Sakani Project Queries Description :- >

Dashboard APIs: -

1. TotalListings API
2. topCommunityBySale API
3. topcommunityByRent API :
4. byRentGraph API:
5. bySaleGraph API

Leads APIs: -

1. create lead API
2. Get all leads API

Properties APIs: -

1. addPropertyData API
2. getPropertyData APi

Profile APIs: -

1. Add profile API:
2. Update password API:

Agents APIs: -

1. Add Agent API
2. Get All Agents API
3. Get agent by id API
4. Update Agent API
5. Delete Agent API

Settings APIs: -

1. Add SubAdmin/ super admin/ Admin API
2. Get all Sub admin/super admin/ Admin API
3. Update Sub admin/super admin/ Admin API
4. Delete Sub admin/super admin/ Admin API

Dashboard APIs: -

1. TotalListings API = >

```
const totalListings = async (previousDate, currentDate, mobile) => {
    try {
        const userData = await RQuery(
            `select * from users where contactNo = '${mobile}';`
        );
        var totalListings = []
        if (userData[0].role == 'Super Admin') {
            totalListings = await RQuery(`
        SELECT
          count(*) as totalListing
        FROM
          sakanisV2 ;
      `)
        }
        else if (userData[0].role == 'Admin' || userData[0].role == 'Sub Admin') {
            totalListings = await RQuery(`
        SELECT
          count(*) as totalListing
        FROM
          sakanisV2
        WHERE
          broker_company_name = '${userData[0].companyName}';
      `)
        } else {
            totalListings = await RQuery(`
      SELECT
        count(*) as totalListing
      FROM
        sakanisV2
      WHERE
        broker_phone = '${userData[0].contactNo}'
      `)
        }

        return totalListings[0].totalListing || 0;
```

```
    } catch (error) {
    }
};
```

Description: -

This API function retrieves the total number of property listings based on the user's role and associated data. The function takes in three parameters: `previousDate`, `currentDate`, and `mobile`.

Queries: -

The function executes different SQL queries based on the role of the user making the API request:

1. **Query for "Super Admin":**
   - This query retrieves the total number of property listings from the 'sakanisV2' table without any specific filtering.

2. **Query for "Sub Admin" or "Admin":**
   - This query retrieves the total number of property listings from the 'sakanisV2' table associated with the user's company name, filtered based on the 'broker_company_name' field.

3. **Query for other roles:**
   - This query retrieves the total number of property listings from the 'sakanisV2' table associated with the user's contact number, filtered based on the 'broker_phone' field.

2. topCommunityBySale API = >

```
const topCommunityBySale = async (previousDate, currentDate, mobile) => {

    const userData = await RQuery(
        `select * from users where contactNo = '${mobile}';`
    );
```

```
    var saleData = []
    if (userData[0].role == "Super Admin") {
        saleData = await RQuery(`
        SELECT
            community,
            short_description as building_name,
            property_type,
            leadCount,
            @rowNumber := @rowNumber + 1 AS ranking
        FROM
            (
                SELECT
                    community,
                    short_description,
                    property_type,
                    sum(X.ids) AS leadCount
                FROM
                    sakanisV2 AS sk
                    INNER JOIN (
                        SELECT
                            count(*) AS ids,
                            propertyId
                        FROM
                            leads
                            INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                        WHERE
                            sakanisV2.purpose_1 = 'buy'
                            AND (date(leads.createdAt) BETWEEN
date('${previousDate}') AND date('${currentDate}'))
                        GROUP BY
                            propertyId
                    ) X ON sk.id = X.propertyId
                WHERE
                    sk.purpose_1 = 'buy'
                GROUP BY
                    community,
                    short_description,
                    property_type
            ) AS leadCounts
            CROSS JOIN (SELECT @rowNumber := 0) AS r
```

```
        ORDER BY
            leadCount DESC;
        `)
    } else if (userData[0].role == "Sub Admin" || userData[0].role == 'Admin') {
        saleData = await RQuery(`
        SELECT
            community,
            short_description as building_name,
            property_type,
            leadCount,
            @rowNumber := @rowNumber + 1 AS ranking
        FROM
            (
                SELECT
                    community,
                    short_description,
                    property_type,
                    sum(X.ids) AS leadCount
                FROM
                    sakanisV2 AS sk
                    INNER JOIN (
                        SELECT
                            count(*) AS ids,
                            propertyId
                        FROM
                            leads
                            INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                        WHERE
                            sakanisV2.broker_company_name =
'${userData[0].companyName}'
                            AND sakanisV2.purpose_1 = 'buy'
                            AND (date(leads.createdAt) BETWEEN
date('${previousDate}') AND date('${currentDate}'))
                        GROUP BY
                            propertyId
                    ) X ON sk.id = X.propertyId
                WHERE
                    sk.broker_company_name = '${userData[0].companyName}'
                    AND sk.purpose_1 = 'buy'
                GROUP BY
```

```
                    community,
                    short_description,
                    property_type
            ) AS leadCounts
            CROSS JOIN (SELECT @rowNumber := 0) AS r
        ORDER BY
            leadCount DESC;
        `)
    } else {
        saleData = await RQuery(`
        SELECT
            community,
            short_description,
            property_type,
            leadCount,
            @rowNumber := @rowNumber + 1 AS rowNumber
        FROM
            (
                SELECT
                    community,
                    short_description,
                    property_type,
                    sum(X.ids) AS leadCount
                FROM
                    sakanisV2 AS sk
                    INNER JOIN (
                        SELECT
                            count(*) AS ids,
                            propertyId
                        FROM
                            leads
                            INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                        WHERE
                            sakanisV2.broker_phone = '${userData[0].contactNo}'
                            AND sakanisV2.purpose_1 = 'buy'
                            AND (date(leads.createdAt) BETWEEN
date('${previousDate}') AND date('${currentDate}'))
                        GROUP BY
                            propertyId
                    ) X ON sk.id = X.propertyId
```

```
                WHERE
                    sk.broker_phone = '${userData[0].contactNo}'
                    AND sk.purpose_1 = 'buy'
                GROUP BY
                    community,
                    short_description,
                    property_type
            ) AS leadCounts
            CROSS JOIN (SELECT @rowNumber := 0) AS r
        ORDER BY
            leadCount DESC;
`)
    }
    return saleData;

};
```

Description:

This API function retrieves the top communities based on lead counts for property sales within a specified date range. The function takes in three parameters: `previousDate`, `currentDate`, and `mobile`.

Query Details:

1. **Query for "Super Admin":**
   - This query retrieves the top communities for all properties where the purpose is set to 'buy' and lead counts are calculated within the specified date range.
   - It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'purpose_1' set to 'buy'.
   - The lead counts for each property are calculated using the inner query (aliased as 'X').
   - The results are grouped by community, short_description (building name), and property_type, and ranked based on lead counts in descending order.

2. **Query for "Sub Admin" or "Admin":**

- This query retrieves the top communities for properties associated with the user's company, where the purpose is set to 'buy', and lead counts are calculated within the specified date range.
- It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'broker_company_name' matching the user's company name and 'purpose_1' set to 'buy'.
- The lead counts for each property are calculated using the inner query (aliased as 'X').
- The results are grouped by community, short_description (building name), and property_type, and ranked based on lead counts in descending order.

3. **Query for other roles:**
- This query retrieves the top communities for properties associated with the user's contact number (broker_phone), where the purpose is set to 'buy', and lead counts are calculated within the specified date range.
- It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'broker_phone' matching the user's contact number and 'purpose_1' set to 'buy'.
- The lead counts for each property are calculated using the inner query (aliased as 'X').
- The results are grouped by community, short_description, and property_type, and ranked based on lead counts in descending order.

