

Relazione Progetto

Pizzeria Taurus

Un sito web per ordinare e recensire pizze online

Corso: Tecnologie Web

Autore: Ruben Biskupec

Relazione Progetto

Pizzeria Taurus
Un sito web per ordinare e recensire pizze online

da

Ruben Biskupec

Nome Studente	Matricola
Ruben Biskupec	134731

Professoressa:	C. Canali
Assistente:	F. Faenza
Facoltà:	Informatica
Dipartimento:	FIM

Contents

1	Traccia del progetto	1
2	Architettura e Tecnologie	3
3	Utenti e Auth	7
4	Modelli	9
5	Testing	13
6	Risultati	15
7	Riflessioni	19

1

Traccia del progetto

Il progetto ha lo scopo di realizzare un sito web utilizzabile da una qualsiasi pizzeria, nello specifico e' stato disegnato con in mente la Pizzeria Taurus, ovvero la pizzeria di famiglia.

Le funzionalita' previste dalla piattaforma per ora sono le seguenti:

- **Visione del menu':** Visualizzare tutte le pizze del menu' attualmente disponibili. La disponibilita' e' calcolata dinamicamente in base alla disponibilita' degli ingredienti e impasti disponibili. Il menu' e' categorizzato per tipologia di cibo: Pizze, Saltimbocca e Calzoni. Ci sono inoltre le bevande.
- **Home page:** Pagina in cui si posso vedere i dati comunemente ricercati come orari di apertura, breve storie della pizzeria, ecc.
- **Registrazione utente:** Un cliente si puo' registrare sul sito per avere accesso ad ulteriori funzionalita', ovvero effettuare ordini, sia a domicilio che a casa, e recensirli.
- **Login/Logout utente:** Una volta registrati, si puo' fare log-in nel sito con le proprie credenziali (email e password). Ovviamente si puo' anche effettuare il log-out per accedere con altre credenziali.
- **Pizze Custom:** E' permesso di ordinare pizze customizzate partendo da quelle base presenti sul menu'. La customizzazione consiste nello scegliere il tipo di impasto, ingredienti aggiuntivi, dimensione della pizza (piccola, normale, tirata, alta) e aggiunte di eventuali note (es. rimozione ingredienti, ben cotta, tagliata).
- **Carrello:** Si aggiungono le pizze desiderate nel carrello. Queste vengono salvate nel carrello tramite localStorage del Browser in modo da essere persistenti nel caso si dovesse ricaricare la pagina.
- **Ordini:** Una volta soddisfatti con le pizze nel carrello si puo' confermare l'ordine, specificando se lo si vuole da asporto o a domicilio, e quindi l'indirizzo di consegna, la data e l'orario desiderati.
- **Recensioni:** Completato l'ordine, lo si puo' recensire. La recensione consiste in titolo, descrizione e una valutazione fino a 5 stelle. Queste devono sempre essere collegare a un ordine. Si possono poi modificare o eliminare.
- **About page:** Pagina in cui e' possibile visualizzare le recensioni di tutti i clienti.
- **User page:** Pagina in cui e' possibile visualizzare i propri ordini passati, fare nuove recensioni o modificarle/eliminarle. E' inoltre disponibile una vista dei propri dati utente: nome, cognome, email, indirizzo, telefono.
- **Admin page per ordini odierni:** Sono presenti 2 pagine nascoste riservate ai solo utenti admin (is_staff=True). In questa si possono visualizzare gli ordini da completare per questa serata ordinati per l'ora di consegna. E' presente un pulsante per completare l'ordine che cambia lo stato dell'ordine e lo toglie dalla vista ordini corrente. La pagina va in polling ogni minuto per aggiornare sempre gli ordini.
- **Admin page per ricerca ordini:** Da questa questa pagina admin si possono visualizzare gli ordini filtrati per start_date e end_date. Verranno quindi mostrati tutti gli ordini che hanno una data di consegna compresa tra le 2 date.

- **CRUD di elementi del menu':** Per modificare queste entita' verra' utilizzata la pagina admin fornita da Django, ha tutto il necessario per compiere questo scopo. Inoltre queste modifiche avverranno molto di rado, motivo per cui non era conveniente creare un'interfaccia a parte per compiere le medesime funzionalita'.
- **Mobile friendly:** L'applicativo e' stato sviluppato con in mente un approccio mobile-first, e' quindi molto responsive per tutte le dimensioni di schermo.

2

Architettura e Tecnologie

Per il progetto e' stato utilizzato Django con Django Rest Framework per il backend, Sqlite come database e Vue.js per sviluppare il frontend della Single Page Application.

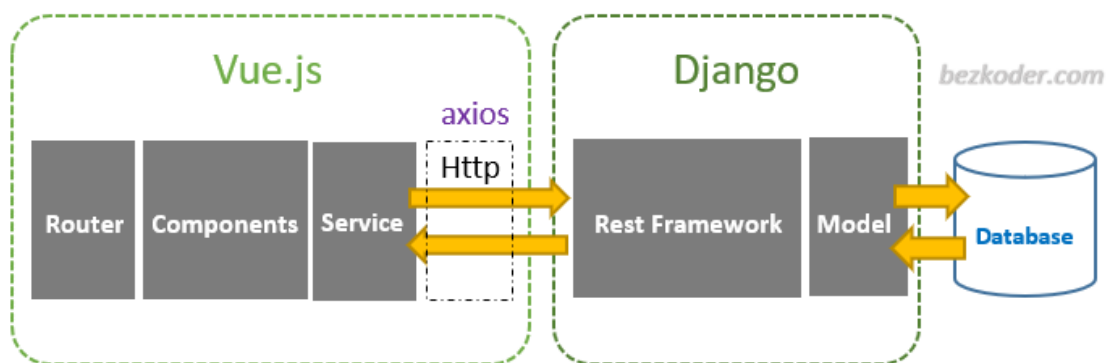


Figure 2.1: Architettura

In particolare, per il frontend:

