A logo with text on it

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Student Name:** | Olalekan Ade-Ojo | |
| **Student ID:** | X23287951 | |
| **Programme:** | HDCSDEV\_INT |
| **Module:** | Distributed Systems | |
| **Lecturer:** | Caitriona Nic Lughadha | |
| **Submission Due Date:** | [Due on 18th April, 2025 @ 23:59PM](https://moodle2024.ncirl.ie/course/section.php?id=37136) | |
| **Project Title:** | Smart Home (Connected Lighting Device, Climate control and smart Security) | |
|  |  | |

|  |  |
| --- | --- |
| **Signature:** | Olalekan |
| **Date:** | 02/04/2025 |

**Domain Description**

The Smart-Home system is designed to emulate the main functionalities of an intelligent automation system that integrates climate control, lighting, and security monitoring into a single application. Each of the three services is designed to operate independently while also communicating with the main server through gRPC. These services respond to user inputs using a centralized GUI (Graphical User Interface) and simulate actions such as but not limited to adjusting room temperature, controlling lighting brightness, and monitoring or securing the household premises (Using the smart camera feature). The system is secured using an authentication mechanism based on a reusable **Auth** message, which eliminates the need for external constant classes.

Smart Climate Service:

* Adjust and maintain indoor temperature. (Unary )
* Monitor current temperature and humidity levels. (Server Streaming)
* Bidirectional control of temperature changes in real time. (Bidirectional Streaming)

Smart Lighting Service:

* Toggle lights on or off. (Unary)
* Adjust brightness. (Unary)
* Monitor light status. (Server Streaming)
* Batch process multiple light commands. (Client Streaming)

Smart Security Service:

* Lock and unlock doors. (Unary)
* Stream live camera feeds. (Server streaming)
* Monitor security events such as motion, door, and alarm triggers. (Server Streaming)

Together, these services simulate a complete smart home environment where the user can manage essential features through a single GUI.

**Service Definition and RPC (For all Services)**

The SmartHome system is organized into three main services:

1. Smart Climate Service
2. Smart Lighting Service
3. Smart Security Service

Each service is defined using gRPC proto files, where all supported RPC methods are declared. The system uses all four types of gRPC invocation patterns:

* Unary
* Server Streaming
* Client Streaming
* Bidirectional Streaming

The Unified Authentication method containing the Device ID to ensure proper secure communication and Authentication, an instance of the authentication {**Auth**} is parsed into each and every service in the smarthome project.. code is highlighted below

1. // Simulates authentication check for API key and device ID

2. private boolean authenticate(Auth auth) {

3. return "valid-api-key".equals(auth.getApiKey()) && auth.getDeviceId().startsWith("device-");

4. }

5.

//Proto Instance

1. // Used for authentication in service requests

2. message Auth {

3. string api\_key = 1; // API key used to verify client access

4. string device\_id = 2; // The unique identifier of the requesting device

5. }

6.

Below is a full breakdown of each service’s method-level definitions and associated request-response structure.

**Smart Climate Service**

1. **Set Temperature - Unary** : Sets the target temperature for a specific device. Requests float temperature and sets in desired temperature in °C. It has the Unified Authentication (Auth auth: Contains api\_key and device\_id). StatusResponse provides a success message string. This is used for direct, one-off temperature configuration.

1. @Override

2. public void setTemperature(SetTemperatureRequest request, StreamObserver<StatusResponse> responseObserver) {

3. // Check API key and device ID

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Get the device by ID

10. ClimateDevice device = devices.get(request.getAuth().getDeviceId());

11. if (device == null) {

12. // Return not found message if device doesn't exist

13. responseObserver.onNext(StatusResponse.newBuilder()

14. .setSuccess(false)

15. .setMessage("Device not found")

16. .build());

17. responseObserver.onCompleted();

18. return;

19. }

20.

21. float temperature = request.getTemperature();

22. // Validate temperature range (10–30 °C)

23. if (temperature < 10.0f || temperature > 30.0f) {

24. responseObserver.onNext(StatusResponse.newBuilder()

25. .setSuccess(false)

26. .setMessage("Temperature must be between 10.0 and 30.0 degrees")

27. .build());

28. responseObserver.onCompleted();

29. return;

30. }

31.