3. topcommunityByRent API : = >

```
const topCommunityByRent = async (previousDate, currentDate, mobile) => {
    const userData = await RQuery(
        `select * from users where contactNo = '${mobile}';`
    );
    var saleData = []
    if (userData[0].role == "Super Admin") {
        saleData = await RQuery(`
      SELECT
          community,
          short_description as building_name,
          property_type,
          leadCount,
          @rowNumber := @rowNumber + 1 AS ranking
      FROM
          (
              SELECT
                  community,
```

```
                short_description,
                property_type,
                sum(X.ids) AS leadCount
            FROM
                sakanisV2 AS sk
                INNER JOIN (
                    SELECT
                        count(*) AS ids,
                        propertyId
                    FROM
                        leads
                        INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                    WHERE
                        sakanisV2.purpose_1 = 'rent'
                        AND (date(leads.createdAt) BETWEEN date('${previousDate}')
AND date('${currentDate}'))
                    GROUP BY
                        propertyId
                ) X ON sk.id = X.propertyId
            WHERE
                sk.purpose_1 = 'rent'
            GROUP BY
                community,
                short_description,
                property_type
        ) AS leadCounts
        CROSS JOIN (SELECT @rowNumber := 0) AS r
    ORDER BY
        leadCount DESC;
    `)
  } else if (userData[0].role = "Sub Admin" || userData[0].role == 'Admin') {
      saleData = await RQuery(`
    SELECT
        community,
        short_description as building_name,
        property_type,
        leadCount,
        @rowNumber := @rowNumber + 1 AS ranking
    FROM
        (
```

```
            SELECT
                community,
                short_description,
                property_type,
                sum(X.ids) AS leadCount
            FROM
                sakanisV2 AS sk
                INNER JOIN (
                    SELECT
                        count(*) AS ids,
                        propertyId
                    FROM
                        leads
                        INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                    WHERE
                        sakanisV2.broker_company_name =
'${userData[0].companyName}'
                        AND sakanisV2.purpose_1 = 'rent'
                        AND (date(leads.createdAt) BETWEEN date('${previousDate}')
AND date('${currentDate}'))
                    GROUP BY
                        propertyId
                ) X ON sk.id = X.propertyId
            WHERE
                sk.broker_company_name = '${userData[0].companyName}'
                AND sk.purpose_1 = 'rent'
            GROUP BY
                community,
                short_description,
                property_type
        ) AS leadCounts
        CROSS JOIN (SELECT @rowNumber := 0) AS r
    ORDER BY
        leadCount DESC;
    `)
  } else {
      saleData = await RQuery(`
    SELECT
        community,
        short_description,
```

```
        property_type,
        leadCount,
        @rowNumber := @rowNumber + 1 AS rowNumber
    FROM
        (
            SELECT
                community,
                short_description,
                property_type,
                sum(X.ids) AS leadCount
            FROM
                sakanisV2 AS sk
                INNER JOIN (
                    SELECT
                        count(*) AS ids,
                        propertyId
                    FROM
                        leads
                        INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                    WHERE
                        sakanisV2.broker_phone = '${userData[0].contactNo}'
                        AND sakanisV2.purpose_1 = 'rent'
                        AND (date(leads.createdAt) BETWEEN date('${previousDate}')
AND date('${currentDate}'))
                    GROUP BY
                        propertyId
                ) X ON sk.id = X.propertyId
            WHERE
                sk.broker_phone = '${userData[0].contactNo}'
                AND sk.purpose_1 = 'rent'
            GROUP BY
                community,
                short_description,
                property_type
        ) AS leadCounts
        CROSS JOIN (SELECT @rowNumber := 0) AS r
    ORDER BY
        leadCount DESC;
`)
    }
```

```
    return saleData;
};
```

Description:
This API function retrieves the top communities based on lead counts for property rentals within a specified date range. The function takes in three parameters: `previousDate`, `currentDate`, and `mobile`.

Queries:
The function executes different SQL queries based on the role of the user making the API request:

1. Query for "Super Admin":
- This query retrieves the top communities for all rental properties where the purpose is set to 'rent', and lead counts are calculated within the specified date range.
- It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'purpose_1' set to 'rent'.
- The lead counts for each property are calculated using the inner query (aliased as 'X').
- The results are grouped by community, short_description (building name), and property_type, and ranked based on lead counts in descending order.

2. Query for "Sub Admin" or "Admin":
- This query retrieves the top communities for rental properties associated with the user's company name, where the purpose is set to 'rent', and lead counts are calculated within the specified date range.
- It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'broker_company_name' matching the user's company name and 'purpose_1' set to 'rent'.
- The lead counts for each property are calculated using the inner query (aliased as 'X').
- The results are grouped by community, short_description (building name), and property_type, and ranked based on lead counts in descending order.

3. Query for other roles:
- This query retrieves the top communities for rental properties associated with the user's contact number (broker_phone), where the purpose is set to 'rent', and lead counts are calculated within the specified date range.

- It joins the 'sakanisV2' and 'leads' tables based on propertyId and filters properties with 'broker_phone' matching the user's contact number and 'purpose_1' set to 'rent'.
- The lead counts for each property are calculated using the inner query (aliased as 'X').
- The results are grouped by community, short_description, and property_type, and ranked based on lead counts in descending order.

