ChatGPT

No-Code Automations Engine for Rainbow CRM

Rainbow CRM will support **trigger-action** automation rules (like Monday.com or Zapier). Each rule has a *trigger* (an event or condition) and an *action* (what to do). For example, a Monday.com template reads: **“When a status changes to something, delete the item”** – here the trigger is “status changes” and the action is “delete the item” [1]. In our system users might configure rules like “When status changes to ‘Done’, notify [User]” or “Every Monday at 9am, create a follow-up item”.

Key triggers and **actions** can include (but are not limited to):

- **Trigger types**:
- `on_field_change`: when a specific field (e.g. a *Status* column) is updated and matches a condition (e.g. new value = “Done”).
- `on_date_arrive`: when a date/time field (e.g. *Due Date*) reaches today or a specific date.
- `on_item_create`: when a new item is created on a board.
- `cron_schedule`: a user-defined schedule (using cron syntax) that fires periodically (e.g. “every Monday” or daily at 8:00).
- **Action types**:
- `notify_user`: send an in-app or push notification to a user or team.
- `set_field_value`: update a field on the item (e.g. set a status or date column).
- `create_item`: add a new item to a (possibly other) board.
- `send_email`: send an email (via SMTP or integrated email service).
- `call_webhook`: make an HTTP POST to a configured webhook URL with item details.

Each rule will be stored in a database table (`automation_rules`), and every time it fires, we record an entry in an `automation_logs` table for auditing.

Database Schema

We can model rules and logs in PostgreSQL (or any RDBMS). For flexibility we store trigger/action configurations in JSON columns. Example SQL might be:

```
CREATE TABLE automation_rules (
  rule_id     SERIAL PRIMARY KEY,
  board_id    INTEGER NOT NULL,
  trigger_type VARCHAR(50) NOT NULL,
  trigger_config JSONB,     -- e.g. {"field": "status", "operator": "=",
"value": "Done"} or {"cron": "0 9 * * MON"}
  action_type   VARCHAR(50) NOT NULL,
  action_config JSONB,      -- e.g. {"user_id": 17} or {"field": "status",
"value": "Ready"}
  created_by    INTEGER,
  created_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX ON automation_rules(board_id);

CREATE TABLE automation_logs (
  log_id       SERIAL PRIMARY KEY,
  rule_id      INTEGER REFERENCES automation_rules(rule_id),
  triggered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  action_type  VARCHAR(50),
  action_status VARCHAR(20),  -- e.g. 'success' or 'failed'
  error        TEXT,
  details      JSONB         -- optional context like item_id, field changes,
etc.
);
CREATE INDEX ON automation_logs(rule_id);
```

In this design, each `trigger_config` and `action_config` is a JSON object that the system parses when evaluating the rule. For example, a `trigger_config` for `on_field_change` might be:

```
{ "field": "status", "operator": "=", "value": "Done" }
```

and for `cron_schedule` it might be:

~~~
{ "cron": "0 9 * * MON" }
```.  Action configs would similarly hold target values (e.g. `{"user_id":
42}` for `notify_user`, or `{"email": "client@example.com", "subject": "...",
"body": "..."}` for `send_email`).

## Backend/Worker Logic

A background **Worker** process will monitor and execute automation rules.
It will handle events and schedules, and call actions reliably (with error
handling and retries). Key parts of the logic include:

- **Item Update / Field Change**: When an item is updated (in the ORM or
API), check for `on_field_change` rules on that board.  For each rule
matching the updated field and satisfying the condition (e.g. new status
= “Done”), invoke the action.  Similarly, on item creation, check any
`on_item_create` rules.
- **Scheduled Triggers**: Use a scheduler (like [`node-cron`](https://
github.com/node-cron/node-cron)) to handle date-based and cron triggers.  For
example, we can set up:
  ```js
  const cron = require('node-cron');
~~~

~~~
  // Every day at midnight, check date triggers
  cron.schedule('0 0 * * *', () => { checkDateArriveTriggers(); });
  // Example from node-cron usage:
  // cron.schedule('5 * * * * *', () => console.log('runs every minute at 5
seconds') );
  ```
  As shown in the LogRocket guide on `node-cron`, scheduling tasks by cron
expression in Node.js is straightforward 【21†L155-L163】. Each scheduled
tick the worker queries `automation_rules` where `trigger_type =
'on_date_arrive'` and evaluates date fields against the current date (and
also checks `cron_schedule` rules that match now).
- **Action Execution**: When a rule fires, call a function like
`executeAction(rule, item)`. This function should handle each action type
in a `switch` or if/else. For example:
  ```js
  async function executeAction(rule, item) {
    try {
      switch(rule.action_type) {
        case 'notify_user':
          // send notification to user ID in rule.action_config.user_id
          break;
        case 'set_field_value':
          // update item: item[rule.action_config.field] =
rule.action_config.value
          break;
        case 'create_item':
          // insert new item on the specified board
          break;
        case 'send_email':
          // call email API with config
          break;
        case 'call_webhook':
          // fetch POST to rule.action_config.url with item data
          break;
      }
      // Log success
      await db.query(
        'INSERT INTO automation_logs(rule_id, action_type, action_status,
details) VALUES ($1,$2,$3,$4)',
        [rule.rule_id, rule.action_type, 'success', { item_id: item.id }]
      );
    } catch(err) {
      // Log failure with error message
      console.error('Automation error:', err);
      await db.query(
        'INSERT INTO automation_logs(rule_id, action_type, action_status,
error, details) VALUES ($1,$2,$3,$4,$5)',
~~~

```
        [rule.rule_id, rule.action_type, 'failed', err.toString(), {
item_id: item.id }]
      );
    }
  }
```