32. // Update the device target temperature

33. device.targetTemperature = temperature;

34.

35. // Respond with success

36. responseObserver.onNext(StatusResponse.newBuilder()

37. .setSuccess(true)

38. .setMessage("Temperature set to " + temperature + "°C")

39. .build());

40. responseObserver.onCompleted();

41. }

42.

2. **Get Temperature – Unary** : Fetches current and target temperature along with humidity. It Requests, Device Containing id, name and location. It has the Unified Authentication (Auth auth: Contains api\_key and device\_id). TemperatureResponse returns; float current\_temp, float target\_temp, float humidity. This is used by the GUI to display real-time climate data.

1. @Override

2. public void getTemperature(GetTemperatureRequest request, StreamObserver<TemperatureResponse> responseObserver) {

3. // Authenticate client

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. ClimateDevice device = devices.get(request.getAuth().getDeviceId());

10. if (device == null) {

11. responseObserver.onError(Status.NOT\_FOUND.asRuntimeException());

12. return;

13. }

14.

15. // Simulate temperature & humidity fluctuations

16. device.currentTemperature += (device.targetTemperature - device.currentTemperature) \* 0.1f;

17. device.currentTemperature += (random.nextFloat() - 0.5f) \* 0.2f;

18. device.humidity += (random.nextFloat() - 0.5f) \* 1.0f;

19.

20. // Build response and return climate data

21. responseObserver.onNext(TemperatureResponse.newBuilder()

22. .setCurrentTemp(device.currentTemperature)

23. .setTargetTemp(device.targetTemperature)

24. .setHumidity(device.humidity)

25. .build());

26. responseObserver.onCompleted();

27. }

28.

3. **Stream Climate Updates – Server Streaming** : Streams regular climate readings to the client based on a fixed time interval. It Requests Device and It has the Unified Authentication (Calling Auth auth: Contains api\_key and device\_id). It sets the time interval betweene updates. Response stream includes Climate update; float temperature, float humidity and timestamp. This is used for continuous monitoring scenarios in the GUI.

1. @Override

2. public void streamClimateUpdates(ClimateStreamRequest request, StreamObserver<ClimateUpdate> responseObserver) {

3. if (!authenticate(request.getAuth())) {

4. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

5. return;

6. }

7.

8. ClimateDevice device = devices.get(request.getAuth().getDeviceId());

9. if (device == null) {

10. responseObserver.onError(Status.NOT\_FOUND.asRuntimeException());

11. return;

12. }

13.

14. // Limit interval range to between 1 and 60 seconds

15. int interval = Math.max(1, Math.min(request.getUpdateIntervalSeconds(), 60));

16.

17. try {

18. // Stream updates continuously until thread is interrupted

19. while (!Thread.currentThread().isInterrupted()) {

20. // Simulate sensor updates

21. device.currentTemperature += (device.targetTemperature - device.currentTemperature) \* 0.1f;

22. device.currentTemperature += (random.nextFloat() - 0.5f) \* 0.2f;

23. device.humidity += (random.nextFloat() - 0.5f) \* 1.0f;

24.

25. // Build and send update

26. ClimateUpdate update = ClimateUpdate.newBuilder()

27. .setTemperature(device.currentTemperature)

28. .setHumidity(device.humidity)

29. .setTimestamp(System.currentTimeMillis())

30. .build();

31.

32. responseObserver.onNext(update);

33. Thread.sleep(interval \* 1000L);

34. }

35. } catch (InterruptedException e) {

36. // Gracefully exit on interruption

37. Thread.currentThread().interrupt();

38. } finally {

39. // Close the stream

40. responseObserver.onCompleted();

41. }

42. }

43.