4. byRentGraph API: = >

```
const byRentGraph = async (previousDate, currentDate, mobile) => {
    const userData = await RQuery(
        `select * from users where contactNo = '${mobile}';`
    );
    var saleData = []
    var [email, chat, call, total] = [0, 0, 0, 0];
    if (userData[0].role == "Super Admin") {
        [email, chat, call, total] = await Promise.all([
            RQuery(`
        SELECT
          JSON_ARRAYAGG(count_id) AS count,
          JSON_ARRAYAGG(hour_am_pm) AS hour
          FROM (
            SELECT
              count(id) as count_id,
              hour_am_pm
            FROM
                (
                    SELECT
                        leads.id,
                        DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                    FROM
                        leads
                        INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                    WHERE
                        communication_mode = 'email'
                        AND sakanisV2.purpose_1 = 'rent'
                        AND (
                            date(leads.createdAt) BETWEEN date('${previousDate}')
                            AND date('${currentDate}')
                        )
```

```
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
`),
      RQuery(`
SELECT
  JSON_ARRAYAGG(count_id) AS count,
  JSON_ARRAYAGG(hour_am_pm) AS hour
  FROM (
    SELECT
      count(id) as count_id,
      hour_am_pm
    FROM
        (
            SELECT
                leads.id,
                DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
            FROM
                leads
                INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
            WHERE
                communication_mode = 'chat'
                AND sakanisV2.purpose_1 = 'rent'
                AND (
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
```

```
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
          RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
          hour_am_pm
      FROM
          (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  communication_mode = 'call'
                  AND sakanisV2.purpose_1 = 'rent'
                  AND (
                      date(leads.createdAt) BETWEEN date('${previousDate}')
                      AND date('${currentDate}')
                  )
              ORDER BY
                  leads.createdAt asc
          ) AS subquery
      GROUP BY
```

```
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
`),
        RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              sakanisV2.purpose_1 = 'rent'
              AND (
                  date(leads.createdAt) BETWEEN date('${previousDate}')
                  AND date('${currentDate}')
              )
          ORDER BY
              leads.createdAt asc
      ) AS subquery
  GROUP BY
      hour_am_pm
  ORDER BY
      STR_TO_DATE(
```

```
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC)agg;
      `)
        ])
    } else if (userData[0].role == "Sub Admin" || userData[0].role == 'Admin') {
        [email, chat, call, total] = await Promise.all([
            RQuery(`
        SELECT
          JSON_ARRAYAGG(count_id) AS count,
          JSON_ARRAYAGG(hour_am_pm) AS hour
          FROM (
            SELECT
              count(id) as count_id,
              hour_am_pm
            FROM
                (
                    SELECT
                        leads.id,
                        DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                    FROM
                        leads
                        INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                    WHERE
                        communication_mode = 'email'
                        AND sakanisV2.broker_company_name =
'${userData[0].companyName}'
                        AND sakanisV2.purpose_1 = 'rent'
                        AND (
                            date(leads.createdAt) BETWEEN date('${previousDate}')
                            AND date('${currentDate}')
                        )
                    ORDER BY
                        leads.createdAt asc
                ) AS subquery
            GROUP BY
```

```
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  communication_mode = 'chat'
                  AND sakanisV2.broker_company_name =
'${userData[0].companyName}'
                  AND sakanisV2.purpose_1 = 'rent'
                  AND (
                      date(leads.createdAt) BETWEEN date('${previousDate}')
                      AND date('${currentDate}')
                  )
              ORDER BY
                  leads.createdAt asc
            ) AS subquery
        GROUP BY
```

```
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
          RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
          hour_am_pm
      FROM
          (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  communication_mode = 'call'
                  AND sakanisV2.broker_company_name = '${userData[0].companyName}'
                  AND sakanisV2.purpose_1 = 'rent'
                  AND (
                      date(leads.createdAt) BETWEEN date('${previousDate}')
                      AND date('${currentDate}')
                  )
              ORDER BY
                  leads.createdAt asc
          ) AS subquery
      GROUP BY
          hour_am_pm
```

```
      ORDER BY
          STR_TO_DATE(
              CONCAT(
                  SUBSTRING(hour_am_pm, 1, 2),
                  ' ',
                  SUBSTRING(hour_am_pm, -2)
              ),
              '%h %p'
          ) ASC)agg;
    `),
          RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
          hour_am_pm
      FROM
          (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  sakanisV2.broker_company_name = '${userData[0].companyName}'
                  AND sakanisV2.purpose_1 = 'rent'
                  AND (
                      date(leads.createdAt) BETWEEN date('${previousDate}')
                      AND date('${currentDate}')
                  )
              ORDER BY
                  leads.createdAt asc
          ) AS subquery
      GROUP BY
          hour_am_pm
      ORDER BY
          STR_TO_DATE(
```

```
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
  `)
    ])
} else {
    [email, chat, call, total] = await Promise.all([
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'email'
                    AND sakanisV2.broker_phone = '${userData[0].contactNo}'
                    AND sakanisV2.purpose_1 = 'rent'
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
```

```
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'chat'
                    AND sakanisV2.broker_phone = '${userData[0].contactNo}'
                    AND sakanisV2.purpose_1 = 'rent'
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
```

```
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
        hour_am_pm
      FROM
          (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  communication_mode = 'call'
                  AND sakanisV2.broker_phone = '${userData[0].contactNo}'
                  AND sakanisV2.purpose_1 = 'rent'
                  AND (
                      date(leads.createdAt) BETWEEN date('${previousDate}')
                      AND date('${currentDate}')
                  )
              ORDER BY
                  leads.createdAt asc
          ) AS subquery
      GROUP BY
          hour_am_pm
      ORDER BY
          STR_TO_DATE(
```

```
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
    `),
        RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
          hour_am_pm
      FROM
          (
            SELECT
                leads.id,
                DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
            FROM
                leads
                INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
            WHERE
                sakanisV2.broker_phone = '${userData[0].contactNo}'
                AND sakanisV2.purpose_1 = 'rent'
                AND (
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
      GROUP BY
          hour_am_pm
      ORDER BY
          STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
```

```
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC)agg;
      `)
        ])
    }

    return {
        email: {
            count: JSON.parse(email[0].count) || [],
            hour: JSON.parse(email[0].hour) || []
        },
        chat: {
            count: JSON.parse(chat[0].count) || [],
            hour: JSON.parse(chat[0].hour) || []
        },
        call: {
            count: JSON.parse(call[0].count) || [],
            hour: JSON.parse(call[0].hour) || []
        },
        total: {
            count: JSON.parse(total[0].count) || [],
            hour: JSON.parse(total[0].hour) || []
        }
    }
};
```

Description:
This API function generates a graph data set representing lead counts for property rentals based on communication modes (email, chat, call) within a specified date range. The function takes in three parameters: `previousDate`, `currentDate`, and `mobile`.

Queries:
The function executes different SQL queries based on the role of the user making the API request:

1. Query for "Super Admin":

- This query retrieves lead counts and hours for email communication mode for all rental properties within the specified date range.
- Similar queries are executed for chat and call communication modes for rental properties.
- The results are grouped and ordered by hour_am_pm in ascending order, which represents the hour in 12-hour format (e.g., "08 AM", "05 PM").

2. Query for "Sub Admin" or "Admin":
- These queries are similar to the ones executed for "Super Admin" but include an additional condition to filter rental properties associated with the user's company name (broker_company_name).

3. Query for other roles:
- These queries are similar to the ones executed for "Super Admin" but include an additional condition to filter rental properties associated with the user's contact number (broker_phone).

