### ****ThingsBoard Dashboard System Overview****

The **ThingsBoard Dashboard System** is a key feature of the ThingsBoard platform, designed to provide real-time monitoring and visualization for Internet of Things (IoT) devices. The system allows users to create customizable, interactive dashboards to display device data, telemetry, and control various aspects of IoT devices in a visual manner.

**Dashboards**

* **Definition:** Dashboards are collections of widgets (charts, gauges, maps, tables) that visualize IoT data in a user-friendly interface.
* **Purpose:** Provides insights into device status, performance, or other metrics in real-time.
* **Features:**
  + **Multiple Dashboards**: You can create multiple dashboards for different purposes or different groups of users.
  + **Customizable Layouts**: Drag-and-drop layout design, resizing, and widget organization.
  + **Interactive Widgets**: Allows for interaction with data (e.g., clicking on a device to show detailed information).
  + **Multi-tenant Support**: Dashboards can be customized and segmented for different tenants or user groups.

**2️⃣ Widgets**

* **Definition:** A widget is a component that displays a specific piece of information. This could be a chart, gauge, table, map, or any other UI element that renders data dynamically.
* **Widget Types**:
  + **Charts**: Bar charts, line charts, pie charts, etc., to visualize time-series data.
  + **Gauges**: Show values like temperature, humidity, or pressure in a circular dial.
  + **Maps**: Display device locations on a geographical map (with geospatial data).
  + **Tables**: Display a tabular list of data, useful for device status or logs.
  + **Indicators**: Used to show on/off statuses or other boolean data.

**3️⃣ Widget Bundles**

* **Definition:** Widget Bundles are groups of related widgets packaged together. They define how widgets are rendered, configured, and styled.
* **Purpose:** Widget Bundles allow you to package multiple widgets with preset configurations for easy reuse and sharing across dashboards.

**4️⃣ Templates**

* **Definition:** Templates are predefined widget configurations that allow for quick dashboard creation and consistency in design.
* **Use Case:** If you have similar dashboards across devices or tenants, templates can be used to apply the same configuration (e.g., widget layout, theme, widget type) across multiple dashboards.

**5️⃣ Themes and Layouts**

* **Themes:** Themes define the overall look and feel of the dashboard, including colors, fonts, and styling.
* **Layouts:** Layouts allow users to position and resize widgets on the dashboard. It supports **drag-and-drop** functionality to easily customize the dashboard.

**6️⃣ Real-Time Data & Telemetry**

* **Telemetry Data:** Dashboards can visualize telemetry data, which includes real-time or historical data sent by IoT devices. ThingsBoard provides a streaming mechanism for sending this data.
* **WebSocket Streaming:** Real-time data is pushed from the server to the dashboard using WebSockets, ensuring that data is updated instantaneously without the need for refreshing.

**7️⃣ Device Control**

* **Remote Device Control:** Dashboards can allow users to send commands to devices directly through the dashboard interface. For example, you can turn on/off a device, set parameters, or trigger specific actions based on the data shown in the dashboard.
* **Device and Asset Management:** Besides viewing the data, dashboards can also interact with devices (e.g., changing their configurations or settings).

**8️⃣ User & Role-Based Access**

* **User Roles:** ThingsBoard supports role-based access control (RBAC), allowing you to assign different permissions to users based on their role (e.g., Admin, User, Viewer).
* **Multi-Tenant Support:** Different dashboards can be made available to different tenants or user groups.

**9️⃣ API Integration**

* **REST APIs**: ThingsBoard provides REST APIs that allow users to programmatically interact with dashboards, widgets, and device data. This means you can automate dashboard creation, modify widget configurations, or fetch telemetry data using the ThingsBoard backend.
* **Widget API Integration**: Custom widgets can be integrated into dashboards using APIs.

