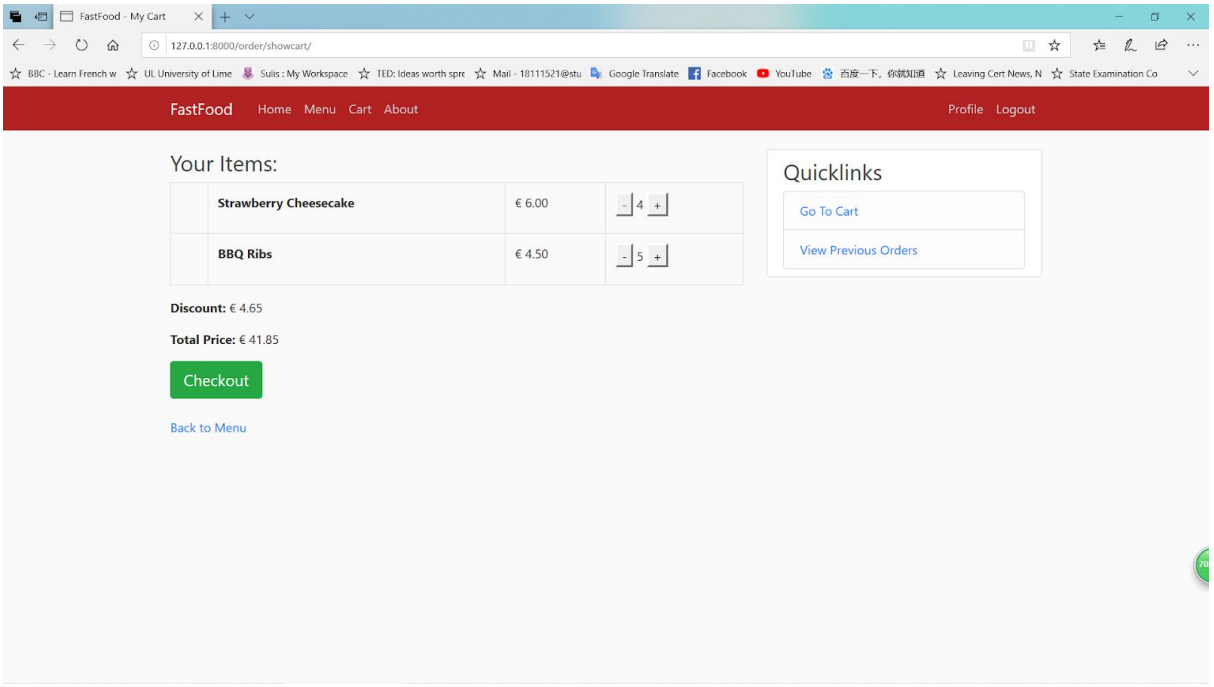
Login



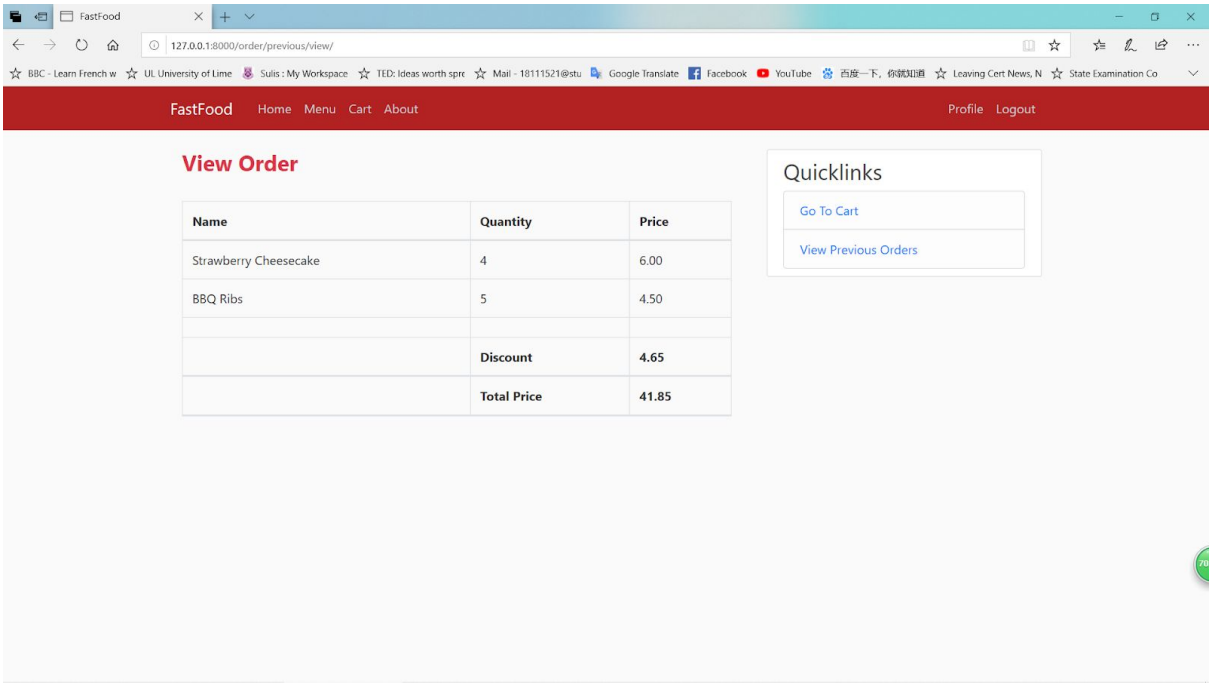
Food Menu



