New INDIGO IAM Dashboard

Jacopo Gasparetto
INFN

ICSC
Centro Nazionale di Ricerca in HPC,
Big Data and Quantum Computing

INDIGO IAM Technical Hackathon @ CERN
10-13 February 2025

INFN
CNAF

Outline

- New Dashboard
  - Motivation
  - Introduction
  - Architecture
  - Technologies
  - Authentication/Authorization
  - Development Status
  - Live Demo
- Introduction to End-to-End testing
  - Live Demo (?)
- Introduction to telemetry
  - Key concepts
  - Instrumentation
  - Deployment strategies
  - Live Demo

New IAM Dashboard

Motivation

- Current dashboard is based on JavaServer Pages (JSP) and Angular.js which is in EOL since January 2022
- Drop deprecated libraries in order to increase security (mandatory to use IAM on Italian projects)
- Decouple the frontend logic from the login service (backend)
- Modern web development
- Lightweight and responsive
- Customization (anyone can fork and extended/modify the dashboard with for needs)

IAM Dashboard: a Next.js/React web application

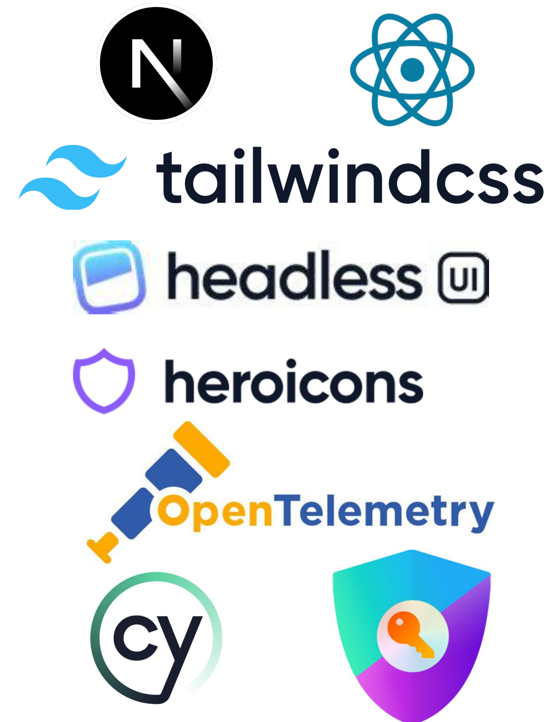
- The new IAM Dashboard is a web application written in TypeScript on top of the Next.js framework, the official framework indicated by the React developers.
- React vs Next.js
  - React is a *library* that allows the development of reusable web “reactive components”. It embraces the declarative paradigm: the user interacts with the dashboard changing some states, and the components react (update) accordingly to the new state
  - Next.js is a *framework* with a huge set of features. Among these, Next.js allows the development of a *Server Side Rendering* based web server with a *Backend For Frontend* (BFF) model.
- In other words, the new IAM Dashboard is a Next.js application whose components are written in React.
- In this context, the new IAM Dashboard is both a Node.js web server which servers static and dynamically rendered content (compiled html, js and css) AND a real API that the browser interacts with.

https://nextjs.org/docs

Technologies

The dashboard relies on few third party libraries chosen because of their popularity, support and consistency