5. bySaleGraph API :-

```
const bySaleGraph = async (previousDate, currentDate, mobile) => {
    const userData = await RQuery(
        `select * from users where contactNo = '${mobile}';`
    );
    var saleData = []
    var [email, chat, call, total] = [0, 0, 0, 0];
    if (userData[0].role == "Super Admin") {
        [email, chat, call, total] = await Promise.all([
            RQuery(`
        SELECT
          JSON_ARRAYAGG(count_id) AS count,
          JSON_ARRAYAGG(hour_am_pm) AS hour
          FROM (
            SELECT
              count(id) as count_id,
              hour_am_pm
            FROM
                (
                    SELECT
                        leads.id,
                        DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                    FROM
```

```
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'email'
                    AND sakanisV2.purpose_1 = 'buy'
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
```

```
                WHERE
                    communication_mode = 'chat'
                    AND sakanisV2.purpose_1 = 'buy'
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
          RQuery(`
    SELECT
    JSON_ARRAYAGG(count_id) AS count,
    JSON_ARRAYAGG(hour_am_pm) AS hour
    FROM (
      SELECT
        count(id) as count_id,
          hour_am_pm
      FROM
          (
              SELECT
                  leads.id,
                  DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
              FROM
                  leads
                  INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
              WHERE
                  communication_mode = 'call'
```

```
                AND sakanisV2.purpose_1 = 'buy'
                AND (
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
`),
      RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              sakanisV2.purpose_1 = 'buy'
              AND (
                  date(leads.createdAt) BETWEEN date('${previousDate}')
```

```
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
  `)
    ])
} else if (userData[0].role == "Sub Admin" || userData[0].role == 'Admin') {
    [email, chat, call, total] = await Promise.all([
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'email'
                    AND sakanisV2.broker_company_name =
'${userData[0].companyName}'
```

```
                    AND sakanisV2.purpose_1 = 'buy'
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'chat'
                    AND sakanisV2.broker_company_name =
'${userData[0].companyName}'
```

```
                AND sakanisV2.purpose_1 = 'buy'
                AND (
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC) agg;
`),
      RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              communication_mode = 'call'
              AND sakanisV2.broker_company_name = '${userData[0].companyName}'
              AND sakanisV2.purpose_1 = 'buy'
```

```
                AND (
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
`),
        RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              sakanisV2.broker_company_name = '${userData[0].companyName}'
              AND sakanisV2.purpose_1 = 'buy'
              AND (
                  date(leads.createdAt) BETWEEN date('${previousDate}')
```

```
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
  `)
    ])
} else {
    [email, chat, call, total] = await Promise.all([
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'email'
                    AND sakanisV2.broker_phone = '${userData[0].contactNo}'
                    AND sakanisV2.purpose_1 = 'buy'
```

```
                    AND (
                        date(leads.createdAt) BETWEEN date('${previousDate}')
                        AND date('${currentDate}')
                    )
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC) agg;
    `),
        RQuery(`
    SELECT
      JSON_ARRAYAGG(count_id) AS count,
      JSON_ARRAYAGG(hour_am_pm) AS hour
      FROM (
        SELECT
          count(id) as count_id,
          hour_am_pm
        FROM
            (
                SELECT
                    leads.id,
                    DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
                FROM
                    leads
                    INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
                WHERE
                    communication_mode = 'chat'
                    AND sakanisV2.broker_phone = '${userData[0].contactNo}'
                    AND sakanisV2.purpose_1 = 'buy'
                    AND (
```

```
                    date(leads.createdAt) BETWEEN date('${previousDate}')
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC) agg;
`),
      RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              communication_mode = 'call'
              AND sakanisV2.broker_phone = '${userData[0].contactNo}'
              AND sakanisV2.purpose_1 = 'buy'
              AND (
                  date(leads.createdAt) BETWEEN date('${previousDate}')
```

```
                    AND date('${currentDate}')
                )
            ORDER BY
                leads.createdAt asc
        ) AS subquery
    GROUP BY
        hour_am_pm
    ORDER BY
        STR_TO_DATE(
            CONCAT(
                SUBSTRING(hour_am_pm, 1, 2),
                ' ',
                SUBSTRING(hour_am_pm, -2)
            ),
            '%h %p'
        ) ASC)agg;
`),
      RQuery(`
SELECT
JSON_ARRAYAGG(count_id) AS count,
JSON_ARRAYAGG(hour_am_pm) AS hour
FROM (
  SELECT
    count(id) as count_id,
      hour_am_pm
  FROM
      (
          SELECT
              leads.id,
              DATE_FORMAT(leads.createdAt, '%h %p') AS hour_am_pm
          FROM
              leads
              INNER JOIN sakanisV2 ON propertyId = sakanisV2.id
          WHERE
              sakanisV2.broker_phone = '${userData[0].contactNo}'
              AND sakanisV2.purpose_1 = 'buy'
              AND (
                  date(leads.createdAt) BETWEEN date('${previousDate}')
                  AND date('${currentDate}')
              )
```

```
                ORDER BY
                    leads.createdAt asc
            ) AS subquery
        GROUP BY
            hour_am_pm
        ORDER BY
            STR_TO_DATE(
                CONCAT(
                    SUBSTRING(hour_am_pm, 1, 2),
                    ' ',
                    SUBSTRING(hour_am_pm, -2)
                ),
                '%h %p'
            ) ASC)agg;
      `)
        ])
    }

    return {
        email: {
            count: JSON.parse(email[0].count) || [],
            hour: JSON.parse(email[0].hour) || []
        },
        chat: {
            count: JSON.parse(chat[0].count) || [],
            hour: JSON.parse(chat[0].hour) || []
        },
        call: {
            count: JSON.parse(call[0].count) || [],
            hour: JSON.parse(call[0].hour) || []
        },
        total: {
            count: JSON.parse(total[0].count) || [],
            hour: JSON.parse(total[0].hour) || []
        }
    }
};
```

Description:

This API function generates a graph data set representing lead counts for property sales based on communication modes (email, chat, call) within a specified date range. The function takes in three parameters: `previousDate`, `currentDate`, and `mobile`.

Queries:
The function executes different SQL queries based on the role of the user making the API request:

1. Query for "Super Admin":
- This query retrieves lead counts and hours for email communication mode for all property sales within the specified date range.
- Similar queries are executed for chat and call communication modes for property sales.
- The results are grouped and ordered by hour_am_pm in ascending order, which represents the hour in 12-hour format (e.g., "08 AM", "05 PM").

2. Query for "Sub Admin" or "Admin":
- These queries are similar to the ones executed for "Super Admin" but include an additional condition to filter property sales associated with the user's company name (broker_company_name).

3. Query for other roles:
- These queries are similar to the ones executed for "Super Admin" but include an additional condition to filter property sales associated with the user's contact number (broker_phone).

Leads APIs: -

1. **create lead API:->**

```
INSERT INTO sakani.leads
      (
        communication_mode,
        community,
        createdAt,
        email,
        event_type,
        name,
        phoneNo,
        propertyId,
        purpose,
        ref_no,
        sakani_id,
        type
      ) VALUES
      (
        '${lead.communication_mode}',
        '${lead.community}',
        '${lead.createdAt}',
        '${lead.email}',
        '${lead.event_type}',
        '${lead.name}',
        '${lead.phoneNo}',
        '${lead.propertyId}',
        '${lead.purpose}',
        '${lead.ref_no}',
        '${lead.sakani_id}',
        '${lead.type}'
      )
    `)
```

Description:

This API endpoint allows users to add a new lead to the database. The API expects the lead information to be provided in the request body in JSON format. It performs basic validation on

the incoming data and, if valid, inserts the lead data into the "leads" table of the "sakani" database.

Database Insertion:
If the data is valid, the API creates a `lead` object with the provided information and a `createdAt` field with the current date and time. It then inserts this data into the "leads" table of the "sakani" database using an SQL INSERT query.

2. **Get all leads API :**

```
router.get("/all/:mobile", async (req, res) => {
  try {

    const userData = await RQuery(
      `SELECT * FROM users WHERE contactNo = '${req.params.mobile}';`
    );

    if (userData.length === 0) {
      return res.status(404).send({
        status: false,
        error: "User not found",
        path: req.path,
      });
    }
    var allProperties = "ALL PROPERTIES";
    if (userData[0].role == "Super Admin") {
      var allProperties = await RQuery(`
        SELECT
          leads.*
        FROM
          leads INNER JOIN
          sakanisV2 ON leads.propertyId = sakanisV2.id
      `);
    } else if (userData[0].role != "Sub Admin" || userData[0].role != "Admin") {
      var allProperties = await RQuery(`
        SELECT
          leads.*
        FROM
          leads INNER JOIN
```

```
        sakanisV2 ON leads.propertyId = sakanisV2.id
      WHERE
        sakanisV2.broker_company_name = '${userData[0].companyName}'
    `);
  } else {
    var allProperties = await RQuery(`
      SELECT
        leads.*
      FROM
        leads INNER JOIN
        sakanisV2 ON leads.propertyId = sakanisV2.id
      WHERE
        sakanisV2.broker_phone = '${userData[0].contactNo}'
    `);
  }
  if (
    allProperties.length == 0
  ) {
    return res.status(404).send({
      status: false,
      error: "No properties found for the user",
      path: req.path,
    });
  }

  data = allProperties

  return res.status(200).send({
    status: true,
    message: "ALL leads data fetched successfully",
    path: req.path,
    data: data,
  });
} catch (err) {
  console.error(err);
  return res.status(500).send({
    status: false,
    error: err.message,
    path: req.path,
  });
```

```
  }
});
```

Description:
This API endpoint fetches all the leads data associated with the given mobile number. The user's role is determined based on the mobile number, and depending on the role, the API retrieves different sets of leads data.