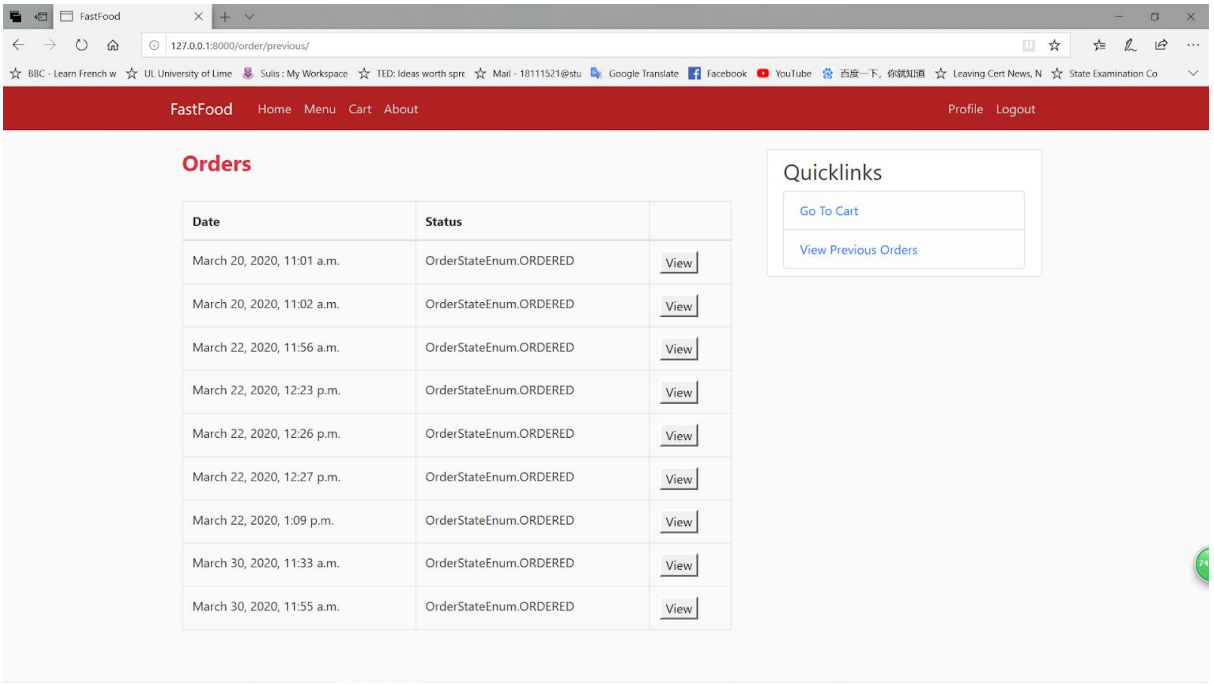
Shopping Cart



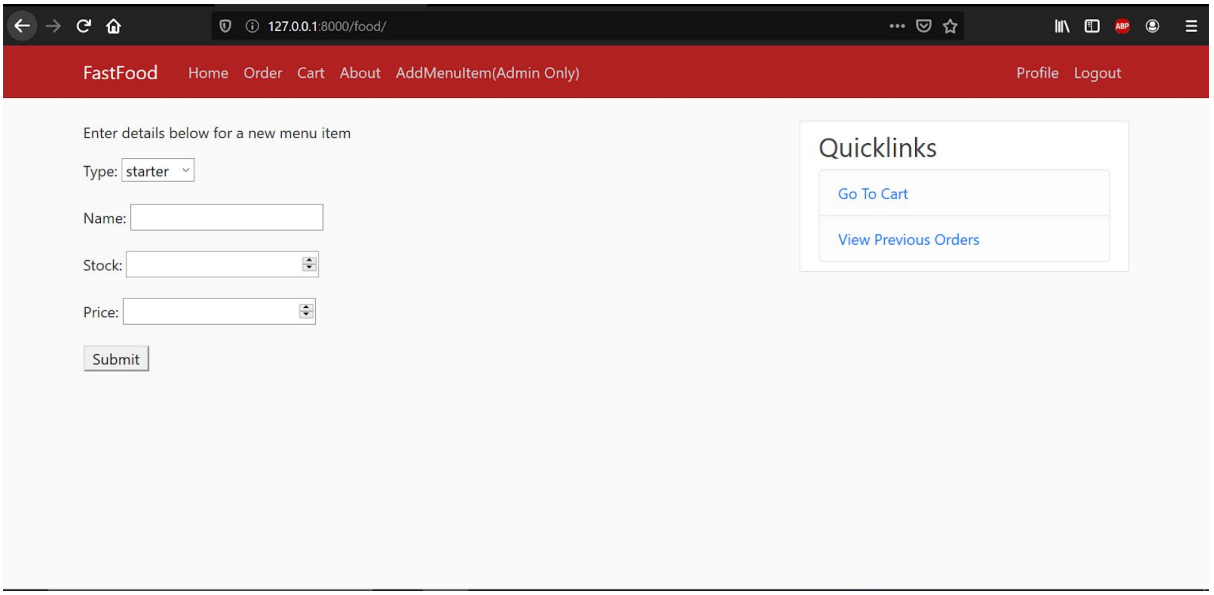
View Order



View Previous Order(s)