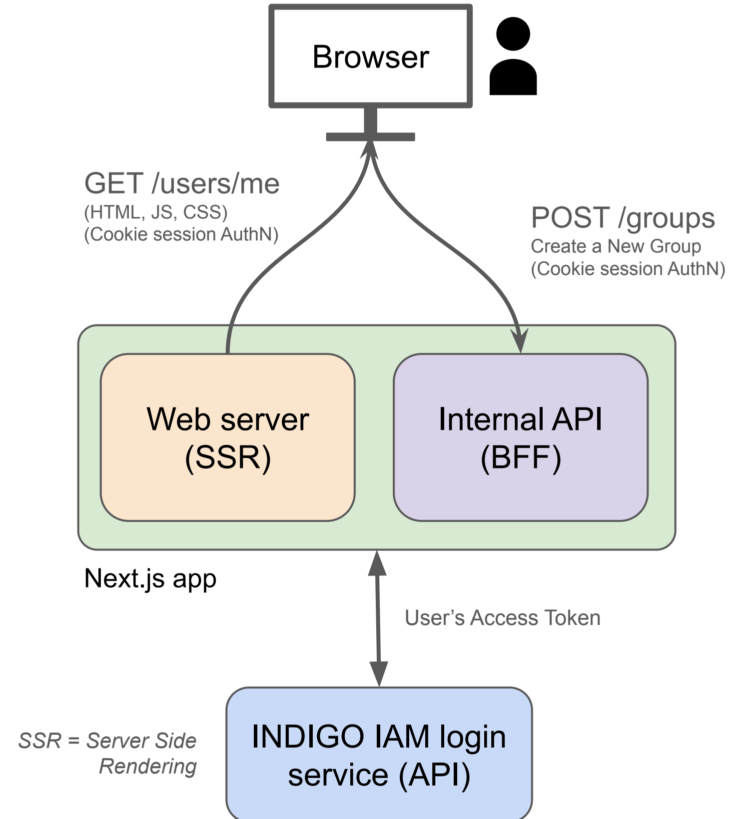
Tool	Description
Next.js v15	Server Side Rendering Framework
React v19	UI components
Tailwind CSS v5	CSS and styling
Heroicons v2 (by Tailwind)	Icons
Headless UI v2 (by Tailwind)	Unstylized component library for React
Auth.js v5	OIDC/OAuth2 library for web applications
OpenTelemetry	Telemetry (metrics, traces)
Cypress v14 (TDB)	End to end testing framework



General Architecture: Backend For Frontend (BFF)

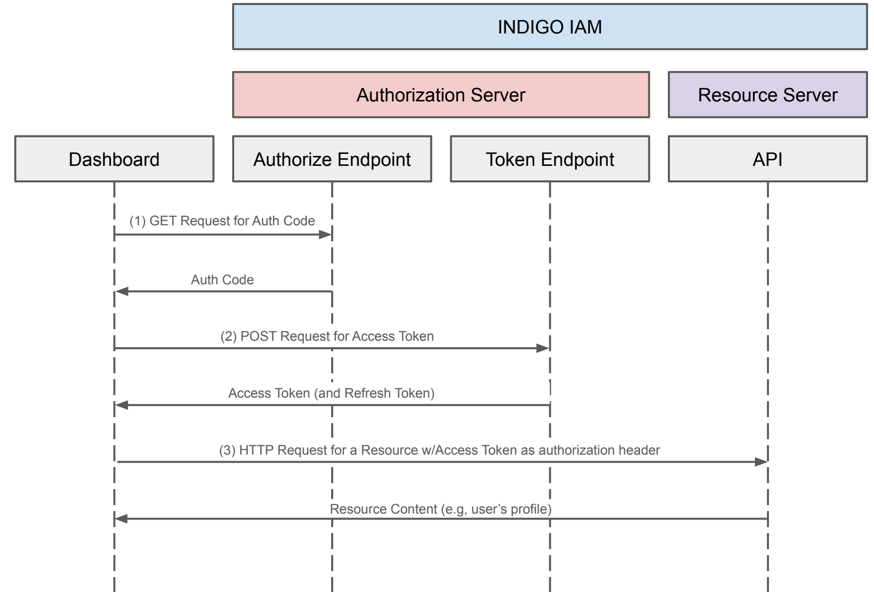
1. When the user asks for a page, the **BFF queries the IAM login service** API using the user's Access Token (AT)
2. **IAM login service** API **returns** the JSON payload **to Next.js**
3. **Next.js renders the page** with the data received by IAM login service and **returns it to the user's browser**

- The browser **NEVER connects** directly **to IAM login service**.
- Each request from the browser to the IAM login service API is proxied by the BFF which injects the user's AT as HTTP authentication header
- **The distinction** between Web server and BFF is purely logical and **is completely transparent to developer**



Authentication/Authorization

- **The dashboard (BFF) is a** IAM registered OIDC/OAuth2 **client** with **Authorization Code Flow**
- **The dashboard asks** IAM login service for **an access token** in behalf of the user
- The dashboard **associates the AT to the user** and **stores it somewhere**
- The dashboard (**BFF**) **uses the AT** to **perform requests to the** protected IAM login service **APIs**



