Deployment Guide

This project deploys as a **single Render Web Service** (Node.js) from the **repository root**.

- In production, the server serves the built client from `client/dist`.
- MongoDB is hosted on **MongoDB Atlas**.
- Google Sheets sync is performed asynchronously via **MongoDB Atlas App Services Triggers**.

---

1) MongoDB Atlas (Database) — Create Cluster + IP Allowlist + Connection String

One-line checklist (do these in order)

1. Create a MongoDB Atlas account → create a cluster → create a database user → configure **Network Access (IP allowlist)** → copy the **SRV connection string** and set it as `MONGODB_URI`.

Detailed steps

1. **Create Atlas account**
   - Go to https://www.mongodb.com/cloud/atlas and create/login to your account.
2. **Create a cluster**
   - In Atlas: Build a Database → choose a cluster (M0 free tier is fine for a demo).
3. **Create a database user (username/password)**
   - In Atlas: Database Access → Add New Database User
   - Use a strong password.
4. **Configure allowed IPs (Network Access)**
   - In Atlas: Network Access → IP Access List → Add IP Address

   You have two options:

   - **Development (quick)**: allow your current IP.
   - **Production (Render)**: allow your Render service's outbound IP ranges.

   Render outbound IPs are documented here (and can be copied from the Render dashboard for your service):

   - https://render.com/docs/outbound-ip-addresses

   Render dashboard path to obtain your outbound IPs:

   - Service → Connect (top-right) → Outbound tab → copy the CIDR ranges.

   Important notes:

- Atlas accepts CIDR ranges; your service may use any IP in the listed ranges.
- After adding IP rules, wait a minute and retry connecting.

5. **Retrieve the Atlas connection URL**

   - In Atlas: Database → Connect → Drivers
   - Copy the **SRV** string:
     - `mongodb+srv://<USER>:<PASSWORD>@<CLUSTER_HOST>/<DB_NAME>?retryWrites=true&w=majority`

6. **Sanity-check your connection string**

   - Confirm:
     - Username/password are the database user (not Atlas login)
     - Database name matches what you want (example: `formsuite`)
     - Your IP allowlist includes Render outbound IP ranges (for production)

---

2) Google Cloud — Create Service Account Credentials (Google Sheets API)

This project requires a Google Service Account JSON credential to access the Google Sheets API.

Steps (exact workflow)

1. **Create or select a Google Cloud project**

   - Visit https://console.cloud.google.com/
   - Select an existing project or create a new one.

2. **Enable the Google Sheets API**

   - APIs & Services → Library
   - Search: “Google Sheets API”
   - Click Enable

3. **Create a Service Account**

   - IAM & Admin → Service Accounts
   - Click Create Service Account
   - Provide a name (e.g., `formsuite-sheets-sync`)

4. **Create a JSON key for the service account**

   - Open the created service account
   - Keys → Add Key → Create new key → JSON
   - Download the JSON file

5. **Share each target Google Sheet with the service account email**

   - Open the Google Sheet in the browser
   - Share → add the service account’s `client_email` (from the JSON)

- Give at least “Editor” permission

Where you will use this JSON

You will use the **same JSON** in two places:

1. Render (server runtime) as `GOOGLE_SERVICE_ACCOUNT_JSON`
2. Atlas App Services (trigger runtime) as an App Services **Value** referencing a **Secret** that contains the JSON.

Security note:

- Never commit this JSON file or paste it into source control.

---

3) MongoDB Atlas App Services — Create App + Secrets/Values + Function + Trigger

This project uses Atlas App Services Database Triggers to sync form responses to Google Sheets.

A) Create an App Services Application

1. In Atlas, go to **App Services**
2. Click **Create a New App**
3. Choose your Atlas cluster (the same one used by `MONGODB_URI`)
4. Name the app (e.g., `FormSuiteApp`)

B) Configure Secrets and Values (Environment)

You need to store your service account JSON securely.

1. In the App Services app, go to **Values**
2. Click **Create New Value**
    - Type: **Secret**
    - Name: `google_creds_secret`
    - Value: paste the full service account JSON (exact content of the downloaded JSON)
3. Save the secret.
4. Click **Create New Value**
    - Name: `GOOGLE_SERVICE_ACCOUNT_JSON`
    - Type: **Secret**
    - Select : **Link to Secret**
    - Point it to: `google_creds_secret`

    Optional but common:

    - Add values for sheet scopes, retry limits, or a log level.

