Bruno Manuel Quintana
CS5722 - University of Limerick
Student Id: 20225547

This software is an improved version of the
initial project submitted to CS5721.

Project Description **5**
Project Overview **5**
Business Rules **5**
Technical Implementation **6**
Use of framework 6
Libraries used 6
Django-resized 6
Pytest 6
Pytest-mock 6
Djangorestframework 6
Use as Framework **7**
Framework Repository 7
Application Repository 7
Use of REST API **8**
List of endpoints 9
Hotel Endpoint Demonstration 9
View from self-hosted http client 10
Bad Code Smells **11**
Bloaters 11
Object-Orientation Abusers 11
Change Preventers 11
Dispensables 11
Couplers 12
Refactorings Applied 12
Code **13**
Design Pattern 13
Dependency Injection 13
Theory 13
Code Example 13
ORM 14
Code Example 14
Behavioral Patterns 15
Template Method 15
Business Requirement 15
Theory 15
Implementation 16
Code 17
Command 18
Theory 18
Implementation 18
Code 18
Memento - Backend - Python 19

Business Requirement 19
Theory 19
Implementation 20
Code 20
Unit Test 21
Memento - Frontend - Javascript 21
Business Requirement 21
Implementation 22
Code 22
The visitor 23
Business Requirement 23
Theory 23
Implementation 23
Code 23
Architectural Patterns 26
The interceptor 26
Business Requirements 26
Theory 26
Implementation 26
Request-Response 28
Theory 28
Implementation 28
Code 29
Creational Patterns 31
The Factory Method 31
Structural Patterns 32
Composite 32
Business Requirements 32
Theory 32
Implementation 32
Code 32
Bridge 33
Business Requirement 34
Theory 34
Implementation 34
Code 34
Proxy 39
Code Metrics 40
Non-Functional requirements 41
Security: 41
Cross site scripting (XSS) protection 41
Cross site request forgery (CSRF) protection 41
SQL injection protection 41
SSL/HTTPS 41

Extensibility **42**
Application 42
Performance **45**
Same as the original project 45
Future Development **45**
DevOps **45**
Local Development 46
Deploy Production 47
Unit Test **47**
Known Issues **48**
References **49**

Project Description

The Hotel Management System was initially developed with the idea of being used by one hotel in Kerry, known as the client. The client grew and the requirements have changed.

The new version which consists of all features previously requested, will be improved by using Design Patterns, it will also require a REST API for partner companies such as bookings.com to be able to make instant reservations. Kerry Hotel will franchise its brand as an expanding strategy, therefore other features will be refactored so that franchisees can use the system as a framework as a starting point, but they will be extended as needed since it will be independently run.

Project Overview

The family friendly Central Hotel is situated in the center of Lis towel, in Co Kerry. As the Central Hotel strives to improve their customer service and as part of an overall investment in the business. They have secured an investment to expand the hotel brand into new franchise branches that will be individually owned. Each branch will have its own cloud servers but the data will not be shared between hotels, and therefore each branch will technologically operate individually.

The Central Hotel has also received a COVID-19 relief that will allow them to invest in online marketing, in the form of a featured hotel in Bookings.com. In order to accept instant bookings from external systems, it needs the design and implementation of a secured REST API.

Services of Dynamic Solutions will refactor the previous version into a framework that can be easily extended to suit these new requirements.

Business Rules

The system continues to be flexible in terms of business rules. Every hotel is able to define its own structure, number of rooms, type of rooms, etc.

Technical Implementation

Use of framework

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development.

Libraries used

External libraries have been used to full-fill this project some of the most important are:

Django-resized

Resizes image origin to specific size. It is used when pictures of rooms are uploaded to the system.

Pytest

The pytest framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

Pytest-mock

This plugin provides a mocker fixture which is a thin-wrapper around the patching API provided by the mock package.

Djangorestframework

Django REST framework is a powerful and flexible toolkit for building Web APIs.

Use as Framework

The main project has been moved to a new repository. This repository will be “the framework”, while each hotel will install on their cloud servers “the application”, which uses the framework as an external dependency.

Framework Repository

https://github.com/bruno911/hms-framework

Application Repository

https://github.com/bruno911/hms-application

The application will extend the framework. Each branch will have its own custom application that can be changed extensible.

The framework remains in active development, each branch will be able to upgrade by changing the used version from requirements.txt. By using the default dependency manager from Python: “Pip”, this can be easily achieved. For ease of demonstration, we will use a github repository as our “dependency repository”, when deploying production the recommended way is to either as open source and publicly available to everyone at https://pypi.org/ or else as a private repository in github and set up the private keys.

Each framework release will be shipped with a change log list, new features and compatibility issues.

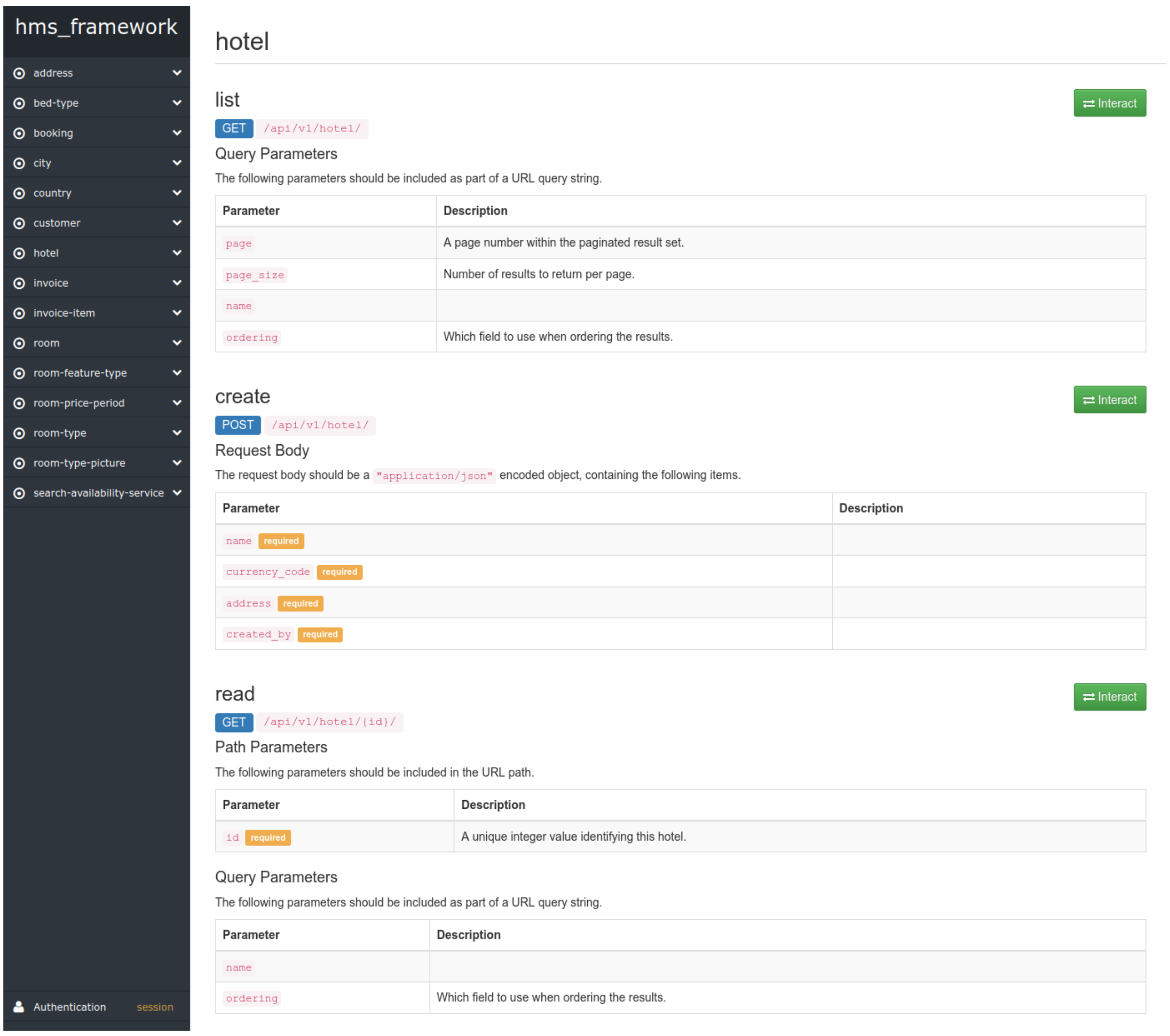
Use of REST API

A full REST Api has been implemented, it can be either used by Hotel branches who may want to fully implement their own UI, or for external companies such as Booking.com to make instant bookings by integrating. The API is protected by a token, permissions and groups can be set from the admin panel.

The full code for the REST API can be seen at:
https://github.com/bruno911/hms-framework/commit/a259f84e8720d52db6ce26b7b8ccf3d5d4da124e

The full unit tests for the REST API can be seen at:
https://github.com/bruno911/hms-framework/commit/eae6593c69d7335aff081c65782ed3524e0d4b05

The full API documentation can be seen at:
http://127.0.0.1:8000/api/docs/



hotel

list

GET `/api/v1/hotel/`

Query Parameters

The following parameters should be included as part of a URL query string.

Parameter	Description
page	A page number within the paginated result set.
page_size	Number of results to return per page.
name	
ordering	Which field to use when ordering the results.

create

POST `/api/v1/hotel/`

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
name required	
currency_code required	
address required	
created_by required	

read

GET `/api/v1/hotel/{id}/`

Path Parameters

The following parameters should be included in the URL path.

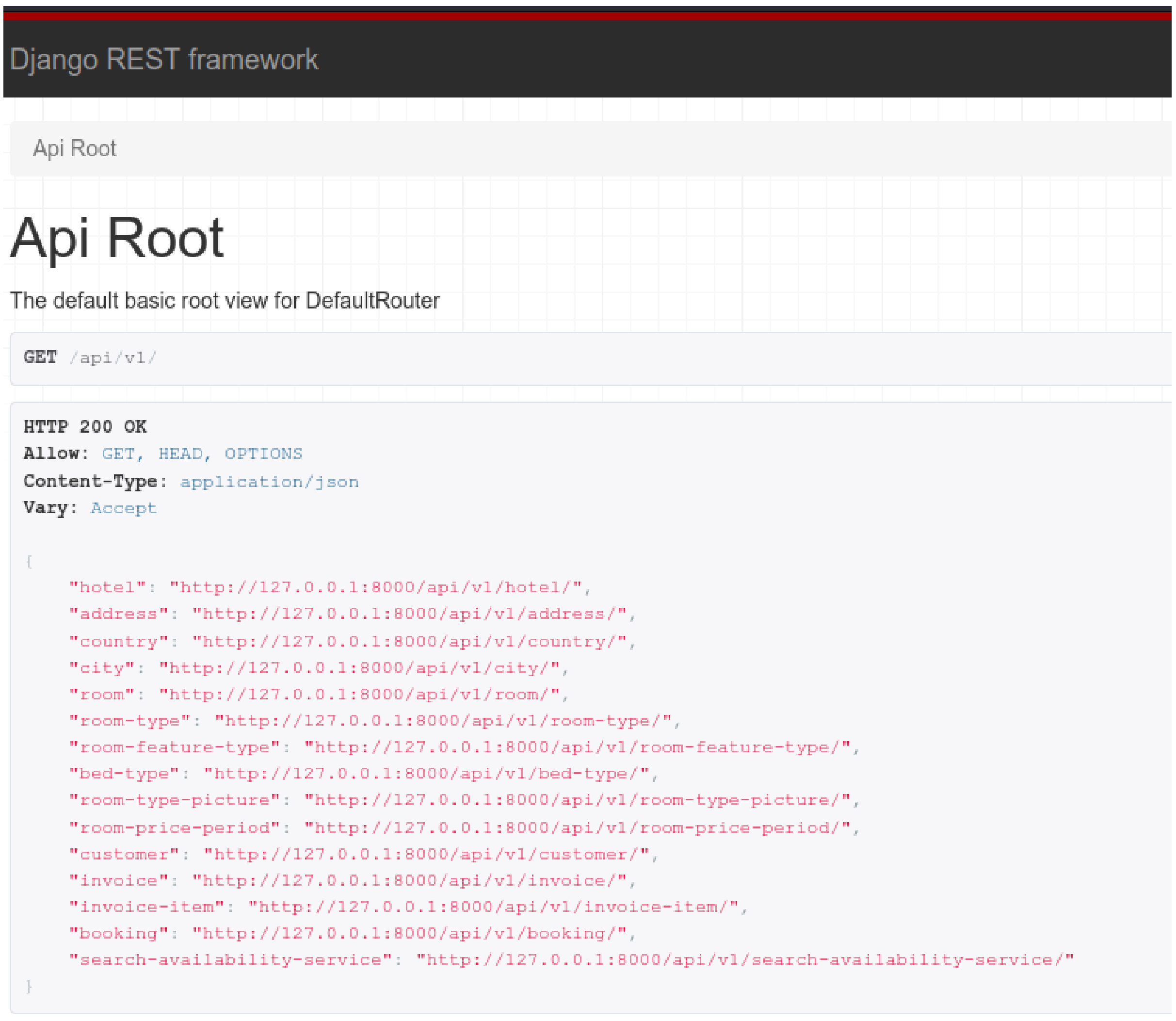
Parameter	Description
id required	A unique integer value identifying this hotel.

Query Parameters

The following parameters should be included as part of a URL query string.

Parameter	Description
name	
ordering	Which field to use when ordering the results.

List of endpoints


Django REST framework

Api Root

Api Root

The default basic root view for DefaultRouter

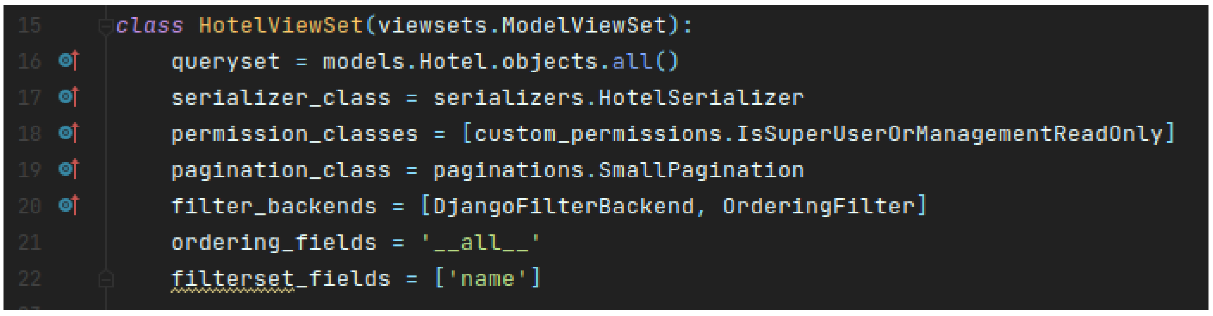
```
GET /api/v1/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "hotel": "http://127.0.0.1:8000/api/v1/hotel/",
    "address": "http://127.0.0.1:8000/api/v1/address/",
    "country": "http://127.0.0.1:8000/api/v1/country/",
    "city": "http://127.0.0.1:8000/api/v1/city/",
    "room": "http://127.0.0.1:8000/api/v1/room/",
    "room-type": "http://127.0.0.1:8000/api/v1/room-type/",
    "room-feature-type": "http://127.0.0.1:8000/api/v1/room-feature-type/",
    "bed-type": "http://127.0.0.1:8000/api/v1/bed-type/",
    "room-type-picture": "http://127.0.0.1:8000/api/v1/room-type-picture/",
    "room-price-period": "http://127.0.0.1:8000/api/v1/room-price-period/",
    "customer": "http://127.0.0.1:8000/api/v1/customer/",
    "invoice": "http://127.0.0.1:8000/api/v1/invoice/",
    "invoice-item": "http://127.0.0.1:8000/api/v1/invoice-item/",
    "booking": "http://127.0.0.1:8000/api/v1/booking/",
    "search-availability-service": "http://127.0.0.1:8000/api/v1/search-availability-service/"
}
```

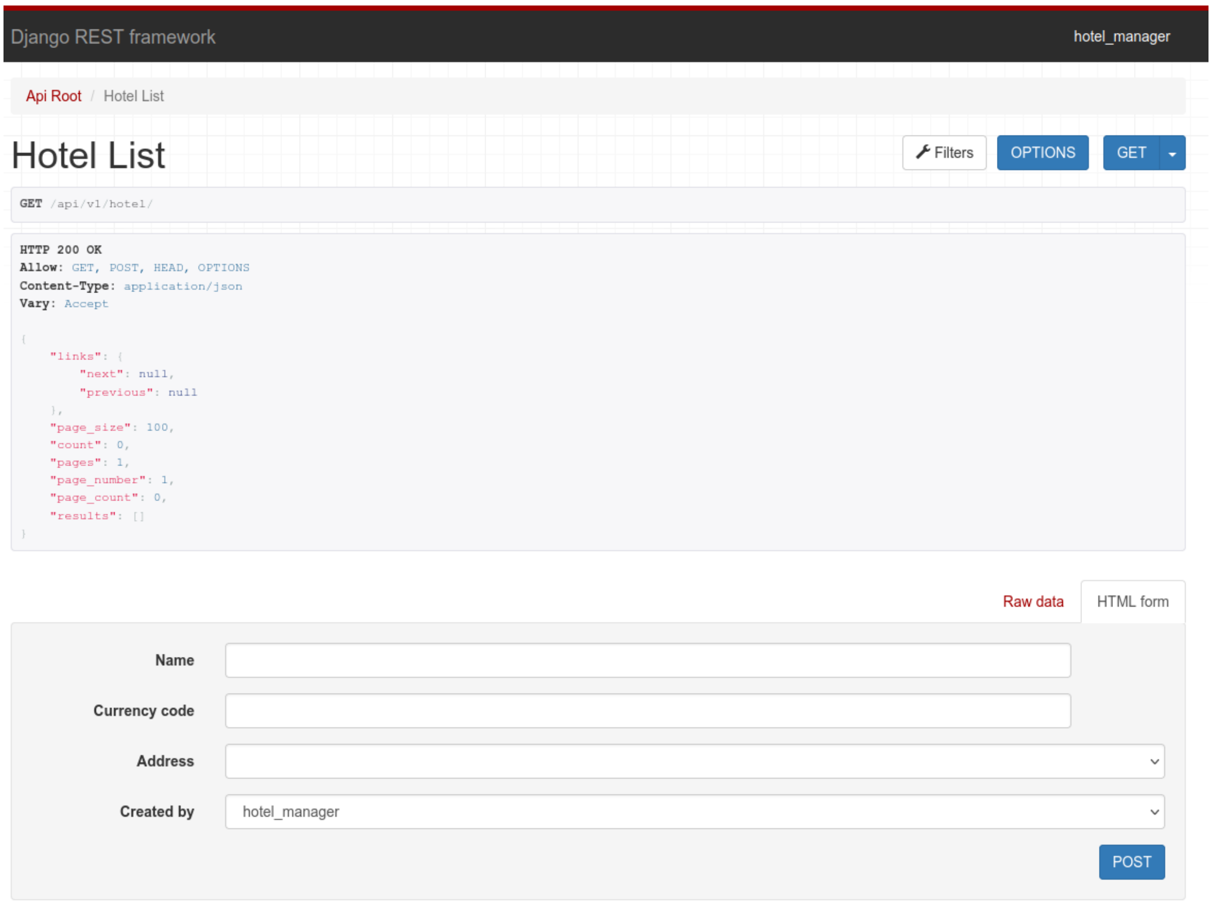
Hotel Endpoint Demonstration

please note more complex examples are available on the commit above.

Endpoints and HTTP Options
POST http://127.0.0.1:8000/api/v1/hotel/
GET http://127.0.0.1:8000/api/v1/hotel/{id}
PUT http://127.0.0.1:8000/api/v1/hotel/{id}
DELETE http://127.0.0.1:8000/api/v1/hotel/{id} (not-allowed)


```python
class HotelViewSet(viewsets.ModelViewSet):
    queryset = models.Hotel.objects.all()
    serializer_class = serializers.HotelSerializer
    permission_classes = [custom_permissions.IsSuperUserOrManagementReadOnly]
    pagination_class = paginations.SmallPagination
    filter_backends = [DjangoFilterBackend, OrderingFilter]
    ordering_fields = '__all__'
    filterset_fields = ['name']
```

View from self-hosted http client



Hotel List

```
GET /api/v1/hotel/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "links": {
        "next": null,
        "previous": null
    },
    "page_size": 100,
    "count": 0,
    "pages": 1,
    "page_number": 1,
    "page_count": 0,
    "results": []
}
```

Name	
Currency code	
Address	
Created by	hotel_manager

Bad Code Smells

A check list of the possible bad code smells has been followed to ensure quality.

Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).(refactoring.guru)

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Not Detected

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.(refactoring.guru)

- Alternative Classes with Different Interfaces
- Refused Bequest
- Switch Statements
- Temporary Field

Not Detected

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.(refactoring.guru)

- Divergent Change
- Parallel Inheritance Hierarchies
- Shotgun Surgery

Not Detected

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.(refactoring.guru)

- Comments
- Duplicate Code
- Data Class

- Dead Code
- Lazy Class
- Speculative Generality

Not Detected

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.(refactoring.guru)

- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Message Chains
- Middle Man

Not Detected

Refactorings Applied

The main refactoring has been to move all controller’s actions into Commands.

The controller's actions with too many parameters has been replaced with a Command that takes a Request parameter and responds with a response class.

There has also been the implementation of design patterns as described above.

