Database / Storage Strategy:

- Used an in-memory Map for simplicity during tech test. In production environment, this structure can be replaced with MongoDB if data persistence is needed.
- In memory with nested map structure but resets upon service restart (non-persistent).
- Using Map avoids need for full database setup.

Structure: Nested Map structure.
- Outer Map represents groups, where each group name is a key.
- Each group's value is an inner Map that stores instances by their id.

Instance Data: Each instance's data will include:
- createdAt: Timestamp when the instance was first registered.
- updatedAt: Timestamp of the last heartbeat.
- meta: Optional metadata sent by the client.

```
{
  "groupA": {
    "instance1": {
      "updatedAt": 1620000000,
      "meta": {
        "foo": "bar"
      }
    },
    "instance2": {
      "updatedAt": 1620000050,
      "meta": {
        "foo": "baz"
      }
    }
  },
  "groupB": {
    "instance3": {
      "updatedAt": 1620000100,
      "meta": {
        "foo": "qux"
      }
    }
  }
}
```

Reflection:

Use of async/await:

- The repo layer uses async/await to simulate interactions with a real database like MongoDB. This allows easy swapping of the Map with a real database without significant refactoring; the service is already structured to handle async operations.
- This approach was intended to aid flexibility, making it easier to add or replace the underlying storage system with minimal impact on other parts of the codebase.

Discovery Service Overview:

- Tracks active instances of different client applications.
- Handles periodic heartbeats to maintain up-to-date instance status.
- Removes expired instances that stop sending heartbeats.

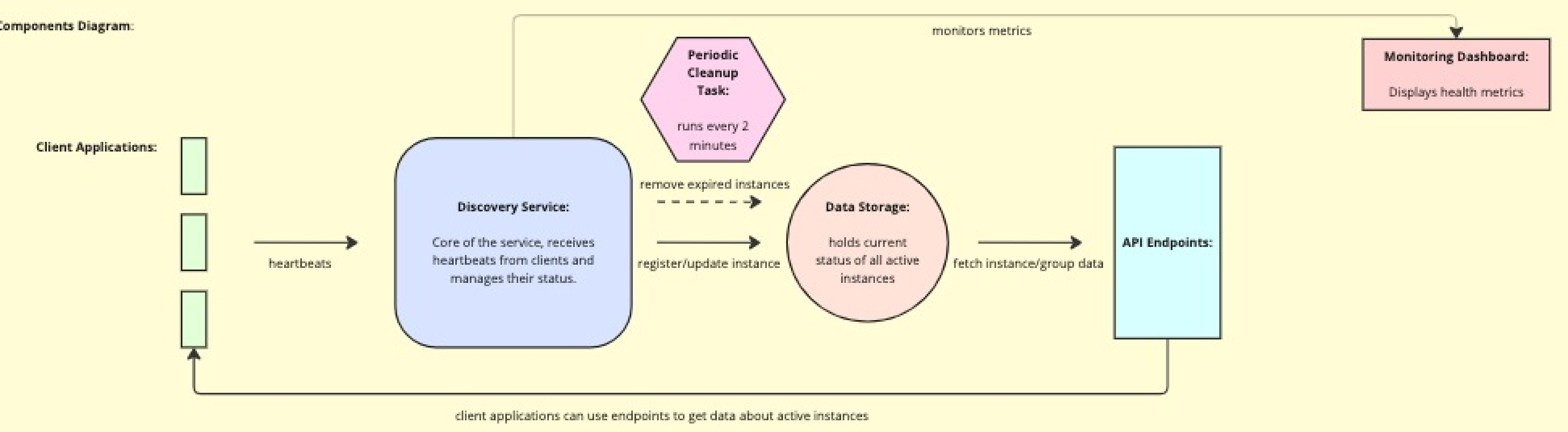
Requirements:

Handle client heartbeats.
Register and update instances.
Remove instances that have not sent heartbeats within timeframe.
Provide endpoints for instance and group summaries.

Implementation Steps Summary:

Set up project and dependencies.
Implement service with express and typescript.
Code endpoints.
Add heartbeat expiry logic.

Components Diagram:



Endpoint Flow:

POST /:group/:id
Registers or updates an application instance in the specified group.

Check Existence: Service checks if instance already exists.
Update or Register:
- If instance exists, update updatedAt.
- If doesn't exist, create new entry with createdAt, updatedAt and meta from request body.

Example Request
POST /particle-detector/e335175a-eace-4a74-b99c-c6466b6afadd

```
{
 "meta": {
     "foo": 1,
     "bar": "some additional info"
  }
}
```

Example Response

```
{
  "id": "e335175a-eace-4a74-b99c-c6466b6afadd",
  "group": "particle-detector",
  "createdAt": 1571418096158,
  "updatedAt": 1571418124127,
  "meta": {
     "foo": 1,
     "bar": "some additional info"
  }
}
```

DELETE /:group/:id
Unregisters an application instance.

Remove Instance: Service checks if instance exists, if so removes it.