**Dashboard System Workflow:**

1. **Create a Dashboard**: Users can create a new dashboard from the ThingsBoard UI and begin adding widgets.
2. **Customize Widgets**: Widgets are added to the dashboard and configured to show specific device data or telemetry.
3. **Apply Layouts**: Widgets can be arranged in grids, resized, or set with templates to create the dashboard layout.
4. **Monitor Devices**: Once the dashboard is live, real-time data is streamed into the widgets, providing users with live device status and telemetry.
5. **User Interaction**: Users can interact with the dashboard by clicking on devices, opening detailed views, or controlling devices remotely.

**Dashboard System Features:**

* **Drag-and-Drop Layouts**: Rearrange widgets easily.
* **Customizable Widget Settings**: Change chart type, set thresholds, and apply custom styles.
* **Real-Time Data Streaming**: WebSocket-based streaming of telemetry data.
* **Role-Based Access Control (RBAC)**: Different users have different access to dashboards.
* **Multiple Dashboards**: Support for multiple dashboards per tenant/user.
* **Reusable Templates & Widgets**: Easy reusability of widget bundles and templates.
* **Advanced Visualizations**: Support for advanced chart types like heatmaps, time-series graphs, and histograms.

**Use Cases for ThingsBoard Dashboard System:**

1. **Smart City Monitoring**: Visualize and control IoT devices such as smart streetlights, cameras, and sensors across a city.
2. **Industrial IoT (IIoT)**: Monitor industrial equipment like pumps, machines, and motors in factories, along with real-time telemetry.
3. **Energy Management**: Manage energy consumption, track performance of renewable energy sources (e.g., solar, wind), and visualize energy metrics.
4. **Fleet Management**: Monitor vehicle telemetry, such as location, speed, and fuel consumption in real-time.

**How to Integrate Custom Dashboards**

If you need custom dashboards or widgets, you can:

* **Extend ThingsBoard** using the **Widgets Library**.
* **Create Custom Widgets** with **React** or **Angular**.
* **Programmatically Create Dashboards** using ThingsBoard **REST APIs**.

If you're looking to **customize the dashboard system further** or integrate it with your own platform, let me know, and I can help with specific use cases or code examples!

**Project Structure**

You will create **two separate React projects**:

1. **Dashboard Web App** - The main application that hosts the dashboard UI.
2. **Widget Library** - A separate React component library containing reusable widgets, which can be plugged into the dashboard.

**Create the Widget Library as a Separate React Component Library**

This will be a standalone React library that exports various widgets.

**Step 1: Create the Widget Library Project**

Run the following command to create a new React component library using **Vite**:

sh

CopyEdit

npx create-vite widget-library --template react-ts

Go inside the project:

sh

CopyEdit

cd widget-library

Initialize it as a library:

sh

CopyEdit

npm init -y

Update package.json to make it a library:

json

CopyEdit

{

"name": "widget-library",

"version": "1.0.0",

"main": "dist/index.js",

"module": "dist/index.esm.js",

"types": "dist/index.d.ts",

"files": ["dist"],

"scripts": {

"build": "vite build",

"prepublishOnly": "npm run build"

},

"peerDependencies": {

"react": "^18.0.0",

"react-dom": "^18.0.0"

}

}

**Step 2: Create a Widget Component**

Inside src/components, create a sample widget component:

📂 widget-library/src/components/ChartWidget.tsx

tsx

CopyEdit

import React from "react";

import { LineChart, Line, XAxis, YAxis, Tooltip, ResponsiveContainer } from "recharts";

const ChartWidget = ({ data }: { data: any[] }) => {

return (

<ResponsiveContainer width="100%" height={300}>

<LineChart data={data}>

<XAxis dataKey="time" />

<YAxis />

<Tooltip />

<Line type="monotone" dataKey="value" stroke="#8884d8" />

</LineChart>

</ResponsiveContainer>

);

};

export default ChartWidget;

**Step 3: Export Components from Library**

Create src/index.ts to export all components:

📂 widget-library/src/index.ts

tsx

CopyEdit

export { default as ChartWidget } from "./components/ChartWidget";

**Step 4: Build and Publish the Library**

Run:

sh

CopyEdit

npm run build

If you want to use it locally, link the package:

sh

CopyEdit

npm link

Alternatively, you can publish it to **npm**:

sh

CopyEdit

npm publish --access public

**2️⃣ Create the Dashboard Web App**

This is the main React project where the widget library will be used.