Code

Design Pattern

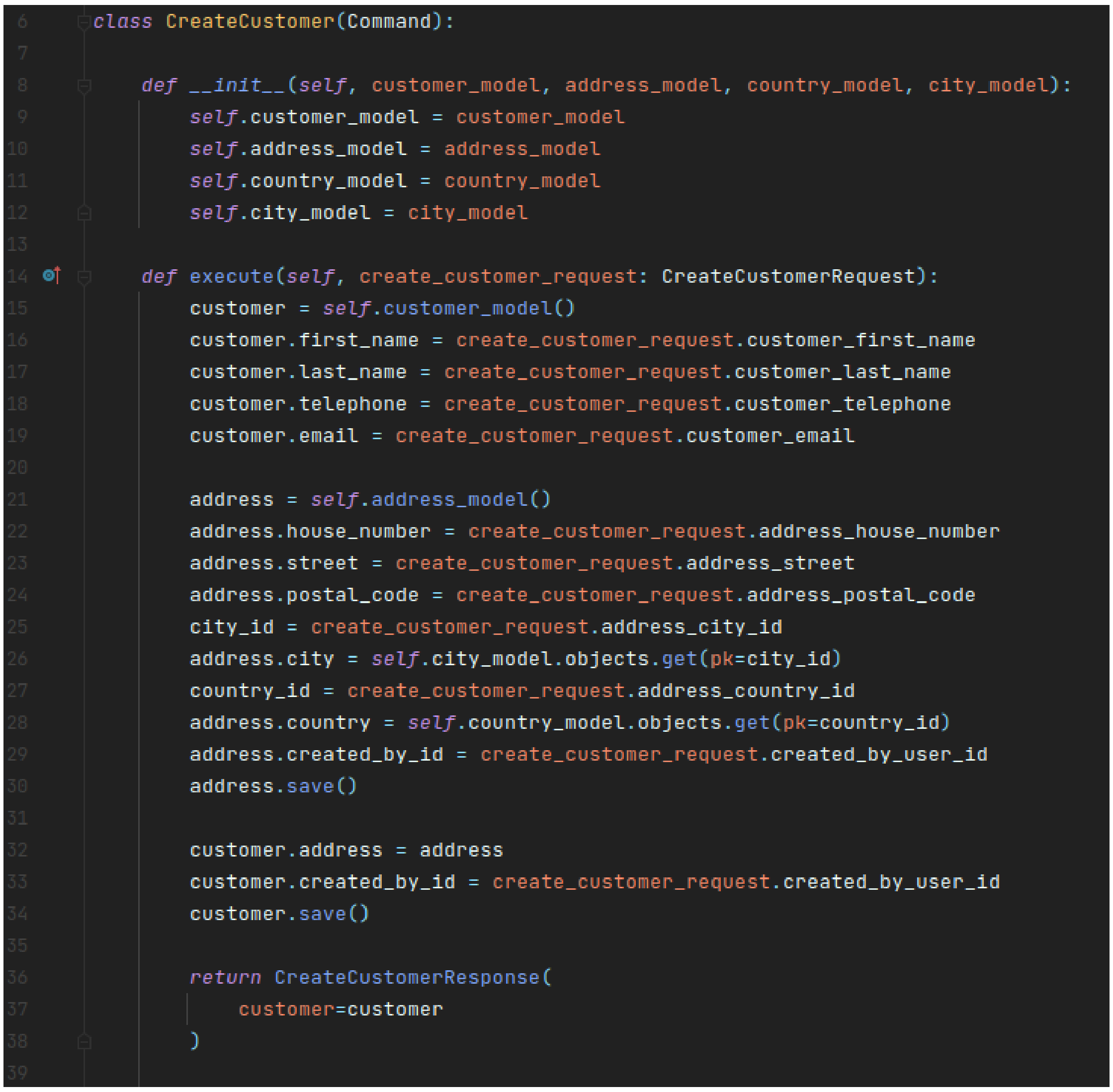
Dependency Injection

Theory

Dependency injection pattern got popular in the languages with static typing, like Java. Dependency injection is a principle that helps to achieve an inversion of control. (Dependency injection and inversion of control in Python)

Python is an interpreted language with dynamic typing. There is an opinion that dependency injection doesn't work for it as well as it does for Java. A lot of the flexibility is already built in. Also there is an opinion that a dependency injection framework is something that Python developers rarely need. Python developers say that dependency injection can be implemented easily using language fundamentals.(Dependency injection and inversion of control in Python)

Code Example



```python
class CreateCustomer(Command):

    def __init__(self, customer_model, address_model, country_model, city_model):
        self.customer_model = customer_model
        self.address_model = address_model
        self.country_model = country_model
        self.city_model = city_model

    def execute(self, create_customer_request: CreateCustomerRequest):
        customer = self.customer_model()
        customer.first_name = create_customer_request.customer_first_name
        customer.last_name = create_customer_request.customer_last_name
        customer.telephone = create_customer_request.customer_telephone
        customer.email = create_customer_request.customer_email

        address = self.address_model()
        address.house_number = create_customer_request.address_house_number
        address.street = create_customer_request.address_street
        address.postal_code = create_customer_request.address_postal_code
        city_id = create_customer_request.address_city_id
        address.city = self.city_model.objects.get(pk=city_id)
        country_id = create_customer_request.address_country_id
        address.country = self.country_model.objects.get(pk=country_id)
        address.created_by_id = create_customer_request.created_by_user_id
        address.save()

        customer.address = address
        customer.created_by_id = create_customer_request.created_by_user_id
        customer.save()

        return CreateCustomerResponse(
            customer=customer
        )
```

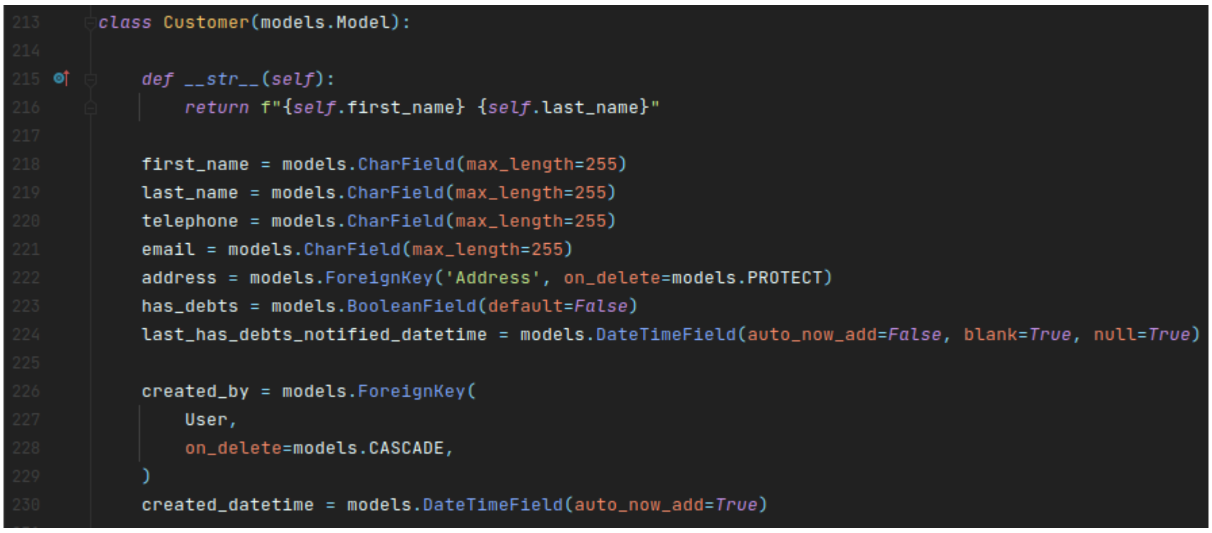
The construct takes the ORM models as parameters, for then being used by the execute method. This allows ORM models to be replaced and gives flexibility. One usage of this pattern is usually when you want to decouple the business logic from the framework, if during future developments we want to change the ORM library or Framework we will only update the factories or the Dependency Injection Config file, which is usually a mapping of parameters with classes.

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.(Quick start with Django ORM)

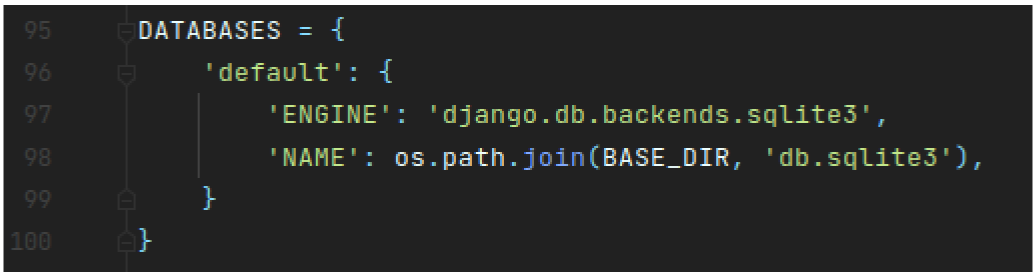
This allows the developers to use a common syntax independently of the relational database behind it, and also facilitates the migration of databases. With today's cloud services, it is a common practice to switch from MySql to for example AWS Aurora to improve performance, reduce maintenance and/or reduce costs.

Code Example



```python
class Customer(models.Model):

    def __str__(self):
        return f"{self.first_name} {self.last_name}"

    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    telephone = models.CharField(max_length=255)
    email = models.CharField(max_length=255)
    address = models.ForeignKey('Address', on_delete=models.PROTECT)
    has_debts = models.BooleanField(default=False)
    last_has_debts_notified_datetime = models.DateTimeField(auto_now_add=False, blank=True, null=True)

    created_by = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
    )
    created_datetime = models.DateTimeField(auto_now_add=True)
```

In this example, we defined a Customer entity, the way we want it to be, however we are not saying if this is MySql, Postgresql or any other database. The database becomes a minor detail. The ORM takes care about it based on the database config definition (settings.py):



```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

By executing the code:

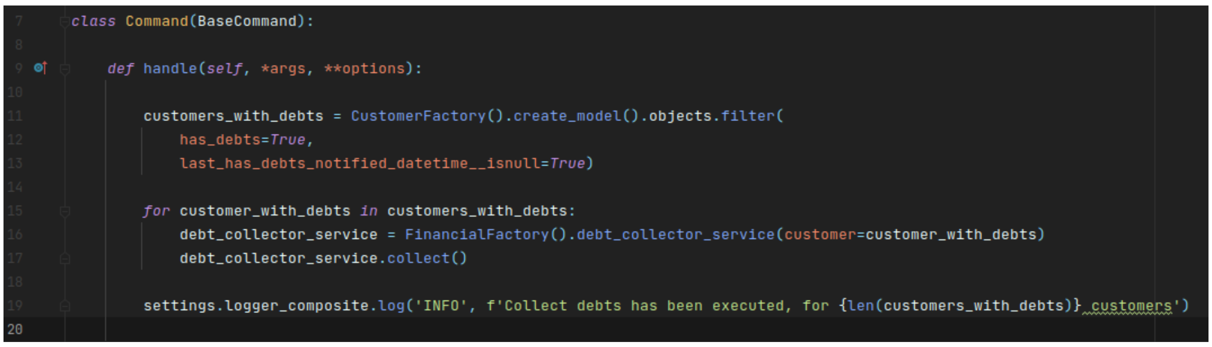
```
python manage.py makemigrations
```

The ORM will create a script to build the database, every change to a model requires a new `makemigrations`. These scripts to build the database are then deployed and they have the history of how the database has been evolving. By executing:

```
python manage.py migrations
```

Changes are made persistent to the database.

Instead of having to write plain queries, ORM has its own syntax to do it:



```python
class Command(BaseCommand):

    def handle(self, *args, **options):

        customers_with_debts = CustomerFactory().create_model().objects.filter(
            has_debts=True,
            last_has_debts_notified_datetime__isnull=True)

        for customer_with_debts in customers_with_debts:
            debt_collector_service = FinancialFactory().debt_collector_service(customer=customer_with_debts)
            debt_collector_service.collect()

        settings.logger_composite.log('INFO', f'Collect debts has been executed, for {len(customers_with_debts)} customers')
```

On this command line script, we use a Factory to get the Customer ORM model, and we say to filter by has_debts=True which is the equivalent in MySql to:

```
select * from customer where has_debts=1;
```

We also filter by the last_has_debts_notified_datetime__isnull=True which is the equivalent in MySql to:

```
select * from customer where last_has_debts_notified is null;
```

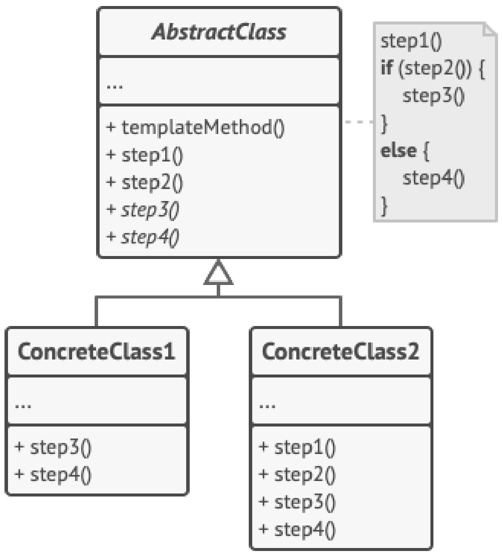
Behavioral Patterns

Template Method

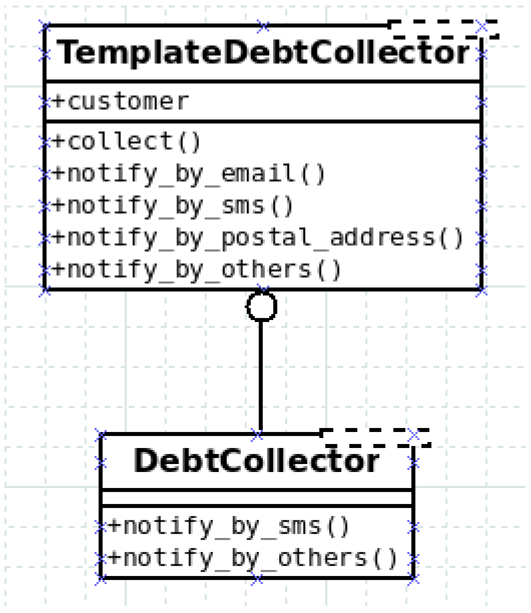
Business Requirement

During the Debt collection, by default the guest will be notified by Email, SMS, Postal Letter.
The HMS framework will allow developers to implement all of them, override these with its own email, sms and postal letter API providers, and we also leave it open to an extra step called “Other” so that if for example a hotel prefers to notify by WhatsApp using Facebook API it can. This pattern allows us to do it in a clean way.

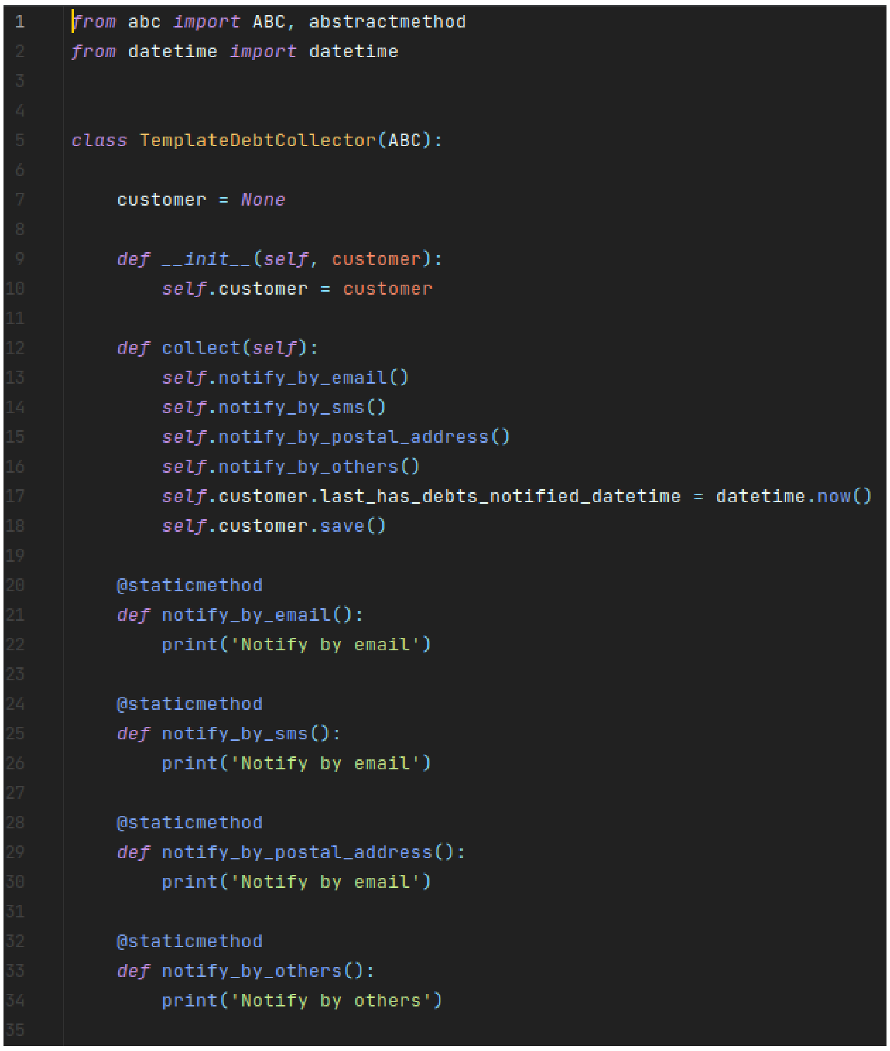
Theory



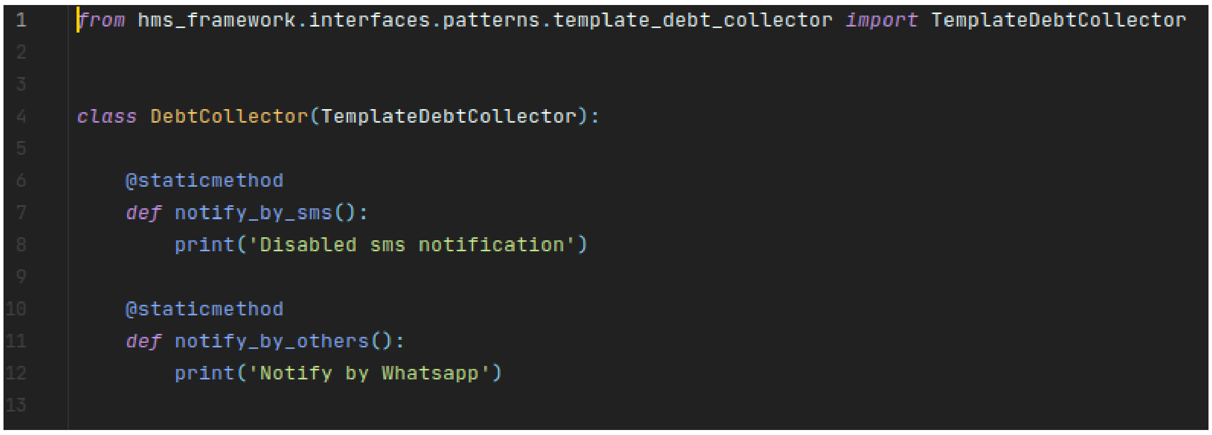
Implementation



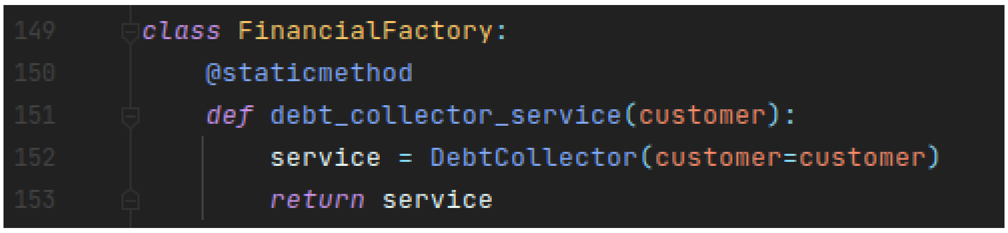
Code



```python
from abc import ABC, abstractmethod
from datetime import datetime


class TemplateDebtCollector(ABC):

    customer = None

    def __init__(self, customer):
        self.customer = customer

    def collect(self):
        self.notify_by_email()
        self.notify_by_sms()
        self.notify_by_postal_address()
        self.notify_by_others()
        self.customer.last_has_debts_notified_datetime = datetime.now()
        self.customer.save()

    @staticmethod
    def notify_by_email():
        print('Notify by email')

    @staticmethod
    def notify_by_sms():
        print('Notify by email')

    @staticmethod
    def notify_by_postal_address():
        print('Notify by email')

    @staticmethod
    def notify_by_others():
        print('Notify by others')

```



```python
from hms_framework.interfaces.patterns.template_debt_collector import TemplateDebtCollector


class DebtCollector(TemplateDebtCollector):

    @staticmethod
    def notify_by_sms():
        print('Disabled sms notification')

    @staticmethod
    def notify_by_others():
        print('Notify by Whatsapp')

```

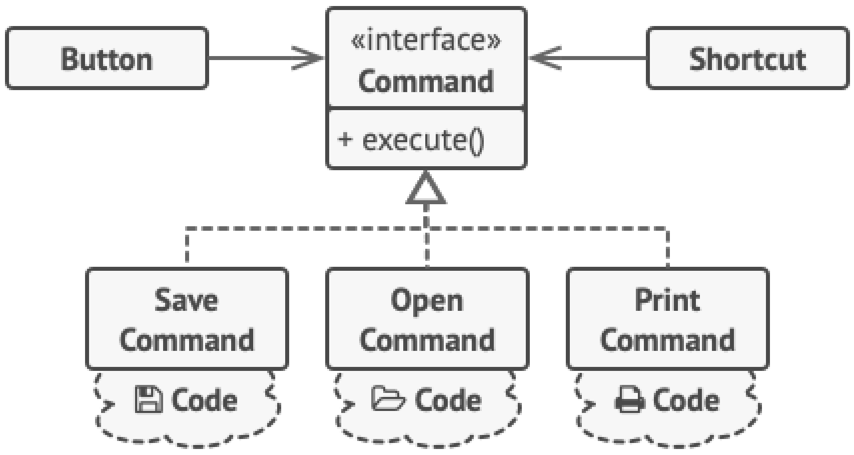