Example Request
DELETE /particle-detector/e335175a-eace-4a74-b99c-c6466b6afadd

Example Response

```
{
 "message": "Instance e335175a-eace-4a74-b99c-c6466b6afadd in group particle-detector removed successfully."
}
```

GET /
Returns a summary of all currently registered groups

Fetch Summary: Service retrieves each group's summary, including total instances and timestamps.

Example Request
GET /

Example Response

```
[
 {
     "group": "particle-detector",
     "instances": 4,
     "createdAt": 1571418096158,
     "lastUpdatedAt": 1571418124127
  },
  {
     "group": "data-analyzer",
     "instances": 2,
     "createdAt": 1571419100000,
     "lastUpdatedAt": 1571419200000
  }
]
```

GET /:group
Returns details of all instances within a specified group.

Fetch Instances: Service retrieves all instances within specified group.

Example Request
GET /particle-detector

Example Response

```
[
 {
    "id": "e335175a-eace-4a74-b99c-c6466b6afadd",
    "group": "particle-detector",
    "createdAt": 1571418096158,
    "updatedAt": 1571418124127,
    "meta": {
      "foo": 1,
      "bar": "some additional info"
    }
  },
  {
    "id": "b993175a-eace-4a74-b99c-c6466b6afbb1",
    "group": "particle-detector",
    "createdAt": 1571418100000,
    "updatedAt": 1571418150000,
    "meta": {
      "foo": 2,
      "bar": "another instance"
    }
  }
]
```

GET /instances

Purpose: Retrieves all instances across all groups.
Example Response:

```
[
  {
    "id": "e335175a-eace-4a74-b99c-c6466b6afadd",
    "group": "particle-detector",
    "createdAt": 1571418096158,
    "updatedAt": 1571418124127,
    "meta": {
      "foo": 1,
      "bar": "some additional info"
    }
  },
  {
    "id": "b993175a-eace-4a74-b99c-c6466b6afbb1",
    "group": "data-analyzer",
    "createdAt": 1571418100000,
    "updatedAt": 1571418150000,
    "meta": {
      "foo": 2,
      "bar": "another instance"
    }
  }
]
```

Periodic Cleanup Strategy:

<u>How should it interact?</u>

Directly with Repository:
Pros:
- Direct access to data retrieval and deletion.

Cons:
- Bypasses business logic in the service, which could cause issues if other parts of the app rely on the service for instance manipulation.
- Logic dependent on repository structure, which could make future changes more difficult if additional logic is added to instance removal.

Best for:
- When the cleanup process is purely about managing data without any need for business logic, e.g., purging records based solely on a timestamp

With the Service
Pros:
- Service-oriented architecture by keeping business logic central to the service.
- Flexibility - if more logic is added to instance removal (e.g., logging, conditional checks), it can be handled in one place.

Cons:
- Maybe abstracted, potentially unnecessary if there is no specific business logic in service to worry about.
- Handle errors carefully.

Best for:
- When there's a chance that instance cleanup might involve more than just data deletion, or if you want consistency in instance manipulation across the app.

Outcome:
- I am leaning on the side of **using the service** to keep everything modular and inline with service-oriented design, it should also allow for future flexibility of business logic changes etc.

Reflection:

This was challenging to consider and I switched between the two a few times in development, in the end, I went with setInterval rather than setTimeout due to:

- setInterval allows for the task to run at regular intervals. This should mean that expired instances are removed without the possibility of variation that can happen with setTimeout when it waits for previous task to complete.
- setInterval means no additional logic to reschedule the task, making the cleanup logic straightforward to read and maintain.

Ideally, in a real-world scenario, I would likely use something like Redis or a cloud provider / some sort of scheduled task.

Approach

My approach aims to focus on simplicity and clarity within the constraints of the technical test. The core requirements — registration, retrieval and removal of instances via RESTful endpoints — are met, while focusing on maintainable and modular code.
Given the time frame, I have used in-memory storage for simplicity but noted how to achieve horizontal scalability in a real-world scenario below:

$\downarrow$

Endpoints: Each endpoint is designed to handle common scenarios as well as edge cases (e.g., handling existing instances on re-registration).

Periodic Cleanup: A cleanup process is triggered at configurable intervals to remove expired instances; I've noted how distributed cleanup could be implemented for scaling.

Metrics: Implemented basic metrics (active instances, expired instances, heartbeat rate error rate) to track service health. These metrics can be extended to monitor additional performance indicators.

$\longrightarrow$

Scalability

For a real-world, production setup with multiple instances, these adjustments might be necessary:

- Swap in-memory storage for a distributed database like Redis or MongoDB. This should allow multiple instances to access and manage the same data. Redis can sync data across processes or instances.
- Use something like Redis locks or similar, to manage cleanup across instances.
- Run the cleanup in a separate process (e.g., using something like child_process.fork in Node.js), the server(s) can handle requests without blocking cleanup operations.
- Move data like active instance counts etc to something like Redis or Prometheus so any instance can handle requests not relying on data stored locally.
- Centralise metrics using Prometheus to aggregate from multiple instances. Could configure each instance to push metrics to Prometheus.