

Node-Red Work Sample — UI Builder with SMTP Notification

Tender Work Sample— Build & Evaluation Guide

07 July 2026

Table of Contents

Document Overview	3
Developer Resources and References	3
UI Builder resources	3
Add-on references	3
Work Sample Overview	3
What you build.....	4
Architecture (how the pieces fit).....	4
Scope.....	4
Expected effort.....	4
Required capabilities.....	4
Runtime notes	5
Terminology note	5
Scenario: Contact Request Notification	5
User story.....	5
Form fields.....	5
End-to-end flow.....	6
Message contract	6
Example email.....	7
Implementation Outline.....	7
Functional requirements	7
Non-functional expectations	7
Step 1 — Define the action and the contract.....	8
Step 2 — Create the UIBUILDER endpoint and verify transport	8
Step 3 — Build the custom form (frontend)	8
Step 4 — Implement the ORCE dispatcher	8
Step 5 — Validate the form (backend responsibility)	8
Step 6 — Send the notification (backend SMTP responsibility)	8
Step 7 — Build the normalized response and handle it in the UI.....	9

Step 8 — Document the result.....	9
Submission Instructions	9
What to submit	9
How to package	9
Configuration and secrets	10
Assessment	10
Annex A — Build Guideline	11
Step-by-step implementation guideline for a custom frontend with UIBUILDER and generic backend flows in ORCE.....	11
Document goal	11
What this guide covers	11
1. Architecture at a glance.....	12
2. What UIBUILDER is in practical terms	12
3. End to end interaction model.....	13
4. Standard implementation blueprint.....	13
5. Frontend implementation guideline.....	14
6. Frontend state model	15
7. Message contract guideline.....	16
8. ORCE backend design guideline	17
9. Frontend and backend integration pattern	18
10. Async work and error handling.....	19

Document Overview

This document is a self-contained, static work sample for tender use. It is a single document and requires access to a source-code repository and separate files as listed only in THIS document— everything needed is included below, including the full build guideline as Annex A.

By successfully completing this work sample the bidder demonstrates practical Node-RED know-how **without requiring an official project reference to prove technical capability with regards to the requirement “Usage of Node-Red based Workflows or similar flow-based visual programming for building automations, integrations, and event-driven applications” as stated in Chapter III 5.1 of the process description included in the tender documents.**

Developer Resources and References

To understand the overall conceptual approach, the following resources should be reviewed alongside this document.

UI Builder resources

- **FAP UI Builder guideline** — included in full as **Annex A — Build Guideline** for single flows at the end of this document.

Add-on references

1. **Orchestration Engine (ORCE)** — <https://github.com/eclipse-xfsc/orchestration-engine>
2. **Reference FAP repository** — <https://github.com/eclipse-xfsc/facis-fap-partner-onboarding>
3. **UIBUILDER GitHub repository** — <https://github.com/TotallyInformation/node-red-contrib-uibuilder>
4. **UIBUILDER package page** — <https://flows.nodered.org/node/node-red-contrib-uibuilder>
5. **Vue.js official documentation** — <https://vuejs.org/guide/introduction>
6. **Node-RED official documentation** — <https://nodered.org/docs/>

Work Sample Overview

The XFSC ORCE is a low-code orchestration engine (based on Node-RED) that uses the **ORCE UI Builder (UIBUILDER)** to serve a custom web UI and connect it to backend flows.

What you build

A small **Contact Request Notification** feature, implemented as the guideline in Annex A prescribes:

- a **custom UIBUILDER frontend** (one contact form screen), and
- a set of **Node-RED backend flows** that validate the input and send an email notification, communicating over one stable message contract.

Architecture (how the pieces fit)

Browser (UIBUILDER form)

| command message { action: "submitContactRequest", payload: {...} }



ORCE / Node-RED

uibuilder in → normalize → validate envelope → route by action

→ validate form (field errors if invalid)

→ send notification (SMTP, backend "Notification" responsibility)

→ build normalized result → uibuilder out

| result message { ok, ui.toast, errors }



Browser updates state and shows success / error

Key point: **SMTP lives in the backend**, as a Notification responsibility inside ORCE that the workflow triggers after validation. The UIBUILDER frontend never talks to SMTP directly. It only sends a command and renders the result.