```python
class FinancialFactory:
    @staticmethod
    def debt_collector_service(customer):
        service = DebtCollector(customer=customer)
        return service
```

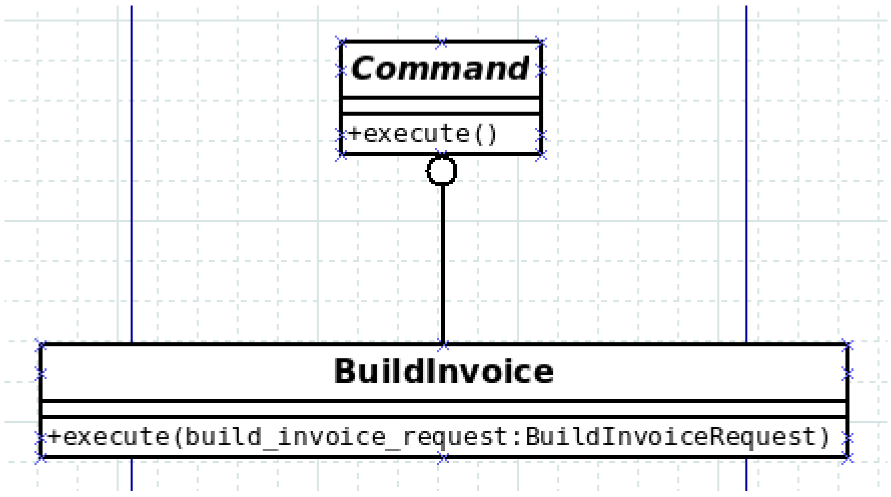
Command

The command pattern will encapsulate a behaviour so that the same logic could be reused. Since the logic will now be accessed via REST API but also via Django View, the only difference will be how the parameters are collected and how the response is returned.
A Django view will return a parsed html, while REST API will return a json and an http status code.
Commands can then be used via command line if they need to be executed via crontab as a scheduled task without much effort.

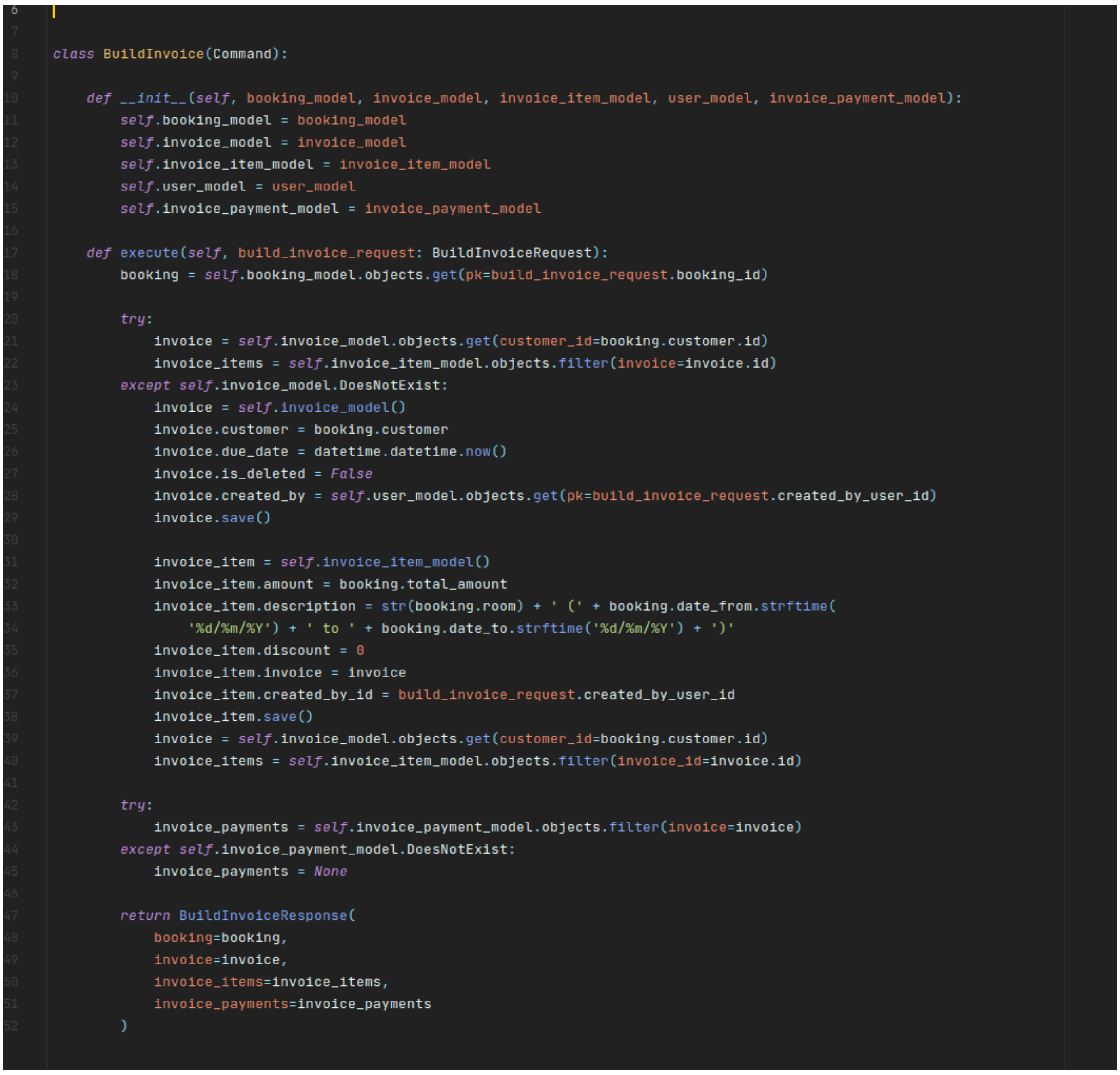
Theory



Implementation



Code



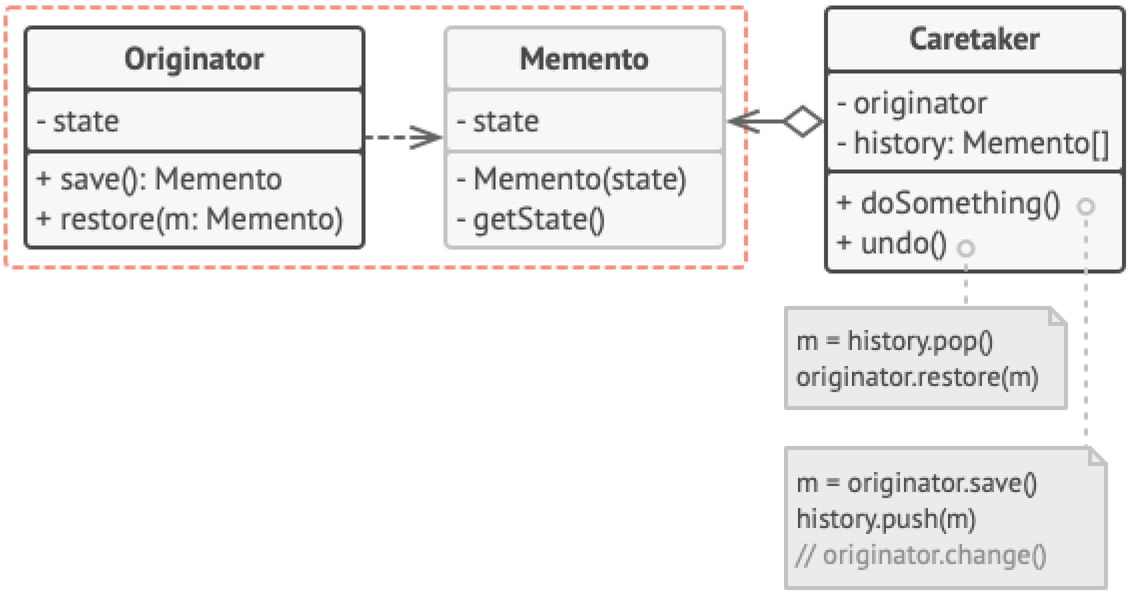
```python
class BuildInvoice(Command):

    def __init__(self, booking_model, invoice_model, invoice_item_model, user_model, invoice_payment_model):
        self.booking_model = booking_model
        self.invoice_model = invoice_model
        self.invoice_item_model = invoice_item_model
        self.user_model = user_model
        self.invoice_payment_model = invoice_payment_model

    def execute(self, build_invoice_request: BuildInvoiceRequest):
        booking = self.booking_model.objects.get(pk=build_invoice_request.booking_id)

        try:
            invoice = self.invoice_model.objects.get(customer_id=booking.customer.id)
            invoice_items = self.invoice_item_model.objects.filter(invoice=invoice.id)
        except self.invoice_model.DoesNotExist:
            invoice = self.invoice_model()
            invoice.customer = booking.customer
            invoice.due_date = datetime.datetime.now()
            invoice.is_deleted = False
            invoice.created_by = self.user_model.objects.get(pk=build_invoice_request.created_by_user_id)
            invoice.save()

            invoice_item = self.invoice_item_model()
            invoice_item.amount = booking.total_amount
            invoice_item.description = str(booking.room) + ' (' + booking.date_from.strftime(
                '%d/%m/%Y') + ' to ' + booking.date_to.strftime('%d/%m/%Y') + ')'
            invoice_item.discount = 0
            invoice_item.invoice = invoice
            invoice_item.created_by_id = build_invoice_request.created_by_user_id
            invoice_item.save()
            invoice = self.invoice_model.objects.get(customer_id=booking.customer.id)
            invoice_items = self.invoice_item_model.objects.filter(invoice_id=invoice.id)

        try:
            invoice_payments = self.invoice_payment_model.objects.filter(invoice=invoice)
        except self.invoice_payment_model.DoesNotExist:
            invoice_payments = None

        return BuildInvoiceResponse(
            booking=booking,
            invoice=invoice,
            invoice_items=invoice_items,
            invoice_payments=invoice_payments
        )
```

Memento - Backend - Python

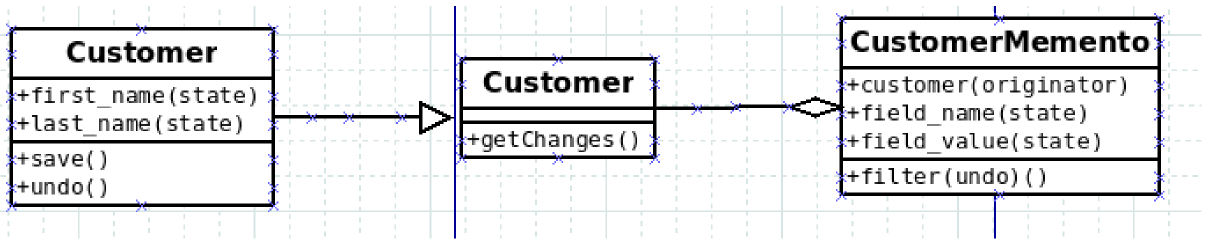
Business Requirement

The Memento pattern will be used to undo personal guests information. Due to debt collection, if a person decides from a partner's website i.e. bookings.com to change his contact details information, we want to keep the state of previous details in order to be able to rollback the modifications. For now we will only store the data, future business requirements may request to allow rollback via API or only via UI.

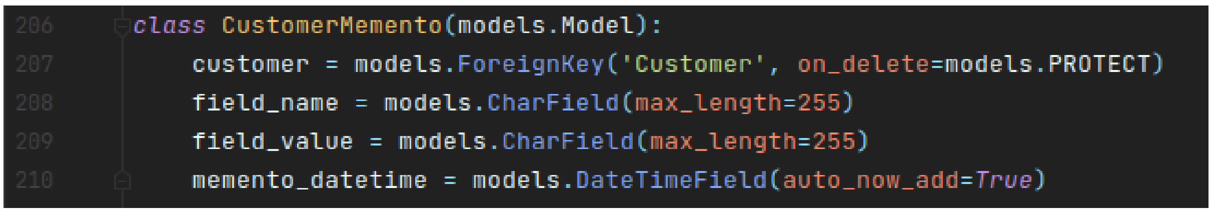
Theory



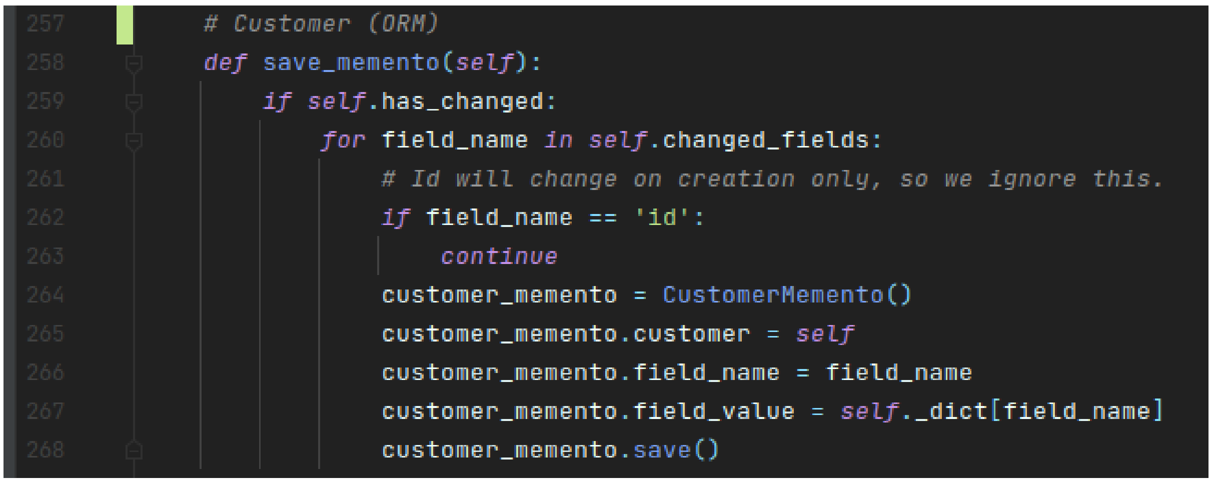
Implementation



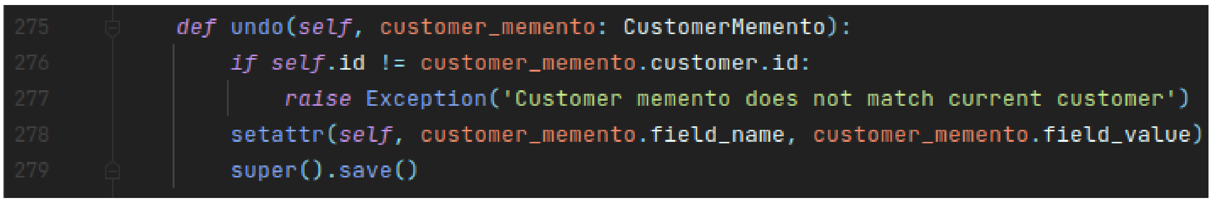
Code



```python
class CustomerMemento(models.Model):
    customer = models.ForeignKey('Customer', on_delete=models.PROTECT)
    field_name = models.CharField(max_length=255)
    field_value = models.CharField(max_length=255)
    memento_datetime = models.DateTimeField(auto_now_add=True)
```

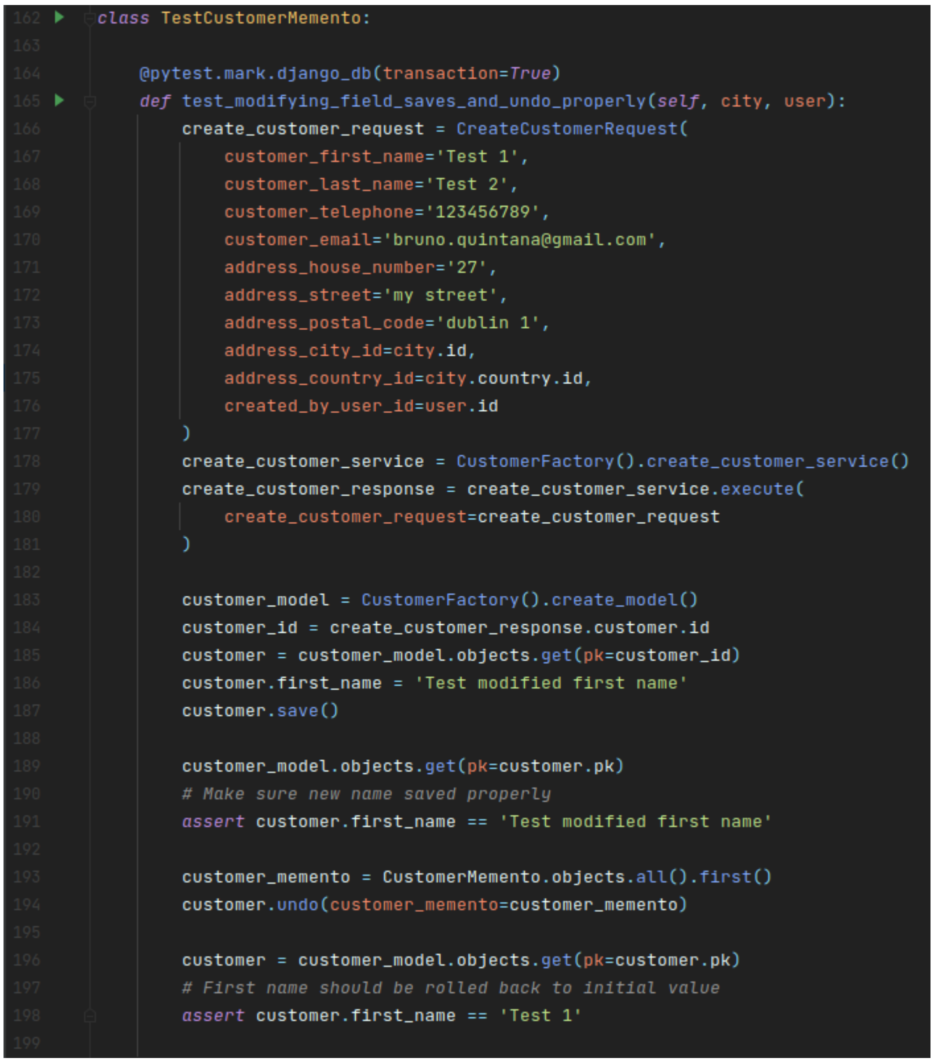


```python
    # Customer (ORM)
    def save_memento(self):
        if self.has_changed:
            for field_name in self.changed_fields:
                # Id will change on creation only, so we ignore this.
                if field_name == 'id':
                    continue
                customer_memento = CustomerMemento()
                customer_memento.customer = self
                customer_memento.field_name = field_name
                customer_memento.field_value = self._dict[field_name]
                customer_memento.save()
```



```python
    def undo(self, customer_memento: CustomerMemento):
        if self.id != customer_memento.customer.id:
            raise Exception('Customer memento does not match current customer')
        setattr(self, customer_memento.field_name, customer_memento.field_value)
        super().save()
```

Unit Test



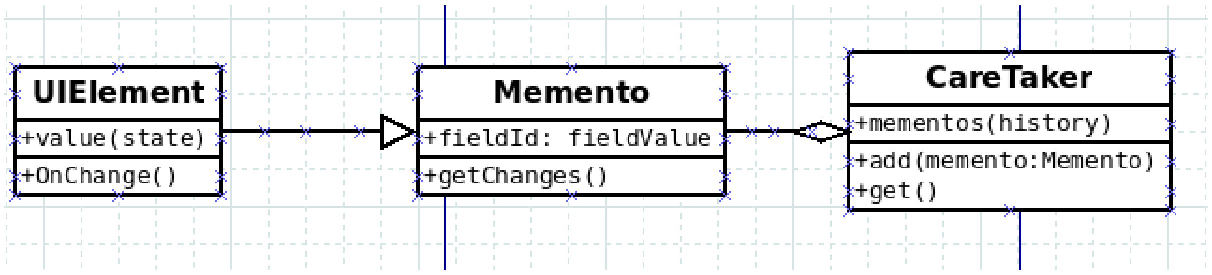
```python
class TestCustomerMemento:

    @pytest.mark.django_db(transaction=True)
    def test_modifying_field_saves_and_undo_properly(self, city, user):
        create_customer_request = CreateCustomerRequest(
            customer_first_name='Test 1',
            customer_last_name='Test 2',
            customer_telephone='123456789',
            customer_email='bruno.quintana@gmail.com',
            address_house_number='27',
            address_street='my street',
            address_postal_code='dublin 1',
            address_city_id=city.id,
            address_country_id=city.country.id,
            created_by_user_id=user.id
        )
        create_customer_service = CustomerFactory().create_customer_service()
        create_customer_response = create_customer_service.execute(
            create_customer_request=create_customer_request
        )

        customer_model = CustomerFactory().create_model()
        customer_id = create_customer_response.customer.id
        customer = customer_model.objects.get(pk=customer_id)
        customer.first_name = 'Test modified first name'
        customer.save()

        customer_model.objects.get(pk=customer.pk)
        # Make sure new name saved properly
        assert customer.first_name == 'Test modified first name'

        customer_memento = CustomerMemento.objects.all().first()
        customer.undo(customer_memento=customer_memento)

        customer = customer_model.objects.get(pk=customer.pk)
        # First name should be rolled back to initial value
        assert customer.first_name == 'Test 1'
```

Memento - Frontend - Javascript

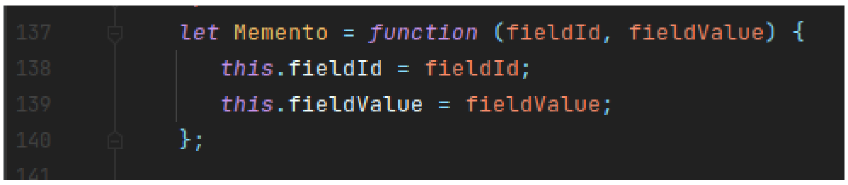
Business Requirement

In order to improve user experience the Memento pattern will be used to undo personal guests information in the UI.

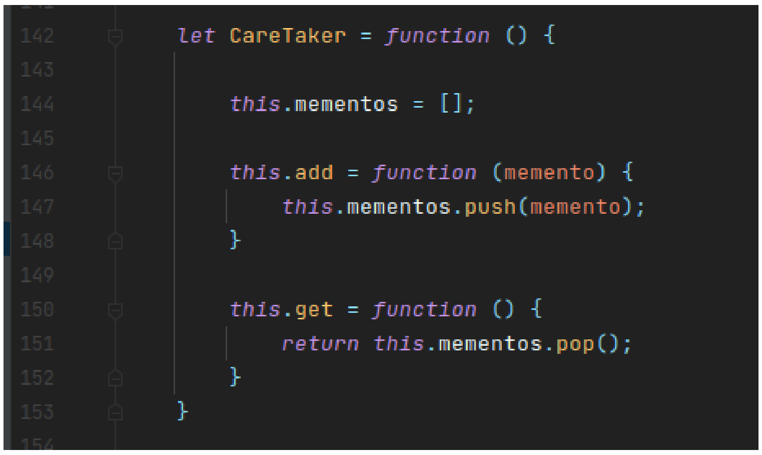
Implementation



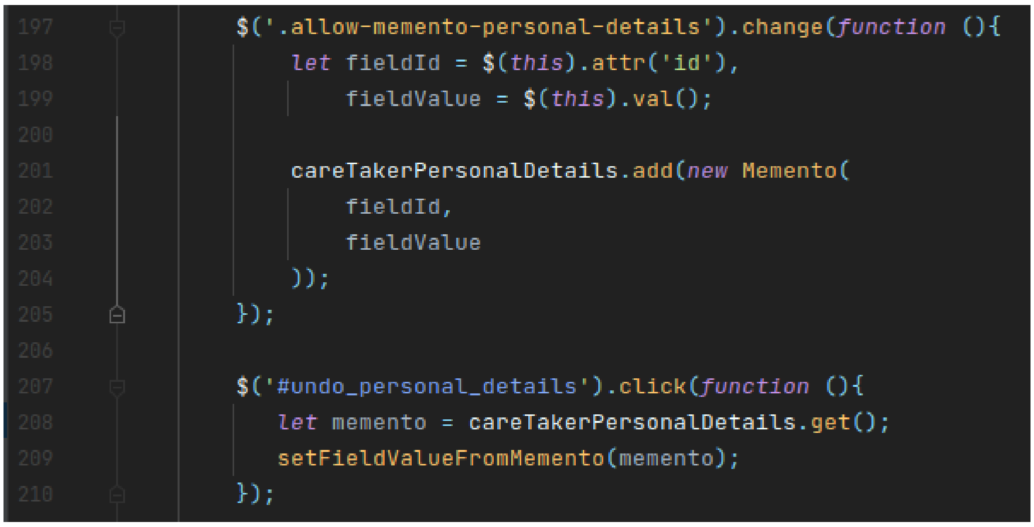
Code



