React Django Patterns and Structuring

This document contains patterns and structuring for Enterprise Web Application using React and Django Development Stack.

Module Structure

There has to be an integrated Django-React Application. Each Django Project will have Django apps. Each app will have logically grouped submodules.

An Example Project Structure:

Consider the folders and files over and above the DRF and Django framework files:

```
VendorProject
|- commons
    |- io
        |- MOSDMSRepo.py
        |- FSRepo.py
        |- MQ.py
        |- Cache.py
        |- LDAPProvider.py
        |- ...
    |- utilities
        |- auth
        |- ...
    |- validators
        |- validators.py
|- src
|- MDM / api
|- VendorProject
    |- constants.py
    |- utils
        |- ...
    |- responses.py
    |- value_objects.py
    |- ...
|- procurement
|- vendor
|- ...
```

Folder	Description
commons	Commons library, such utility functions, network infrasctructure, generic validators etc.
src	src folder for React App.
MDM / api	Master Data manegement App and its API

Folder	Description
procurement	Example Django App related to Procurement Submodule.
vendor	Example Django App related to Vendor Submodule.

Django App / Sub-module Structure

Over an above the DRF and Django framework files, consider the following structure:

```
[django_app / submodule]
|-...
|- constants.py
|- entities.py
|- exceptions.py
|- interface_tests.py
|- managers.py
|- responses.py
|- models.py
|- serializers.py
|- service.py
|- tests.py
|- urls.py
|- validators.py
|- value_objects.py
|- views.py
|-...
```

Folder	Description
constants.py	App / Module level constants.
entities.py	Proxy models which contains the Business Logic related to business model. Each functions should be atomic and single responsibility. The methods should be pure methods should not use anything other than instance variables and args.
exceptions	Globals for app/module level exceptions
interface_tests.py	test files relates to interface testing / Load Testing on API.
managers.py	Model manager which will do something analogous to a DAO.
models.py	Django Models
responses.py	Custom responses defined for the routes / api endpoints.
serializers.py	DRF Serializers.
service.py	Service Classes for a given module for each entities

Folder	Description
tests.py	Unit Tests for given module
urls.py	Django URL Mapping
validators.py	Validation logic related to Domain Modules. These validators can be injected in related serializers / forms / other orchestration logics.
views.py	API Views. Ideally views should call only the service classes for orchestration, workflow and wiring logic. Views should be free from logical detailing.

Domain Models - Django Models + Proxy Models

Since Django Models of Django Database API, are replication of tables, they are represent fields as class attributes instead of instance attributes.

There has to be two representations here:

1. Django Model for Field Representation

This will contain DB Specific Information for mapped to the domain fields:

In `vendor/models.py`:

```
class VendorModel(models.Model):
    vendor_name = models.TextField(max_length=50)
    vendor_category_id = models.IntegerField()
    created_by = models.ForeignKey(to=User, on_delete=models.CASCADE)
    created_on = models.DateTimeField()


class WorkOrderModel(models.Model):
    vendor = models.ForeignKey(to=VendorModel, on_delete=models.CASCADE)
    amount = models.FloatField()
    ### obviously there are going to be more details here
```

2. Proxy Models for Business Logic

This will be proxy model to the Django model related to a given entity and will contain business logic of the model.

This class will not have fields, it will have member functions containing only the business logic. The member functions will not use any of the infrastructure or any other related objects. These member functions will contain pure functions with pure business logic. Also these member functions will use only member fields, method fields or variables found in the arguments for processing a business logic.

All business logic function should be written in single responsibility, atomic functions.

In `vendor/entities.py`:

```
class VendorWorkorderOverLimitException(Exception):pass

class Vendor(VendorModel):
    MAX_ALLOWED_WORKORDERS = 2  # Ideally should come from DB

    def can_allocate_workorder(self, vendorWorkorders,
MAX_ALLOWED_WORKORDERS ):
        return not len(vendorWorkorders) >= MAX_ALLOWED_WORKORDERS

    class Meta:
        proxy = True



class WorkOrder(WorkOrderModel):

    class Meta:
        proxy = True
```

3. TDD of Business Models

Each Business models should be tested for all uses cases and functional requirements irrespective of persistence:

```
# Create your tests here.
class VendorTestCases(TestCase):
    obj = None

    def setUp(self):
        ## creating user
        User.objects.create_user("John",password="John")

Vendor.objects.create_vendor(vendor_name="John",vendor_category_id=1,
created_by=User.objects.get(pk=1),created_on=datetime.now())

    def test_create_vendor(self):
        ## check if user is created

self.assertEqual(Vendor.objects.get_vendor_by_id(1).vendor_name,"John")
        ## can be more elaborative

    def test_add_work_order(self):
        ## load a vendor
        vendor = Vendor.objects.get_vendor_by_id(1)
        workorders = vendor.workordermodel_set.count()
        if vendor.can_allocate_workorder(workorders,
Vendor.MAX_ALLOWED_WORKORDERS):
                vendor.allocate_workorder(amount =10)
                self.assertEqual(vendor.workordermodel_set.count(),
workorders+1)
## and so on
...
```

Django Model Manager

There has to be a Django Model Manager as a DAO (Not in literal terms but to be taken as analogy), which manages are Data Access logic of the Entity:

A Sample Usage

There has to be a base manager which takes care of common DAO logic.