Add A New Menu Item



Added Values

Django ORM

The Django web framework includes a default object-relational mapping layer (ORM) that can be used to interact with application data from various relational databases such as SQLite, PostgreSQL and MySQL. The Django ORM is an implementation of the object-relational mapping (ORM) concept.

In our project, we utilised ORM with a SQLite database which stored tables such as User, Order and Item. Once we created our data models, Django automatically gave us a database-abstraction API that let us create, retrieve, update and delete objects which linked with database rows. This simplified our interaction with the database as we had no need for creation of our own SQL statements.

```
def view_orders(request):
    user = request.user
    orders = Order.objects.filter(user=user)
```

```
def create_order_item(self, order: Order):
    order_item = OrderItem(item=self._item, order=order, amount=self._amount)
    order_item.save()
```

User Activation Strategy

In many enterprise projects, the activation authentication of new users is a crucial link. At present, the most common user activation channels mainly include three methods, mail, SMS and manual review. In this project, when the user registers, an activation link is sent to the email address provided by the user. The user needs to click the activation link in order to log in normally. Activation links in mailboxes should have the following characteristics: first, there should be a unique identity of the user in the link, and there should be an expiration time and a limit on the number of times the identity can be used.

Django offers multiple tools needed in the development, including the ability to send mail. In order to send an email, the recipient's email address, as well as the sender's username,

password, server and content are required. The images below show the config required for sending an email and send_activate_email() method.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'cs4227projectactivate@gmail.com'
EMAIL_HOST_PASSWORD = 'znc12345678'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_USE_SSL = False

SERVER_HOST = '127.0.0.1'
SERVER_PORT = 8000
```

```
def send_activate_email(username, receiver_email, token):
    subject = 'CS4227 Project | Activate Account'
    from_email = EMAIL_HOST_USER
    recipient_list = [receiver_email, ]
    data = {
        'username': username,
        'activate_url': 'http://{}:{}/user/activate/?token={}'.format(SERVER_HOST, SERVER_PORT, token),
    }
    html_message = loader.get_template('user/activate_message.html').render(data)
    send_mail(subject=subject, message="",
              from_email=from_email, recipient_list=recipient_list,
              fail_silently=False, html_message=html_message)
```

The email received by the user needs to contain an activation link, which should include a unique token to identify the user. In this project, the token is stored in the cache as the key, and UUID is used to generate the token. A UUID (Universally Unique IDentifier) is a 128-bit number used to uniquely identify some object or entity on the Internet.

Redis

Redis (Remote Dictionary Server) is an open source, in-memory data structure store, which can be used as a database, cache and message broker (Redis 2020). In our project, the Redis was used as cache to store user tokens. The following figure shows the configuration of Redis as cache in Django settings.

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    }
}
```

As mentioned previously, the token used to activate the user needs to be stored in the cache. The cache is saved as key-value format, where key is the token generated with UUID

and value is the user's id. And it is valid for 24 hours. After the user is activated, the cache should be deleted.

```
token = uuid.uuid4().hex
cache.set(token, user.id, timeout=60*60*24)
send_activate_email(username=username, receiver_email=email, token=token)
```

```
def activate(request):
    token = request.GET.get('token')
    user_id = cache.get(token)
    if user_id:
        # allow user to activate email once only
        cache.delete(token)
        user = User.objects.get(pk=user_id)
        user.is_active = True
        user.save()
        return redirect(reverse('login'))

    return render(request, 'user/activate_failed.html')
```

Keys accessed from *redis-cli* after a user registered.

```
(base) F:\PyCharmProjects\CS4227-Project>redis-cli
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> keys *
1) ":1:34eb046abfc14ba991a332ce4b37bf90"
2) ":1:test"
127.0.0.1:6379[1]>
```

Data Security and Username Pre-check

In order to ensure the security of the password of the user module, we usually encrypt sensitive data in terms of database storage, such as passwords, credit card numbers, etc.. The hashers module in Django provides two methods make_password() and check_password(), which are used to encode passwords during registration and verify passwords during login. They implement the use of Django's built-in password mechanism in custom user tables.

```
# encode password
password = make_password(password)
```

```
if check_password(password, user.password):
    if user.is_active:
        # save user into session
        request.session['user_id'] = user.id
        # redirect to Me when successful
        return redirect(reverse('me'))
    else:
        request.session['error_message'] = "Not activated yet"
        return redirect(reverse('login'))
else:
    request.session['error_message'] = "Wrong password"
    return redirect(reverse('login'))
```