C) Install Dependencies

1. In the App Services app, go to **Dependencies**
2. Click **Add Dependency**
   - Name: `crypto`
   - Version: `^1.0.1` (or latest)
3. Save

D) Create the App Services Function

1. In the App Services app, go to **Functions**
2. Click **Create New Function**
   - Name (example): `syncFormResponseToGoogleSheet`
3. Paste your function code

   Below is the function code template:

```
exports = async function (changeEvent) {
  const FIXED_HEADERS = ["ID", "NAME", "EMAIL"];

  const fullDocument = changeEvent.fullDocument;
  const operationType = changeEvent.operationType;
  const documentId = changeEvent.documentKey._id;

  const mongodb = context.services.get("<mongodb-atlas>");
  const db = mongodb.db("<your-database-name>");
  const formsCollection = db.collection("forms");
  const responsesCollection = db.collection("formresponses");

  console.log(`Operation: ${operationType}, Response ID:
${documentId}`);

  if (fullDocument.sheetSyncStatus !== "pending") {
    return { skipped: true, reason: "not_pending" };
  }

  try {
    const serviceAccountJson = context.values.get(
      "GOOGLE_SERVICE_ACCOUNT_JSON"
    );

    if (!serviceAccountJson) {
      console.error("GOOGLE_SERVICE_ACCOUNT_JSON value not found!");
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "failed",
        "Credentials not configured"
      );
      return { error: "credentials_not_found" };
    }
```

```
    let credentials;
    try {
      credentials = JSON.parse(serviceAccountJson);
    } catch (parseErr) {
      console.error("Failed to parse credentials JSON: " +
parseErr.message);
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "failed",
        "Invalid credentials JSON"
      );
      return { error: "invalid_json" };
    }

    const form = await formsCollection.findOne({ _id:
fullDocument.formId });

    if (!form) {
      console.error("Form not found for formId: " +
fullDocument.formId);
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "failed",
        "Form not found"
      );
      return { error: "form_not_found" };
    }

    if (!form.googleSheetUrl) {
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "synced",
        null
      );
      return { success: true, reason: "no_sheet_url" };
    }

    const accessToken = await getAccessToken(credentials);

    const spreadsheetId = getSheetIdFromUrl(form.googleSheetUrl);
    if (!spreadsheetId) {
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "failed",
        "Invalid sheet URL format"
      );
      return { error: "invalid_sheet_url" };
    }
```

```
    const headersResponse = await context.http.get({
      url:
`https://sheets.googleapis.com/v4/spreadsheets/${spreadsheetId}/values
/Sheet1!1:1`,
      headers: {
        Authorization: [`Bearer ${accessToken}`],
        "Content-Type": ["application/json"],
      },
    });

    if (headersResponse.statusCode !== 200) {
      const errText = headersResponse.body.text();
      console.error("Failed to get headers: " + errText);
      await updateSyncStatus(
        responsesCollection,
        documentId,
        "failed",
        "Sheet access error: " + headersResponse.statusCode
      );
      return { error: "sheet_access_error", details: errText };
    }

    const headersData = JSON.parse(headersResponse.body.text());
    let headers = headersData.values?.[0] || [];

    if (headers.length < 3) {
      headers = [...FIXED_HEADERS];
    }

    const newHeaders = [...headers];
    let headersChanged = false;

    const questionTitleToHeaderIndex = {};
    newHeaders.forEach((h, i) => {
      questionTitleToHeaderIndex[h] = i;
    });

    form.questions.forEach((q) => {
      const cleanTitle = q.title.trim();
      if (questionTitleToHeaderIndex[cleanTitle] === undefined) {
        newHeaders.push(cleanTitle);
        questionTitleToHeaderIndex[cleanTitle] = newHeaders.length -
1;
        headersChanged = true;
      }
    });

    if (headersChanged) {
      await context.http.put({
        url:
`https://sheets.googleapis.com/v4/spreadsheets/${spreadsheetId}/values
/Sheet1!1:1?valueInputOption=USER_ENTERED`,
        headers: {
          Authorization: [`Bearer ${accessToken}`],
```

```
            "Content-Type": ["application/json"],
          },
          body: JSON.stringify({ values: [newHeaders] }),
        });
      }

      const rowData = new Array(newHeaders.length).fill("");
      rowData[0] = fullDocument.userMetadata?.id || "Anonymous";
      rowData[1] = fullDocument.userMetadata?.name || "Anonymous";
      rowData[2] = fullDocument.userMetadata?.email || "Anonymous";

      let answers = fullDocument.answers;
      if (answers && typeof answers.entries === "function") {
        answers = Object.fromEntries(answers);