Scope

In scope:

- One UIBUILDER form screen (custom frontend).
- One message contract (command request / result response).
- ORCE backend: a small dispatcher routing to a `validate form` step and a `send notification (SMTP)` step.
- Success and error feedback returned to the UI.

Out of scope:

- Authentication, user management, persistence/databases.
- Production hardening, scaling, or deployment automation.
- multi-stage dashboard

Expected effort

A standard developer should be able to complete the sample in **approximately 4 hours**.

Required capabilities

- Serve a custom UI with the ORCE UI Builder (UIBUILDER).

- Send and receive messages over one stable contract.
- Implement an ORCE dispatcher that routes by action.
- Validate input in a backend flow and return structured field errors.
- Integrate a simple external API (SMTP) as a backend Notification responsibility.
- Return explicit success / failure and map low-level errors to frontend-safe errors.

Runtime notes

This document is the complete work sample and is not distributed as a runnable repository. The bidder must run ORCE locally to complete the task. The ORCE UI Builder can be installed via **Menu > Manage palette > Install > node-red-contrib-uibuilder**. For bidder submissions, SMTP credentials must be provided through environment variables, without real secret values.

Terminology note

For developers new to ORCE, the building blocks map to widely known low-code tools:

In this sample	Practical equivalent
ORCE engine	Node-RED-based orchestration runtime
ORCE UI Builder	UIBUILDER (<code>node-red-contrib-uibuilder</code>) custom web UI
ORCE workflow	Backend flows: dispatcher + validate + send-notification
SMTP integration	Backend Notification responsibility (email/SMTP node or SMTP client)

Scenario: Contact Request Notification

Build this the way the Build Guideline (Annex A) describes: a custom UIBUILDER frontend talking to ORCE backend flows over **one message contract**. SMTP is a backend Notification responsibility — the frontend never calls SMTP directly.

User story

As a user, I want to submit a contact request through a simple UI, so that the responsible team receives an email notification.

Form fields

Field	Required	Notes
Name	Yes	Free text
Email	Yes	Must match a basic email format
Subject	Yes	Free text
Message	Yes	Multi-line text

End-to-end flow

1. The browser opens the form served by UIBUILDER and starts the UIBUILDER client.
2. The user fills in the fields and submits.
3. The frontend sends **one command message** (action: "submitContactRequest") — it does not validate business rules locally beyond basic UX hints, and it does not talk to SMTP.
4. ORCE receives the message: `uibuilder in` → `normalize` → `validate envelope` → `route by action`.
5. The **validate form** step checks required fields and email format. On failure it returns `errors.fields` and stops (no email sent).
6. The **send notification** step (backend SMTP responsibility) sends the email using configuration from environment variables and maps any SMTP failure to a frontend-safe error.
7. ORCE builds **one normalized result** and returns it via `uibuilder out`.
8. The frontend reduces the result: shows a success notification, or renders field errors / a general error, staying on the same screen.

Message contract

This is the contract for the sample:

Request — browser → ORCE:

```
{
  "type": "command",
  "action": "submitContactRequest",
  "requestId": "uuid-from-browser",
  "payload": {
    "name": "Jane Doe",
    "email": "jane@example.com",
    "subject": "Question about onboarding",
    "message": "Could you share the next steps?",
  },
  "meta": { "source": "dashboard" }
}
```

Success response — ORCE → browser:

```
{
  "type": "result",
  "ok": true,
  "action": "submitContactRequest",
  "requestId": "same-request-id",
  "data": { "messageId": "smtp-message-id-or-mock" },
  "ui": { "toast": { "kind": "success", "message": "Your request was sent." } },
  "errors": null
}
```

Validation error response — field-level:

```
{
  "type": "result",
  "ok": false,
  "action": "submitContactRequest",
  "requestId": "same-request-id",
  "ui": { "toast": { "kind": "error", "message": "Please correct the highlighted fields." } },
  "errors": { "fields": { "email": "Please enter a valid email address." } }
}
```