- **Vue:** E' uno dei framework Javascript piu' utilizzato per sviluppare SPA. Per questo sito ci tenevo ad evitare i refresh della pagina ad ogni click. Inoltre ero gia' familiare con le basi del framework grazie ad una esperienza lavorativa e volevo approfondirlo siccome mi ero trovato bene nell'utilizzo. Offre anche un'estensione del browser, Vue Developer Tools che e' molto comoda. <https://v2.vuejs.org/>
- **Router:** E' il router ufficiale e uno componenti standard e serve per la navigazione all'interno dell'applicazione. E' ben documentato e ci sono molte funzionalita' disponibili. <https://router.vuejs.org/>
- **Vuetify:** Libreria grafica basata sul framework di Material Design. E' una libreria di UI molto ampia che gia' conoscevo, ha anche una buona integrazione con le icone e permette di sviluppare in fretta interfacce senza preoccuparsi troppo dell'HTML e CSS. Inoltre supporta molto bene le viste da mobile. <https://vuetifyjs.com/en/>
- **Axios:** E' la libreria standard per consumare API HTTP in modo promise-based. L'utilizzo della libreria e' stato circoscritto principalmente alla classe del servizio Http, che si occupa appunto di effettuare chiamate. Sono state implementate delle chiamate sia con async/await che con delle callback. Molti dei dati ottenuti sono stati poi inseriti nello store di Vuex. <https://v2.vuejs.org/v2/cookbook/using-axios-to-consume-apis.html?redirect=true>
- **Vuex:** E' una libreria che implementa un pattern di state management, e' simile a Flux per React. E' la libreria standard di state management per l'ecosistema Vue, quindi e' ben documentata e

piena di esempi. Viene utilizzata per avere un'unica fonte di verita' su cui si riferiscono i componenti figli. E' stata molto comoda per evitare di dover passare i dati ai componenti figli su e giu' tramite props e eventi. Si puo' controllare lo storico degli eventi di mutazione dei developer tools di vue, utile per debuggare.
<https://vuex.vuejs.org/>

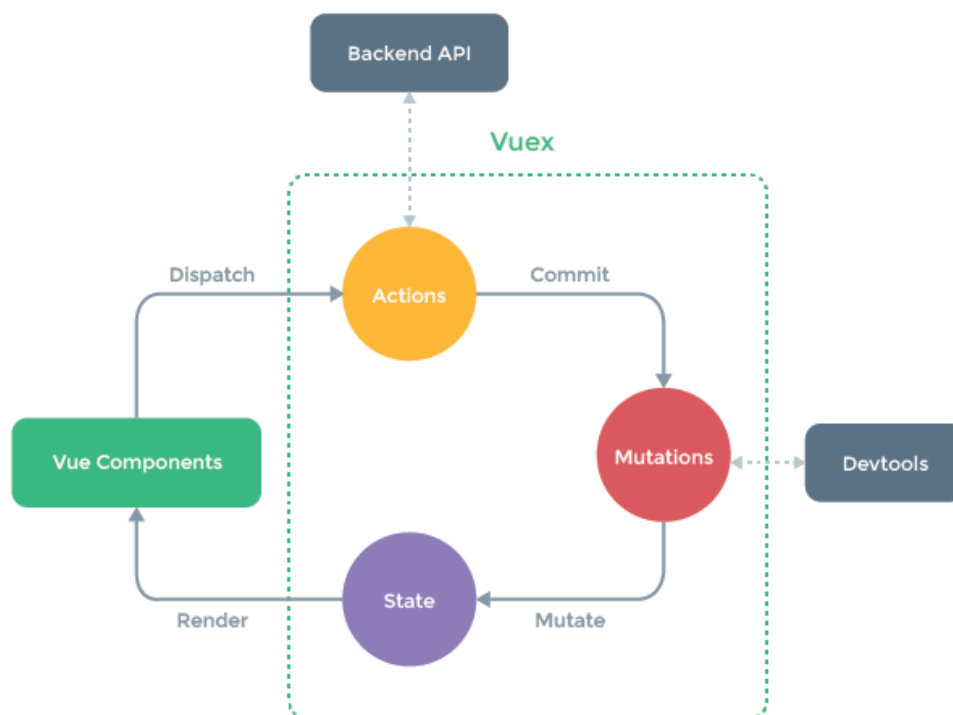


Figure 2.2: Vuex State Management Pattern

L'App root di Vue contiene l'header che viene sempre mostrato e il Router. Il Router fa vedere delle View in base all'URL. Le View sono composte da diversi componenti riutilizzabili. Se si vuole effettuare chiamate HTTP si utilizza l'HttpService che utilizza axios per comunicare. Se i componenti vogliono aggiornare lo stato di Vuex possono chiamare delle azioni tramite dispatch, queste effettueranno delle chiamate asincrone al server e committeranno il cambiamento, creando così la mutazione. Una volta mutati i dati, i componenti che li utilizzano saranno aggiornati dinamicamente.