      }
      answers = answers || {};

      form.questions.forEach((q) => {
        const cleanTitle = q.title.trim();
        const index = questionTitleToHeaderIndex[cleanTitle];
        if (index !== undefined) {
          const ans = answers[q.id];
          let val = "";
          if (Array.isArray(ans)) {
            val = "'" + ans.join(", ");
          } else if (ans !== undefined && ans !== null) {
            val = String(ans);
          }
          rowData[index] = val;
        }
      });

      let rowNumber;

      if (fullDocument.googleSheetRowNumber) {
        const rowNum = fullDocument.googleSheetRowNumber;
        const updateResp = await context.http.put({
          url:
`https://sheets.googleapis.com/v4/spreadsheets/${spreadsheetId}/values
/Sheet1!A${rowNum}?valueInputOption=USER_ENTERED`,
          headers: {
            Authorization: [`Bearer ${accessToken}`],
            "Content-Type": ["application/json"],
          },
          body: JSON.stringify({ values: [rowData] }),
        });

        if (updateResp.statusCode !== 200) {
          throw new Error("Update failed: " + updateResp.body.text());
        }
        rowNumber = rowNum;
      } else {
        const appendResp = await context.http.post({
          url:
```

```
`https://sheets.googleapis.com/v4/spreadsheets/${spreadsheetId}/values
/Sheet1!A1:append?
valueInputOption=USER_ENTERED&insertDataOption=INSERT_ROWS`,
        headers: {
          Authorization: [`Bearer ${accessToken}`],
          "Content-Type": ["application/json"],
        },
        body: JSON.stringify({ values: [rowData] }),
      });

      if (appendResp.statusCode !== 200) {
        throw new Error("Append failed: " + appendResp.body.text());
      }

      const appendData = JSON.parse(appendResp.body.text());
      const updatedRange = appendData.updates?.updatedRange;
      if (updatedRange) {
        const match = updatedRange.match(/[A-Z]+(\d+)/);
        if (match) {
          rowNumber = parseInt(match[1], 10);
        }
      }

      if (!rowNumber) {
        throw new Error("Could not determine row number");
      }
    }

    console.log("Success! Row: " + rowNumber);

    await responsesCollection.updateOne(
      { _id: documentId },
      {
        $set: {
          googleSheetRowNumber: rowNumber,
          sheetSyncStatus: "synced",
          sheetSyncError: null,
        },
        $inc: { sheetSyncAttempts: 1 },
      }
    );

    return { success: true, rowNumber: rowNumber };
  } catch (error) {
    console.error("Error: " + (error.message || String(error)));
    await updateSyncStatus(
      responsesCollection,
      documentId,
      "failed",
      error.message || String(error)
    );
    return { error: error.message };
  }
};
```

```
async function getAccessToken(credentials) {
  const crypto = require("crypto");

  const header = { alg: "RS256", typ: "JWT" };
  const now = Math.floor(Date.now() / 1000);
  const payload = {
    iss: credentials.client_email,
    scope: "https://www.googleapis.com/auth/spreadsheets",
    aud: "https://oauth2.googleapis.com/token",
    iat: now,
    exp: now + 3600,
  };

  const base64UrlEncode = (data) => {
    return Buffer.from(JSON.stringify(data))
      .toString("base64")
      .replace(/\+/g, "-")
      .replace(/\//g, "_")
      .replace(/=+$/, "");
  };

  const headerB64 = base64UrlEncode(header);
  const payloadB64 = base64UrlEncode(payload);
  const signatureInput = `${headerB64}.${payloadB64}`;

  const sign = crypto.createSign("RSA-SHA256");
  sign.update(signatureInput);
  const signature = sign
    .sign(credentials.private_key, "base64")
    .replace(/\+/g, "-")
    .replace(/\//g, "_")
    .replace(/=+$/, "");

  const jwt = `${signatureInput}.${signature}`;

  const tokenResp = await context.http.post({
    url: "https://oauth2.googleapis.com/token",
    headers: { "Content-Type": ["application/x-www-form-urlencoded"]
},
    body: `grant_type=urn:ietf:params:oauth:grant-type:jwt-
bearer&assertion=${jwt}`,
  });

  if (tokenResp.statusCode !== 200) {
    throw new Error("Token request failed: " + tokenResp.body.text());
  }

  const tokenData = JSON.parse(tokenResp.body.text());

  if (tokenData.error) {
    throw new Error(
      "Token error: " + (tokenData.error_description ||
tokenData.error)
```

```
    );
  }

  return tokenData.access_token;
}

function getSheetIdFromUrl(url) {
  const match = url.match(/\/d\/([a-zA-Z0-9-_]+)/);
  return match ? match[1] : null;
}

async function updateSyncStatus(collection, documentId, status, error)
{
  await collection.updateOne(
    { _id: documentId },
    {
      $set: { sheetSyncStatus: status, sheetSyncError: error },
      $inc: { sheetSyncAttempts: 1 },
    }
  );
}
```