OAuth2 Authorization Code flow (PKCE is not shown in figure)

In the OAuth2 language, IAM login service plays both the roles of an Authorization Server (AT issuer) and a Resource Server (who owns the protected resources, i.e, APIs)

Authentication/Authorization

- IAM issues a token that the dashboard sends back to query the protected resources (IAM APIs)
- The dashboard needs to use a privileged IAM client credential
    - E.g., `iam:admin.write/read` and `scim:write/read`
- Authorization (Admin vs User) is capability/scope based
- Admin/scim scopes are filtered by IAM login service policies for non admin users (since v1.11.0)
- All operations of non admin users rely on Me endpoints

Authentication/Authorization: how users log in

- Authentication via OpenID/OAuth2 is provided by the Auth.js library
- When the user lands on the dashboard, the application looks for a session cookie. If the cookie is missing or expired, users are redirected to the IAM login service `/authorize` endpoint passing parameters such as the redirect uri (the dashboard endpoint), required scopes and client id
- Users log in within the IAM login service page as usual
- After authentication, users are redirected back to the dashboard, which completes the Authorization Flow receiving the access token and verifying it
- Then, the BFF sends a session cookie to the browser which sends it back to the BFF at each request. The cookie identifies the user with a session id and somehow we need to store the AT and associate it with that session id

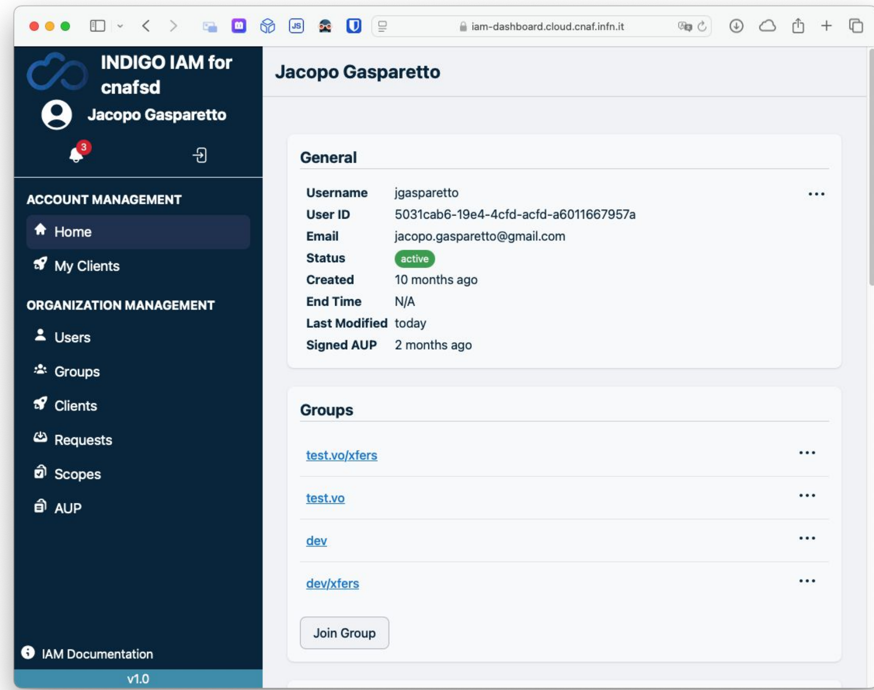
Authentication/Authorization: where do we store the token?

- Where to store the token and what to do with it depends on the developer
- There are two main approaches
  - Saving the Access Token *within* the (encrypted) cookie payload
  - Storing the Access Token into an external database
- Saving the AT inside the cookie is straightforward and almost cost-free. The token is actually stored on the user's browser local storage and sent back to the BFF for each request. **It is considered less safe because if the cookie is stolen, a bad actor can gain access to the resources.**
- Saving the AT on database is more secure because it is never exposed to the browser. On the other hand, **the complexity is much higher** (custom automations need to be implemented to clean the database of expired tokens while maintaining the association between session user ID and valid access token). In case of several replicas, the database must be synchronized among all replicas.