In addition, in order to improve the user experience, we have also used Ajax to achieve the username asynchronous verification feature in registration.

```
$(function () {

    let $username = $("#username_input");

    $username.change(function () {
        let username = $username.val().trim();
        if (username.length) {
            $.getJSON('/user/checkuser/', {'username': username}, function (data) {
                console.log(data);
                let $username_msg = $("#username_msg");
                if (data["status"] === 200) {
                    $username_msg.html("Valid username").css("color", "green");
                } else if (data["status"] === 901) {
                    $username_msg.html("Username already exists!").css("color", "red");
                }
            });
        }
    });
});
```

```
def check_user(request):
    """check if the username already exists or not when registration"""

    username = request.GET.get('username')
    users = User.objects.filter(username=username)
    data = {
        "status": HTTP_USER_OK,
        "msg": "Valid username",
    }
    if users.exists():
        data["status"] = HTTP_USER_EXIST
        data["msg"] = "Username already exists!"
    return JsonResponse(data=data)
```

REST Framework

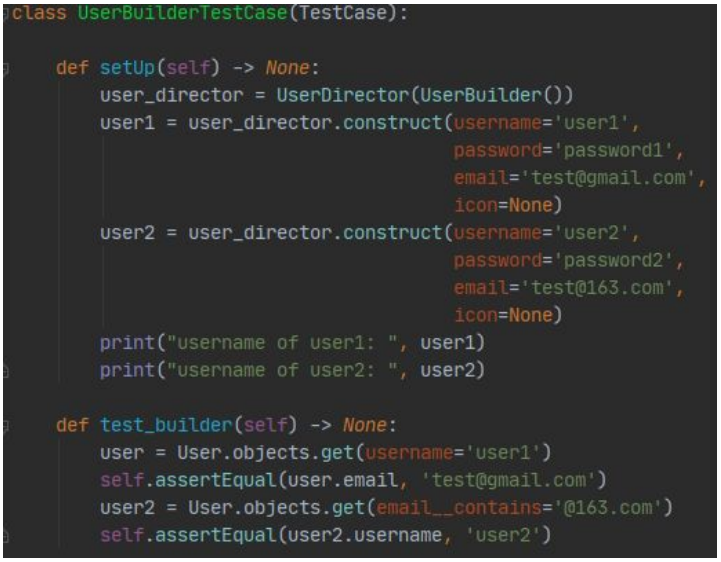
Refer to Architectural and Design patterns for use of REST framework in implementation.

Testing

In Django, automated testing can be implemented by doctests or unit tests. The testing part for our project has been implemented via Unit test by using some of Django's built-in test auxiliary classes, such as TestCase and TestClient. Django unit tests are implemented based on classes. When running automated tests, the test runner will look for the unit test case class (inherited from TestCase) in the test*.py file in the directory, and execute the function beginning with "test" in the test class.

The images below have demonstrated some main test cases we used to test the design patterns in the project. For instance, the UserBuilderTestCase tested our Builder design pattern, we set up some testing user objects by using our UserDirector, and then a temporary user object was successfully saved in the database. We directly check the user results stored in the database by directly calling the database query statement that comes with Django.

Test Builder Design Pattern:



```
class UserBuilderTestCase(TestCase):

    def setUp(self) -> None:
        user_director = UserDirector(UserBuilder())
        user1 = user_director.construct(username='user1',
                                        password='password1',
                                        email='test@gmail.com',
                                        icon=None)
        user2 = user_director.construct(username='user2',
                                        password='password2',
                                        email='test@163.com',
                                        icon=None)
        print("username of user1: ", user1)
        print("username of user2: ", user2)

    def test_builder(self) -> None:
        user = User.objects.get(username='user1')
        self.assertEqual(user.email, 'test@gmail.com')
        user2 = User.objects.get(email__contains='@163.com')
        self.assertEqual(user2.username, 'user2')
```

Test Memento Design Pattern:

```python
class MementoTestCase(TestCase):

    @classmethod
    def setUpTestData(cls):
        """set init state"""
        cls.state = 'state 1'

    def test_memento(self) -> None:
        originator = Originator(state=self.state)
        caretaker = Caretaker(originator=originator)
        caretaker.save()
        self.assertEqual(originator.get_state(), 'state 1')
        originator.set_state('state 2')
        caretaker.save()
        self.assertEqual(originator.get_state(), 'state 2')
        # roll back to original state
        caretaker.undo()
        caretaker.undo()
        print(originator.get_state())
        self.assertEqual(originator.get_state(), 'state 1')
```

Test Command Design Pattern:

```python
class OrderFrameworkTestCase(TestCase):
    """test case for order framework with Command DP"""

    @classmethod
    def setUpTestData(cls):
        """create an order"""
        cls.framework = OrderFramework()
        cls.user = User.objects.create(username='username',
                                       password='password',
                                       email='email',
                                       icon=None, is_active=True)
        cls.order = cls.framework\
            .create_order(user_id=cls.user.id,
                          total_price=100)

    def test_order_framework_with_command_dp(self):
        user = User.objects.get(username='username')
        self.assertEqual(self.order.user, user)
        self.assertEqual(self.order.total_price, 100)
```