Implementation notes (to align with this repo's data model):

- Form responses are stored as model `FormResponse` (Mongo collection typically `formresponses`).
- Key fields:
  - `formId` → links to `Form` which contains `googleSheetUrl`
  - `answers` → map of questionId → value
  - `userMetadata` → contains `id`, `name`, `email` (defaults to "Anonymous")
  - `sheetSyncStatus` → `pending | synced | failed`
  - `googleSheetRowNumber` → numeric row index

E) Create the Database Trigger

1. In the App Services app, go to **Triggers**
2. Click **Add Trigger** → choose **Database Trigger**
3. Configure:
   - **Cluster**: your linked cluster
   - **Database name**: the DB you use in `MONGODB_URI` (example: `formsuite`)
   - **Collection name**: `formresponses` (typical for Mongoose model `FormResponse`)
   - **Operation types**: `Insert`, `Update`
   - **Full Document**:
     - Enable "Full Document" for update events (so the function can see the full doc)
4. Set the trigger's function to:
   - `syncResponseToGoogleSheet`

5. Save and enable the trigger

E) Verify trigger behavior

- Create a form and ensure its Google Sheet is shared with the service account
- Submit a response
- In App Services: check Logs (or function logs) and confirm `sheetSyncStatus` eventually becomes `synced`

---

4) Render — Deploy the Web Service (recommended: repo root)

This repository is a monorepo with `client/`, `server/`, and `shared/`.

Important (monorepo root directory rule):

Steps

1. Create a Render account
    - https://render.com/
2. Create a Web Service
    - Render Dashboard → New → Web Service
3. Connect GitHub and select your repo
    - Choose the branch you want to deploy (commonly `master` or `main`).
4. Root Directory
    - Set it to **server**
5. Set Build/Start commands

Use commands that work from the repository root:

- Build Command:
    - `npm run build`
- Start Command:
    - `npm start`

Notes:

- The server serves static files from `client/dist` in production, so the client must be built during the Render build step.

6. Set environment variables on Render

Add these server env vars in Render → Environment:

- `PORT` (Render usually injects this automatically, but you can omit it)
- `NODE_ENV=production`

- `MONGODB_URI=<your Atlas SRV connection string>`
- `JWT_SECRET=<secure random string>`
- `JWT_REFRESH_SECRET=<secure random string>`
- `CLIENT_URL=<only needed for local/dev; production disables CORS in this server>`
- `GOOGLE_SERVICE_ACCOUNT_JSON=<full service account JSON string>`
- `VITE_SERVICE_ACCOUNT_EMAIL=<google's service account client_email>`
- `VITE_THEME_STORAGE_KEY=<string for localStorage theme key>`
- `VITE_USER_STORAGE_KEY=<string for localStorage user key>`
- `VITE_VIEW_PREFERENCE_KEY=<string for localStorage view preference key>`

7. Ensure Atlas IP allowlist is correct (critical)

Before you call the deployment “done”, re-check Atlas Network Access:

- Atlas → Network Access → IP Access List
- Confirm you have allowlisted the outbound IP ranges shown for your Render service.

Render outbound IP documentation (use this to find/copy the IP ranges):

- https://render.com/docs/outbound-ip-addresses

If the service cannot connect to MongoDB in production, 90% of the time it’s because:

- The Atlas IP allowlist does not include the correct Render outbound IP ranges, or
- The connection string username/password are wrong.

Quick production verification checklist

- Render deploy succeeds and service is “Live”
- Visiting the Render URL loads the React UI (served from `client/dist`)
- Logging in works (cookies + CSRF token)
- Creating a form validates Google Sheet access (service account must be shared)
- Submitting a response creates a DB document and (eventually) syncs to Sheets via App Services trigger