Auth.js official docs highly discourage to store Refresh Tokens within the session cookie but considers the storage of the Access Token acceptable. For a better overview about the session strategies consult the official guide.

Development Status

- Most of the current features have been successfully implemented
- Final User Interface and User Experience (UI/UX) not yet defined
- Changes to improve UI/UX are still under investigation
- Code review needed
- End to end tests are being written to increase coverage
- We hope to receive feedback from the users! Starting from this Hackathon :)



Live Demo

End to end testing

How to test a web application? End to end testing

- In the context of web applications, end-to-end (E2E) testing translates to simulating the human interaction with the interface, such as mouse clicks, typing, scrolls, etc. The definition “end-to-end” means that the *entire* chain of services composing the application is tested:
  - when a button is clicked, the event is sent down to the BFF, then to the API, the database and back to the web application that displays the result.
- We need to *robotize* the browser so that it performs a given set of instructions that mimic the human interactions
- Tests must be executed in a Continuous Integration (CI) pipeline
- Several frameworks exist, such as Cypress and Playwright
- We chose Cypress but we are still open to try Playwright as alternative

Example of basic E2E test

- Log in using a predefined set of credentials
- Click “Add new group”, a popup opens
- Type the group name
- Click “Add group” button
- In the search bar, type the name of just created group
- Click “delete group”

```
describe("groups", () ⇒ {
  beforeEach(() ⇒ {
    cy.loginWithIam(
      Cypress.env("IAM_ADMIN_USER"),
      Cypress.env("IAM_ADMIN_PASSWD")
    );
  });

  it("create group", () ⇒ {
    // create a new group
    cy.visit("/groups");
    cy.get("[data-test=add-group").click();
    cy.get("[data-test=modal").within(() ⇒ {
      cy.get("input[type=text]").type("test-bot-group-1");
      cy.get("button[type=submit]").click();
    });
    cy.log("group created");
    // delete group
    cy.wait(100);
    cy.get("[data-test=search-group").type("test-bot-group-1");
    cy.get("[data-test=option]").should("have.length", 1);
    cy.get("[data-test=option]").click();
    cy.get("[data-test=delete]").click();
    cy.get("[data-test=modal]").within(() ⇒ {
      cy.get("button[type='submit']").click();
    });
    cy.log("group deleted");
  });
});
```

Example of E2E test made with Cypress

Introduction to telemetry

Introduction to telemetry: OpenTelemetry

From the official documentation

> *OpenTelemetry is:*
> - ***An observability framework** and toolkit designed **to create** and manage telemetry data such as **traces**, **metrics**, and **logs**.*
> - ***Not an observability backend** like Jaeger, Prometheus, or other commercial vendors.*
> - *…*

OpenTelemetry is collection of components that provide an SDK (Software Development Kit) to enrich an existing code base to *emit* telemetry data.

OpenTelemetry **does not** store any data *per se* and some kind of database(s) is(are) needed

Telemetry data: Traces, Metrics and Logs

Telemetry data is generally grouped in three main entities, or *signals*: traces, metrics and logs.

- **traces**: represent **the path of a request through an application**. They are made up by *spans* that represent the single units of work or operation. Each span has **start and end timestamps**, a **name**, a **parent id** (if child of another span) and **attributes**. A typical trace/span information answers "*how much time this function took to execute?*" and "*what is the runtime call stack at this endpoint?*"
- **metrics**: represent the **measurement emitted by a meter**. Meters can typically be monotonic/non-monotonic counters, histograms and gauges. In the physical world "3 kWh at 2025-02-10 12:00" is the metric produced by the electricity meter (monotonic) of our house. An example could be: "how many requests we received so far at each endpoint?"
- **logs**: timestamped text record, either structured or unstructured, with optional metadata. We do not deal with telemetry logs, as they could easily be written with a simple access/error logger

OpenTelemetry: Instrumentation

- The process of *augmenting* our codebase to emit telemetry data is called **instrumentation**. This means that we are adding code to our existing code.
- OpenTelemetry has SDKs for basically every language on the market.
- For some high level languages, such has JavaScript/TypeScript, OpenTelemetry offers the so called Zero-code Instrumentation feature that enables telemetry with just a couple of lines of configuration.
- For other languages such as C++, there is no automatic instrumentation and the codebase must be manually instrumented at each function of interest and it is up to the developer to choose what they want to monitor.