The next two screenshots have shown some test cases we performed in the project views layer. Django provides us a test client named Client that mimics users interacting with our code at the views level. We have tested some of the main view methods and functionalities under the user and order modules. For example, in the user module, we tested whether the

user was successfully validated by injecting the user object and receiving the JSON data returned from the server side. And in the order module, we tested the basic function of adding an item to the shopping cart. Unlike user module testing, we must first create a session with a logged in user's id number in order to test the shopping cart features. Then after we executed the test case, the JSON returned a status code with a value of 200 and we received a message that the item was added successfully.

Test User Views:

```
class UserViewsTestCases(TestCase):

    def setUp(self) -> None:
        self.client = Client()
        self.credentials = {
            'username': 'naichuan',
            'password': '123',
        }
        self.user = User.objects.create(username='naichuan', password='123',
                                        email='test@gmail.com', icon=None)

    def test_check_user(self):
        response = self.client.get('/user/checkuser/')
        self.assertEqual(response.status_code, 200)
        data = response.json()
        self.assertEqual(data['status'], 200)
        self.assertEqual(data['msg'], 'Valid username')

    def test_user_login(self):
        response = self.client.get('/user/login/')
        self.assertEqual(response.status_code, 200)
        response = self.client.post('/user/login/', self.credentials, follow=True)
        self.assertEqual(response.status_code, 200)
```

Test Order Views:

```
class OrderViewsTestCase(TestCase):
    """a sample test case of json response when adding an item into cart"""

    def setUp(self) -> None:
        self.client = Client()
        self.item = Item.objects.create(name='Test Item', price=10.0, type='STARTER', stock=10)
        self.user = User.objects.create(username='naichuan', password='123')
        self.client.login(username=self.user.username, password=self.user.password)
        session = self.client.session
        session.update({
            'user_id': self.user.id,
            "expire_date": '2020-04-08',
            "session_key": 'my_session_key',
        })
        session.save()

    def test_add_item_to_cart(self):
        itemId = self.item.id
        response = self.client.get('/order/addtocart/', {'itemId': itemId})
        self.assertEqual(response.status_code, 200)
        data = response.json()
        print(data)
        self.assertEqual(data['status'], 200)
        self.assertEqual(data['msg'], 'An item has been added to cart successfully!')
        self.assertEqual(data['num'], 1)
```

Module ↓	statements	missing	excluded	coverage
CS4227_Project__init__.py	0	0	0	100%
CS4227_Project\settings.py	31	0	0	100%
CS4227_Project\urls.py	4	0	0	100%
food__init__.py	0	0	0	100%
food\admin.py	1	0	0	100%
food\foodfactory.py	51	30	0	41%
food\forms.py	9	0	0	100%
food\migrations\0001_initial.py	6	0	0	100%
food\migrations\0002_foodtype_name.py	5	0	0	100%
food\migrations__init__.py	0	0	0	100%
food\models.py	30	2	0	93%
food\tests.py	1	0	0	100%
food\urls.py	3	0	0	100%
food\views.py	33	25	0	24%
manage.py	12	2	0	83%
middleware__init__.py	0	0	0	100%
middleware\middleware.py	19	5	0	74%
order__init__.py	0	0	0	100%
order\admin.py	6	0	0	100%
order\command__init__.py	0	0	0	100%
order\command\command.py	4	1	0	75%
order\command\commandmanager.py	18	5	0	72%
order\command\ordercommand.py	39	9	0	77%
order\command\orderservices.py	32	11	0	66%
order\command\undoable.py	4	1	0	75%
order\composite__init__.py	0	0	0	100%
order\composite\itemcomponent.py	8	2	0	75%
order\composite\itemcomposite.py	19	9	0	53%
order\composite\itemleaf.py	11	5	0	55%
order\factory__init__.py	0	0	0	100%
order\orderframework.py	20	3	0	85%
order\strategy__init__.py	0	0	0	100%
order\tests.py	49	0	0	100%
order\urls.py	3	0	0	100%
order\views.py	179	141	0	21%
order\visitor__init__.py	0	0	0	100%
order\visitor\abstractvisitor.py	32	14	0	56%
order\visitor\stringvisitor.py	34	24	0	29%
order\visitor\visitable.py	11	2	0	82%
rest__init__.py	0	0	0	100%
rest\admin.py	1	0	0	100%
rest\api.py	19	0	0	100%
rest\migrations__init__.py	0	0	0	100%
rest\models.py	1	0	0	100%
rest\serializers.py	18	0	0	100%
rest\tests.py	1	0	0	100%
rest\urls.py	8	0	0	100%
user__init__.py	0	0	0	100%
user\admin.py	3	0	0	100%
user\constants.py	2	0	0	100%
user\email_helper.py	10	6	0	40%
user\migrations\0001_initial.py	5	0	0	100%
user\migrations__init__.py	0	0	0	100%
user\models.py	12	0	0	100%
user\tests.py	33	0	0	100%
user\urls.py	4	0	0	100%
user\userbuilder__init__.py	0	0	0	100%
user\userbuilder\abstractuserbuilder.py	16	7	0	56%
user\userbuilder\userbuilder.py	27	0	0	100%
user\userbuilder\userdirector.py	7	0	0	100%
user\views.py	84	44	0	48%
Total	**1082**	**377**	**0**	**65%**