**Step 1: Create the Dashboard Web App**

Run:

sh

CopyEdit

npx create-vite dashboard-app --template react-ts

cd dashboard-app

**Step 2: Install the Widget Library**

If using a local package:

sh

CopyEdit

npm link widget-library

If published on npm:

sh

CopyEdit

npm install widget-library

**Step 3: Use Widgets in the Dashboard**

Modify src/components/Dashboard.tsx:

📂 dashboard-app/src/components/Dashboard.tsx

tsx

CopyEdit

import React from "react";

import { ChartWidget } from "widget-library";

const Dashboard = () => {

const data = [

{ time: "10:00", value: 50 },

{ time: "10:05", value: 55 },

{ time: "10:10", value: 60 },

];

return (

<div>

<h1>Dashboard</h1>

<ChartWidget data={data} />

</div>

);

};

export default Dashboard;

**3️⃣ Running the Projects**

Start the widget library (if using local linking):

sh

CopyEdit

cd widget-library

npm run build

npm link

Start the dashboard app:

sh

CopyEdit

cd ../dashboard-app

npm run dev

Now, the dashboard app will dynamically import widgets from the **widget library**!

**✅ Benefits of This Approach**

* **Separation of Concerns**: The widget library is independent and can be reused in other projects.
* **Versioning & Updates**: The widget library can be versioned separately and updated in the dashboard.
* **Plug-and-Play Architecture**: New widgets can be added to the widget library without modifying the dashboard app.

Would you like **drag-and-drop functionality** or **dynamic widget loading** next?

In the ThingsBoard platform, the **Dashboard Manager** and related concepts such as **configuration**, **themes**, **layouts**, **access control**, and **widget library integration** form the backbone of the dashboard system. Here's an in-depth explanation of each concept:

**1. Dashboard Manager**

The **Dashboard Manager** is responsible for managing the creation, modification, and deletion of dashboards in ThingsBoard. It handles all the necessary interactions related to dashboards, including:

* **Dashboard Creation**: Allows users to create new dashboards, define widgets, configure settings, and assign layouts.
* **Dashboard Retrieval**: Provides access to dashboards via API calls or UI, allowing for dynamic data loading.
* **Dashboard Modification**: Users can edit dashboards, add new widgets, change settings, and update the layout in real time.
* **Dashboard Deletion**: It supports the removal of dashboards from the system.
* **State Management**: Ensures that the dashboard's state (such as widget configurations) is persistently stored and can be reloaded upon request.

**2. Dashboard Configuration**

The **dashboard configuration** refers to the settings that define the layout and structure of a dashboard. This includes:

* **Widget Configuration**: Specifies the types, data sources, appearance, and behavior of widgets (charts, tables, maps, etc.).
* **Data Source Configuration**: Determines the data sources that widgets will use (e.g., devices, data points, APIs, etc.).
* **Time Configuration**: Defines the time window for data display, including real-time or historical data.
* **Interaction Configuration**: Allows for specifying how users can interact with the dashboard, such as adding/removing widgets, resizing widgets, or configuring widget parameters.

**3. Themes**

**Themes** control the visual appearance of dashboards and widgets. A theme typically includes the following elements:

* **Colors**: Background colors, text colors, border colors, and widget-specific color schemes.
* **Font Styles**: The choice of font family, size, weight, and style across the dashboard.
* **UI Elements**: The design of buttons, tooltips, headers, and other user interface elements to maintain consistency and improve user experience.
* **Predefined Themes**: ThingsBoard supports several predefined themes (e.g., light, dark) that can be applied globally or individually per dashboard or widget.

Custom themes can be created to match the specific branding requirements of an organization or project.