4. **Adjust Temperature – Bidirectional Streaming**: This allows clients to adjust temperature dynamically using a delta while receiving response updates. It Streams Requests as TemperatureAdjustment messages: It requests the instance of the Device followed by Unified Authentication and float temperature\_delta: adjustment step (positive or negative) being able to increase the temperature by 1 degree or reduce by 1 degree. The server responds with streamed TemperatureAdjustment and messages:

* 1. float current\_temperature
  2. float target\_temperature
  3. OperationStatus status

This is used by the +/- buttons in the GUI to adjust the temperature up or down interactively.

1. @Override

2. public StreamObserver<TemperatureAdjustment> adjustTemperature(StreamObserver<TemperatureAdjustmentAck> responseObserver) {

3. return new StreamObserver<TemperatureAdjustment>() {

4.

5. @Override

6. public void onNext(TemperatureAdjustment request) {

7. // Authenticate each request

8. if (!authenticate(request.getAuth())) {

9. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

10. return;

11. }

12.

13. ClimateDevice device = devices.get(request.getAuth().getDeviceId());

14. if (device == null) {

15. responseObserver.onNext(TemperatureAdjustmentAck.newBuilder()

16. .setStatus(OperationStatus.DEVICE\_NOT\_FOUND)

17. .build());

18. return;

19. }

20.

21. // Adjust target temperature if within valid range

22. float newTarget = device.targetTemperature + request.getTemperatureDelta();

23. OperationStatus status;

24. if (newTarget >= 10.0f && newTarget <= 30.0f) {

25. device.targetTemperature = newTarget;

26. status = OperationStatus.SUCCESS;

27. } else {

28. status = OperationStatus.INVALID\_PARAMETER;

29. }

30.

31. // Acknowledge update with current readings

32. responseObserver.onNext(TemperatureAdjustmentAck.newBuilder()

33. .setCurrentTemperature(device.currentTemperature)

34. .setTargetTemperature(device.targetTemperature)

35. .setStatus(status)

36. .build());

37. }

38.

39. @Override

40. public void onError(Throwable t) {

41. System.err.println("AdjustTemperature error: " + t.getMessage());

42. }

43.

44. @Override

45. public void onCompleted() {

46. // Client finished sending adjustments

47. responseObserver.onCompleted();

48. }

49. };

50. }

51.

**Smart Lighting Service**

1. **Toggle Light – Unary**: This service enables the switch command for the smart light to go on or off. It requests the instance of the Device followed by Unified Authentication. It ensures toogle command for true to turn ON, false to turn OFF, It returns a StatusResponse message. This method is directly triggered when the user selects the “Light ON” or “Light OFF” toggle command.

1. @Override

2. public void toggleLight(ToggleRequest request, StreamObserver<StatusResponse> responseObserver) {

3. // Authenticate the client

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Retrieve device

10. LightDevice device = devices.get(request.getDevice().getId());

11. if (device == null) {

12. responseObserver.onNext(StatusResponse.newBuilder()

13. .setSuccess(false)

14. .setMessage("Device not found")

15. .build());

16. responseObserver.onCompleted();

17. return;

18. }

19.

20. // Apply toggle command

21. device.on = request.getOn();

22.

23. // Respond with success

24. responseObserver.onNext(StatusResponse.newBuilder()

25. .setSuccess(true)

26. .setMessage("Light " + (device.on ? "on" : "off"))

27. .build());

28. responseObserver.onCompleted();

29. }

30.

2. **Set Brightness – Unary**: Sets the brightness level of a smart light. It requests the instance of the Device followed by Unified Authentication, it also sets brightness value from 0 – 100. The method returns a StatusResponse indicating if the brightness was set successfully. This is used in the GUI via the slider and “Set Brightness” button.

1. @Override

2. public void setBrightness(BrightnessRequest request, StreamObserver<StatusResponse> responseObserver) {

3. // Authenticate the request

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Find the light device

10. LightDevice device = devices.get(request.getDevice().getId());

11. if (device == null) {

12. responseObserver.onNext(StatusResponse.newBuilder()

13. .setSuccess(false)

14. .setMessage("Device not found")

15. .build());

16. responseObserver.onCompleted();

17. return;

18. }

19.

20. int percent = request.getPercent();

21.