Finally, we used the Coverage tool to measure the code coverage of our unit tests. The Coverage.py is a tool used to test the code coverage of python programs. It can identify which parts of the code have been executed, and identify the executable but not executed code. Coverage testing is usually used to measure the effectiveness and completeness of the test.

Issues

We faced a combination of issues when undertaking this project particularly due to the unforeseen nature of COVID-19. This interruption of the college semester led to the closal of the college therefore leaving team members who were reliant on access to college resources without access to adequate computers for a considerable period of time.

Another issue we faced was the selection of the framework/language. In hindsight choosing a language in which all the teammates were not familiar was probably not the correct decision,but the nature of the closal of the college was not predictable. Choosing the django framework combined with the college closing meant some team members had to learn the language and implement features in parallel at a very late stage of the semester and this proved quite problematic. Additionally, the added pressure of looming FYP and other deadlines added to the already considerable stress caused by these previously mentioned difficulties.

Evaluation / Critique

We believe that the patterns selected were relevant to our architectural use cases and were successful in supporting our selected quality attributes.

For instance, the use of the builder pattern is relevant in the creation of a user because it helps us in building desired User objects with all mandatory attributes and combination of optional attributes, without losing the immutability. For example in our project each user must have at least a username and a password as these are mandatory fields, but the addition of a user profile picture remains an option.

Although we faced issues during the semester due to the choice of language, this project gave us an opportunity to learn a new skill since java is the most prominent language we've all used. Python is one of the most in demand languages currently especially in the field of web development, with Django being its most popular framework. Django aims to provide simplicity, flexibility, reliability, and scalability in the creation of web applications.

Django was especially useful in achieving some of our non-functional requirements. By default django prevents most common security mistakes including XSS (cross-site scripting), CSRF (cross site request forgery) and SQL injection protection. Django is also extremely extendable and third-party packages can easily be added to an app. In this project we added packages such as the Django REST Framework and Redis.

References

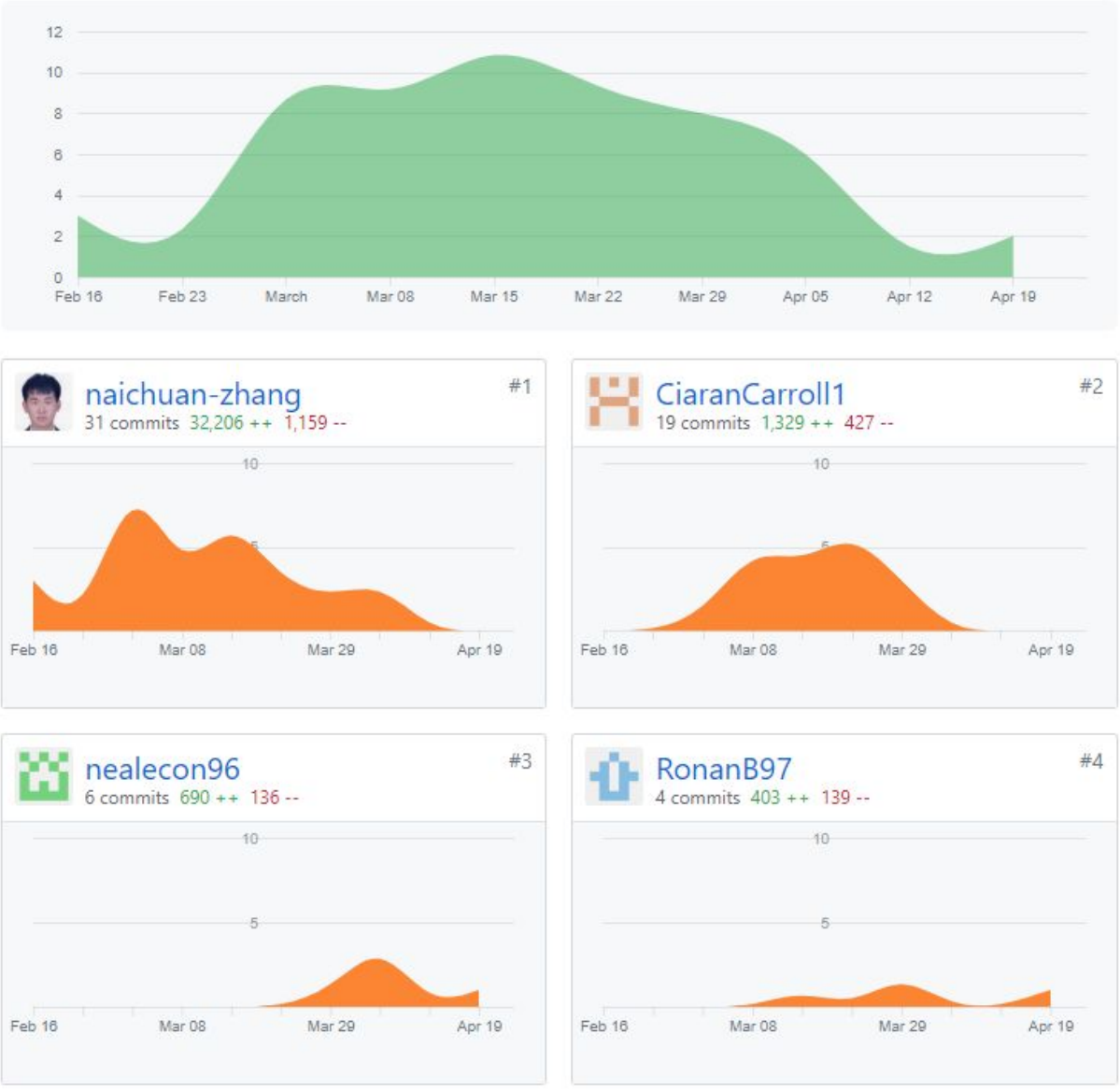
Redis, *Introduction to Redis*, available: https://redis.io/topics/introduction [accessed 23 Mar 2020].