It's important to wrap action code in `try/catch` and log errors (potentially with a retry strategy). Best practices in scheduled tasks emphasize using try/catch and logging for reliability【7†L389-L397】. For example, the LogRocket guide suggests:

```js
cron.schedule('5 * * * *', () => {
  try {
    // ...action logic...
  } catch (error) {
    console.error('Task failed:', error);
    // retry or log
  }
});
```

which we would emulate for each automated action.

- **Error Handling and Retries**: Depending on the action type (especially external calls like email or webhook), implement retries (exponential backoff) and definitely record failures in `automation_logs`. This audit log table will hold one row per trigger firing, so admins can review what was done or why something failed.

API Endpoints

We expose a simple REST API to manage rules per board. For example (using Express.js pseudo-code):

```js
// Create a new automation rule on a board
app.post('/api/boards/:boardId/automations', async (req, res) => {
  const { boardId } = req.params;
  const { trigger_type, trigger_config, action_type, action_config } =
req.body;
  // (Validate input...)
  const result = await db.query(
    `INSERT INTO automation_rules
      (board_id, trigger_type, trigger_config, action_type, action_config,
created_by)
     VALUES ($1,$2,$3,$4,$5,$6) RETURNING *`,
    [boardId, trigger_type, trigger_config, action_type, action_config,
req.user.id]
  );
  res.json(result.rows[0]);
});

// List all rules for a board
app.get('/api/boards/:boardId/automations', async (req, res) => {
  const { boardId } = req.params;
  const result = await db.query(
    `SELECT * FROM automation_rules WHERE board_id = $1`, [boardId]
  );
  res.json(result.rows);
});

// Delete a rule by ID
app.delete('/api/automations/:ruleId', async (req, res) => {
  const { ruleId } = req.params;
  await db.query(`DELETE FROM automation_rules WHERE rule_id = $1`,
[ruleId]);
  res.status(204).send();
});
```

These endpoints allow the frontend to create, list, and remove rules. (Authentication/authorization checks should ensure only board members or admins can do this.)

Frontend (React) – Automation Builder UI

On the front-end, we build a **No-Code Automation Builder** interface. This is a form-driven UI (wizard) that lets users select triggers and actions step-by-step. Key ideas:

- **Trigger Selection**: First, the user picks a trigger type from a dropdown (e.g. “Field changes”, “Date arrives”, “Item created”, “Schedule”). Then additional controls appear:
- If “Field changes” is chosen, show another dropdown listing board fields (e.g. *Status*, *Priority*, custom fields), plus an operator (=, ≠, etc.) and a value input.
- If “Date arrives” is chosen, show a date field selector and maybe offset (e.g. “0 days before”).
- If “Schedule” is chosen, show a schedule builder (could be a cron expression input or friendly recurrence like daily/weekly time).
- **Action Selection**: Next, the user selects an action type (“Notify user”, “Set field”, etc.). Based on the choice:
- For `notify_user`, show a user/team dropdown or email input.
- For `set_field_value`, show a field dropdown and a value input.
- For `send_email`, show fields for recipient, subject, body.
- For `call_webhook`, show an input for URL and possibly headers or payload template.
- **Form State & Logic**: In React we manage state for the current step. For example:

```
function AutomationBuilder({ boardId }) {
  const [step, setStep] = useState(1);
  const [triggerType, setTriggerType] = useState('');
  const [triggerConfig, setTriggerConfig] = useState({});
  const [actionType, setActionType] = useState('');
  const [actionConfig, setActionConfig] = useState({});
  // (Load existing rules for the board)
  const [rules, setRules] = useState([]);
  useEffect(() => {
    fetch(`/api/boards/${boardId}/automations`)
      .then(res => res.json()).then(setRules);
  }, [boardId]);

  // Step 1: pick trigger
  if (step === 1) {
    return (
      <div>
        <h3>When this happens...</h3>
        <select value={triggerType} onChange={e =>
setTriggerType(e.target.value)}>
          <option value="">Select trigger</option>
          <option value="on_field_change">Field changes</option>
          <option value="on_date_arrive">Date arrives</option>
          <option value="on_item_create">Item created</option>
          <option value="cron_schedule">On schedule</option>
        </select>
        {triggerType === 'on_field_change' && (
          <>
            <select onChange={e => setTriggerConfig({ ...triggerConfig, field:
e.target.value })}>
              <option value="">Select field</option>
              <option value="status">Status</option>
              <option value="priority">Priority</option>
              {/* ...other fields */}
            </select>
            <select onChange={e => setTriggerConfig({ ...triggerConfig,
operator: e.target.value })}>
              <option value="=">=</option>
              <option value="!=">≠</option>
            </select>
            <input
              placeholder="Value"
              onChange={e => setTriggerConfig({ ...triggerConfig, value:
e.target.value })}
            />
          </>
        )}
        {/* Additional UI for other trigger types ... */}
        <button onClick={() => setStep(2)}>Next</button>
      </div>
    );
  }
  // Step 2: pick action
  if (step === 2) {
    return (
      <div>
        <h3>Then do this...</h3>
        <select value={actionType} onChange={e => setActionType(e.target.value)}
>
          <option value="">Select action</option>
          <option value="notify_user">Notify User</option>
          <option value="set_field_value">Set Field</option>
          <option value="create_item">Create Item</option>
          <option value="send_email">Send Email</option>
          <option value="call_webhook">Call Webhook</option>
        </select>
        {actionType === 'notify_user' && (
          <select onChange={e => setActionConfig({ user_id: e.target.value })}>
            {/* populate with board members or teams */}
            <option value="">Select user</option>
            <option value="17">Alice</option>
            <option value="23">Bob</option>
          </select>
        )}
        {actionType === 'set_field_value' && (
          <>
            <select onChange={e => setActionConfig({ ...actionConfig, field:
e.target.value })}>
              <option value="">Field</option>
              <option value="status">Status</option>
              <option value="priority">Priority</option>
            </select>
            <input
              placeholder="Value"
              onChange={e => setActionConfig({ ...actionConfig, value:
e.target.value })}
            />
          </>
        )}
        {/* Additional UI for other actions... */}
        <button onClick={() => {
          // Save the new rule via API
          fetch(`/api/boards/${boardId}/automations`, {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({ trigger_type: triggerType, trigger_config:
triggerConfig,
                                   action_type: actionType, action_config:
actionConfig })
```

```
          }).then(() => {
            setStep(1); // go back to listing
            // reload rules...
          });
        }}>Save Rule</button>
      </div>
    );
  }
  // After steps, show list of rules
  return (
    <div>
      <h3>Automation Rules</h3>
      {rules.map(rule => (
        <div key={rule.rule_id} className="rule-card">
          <b>When</b> {rule.trigger_type} {/* format rule.trigger_config */}
          <b>Then</b> {rule.action_type} {/* format rule.action_config */}
          <button onClick={() => {/* delete rule */}}>Delete</button>
        </div>
      ))}
    </div>
  );
}
```

This React code is a simplified scaffold. It conditionally renders form controls based on the selected trigger/action type. Libraries like SurveyJS show similar dynamic form building: e.g. SurveyJS's Form Builder is an open-source React component that generates forms (with drag-and-drop) and supports conditional rules [4]. In our custom UI, we manually show/hide fields based on state (e.g. `triggerType === 'on_field_change'`).

Finally, we would style the UI and add validation. Each board's page would list existing rule *cards* (showing e.g. "When [field]=[value] Then [action]"), with Edit/Delete buttons. A prebuilt template (like "Notify client when status is Ready") could be provided by populating a default form.

Overall, this design covers the **full stack** for no-code automations: database tables for rules/logs, backend workers and schedulers to watch for triggers and execute actions (with logging), REST endpoints to manage rules, and a React-based form UI to let users build and view rules. The use of JSON configs and cron syntax gives flexibility, while robust worker code (with try/catch and audit logging) ensures reliable execution [2] [3].

Sources: We follow standard trigger-action patterns (as in Monday.com's automation model [1]), use cron scheduling libraries for timed events [2], and adopt dynamic form techniques (similar to SurveyJS's conditional form builder [4]) to implement the UI.

---

[1] Automation and Integration actions – Support
https://support.monday.com/hc/en-us/articles/360017556179-Automation-and-Integration-actions

[2] [3] Scheduling tasks in Node.js using node-cron - LogRocket Blog
https://blog.logrocket.com/task-scheduling-or-cron-jobs-in-node-using-node-cron/

[4] React Form Builder | Generate dynamic HTML forms using JSON
https://surveyjs.io/survey-creator/documentation/get-started-react