22. // Validate brightness range (0-100%)

23. if (percent < 0 || percent > 100) {

24. responseObserver.onNext(StatusResponse.newBuilder()

25. .setSuccess(false)

26. .setMessage("Brightness must be between 0 and 100")

27. .build());

28. responseObserver.onCompleted();

29. return;

30. }

31.

32. // Set brightness and turn light on if brightness > 0

33. device.brightness = percent;

34. device.on = percent > 0;

35.

36. // Respond with success message

37. responseObserver.onNext(StatusResponse.newBuilder()

38. .setSuccess(true)

39. .setMessage("Brightness set to " + percent + "%")

40. .build());

41. responseObserver.onCompleted();

42. }

43.

3. **Stream Light Status – Server Streaming**: Periodically streams the current state (on/off and brightness) of a smart light. It requests the instance of the Device (ID) followed by Unified Authentication, and update interval seconds. It Streams light status responses containing brightness and timestamp. It is useful for live dashboard-style updates (not directly wired into the current GUI).

1. @Override

2. public void streamLightStatus(LightStreamRequest request, StreamObserver<LightStatus> responseObserver) {

3. // Check auth

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. LightDevice device = devices.get(request.getDevice().getId());

10. if (device == null) {

11. responseObserver.onError(Status.NOT\_FOUND.asRuntimeException());

12. return;

13. }

14.

15. // Use a capped interval between 1 and 60 seconds

16. int interval = Math.max(1, Math.min(request.getUpdateIntervalSeconds(), 60));

17.

18. try {

19. while (!Thread.currentThread().isInterrupted()) {

20. // Build current light status

21. LightStatus status = LightStatus.newBuilder()

22. .setIsOn(device.on)

23. .setBrightness(device.brightness)

24. .setTimestamp(System.currentTimeMillis())

25. .build();

26.

27. // Send update to client

28. responseObserver.onNext(status);

29.

30. // Wait before sending next update

31. Thread.sleep(interval \* 1000L);

32. }

33. } catch (InterruptedException e) {

34. Thread.currentThread().interrupt(); // Restore interrupt flag

35. } finally {

36. responseObserver.onCompleted(); // Finish stream when interrupted

37. }

38. }

39.

4. **Batch Light Control – Client Streaming**: This service allows miltiple light commands in a batch an sending them as a stream of commands to be handled at the same time or poeriodically based on the command requested. The client sends a stream of LightControlCommand, each containing; the instance of the Device (ID) followed by Unified Authentication, and one of command (either to toggle on or off or set Brightness). The server returns a single final StatusResponse after processing all commands.

This is triggered using the GUI’s “Add to Batch” and “Send Batch” process for bulk operations.

1. @Override

2. public StreamObserver<LightControlCommand> batchLightControl(StreamObserver<StatusResponse> responseObserver) {

3. return new StreamObserver<LightControlCommand>() {

4. int successCount = 0;

5. int totalCount = 0;

6.

7. @Override

8. public void onNext(LightControlCommand request) {

9. totalCount++;

10.

11. // Authenticate each command

12. if (!authenticate(request.getAuth())) return;

13.

14. LightDevice device = devices.get(request.getDevice().getId());

15. if (device == null) return;

16.

17. // Handle toggle command

18. if (request.hasToggle()) {

19. device.on = request.getToggle();

20. successCount++;

21. }

22.

23. // Handle brightness command

24. else if (request.hasSetBrightness()) {

25. int brightness = request.getSetBrightness();

26. if (brightness >= 0 && brightness <= 100) {

27. device.brightness = brightness;

28. device.on = brightness > 0;

29. successCount++;

30. }

31. }

32. }

33.

**Smart Security Service**

1. **Lock Door – Unary**: This service locks a door for a specific door (none specified in this project, only a central door lock) in the security device. . It requests the instance of the Device (ID) followed by Unified Authentication, It returns a StatusResponse confirming the door lock action. Activated through the “Lock Door” button in the Security GUI section.