Per il backend:

- **Sqlite:** Un database embedded, fornito di default da Django. Per applicazioni che non devono scalare come questa e' una ottima scelta, anche perche' si evita il roundtrip time al database, seppur breve. Inoltre mantiene bassa la complessita' del progetto e ed' pronto sin da subito senza configurazioni o modifiche.
<https://sqlite.org/docs.html>
- **Django:** Framework base del backend, utilizzato per la creazione dei Modelli delle Classi. Fornisce l'admin panel, la CLI per la creazione di superuser, start del server, migrazioni, ecc. e l'ORM per interfacciarsi con il DB. Ha una buona struttura e divisione in applicazioni e molta documentazione. <https://www.djangoproject.com/>
- **Django Rest Framework:** E' un'applicazione di Django, permette di creare api Web, in questo caso una RESR API. Offre i Serializzatori, simili ai form, policy di autenticazione e un browser di API molto comodo.
<https://vuetifyjs.com/en/>
- **Function Based Views:** Ho scelto le function based views perche' sono stato attirato dal fatto di vedere i verbi ammessi (es.GET, POST) in cima agli endpoint. Tornando indietro probabilmente sceglierei le Class Based Views, siccome mi sono accorto a posteriori che erano documentare meglio e piu' utilizzate.
- **Cors Headers:** Applicazione Django per gestire le intestazioni CORS. Si specifica in settings.py la porta 8080 che e' quella default di Vue.

```
1 CORS_ALLOWED_ORIGINS = [  
2     "http://localhost:8080"  
3 ]
```

<https://pypi.org/project/django-cors-headers/>

- **Django Filters:** Applicazione Django per gestire le i filtri dell'url nelle richieste all'API.
https://django-filter.readthedocs.io/en/stable/guide/rest_framework.html
- **Auth token e Djoser:** Applicazioni per gestire l'autenticazione e autorizzazione tramite auth token. Verranno approfonditi nella prossima sezione.
https://djoser.readthedocs.io/en/latest/getting_started.html

Software e risorse utilizzati durante lo sviluppo:

- **Chrome devtools:** Parte essenziale per lo sviluppo dell'applicativo. Sia l'esploratore del DOM per controllare la struttura HTML e le classi CSS applicate, che per il debug in sources e la stampa di errori in console, ma anche il controllo della network tab con l'analisi del payload, header e delle risposte alle richieste HTTP eseguite.
- **DB Browser per SQLite:** Interfaccia grafica per interfacciarsi con il database SQLite ed effettuare query in SQL direttamente sul DB. Utile all'inizio del progetto in fase di debug durante la creazione dei Model e in fasi di debug.
- **PyCharm:** IDE Python, sono rimasto sorpreso dall'accuratezza dei warning e dall'autocomplete. Sembrava quasi di programmare in un linguaggio fortemente tipizzato come Java.
- **GitHub:** Repository cloud del progetto Git.
- **Postman:** Tool grafico per effettuare chiamate HTTP. Permette di aprire varie tab con URL, azioni, header e body diversi. Comodissimo per testare il corretto funzionamento dell'API.
<https://www.postman.com/>

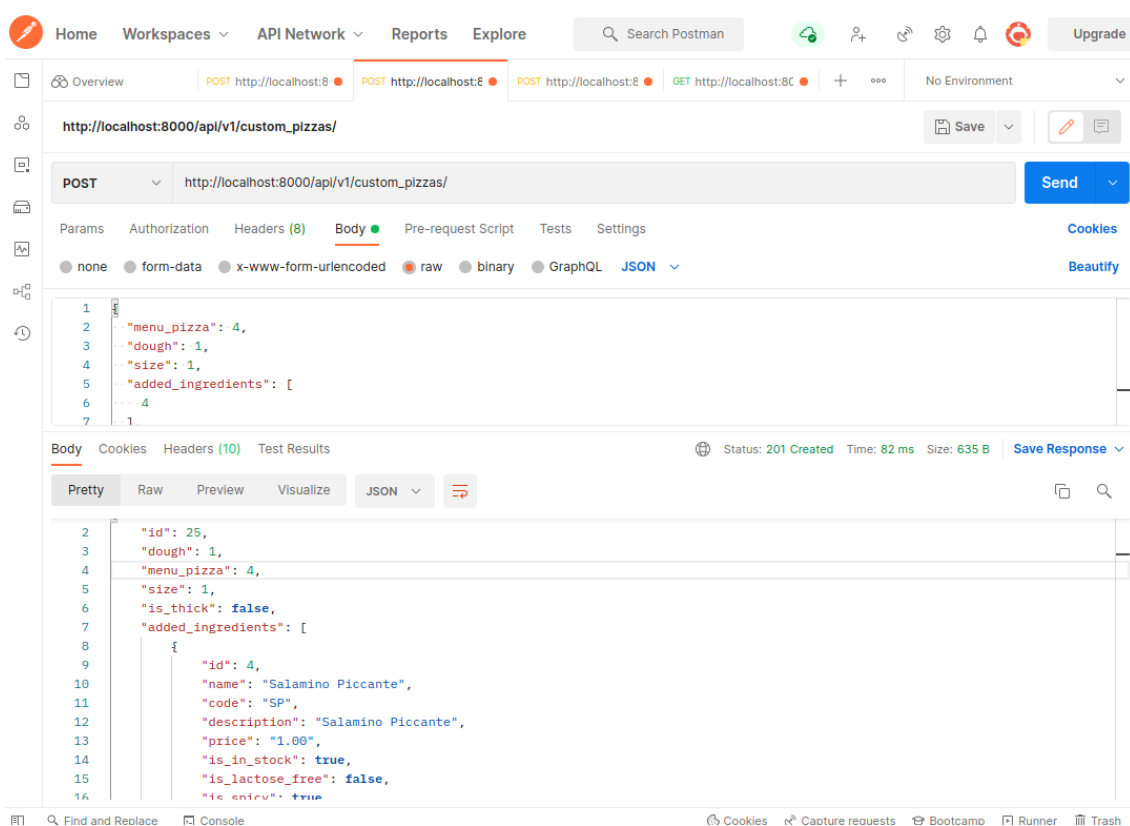


Figure 2.3: UI di Postman

3

Utenti e Auth

Gli utenti sono suddivisi in 4 gruppi:

- **Anonimo:** Questo utente puo' visualizzare le tab di Home, About us e Menu. Cliccando su ordina verra' reindirizzato alla pagina di sign-up.
- **Customer:** Ottiene vari diritti di lettura e scrittura di entita'. Puo' creare pizze custom, ordini e recensioni. Ha accesso alle viste di Ordine e Account.
- **Staff** Ha accesso a tutto il sito, avendo 2 viste extra nascoste accessibili sotto al path `"/admin/"`. E' inteso per chi lavora in pizzeria. Devono essere creati dallo superuser nell'admin panel, il sito non permette di registrarsi con questo ruolo.
- **Admin** E' il superuser. Ha accesso all'admin panel generato da Django, dove puo' aggiungere/-togliere elementi dal menu' modificare prezzi, ecc. Ruolo inteso per i titolati della pizzeria.