Sarcar, V. (2018) *Java Design Patterns: A Hands-On Experience with Real-World Example*, O'Reilly, available: https://learning.oreilly.com/library/view/java-design-patterns/9781484240786/html/395506_2_En_19_Chapter.xhtml [accessed 30 Mar 2020].

https://refactoring.guru/design-patterns/memento/python/example [accessed 12 March 2020]

Contributions

Team Member Contribution - GitHub



Team Member Contribution - Source Code

Note: All of the apps.py, __init__.py, migrations and settings files are excluded.

Python Files (.py)	Contributor(s)	Lines of Code
CS4227_Project\urls.py	ALL	28
food\admin.py	Ronan	4
food\foodfactory.py	Ronan	65
food\forms.py	Ronan	11
food\models.py	Ronan	51
food\urls.py	Ronan	7
food\views.py	Ronan	57
food\interceptor\Context.py	Ronan	31
food\interceptor\food_dispatcher.py	Ronan	18
food\interceptor\food_interceptor.py	Ronan	15
food\interceptor\interceptor_framework.py	Ronan	21
middleware\middleware.py	Naichuan, Ciarann, Neale	43
order\admin.py	Ciaran	8
order\command\command.py	Naichuan	9
order\command\commandmanager.py	Naichuan	31
order\command\ordercommand.py	Naichuan	52
order\command\orderservices.py	Naichuan	49
order\command\undoable.py	Naichuan	9
order\composite\itemcomponent.py	Ciaran	15

order\composite\itemcomposite.py	Ciaran	28
order\composite\itemleaf.py	Ciaran	17
order\factory\discount\abstractdiscount.py	Ciaran	9
order\factory\discount\golddiscount.py	Ciaran	9
order\factory\discount\nodiscount.py	Ciaran	9
order\factory\discount\silverdiscount.py	Ciaran	9
order\factory\discountfactory.py	Ciaran	24
order\factory\orderitemfactory.py	Ciaran	8
order\memento\caretaker.py	Naichuan	30
order\memento\memento.py	Naichuan	12
order\memento\originator.py	Naichuan	29
order\models.py	Naichuan, Ciaran, Neale	98
order\forms.py	Neale	13
order\strategy\paycard.py	Neale	9
order\strategy\paycontext.py	Neale	18
order\strategy\paypaypal.py	Neale	9
order\strategy\strategy.py	Neale	8
order\orderframework.py	Naichuan	31
order\urls.py	Naichuan, Ciaran, Neale	20
order\views.py	Naichuan, Ciaran, Neale	314

order\visitor\abstractvisitor.py	Ciaran	43
order\visitor\stringvisitor.py	Ciaran	41
order\visitor\visitable.py	Ciaran	18
rest\admin.py	-	0
rest\api.py	Ciaran	36
rest\models.py	-	0
rest\serializers.py	Ciaran	27
rest\urls.py	Ciaran	12
user\admin.py	Naichuan	4
user\constants.py	Naichuan	3
user\email_helper.py	Naichuan	18
user\models.py	Naichuan	17
user\urls.py	Naichuan	17
user\userbuilder\abstractuserbuilder.py	Naichuan	32
user\userbuilder\userbuilder.py	Naichuan	38
user\userbuilder\userdirector.py	Naichuan	16
user\views.py	Naichuan	131
logger.py	Naichuan, Ronan	16
order\tests.py	Naichuan	79
user\tests.py	Naichuan	55
delivery\app.py	Neale	5
delivery\models.py	Neale	2
delivery\url.py	Neale	8
delivery\views.py	Neale	35

delivery\memento\memento.py	Neale	8
delivery\memento\caretaker.py	Neale	25
delivery\memento\concretememento.py	Neale	9
delivery\memento\originator.py	Neale	21