Integration (SMTP) error response — action-level:

```
{
  "type": "result",
  "ok": false,
  "action": "submitContactRequest",
  "requestId": "same-request-id",
  "ui": { "toast": { "kind": "error", "message": "Could not send your request. Please try again." } },
  "errors": { "action": "smtpDeliveryFailed" }
}
```

Example email

To: <configured recipient, e.g. SMTP_TO>
From: <configured sender, e.g. SMTP_FROM>
Subject: New contact request: {{subject}}

Name: {{name}}
Email: {{email}}
Priority: {{priority}}

Message:
{{message}}

Implementation Outline

Functional requirements

- Serve a custom UIBUILDER form capturing name, email, subject, message, priority.
- One command request → one result response (see the Scenario section).
- Validate required fields and basic email format **in ORCE** (server-side).
- Send one email via SMTP as a backend Notification responsibility.
- Return explicit success and structured error states to the UI.

Non-functional expectations

- No hardcoded credentials.

- Clear separation of layers: UIBUILDER bridges only; ORCE owns business logic; the UI owns presentation state.
- One inbound handler and one outbound command on the frontend (no per-action parsers).
- Readable, self-explanatory setup steps.
- Minimal scope — not production hardening.

Step 1 — Define the action and the contract

Decide the single action name (`submitContactRequest`) and the request/response envelopes before writing UI code. Use the contract in the Scenario section (message contract). Defining the contract first prevents payload drift between frontend and backend.

Step 2 — Create the UIBUILDER endpoint and verify transport

Add a UIBUILDER node, serve the page, and confirm browser ↔ backend messaging with a simple ping before building the form. The browser should never need to guess backend behavior.

Step 3 — Build the custom form (frontend)

Create the contact form (name, email, subject, message, priority) as a custom UIBUILDER page. Keep it to one screen. Wire a single outbound `submit(action, payload)` that sends the command, and a single inbound handler that reduces the result into UI state (toast + field errors). Keep presentation state (`ui`) separate from business data.

Step 4 — Implement the ORCE dispatcher

In ORCE: `uibuilder in → normalize request → validate envelope → route by action`. The dispatcher only routes; it must not contain all the business logic. Route `submitContactRequest` to the steps below.

Step 5 — Validate the form (backend responsibility)

Validate required fields and basic email format. On failure, return a result with `ok: false` and `errors.fields`, and **do not send an email**. Separate validation from execution.

Step 6 — Send the notification (backend SMTP responsibility)

Implement a `send_notification` step that maps the validated payload to the email template (see the Scenario section) and sends it via SMTP — using an email/SMTP node or any SMTP client. Read all SMTP settings from environment variables; **do not hardcode credentials**.

```
SMTP_HOST=
SMTP_PORT=
SMTP_USERNAME=
SMTP_PASSWORD=
```

SMTP_FROM=
SMTP_TO=

Map any low-level SMTP error to a frontend-safe action-level error (`errors.action`) before returning. Never let a missing property imply success.

Step 7 — Build the normalized response and handle it in the UI

ORCE builds one `result` (success or explicit failure) and returns it via `uibuilder out`. The frontend reducer:

- shows a success notification on `ok: true`,
- renders inline field errors on `errors.fields`,
- shows a general error banner on `errors.action`, and
- keeps the user on the same screen for any error.

Step 8 — Document the result

Provide short setup instructions, screenshots (form + success + error), and a brief explanation of design decisions (how you split UI / contract / validation / notification).

Submission Instructions

What to submit

The deliverable is a **UIBUILDER frontend and an Node-RED flow** that together implement the scenario. Concretely:

- the **UIBUILDER front-end files** (the served `index.html / index.js / index.css`, or an export of them);
- the **exported Node-RED flow(s)** (`flows.json`) implementing the dispatcher, the validation step, and the SMTP notification step;
- a `.env.example` listing the required SMTP variables (no real values);
- **screenshots** of the form and of the success and error states (and of the flow, if helpful);
- short setup instructions and a brief explanation of design decisions (how you split UI / contract / validation / notification).