Per creare il modello dell'utente ho esteso `AbstractUser`. Avevo la necessita' di inserire campi aggiuntivi come l'indirizzo e il numero di telefono, inoltre ho optato di utilizzare l'email come chiave primaria invece dell'username. Questa potra' essere utilizzata in futuro per inviare email di notifica sullo stato dell'ordine al cliente, inoltre sono certo che sia `Unique` anche al di fuori del mio Database cliente. Infine ritengo che avere sia email che username crei spesso confusione quindi ho deciso di eliminare completamente lo username.

```
1 class User(auth_models.AbstractUser):
2     email = models.CharField(max_length=255, unique=True)
3     password = models.CharField(max_length=255)
4     is_staff = models.BooleanField(default=False)
5     username = None
6     first_name = models.CharField(max_length=255)
7     last_name = models.CharField(max_length=255)
8     address = models.CharField(max_length=255)
9     cellphone_number = models.CharField(max_length=255)
10
11     objects = UserManager()
12
13     USERNAME_FIELD = "email"
14     REQUIRED_FIELDS = ["first_name", "last_name"]
15
16     def __str__(self):
17         return "Cliente " + self.first_name + " " + self.last_name
18
```

Siccome estendo la classe `AbstractUser`, ho dovuto implementare anche `UserManager`. Infine, in `settings.py` bisogna specificare: `AUTH_USER_MODEL = 'operations.User'`. Questo modello ha funzionato bene e non mi ha dato problemi. In precedenza avevo provato a creare una classe `Customer` che avesse una `OneToOneRelationship` con `User`, e aggiungere i campi aggiuntivi solamente in questa, ma ho riscontrato difficolta' nella creazione degli utenti. Per valutare i pro e contro ho consultato questo blog: [simpleisbetterthancomplex: how-to-extend-django-user-model](#), che ho trovato molto utile, nonostante non sia riuscito a far funzionare una delle soluzioni proposte.

Per l'autenticazione ho utilizzato Authtoken di Django con l'app Djoser, una implementazione REST del sistema di autenticazione di Django pensata per integrarsi meglio con l'architettura delle Single Page App. Funziona anche con un custom user model e genera una serie di endpoint.

Ad esempio una POST su `/auth/users/` crea l'utente, mentre una POST su `/auth/token/login/` restituisce un Token da inserire nell'header delle richieste HTTP. Una volta ottenuto il `auth_token` nel frontend, lo salvo nello store Vuex, nel localStorage e lo aggiungo all'header. A un refresh della pagina si cerca se il token e' gia' presente su localStorage, in caso positivo si popola subito l'header.

"Authorization": "Token {auth_token}"

Questo token viene gestito in automatico quando si effettuano chiamate HTTP. Tramite sessioni e middleware Django collega l'autenticazione nell'oggetto request. Si potra' quindi accedere all'utente della richiesta con un semplice `user = request.user`.

Ecco qui un piccolo snippet dimostrativo:

```
1 @api_view(['GET', 'POST'])
2 @permission_classes([IsAuthenticated])
3 def orders(request):
4     user = request.user
```

Purtroppo mi ero perso questo dettaglio leggendo la documentazione, e ci ho messo qualche ora a capire come funzionasse questo passaggio, ovvero come ottenere l'utente dato un token. Una volta capito, lo ho trovato molto comodo.

Con il decoratore `@permission_classes` si possono dichiarare le tipologie di utenti che possono avere accesso all'endpoint. Esempi sono: `AllowAny`, `IsAuthenticated`, `IsAuthenticatedOrReadOnly`, `IsAdminUser`. Si possono anche creare permessi custom.

Si puo' inoltre specificare nei settings.py la configurazione di default:

```
1 REST_FRAMEWORK = {
2     'DEFAULT_AUTHENTICATION_CLASSES': [
3         'rest_framework.authentication.TokenAuthentication',
4     ],
5     'DEFAULT_PERMISSION_CLASSES': [
6         'rest_framework.permissions.IsAuthenticated',
7     ]
8 }
```

4

Modelli

Ho diviso la logica sotto a 2 applicazioni: Menu e Operations.

L'app Menu contiene le seguenti classi:

- **Ingredient:** Rappresenta gli ingredienti da aggiungere sulla pizza: pomodoro, mozzarella, prosciutto crudo, ecc.
- **PizzaType:** Al momento contiene le tipologie Pizza, Calzone e Saltimbocca. Probabilmente sarebbe potuto essere un Enum ma così' ho fatto flessibilità nel caso in futuro si vogliano aggiungere altre tipologie come per esempio hot dog.
- **Dough** Indica l'impasto.
- **Size** Indica la grandezza della pizza.
- **MenuPizza** Indica le pizze del menu. Ha una ForeignKey a PizzaType, e un ManyToManyField per gli ingredienti.
- **CustomPizza** E' la pizza che ordinare effettivamente il cliente. Ha 3 ForeignKey per MenuPizza, Dough e Size, inoltre una ManyToMany per gli ingredienti aggiuntivi. Probabilmente rappresentare ogni pizza ordinata con una riga nuova nel DB non e' la soluzione piu' efficiente siccome avremo molti duplicati. Data la poca mole di dati non dovrebbe essere un problema per il database.
- **HalfMeterPizza** E' il mezzo metro, puo' avere fino a 3 gusti che sono rappresentati come ForeignKey a CustomPizza. Per creare un mezzo metro bisogna quindi prima creare fino a 3 CustomPizzas e poi referenziarle. Non ha la colonna Dough perche' non e' possibile ordinarlo con impasto speciale.
- **Beverage** Sono le bevande, hanno le flag is_in_stock, is_beer, is_alcoholic.