```
let Memento = function (fieldId, fieldValue) {
    this.fieldId = fieldId;
    this.fieldValue = fieldValue;
};
```



```
let CareTaker = function () {

    this.mementos = [];

    this.add = function (memento) {
        this.mementos.push(memento);
    }

    this.get = function () {
        return this.mementos.pop();
    }
}
```



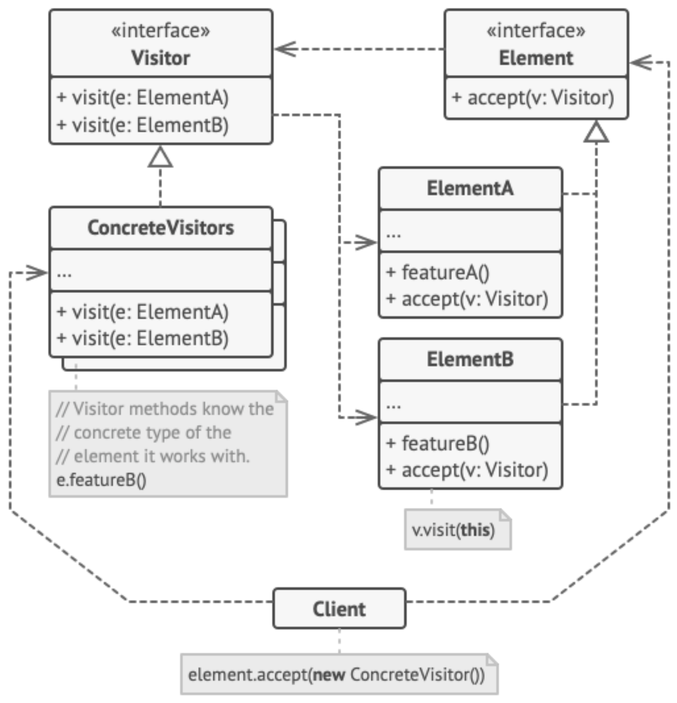
```
$('.allow-memento-personal-details').change(function (){
    let fieldId = $(this).attr('id'),
        fieldValue = $(this).val();

    careTakerPersonalDetails.add(new Memento(
        fieldId,
        fieldValue
    ));
});

$('#undo_personal_details').click(function (){
   let memento = careTakerPersonalDetails.get();
   setFieldValueFromMemento(memento);
});
```

The visitor

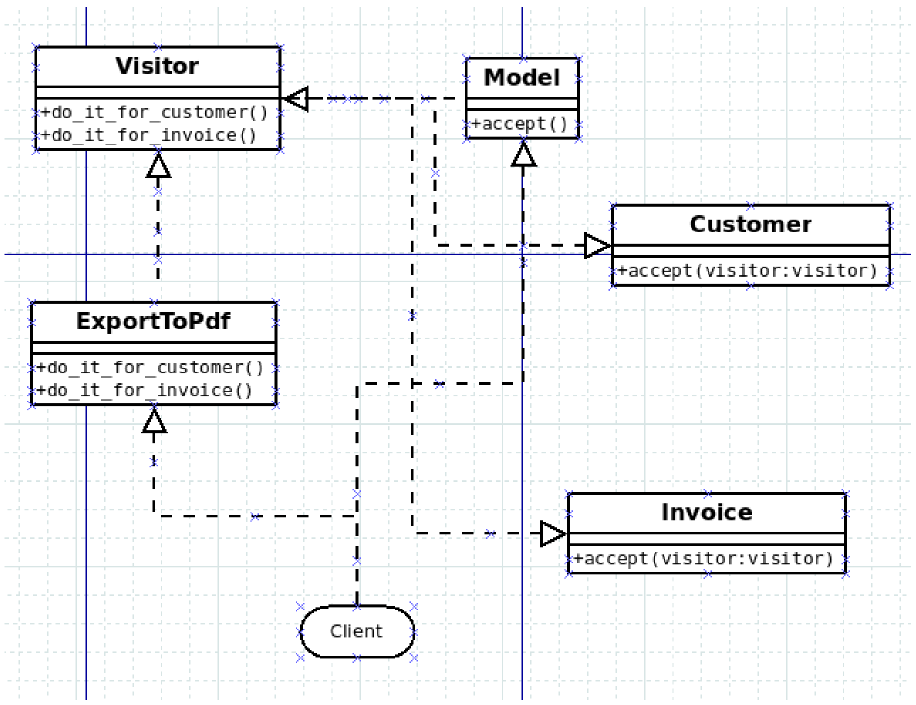
Business Requirement

The Hotel Manager requested to have a daily PDF with all new customers, while the Hotel Accountant requested to have a daily PDF with all new invoices, he may also request in future as Excel File.

Theory

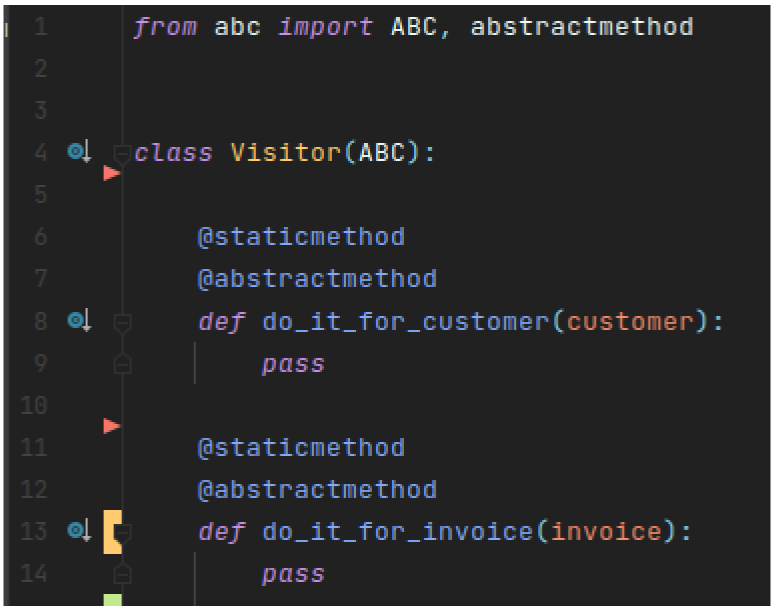


Implementation



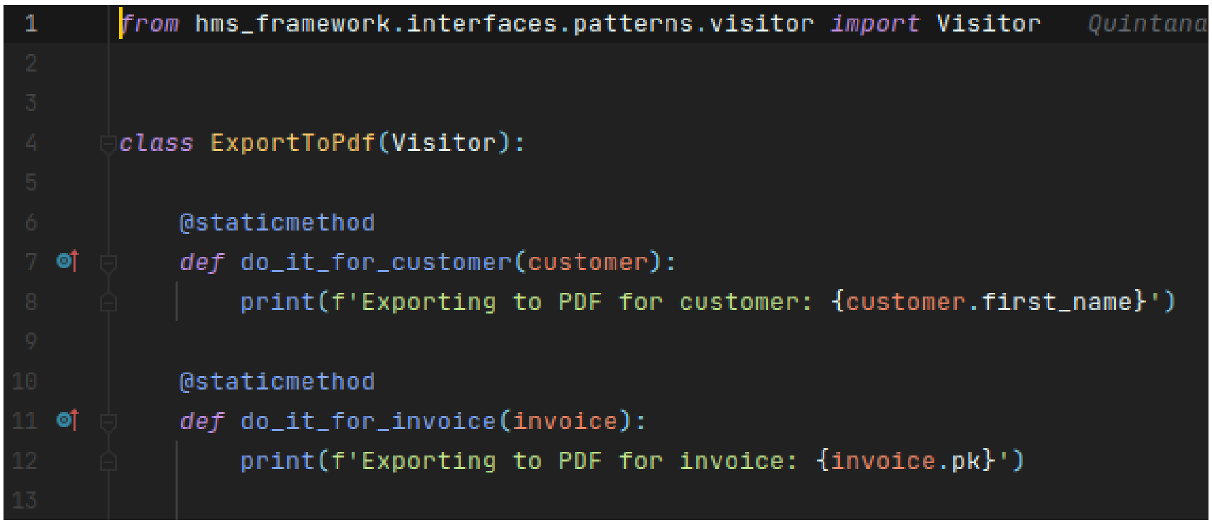
Code

Interface:



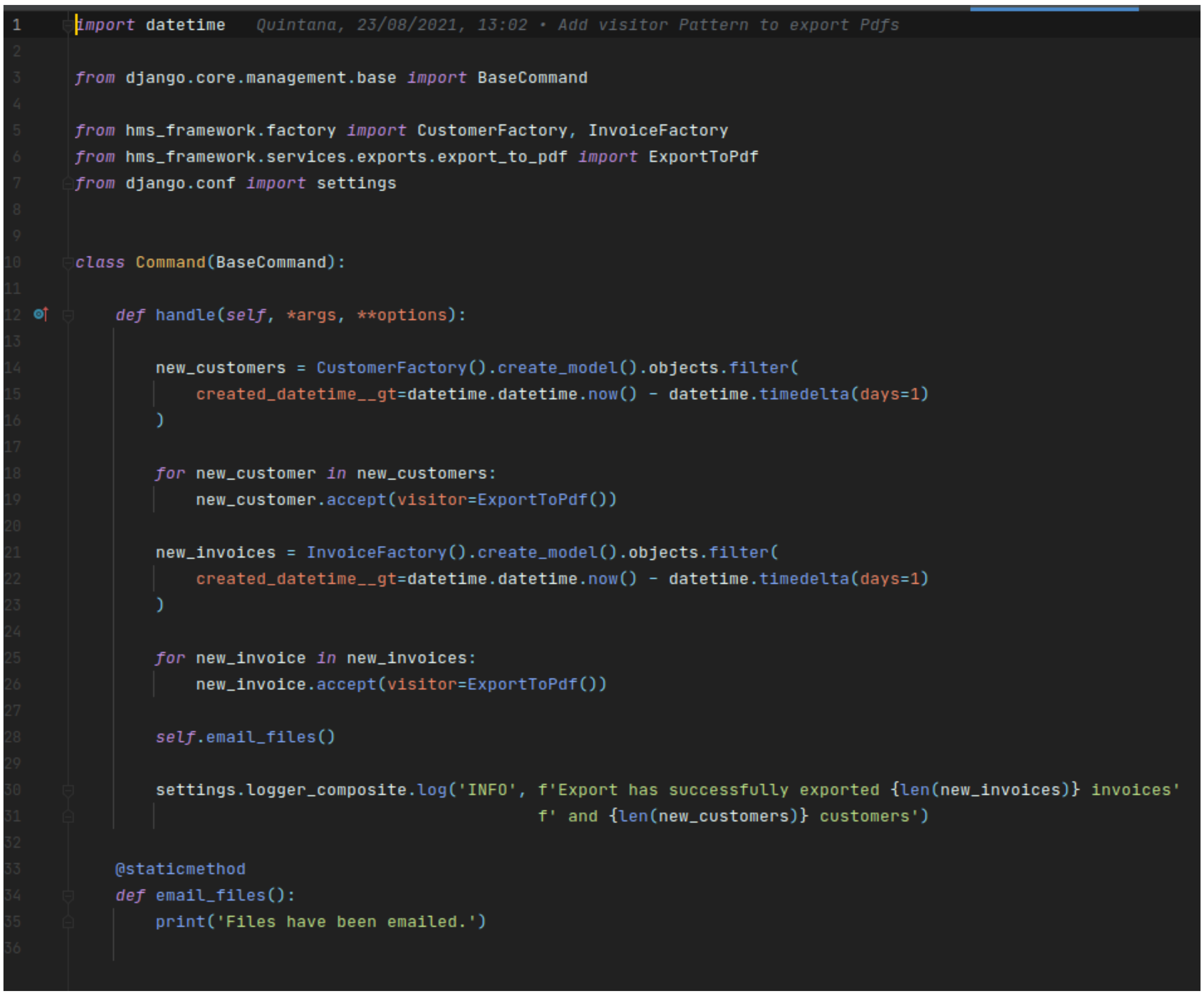
```python
from abc import ABC, abstractmethod


class Visitor(ABC):

    @staticmethod
    @abstractmethod
    def do_it_for_customer(customer):
        pass

    @staticmethod
    @abstractmethod
    def do_it_for_invoice(invoice):
        pass
```

Concrete:



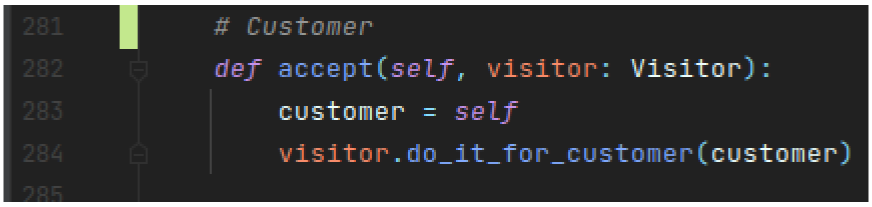
```python
from hms_framework.interfaces.patterns.visitor import Visitor    Quintana


class ExportToPdf(Visitor):

    @staticmethod
    def do_it_for_customer(customer):
        print(f'Exporting to PDF for customer: {customer.first_name}')

    @staticmethod
    def do_it_for_invoice(invoice):
        print(f'Exporting to PDF for invoice: {invoice.pk}')
```

Client (crontab command):

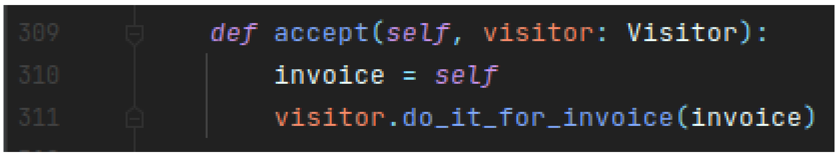


```python
import datetime   Quintana, 23/08/2021, 13:02 • Add visitor Pattern to export Pdfs

from django.core.management.base import BaseCommand

from hms_framework.factory import CustomerFactory, InvoiceFactory
from hms_framework.services.exports.export_to_pdf import ExportToPdf
from django.conf import settings


class Command(BaseCommand):

    def handle(self, *args, **options):

        new_customers = CustomerFactory().create_model().objects.filter(
            created_datetime__gt=datetime.datetime.now() - datetime.timedelta(days=1)
        )

        for new_customer in new_customers:
            new_customer.accept(visitor=ExportToPdf())

        new_invoices = InvoiceFactory().create_model().objects.filter(
            created_datetime__gt=datetime.datetime.now() - datetime.timedelta(days=1)
        )

        for new_invoice in new_invoices:
            new_invoice.accept(visitor=ExportToPdf())

        self.email_files()

        settings.logger_composite.log('INFO', f'Export has successfully exported {len(new_invoices)} invoices'
                                              f' and {len(new_customers)} customers')

    @staticmethod
    def email_files():
        print('Files have been emailed.')
```

Accept methods on both ORM Models:



```python
    # Customer
    def accept(self, visitor: Visitor):
        customer = self
        visitor.do_it_for_customer(customer)
```



```python
    def accept(self, visitor: Visitor):
        invoice = self
        visitor.do_it_for_invoice(invoice)
```

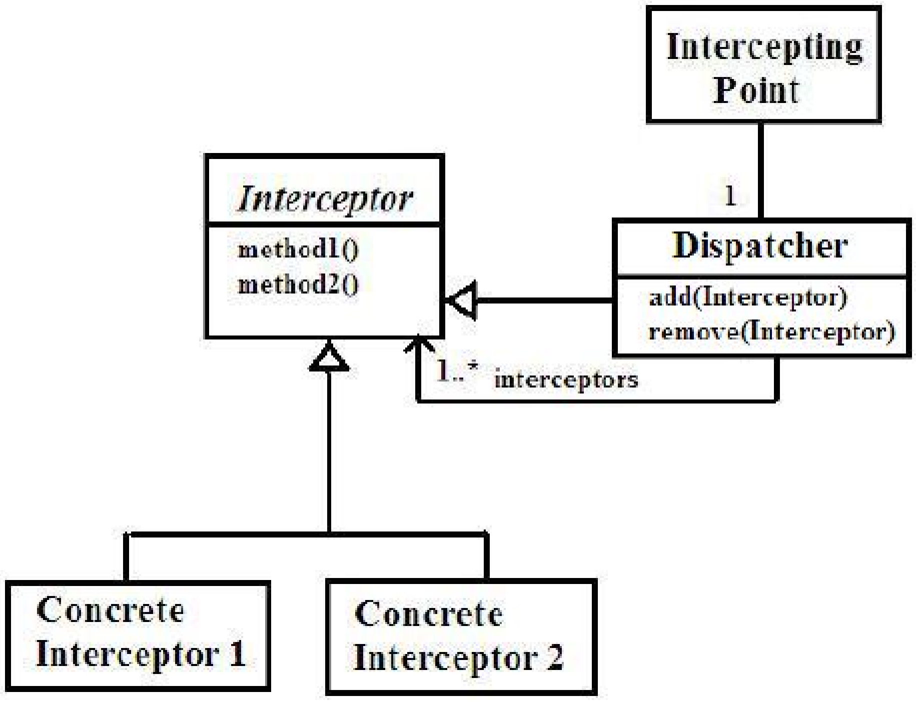
Architectural Patterns

The interceptor

Business Requirements

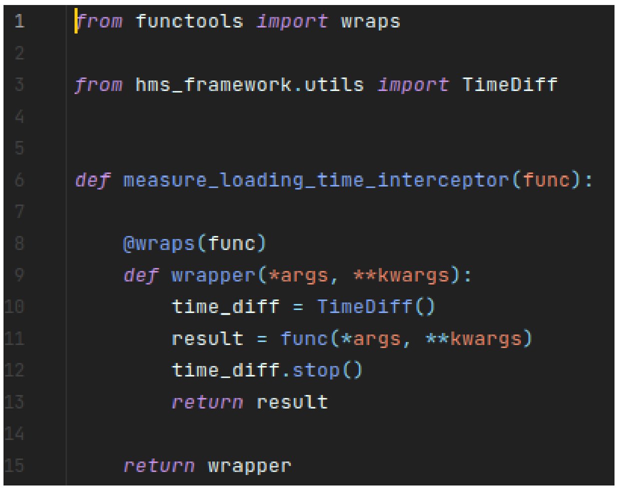
The hotel CTO required that an interceptor be used to quickly enable and disable loading time measures on any function that exists within the framework/application.

Theory

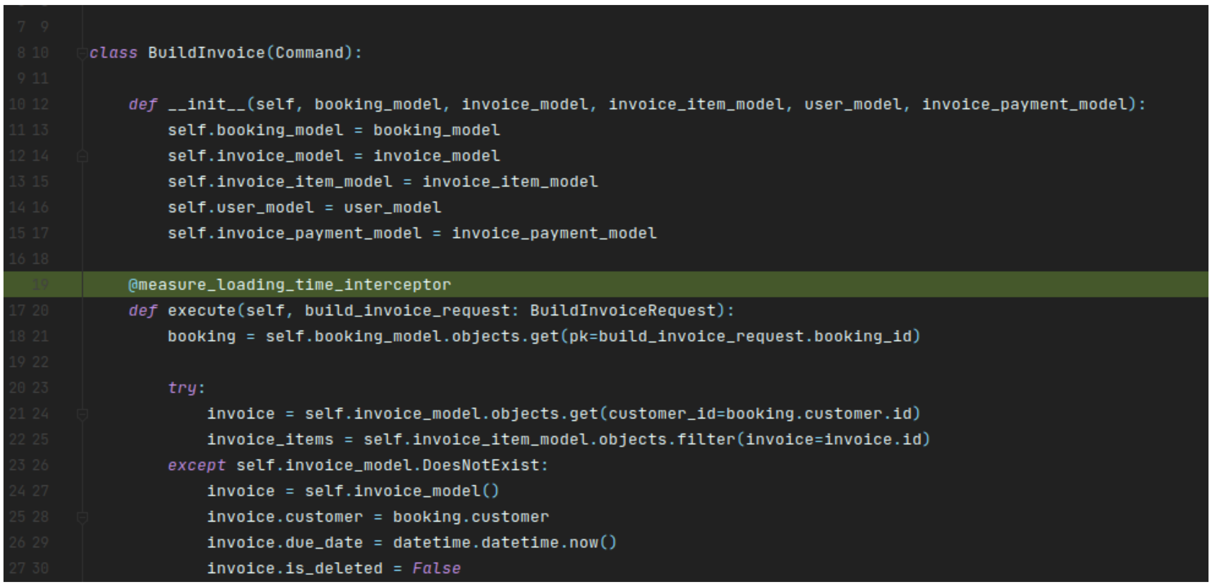


Implementation

Python simplifies the interceptor by adding @name_of_interceptor above the function, therefore Python acts as the dispatcher to the concrete implementation.