How to package

Preferred a zipped folder:

```
your-submission/  
  README.md           # how to run / review  
  flows.json          # exported ORCE / Node-RED flow(s)  
  ui/                 # UIBUILDER front-end files (index.html/js/css) or export  
assets/screenshots/  # form, success, error (and flow) screenshots
```

```
.env.example          # required SMTP variables, no real values
```

Configuration and secrets

- Provide a `.env.example` listing the required variables (`SMTP_HOST`, `SMTP_PORT`, `SMTP_USERNAME`, `SMTP_PASSWORD`, `SMTP_FROM`, `SMTP_TO`).
- **Do not include real credentials or secrets** in the submission.
- If a live SMTP server is not available, a local test server (e.g. Mailpit/MailHog) or a clearly documented mock is acceptable.

Assessment

All three acceptance criteria must be on “pass” for the bidder to proof the requirement “Usage of Node-Red based Workflows or similar flow-based visual programming for building automations, integrations, and event-driven applications” as stated in Chapter III 5.1 of the process description included in the tender documents.

Acceptance criterion	Pass?
The UIBUILDER form renders and accepts the following listed fields: name, email address, subject, priority, message.	<input type="checkbox"/>
All field validation happens in ORCE: a missing required field returns “errors.fields” and no email is sent.	<input type="checkbox"/>
Valid input sends exactly one email and returns “ok”: true with a success notification in the UI.	<input type="checkbox"/>

Annex A — Build Guideline

Step-by-step implementation guideline for a custom frontend with UIBUILDER and generic backend flows in ORCE

Document goal

This document explains a suggested way to build a dashboard-oriented interface for ORCE work sample purposes.

The purpose is to describe a reusable implementation approach that any development team can follow so that:

- the frontend follows a consistent reference-style dashboard pattern,
- the architecture remains consistent across dashboard-oriented implementations,
- the frontend and backend are separated cleanly,
- the implementation can be used directly by developers or fed to an AI assistant or vibe coding workflow.

In other words, this is a practical build guide for reproducing the same UI language, the same interaction style, and the same frontend-backend split, while still allowing each team to implement its own business logic.

What this guide covers

This guide explains:

- what UIBUILDER is and what role it plays,
 - how the frontend and backend connect end to end,
 - how to design generic ORCE backend flows without coupling them too tightly to pages,
 - how to define clean frontend state, contracts, routing, and asynchronous handling,
-

What this guide enforces

This guide recommends the following principles:

1. **Frontend should be custom**, not assembled from low-flexibility wizard tooling.
2. **UIBUILDER should act as the bridge**, not as the location of business logic.
3. **ORCE should own orchestration and backend behavior**.
4. **The UI should follow the reference-style dashboard pattern**, not an unrelated dashboard style.
5. **The implementation should remain reusable and maintainable**, especially for future reference-style UIs.

1. Architecture at a glance

At a high level, the solution should always be understood as four layers:

Layer	Owns	Should contain	Should not contain
Frontend	visible UI state	templates, form state, transitions, user interaction, local formatting, loading and feedback state	orchestration logic, deep validation rules, external integration sequencing
UIBUILDER	browser to ORCE bridge	page endpoint, browser connection, message send and receive, helper libraries	business rules
ORCE	orchestration and backend logic	routing, validation, persistence, async jobs, service integration, response shaping	DOM, CSS, page layout concerns
Services	external capabilities	DB, PKI, APIs, registries, storage, notification channels	page logic

Core design principle

The browser owns **presentation state**.

ORCE owns **business logic**.

The contract between them keeps both sides synchronized.

That rule should remain true in every FAP-like implementation.

2. What UIBUILDER is in practical terms

UIBUILDER is the layer that allows you to:

- serve a custom web page from the Node-RED or ORCE environment,
- connect that page to backend flows,
- send messages from browser to backend,
- receive normalized messages back from backend,
- and keep the UI fully custom.

Practical mental model

Think of UIBUILDER as the **doorway**, not the **brain**.

The browser enters through the UIBUILDER URL.

The browser sends messages through the UIBUILDER client.

The browser receives responses through the same bridge.
The actual decisions still live in ORCE.

Why that matters

This keeps the frontend flexible and branded, while still allowing the backend to remain flow-based and orchestration-driven.