Static Files (.css and .js)	**Contributor(s)**	**Lines of Code**
static\app\main\css\main.css	Naichuan	10
static\app\main\js\cart.js	Naichuan	51
static\app\main\js\me.js	Naichuan	10
static\app\main\js\show.js	Naichuan	38
static\app\main\js\view_orders.js	Ciaran	4
static\app\user\css\user.css	-	-
static\app\user\js\register.js	Naichuan	33
static\app\main\js\make_payment.js	Neale	12
static\app\main\js\view_delivery.js	Neale	4
static\app\main\js\view_order.js	Neale	17

Template Files (.html)	**Contributor(s)**	**Lines of Code**
templates\main\adminCheck.html	Ronan	18
templates\main\cart.html	Naichuan, Ciaran	54

templates\main\food.html	Ronan	11
templates\main\home.html		
templates\main\me.html	Naichuan	33
templates\main\order.html	Ciaran	11
templates\order\checkout.html	Ciaran	18
templates\order\create_order.html	Ciaran	71
templates\order\data.html	Ciaran	4
templates\order\show.html	Naichuan	49
templates\order\view_order.html	Ciaran, Neale	58
templates\order\view_orders.html	Ciaran	33
templates\order\paymentcomp.html	Neale	8
templates\user\activate.html	Naichuan	7
templates\user\activate_failed.html	Naichuan	7
templates\user\activate_message.html	Naichuan	7
templates\user\login.html	Naichuan	24
templates\user\register.html	Naichuan	39
templates\base.html	Naichuan, Ciaran, Neale	94
templates\base_user.html	Naichuan	16
templates\delivery\delivery.html	Neale	33
templates\delivery\view_delivery.html	Neale	36

Team Member	**Team Contribution (LoC)**

Ciaran Carroll	1092
Naichuan Zhang	1324
Neale Conway	554
Ronan Barry	403

Team Member Contribution - Report

Team Member	Team Contribution
Ciaran Carroll	System Architecture, Factory/Composite/Visitor/REST Design Patterns, Added Values (Django ORM),
Naichuan Zhang	Functional Requirements, Use Case Diagram, Use Case Descriptions, Builder/Command/Memento Design Patterns, Sequence Diagram, Added Values, Testing
Neale Conway	Scenario, Non-Functional Requirements, Tactics To Support Quality Attributes, Strategy/Memento Design Patterns, Issues, Evaluation
Ronan Barry	Use Case Descriptions, GUI Screenshots, Factory and Interceptor Design Patterns, Class Diagrams, Python Class UML Diagram, Django Model Dependency, Javascript Module Dependency Diagram

CS4227: Software Design & Architecture
Guidance on Marking Scheme for Team-Based Project:
Semester 2, 2019-2020
(Version 1 10-02-2020)

Name 1:	ID1:
Name 2:	ID2:
Name 3:	ID3:
Name 4:	ID4:

	Item	Detailed Description	Marks Allocated Sub-	Marks Allocated Total	Marks Awarded
1-2	Presentation	• General Presentation • Adherence to guidelines i.e front cover sheet, blank marking scheme, table of contents		**2**	
3	Requirements	• Narrative, Use Case diagram, and SAMPLE Use Case Description • Discussion on NFRs and tactics	1 2	**3**	
4	Discussion on Architectural and Design Patterns	• The Interceptor architectural pattern. • BRIEF discussion of 5 design patterns from CS4227 • Discussion on 7th pattern	1 2 1	**4**	
5	System Architecture			**2**	
6	Structural and Behavioural Diagram	• Class Diagram with package iconography • Interaction diagram for key use case	4 1	**5**	
7	Code	• Matches/Supports/Realises diagrams • Interceptor pattern correctly implemented. • 5 Design Patterns from CS4227 correctly implemented • 7th Pattern • Exposes intent, naming conventions clearly identify design patterns used • At least 4 packages, one developed by each team member • Use of version control	3 3 10 3 2 P/F P/F	**15**	
8	Added Value	Two examples, 3 marks each.		**6**	
9	Testing	Should be automated.		**P/F**	
10	Issues	Satisfactorily documented. No marks awarded.		N/A	
11	Evaluation / Critique	Is it the case that the patterns selected supported relevant architectural use cases? If not, why not? Any alternatives?		**2**	
12	References			**1**	
	Interview Week 11 or 12	• Competent code inspection • Working demo		**(P/F)**	
	SUB-TOTAL (A)			**40**	

PENALTIES					
	Description	**S1**	**S2**	**S3**	**S4**
1	Late Submission				
2	Failure to contribute to coding effort				
3	Failure to contribute to writing of report				
4	Failure to report problems with team dynamics				
5	Failure to contribute to demo week 13				
	Sub-total (B)				

		FINAL MARKS AWARDED		
	Student1	**Student2**	**Student 3**	**Student 4**
(A-B)				