Nota: HalfMeterPizza e Beverage sono solo implementati come modello, non e' possibile ordinarli. Mancano i form e endpoint per l'API. Le flag is_vegetaria, is_spicy, is_lactose_free non sono ancora utilizzate. L'intenzione e' quella di mostrare queste flag affianco alle pizze tramite icone in futuro.

L'app Operations contiene le seguenti classi:

- **UserManager:** Per creare User e SuperUser
- **User:** Utente
- **Status** Rappresenta lo stato dell'ordine: 1 da completare, gli altri id successivi possono customizzati a piacere. (es. in preparazione, in forno, in consegna, consegnata, ecc.)
- **Order** Una delle classi chiave del database. Ha un User come ForeignKey e delle relazioni ManyToMany per CustomPizzas, HalfMeterPizzas e Beverages.
- **Review** Rappresenta la recensione, ha User come ForeignKey e Order come OneToOneField.

Penso che sarebbe stato piu' corretto avere un'app separate per la sola gestione di User, per avere una piu' netta separazione di responsabilita'.

Figure 4.1: UML generato da PyCharm - Complessivo

Figure 4.2: UML generato da PyCharm - Parte 1

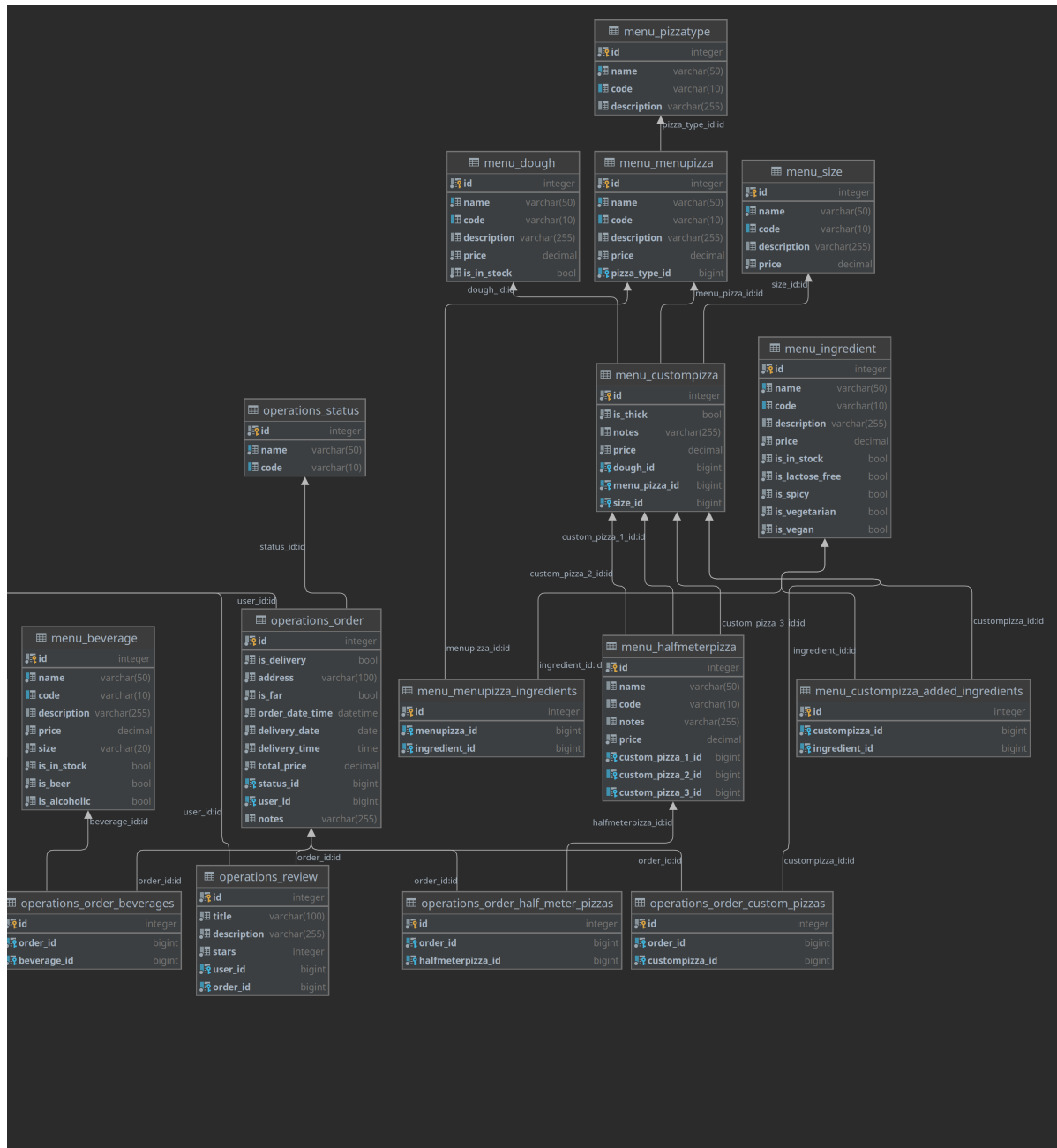


Figure 4.3: UML generato da PyCharm - Parte 2

5

Testing

Per la scrittura dei test ho utilizzato la libreria PyTest. I test sono nella cartella `/test/operations`, e ho dovuto inserire anche il file `pytest.ini` con la configurazione nella root del backend.

Ho effettuato 2 serie di test: utente e review.