Queries Description: -

1. Query to retrieve user data:
- This query retrieves user data from the `users` table based on the provided mobile number.
- The `req.params.mobile` is used to filter the results and find the user.

2. Role-based query for Super Admin:
- If the user role is "Super Admin," this query retrieves all leads data from the `leads` table.
- It performs an INNER JOIN with the `sakanisV2` table based on the `propertyId`.
- The `leads` and `sakanisV2` tables are linked using the `INNER JOIN` clause.
- All leads data associated with any property are retrieved.

3. Role-based query for Sub Admin and Admin:
- If the user role is "Sub Admin" or "Admin," this query retrieves leads data from the `leads` table based on the company name (`broker_company_name`).
- It performs an INNER JOIN with the `sakanisV2` table based on the `propertyId`.
- The `leads` and `sakanisV2` tables are linked using the `INNER JOIN` clause.
- Leads data associated with the user's company are retrieved.

4. Role-based query for Normal Users:
- For normal users, this query retrieves leads data from the `leads` table based on the user's contact number (`broker_phone`).
- It performs an INNER JOIN with the `sakanisV2` table based on the `propertyId`.
- The `leads` and `sakanisV2` tables are linked using the `INNER JOIN` clause.
- Leads data associated with the user's contact number are retrieved.

5. Response handling:

- If no lead data is found based on the user role and associated properties or contact number, it returns a 404 status code with an error message.
- Otherwise, it returns a 200 status code with a success message and the fetched leads data.

=> The API efficiently handles different user roles and retrieves leads data accordingly.

Properties APIs: -

1. **addPropertyData API :-**

```
const addPropertyData = async (req, res) => {
  try {
    let url = "https://api.sakanihomes.com/api/v1/properties/";

    while (url !== null) {
      const response = await axios.get(url);
      url = response.data.next;

      for (const item of response.data.results) {
        const collection = await connectToMongoDB("sakani");

        let existingProperty = await collection.findOne({ id: item.id });
        if (!existingProperty) {
          let data = {
            id: item?.id,
            building_name: item?.building_name,
            description: item?.description,
            property_type: item?.property_type,
            property_images: item?.property_images,
            short_description: item?.short_description,
            size: item?.size,
            amenities: item?.amenities,
            bedrooms: item?.bedrooms,
            bathrooms: item?.bathrooms,
```

```
            furnishing: item?.furnishing,
            purpose_1: item?.purpose_1,
            purpose_2: item?.purpose_2,
            price: item?.price,
            community: item?.community,
            sub_community: item?.sub_community,
            created: item?.created,
            point: item?.point,
            broker_phone: item?.broker_phone,
            broker_company_name: item?.broker_company_name,
            xml_community: item?.xml_communities,
          };

          const query = await collection.insertOne(data);
        }
      }
    }

    return res
      .status(200)
      .send({ status: true, message: "Data populated successfully." });
  } catch (error) {
    return res.status(500).send({ status: false, message: error.message });
  }
};
```

Description: -

This API endpoint fetches property data from an external API (https://api.sakanihomes.com/api/v1/properties/) and populates it into the MongoDB database named "sakani." The API uses Axios to make GET requests to the external API and fetches paginated property data. For each property, it checks if the property with the same `id` already exists in the database. If not, it inserts the property data into the MongoDB collection.

2. **getPropertyData APi :-**

```
const getPropertyData = async (req, res) => {
  try {
```

```
    const userData = await RQuery(
      `select * from users where contactNo = '${req.params.mobile}';`
    );
    var collection = []
    if (userData[0].role == 'Super Admin') {
      collection = await RQuery(`
        SELECT * FROM sakanisV2;
      `);
    } else if (userData[0].role == 'Sub Admin' || userData[0].role == 'Admin') {
      collection = await RQuery(`
        SELECT * FROM sakanisV2 WHERE broker_company_name =
'${userData[0].companyName}';
      `);
    } else {
      collection = await RQuery(`
        SELECT * FROM sakanisV2 WHERE broker_phone = '${userData[0].contactNo}';
      `);
    }


    // const query = collection
    //   .find()
    //   .project({
    //     id: 1,
    //     referenceNo: 1,
    //     building_name: 1,
    //     short_description: 1,
    //     community: 1,
    //     sub_community: 1,
    //     purpose_1: 1,
    //     purpose_2: 1,
    //     property_type: 1,
    //     bedrooms: 1,
    //     bathrooms: 1,
    //     size: 1,
    //     price: 1,
    //   })
    //   .limit(100); // Adjust the limit based on requirements

    const data = collection;
```

```
    // const data = await query.toArray();

    const result = data.map((item) => {
      const {
        id,
        referenceNo,
        building_name,
        short_description,
        community,
        sub_community,
        purpose_1,
        purpose_2,
        property_type,
        bedrooms,
        bathrooms,
        size,
        price,
      } = item;

      return {
        property_id: id,
        property_ref_no: size,
        title: short_description,
        community: sub_community + ", " + community,
        purpose: purpose_1,
        type: property_type + ", " + purpose_2,
        bedrooms,
        bathrooms,
        price,
      };
    });

    return res.status(200).send({ status: true, data: result });
  } catch (error) {
    return res
      .status(500)
      .send({ status: false, message: "Internal Server Error" });
  }
};
```

Description: -

This API retrieves property data based on the user's role and company affiliation. The API performs different SQL queries depending on the user's role and then processes the results to create a formatted response.

Query Descriptions: -

1. The API starts by querying the `users` table to get user data based on the provided `mobile` parameter. The query retrieves all columns from the `users` table where the `contactNo` matches the given mobile number.

2. The API then checks the `role` field of the first user data object returned by the query. Based on the `role`, it proceeds with different SQL queries to retrieve property data.

3. If the user's `role` is 'Super Admin', the API queries the `sakanisV2` table to retrieve all records.

4. If the user's `role` is 'Sub Admin' or 'Admin', the API queries the `sakanisV2` table to retrieve records where the `broker_company_name` matches the `companyName` of the user.

5. If the user's `role` is neither 'Super Admin', 'Sub Admin', nor 'Admin', the API queries the `sakanisV2` table to retrieve records where the `broker_phone` matches the `contactNo` of the user.

6. After retrieving the property data, the API maps over the results and processes each property to create the final response format.

7. The response data contains property details like property ID, reference number, title, community, purpose, type, number of bedrooms, number of bathrooms, and price.

8. The API returns the formatted response with a HTTP Status Code 200 if successful or 500 if there's an internal server error.

Profile APIs: -

1. **Add profile API:**

```
const updateProfile = async (id, userName, companyName, email, contactNo) => {
  const checkExistingUser = await RQuery(`
            SELECT
                *
            FROM
                sakani.users
            WHERE

               id = ${id};
        `);
  if (checkExistingUser.length === 0) {
    return {
      flag: false,
      message: `User with id ${id} does not exist.`,
    };
  }
  const result = WQuery(`
                UPDATE
                    users
                SET
                    username= '${userName}',
                    companyName='${companyName}',
                    email='${email}',
                    contactNo='${contactNo}'
                WHERE
                    id = ${id};
            `);

  return {
    flag: true,
    message: result,
  };
};
```

Description : -

This function updates the profile of a user in the "users" table of the "sakani" database. It takes several parameters representing the new profile information and the user's ID to identify the user whose profile needs to be updated.