Instrumenting a Next.js application

- Next.js offers a modified package for OpenTelemetry which does the entire magic.
- It enables basic telemetry for traces only.
- It is still possible to have a finer grained control about where to put spans and metrics in your codebase

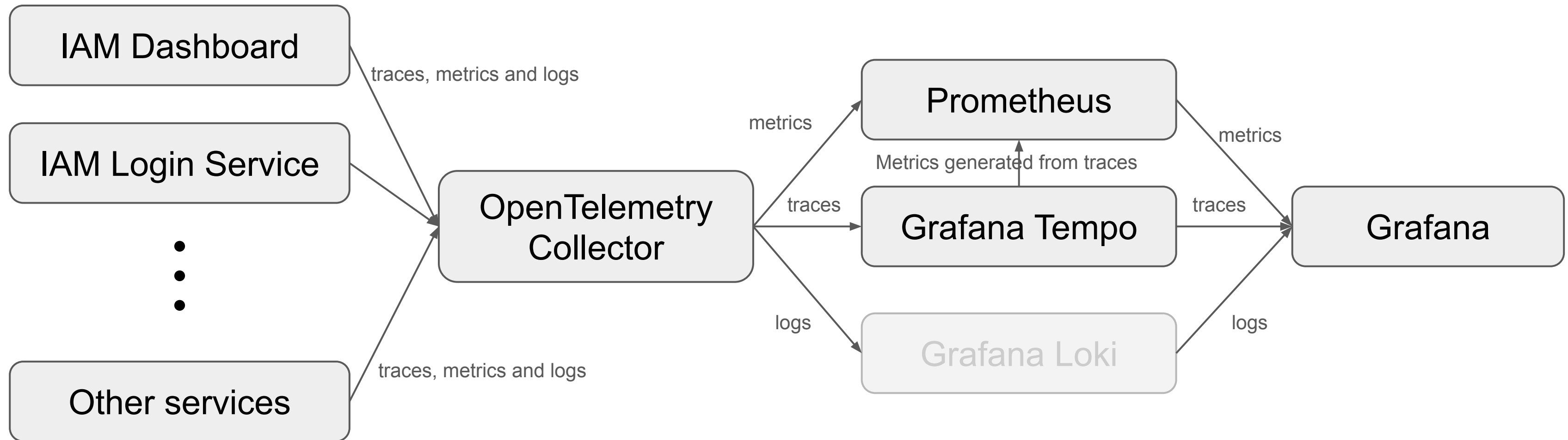
```
import { registerOTel } from '@vercel/otel'

export function register() {
  registerOTel({ serviceName: 'iam-dashboard' })
}
```

Collect telemetry data

- An application can be made up of several components (micro-services), each sending telemetry data from different endpoint but still logically representing the same application. For example, a trace with its root span can be opened by a NGINX reverse proxy that proxies the request to an API. It is important to reconstruct those traces to understand the complete path of the request.
- An application can have manyfold replicas, each sending telemetry data.
- Due to the nature of the data, different signals (traces, metrics and logs) must be stored on different kind of databases. They can also be sent through different protocols (http, gRPC, kafka, ect.).
- OpenTelemetry offers a service called OpenTelemetry Collector which receives, processes and then exports telemetry data to the appropriate backends.

Collect telemetry data: case study

- For our case study, we choose Prometheus as metrics backend and Grafana Tempo as traces backend
- Grafana is then used as web application to visualize KPIs, stats and plots
- A particular useful feature of Grafana Tempo is its built-in metrics generator which produces new metrics from the traces, such as the total count of requests and the latency histogram



Live Demo

Conclusion

- The core architecture has been successfully built
- Results look promising and we can close the “proof of concept” phase
- Many UI/UX decision still need to been taken yet (we would love to receive feedback from users)
- Several minor bugs have to be fixed
- Telemetry has been proven to be a great tool to inspect, monitor and debug IAM dashboard and we would like to implement it also in IAM login service codebase