I test dell'utente includono:

- **SignUp:** Si controlla tramite POST che l'utente venga creato correttamente.
- **Login Success:** Creato l'utente, si controlla che funzioni bene il log in con Djoser e che venga restituito un Auth Token.
- **Login Fail:** Si tenta il login con una password sbagliata e si controlla che il login fallisca.
- **Get Account:** Per controllare che i dati restituiti con la GET siano corretti e che l'utente creato non sia di tipo `is_staff`.
- **Logout:** Per controllare che il Logout funzioni correttamente. Dopo la POST si controlla per l'HTTP Status 204.

Tutti questi test vengono effettuati tramite chiamate all'API.

Per effettuare i test delle Review, che sono la parte finale del workflow, ho dovuto prima creare tutti gli oggetti necessari, ovvero: Dough, Size, Ingredients, PizzaType, MenuPizza, CustomPizza, Status. Inoltre ho creato un utente autenticato con il codice dei test di prima.

Sono stati poi testati tramite chiamata all'API:

- **Create Order:** Creazione di un ordine con una CustomPizza.
- **Create Review:** Creazione della recensione relativa all'ordine appena creato.
- **Update Review:** Modifica della recensione.
- **Delete Review:** Eliminazione della recensione.

Questi test sono stati fatti in un unico test: `test_review.py` per evitare di dover creare ogni volta tutti gli oggetti. Ho provato ad implementare le fixture ma ho avuto qualche problema quindi ho lasciato stare per ora, ma le voglio approfondire di nuovo in futuro.

E' la prima volta che scrivevo Unit Test ed inizialmente ero un po' spaventato. Si sono rivelati meno noiosi del previsto e soprattutto molto utili, mi hanno fatto trovare diversi piccoli bug che poi ho sistemato. Nei miei prossimi progetti cercherò di implementarli sin da subito, per evitare di costruire il progetto su basi non solide, seguendo la filosofia del Test Driven Development di cui ora comprendo l'utilità.


```
review_payload = dict(
    description="Review",
    order=1,
    stars=5,
    title="Title"
)
response = client.post("/api/v1/reviews/", review_payload, HTTP_AUTHORIZATION='Token {}'.format(auth_token))
assert response.status_code == 201

review_payload_update = dict(
    description="Updated Review",
    order=1,
    stars=5,
    title="Title"
)
response_update = client.put("/api/v1/reviews/1", review_payload_update, HTTP_AUTHORIZATION='Token {}'.format(auth_token))
assert response_update.status_code == 200

response_delete = client.delete("/api/v1/reviews/1", HTTP_AUTHORIZATION='Token {}'.format(auth_token))
assert response_delete.status_code == 204
test_create_review()

✓ Tests passed: 6 of 6 tests - 3 sec 376 ms

test_review.py::test_create_review
test_user.py::test_sign_up
test_user.py::test_log_in_user
test_user.py::test_log_in_user_fail
test_user.py::test_get_user_me
test_user.py::test_get_user_log_out

===== 6 passed, 1 warning in 4.64s =====

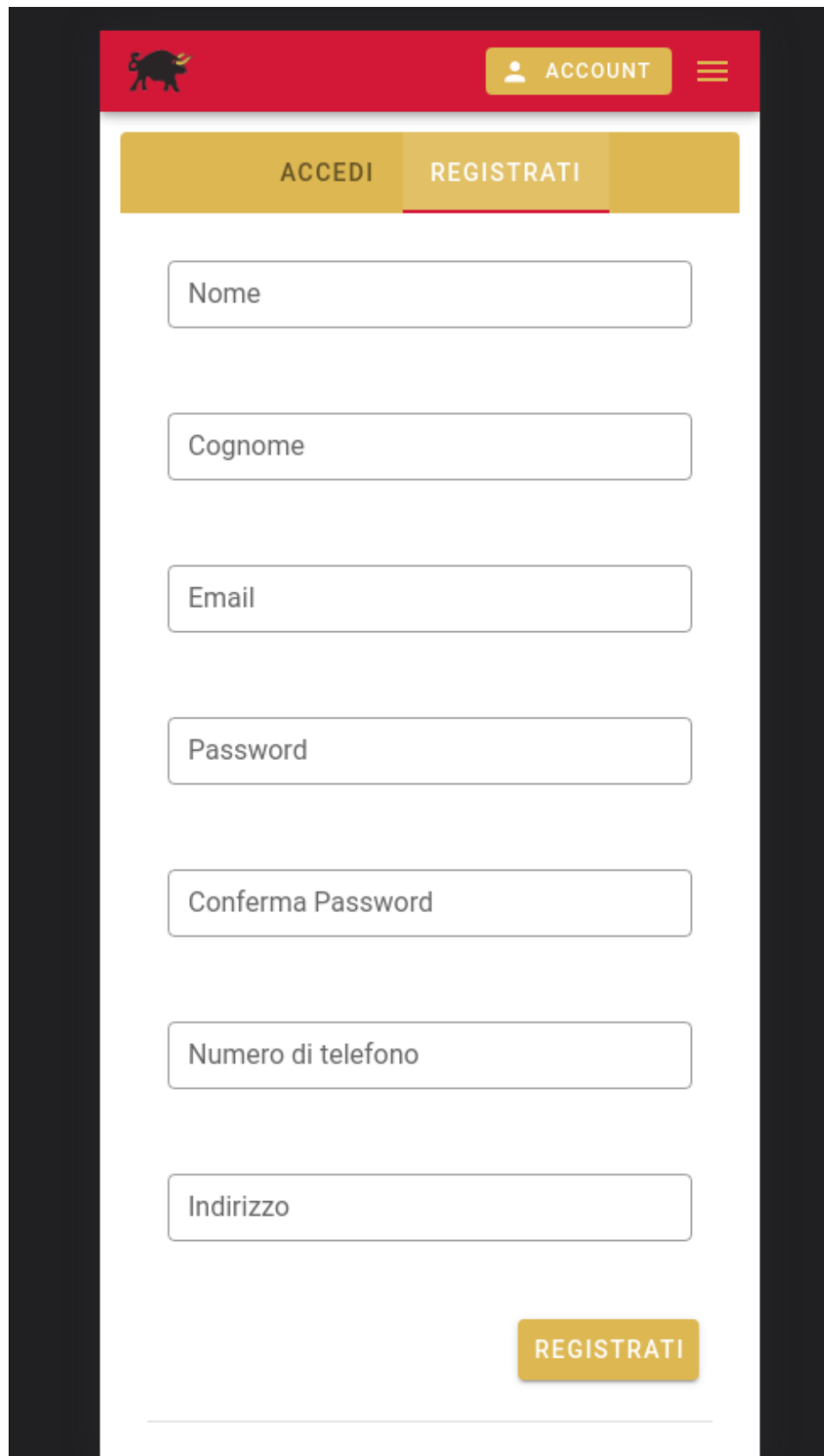
Process finished with exit code 0
```