**4. Layouts**

The **dashboard layout** defines how the dashboard is structured and how widgets are positioned within it. Key aspects of the layout include:

* **Grid-based Layout**: Widgets are arranged in a grid with defined row and column positions, allowing for flexible and dynamic resizing.
* **Responsive Layouts**: Dashboards are responsive to different screen sizes and device types (e.g., mobile, tablet, desktop). Layouts adapt based on the device's screen size.
* **Customizable Layouts**: Users can modify the layout by dragging and dropping widgets, adjusting their sizes, and organizing the dashboard content to suit their needs.
* **Predefined Layout Templates**: Users can select from predefined templates or create custom layouts to fit specific use cases.

**5. Access Control**

**Access Control** in ThingsBoard ensures that only authorized users can access and modify dashboards. It consists of:

* **User Roles**: ThingsBoard supports role-based access control (RBAC), where users can be assigned specific roles that grant access to dashboards or widgets based on their permissions.
* **Permissions**: Specific permissions can be granted to users or groups for viewing, editing, or managing dashboards. Permissions can be defined at both the **dashboard** level and the **widget** level.
* **Secure Data**: Sensitive information displayed in dashboards can be restricted based on user access, ensuring that only authorized users can view specific data (e.g., device data or historical trends).

**6. Widget Library Integration**

The **Widget Library** is a collection of prebuilt widgets (like charts, tables, maps, etc.) that can be used within dashboards. The integration of the widget library into the ThingsBoard system includes:

* **Widget Types**: A variety of widget types are available for different use cases, such as time-series charts, gauges, tables, and maps. These widgets come with their own configurations for data sources, appearance, and behavior.
* **Custom Widgets**: Users can create custom widgets using the **widget SDK**, which can then be added to the widget library. This allows users to extend ThingsBoard's functionality.
* **Widget Bundles**: Widget bundles are groups of related widgets packaged together. For example, a "sensor bundle" might include widgets for displaying sensor readings, graphs, and alarms. Widget bundles can be reused across multiple dashboards.
* **Widget Configurator**: This tool helps users to configure widgets and define their settings, including data sources, appearance, and interaction options.
* **Dynamic Widget Loading**: The widget library can be dynamically loaded based on the dashboard settings and configuration, enabling flexibility in dashboard design and usage.

**Diagram Representation**

Here’s a summary of the architecture involving the Dashboard Manager, Configuration, Themes, Layouts, Access Control, and Widget Library Integration in ThingsBoard:

plantuml

CopyEdit

@startuml

actor User

participant "Dashboard Manager" as DM

participant "Widget Library" as WL

participant "Theme Configurator" as TC

participant "Layout Configurator" as LC

participant "Access Control" as AC

participant "Dashboard Service" as DS

User -> DM: Request Dashboard

DM -> WL: Load Widgets

DM -> TC: Apply Theme

DM -> LC: Apply Layout

DM -> AC: Check Access

AC -> DM: Access Granted/Denied

DM -> DS: Fetch Dashboard Configuration

DS -> DM: Return Dashboard Configuration

DM -> User: Display Dashboard

@enduml

**Conclusion**

* **Dashboard Manager** acts as the controller for creating, managing, and deleting dashboards.
* **Configuration** allows you to define how the widgets behave and interact with data.
* **Themes** control the aesthetic aspects of the dashboard, ensuring a consistent user experience.
* **Layouts** determine the arrangement of widgets within the dashboard, making it flexible and responsive.
* **Access Control** ensures that dashboards are only accessible to authorized users based on roles and permissions.
* **Widget Library Integration** allows users to choose, configure, and customize widgets, providing flexibility in dashboard design.

Each of these components works together to ensure a dynamic, secure, and customizable dashboarding experience in ThingsBoard. Would you like more details on any specific component or need assistance with something else?

