**Technical Assignment Requirements**

|  |  |  |
| --- | --- | --- |
| **Sl.no** | **Requirements** | **Solutions** |
| 11 | The code should handle invalid inputs | * + Handled invalid inputs by returning appropriate error messages and status codes.   + method in TimerView uses TimerSerializer to validate and save the timer data. This ensures that only valid inputs are processed: |
| 2 | The firing of the webhook must not be cancelled by process restarts. Any timers that expired while the application was down should be triggered once the application comes back up. | * It is working as expected , using celery, redis, celerybeat SCHEDULE tasks * Using function like check\_expired\_timers, it is handled appropriately. |
| 3 | The solution should support horizontal scalability (running on multiple servers) to handle an increasing number of timers, including their creation and webhook firing. | Celery supports running on multiple servers, handling an increasing number of tasks. Ensure that your Celery workers are configured properly for distributed execution. It also uses a message broker like Redis for scalable message handling. |
| 4 | Eacht timer must be fired only once. | Yes we ensure this is handled in coding logic using is\_fired bool variable |
| 5 | The solution must be implemented in Python. Here at Sendcloud we use Django+(Django REST Framework or Django Ninja), FastAPI and Flask. | Yes it is implemented in Django+(Django REST Framework) |
| 6 | Wrap your application and all its dependencies in Docker container(s). Describe clearly the build steps and how to run your application in a README file. The solution should be easy to run and build for our reviewers | Application and dependencies are wrapped in Docker containers. Include a Dockerfile and a docker-compose.yml  Yes it is wrapped in Docker container. Build steps described in README file |
| 7 | Tests are mandatory. We believe in sensible testing rather than achieving 100% coverage, so make sure the important parts of your solution are tested. | Yes unit tests are included  Use following commands to run:  docker-compose run tests  docker-compose exec web python manage.py test  docker-compose exec web coverage report  docker-compose exec web coverage html |
| 8 | Your code should be reasonably documented, save for the blocks that are self-explanatory. Usage of docstrings is highly encouraged. | Yes documented with Docstrings, with explanation |
| 9 | Submit your code through the link of the e-mail that we’ve sent (with this doc) | Yes submitted |
| 10 | Assumptions made :  Anyassumptions made. For example, “timers are never scheduled later than X days into the future”. | * “time\_left” is displayed in integer seconds for clarity and simplicity. This way, the users of your API can easily understand and work with the value without worrying about fractional seconds. * Used Postman to test end points * Used sites to POST request url webhook triggering: <https://webhook.site/c91aafb2-0e75-4cc6-bd1f-bc3888b1f629> * Used Docker Desktop |
| 11 | Any changes you would make (if any) in order to support a high-traffic (e.g. 100 timer creation requests per second) production environment. | * + **Service Replicas**: Deploy multiple replicas of your services to handle increased traffic.   + Scaling your Celery workers and message broker. Using autoscaling and load balancing to handle increased traffic efficiently.   + configure service replicas and potentially use Docker Swarm or Kubernetes for orchestrating multiple containers   + **Health Checks**: Add health checks to ensure services are running correctly.   + **Service Replicas**: Use Docker Swarm or Kubernetes for managing service replicas.  ****Asynchronous Processing****  * **Celery**: Use Celery for handling time-consuming tasks asynchronously. Ensure Celery workers can scale horizontally. * **Task Queues**: Use task queues (e.g., RabbitMQ, Redis) to manage and distribute tasks efficiently  ****Database Optimization****  * **Connection Pooling**: Use connection pooling to manage database connections efficiently. * **Indexes**: Ensure proper indexing on frequently queried columns. * **Read Replicas**: Use read replicas to distribute read queries and reduce load on the primary database.  ****Rate Limiting and Throttling****  * **Rate Limiting**: Implement rate limiting to prevent abuse and ensure fair usage. Use Django’s built-in throttling or third-party libraries like django-ratelimit. * **API Gateway**: Use an API gateway (e.g., Kong, AWS API Gateway) to manage traffic and enforce rate limits.  ****Load Balancing****  * **Load Balancer**: Use a load balancer (e.g., AWS ELB, Google Cloud Load Balancer, Nginx) to distribute incoming traffic across multiple server instances. * **Auto-Scaling**: Enable auto-scaling to automatically add or remove instances based on traffic load.  . ****Caching****  * **Database Query Caching**: Use caching mechanisms like Redis or Memcached to cache frequently accessed data and reduce database load. * **HTTP Response Caching**: Cache API responses where applicable to reduce repeated processing.   **Horizontal Scaling**   * **Container Orchestration**: Use container orchestration platforms like Kubernetes or Docker Swarm to manage and scale containers.   **Service Replicas**: Deploy multiple replicas of your services to handle increased traffic. . ****Monitoring and Logging****  * **Monitoring**: Implement monitoring tools (e.g., Prometheus, Grafana) to track performance metrics and detect issues. * **Logging**: Use centralized logging (e.g., ELK stack) to aggregate and analyze logs for troubleshooting |
| 12 | Please write clean & tidy code. Within our backend team, we enforce strict PEP8 compliance. | Yes followed PEP8 , used ruff, black, isort formatting |