Figure 5.1: Test PyTest su PyCharm

6

Risultati

Mostrero' qualche screenshot del sito nelle sue funzionalita' principali.



The image shows a mobile application interface for registration. At the top, there is a red header bar containing a black bull icon on the left, a yellow button with a person icon and the text "ACCOUNT" in the center, and a yellow menu icon on the right. Below the header, there is a yellow bar with two tabs: "ACCEDI" and "REGISTRATI". The "REGISTRATI" tab is selected, indicated by a red underline. Below the tabs, there are seven white input fields with rounded corners, each containing a label: "Nome", "Cognome", "Email", "Password", "Conferma Password", "Numero di telefono", and "Indirizzo". At the bottom right, there is a yellow button with the text "REGISTRATI".

ACCOUNT

ACCEDI REGISTRATI

Nome

Cognome

Email

Password

Conferma Password

Numero di telefono

Indirizzo

REGISTRATI

Figure 6.1: Registrazione da Mobile

Pizzeria Taurus HOME ABOUT MENU ORDINA ACCOUNT

Il mio Account

Email:

Nome:

Cognome:

Indirizzo:

Telefono:

LOG OUT

Lista ordini

Id ordine	Data consegna	Ora consegna	Indirizzo	Totale
24	2022-04-15	21:45:00	Via Brescia	18.00

Rows per page: 5 1-1 of 1

Recensioni

Lascia una recensione per un ordine

Id ordine:

Titolo recensione:

Modifica o Elimina Recensioni

Inserisci l'id dell'ordine della recensione che vuoi modificare

Id ordine:

Titolo recensione:

Figure 6.2: Account da Wide Monitor

Pizzeria Taurus HOME ABOUT MENU ORDINA ACCOUNT

Ordine

Tipologia:

Base:

Ingredienti base:

Ingredienti aggiuntivi:

Impasto:

Dimensione:

Note:

Prezzo: € 8

AGGIUNGI ALL'ORDINE

Carrello

Base	Impasto	Dimensione	Alta	Prezzo
Margherita	Classico	Normale	false	7

Rows per page: 5 1-1 of 1

Informazioni ordine

☒ Consegna a domicilio

Indirizzo:

2022 Sat, Apr 16 20:45

April 2022

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

45

Note:

Totale ordine: € 9

CONFERMA ORDINE

Lista ordini

Id ordine	Data consegna	Ora consegna	Indirizzo	Totale
-----------	---------------	--------------	-----------	--------

