WanderJaunt

Email: hi@wanderjaunt.com
Sitio web: www.wanderjaunt.com

Custom Fields Application

Aníbal Rodríguez, Software Engineer
Email: arodriguez@wanderjaunt.com

WanderJaunt

Index

Summary

WanderJaunt relies his entire operation on several employees around the country, whose communicate between them in digital ways, and is the software engineer's responsibility to provide those necessary tools to manage the day by day.

This document is a specification of one of the company pains; to save unstructured data in our databases, the need is to count with a web application with dynamic formularies. With dynamic we refer to different input types like checkbox, only text, images, etc.

The challenge in this project, is to achieve dynamic formularies, we need to save data without structures and that is a problem when we work with relational databases, and that is our case. On the hand, do exists technical options like to save data in an unstructured way in a structure scheme. On the hand, we need to resolve scalability and resilience too.

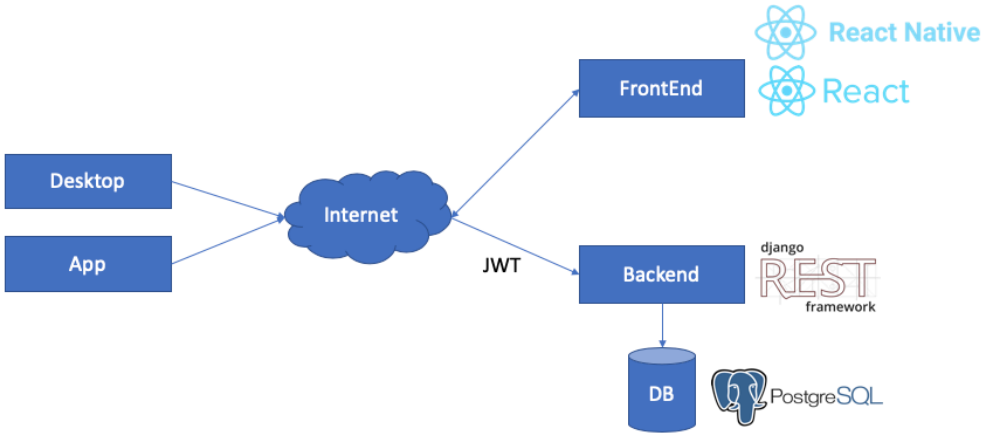
Scope

This App has the need to be scalable and resilient, and due to this, we will build an isolated app with the only responsibility of add, view, update and delete formularies created in the same App.

The entire project will have a version for modern browsers and another for Mobile Applications (Android, iOS), besides it will have a backend API with JWT for HTTP requests, and with that stack we can put this project in any cloud infrastructure with access to our database cluster for scalability. That said, is important to mention that with this implementation, we can deploy our frontend and backend separately improving our times of maintenance.

Ecosystem

The following diagram shows a simple review about the product ecosystem. The actors of system, our frontend and backend stack, and our security method for the API.



Data Management

The custom fields, represent a problem for a relational database like ours. So, we will implement AVRO JSON Schemas (https://avro.apache.org/docs/current/spec.html) to save unstructured data. We will save type of forms in the frontend, that will save the data structured by Input type, and we will can save input templates. But, this is the first

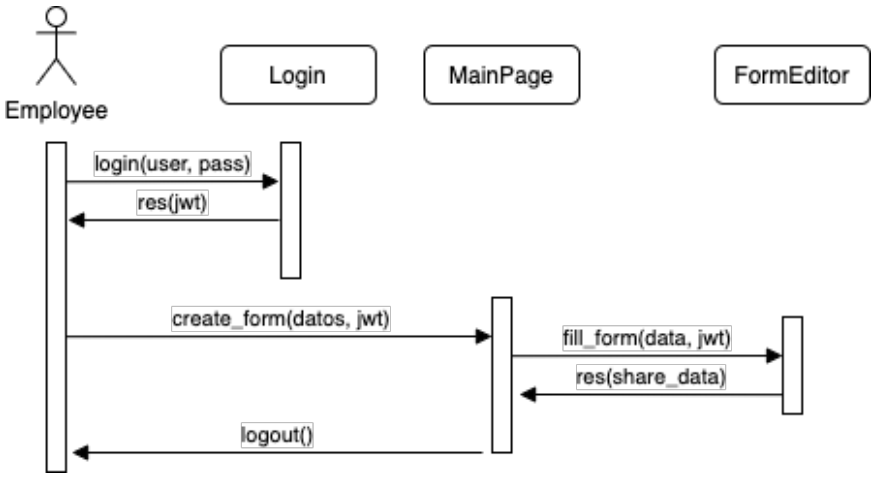
version of the software, so, we will upload the input templates in the database by query, to avoid to lose time on that, and deploy our product the fast as possible.

