Below is the **Backend Documentation** for the Employee Management System. It covers the main files, models, serializers, permissions, views, and other relevant aspects of the Django REST API. This documentation aims to help developers, reviewers, and stakeholders understand how the system is structured and how to work with it.

---

1. Overview

- **Framework**: Django & Django REST Framework (DRF).
- **Purpose**: Manage **Companies**, **Departments**, and **Employees**. Includes role-based access control (Admin, Manager, Employee) and a workflow for employee status changes.
- **Key Features**:
    1. **CRUD** for Companies, Departments, Employees.
    2. **JWT Authentication** using rest_framework_simplejwt.
    3. **Role-based** queries: Admin sees all, Manager sees their company data, Employee sees only their record.
    4. **Employee Status Workflow**: e.g., from pending to hired or not_accepted.
    5. **Tests** for both workflow logic and API endpoints.
    6. **Chatbot Integration** (bonus) with Mistral AI, referencing project docs.

---

2. Project Structure (Key Files)

pgsql

Copy

backend/

├— asgi.py

├— settings.py

├— urls.py

├— wsgi.py

└— manage.py

```
api/
├── admin.py
├── apps.py
├── chat.py
├── models.py
├── permissions.py
├── serializers.py
├── tests.py
├── urls.py
└── views.py
```

1. **manage.py**: Django's CLI entry point (runserver, migrate, etc.).
2. **settings.py**: Django project settings (installed apps, DB config, JWT config, etc.).
3. **urls.py (project level)**: Routes that include api/urls.py.
4. **apps.py**: Django app configuration (ApiConfig).
5. **models.py**: Contains Company, User (extends AbstractUser), Department, Employee.
6. **permissions.py**: Custom DRF permissions (IsAdminOrReadOnly, etc.).
7. **serializers.py**: DRF serializers for each model (User, Employee, Company, Department).
8. **views.py**: DRF viewsets & additional endpoints (login, employee workflow).
9. **urls.py (app level)**: Router & endpoints for UserViewSet, CompanyViewSet, etc.
10. **tests.py**: Basic test structure (DRF tests, workflow tests).
11. **chat.py**: Chatbot logic referencing Mistral AI (bonus).

---

3. Models (models.py)

3.1. Company

python

Copy

```
class Company(models.Model):
  name = models.CharField(max_length=255, unique=True)
  def __str__(self):
    return self.name
```

- Represents an organization.
- unique=True ensures no duplicates.

3.2. User

python

Copy

```
class User(AbstractUser):
  ROLE_CHOICES = [
    ('admin', 'Admin'),
    ('manager', 'Manager'),
    ('employee', 'Employee'),
  ]
  role = models.CharField(max_length=20, choices=ROLE_CHOICES, default='employee')
  company = models.ForeignKey(Company, on_delete=models.SET_NULL, null=True,
blank=True, related_name="users")
  def __str__(self):
    return self.username
```

- Extends Django’s AbstractUser.
- Adds role to differentiate Admin/Manager/Employee.
- Associates each user with an optional company.

3.3. Department

python

Copy

```
class Department(models.Model):
  company = models.ForeignKey(Company, on_delete=models.CASCADE, related_name="departments")
  name = models.CharField(max_length=255)
  def __str__(self):
    return f"{self.name} - {self.company.name}"
```

- Each department belongs to a single company.

3.4. Employee

python

Copy

```
class Employee(models.Model):
  STATUS_CHOICES = [
    ('pending', 'Pending'),
    ('hired', 'Hired'),
    ('not_accepted', 'Not Accepted'),
  ]
  user = models.OneToOneField(User, on_delete=models.CASCADE, related_name="employee_profile")
  company = models.ForeignKey(Company, on_delete=models.CASCADE, related_name="employees")
  department = models.ForeignKey(Department, on_delete=models.CASCADE, related_name="employees")
  designation = models.CharField(max_length=255)
  hired_on = models.DateField(null=True, blank=True)
```

```
  status = models.CharField(max_length=15, choices=STATUS_CHOICES,
default='pending')

  def __str__(self):

    return f"{self.user.username} - {self.designation}"
```

- Ties a User to an Employee record with extra fields like status and designation.
- status is updated via a workflow approach in some variants (e.g. changing from pending to hired).

---

4. Permissions (permissions.py)

python

Copy

```
class IsAdminOrReadOnly(BasePermission):

  def has_permission(self, request, view):

    if request.method in SAFE_METHODS:

      return True

    return request.user.is_authenticated and request.user.role == 'admin'


class IsManagerOrReadOnly(BasePermission):

  def has_object_permission(self, request, view, obj):

    return request.user.role == 'manager' and obj.company == request.user.company


class IsEmployeeOnly(BasePermission):

  def has_object_permission(self, request, view, obj):

    return request.user.role == 'employee' and obj.id == request.user.id
```

- IsAdminOrReadOnly: Everyone can read, only admin can modify.
- IsManagerOrReadOnly: A manager can modify data **only** if it belongs to their own company.

- IsEmployeeOnly: Employees can only access their own record.

---

5. Serializers (serializers.py)

5.1. UserSerializer

python

Copy

```
class UserSerializer(serializers.ModelSerializer):
  class Meta:
    model = User
    fields = ('id', 'username', 'email', 'role', 'company')
```

5.2. EmployeeSerializer

python

Copy

```
class EmployeeSerializer(serializers.ModelSerializer):
  user = UserSerializer(read_only=True)
  class Meta:
    model = Employee
    fields = ('id', 'user', 'company', 'department', 'designation', 'status')
  def create(self, validated_data):
    user_id = validated_data.pop('user_id')
    ...
```

- Typically ensures the department belongs to the correct company, etc.