1. @Override

2. public void lockDoor(LockRequest request, StreamObserver<StatusResponse> responseObserver) {

3. // Authenticate the request

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Lookup door by device ID

10. DoorDevice door = doors.get(request.getDevice().getId());

11. if (door == null) {

12. // Respond with error if door not found

13. responseObserver.onNext(StatusResponse.newBuilder()

14. .setSuccess(false)

15. .setMessage(" Door not found")

16. .build());

17. responseObserver.onCompleted();

18. return;

19. }

20.

21. // Lock the door and respond

22. door.locked = true;

23. responseObserver.onNext(StatusResponse.newBuilder()

24. .setSuccess(true)

25. .setMessage(door.name + " locked")

26. .build());

27. responseObserver.onCompleted();

28. }

29.

2. **Unlock Door – Unary**: This service unlocks the specified smart door. It requests the instance of the Device (ID) followed by Unified Authentication, It returns a StatusResponse confirming the door unlocked action. This is triggered from the GUI “Unlock Door” control.

1. @Override

2. public void unlockDoor(LockRequest request, StreamObserver<StatusResponse> responseObserver) {

3. // Authenticate the request

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Lookup door

10. DoorDevice door = doors.get(request.getDevice().getId());

11. if (door == null) {

12. // Respond if door not found

13. responseObserver.onNext(StatusResponse.newBuilder()

14. .setSuccess(false)

15. .setMessage(" Door not found")

16. .build());

17. responseObserver.onCompleted();

18. return;

19. }

20.

21. // Unlock and acknowledge

22. door.locked = false;

23. responseObserver.onNext(StatusResponse.newBuilder()

24. .setSuccess(true)

25. .setMessage(door.name + " unlocked")

26. .build());

27. responseObserver.onCompleted();

28. }

29.

3. **Stream Camera Feed – Server Streaming**: This service streams live camera frames from a security device to the client. It requests the string camera\_id, unified authentication (Auth) and frame\_rate: frames per second. It streams/returns CameraFrame by using; bytes image\_data, repeated DetectedObject and timestamp

This is used by the camera viewing section of the security interface to simulate real-time feeds.

1. @Override

2. public void streamCameraFeed(CameraRequest request, StreamObserver<CameraFrame> responseObserver) {

3. // Validate authentication

4. if (!authenticate(request.getAuth())) {

5. responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());

6. return;

7. }

8.

9. // Validate camera ID

10. if (!cameras.containsKey(request.getCameraId())) {

11. responseObserver.onError(Status.NOT\_FOUND.asRuntimeException());

12. return;

13. }

14.

15. // Determine frame rate and interval

16. int frameRate = Math.max(1, Math.min(request.getFrameRate(), 30));

17. long frameInterval = 1000 / frameRate;

18. int frameCount = 0;

19.

20. try {

21. // Send frames indefinitely unless interrupted

22. while (!Thread.currentThread().isInterrupted()) {

23. // Simulate fake image data

24. byte[] frameData = new byte[1024];

25. random.nextBytes(frameData);

26.

27. // Build frame message

28. CameraFrame.Builder frameBuilder = CameraFrame.newBuilder()

29. .setImageData(com.google.protobuf.ByteString.copyFrom(frameData))

30. .setTimestamp(System.currentTimeMillis());

31.

32. // Occasionally simulate detecting a person in the frame

33. if (frameCount % 30 == 0) {

34. DetectedObject person = DetectedObject.newBuilder()

35. .setType("person")

36. .setConfidence(0.7f + random.nextFloat() \* 0.25f)

37. .setBoundingBox(Rect.newBuilder()

38. .setX(random.nextInt(100))

39. .setY(random.nextInt(100))

40. .setWidth(20 + random.nextInt(20))

41. .setHeight(40 + random.nextInt(20))

42. .build())

43. .build();

44. frameBuilder.addObjects(person);

45. }

46.

47. // Send frame to client

48. responseObserver.onNext(frameBuilder.build());

49. frameCount++;

50.

51. // Wait before sending next frame

52. Thread.sleep(frameInterval);

53. }

54. } catch (InterruptedException e) {

55. Thread.currentThread().interrupt(); // Restore interrupt flag

56. } finally {

57. responseObserver.onCompleted(); // Complete stream

58. }

59. }

60.

