Project: Enterprise Task Management System Backend & Frontend

Overview

Duration: 5 working days **Objective:** Build a containerized full-stack task management platform demonstrating proficiency with our technology stack.

⚠ **Important Notes:**

- Docker delivery is MANDATORY - the project must run with `docker-compose up`
- The project scope is intentionally large - we expect 40-60% feature completion. The different points in the requirements are only to guide you on where we want you to focus, and you are not expected to complete every single feature, even in the mandatory part.
- Focus on quality over quantity, but Docker setup and core APIs are required

---

How to Submit

Your submission should be a public Git repository containing a ready-to-use docker-compose.yml and a ready-to-use .env.sample file. You must also reply to the email you received with the Git link before the time limit.

Project Requirements

Part A: MANDATORY Requirements (Must Complete)

1. Docker Infrastructure

Your entire project MUST be containerized and run with a single command:

```
docker-compose up
```

Required Services:

- PostgreSQL 15+ database
- Redis 7+ for caching and Celery broker
- Django application server
- Celery worker for background tasks
- Celery beat for scheduled tasks

Docker Requirements:

- Create your own `docker-compose.yml` from scratch
- Multi-stage Dockerfiles for optimized images
- Environment variables via `.env` file
- Health checks for all services
- Proper service dependencies and startup order
- Volume persistence for database data
- Automatic database migrations on startup
- Network configuration for inter-service communication
- Proper logging configuration

---

2. Django REST API

Authentication Endpoints:

- `POST /api/auth/register/`
- `POST /api/auth/login/`
- `POST /api/auth/logout/`
- `POST /api/auth/refresh/`

User Management:

- `GET /api/users/` (list with pagination)
- `GET /api/users/{id}/`
- `PUT /api/users/{id}/`
- `GET /api/users/me/`

Task Management:

- `GET /api/tasks/` (with filtering, search, pagination)
- `POST /api/tasks/`
- `GET /api/tasks/{id}/`
- `PUT /api/tasks/{id}/`
- `PATCH /api/tasks/{id}/`
- `DELETE /api/tasks/{id}/`

Task Operations:

- `POST /api/tasks/{id}/assign/`
- `POST /api/tasks/{id}/comments/`
- `GET /api/tasks/{id}/comments/`
- `GET /api/tasks/{id}/history/`

Required Task Model:

```
class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    status = models.CharField(choices=STATUS_CHOICES)
    priority = models.CharField(choices=PRIORITY_CHOICES)
    due_date = models.DateTimeField()
    estimated_hours = models.DecimalField()
    actual_hours = models.DecimalField(null=True)

    # Relationships
    created_by = models.ForeignKey(User)
    assigned_to = models.ManyToManyField(User)
    tags = models.ManyToManyField(Tag)
    parent_task = models.ForeignKey('self', null=True)

    # Metadata
    metadata = models.JSONField(default=dict)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    is_archived = models.BooleanField(default=False)
```

---

3. PostgreSQL Database with Django ORM

Required Models:

- User (extend Django AbstractUser)
- Task
- Comment
- Tag
- TaskAssignment (through model)
- TaskHistory (audit log)
- Team
- TaskTemplate

ORM Requirements:

- Use Django ORM exclusively (no raw SQL unless justified)
- Implement custom model managers
- Use `select_related()` and `prefetch_related()`
- Database indexes on frequently queried fields
- Soft delete implementation
- Model validation and signals

PostgreSQL Features:

- `JSONField` for metadata
- Full-text search on task descriptions
- Database constraints
- Proper migrations

---

4. Celery Background Tasks

Required Celery Tasks:

```
@shared_task
def send_task_notification(task_id, notification_type):
    """Send email notifications for task events"""
    pass

@shared_task
def generate_daily_summary():
    """Generate daily task summary for all users"""
    pass

@shared_task
def check_overdue_tasks():
    """Mark tasks as overdue and notify assignees"""
    pass

@shared_task
def cleanup_archived_tasks():
    """Delete archived tasks older than 30 days"""
    pass
```

Celery Beat Schedule:

- Daily summary
- Hourly overdue check
- Weekly cleanup

---

5. Frontend Application

Your project **MUST** include a simple frontend to showcase the core API functionality. This demonstrates your ability to build a full-stack application and interact with a backend service using a server-side rendering approach.

Objective: Build a basic UI using **Django's templating engine**. We are not looking for a complex, production-ready UI, but a working demonstration of how you would render HTML pages from the server and handle user interaction.