```python
from functools import wraps

from hms_framework.utils import TimeDiff


def measure_loading_time_interceptor(func):

    @wraps(func)
    def wrapper(*args, **kwargs):
        time_diff = TimeDiff()
        result = func(*args, **kwargs)
        time_diff.stop()
        return result

    return wrapper
```

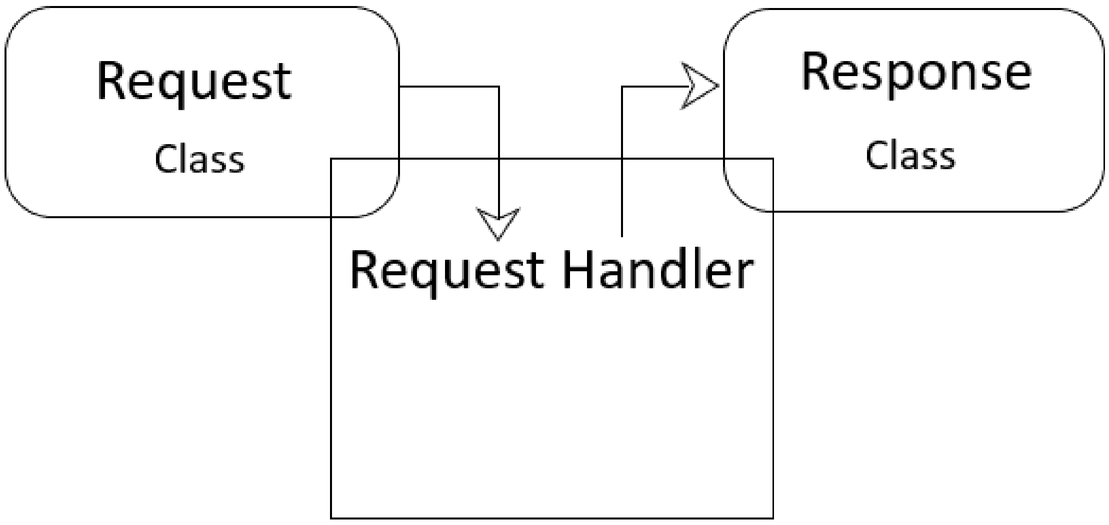


```
class BuildInvoice(Command):

    def __init__(self, booking_model, invoice_model, invoice_item_model, user_model, invoice_payment_model):
        self.booking_model = booking_model
        self.invoice_model = invoice_model
        self.invoice_item_model = invoice_item_model
        self.user_model = user_model
        self.invoice_payment_model = invoice_payment_model

    @measure_loading_time_interceptor
    def execute(self, build_invoice_request: BuildInvoiceRequest):
        booking = self.booking_model.objects.get(pk=build_invoice_request.booking_id)

        try:
            invoice = self.invoice_model.objects.get(customer_id=booking.customer.id)
            invoice_items = self.invoice_item_model.objects.filter(invoice=invoice.id)
        except self.invoice_model.DoesNotExist:
            invoice = self.invoice_model()
            invoice.customer = booking.customer
            invoice.due_date = datetime.datetime.now()
            invoice.is_deleted = False
```

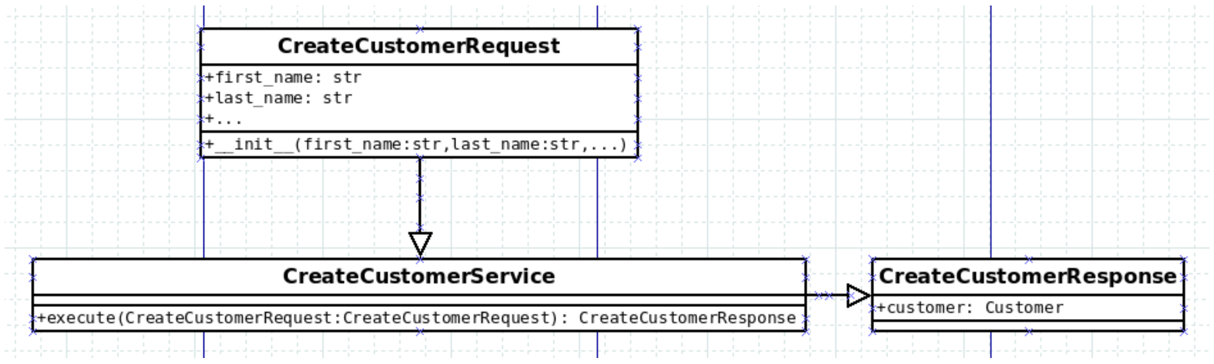
Request-Response

This pattern is probably one of the most used world wide, as it is the base of everything including the Internet. The adaptation into the programming world results as a main class “The Handler”, in the case CreateCustomer, that takes as the only parameter the CreateCustomerRequest, and returns the CreateCustomerResponse.
By having a class instance as a parameter, we are no longer coupling the service to the method of execution: browser will send parameters as POST request, console will send parameters as ARGS, and so on. It also leaves the function signature intact if a new parameter is added as it will be a new attribute to the class Request, the same happens for the Response.

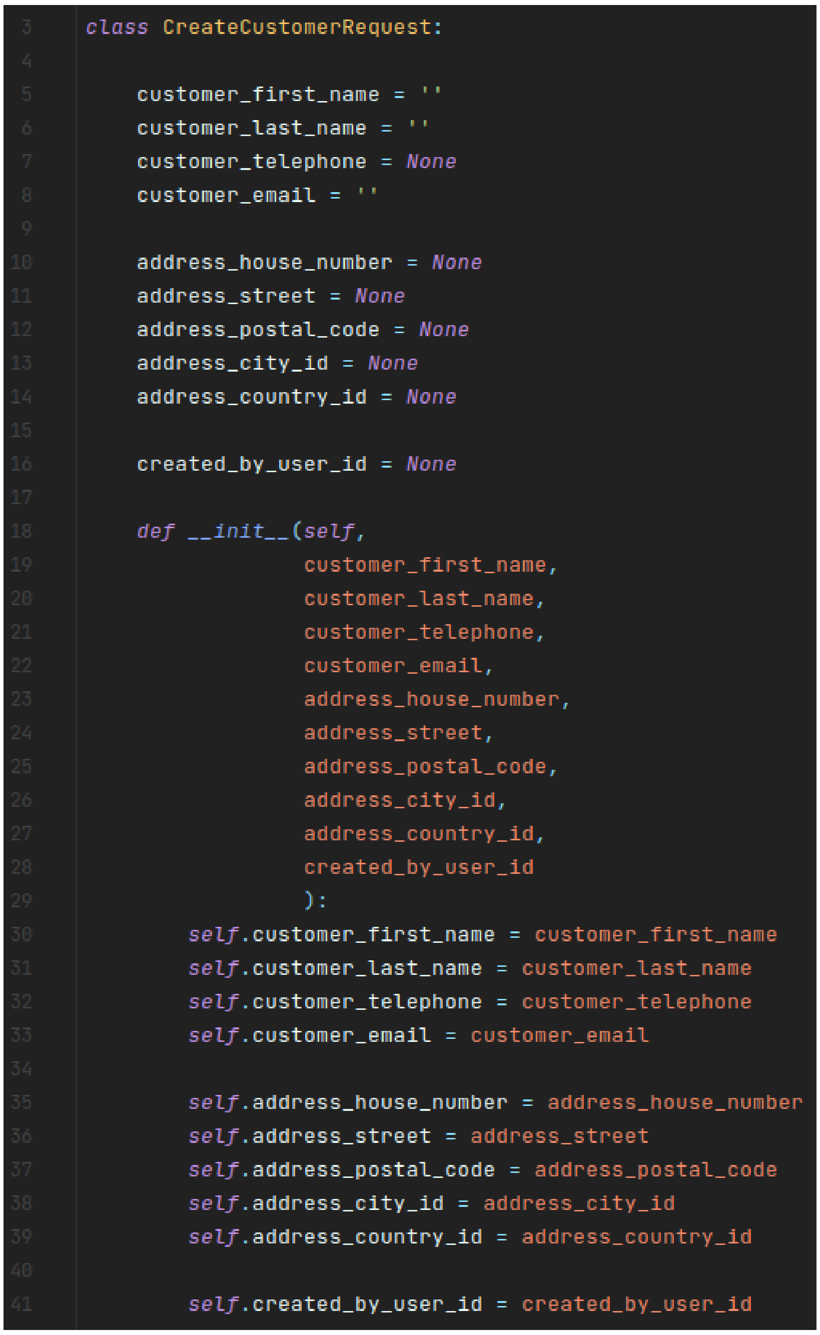
Theory



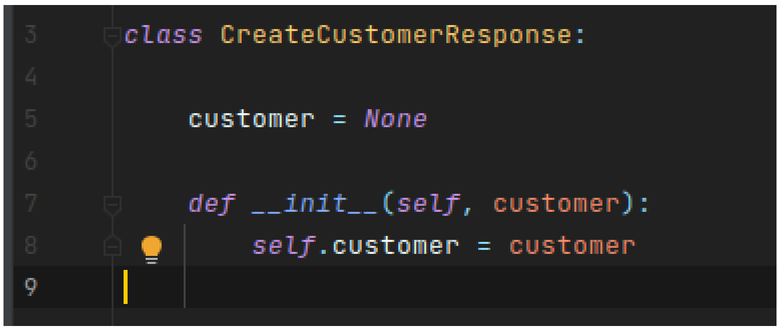
Implementation


CreateCustomerRequest
+first_name: str
+last_name: str
+...
+__init__(first_name:str,last_name:str,...)
CreateCustomerService
+execute(CreateCustomerRequest:CreateCustomerRequest): CreateCustomerResponse
CreateCustomerResponse
+customer: Customer

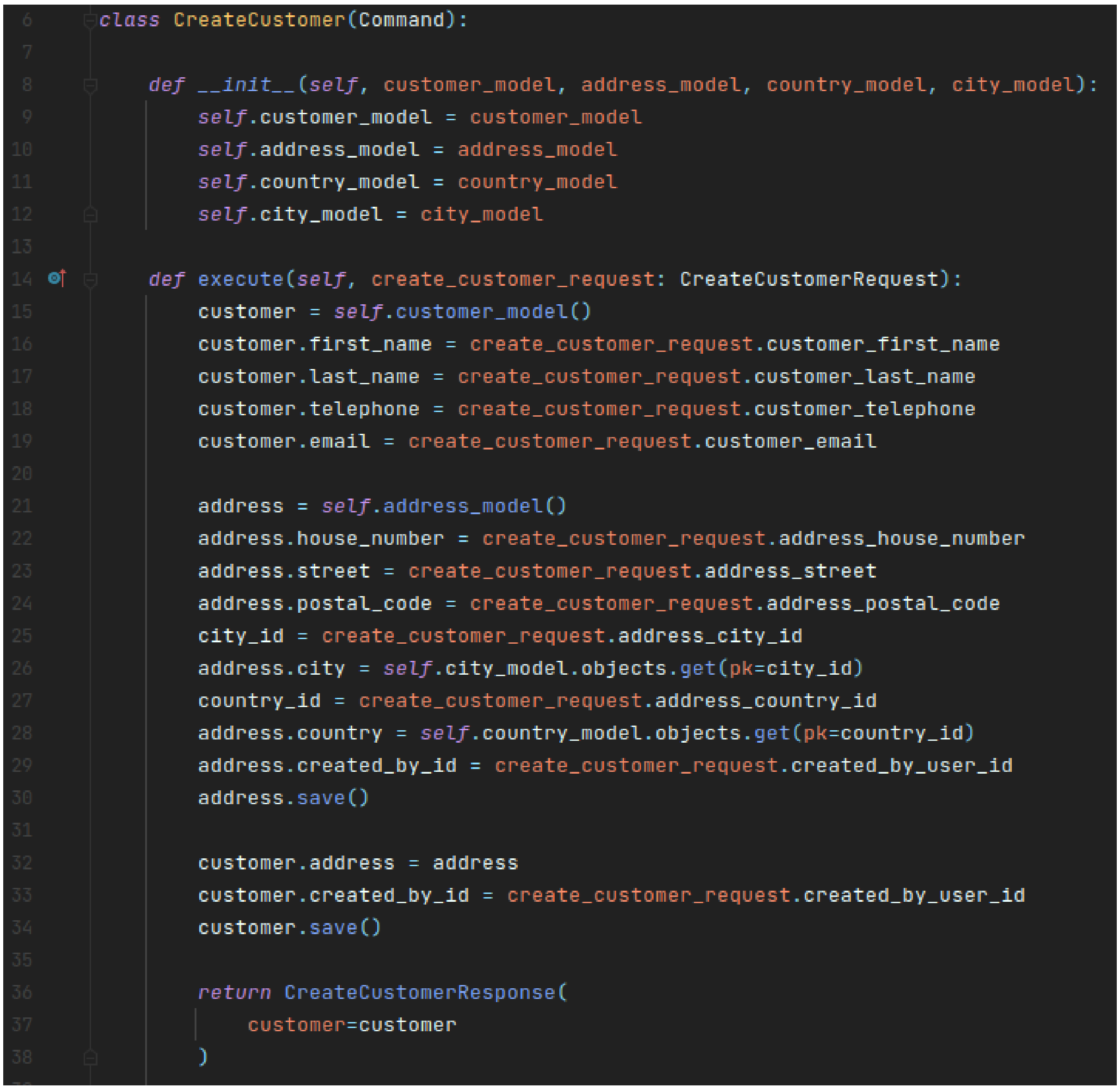
Code



```python
class CreateCustomerRequest:

    customer_first_name = ''
    customer_last_name = ''
    customer_telephone = None
    customer_email = ''

    address_house_number = None
    address_street = None
    address_postal_code = None
    address_city_id = None
    address_country_id = None

    created_by_user_id = None

    def __init__(self,
                 customer_first_name,
                 customer_last_name,
                 customer_telephone,
                 customer_email,
                 address_house_number,
                 address_street,
                 address_postal_code,
                 address_city_id,
                 address_country_id,
                 created_by_user_id
                 ):
        self.customer_first_name = customer_first_name
        self.customer_last_name = customer_last_name
        self.customer_telephone = customer_telephone
        self.customer_email = customer_email

        self.address_house_number = address_house_number
        self.address_street = address_street
        self.address_postal_code = address_postal_code
        self.address_city_id = address_city_id
        self.address_country_id = address_country_id

        self.created_by_user_id = created_by_user_id
```



```python
class CreateCustomerResponse:

    customer = None

    def __init__(self, customer):
        self.customer = customer

```

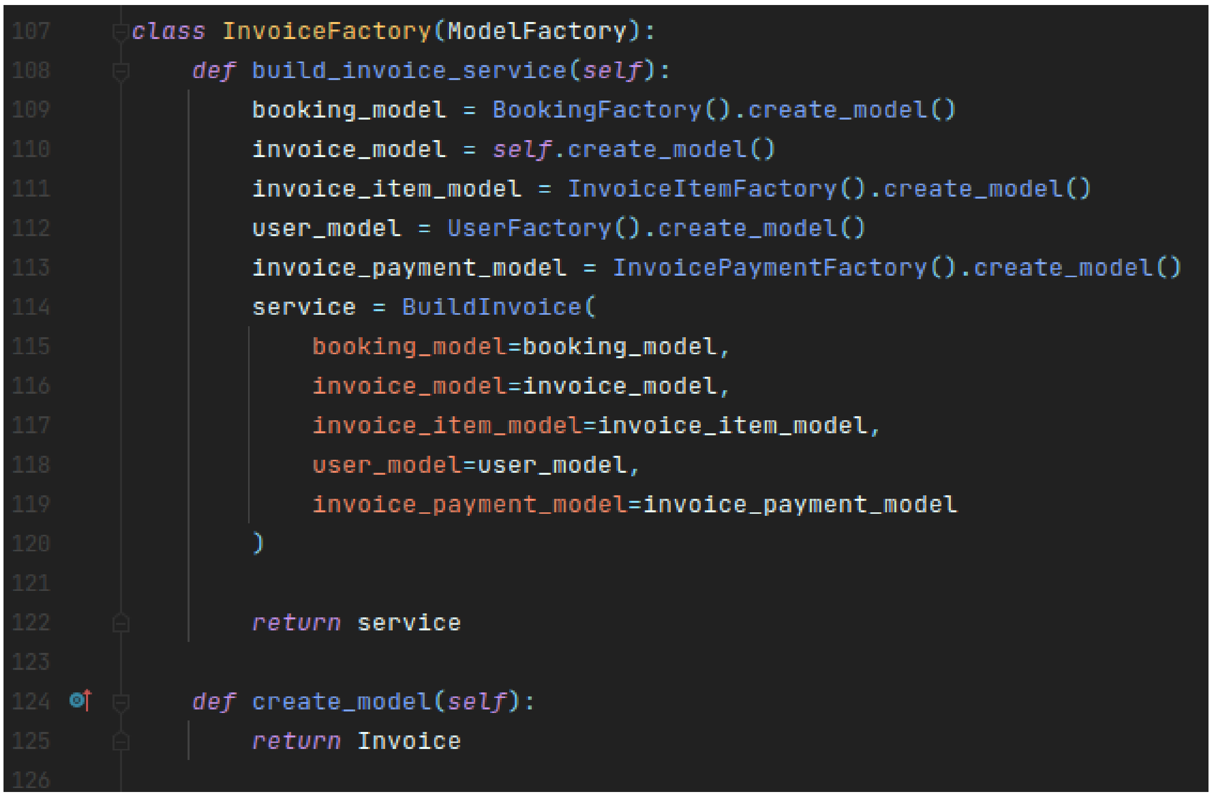


```python
class CreateCustomer(Command):

    def __init__(self, customer_model, address_model, country_model, city_model):
        self.customer_model = customer_model
        self.address_model = address_model
        self.country_model = country_model
        self.city_model = city_model

    def execute(self, create_customer_request: CreateCustomerRequest):
        customer = self.customer_model()
        customer.first_name = create_customer_request.customer_first_name
        customer.last_name = create_customer_request.customer_last_name
        customer.telephone = create_customer_request.customer_telephone
        customer.email = create_customer_request.customer_email

        address = self.address_model()
        address.house_number = create_customer_request.address_house_number
        address.street = create_customer_request.address_street
        address.postal_code = create_customer_request.address_postal_code
        city_id = create_customer_request.address_city_id
        address.city = self.city_model.objects.get(pk=city_id)
        country_id = create_customer_request.address_country_id
        address.country = self.country_model.objects.get(pk=country_id)
        address.created_by_id = create_customer_request.created_by_user_id
        address.save()

        customer.address = address
        customer.created_by_id = create_customer_request.created_by_user_id
        customer.save()

        return CreateCustomerResponse(
            customer=customer
        )
```

Creational Patterns

The Factory Method

By centralizing the creation of classes to one place, it becomes easier to maintain and exchange dependencies without affecting the whole code.



```python
class InvoiceFactory(ModelFactory):
    def build_invoice_service(self):
        booking_model = BookingFactory().create_model()
        invoice_model = self.create_model()
        invoice_item_model = InvoiceItemFactory().create_model()
        user_model = UserFactory().create_model()
        invoice_payment_model = InvoicePaymentFactory().create_model()
        service = BuildInvoice(
            booking_model=booking_model,
            invoice_model=invoice_model,
            invoice_item_model=invoice_item_model,
            user_model=user_model,
            invoice_payment_model=invoice_payment_model
        )

        return service

    def create_model(self):
        return Invoice
```

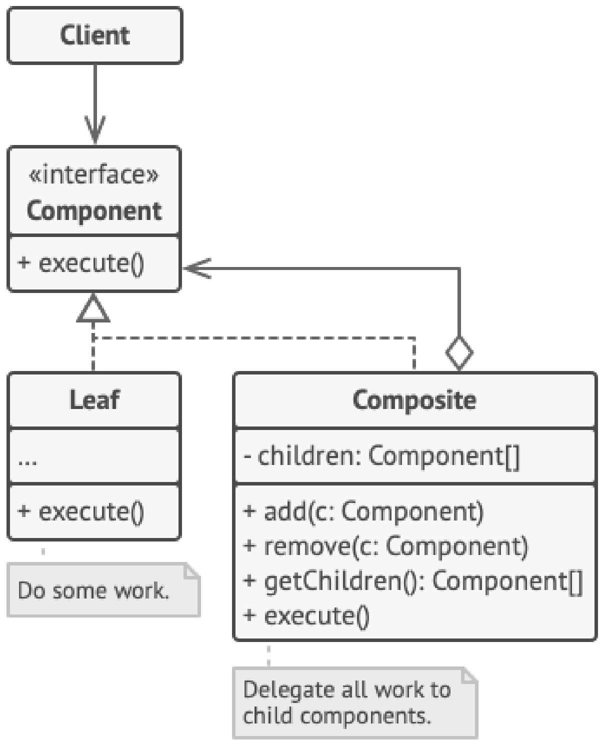
Structural Patterns

Composite

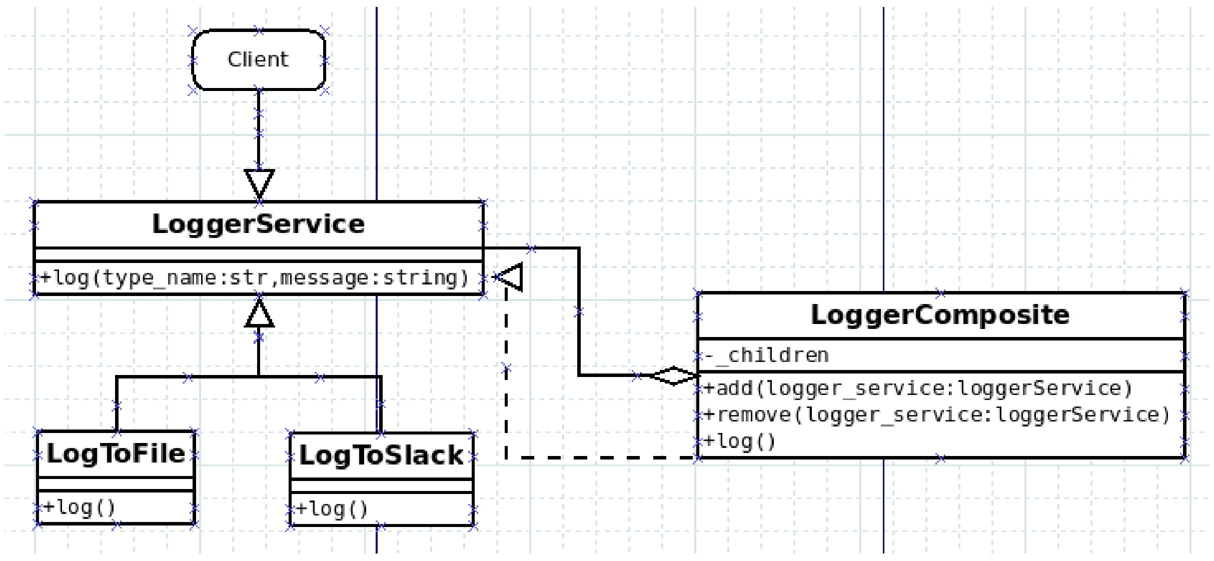
Business Requirements

The hotel CTO required an easy way to add loggers application wise, by using the composite pattern each CTO can simply modify the settings.py in the application and add/remove logger handlers.

Theory

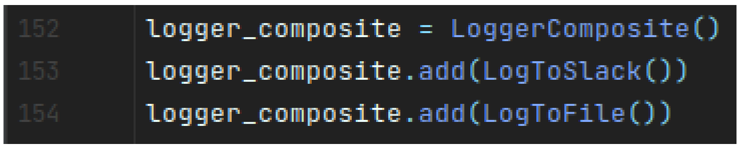


Implementation


Client
LoggerService
+log(type_name:str,message:string)
LogToFile
+log()
LogToSlack
+log()
LoggerComposite
-_children
+add(logger_service:loggerService)
+remove(logger_service:loggerService)
+log()

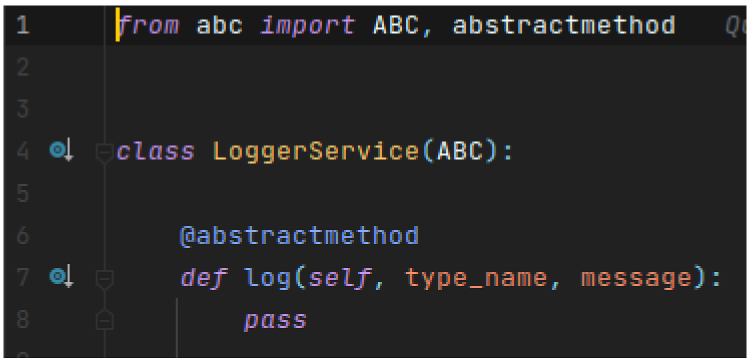
Code

settings.py



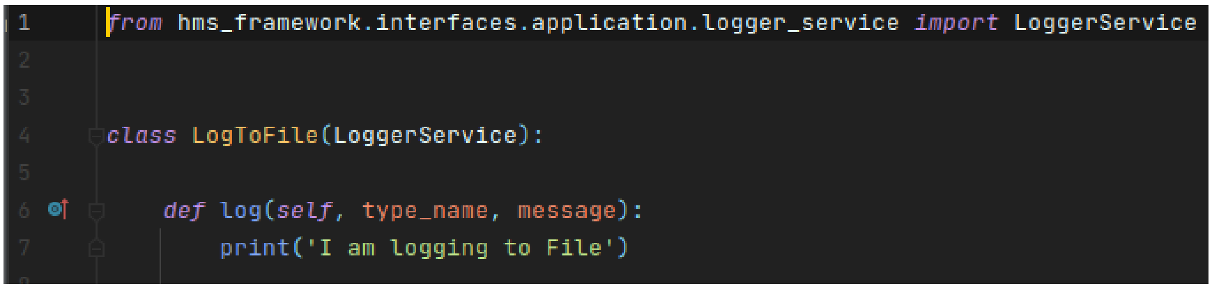
```python
logger_composite = LoggerComposite()
logger_composite.add(LogToSlack())
logger_composite.add(LogToFile())
```

Interface:

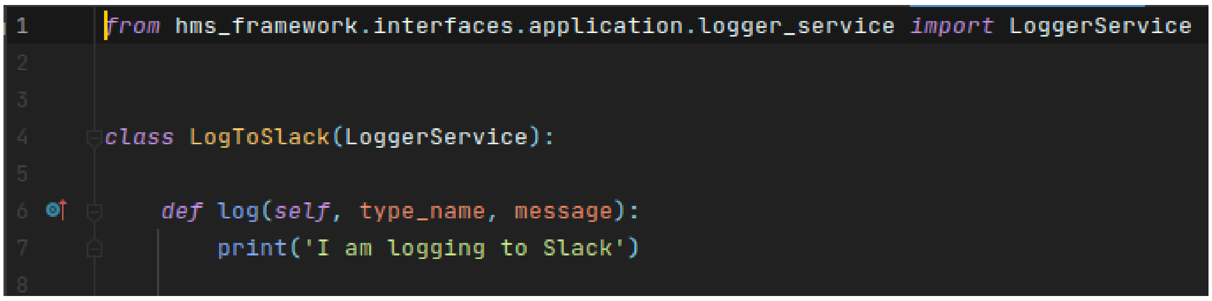