That is exactly the model you want when the UI must feel like a real product surface and not like a generic low-code screen.

3. End to end interaction model

The recommended lifecycle is:

1. The browser opens the page from the UIBUILDER endpoint.
2. The page starts the UIBUILDER client connection.
3. A user action triggers a structured message.
4. ORCE receives the message.
5. ORCE validates it and routes it to the correct backend responsibility.
6. ORCE performs validation, persistence, integration, or asynchronous work.
7. ORCE returns a normalized response message.
8. The frontend updates its state and re-renders the screen.
9. If work is long-running, ORCE can stream progress events before the final result.

Core rule

The browser should never need to guess backend behavior.

The backend should always respond in a form the browser can consume predictably.

4. Standard implementation blueprint

Before writing any UI code, define the implementation in this order:

Step	Define	Output	Owned by	Why first
1	business stages	stage list and transitions	product + backend	prevents UI-only thinking
2	request and event contract	request and response envelopes	backend + frontend	prevents payload drift
3	backend routing model	root dispatcher and subflows	backend	keeps ORCE clean
4	frontend view model	what each screen must render	frontend	reduces conditional spaghetti

5	design tokens	colors, spacing, radii, states	frontend or design	keeps implementation visually consistent
---	---------------	--------------------------------	--------------------	--

Common mistake to avoid

Do **not** start by designing screens first and contracts later.

That approach usually creates:

- duplicated state,
- inconsistent payloads,
- hidden logic in the browser,
- and pages that are difficult to maintain.

5. Frontend implementation guideline

The frontend should be implemented as a **custom dashboard**, not as a simple wizard-only shell.

Recommended file structure

```

/dashboard
  index.html
  index.css
  index.js
  pages/
    welcome.js
    profile.js
    documents.js
    review.js
  components/
    app-shell.js
    sidebar.js
    status-chip.js
    toast.js
    modal.js
    form-field.js
    progress-bar.js
  services/
    transport.js
    state.js
    reducers.js
  assets/
    logos/
    icons/

```

Responsibility rules

- index.html should remain minimal.
- index.css should centralize design tokens and core component styles.

- `index.js` should bootstrap the app, initialize UIBUILDER, and mount the state and render cycle.
- `pages/` should contain view-specific logic only.
- `components/` should stay reusable and visually consistent.
- `services/transport.js` should be the single place where UIBUILDER interaction is defined.
- `services/state.js` should define the frontend state container.
- `services/reducers.js` should normalize every backend message into state updates.

6. Frontend state model

A clean dashboard feels clean because the browser state is clean.

Even when the backend is complex, the frontend should reduce everything to a small number of concepts.

Recommended state blocks

State block	Purpose	Typical source	Rule
session	correlation, restore, resume	server bootstrap	never invent it locally
step or stage	current screen	server plus controlled local transitions	use one source of truth
model	business data shown on the screen	server responses and local form input	after submit, server becomes source of truth
ui	loading, modal, toast, focus, local transient state	frontend only	do not mix with business data
errors	field, action, integration, or system errors	backend and local validation	keep structured and explicit

Recommended browser state skeleton

```
const state = {
  sessionId: null,
  step: 'welcome',
  model: {
    profile: {},
    files: {},
    records: {},
    results: {}
  },
  ui: {
    loading: false,
    modal: null,
    toast: null,
    pendingAction: null
  }
}
```

```
  },  
  errors: {}  
}
```

State rule

The `model` block contains business data.

The `ui` block contains purely presentational state.

The `errors` block contains actionable feedback.

Do not leak visual concerns into the backend contract unless truly needed.

7. Message contract guideline

A strong contract is the most important engineering decision in this architecture.

The browser should not need a different parser for every action.