4. **Monitor Security Events – Bidirectional Streaming**: This Service receives incoming security events and returns alerts if certain conditions are met. Client streams SecurityEvent, which may contain: MotionEvent, DoorEvent, or AlarmEvent and devicce authentication.

Server responds with streamed SecurityAlert messages, which include: alert\_id, message, timestamp, and the triggering event. This feature ensures the system is intelligent responding in real-time to threats or unexpected behavior.

1. @Override

2. public StreamObserver<SecurityEvent> monitorSecurityEvents(StreamObserver<SecurityAlert> responseObserver) {

3. return new StreamObserver<SecurityEvent>() {

4.

5. @Override

6. public void onNext(SecurityEvent event) {

7. // Skip if unauthenticated

8. if (!authenticate(event.getAuth())) {

9. return;

10. }

11.

12. SecurityAlert alert = null;

13.

14. // Handle motion detection events

15. if (event.hasMotion()) {

16. MotionEvent motion = event.getMotion();

17. if (motion.getConfidence() > 0.8f) {

18. alert = SecurityAlert.newBuilder()

19. .setAlertId("motion-" + System.currentTimeMillis())

20. .setMessage("High confidence motion detected by camera " + motion.getCameraId())

21. .setTimestamp(System.currentTimeMillis())

22. .setEvent(event)

23. .build();

24. }

25. }

26.

27. // Handle door events (opened while locked)

28. else if (event.hasDoor()) {

29. DoorEvent doorEvent = event.getDoor();

30. DoorDevice door = doors.get(doorEvent.getDoorId());

31. if (door != null && doorEvent.getOpened() && door.locked) {

32. alert = SecurityAlert.newBuilder()

33. .setAlertId("door-" + System.currentTimeMillis())

34. .setMessage("Door " + door.name + " opened while locked!")

35. .setTimestamp(System.currentTimeMillis())

36. .setEvent(event)

37. .build();

38. }

39. }

40.

41. // Handle alarm triggered events

42. else if (event.hasAlarm()) {

43. AlarmEvent alarmEvent = event.getAlarm();

44. if (alarmEvent.getTriggered()) {

45. String alarmType = alarmEvent.getType().name().toLowerCase().replace("\_", " ");

46. alert = SecurityAlert.newBuilder()

47. .setAlertId("alarm-" + System.currentTimeMillis())

48. .setMessage(alarmType.substring(0, 1).toUpperCase() + alarmType.substring(1) + " alarm triggered!")

49. .setTimestamp(System.currentTimeMillis())

50. .setEvent(event)

51. .build();

52. alarmStatus = true;

53. }

54. }

55.

56. // Send alert if generated

57. if (alert != null) {

58. responseObserver.onNext(alert);

59. }

60. }

61.

62. @Override

63. public void onError(Throwable t) {

64. System.err.println("MonitorSecurityEvents error: " + t.getMessage());

65. }

66.

67. @Override

68. public void onCompleted() {

69. responseObserver.onCompleted(); // Notify client that the stream ended

70. }

71. };

72. }

73.

**Service Implementations**

This section details the logic behind the actual implementations of each SmartHome service: **Climate**, **Lighting**, and **Security**. These implementations are responsible for receiving incoming gRPC requests, executing the desired logic (e.g., updating a device state, streaming data, or aggregating commands), and responding appropriately. They are defined by overriding the methods exposed in the generated “ServiceImplBase classes” in other words they duplicate the already generated methods allowing proper customization to fit our requirements.

**1.  ClimateServiceImpl.java:** The ClimateServiceImpl class handles all temperature-related operations. Devices are stored in a ConcurrentHashMap<String, ClimateDevice> to simulate active climate sensors.