```python
from abc import ABC, abstractmethod


class LoggerService(ABC):

    @abstractmethod
    def log(self, type_name, message):
        pass
```

Concretes:

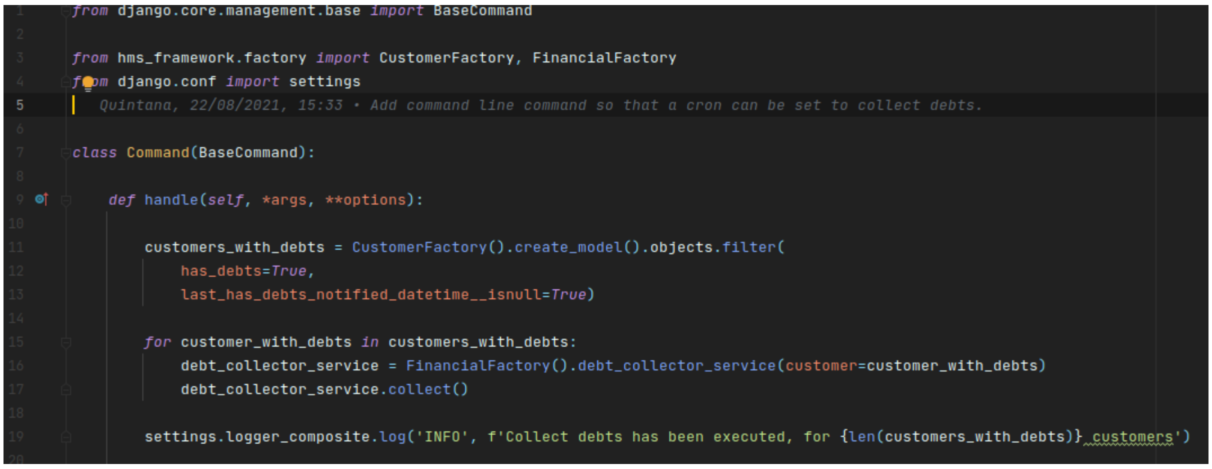


```python
from hms_framework.interfaces.application.logger_service import LoggerService


class LogToFile(LoggerService):

    def log(self, type_name, message):
        print('I am logging to File')
```



```python
from hms_framework.interfaces.application.logger_service import LoggerService


class LogToSlack(LoggerService):

    def log(self, type_name, message):
        print('I am logging to Slack')
```

Console command to be run by crontab:



```python
from django.core.management.base import BaseCommand

from hms_framework.factory import CustomerFactory, FinancialFactory
from django.conf import settings
    Quintana, 22/08/2021, 15:33 • Add command line command so that a cron can be set to collect debts.

class Command(BaseCommand):

    def handle(self, *args, **options):

        customers_with_debts = CustomerFactory().create_model().objects.filter(
            has_debts=True,
            last_has_debts_notified_datetime__isnull=True)

        for customer_with_debts in customers_with_debts:
            debt_collector_service = FinancialFactory().debt_collector_service(customer=customer_with_debts)
            debt_collector_service.collect()

        settings.logger_composite.log('INFO', f'Collect debts has been executed, for {len(customers_with_debts)} customers')
```

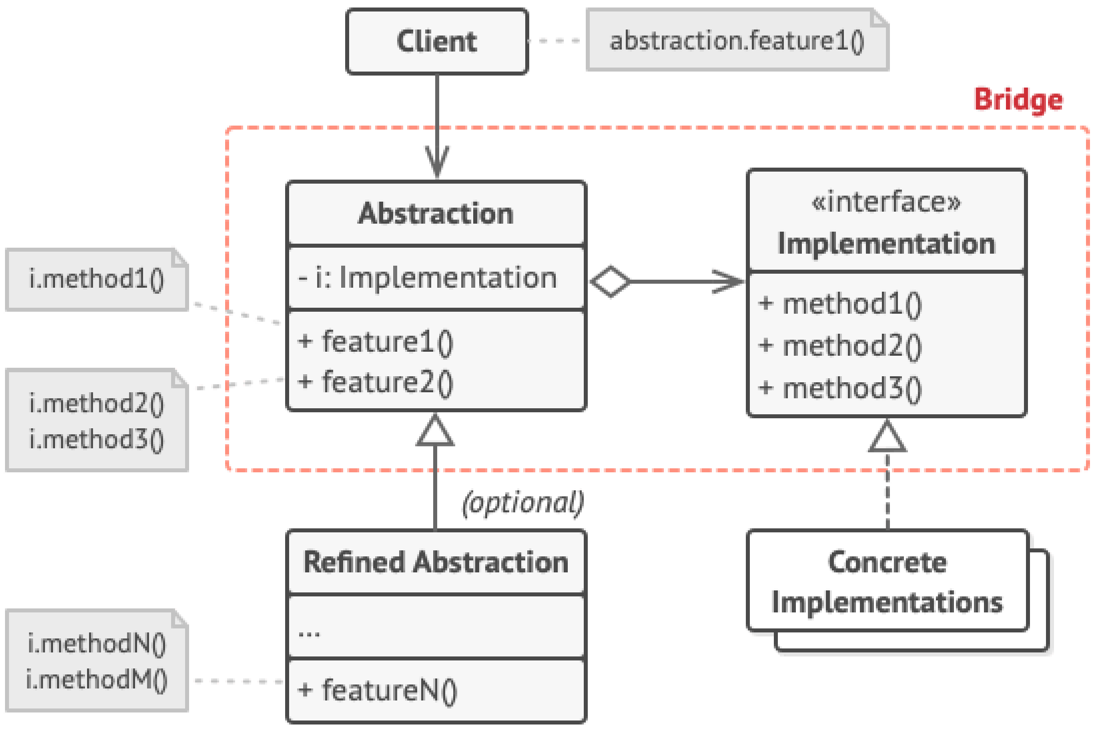
Bridge

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies.

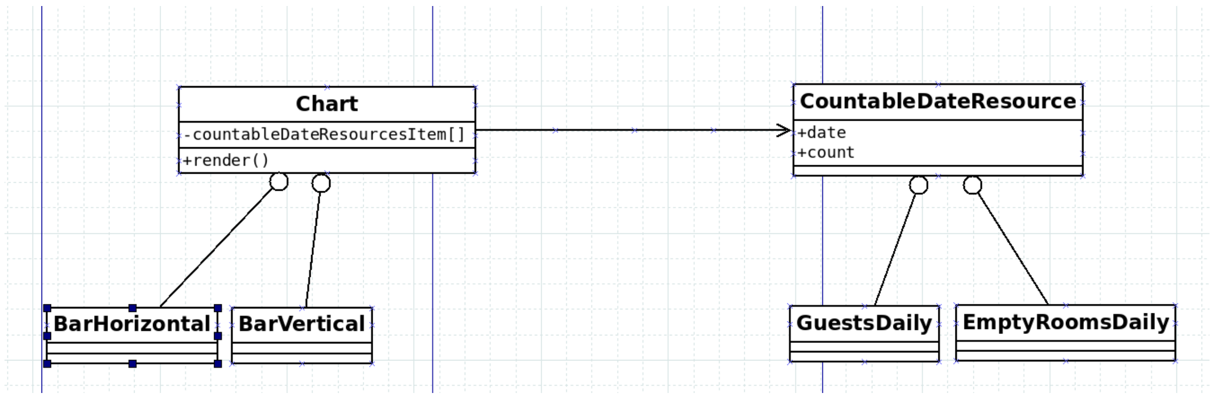
Business Requirement

Within the dashboard hotels owners want to see charts of different metrics of the business. The metrics would always be related to guests, rooms and profits, but it can grow over the time and the charts will be "horizontal bar" and "vertical bar" but these can grow over the time too. It can also happen that in the future they may need to exchange chart types dynamically as some owners may prefer one chart representation over another.

Theory

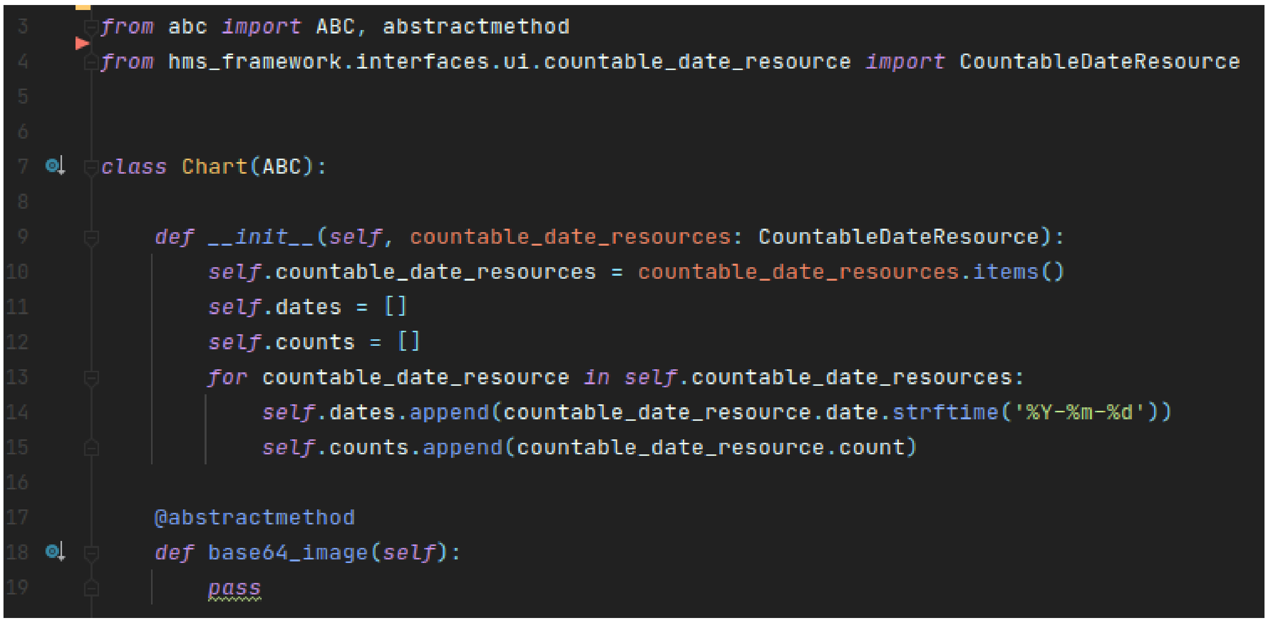


Implementation

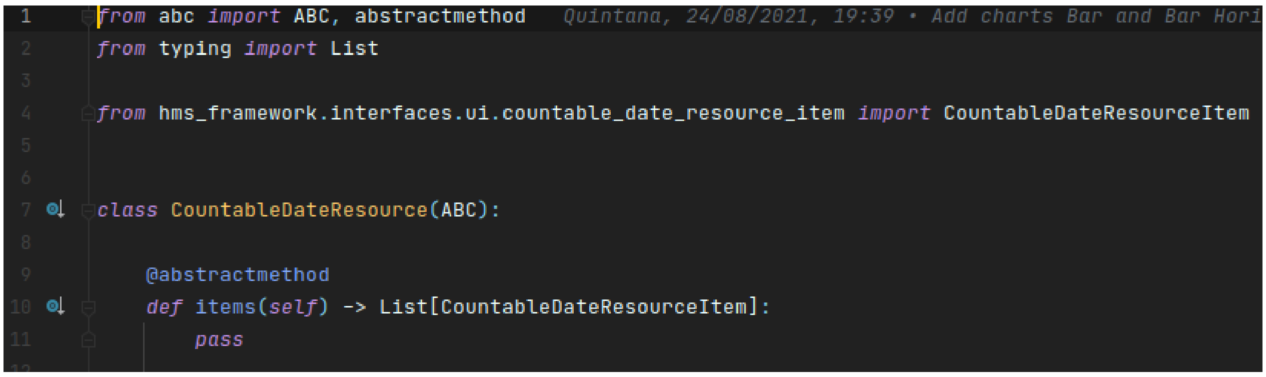


Code

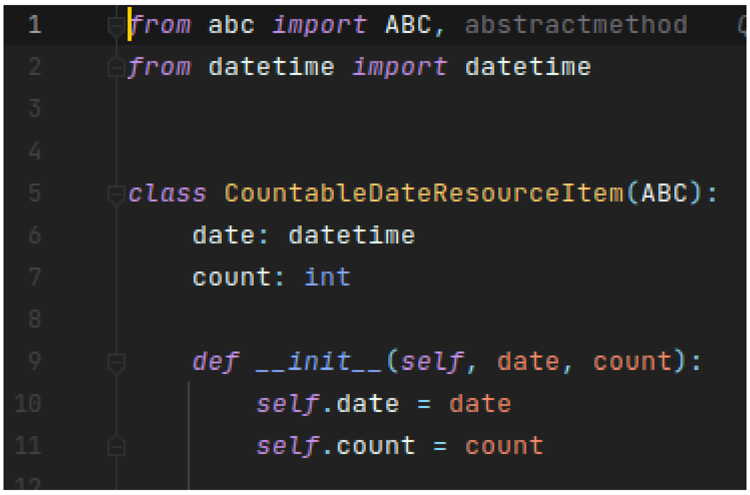
Interfaces



```python
from abc import ABC, abstractmethod
from hms_framework.interfaces.ui.countable_date_resource import CountableDateResource


class Chart(ABC):

    def __init__(self, countable_date_resources: CountableDateResource):
        self.countable_date_resources = countable_date_resources.items()
        self.dates = []
        self.counts = []
        for countable_date_resource in self.countable_date_resources:
            self.dates.append(countable_date_resource.date.strftime('%Y-%m-%d'))
            self.counts.append(countable_date_resource.count)

    @abstractmethod
    def base64_image(self):
        pass
```

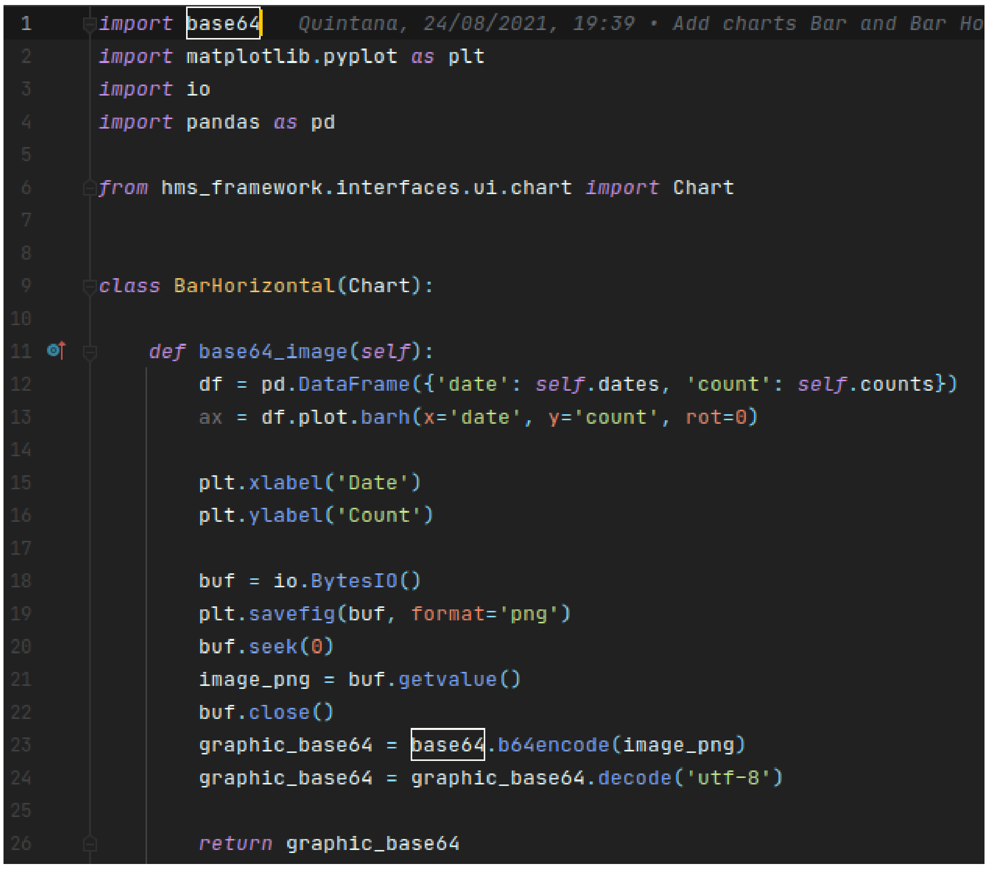


```python
from abc import ABC, abstractmethod    Quintana, 24/08/2021, 19:39 • Add charts Bar and Bar Hori
from typing import List

from hms_framework.interfaces.ui.countable_date_resource_item import CountableDateResourceItem


class CountableDateResource(ABC):

    @abstractmethod
    def items(self) -> List[CountableDateResourceItem]:
        pass
```

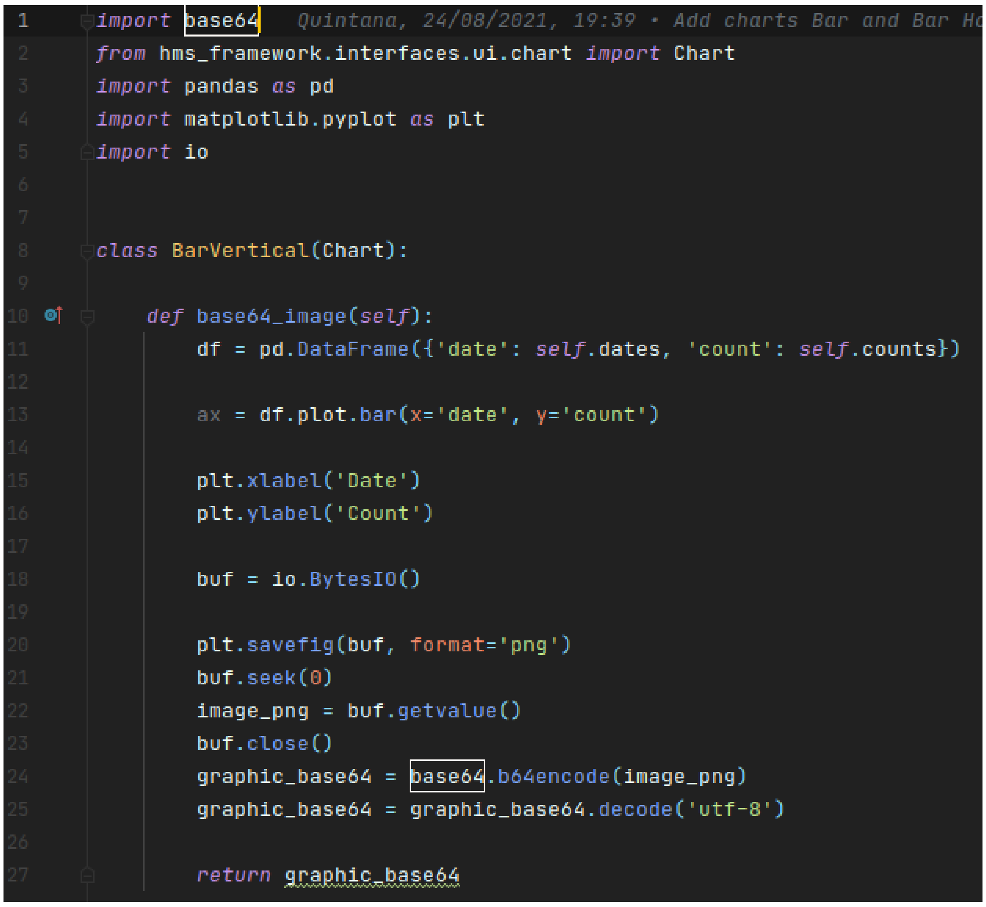