Required Functionality:

- **Authentication:** Create Django views and templates for user login and logout. After logging in, the user should be redirected to the task list.
- **Task List:** A page that displays a list of tasks rendered with Django templates.
- **Task Management:** Simple forms to create a new task and view the details of an existing task.

Implementation Notes:

- The frontend will be served directly by the Django application server. No separate frontend service is required in the `docker-compose.yml`.
- You may use a minimal amount of vanilla JavaScript if needed for client-side interactions, but the primary rendering must be server-side.

Part B: Extended Features (Complete as Many as Possible)

6. Business Logic & Automation

Task Workflow Engine:

- Status transition validation
- Automatic task assignment based on rules
- Task templates with variable substitution
- Recurring task generation
- SLA tracking and escalation

Smart Features:

- Workload balancing algorithm
- Priority calculation based on multiple factors
- Dependency management (blocking/blocked tasks)
- Critical path identification
- Business hours calculation

Automation Rules:

```
# Example rules to implement
- Auto-assign based on user availability
- Escalate high-priority overdue tasks
- Send reminders before due date
- Update parent task when subtasks complete
- Calculate team velocity metrics
```

---

7. Kafka Event Streaming

Add Kafka and Zookeeper to your Docker Compose setup.

Event Producers:

- Task lifecycle events (created, updated, assigned, completed)
- User activity tracking
- System alerts and notifications
- Audit trail events

Event Consumers:

- Real-time notification processor
- Activity feed generator
- Analytics data aggregator
- Search index updater
- Audit log writer

Topics to Implement:

- `task-events`
- `user-activities`
- `system-notifications`
- `analytics-events`

---

8. Flask Analytics Microservice

Create a separate Flask service for analytics and reporting:

Endpoints:

```
# Analytics
GET /api/v1/analytics/dashboard
GET /api/v1/analytics/user/{user_id}/stats
GET /api/v1/analytics/team/{team_id}/performance
GET /api/v1/analytics/tasks/distribution

# Reports
POST /api/v1/reports/generate
GET /api/v1/reports/{report_id}
GET /api/v1/reports/{report_id}/download

# Export/Import
POST /api/v1/export/tasks
POST /api/v1/import/tasks
GET /api/v1/import/{job_id}/status
```

Requirements:

- Separate Docker container
- Connect to same PostgreSQL
- Independent Redis cache
- RESTful API design
- Async report generation

---

9. Advanced Features (Choose Based on Interest)

Full-Text Search:

- PostgreSQL full-text search implementation
- Search across tasks, comments, and tags
- Search suggestions and autocomplete
- Search filters and facets

Notification System:

- Email notifications (mock/console output)
- Webhook notifications
- Notification preferences per user
- Notification templates
- Delivery tracking and retry logic

Performance Optimizations:

- Redis caching layer
- Database query optimization
- API response caching
- Connection pooling
- Bulk operations

Security Features:

- API rate limiting per user
- JWT authentication with refresh tokens
- Role-based access control (RBAC)
- API key management
- Audit logging

Time Tracking:

- Task time logs
- Automatic tracking based on status changes
- Timesheet generation
- Overtime calculations

Deliverables

1. GitHub Repository Structure

```
task-management-system/
├── docker-compose.yml          # MANDATORY - you create this
├── docker-compose.prod.yml     # Production config (optional)
├── .env.sample                 # MANDATORY - all env variables
├── .gitignore
├── README.md                   # MANDATORY
│
├── django_backend/
│   ├── Dockerfile              # MANDATORY - you create this
│   ├── requirements.txt
│   ├── manage.py
│   ├── config/
│   │   ├── settings.py
│   │   ├── urls.py
│   │   ├── wsgi.py
│   │   └── celery.py
│   ├── apps/
│   │   ├── tasks/
│   │   │   └── templates/        # For task-related templates
│   │   ├── users/
│   │   │   └── templates/        # For auth-related templates
│   │   └── common/
│   └── scripts/
│       └── entrypoint.sh       # Docker entrypoint
│
├── flask_analytics/            # Optional
│   ├── Dockerfile
│   ├── requirements.txt
│   └── app.py
│
├── kafka_consumers/            # Optional
│   └── consumers.py
│
└── docs/
    ├── API_DOCUMENTATION.md
    ├── ARCHITECTURE.md
    └── DECISIONS.md            # MANDATORY
```

2. Required Documentation

README.md must include:

~~~
 # Task Management System

## Quick Start
```bash
git clone <repo>
cd task-management-system
cp .env.sample .env
docker-compose up
~~~

API Documentation

`API_DOC.md` file explaining **VERY BASICALLY** how to your API. Some tools may help you to generate it.

Architecture

Brief description of architecture