* **SetTemperature** – *Unary RPC*: Updates the targetTemperature of a given device. It performs full authentication using Auth, validates temperature bounds (10°C to 30°C), and sends back a StatusResponse. This function simulates a real-world thermostat temperature setting.
* **GetTemperature** – *Unary RPC*: Retrieves the currentTemperature, targetTemperature, and humidity. The temperature is auto-updated by simulating environmental changes and sensor delay with randomization logic. The response is constructed using the TemperatureResponse message.
* **StreamClimateUpdates –** *Server Streaming RPC***:** This functionality is responsible for continuously sending real-time climate readings to the client application. It streams temperature and humidity updates based on a set interval specified in the update\_interval\_seconds parameter. Once the request is authenticated and the corresponding device is verified, the server enters a loop, emitting updates at the requested interval. The stream runs persistently and only stops either when the thread is externally interrupted or when the client-side cancels or completes the connection.
* **AdjustTemperature –** *Bidirectional Streaming RPC* : This operation enables responsive temperature tuning by accepting a stream of Temperature +/- command messages from the client. Each adjustment contains a delta value that shifts the target temperature in this case by + or - 1. The server processes each delta immediately, applies it to the associated device (while ensuring it remains within the 10 degree C to 30 degree C safe range), and sends back an response through a corresponding TemperatureAdjustmentAck message. This setup ensures instant feedback and synchronized temperature control, suitable for the GUI panel.

**2. LightingServiceImpl.java:** The LightingServiceImpl class manages a collection of smart lighting devices, with each device maintaining two primary attributes: isOn or Off and brightness (0–100%).

* **ToggleLight – Unary RPC:** This method is responsible for toggling a light ON or OFF. It accepts a ToggleRequest containing device details and an Auth token. After validating the user and verifying the device exists, it sets the on command accordingly and responds with a StatusResponse indicating success or failure. This is triggered directly from the GUI by clicking the on or off button.
* **SetBrightness – Unary RPC:** This adjusts the brightness level of a specified light. Brightness is to be within 0 to 100 percent. Any brightness value above zero automatically indicates the device as “ON”. Input validation is enforced, and a StatusResponse is returned to indicate result in the text result area. Used in GUI when setting brightness via slider or batch command.
* **StreamLightStatus – Server Streaming RPC:** Continuously streams light state updates to the client at a user-defined interval. Each message sent contains the current ON/OFF status and brightness level. It is useful for the GUI-side feedback, ensuring the displayed light status is accurate in real-time. This method only ends when the client stops the request or the thread is interrupted.
* **BatchLightControl – Client Streaming RPC:** Enables sending multiple light control commands as a stream. Each command can be either a toggle or brightness change. All commands are processed one after the other. The requests are sent using the Send Batch command and upon receiving all requests, the server sends a final StatusResponse summarizing how many commands were processed successfully. This function is used in the Lighting tab for sending multiple commands in one go (batch execution).

**3. SecurityServiceImpl.java:** The SecurityServiceImpl class handles multiple instances of the home security infrastructure. It manages both door locks, Unlocks and surveillance cameras.Devices are organized into internal maps (doors and cameras) for easy lookup.

* **LockDoor / UnlockDoor – Unary RPC:** These methods allow direct control over door locks (Front door as indicated in the GUI). Each call includes a device ID and an Auth token. If the device is found and validated, the lock command is updated and a success message is returned. Invalid requests or unauthorized access will result in an error being sent to the client. These are triggered from the GUI’s security panel (lock/unlock buttons).
* **StreamCameraFeed – Server Streaming RPC:** This feature simulates a real world video feed from a specified camera by sending the framerates (mimicking real life image streaming). Frames are generated at a user-defined frame rate and sent to the client in a continuous stream. Each CameraFrame includes binary image data and a timestamp. It is useful for GUI live stream feedback, it also features a Stop Camera button allowing the user to stop the camera streaming at any time they wish.
* **MonitorSecurityEvents – Bidirectional Streaming RPC:** This is a live bidirectional stream between client and server. The client sends in real-time SecurityEvent messages like motion detection, door access, or alarm triggers. The server processes each event, evaluates conditions (e.g., high-confidence motion, door unlocked violation), and sends back a SecurityAlert in real-time. This is vital for real-time monitoring and alert generation within the SmartHome Security module. Although some of the functions are not defined in the GUI, the code highlighting the extra functions is highlighted below.