```python
from abc import ABC, abstractmethod
from datetime import datetime


class CountableDateResourceItem(ABC):
    date: datetime
    count: int

    def __init__(self, date, count):
        self.date = date
        self.count = count
```

Concrete

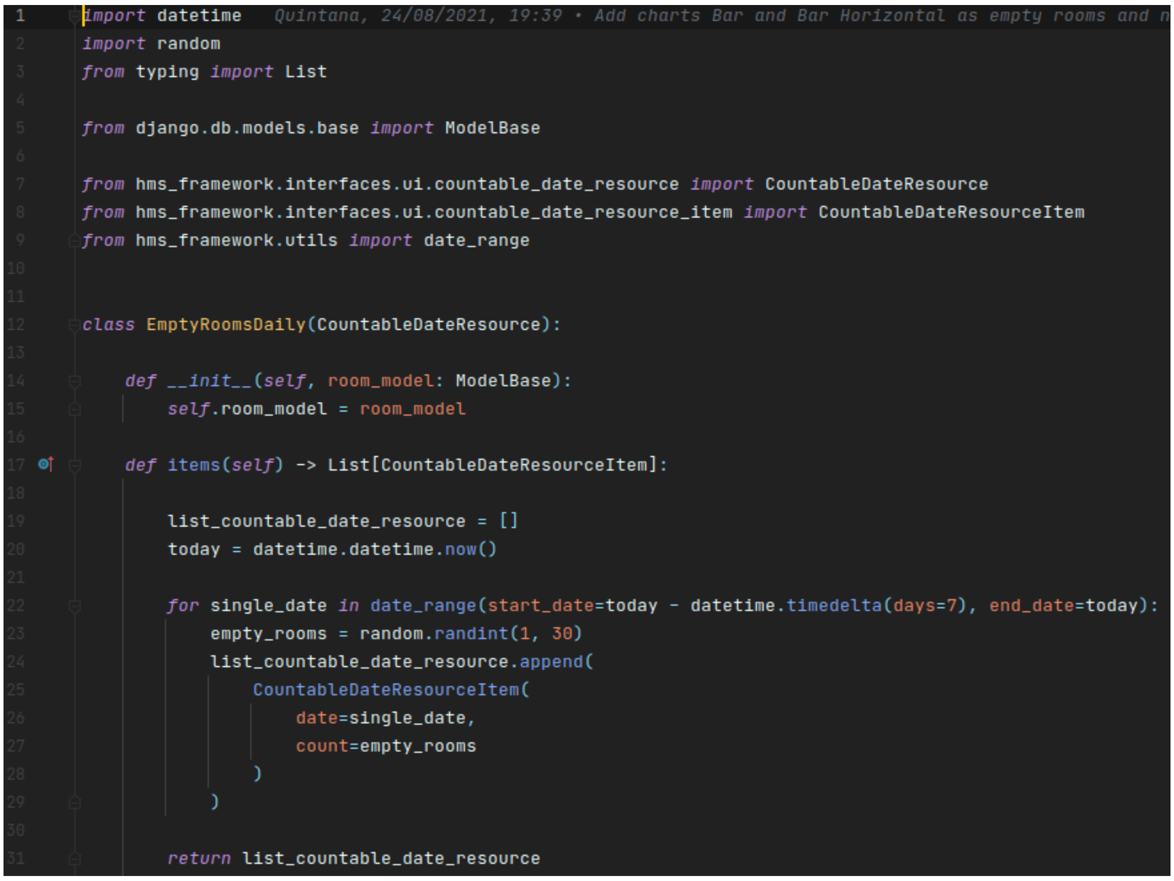


```python
import base64    Quintana, 24/08/2021, 19:39 • Add charts Bar and Bar Ho
import matplotlib.pyplot as plt
import io
import pandas as pd

from hms_framework.interfaces.ui.chart import Chart


class BarHorizontal(Chart):

    def base64_image(self):
        df = pd.DataFrame({'date': self.dates, 'count': self.counts})
        ax = df.plot.barh(x='date', y='count', rot=0)

        plt.xlabel('Date')
        plt.ylabel('Count')

        buf = io.BytesIO()
        plt.savefig(buf, format='png')
        buf.seek(0)
        image_png = buf.getvalue()
        buf.close()
        graphic_base64 = base64.b64encode(image_png)
        graphic_base64 = graphic_base64.decode('utf-8')

        return graphic_base64
```

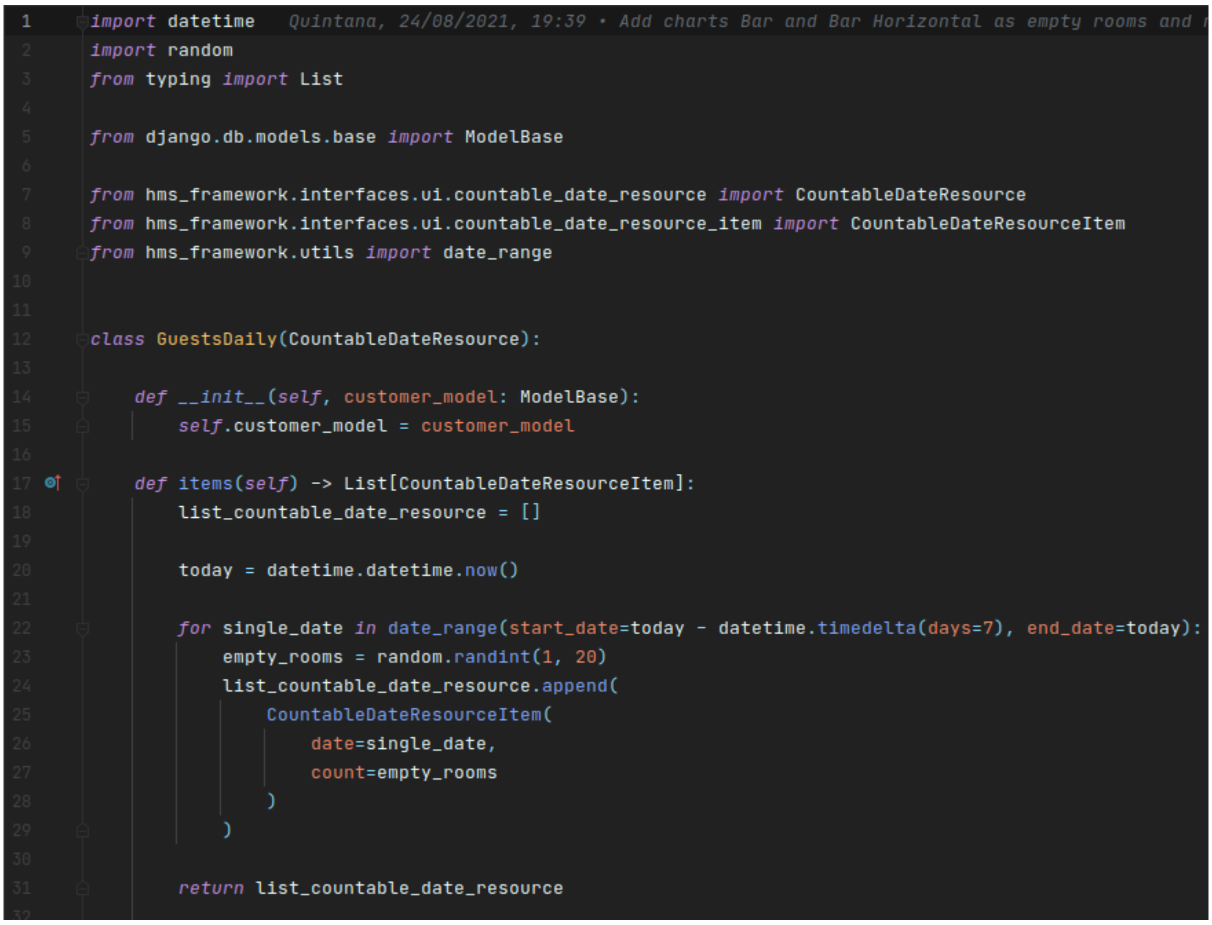


```
import base64    Quintana, 24/08/2021, 19:39 • Add charts Bar and Bar Ho
from hms_framework.interfaces.ui.chart import Chart
import pandas as pd
import matplotlib.pyplot as plt
import io


class BarVertical(Chart):

    def base64_image(self):
        df = pd.DataFrame({'date': self.dates, 'count': self.counts})

        ax = df.plot.bar(x='date', y='count')

        plt.xlabel('Date')
        plt.ylabel('Count')

        buf = io.BytesIO()

        plt.savefig(buf, format='png')
        buf.seek(0)
        image_png = buf.getvalue()
        buf.close()
        graphic_base64 = base64.b64encode(image_png)
        graphic_base64 = graphic_base64.decode('utf-8')

        return graphic_base64
```

Refined Abstraction



```
import datetime    Quintana, 24/08/2021, 19:39 • Add charts Bar and Bar Horizontal as empty rooms and n
import random
from typing import List

from django.db.models.base import ModelBase

from hms_framework.interfaces.ui.countable_date_resource import CountableDateResource
from hms_framework.interfaces.ui.countable_date_resource_item import CountableDateResourceItem
from hms_framework.utils import date_range


class EmptyRoomsDaily(CountableDateResource):

    def __init__(self, room_model: ModelBase):
        self.room_model = room_model

    def items(self) -> List[CountableDateResourceItem]:

        list_countable_date_resource = []
        today = datetime.datetime.now()

        for single_date in date_range(start_date=today - datetime.timedelta(days=7), end_date=today):
            empty_rooms = random.randint(1, 30)
            list_countable_date_resource.append(
                CountableDateResourceItem(
                    date=single_date,
                    count=empty_rooms
                )
            )

        return list_countable_date_resource
```

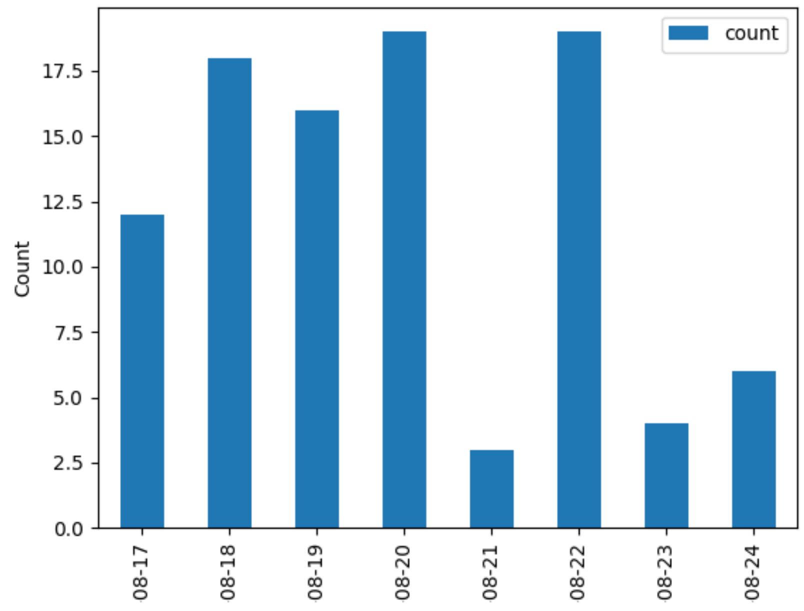


```python
import datetime   Quintana, 24/08/2021, 19:39 • Add charts Bar and Bar Horizontal as empty rooms and
import random
from typing import List

from django.db.models.base import ModelBase

from hms_framework.interfaces.ui.countable_date_resource import CountableDateResource
from hms_framework.interfaces.ui.countable_date_resource_item import CountableDateResourceItem
from hms_framework.utils import date_range


class GuestsDaily(CountableDateResource):

    def __init__(self, customer_model: ModelBase):
        self.customer_model = customer_model

    def items(self) -> List[CountableDateResourceItem]:
        list_countable_date_resource = []

        today = datetime.datetime.now()

        for single_date in date_range(start_date=today - datetime.timedelta(days=7), end_date=today):
            empty_rooms = random.randint(1, 20)
            list_countable_date_resource.append(
                CountableDateResourceItem(
                    date=single_date,
                    count=empty_rooms
                )
            )

        return list_countable_date_resource
```

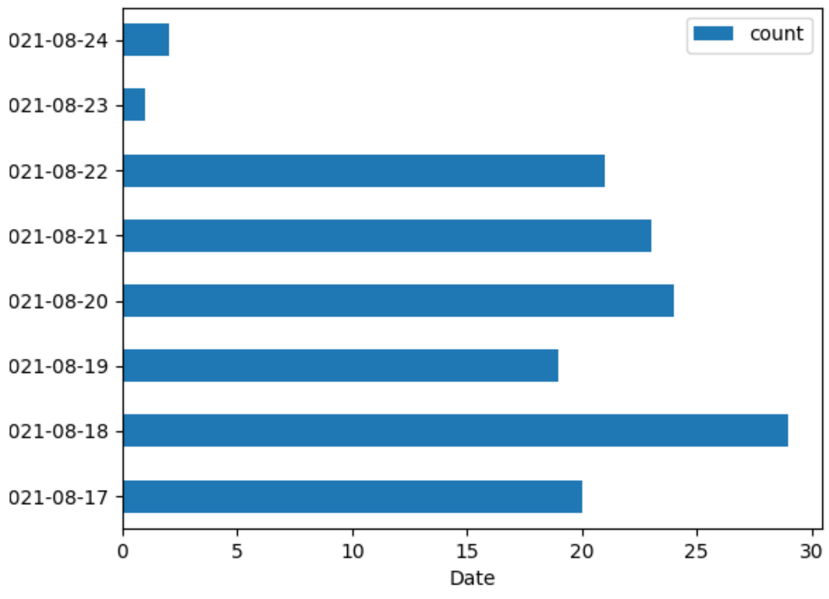
UI Result

Occupancy report

Guest Daily Chart

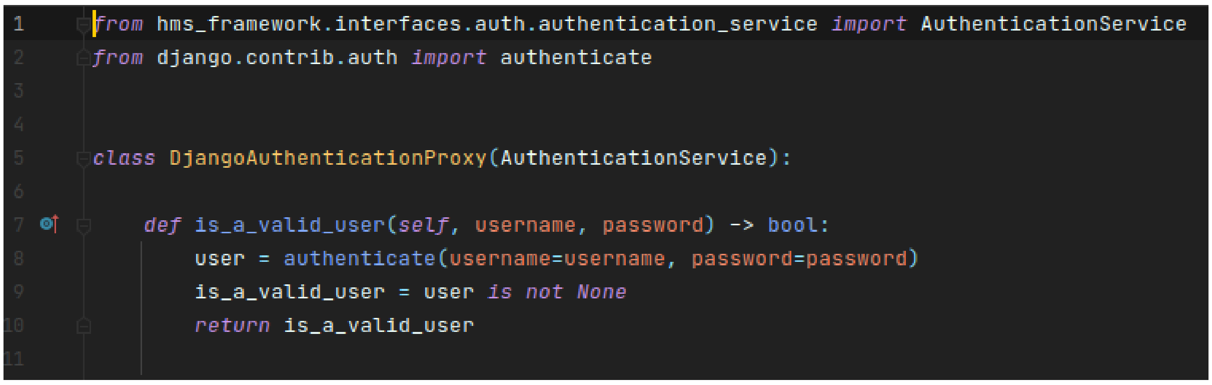


Empty Rooms Chart



Proxy

Used to decouple some Django functions as well as external libraries, so upgrading them will require to change only the Proxy classes.



```python
from hms_framework.interfaces.auth.authentication_service import AuthenticationService
from django.contrib.auth import authenticate


class DjangoAuthenticationProxy(AuthenticationService):

    def is_a_valid_user(self, username, password) -> bool:
        user = authenticate(username=username, password=password)
        is_a_valid_user = user is not None
        return is_a_valid_user

```

Code Metrics

Radon has been used to analyze Python source files and compute Cyclomatic Complexity.

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.(Cyclomatic complexity - Wikipedia)

Install:

```
pip install radon
```

Usage:

```
radon cc -e "*test*" *
```

Ranking goes from A to F, being A not complex and F really complex.

```
hms_framework/admin.py
        C 12:0 RoomAdmin - A
hms_framework/utils.py
        F 35:0 date_range - A
        C 8:0 TimeDiff - A
        F 40:0 class_for_name - A
        M 10:4 TimeDiff.__init__ - A
        M 15:4 TimeDiff.stop - A
        M 21:4 TimeDiff.step - A
        M 28:4 TimeDiff.print_stop - A
        M 31:4 TimeDiff.print_step - A
hms_framework/models.py
        M 258:4 Customer.save_memento - A
        C 344:0 Booking - A
        M 237:4 Customer.diff - A
        C 286:0 Invoice - A
        M 297:4 Invoice.invoice_items_total - A
        M 356:4 Booking.total_amount - A
        C 19:0 Hotel - A
        C 45:0 Address - A
        C 71:0 City - A
        C 89:0 Country - A
        C 117:0 Room - A
        C 140:0 RoomType - A
        C 149:0 RoomFeatureType - A
        C 166:0 BedType - A
        C 181:0 RoomPricePeriod - A
        C 214:0 Customer - A
        M 252:4 Customer._dict - A
        M 275:4 Customer.undo - A
        M 21:4 Hotel.__str__ - A
        M 48:4 Address.__str__ - A
        M 74:4 City.__str__ - A
        M 92:4 Country.__str__ - A
        M 119:4 Room.__str__ - A
        M 142:4 RoomType.__str__ - A
        M 151:4 RoomFeatureType.__str__ - A
        M 168:4 BedType.__str__ - A
        C 175:0 RoomTypePicture - A
        M 183:4 RoomPricePeriod.__str__ - A
        C 207:0 CustomerMemento - A
        M 216:4 Customer.__str__ - A
        M 233:4 Customer.__init__ - A
        M 244:4 Customer.has_changed - A
        M 248:4 Customer.changed_fields - A
        M 270:4 Customer.save - A
        M 281:4 Customer.accept - A
        M 308:4 Invoice.accept - A
        C 313:0 InvoiceItem - A
        C 325:0 InvoicePayment - A
hms_framework/factory.py
        C 41:0 BookingFactory - A
        M 45:4 BookingFactory.make_booking_service - A
        C 26:0 CustomerFactory - A
        C 75:0 RoomFactory - A
        C 80:0 RoomTypeFactory - A
        C 85:0 AddressFactory - A
        C 90:0 CountryFactory - A
        C 95:0 CityFactory - A
        C 100:0 AuthFactory - A
        C 107:0 InvoiceFactory - A
        C 128:0 InvoiceItemFactory - A
        C 133:0 InvoicePaymentFactory - A
        C 149:0 FinancialFactory - A
        C 156:0 UserFactory - A
        C 161:0 ChartFactory - A
        M 27:4 CustomerFactory.create_model - A
        M 30:4 CustomerFactory.create_customer_service -
A
        M 42:4 BookingFactory.create_model - A
        M 68:4 BookingFactory.search_availability_service -
A
        M 76:4 RoomFactory.create_model - A
        M 81:4 RoomTypeFactory.create_model - A
        M 86:4 AddressFactory.create_model - A
        M 91:4 CountryFactory.create_model - A
        M 96:4 CityFactory.create_model - A
        M 101:4 AuthFactory.create_service - A
        M 108:4 InvoiceFactory.build_invoice_service - A
        M 124:4 InvoiceFactory.create_model - A
        M 129:4 InvoiceItemFactory.create_model - A
        M 134:4 InvoicePaymentFactory.create_model - A
        M 137:4
InvoicePaymentFactory.mark_payment_service - A
        M 150:4 FinancialFactory.debt_collector_service - A
        M 157:4 UserFactory.create_model - A
        M 162:4 ChartFactory.empty_rooms_daily - A
        M 168:4 ChartFactory.guests_daily - A
hms_framework/forms.py
        C 8:0 BookingForm - A
        M 18:4 BookingForm.__init__ - A
```

Non-Functional requirements

Security:

Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or Web services, whenever the data is not sufficiently sanitized before including in a page.(Security in Django)
Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof.(Security in Django)

Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.(Security in Django)

Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate. However, as with any mitigation technique, there are limitations.(Security in Django)

SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.(Security in Django)

Django's querysets are protected from SQL injection since their queries are constructed using query parameterization. A query's SQL code is defined separately from the query's parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.(Security in Django)

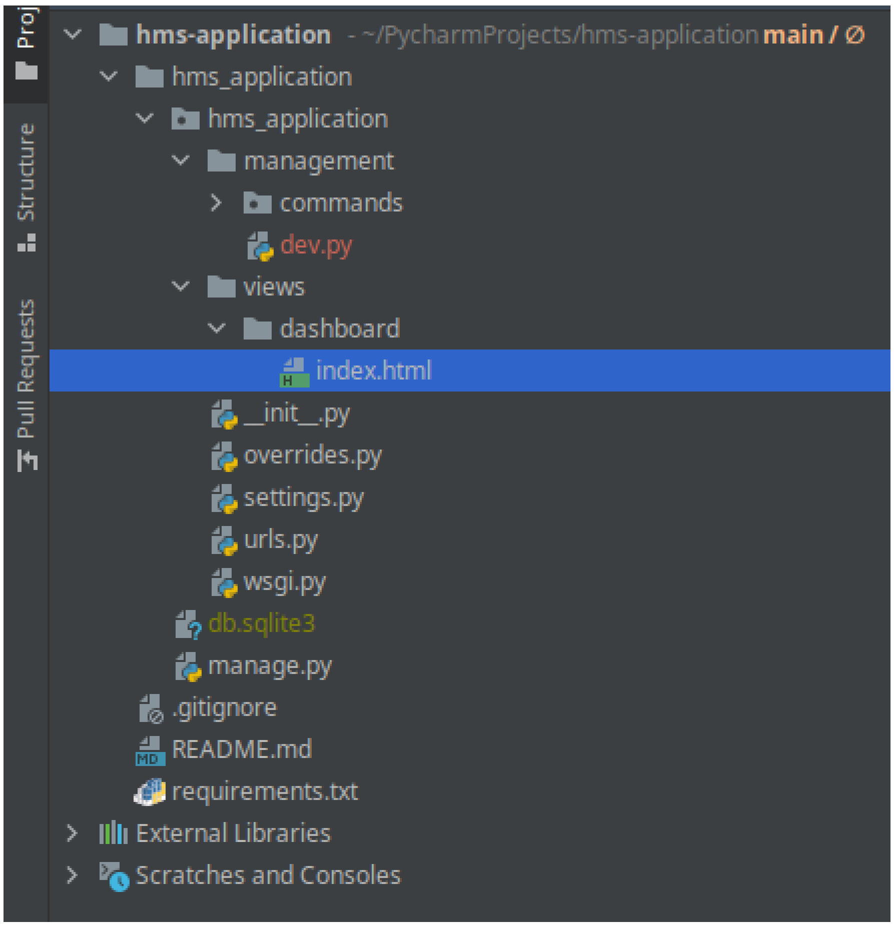
SSL/HTTPS

It is always better for security to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases, active network attackers, to alter data that is sent in either direction.(Security in Django)

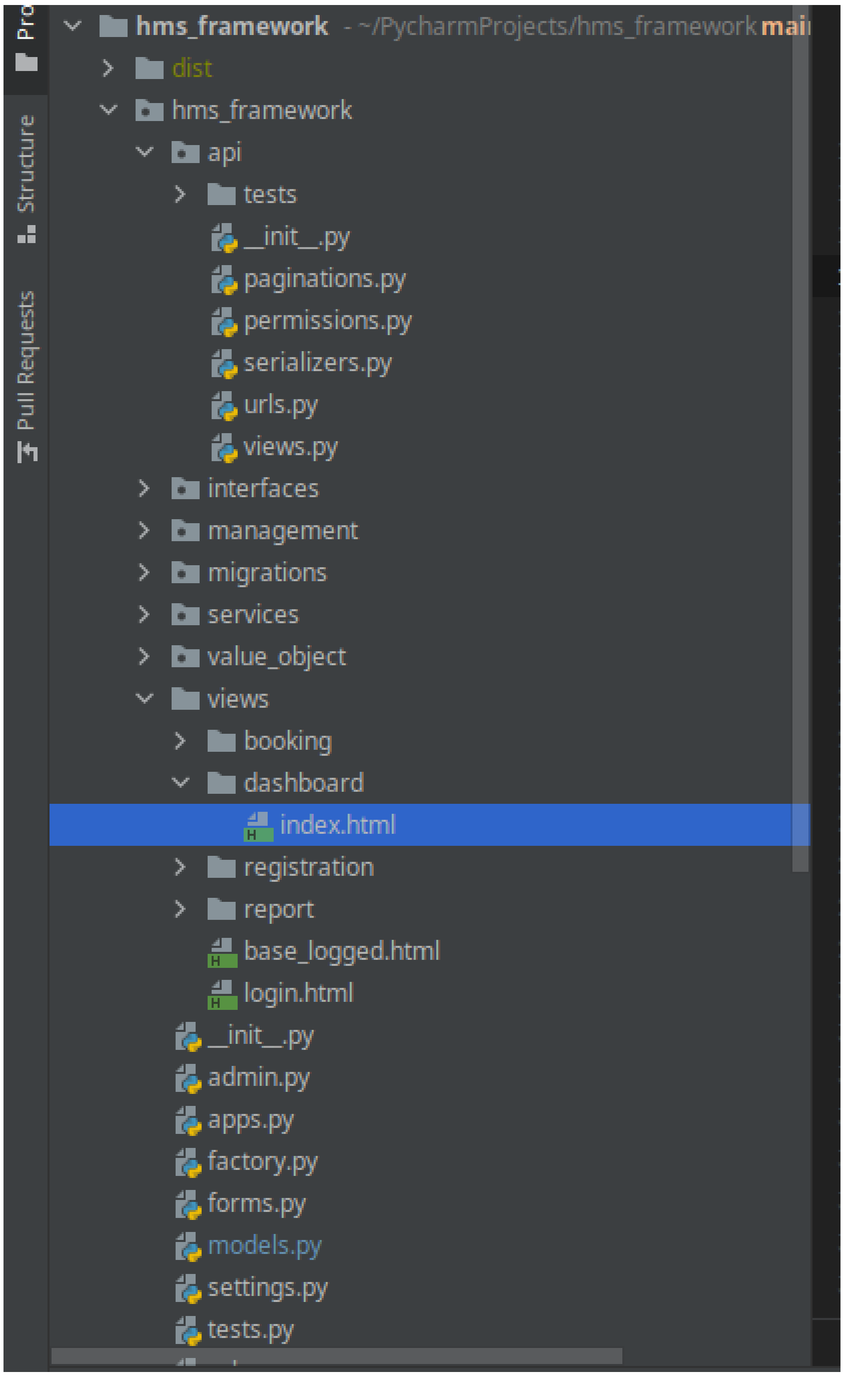
Extensibility

Application

The application can extend all html views by overriding them. Example:



```
hms-application - ~/PycharmProjects/hms-application main / ∅
  hms_application
    hms_application
      management
        commands
        dev.py
      views
        dashboard
          index.html
      __init__.py
      overrides.py
      settings.py
      urls.py
      wsgi.py
    db.sqlite3
    manage.py
  .gitignore
  README.md
  requirements.txt
External Libraries
Scratches and Consoles
```



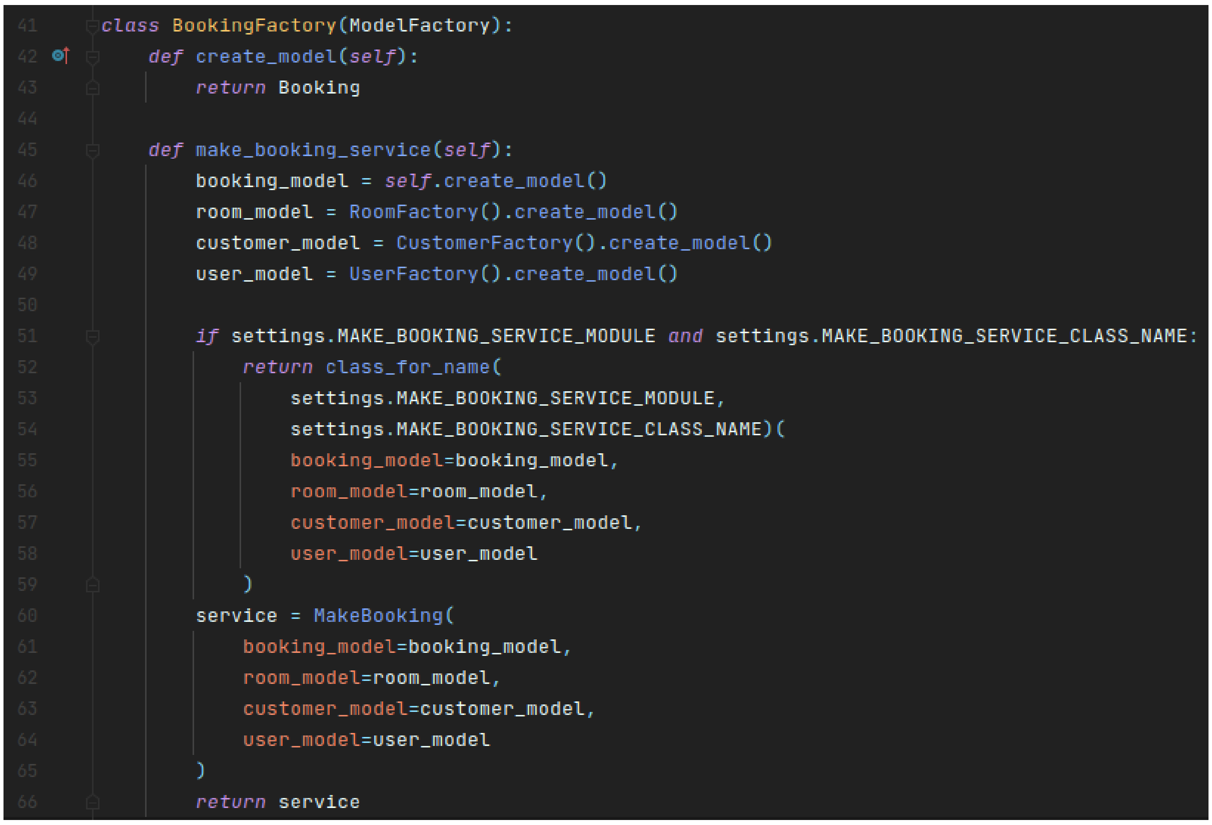
```
hms_framework - ~/PycharmProjects/hms_framework mai
  dist
  hms_framework
    api
      tests
      __init__.py
      paginations.py
      permissions.py
      serializers.py
      urls.py
      views.py
    interfaces
    management
    migrations
    services
    value_object
    views
      booking
      dashboard
        index.html
      registration
      report
      base_logged.html
      login.html
    __init__.py
    admin.py
    apps.py
    factory.py
    forms.py
    models.py
    settings.py
    tests.py
```

Classes can also be extended as for example the MakeBooking Service, currently used by the UI and REST Api.

From **Application Project** -> settings.py you can defined to which class to replace it:

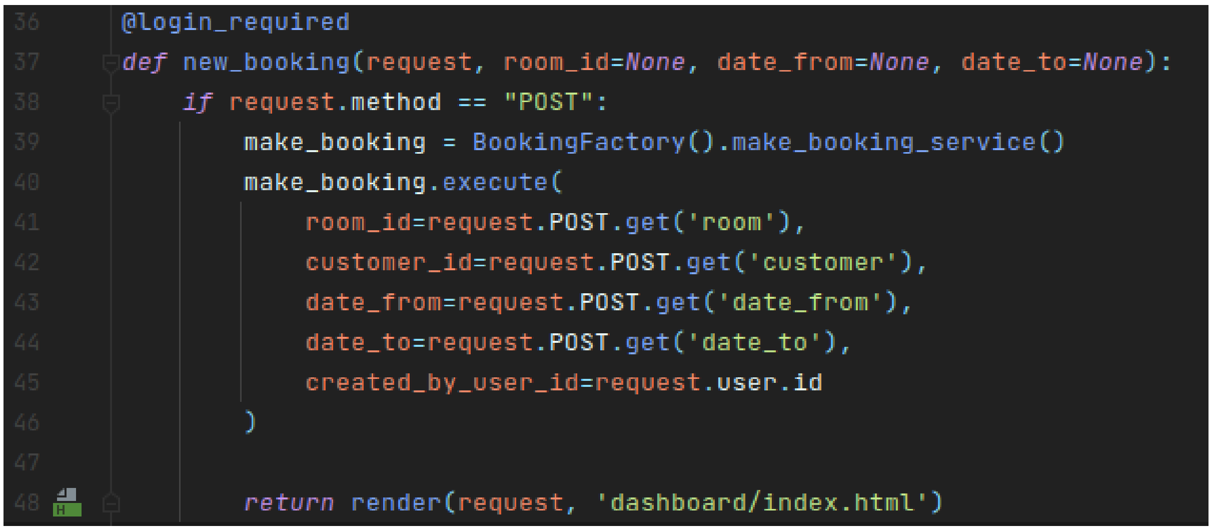
MAKE_BOOKING_SERVICE_MODULE = 'hms_application.overrides'
MAKE_BOOKING_SERVICE_CLASS_NAME = 'MakeBookingOverride'

The factory method located in the **Framework Project** will take care of returning the correct class:



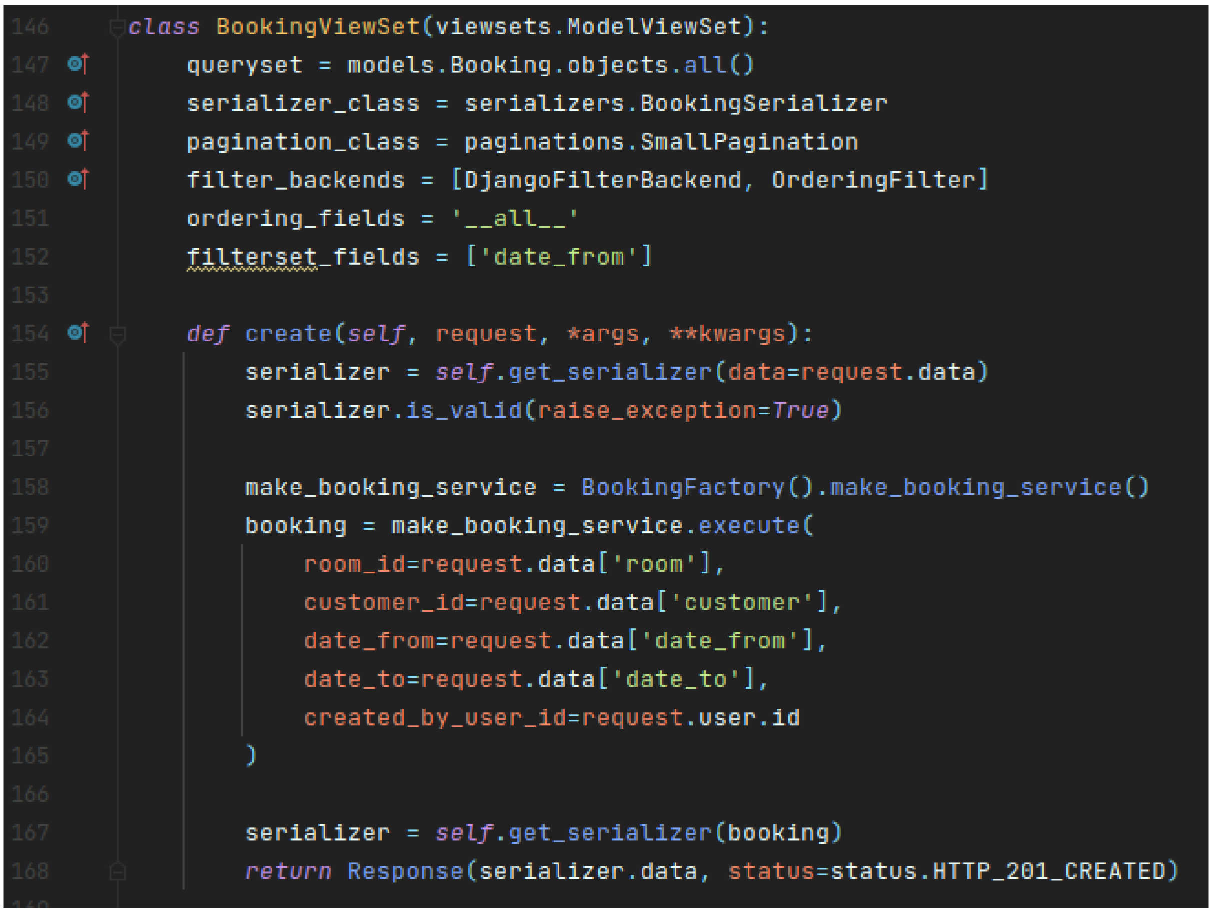
```python
class BookingFactory(ModelFactory):
    def create_model(self):
        return Booking

    def make_booking_service(self):
        booking_model = self.create_model()
        room_model = RoomFactory().create_model()
        customer_model = CustomerFactory().create_model()
        user_model = UserFactory().create_model()

        if settings.MAKE_BOOKING_SERVICE_MODULE and settings.MAKE_BOOKING_SERVICE_CLASS_NAME:
            return class_for_name(
                settings.MAKE_BOOKING_SERVICE_MODULE,
                settings.MAKE_BOOKING_SERVICE_CLASS_NAME)(
                booking_model=booking_model,
                room_model=room_model,
                customer_model=customer_model,
                user_model=user_model
            )
        service = MakeBooking(
            booking_model=booking_model,
            room_model=room_model,
            customer_model=customer_model,
            user_model=user_model
        )
        return service
```

MakeBooking Service is then used in the Html Action, it returns a rendered view to the browser.



```python
@login_required
def new_booking(request, room_id=None, date_from=None, date_to=None):
    if request.method == "POST":
        make_booking = BookingFactory().make_booking_service()
        make_booking.execute(
            room_id=request.POST.get('room'),
            customer_id=request.POST.get('customer'),
            date_from=request.POST.get('date_from'),
            date_to=request.POST.get('date_to'),
            created_by_user_id=request.user.id
        )

        return render(request, 'dashboard/index.html')
```

MakeBooking service is also used by the Rest API, it returns a json to the http client.



```python
class BookingViewSet(viewsets.ModelViewSet):
    queryset = models.Booking.objects.all()
    serializer_class = serializers.BookingSerializer
    pagination_class = paginations.SmallPagination
    filter_backends = [DjangoFilterBackend, OrderingFilter]
    ordering_fields = '__all__'
    filterset_fields = ['date_from']

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        make_booking_service = BookingFactory().make_booking_service()
        booking = make_booking_service.execute(
            room_id=request.data['room'],
            customer_id=request.data['customer'],
            date_from=request.data['date_from'],
            date_to=request.data['date_to'],
            created_by_user_id=request.user.id
        )

        serializer = self.get_serializer(booking)
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

Performance

Same as the original project

The system must be responsive, quickly respond to inputs and requests from the user, any delays between the users' inputs and the system's response (where necessary) should be kept to a minimum. (Hotel Reservation Management System, CS5721)

Loading speed of the system has to be fast. Users should be waiting less than 500 milliseconds for standard pages and less than 2500 milliseconds for pages requiring significant database queries or mutations in order to maintain user engagement by keeping the site responsive.(Hotel Reservation Management System, CS5721)

Future Development

The framework will be extended based on clients feedback after release version 1.

DevOps

Local Development

1. Install python3.7
   ```
   sudo apt install python3.7
   ```
2. Create workspace folder:
   ```
   mkdir ~/h
   cd ~/h
   ```
3. Git clone both repositories:
   ```
   git clone https://github.com/bruno911/hms-application
   git clone https://github.com/bruno911/hms-framework
   ```
4. Create virtual env on framework:
   ```
   mkdir ~/h/python-venvs
   python3.7 -m virtualenv ~/h/python-venvs/hms-framework
   ```
5. Activate virtual environment:
   ```
   source ~/h/python-venvs/hms-framework/bin/activate
   ```
6. Install framework dependencies:
   ```
   cd ~/h/hms-framework/
   python -m pip install -r requirements.txt
   ```
7. Compile the framework:
   ```
   python setup.py sdist
   ```
8. Hard code the path of the tar.gz into the application project:
   ```
   cd ~/h/hms-application
   vim requirements.txt
   ```
   Append:
   ```
   file:PUT_FULL_PATH_HERE/hms-framework/dist/hms-framework-0.1.tar.gz
   ```
9. Create virtual env on application:
   ```
   python3.7 -m virtualenv ~/h/python-venvs/hms-application
   ```
10. Activate virtual environment:
    ```
    source ~/h/python-venvs/hms-application/bin/activate
    ```
11. Install application dependencies:
    ```
    cd ~/h/hms-application/
    python -m pip install -r requirements.txt
    ```
12. Run migrations:
    ```
    cd ~/h/hms-application/hms_application/
    python manage.py makemigrations
    python manage.py migrate
    ```
13. Create a super user for you to use the system:
    ```
    python manage.py createsuperuser
    ```
14. Run local server:
    ```
    python manage.py runserver
    ```
15. Navigate to http://127.0.0.1:8000/admin log in with the new user you have created. Use the admin or go to main system: http://127.0.0.1:8000/

After following this you can also run as a server the framework in standalone so that you can see the API documentation.

1. Stop previous server with control+c and go to:
   `cd ~/h/hms-framework/`
2. Activate virtual environment:
   `source ~/h/python-venvs/hms-application/bin/activate`
3. Run migrations:
   `python manage.py makemigrations`
   `python manage.py migrate`
4. Create a super user for you to use the system:
   `python manage.py createsuperuser`
5. Run local server:
   `python manage.py runserver`
6. Navigate to http://127.0.0.1:8000/admin log in with the new user you have created.
7. Visit: http://127.0.0.1:8000/api/v1/ and http://127.0.0.1:8000/api/docs/

Deploy Production

Use github circles CI to run py.test, all tests must pass in order to push/merge new code.
Use Jenkins with Webhook from github that will trigger a deployment everytime a new “git tag” is created.
Use git tags to keep versioning.

Unit Test

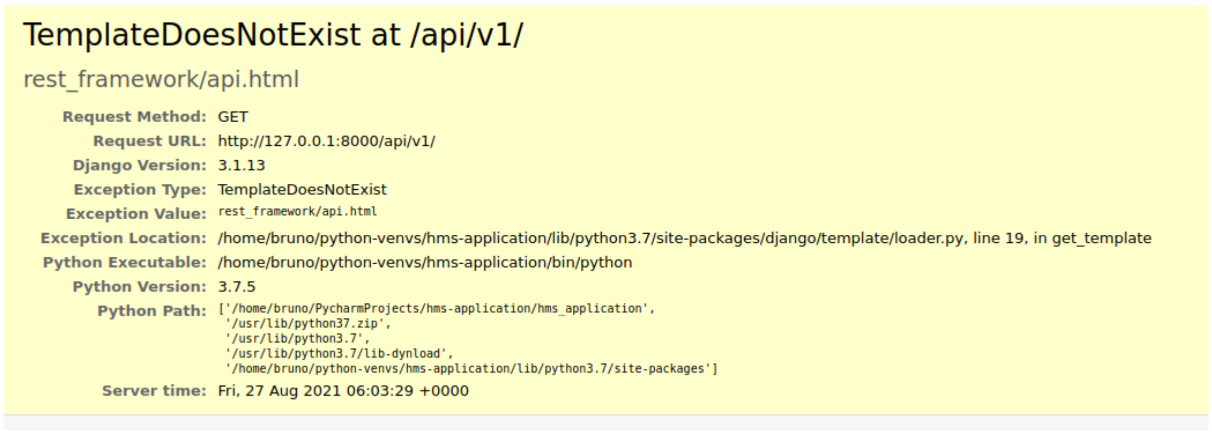
Tests have been written to ensure that code works as expected. The unit tests have been written in a self-documentary code approach so that developers can use it as a way to understand the code and logic behind each part of the system.

In order to run the tests, follow steps for Local installation first, and then:

1. Navigate to the framework project.
   `cd ~/h/hms-framework/`
2. Activate virtual environment:
   `source ~/h/python-venvs/hms-framework/bin/activate`
3. Run the tests:
   `py.test`

Known Issues

API Docs won't open from the Application project, but it will open from the Framework project.



```
TemplateDoesNotExist at /api/v1/
rest_framework/api.html
    Request Method:  GET
       Request URL:  http://127.0.0.1:8000/api/v1/
    Django Version:  3.1.13
    Exception Type:  TemplateDoesNotExist
   Exception Value:  rest_framework/api.html
Exception Location:  /home/bruno/python-venvs/hms-application/lib/python3.7/site-packages/django/template/loader.py, line 19, in get_template
 Python Executable:  /home/bruno/python-venvs/hms-application/bin/python
    Python Version:  3.7.5
       Python Path:  ['/home/bruno/PycharmProjects/hms-application/hms_application',
                      '/usr/lib/python37.zip',
                      '/usr/lib/python3.7',
                      '/usr/lib/python3.7/lib-dynload',
                      '/home/bruno/python-venvs/hms-application/lib/python3.7/site-packages']
       Server time:  Fri, 27 Aug 2021 06:03:29 +0000
```

Login should temporarily happen from:
http://127.0.0.1:8000/admin
Then navigate to http://127.0.0.1:8000

References

Refactoring.guru. (n.d.). Code Smells. [online] Available at:
https://refactoring.guru/refactoring/smells.

Cyclomatic complexity - Wikipedia. [online] Available at:
https://en.wikipedia.org/wiki/Cyclomatic_complexity

Security in Django. [online] Available at:
https://docs.djangoproject.com/en/3.2/topics/security/

Checking Code Metrics with Radon & Python (Code Refactoring). [online] Available at:
https://www.youtube.com/watch?v=tnsl1ZagnrQ

Dependency injection and inversion of control in Python. [online] Available at:
https://python-dependency-injector.ets-labs.org/introduction/di_in_python.html

Quick start with Django ORM. [online] Available at:
https://swapps.com/blog/quick-start-with-django-orm/

Bridge Pattern – Design Patterns (ep 11)
https://www.youtube.com/watch?v=F1YQ7YRjttI&t=2735s

Hotel Reservation Management System, CS5721 SOFTWARE DESIGN, Lecturer: Dr. Anila Mjeda, Semester 1: 2020 – 2021, Group 19
David O'Riordan, Bruno Quintana, Kristina Rutkauskaite, TongLing Liu, Ignatius Bownes