User interaction, Sequence diagram for Client layer UI - Dashboard manager, configuration, themes, layout, widget manager, container, widget groups, types, datahandler, caching UI, api management

###### ChatGPT said:

Here’s a **sequence diagram** that illustrates the **User Interaction** with the **Client Layer (UI)** in the dashboard system. The interactions include:

1. **Dashboard Manager** – Handles dashboard CRUD operations.
2. **Configuration** – Manages dashboard settings and properties.
3. **Themes & Layouts** – Handles UI appearance and arrangement.
4. **Widget Manager** – Manages widget creation, updates, and deletion.
5. **Widget Container** – Loads and organizes widgets.
6. **Widget Groups & Types** – Categorizes widgets into groups and types.
7. **Data Handler** – Fetches and caches data for widgets.
8. **Caching (UI)** – Stores widget/dashboard data to reduce API calls.
9. **API Management** – Interfaces with backend services.

### ****PlantUML Sequence Diagram****

plantuml

CopyEdit

@startuml

actor User

participant "Dashboard UI" as UI

participant "Dashboard Manager" as DashboardManager

participant "Configuration" as Config

participant "Themes & Layout" as ThemesLayout

participant "Widget Manager" as WidgetManager

participant "Widget Container" as WidgetContainer

participant "Widget Groups" as WidgetGroups

participant "Widget Types" as WidgetTypes

participant "Data Handler" as DataHandler

participant "UI Caching" as UICache

participant "API Management" as APIManager

User -> UI: Open Dashboard

UI -> DashboardManager: Load Dashboard

DashboardManager -> Config: Fetch Dashboard Config

Config -> ThemesLayout: Apply Themes & Layout

ThemesLayout -> UI: Update Dashboard UI

User -> UI: Add Widget

UI -> WidgetManager: Create New Widget

WidgetManager -> WidgetTypes: Select Widget Type

WidgetTypes -> WidgetGroups: Assign to Group

WidgetManager -> WidgetContainer: Add to Container

WidgetContainer -> UI: Render Widget

User -> UI: Fetch Widget Data

UI -> DataHandler: Request Widget Data

DataHandler -> UICache: Check Cached Data

alt Cache Hit

UICache -> DataHandler: Return Cached Data

else Cache Miss

DataHandler -> APIManager: Request Data from Backend

APIManager -> DataHandler: Return API Response

DataHandler -> UICache: Cache Data

end

DataHandler -> UI: Return Data

User -> UI: Change Theme/Layout

UI -> ThemesLayout: Update Theme/Layout

ThemesLayout -> UI: Apply Changes

User -> UI: Save Dashboard Config

UI -> Config: Update Dashboard Config

Config -> APIManager: Save Config to Backend

APIManager -> Config: Confirm Update

Config -> UI: Confirm Save Success

@enduml

### ****Explanation of the User Interaction Flow****

1. **Opening Dashboard**:
   * The user opens the dashboard.
   * The UI requests the **Dashboard Manager** to load the dashboard.
   * The **Configuration** service fetches the dashboard settings.
   * The **Themes & Layout** module applies the dashboard’s appearance.
2. **Adding a Widget**:
   * The user selects "Add Widget."
   * The **Widget Manager** creates the widget.
   * The **Widget Types** component helps define the widget.
   * The **Widget Groups** module assigns it to a category.
   * The widget is added to the **Widget Container** and rendered.
3. **Fetching Widget Data**:
   * The user requests data for a widget.
   * The **Data Handler** checks if the data is cached.
   * If cached, it is returned instantly.
   * If not cached, the **API Manager** fetches data from the backend.
   * The new data is cached and returned to the UI.
4. **Changing Theme/Layout**:
   * The user updates the dashboard theme/layout.
   * The **Themes & Layout** module applies the change.
5. **Saving Dashboard Configuration**:
   * The user saves the updated dashboard settings.
   * The **Configuration** service sends the update to the backend.
   * The **API Manager** confirms the update.

### ****Key Takeaways****

* **Optimized Caching**: Reduces redundant API calls, improving performance.
* **Separation of Concerns**: Each component handles a specific role.
* **Efficient Data Handling**: Uses a **Data Handler** to streamline widget updates.
* **Scalability**: Can be expanded to include real-time updates, user permissions, etc.