Recommended request envelope

```
{  
  "type": "command",  
  "action": "saveProfile",  
  "sessionId": "optional-on-first-request",  
  "step": "company-profile",  
  "requestId": "uuid-from-browser-or-server",  
  "payload": {  
    "profile": {  
      "companyName": "Example GmbH"  
    }  
  },  
  "meta": {  
    "clientTs": 1710000000,  
    "source": "dashboard"  
  }  
}
```

Recommended response envelope

```
{  
  "type": "result",  
  "ok": true,  
  "action": "saveProfile",  
  "sessionId": "abc-123",  
  "requestId": "same-request-id",  
  "next": {  
    "step": "documents",  
    "availableActions": ["uploadFiles", "back"]  
  },  
  "data": {  
    "profile": {  
      "companyName": "Example GmbH"  
    }  
  }  
}
```

```

    }
  },
  "ui": {
    "toast": {
      "kind": "success",
      "message": "Profile saved"
    }
  },
  "errors": null
}

```

Recommended async event envelope

```

{
  "type": "event",
  "event": "jobProgress",
  "data": {
    "jobId": "job-77",
    "stage": "calling-service",
    "progress": 60,
    "label": "Processing verification"
  }
}

```

8. ORCE backend design guideline

The backend should not be organized as one giant page-by-page flow.

It should be organized as a root dispatcher plus responsibility-based subflows.

Core rule

A page can call more than one backend responsibility.

A backend responsibility can support more than one page.

Once the team follows that rule, the backend becomes far more maintainable.

Recommended ORCE topology

```

uibuilder in
  -> normalize request
  -> validate envelope
  -> route by action
      -> bootstrap session
      -> validate form
      -> save model
      -> run integration
      -> launch async job
      -> build export
      -> send notification
  -> build normalized response
  -> uibuilder out

```

Flow types you usually need

Flow type	Responsibility	Typical input	Typical output
Bootstrap	create or restore session	page open or resume	session state and first view model
Validation	validate user input and rules	form payload	normalized data or field errors
Persistence	save or retrieve records	validated model	stored model and revision info
Integration	call external systems	domain command	mapped result or safe error
Async worker	track long jobs	job request	job id, progress events, final result
Export	prepare downloadable output	domain record	file metadata, URL, or encoded content
Notification	send email, chat, or audit events	trigger plus context	acknowledgement

Subflow design rules

1. Define one clear input shape.
2. Normalize data at the subflow boundary.
3. Separate validation from execution.
4. Return explicit success or explicit failure.
5. Map low-level errors into frontend-safe errors before returning.

9. Frontend and backend integration pattern

The best frontend integration is intentionally boring.

Use:

- one inbound message handler,
- one reducer path,
- one render path.

Do not create page-specific parsing patterns all over the codebase.

One inbound handler

```
uibuilder.onChange('msg', (msg) => {  
  switch (msg.type) {  
    case 'result':  
      reduceResult(msg)  
      break  
    case 'event':
```

```

        reduceEvent(msg)
        break
    default:
        reduceUnknown(msg)
    }
    render()
})

```

One outbound command function

```

function submit(action, payload) {
    state.ui.loading = true
    state.ui.pendingAction = action
    render()

    uibuilder.send({
        type: 'command',
        action,
        sessionId: state.sessionId,
        step: state.step,
        payload
    })
}

```

Reducer rules

- persist `sessionId` when received,
- switch screens from `next.step`,
- merge only the relevant data block,
- keep `ui.toast` and `ui.modal` separate from the business model,
- keep the same step when errors are returned.

Refresh and recovery

A dashboard should survive refreshes and temporary disconnects.

Recommended rule:

- store only the minimum needed locally, usually `sessionId`,
- ask the backend to restore the current state,
- if restore fails, restart cleanly rather than guessing.

10. Async work and error handling

Async work pattern

When a backend action may take longer, do not make the UI wait blindly.

Use a job model:

1. browser sends command,

2. backend acknowledges quickly,
3. backend emits progress updates,
4. backend emits final result,
5. UI shows progress and remains responsive.

Error handling pattern

Error level	Meaning	Backend handling	Frontend behavior
Field	invalid or missing input	return <code>errors.fields</code>	highlight fields inline
Action	rule or state failure	return <code>errors.action</code>	show banner and stay on current step
Integration	external service issue	return safe mapped error	offer retry or support path
System	unexpected failure	log internal detail, return safe message	show global error and recovery option

Mandatory rule

Never use missing properties to imply success.

Always return explicit success or explicit failure.