```
**DECISIONS.md must include:**
- Features completed and why
- Features skipped and why
- Time allocation breakdown
- Technical challenges faced
- Trade-offs made
- What you would add with more time
- Justification for using Django templates for the frontend
```

3. Testing Requirements

- Unit tests for core models
- API endpoint tests
- Integration test
- Tests must run in Docker

They are optional, but appreciated.

Evaluation Criteria

Technical Skills

- Code quality and organization
- Django/DRF best practices
- Database design
- Error handling
- Docker implementation
- Frontend implementation (using Django templates)

Technology Integration

- Successful Docker orchestration
- Celery/Redis integration
- PostgreSQL usage
- Proper template rendering and form handling

Problem Solving

- Feature prioritization
- Technical decision making
- Challenge resolution
- Architecture design

Documentation

- Code clarity
- Documentation completeness
- Setup instructions
- Git history

Development Timeline Suggestion

Day 1: Docker & Foundation

- Set up Docker infrastructure from scratch
- Create `docker-compose.yml` with all required services
- Django project structure, models, migrations
- Verify all containers communicate properly

Day 2: Core API & Frontend

- Complete authentication system
- Implement all CRUD endpoints
- **Create Django views and templates for login/logout and task list pages**
- Add basic tests

Day 3: Background Tasks & Business Logic

- Implement Celery tasks
- Add Celery Beat scheduled jobs
- **Expand the frontend with forms to create and view tasks**
- Add filtering, pagination, search to the API

Day 4: Advanced Features

- Add Kafka OR Flask microservice
- Implement additional features
- Performance optimizations

Day 5: Polish & Documentation

- Complete all documentation
- Fix bugs
- Ensure Docker runs perfectly
- Final testing

Submission Requirements

▯ Mandatory Checklist

Your submission is **INCOMPLETE** if any of these fail:

- ☐ Project runs with single `docker-compose up` command
- ☐ PostgreSQL database is used (not SQLite)
- ☐ Most or all core API endpoints work
- ☐ A basic frontend UI using Django templates is functional
- ☐ At least 2 Celery tasks are functional
- ☐ `.env.sample` file includes all required variables
- ☐ `README.md` has clear setup instructions
- ☐ `DECISIONS.md` explains your choices

How to Submit

Your submission should be a **public Git repository** containing a **ready-to-use** `docker-compose.yml` and a **ready-to-use** `.env.sample` file.

1. **Test your Docker setup:**

   ```
    # In a clean directory
   git clone <your-repo>
   cd <your-repo>
   cp .env.sample .env
   docker-compose up
   # Verify everything starts without errors
   ```

2. **Verify core features work:**
   - Test authentication endpoints
   - Verify the frontend can log in and display a list of tasks
   - Create, read, update, delete tasks
   - Check Celery tasks are processing
   - Verify database persistence

3. **Send submission email with:**
   - GitHub repository link
   - List of completed features
   - Any special setup instructions

FAQ

Q: Is Docker really mandatory? A: Yes, absolutely. The project must run with `docker-compose up` or it will not be evaluated.

Q: Should I create `docker-compose.yml` from scratch? A: Yes, you must create your own Docker configuration.

Q: Can I develop without Docker locally? A: Yes, but the final submission must work in Docker.

Q: What if I can't finish all features? A: Focus on Part A (mandatory) first, then add Part B features. Quality over quantity.

Q: Can I use additional packages? A: Yes, add them to `requirements.txt` and document why in `DECISIONS.md`.

Q: Should I include sample data? A: Yes, include a seed script that runs automatically on Docker startup.

Q: What Python version should I use? A: Python 3.10+ in your Docker container.

Q: Can I use Docker Compose v2? A: Yes, either v2 or v3 syntax is acceptable.

Tips for Success

1. **Start with Docker** - Create your `docker-compose.yml` on Day 1
2. **Commit frequently** - Show your progress through Git history
3. **Focus on working features** - Better to have 3 features working perfectly than 10 broken ones
4. **Document as you go** - Keep notes for your `DECISIONS.md`
5. **Test in Docker regularly** - Don't wait until the end to test Docker
6. **Use Docker logs** - Implement proper logging to help debugging
7. **Handle environment variables properly** - Use `.env` files correctly

---

Remember: We're evaluating your ability to build production-ready, containerized applications. The Docker setup, core backend features, and the basic frontend demonstration are mandatory.

Good luck!