1. @Override

2. public void onNext(SecurityEvent event) {

3. // Skip if unauthenticated

4. if (!authenticate(event.getAuth())) {

5. return;

6. }

7.

8. SecurityAlert alert = null;

9.

10. // Handle motion detection events

11. if (event.hasMotion()) {

12. MotionEvent motion = event.getMotion();

13. if (motion.getConfidence() > 0.8f) {

14. alert = SecurityAlert.newBuilder()

15. .setAlertId("motion-" + System.currentTimeMillis())

16. .setMessage("High confidence motion detected by camera " + motion.getCameraId())

17. .setTimestamp(System.currentTimeMillis())

18. .setEvent(event)

19. .build();

20. }

21. }

22.

23. // Handle door events (opened while locked)

24. else if (event.hasDoor()) {

25. DoorEvent doorEvent = event.getDoor();

26. DoorDevice door = doors.get(doorEvent.getDoorId());

27. if (door != null && doorEvent.getOpened() && door.locked) {

28. alert = SecurityAlert.newBuilder()

29. .setAlertId("door-" + System.currentTimeMillis())

30. .setMessage("Door " + door.name + " opened while locked!")

31. .setTimestamp(System.currentTimeMillis())

32. .setEvent(event)

33. .build();

34. }

35. }

36.

37. // Handle alarm triggered events

38. else if (event.hasAlarm()) {

39. AlarmEvent alarmEvent = event.getAlarm();

40. if (alarmEvent.getTriggered()) {

41. String alarmType = alarmEvent.getType().name().toLowerCase().replace("\_", " ");

42. alert = SecurityAlert.newBuilder()

43. .setAlertId("alarm-" + System.currentTimeMillis())

44. .setMessage(alarmType.substring(0, 1).toUpperCase() + alarmType.substring(1) + " alarm triggered!")

45. .setTimestamp(System.currentTimeMillis())

46. .setEvent(event)

47. .build();

48. alarmStatus = true;

49. }

50. }

51.

**Use Of Naming Service**

In this SmartHome project, while there is no external registry used like for dynamic discovery, internal **naming conventions and structured identifiers** play a critical role in ensuring proper service interactions and device-specific communication. Each gRPC service method relies on structured naming via device IDs and component labels, forming a lightweight yet effective approach to service resolution. Here’s how it’s organized:

**1. Static Device Identification:** All devices (climate sensors, lights, security doors, and cameras) are initialized and referenced using predefined string-based device IDs such as:

* “device-thermo-1" for the climate controller
* “device-light-1” for lighting systems
* “device-door-1” or “camera-1” for security services

These identifiers are passed in requests through the Device message in the proto. Device instance highlighted below, This ensures that each request is passed directly to the correct device configuration within the backend’s in-memory device maps.

1. message Device {

2.   string id = 1;

3.   string name = 2;

4.   string location = 3;

5. }

6.

**2. Unified Service Access:** All gRPC service clients (climateStub, lightingStub, securityStub) are initialized via a single channel endpoint. This static approach guarantees that all service calls are passed to the same SmartHome server instance without dynamic resolution complexity. The channel (registered port) instance is highlighted in the code below

1. ManagedChannel channel = ManagedChannelBuilder

2.     .forAddress("localhost", 50051)

3.     .usePlaintext()

4.     .build();

5.

**3. Authentication:** To prevent access to device controls, every request includes an Auth message that carries a unique API key and device ID. This structure mimics scoped identity access, where only valid clients can issue service calls. The backend validates this during each service call. Instance of the Auth code in proto is highlighted below.

1. message Auth {

2.   string api\_key = 1;

3.   string device\_id = 2;

4. }

5.

**5. gRPC Interface Isolation:** Each of the service definitions (ClimateService, LightingService, SecurityService) is embodied in its own .proto file and package:

* Smarthome3.climate.ClimateService
* Smarthome3.lighting.LightingService
* Smarthome3.security.SecurityService

This logical separation enables a clean service lookup by namespace, ensuring the SmartHome client knows exactly what method exists in which context.

**Remote Error Handling & Advanced Features**