Queries Explanation : -

1. The function first queries the "users" table in the "sakani" database to check if a user with the specified `id` exists.

2. If the `checkExistingUser` query returns an empty result set (no user with the specified `id`), it returns an object with `flag` set to `false` and a message indicating that the user does not exist.

3. If the user exists (i.e., `checkExistingUser` is not empty), the function proceeds to update the user's profile using the `WQuery` function (which is not defined in the provided code snippet). It executes an `UPDATE` query to modify the `username`, `companyName`, `email`, and `contactNo` fields in the "users" table for the user with the specified `id`.

4. The function returns an object with `flag` set to `true` to indicate a successful update and provides the `result` (which is returned by the `WQuery` function) as the message.

2. Update password API:

```
const updatePassword = async (id, password) => {
  const checkExistingUser = await RQuery(`
            SELECT
                *
            FROM
                sakani.users
            WHERE
               id = ${id};
        `);
  if (checkExistingUser.length === 0) {
    return {
      flag: false,
```

```
      message: `User with id ${id} does not exist.`,
    };
  }
  const result = WQuery(`
                UPDATE
                    users
                SET
                    password= '${password}'
                WHERE
                    id = ${id};
            `);

  return {
    flag: true,
    message: result,
  };
};
```

Description :
This function is used to update the password of a user in the "users" table of the "sakani" database. It takes two parameters: the user's ID (to identify the user whose password needs to be updated) and the new password.

Queries Explanation; -
1. The function first queries the "users" table in the "sakani" database to check if a user with the specified `id` exists.

2. If the `checkExistingUser` query returns an empty result set (no user with the specified `id`), it returns an object with `flag` set to `false` and a message indicating that the user does not exist.

3. If the user exists (i.e., `checkExistingUser` is not empty), the function proceeds to update the user's password using the `WQuery` function (which is not defined in the provided code snippet). It executes an `UPDATE` query to modify the `password` field in the "users" table for

the user with the specified `id`. The new password provided as the parameter will replace the current password.

4. The function returns an object with `flag` set to `true` to indicate a successful password update and provides the `result` (which is returned by the `WQuery` function) as the message.

Agents APIs: -

1. **Add Agent API :-**

```
const addAgent = async (
  password,
  firstname,
  lastname,
  BRN,
  email,
  contactNo,
  whatsapp,
  bio,
  linkedin,
  instagram,
  companyName
) => {
  const checkExistingAgent = await RQuery(`
        SELECT
            *
        FROM
            sakani.users
        WHERE
            email = '${email}'
    `);
  //email validation
  if (!email)
    return {
      flag: false,
      message: "Email is required",
    };
```

```
  // if (!isValidEmail(email.trim()))
  //   return {
  //     flag: false,
  //     message: "Email is not valid. Email only can be created with @gmail.com
domain",
  //   };

  if (checkExistingAgent.length > 0) {
    return {
      flag: false,
      message: "Email Already Exists",
    };
  } else {
    const result = WQuery(`
            INSERT INTO
                users
            (
             username, password, firstname, lastname, BRN, email, companyName,
contactNo, createdAt, whatsapp, bio, linkedin, instagram, role ) VALUES
('${email.split("@")[0]
      }', '${password}', '${firstname}', '${lastname}', '${BRN}', '${email}',
'${companyName}', '${contactNo}', NOW(), '${whatsapp}', '${bio}', '${linkedin}',
'${instagram}', 'Agent')
        `);

    return {
      flag: true,
      message: result,
    };
  }
};
```

Description:

This API is used to add a new agent to the system. It takes various parameters such as the agent's personal information, contact details, and social media profiles. Before adding the agent, it checks if the provided email already exists in the system and validates the email format. If the email is valid and not already registered, it creates a new agent record in the database.

Queries Explanation:

The API uses two SQL queries to check the existing agent and insert a new agent into the database:

- **`checkExistingAgent` query:**

This query checks if an agent with the given email already exists in the system. If the email is already registered, it will prevent creating a duplicate account for the same email.

- **`result` query:**

If the email is not already registered, this query inserts a new agent record into the `users` table with the provided information. It uses the `WQuery` function to execute the query.

2. Get All Agents API :-

```
const getAllAgents = async (companyName, mobile) => {
  const userData = await RQuery(
    `select * from users where contactNo = '${mobile}';`
  );
  var agents = []
  if (userData[0].role == 'Super Admin') {
    agents = await RQuery(`
    SELECT
      id,
      firstname,
      lastname,
      concat(firstname, ' ', ifnull(lastname, '')) as fullname,
      email,
      BRN,
      whatsapp,
      bio,
      linkedin,
      instagram,
      createdAt,
      updatedAt,
      companyName,
      ifnull(A.ids, 0) as propertyCount,
      ifnull(B.ids, 0) as leadCount
    FROM
```

```
    sakani.users as us
    LEFT JOIN (
        SELECT
            count(sakanisV2.id) as ids,
            broker_company_name
        FROM
            sakanisV2
        GROUP BY
          broker_company_name
    ) A ON us.companyName = A.broker_company_name
    LEFT JOIN (
        SELECT
            count(leads.id) as ids,
            broker_company_name
        FROM
            leads
            INNER JOIN sakanisV2 ON sakanisV2.id = leads.propertyId
        GROUP BY
            sakanisV2.broker_company_name
    ) B ON us.companyName = B.broker_company_name
  WHERE
    role = 'Agency';
`);
} else {
  agents = await RQuery(`
  SELECT
    id,
    firstname,
    lastname,
    concat(firstname, ' ', ifnull(lastname, '')) as fullname,
    email,
    contactNo,
    BRN,
    whatsapp,
    bio,
    linkedin,
    instagram,
    createdAt,
    updatedAt,
    companyName,
```

```
    ifnull(A.ids,0) as propertyCount,
    ifnull(B.ids,0) as leadCount
  FROM
    sakani.users as us
    LEFT JOIN (
        SELECT
            count(sakanisV2.id) as ids,
            broker_phone
        FROM
            sakanisV2
        WHERE
            broker_company_name = '${companyName}'
        GROUP BY
            broker_phone
    ) A ON us.contactNo = A.broker_phone
    LEFT JOIN (
        SELECT
            count(leads.id) as ids,
            broker_phone
        FROM
            leads
            INNER JOIN sakanisV2 ON sakanisV2.id = leads.propertyId
        WHERE
            sakanisV2.broker_company_name = '${companyName}'
        GROUP BY
            sakanisV2.broker_phone
    ) B ON us.contactNo = B.broker_phone
  WHERE
    role = 'Agent'
    AND companyName = '${companyName}';
`);
}

// const aggData = await performAggregation("sakanis", [
//   {
//     $lookup: {
//       from: "leads",
//       localField: "id",
//       foreignField: "propertyId",
//       as: "result",
```

```
  //     },
  //   },
  //   {
  //     $addFields: {
  //       leadCount: { $size: "$result" },
  //     },
  //   },
  //   {
  //     $project: {
  //       id: 1,
  //       broker_phone: 1,
  //       leadCount: 1,
  //     },
  //   },
  //   {
  //     $group: {
  //       _id: "$broker_phone",
  //       leadCount: { $sum: "$leadCount" },
  //       propertyCount: { $sum: 1 },
  //     },
  //   },
  // ]);

  // await updateAgentsWithLeadCount(agents, aggData);

  if (agents.length === 0) {
    return {
      flag: false,
      message: "Data doesn't exist",
    };
  } else {
    return {
      flag: true,
      message: "Required data",
      data: agents,
    };
  }
};
```

Description:

This API is used to fetch a list of agents from the system based on certain criteria. The agents can be filtered by their `companyName` and `mobile` (contactNo). The API performs different queries depending on the role of the user making the request. If the user is a "Super Admin," the API retrieves all agents and their respective property and lead counts for all companies. If the user is an "Agent," the API fetches the agents belonging to their company along with their property and lead counts.

