Technical Assessment: Customizable Dashboard Builder

Objective

Build a small web application that allows an authenticated user to create and customize a personal dashboard made up of simple widgets.

The goal is to assess:

- Frontend architecture and state management
- User authentication and persistence
- Clean, maintainable code
- UI design, interactivity, and usability

Expected time: **5–6 hours total**
Candidates should focus on a clean, functional solution rather than extensive polish or features.

Functional Overview

An authenticated user should be able to:

1. Register and sign in
2. Add widgets from a predefined set
3. Remove widgets
4. Edit widget content (where applicable)
5. Save and load their personal dashboard layout

Core Requirements

Frontend

- Framework: Next.Js (preferred) but you can use React.Js also
- State management: Zustand (preferred) but feel free to use the one you are comfortable in

Required features

- **Authentication**
  - Registration and login with email and password
  - Authenticated routes (only logged-in users can access their dashboard)
  - Session persistence across page refreshes (via cookies or tokens)
  - Logout functionality
- **Dashboard**
  - Allow users to add and remove widgets
  - At least three widget types, such as:
    - Clock: displays current time and updates every second
    - Notes: editable text area
    - Todo: simple list of items that can be added, checked, or deleted
  - Optional: a Weather widget showing mock or API-based data
  - "Save Dashboard" button to persist the layout and content
  - On page load, fetch and display the saved dashboard for the logged-in user
- **UX and Visual Design**
  - Simple, clean, and responsive layout (desktop-first is fine)
  - Visible feedback for loading and errors

    - Basic form validation (for example, required fields in auth forms)

Backend

Candidates may use **any backend language or framework** they are comfortable with. Examples include Node.js (Express, Nest.js), Python (Django, FastAPI, Flask), Go, Ruby on Rails, or others.

Note - If you are not comfortable with backend development you can choose to create a pass through backend with Node.js APIs which return SUCCESS in any case.

Requirements

- Implement user registration, authentication, and session handling
- Provide at least the following endpoints:

1. **POST /auth/register**
   Registers a new user (email, password). Returns a session or token on success.

2. **POST /auth/login**
   Logs in an existing user and returns a session or token.

3. **POST /auth/logout**
   Logs out the current user or invalidates the session.

4. **GET /dashboard**
   Auth required. Returns the saved dashboard configuration for the current user.

5. **POST /dashboard**
   Auth required. Saves the dashboard configuration for the current user.

Persistence

- Use any storage system: Sqlite, Postgres, Files
- Each user should have their own dashboard record.
- Store dashboard data as a structured list of widgets with type, position, and optional data.

Note - If you are working with a pass through backend you can choose to save data in files or in browser's localStorage as well so it will simulate persistence.

Data Structure (recommended)

Each dashboard can contain multiple widgets.
A widget should include:

- A unique ID
- A type (for example: clock, notes, todo)
- A position (order in the dashboard)
- Optional data such as note text or todo items

Example structure:

```
None
{
  "widgets": [
    { "id": "w1", "type": "clock", "position": 0 },
    { "id": "w2", "type": "notes", "position": 1, "data": {
"text": "Project meeting at 2 PM" } }
  ]
}
```

User Flow

1. The user registers and logs in.
2. After authentication, they are redirected to the dashboard view.
3. The dashboard displays their saved widgets (if any).
4. The user can add, edit, or rearrange widgets.
5. Clicking “Save Dashboard” sends the updated configuration to the backend.
6. On reload, the saved configuration is loaded automatically.

7. The user can log out, which clears access to protected routes.

Optional / Bonus Features

If time permits, candidates can add:

- Drag-and-drop reordering (for example, with a library like React Beautiful DnD)
- Resizable widgets
- Local caching for offline mode
- Basic testing for critical frontend or backend functionality

---

Deliverables

Candidates should submit:

1. Source code for both frontend and backend (in one repository or two separate folders)
2. Clear setup instructions explaining how to:
    - Install dependencies
    - Start both frontend and backend servers
    - Configure environment variables if required
3. A short README describing:
    - Tools, frameworks, and libraries used
    - Authentication method chosen (cookie-based or token-based)
    - Architecture decisions and trade-offs
    - Known limitations
4. **Optional - In addition to above, deploy the app and share a link for us to preview.**

Evaluation Criteria

Category	Description	Weight
Frontend Architecture	Component structure, clear state management	30%
Interactivity	Page load, adding, removing, editing widgets	30%
UI & UX	Usability, layout, and responsiveness	20%
Code Quality	Clarity, organization, and maintainability	20%