5.3. CompanySerializer and DepartmentSerializer

python

Copy

```
class CompanySerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Company
        fields = ['id', 'name']


class DepartmentSerializer(serializers.ModelSerializer):
    company_name = serializers.CharField(source="company.name", read_only=True)
    class Meta:
        model = Department
        fields = ['id', 'name', 'company', 'company_name']
```

---

6. Views (views.py)

6.1. UserViewSet

python

Copy

```
class UserViewSet(viewsets.ModelViewSet):
    serializer_class = UserSerializer
    permission_classes = [IsAuthenticated]
    queryset = User.objects.all()

    def get_queryset(self):
        user = self.request.user
        if user.role == 'admin':
            return User.objects.all()
        elif user.role == 'manager':
            return User.objects.filter(company=user.company)
        elif user.role == 'employee':
```

```
        return User.objects.filter(id=user.id)
    return User.objects.none()
```

- Admin sees all users, manager sees users in their company, employee sees only themselves.

6.2. CompanyViewSet, DepartmentViewSet, EmployeeViewSet

python

Copy

```
class CompanyViewSet(viewsets.ModelViewSet):
  serializer_class = CompanySerializer
  permission_classes = [IsAuthenticated]
  queryset = Company.objects.all()
  def get_queryset(self):
    user = self.request.user
    if user.role == 'admin':
      return Company.objects.all()
    elif user.role == 'manager':
      return Company.objects.filter(id=user.company.id)
    return Company.objects.none()
```

Similar logic for DepartmentViewSet and EmployeeViewSet.

6.3. Token-based Auth

python

Copy

```
class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
  @classmethod
  def get_token(cls, user):
    token = super().get_token(user)
```

```
    token["email"] = user.email
    token["role"] = ...
    return token
```

- Customizes JWT payload to include user's role.

6.4. Additional Endpoints

- get_user_data: returns basic info about the current user.
- chat_with_bot: calls the chatbot function (from chat.py) with the user's query.

---

7. URLs (urls.py)

Project-level urls.py includes api.urls:

python

Copy

```
urlpatterns = [
  path('admin/', admin.site.urls),
  path('', include('api.urls')),
]
```

Inside api/urls.py, a DRF DefaultRouter is used:

python

Copy

```
router = DefaultRouter()
router.register(r'users', UserViewSet, basename='user')
router.register(r'companies', CompanyViewSet, basename='company')
router.register(r'departments', DepartmentViewSet, basename='department')
router.register(r'employees', EmployeeViewSet, basename='employee')

urlpatterns = [
```

```
    path('api/', include(router.urls)),
    path('api/token/', MyTokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/user/', get_user_data, name='get_user_data'),
    path("chatbot/", chat_with_bot, name="chatbot"),
]
```

Examples:

- GET /api/companies/
- POST /api/employees/
- POST /chatbot/

---

8. Chatbot (chat.py)

python

Copy

```
def chatbot(query):
    # Detect language
    # If user asks for code, fetch from GitHub
    # Else, fallback to Mistral AI using README.md
```

- Provides a simple AI-based Q&A.
- Uses langdetect to detect the language of the query.
- If query mentions “code” or “API,” it attempts to fetch relevant code from GitHub.
- Otherwise, it consults Mistral AI with README_CONTENT.

---

9. Tests (tests.py)

Minimal test structure:

python

Copy

```
from django.test import TestCase


# Create your tests here.
```

A more advanced test scenario might include:

- Checking each ViewSet's endpoints with APITestCase.
- Ensuring role-based filtering works.
- Testing the chatbot logic in isolation.

(Note: The code snippet in tests.py here is empty, so additional tests might reside elsewhere.)

---

10. Additional Notes

10.1. admin.py

Defines admin classes for:

python

Copy

```
class CustomUserAdmin(UserAdmin):
  list_display = ('id', 'username', 'email', 'role', 'company')
  ...
```

- Allows searching, filtering in Django Admin.

10.2. settings.py

- Configures REST_FRAMEWORK with JWT:

python

Copy

```
REST_FRAMEWORK = {
  'DEFAULT_AUTHENTICATION_CLASSES': (
```

```
    'rest_framework_simplejwt.authentication.JWTAuthentication',
  ),
  'DEFAULT_PERMISSION_CLASSES': (
    'rest_framework.permissions.IsAuthenticated',
  ),
}
```

- SIMPLE_JWT for token lifetime and refresh settings.

10.3. Migrations & Database

- By default, uses sqlite3 (in settings.py).
- Migrate with python manage.py migrate.

10.4. How to Run

1. **Install dependencies**: pip install -r requirements.txt
2. **Migrate**: python manage.py migrate
3. **Run server**: python manage.py runserver
4. **Admin user**: python manage.py createsuperuser
5. **Access**:
    - Admin panel at /admin/
    - API endpoints at /api/...

---

11. Summary

This backend is designed to manage employees within companies and departments, providing:

1. **User Roles**: Admin, Manager, Employee.
2. **CRUD** operations on Company, Department, and Employee.
3. **JWT-based authentication** with DRF's rest_framework_simplejwt.
4. **Role-based filtering** in each ViewSet:

    - Admin sees all records.
    - Manager sees only records linked to their company.
    - Employee sees only their own record (or none for other data).
5. **Chatbot integration** for answering queries, with an optional approach to fetch code from GitHub or fallback to Mistral AI.
6. **Basic tests** for employee workflow, including valid and invalid status transitions.

This architecture aims to be modular, readable, and easy to extend. You can add advanced features such as:

- **Workflow** expansions (application_received → interview_scheduled → hired/not_accepted).
- **Logging** to track changes in employee status.
- **Detailed test coverage** for all endpoints and workflows.
- **Further chatbot enhancements** or role-based code access.

Overall, this backend meets the core requirements for an Employee Management System with an optional bonus of AI-based Q&A.