Recensioni

Lascia una recensione per un ordine

Figure 6.3: Ordina da Ultra-Wide Monitor

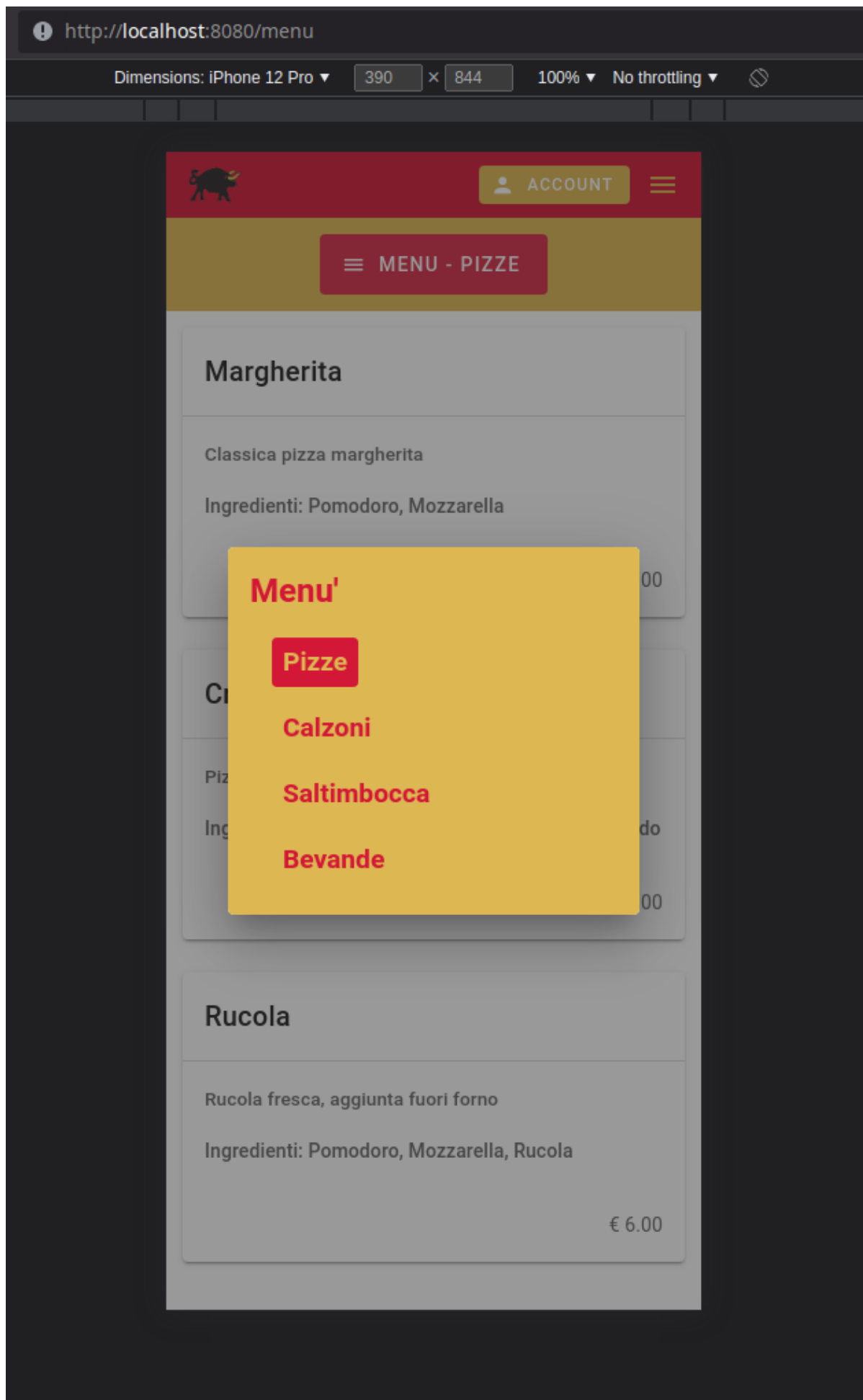


Figure 6.4: Menu da Mobile

7

Riflessioni

Ho riscontrato delle difficoltà nella serializzazione di oggetti complessi, ovvero quelli con relazioni ManyToMany e ForeignKey. A volte una POST con un id di una ForeignKey provava a creare un nuovo oggetto, anche se la ForeignKey esisteva già nel Database, altre volte bisognava estrarre gli id delle foreign key e crearsi l'istanza dell'oggetto in memoria, per poi assegnarla.

La documentazione è molto vasta ed è facile perdersi dettagli importanti, ritengo anche che informazioni collegate siano spesso sparse in sezioni separate quando dovrebbero essere accanto. Inoltre ci sono spesso diversi modi per ottenere lo stesso risultato desiderato, quindi mi sono ritrovato spesso a cercare errori su StackOverflow con pochissimi upvote, Blog e Forum per cercare di capire quale fosse quello giusto da utilizzare per il mio caso. Inoltre ho notato che spesso la documentazione ha esempi molto basilari, e le function based views venivano spesso escluse o messe in secondo piano.

Per andare nel concreto, nella serializzazione del POST di un ordine, ho dovuto gestire a mano molte cose che davo per scontate:

```
1
2 @api_view(['GET', 'POST'])
3 @permission_classes([IsAuthenticated])
4 def orders(request):
5
6     ....
7
8     elif request.method == 'POST':
9
10         user = request.user
11
12         date_arr = request.data["delivery_date"].split("-")
13         delivery_date = datetime.date(int(date_arr[0]), int(date_arr[1]), int(date_arr[2]))
14
15         time_arr = request.data["delivery_time"].split(":")
16         delivery_time = datetime.time(int(time_arr[0]), int(time_arr[1]), 0)
17
18         custom_pizzas = request.data.get('custom_pizzas', None)
19         half_meter_pizzas = request.data.get('half_meter_pizzas', None)
20         beverages = request.data.get('beverages', None)
21
22         is_delivery = request.data["is_delivery"]
23         total_price = float(request.data["total_price"])
24
25         status_instance = Status.objects.get(id=1)
26
27         order = Order.objects.create(user=user,
28                                     delivery_date=delivery_date,
29                                     delivery_time=delivery_time,
30                                     total_price=total_price,
31                                     status=status_instance,
```

```
32         is_delivery=is_delivery
33     )
34
35     if custom_pizzas:
36         order.custom_pizzas.set(custom_pizzas)
37     if half_meter_pizzas:
38         order.half_meter_pizzas.set(half_meter_pizzas)
39     if beverages:
40         order.beverages.set(beverages)
41
42     if total_price != order.calculate_total_price():
43         return Response(status=status.HTTP_406_NOT_ACCEPTABLE)
44
45     serializer = OrderSerializer(order, data=request.data)
46     if serializer.is_valid():
47         serializer.save()
48         return Response(serializer.data, status=status.HTTP_201_CREATED)
49     else:
50         return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Probabilmente ci sono soluzioni piu' semplici, ma risolvendo i vari errori uno a uno mi sono ritrovato con questo codice che non e' molto elegante. In generale non mi e' stato mai troppo chiaro quando una funzionalita' non era supportata di default da DRF e dovevo implementarla io con altri metodi o override. Faccio ancora fatica a capire dove sia questa linea.

Nonostante questi problemi sulla serializzazione e quelli menzionati prima sulla creazione dell'utente con i l'autenticazione, mi sono trovato bene con questo framework. L'ORM e' stato ottimo per le mie esigenze, anche se ho sentito che si complica per query complesse. Inoltre l'app dell'admin e' stata critica per iniziare il progetto in fretta, e non ha avuto alcun conflitto con altre funzionalita'. I messaggi di errore erano spesso chiari. Essendo il primo framework web che abbia utilizzato faccio fatica a dare un giudizio accurato, ma tutto sommato devo dire che mi e' piaciuto. Anche se in futuro probabilmente cercherei di iniziare un progetto in un linguaggio staticamente tipizzato, siccome durante questo sviluppo devo ammettere che ne ho sentito la mancanza.

Per concludere, nella progettazione di questo progetto mi sono divertito molto e continuero' ad aggiungere funzionalita' al sito per portarlo in produzione.