At `commons.manager.py`:

```
## at a common package
class BaseManager(models.Manager):
    def get_all(self):
        return super(BaseManager, self).all()

    def get_by_id(self, id):
        return super(BaseManager, self).get(id=id)

    def get_by_filter(self, **filters):
        return super(BaseManager, self).filter(**filters)

    # and so on...

    ...
```

And there should be specific concretion of Entity Specific DAO Classes:

At `vendor/manager.py`:

```
class VendorManager(BaseManager):

    def get_vendor_workorders(self, vendor):
        return vendor.workordermodel_set.all()

    def get_vendor_workorders_filter(self, **filters):
        return self.workorder_set.filters(**filters)
    def allocate_work_order(self, vendor, **workOrderDict):
        vendor.workordermodel_set.create(**workOrderDict)

    ## and so on

    ...


class WorkOrderManager(BaseManager):

    def create_workorder(self, **kargs):
        return self.create(**kargs)


    ...
```

Test Driven Development:

Similar to Business Model, all unit tests should be conducted for these Unit of Work.

Service Layer

The Service layer will contain the :

- Orchestration
- Workflow and
- Wiring logic

and will be used by the APIs and interfaces:

```
class VendorService:

    # Add vendor
    def add(self, **vendorDict):
        Vendor.objects.create(**vendorDict)

    # Fetch a vendor
    def get_vendor(self, vendorId):
        return Vendor.objects.get_by_id(vendorId)

    # Search a vendor
    def search(self, **args):
        return Vendor.objects.get_by_filters(**args)

    # Allocate Workorder to a vendor
    def allocate_workorder(self, vendorId, **workorder):
        vendor = self.get_vendor(vendorId)
        vendor_workorders = Vendor.objects.get_vendor_workorders(vendor)
        if vendor.can_allocate_workorder(vendor_workorders,
Vendor.maxAllowedWorkOrders):
            Vendor.objects.allocate_work_order(vendor, **workorder)
        else:
            raise VendorWorkorderOverLimitException("Cannot allocate more
than {limit} workorders to a
vendor".format(limit=Vendor.maxAllowedWorkOrders))
```

Domain Events using Signals and Distrbuting to message bus

At certain instances, code needs to be decoupled from service and API code. For example, email sending code should be decoupled from service code by registering mail triggers to a Domain Event. Signals should be used to fire an event and do some operation listening to it. Signals can be implemented by:

1. Listening to default hooks / event of the Django Model
2. Defining custom Signals

Listening to default hooks/events:

Example:

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save

def save_profile(sender, instance, **kwargs):
    instance.profile.save()

post_save.connect(save_profile, sender=User)
```

Some inbuilt hooks / events with Django Model and Request / Response:

```
django.db.models.signals.pre_init:

django.db.models.signals.post_init:

django.db.models.signals.pre_save:

django.db.models.signals.post_save:

django.db.models.signals.pre_delete:

django.db.models.signals.post_delete:

django.db.models.signals.m2m_changed:

django.core.signals.request_started:
```

```
django.core.signals.request_finished:

django.core.signals.got_request_exception:
```

Refer Django Documentation for more details.

Custom Signals

Custom signal should be created, where required, on different application events:

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

Sending Signals:

```
class PizzaStore:
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, 
size=size)
        ...
```

Synchronous / Asynchronous:

Signal events are not Async. For async behavior these signals need to couple with MQ / Redis for Async behavior. The detailing about these async coupling will be elaborated seperately.

Value Objects

Create value objects where required. It helps convert values logically into objects. It also helps logical comparison of objects:

`somepackage/value_objects.py`

```
...


class InvalidEmailAddressException(Exception):
    pass

class EmailAddress(object):
    __email_address = None

    def __init__(self, email_address):
        ## should use internal/django validators instead of regex in below
example
        if re.match(r"(^[-!#$%&'*+/=?^_`{}|~0-9A-Z]+(\.[-!#$%&'*+/=?
^_`{}|~0-9A-Z]+)*"r'|^"([\001-\010\013\014\016-\037!#-\[\]-\177]|\\[\001-
011\013\014\016-\177])*"'r')@(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+
[A-Z]{2,6}\.?$', email_address, re.IGNORECASE):
            self.__email_address = email_address
        else:
            raise InvalidEmailAddressException

    def __repr__(self):
        return self.__email_address

...
```

React Patterns

React App Structure

A proposed structure looks like below:

```
src
|- App
    |- AppConfig.js
    |- AppCustomStyling.css
    |- Utils
        |- APICommunication.jsx
        |- Configs.jsx
        |-...
|- cypress
    |-..
|- cypress.json
|- MDM
    |- [MDMRelated Folders]
    |-...
|- UI
    |- Content
    |- Header
    |- Navbar
    |- Page
    |- Routing
        |- ScreenRouters.js
|- [Module]
    |- ScreenRouters.js
    |- ModuleConstants.js
```

```
      |- [Module]CustomStyling.css
      |- Screens
          |- [ScreenName]
              |- [MainComponent].jsx
              |- Components
                  |- Component1.jsx
                  |- ...
              |- [ComponentDataAccess].js
              |- [Style].scss
              |- __tests__
```

Folder	Description
App	App Level Consfigs and Styling
App/AppConfig.js	App level configs and globals.
App/AppCustomStyling.css	App level styling constants
App/Utils	App Level Utilities
App/Utils/APICommunication	API Communication Service. This will contain response handlers for the fecth/axios request handlers.
App/Utils/Configs	Utility Configs
cypress	folder containing cypress tests (integration tests).
cypress.json	cypress config.
MDM	MDM module folder
UI	Folder which has global layouts and templates of the UI
UI/Content	Layout and Template for Page Content
UI/Header	Layout and Template for Header of the Screen
UI/Navbar	Layout and Template for Left Navigation Bar of the Screen
UI/Page	Layout and Template for Page / Screen
UI/Routing	Routing Constants
UI/Routing/MasterRouters.js	Master Routing Constants
UI/Routing/ScreenRouters.js	Global Screen Constants
[Module]	Module level folder for logically grouped screens.
[Module]/ScreenRouters.js	Module Level routing constants
[Module]/ModuleConstants.js	Module Level constsants
[Module]/[Module]CustomStyling.css	Module lecel styling guidelines
[Module]/Screens	Folder Containing Screens of the given module.

Folder	Description
[Module]/Screens/ [ScreenName]	Folder containing JS/JSX files of a given module.
[Module]/Screens/ [ScreenName]/components	Single Responsibility components for a relative screen
[Module]/Screens/ [ScreenName]/**tests**	Unit test files for a given screen / submodule.
[Module]/Screens/ [ScreenName]/ComponentDataAccess	Data providers.

Atomic and Single Responsibility Components:

Every component class should follow single responsibilty principle. Each screen component should have modular small Atomic components which will be integrated at screen level. For example, in below structure, `MainComponent` is the screen level document, and all related components are in `components` folder. These components will be merged with composition in the `MainComponent`.

```
...
|
|- [Module]
    |- ...
    |- Screens
        |- [ScreenName]
            |- [MainComponent].jsx
            |- Components
                |- Component1.jsx
                |- ...
            |- [ComponentHandlers].js
            |- [Style].scss
            |- __tests__
```

It is suggested that wherever possible, these components should follow Open Closed procedure ("Open for Extension, Closed for Modification"). If we you want to change how components need to be arranged or formatted, there should not be change in the atomic components, it should be handled at the component where you are composing these atomic components.

Some things to follow while passing the props:

1. Make sure you dont use spread operation {...} recklessly and pass unrelated values.
2. Also dont pass whole object in props instead pass only required fields to maintain interface segregation.

Managing Dependency Injection

All components should depend on abstractions and not concretions.

1. Managing Inversion Flow

2. Managing Handlers:

    a. The components should not depend on infrastructure detailing of how values are fetched from HTTP or local storage, or cookie. b. The components should rely on abstraction and work with deduced data.

Example:

```
class UserTable extends React.Component {
    ...

    componentDidMount=()=>{
        fetch(YOUR_URL).then(data=>{
            this.setState({data})
        })
    }
}
```

The above component is coupled with concrete implmentation. Which should be decoupled in implementation details by letting the handler / data-providers do the job or you:

```
class UserDataProvider {
    ...
    async loadUsers() {
        const users = await fetch(YOUR_URLS);
        // and some other config headers. Ideally it should be from comms
functions
        return users.
    }

    ...

    async saveUser(){
        /// save users
    }
}

class UserTable extends React.Component {
    ...

    componentDidMount=()=>{
        const users = userDataProvider.loadUsers()
        this.setState({users})
    }
```

```
  }
```