API Queries Explanation:

The API uses two different SQL queries depending on the user's role:

- **For "Super Admin":**
  - The first query fetches the agent data for all agents where the `role` is "Agency."
  - It performs LEFT JOINs with two subqueries to get the property count and lead count for each agent, based on their `broker_company_name` and `broker_phone` respectively.

- **For "Agent":**
  - The second query fetches the agent data for all agents belonging to the given `companyName`.
  - It also performs LEFT JOINs with two subqueries to get the property count and lead count for each agent, based on their `broker_company_name` and `broker_phone` respectively, but limited to the agents' company.

3. Get agent by id API :-

```
const getAgentById = async (req, res) => {
  const mobileNos = req.body.mobileNo;

  const userData = await RQuery(
    `select * from users where contactNo = '${mobileNos}';`
  );
  var [agentData, leadData, propertyData] = [0, 0, 0, 0]
  if (userData[0].role == 'Super Admin') {
    [agentData, leadData, propertyData] = await Promise.all([
      RQuery(`
        SELECT
          *
        FROM
          sakani.users
```

```
      WHERE
        contactNo = '${req.body.mobileNo}';
    `),
    RQuery(`
      SELECT
        leads.*
      FROM
        leads INNER JOIN
        sakanisV2 ON leads.propertyId = sakanisV2.id
      WHERE
        sakanisV2.broker_company_name = '${userData[0].companyName}'
    `),
    RQuery(`
    SELECT
      *
    FROM
      sakanisV2
    WHERE
      sakanisV2.broker_company_name = '${userData[0].companyName}'
    `)
  ])
} else {
  [agentData, leadData, propertyData] = await Promise.all([
    RQuery(`
      SELECT
        *
      FROM
        sakani.users
      WHERE
        contactNo = '${req.body.mobileNo}';
    `),
    RQuery(`
      SELECT
        leads.*
      FROM
        leads INNER JOIN
        sakanisV2 ON leads.propertyId = sakanisV2.id
      WHERE
        sakanisV2.broker_phone = '${req.body.mobileNo}'
    `),
```

```
      RQuery(`
      SELECT
        *
      FROM
        sakanisV2
      WHERE
        broker_phone = '${req.body.mobileNo}'
      `)
    ])
  }

  // const agentData = await RQuery(
  //   `select * from sakani.users where contactNo  = '${req.body.mobileNo}';`
  // );


  // const data = await performAggregation("sakanis", pipeline);
  if (data.length === 0) {
    return {
      flag: false,
      message: "Data doesn't exist",
    };
  } else {
    return {
      flag: true,
      message: "Required data",
      data: propertyData,
      agentData: agentData,
      leadsData: leadData
    };
  }
};
```

API Description:

This API is used to retrieve detailed information about an agent identified by their mobile number (`contactNo`). The API fetches the agent's data along with the leads and properties associated with the agent. The fetched data includes agent details, leads data, and property data.

API Queries Explanation:

The API performs different SQL queries based on the role of the user making the request. If the user is a "Super Admin," the API retrieves data for all agents, leads, and properties associated with the company to which the "Super Admin" belongs. If the user is an "Agent," the API fetches data only for the specific agent (identified by their `mobileNo`) along with leads and properties associated with that agent's company.

- **`userData` query:** This query fetches the user data (agent data) from the `users` table based on the provided `mobileNos` (contactNo).

- If the user role is "Super Admin":
  - Three queries are executed in parallel using `Promise.all`:
    - The first query fetches the agent data again, including all columns for the agent with the `mobileNos`.
    - The second query retrieves all leads associated with properties belonging to the company of the "Super Admin."
    - The third query fetches all properties associated with the company of the "Super Admin."

- If the user role is "Agent":
  - Similar to the "Super Admin," three queries are executed in parallel using `Promise.all`:
    - The first query fetches the agent data for the specific agent with the provided `mobileNos`.
    - The second query retrieves all leads associated with properties where the broker_phone matches the provided `mobileNos`.
    - The third query fetches all properties associated with the specific agent's company where the broker_phone matches the provided `mobileNos`.

4. **Update Agent API :-**

```
const updateAgent = async (
  id,
  firstname,
  lastname,
  BRN,
  email,
  contactNo,
```

```
  whatsapp,
  bio,
  linkedin,
  instagram,
  companyName
) => {
  const checkExistingAgent = await RQuery(`
          SELECT
              *
          FROM
              sakani.users
          WHERE

             id = ${id};
      `);
  if (checkExistingAgent.length === 0) {
    return {
      flag: false,
      message: `Agent with id ${id} does not exist.`,
    };
  }
  const result = WQuery(`
              UPDATE
                  users
              SET
                  firstname= '${firstname}',
                  lastname='${lastname}',
                  BRN='${BRN}',
                  email='${email}',
                  contactNo='${contactNo}',
                  whatsapp='${whatsapp}',
                  bio='${bio}',
                  linkedin='${linkedin}',
                  instagram='${instagram}',
                  companyName='${companyName}'
              WHERE
                  id = ${id};
          `);

  return {
```

```
    flag: true,
    message: result,
  };
};
```

API Description:

This API is used to update the information of an existing agent in the system. It takes various parameters such as the agent's personal information, contact details, and social media profiles. The API first checks if an agent with the provided `id` exists in the system. If the agent exists, it updates the agent's record in the database with the new information.

API Queries Explanation:

- **`checkExistingAgent` query:**

This query checks if an agent with the given `id` exists in the system. If the agent does not exist, the API returns an error with a message indicating that the agent with the provided `id` does not exist.

- **`result` query:**

If the agent exists, this query updates the agent's record in the `users` table with the provided information based on the `id`.

5. **Delete Agent API :-**

```
const deleteAgent = async (id) => {
  const checkingExistinguserId = await RQuery(`
    SELECT * FROM sakani.users WHERE id= ${id}
  `);

  if (checkingExistinguserId.length == 0) {
    return {
      flag: false,
      message: `Agent with id ${id} does not exist.`,
    };
  } else {
    const result = await WQuery(`
  DELETE FROM sakani.users WHERE id = ${id}
  `);
    return {
      flag: true,
```

```
      message: "Data deleted successfully",
      data: result,
    };
  }
};
```

API Description:

This API is used to delete an existing agent from the system based on the provided agent `id`. The API first checks if an agent with the given `id` exists in the system. If the agent exists, it deletes the agent's record from the database. If the agent does not exist, it returns an error message indicating that the agent with the provided `id` does not exist.

API Queries Explanation:

- **`checkingExistinguserId` query:**

This query checks if an agent with the provided `id` exists in the system. If the agent does not exist, the API returns an error with a message indicating that the agent with the provided `id` does not exist.

- **`result` query:**

If the agent with the provided `id` exists, this query deletes the agent's record from the `users` table based on the `id`.

Settings APIs: -

1. Add SubAdmin/ super admin/ Admin API :-

```
const addSubAdmin = async (
  password,
  firstname,
  lastname,
  email,
  contactNo,
  isAdmin,
  role,
  addRule,
  editRule,
  viewRule,
```

```
  image,
  companyName
) => {
  const checkExistingAdmin = await RQuery(`
        SELECT
            *
        FROM
            users
        WHERE email = '${email}';
    `);

  if (!email)
    return {
      flag: false,
      message: "Email is required",
    };

  // if (!isValidEmail(email.trim()))
  //   return {
  //     flag: false,
  //     message: "Email is not valid. Email only can be created with @pixl.ae
domain",
  //   };

  if (checkExistingAdmin.length > 0) {
    return {
      flag: false,
      message: " Email Already Exists",
    };
  } else {
    const result = WQuery(`
            INSERT INTO
                users
            (
                username,
                password,
                firstname,
                lastname,
                email,
                companyName,
```

```
                contactNo,
                createdAt,
                updatedAt,
                isAdmin,
                role,
                addRule,
                editRule,
                viewRule,
                image
            ) VALUES (
                '${email.split("@")[0]}',
                '${password}',
                '${firstname}',
                '${lastname}',
                '${email}',
                '${companyName}',
                '${contactNo}',
                NOW(),
                NOW(),
                '${isAdmin}',
                '${role}',
                '${addRule}',
                '${editRule}',
                '${viewRule}',
                null
            );
        `);

    return {
      flag: true,
      message: result,
    };
  }
};
```

API Description: =>

This API is used to add a new sub-admin user to the system. Sub-admins are users with administrative privileges but with restricted access compared to the super admin. The API takes various parameters such as the sub-admin's personal information, contact details, role, and access rules. It checks if the provided email already exists in the system and validates the email format. If the email is valid and not already registered, it creates a new sub-admin record in the database.

API Queries Explanation: =>

The API uses a single SQL query to check if the provided email already exists in the system and to insert a new sub-admin record into the `users` table:

- **`checkExistingAdmin` query:**

This query checks if a user with the given email already exists in the system. If the email is already registered, it prevents creating a duplicate account with the same email.

- **`result` query:**

If the email is not already registered, this query inserts a new sub-admin record into the `users` table with the provided information.

2. Get all Sub admin/super admin/ Admin API :-

```
const getAllSubAdmins = async (mobile) => {
  const userData = await RQuery(
    `select * from users where contactNo = '${mobile}';`
  );
  subAdmins = []
  if (userData[0].role == 'Super Admin') {
    subAdmins = await RQuery(`
      SELECT
        id,
        username,
        password,
        concat(firstname,' ', ifnull(lastname,'') ) as fullname,
        firstname,
        lastname,
        email,
        companyName,
        contactNo,
```

```
        createdAt,
        updatedAt,
        isAdmin,
        role,
        addRule,
        editRule,
        viewRule
      FROM
          sakani.users
      WHERE
        role in ('Admin', 'Sub Admin');
    `);
  } else {
    subAdmins = await RQuery(`
      SELECT
        id,
        username,
        password,
        concat(firstname,' ', ifnull(lastname,'') ) as fullname,
        firstname,
        lastname,
        email,
        companyName,
        contactNo,
        createdAt,
        updatedAt,
        isAdmin,
        role,
        addRule,
        editRule,
        viewRule
      FROM
          sakani.users
      WHERE
        role in ('Admin', 'Sub Admin')
        and companyName = '${userData[0].companyName}'
`);
  }

  if (subAdmins.length === 0) {
```

```
    return {
      flag: false,
      message: "Data doesn't exist",
    };
  } else {
    return {
      flag: true,
      message: "Required data",
      data: subAdmins,
    };
  }
};
```

API Description: =>

This API is used to fetch a list of sub-admin users from the system based on certain criteria. The sub-admins can be filtered based on their `contactNo` (mobile number). The API performs different queries depending on the role of the user making the request. If the user is a "Super Admin," the API retrieves all sub-admins and their details. If the user is a "Sub Admin," the API fetches sub-admins belonging to the same company along with their details.

API Queries Explanation: =>

The API uses two different SQL queries based on the role of the user:

- **For "Super Admin":**
  - The first query fetches the user data (sub-admin data) from the `users` table based on the provided `mobile`.
  - The second query retrieves all sub-admins (`role` is "Admin" or "Sub Admin").

- **For "Sub Admin":**
  - The first query fetches the user data (sub-admin data) from the `users` table based on the provided `mobile`.
  - The second query retrieves all sub-admins belonging to the same company as the user.

3. Update Sub admin/super admin/ Admin API :-

```
const updateSubAdminById = async (
  id,
  firstname,
```

```
  lastname,
  email,
  contactNo,
  isAdmin,
  role,
  addRule,
  editRule,
  viewRule,
  image,
  companyName
) => {
  const checkExistingAdmin = await RQuery(`
    SELECT * FROM sakani.users WHERE id = ${id};
  `);

  if (checkExistingAdmin.length > 0) {
    const result = await WQuery(`
      UPDATE users SET
        firstname = '${firstname}',
        lastname = '${lastname}',
        email = '${email}',
        companyName = '${companyName}',
        contactNo = '${contactNo}',
        updatedAt = NOW(),
        isAdmin = '${isAdmin}',
        role = '${role}',
        addRule = '${addRule}',
        editRule = '${editRule}',
        viewRule = '${viewRule}',
        image = null
      WHERE id = ${id};
    `);

    return {
      flag: true,
      message: result,
    };
  } else {
    return {
      flag: false,
```

```
      message: "Sub Admin ID not found",
    };
  }
};
```

API Description: =>

This API is used to update the information of an existing sub-admin user in the system. Sub-admins are users with administrative privileges but with restricted access compared to the super admin. The API takes various parameters to update the sub-admin's personal information, contact details, role, and access rules. It first checks if a sub-admin with the provided `id` exists in the system. If the sub-admin exists, it updates the sub-admin's record in the database with the new information. If the sub-admin with the provided `id` is not found, the API returns an error message indicating that the sub-admin ID is not found.

API Queries Explanation: =>

The API uses two SQL queries to perform the update:

- **`checkExistingAdmin` query:**

This query checks if a sub-admin with the given `id` exists in the system. If the sub-admin does not exist, the API returns an error with a message indicating that the sub-admin with the provided `id` is not found.

- **`result` query:**

If the sub-admin exists, this query updates the sub-admin's record in the `users` table with the provided information based on the `id`.

4. Delete Sub admin/super admin/ Admin API :-

```
const deleteSubAdminById = async (id) => {
  const checkExistingAdmin = await RQuery(`
    SELECT * FROM sakani.users WHERE id = ${id};
  `);

  if (checkExistingAdmin.length > 0) {
    const result = await WQuery(`
      DELETE FROM users WHERE id = ${id};
    `);
```

```
    return {
      flag: true,
      message: result,
    };
  } else {
    return {
      flag: false,
      message: "Sub Admin ID not found",
    };
  }
};
```

API Description: =>

This API is used to delete an existing sub-admin user from the system based on the provided sub-admin `id`. Sub-admins are users with administrative privileges but with restricted access compared to the super admin. The API first checks if a sub-admin with the given `id` exists in the system. If the sub-admin exists, it deletes the sub-admin's record from the database. If the sub-admin with the provided `id` is not found, the API returns an error message indicating that the sub-admin ID is not found.

API Queries Explanation: =>

The API uses two SQL queries to perform the deletion:

- **`checkExistingAdmin` query:**
  This query checks if a sub-admin with the given `id` exists in the system. If the sub-admin does not exist, the API returns an error with a message indicating that the sub-admin with the provided `id` is not found.

- **`result` query:**
  If the sub-admin exists, this query deletes the sub-admin's record from the `users` table based on the `id`.