UI design

The frontend apps (ReactJS and RN) will have the same flow, with different design, but barely different indeed. The app will have a login, and menus for esdmain page, my forms, and an editor for CRUD actions for forms.
The features by menu are:

- Login:
  - We can authenticate and authorize users depending of their credentials
- Main Page:
  - See my last 5 formularies created
  - The most answered formularies
- My Forms
  - To activate/deactivate forms
  - To view the answers of my formularies
  - To answer formularies of another employees that I can see
- Editor:
  - We can create new forms
  - We can continue editing our last form
  - We can add template input schemes for our forms
  - We can set group role view permission

The following sequence diagram describes a flow when we want to create a form.



Backend Design

We will use Django Rest Framework for the API, because is our main tool for backend artifacts. We can host our API on Heroku and set an autoscaling policy.

Security

Use JWT for securing APPs is a strategy when you want to highly scale them. Is easy to implement against OAUTH2, that is more secure. Besides, on payload we can send user data, and when we use HTTPS, the user information is encrypted.

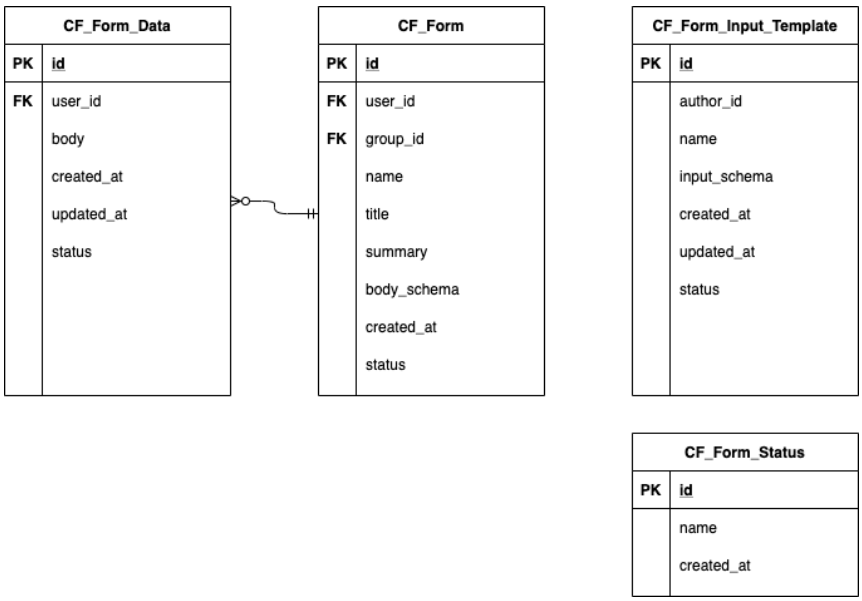
Database

Our database is a clustered PostgreSQL. And we will use JSON types for the AVRO Schema data and types.

Modeling

Database Model

The following entities need to be created for this product. We will continue using our user roles and groups for user references, and they are titled with a “CF” first from Custom Fields denotation.



Data Dictionary

Here we have the fields of the database tables:

1. **CF_Form_Data**:
    a. id: registry id
    b. user_id: user PK
    c. body: the form answered saved in AVRO Schema type
    d. created_at: when the form was answered
    e. updated_at: when the form was updated
    f. status: status of the form

2. **CF_Form**:
    a. Id: registry id
    b. user_id: user PK
    c. group_id: group PK for view permission
    d. name: name of the form
    e. title: title of the form
    f. summary: summary of the form
    g. body_schema: schema defining all the input schemas for the form
    h. created_at: when was created
    i. status: status id for the Form schema
3. CF_Form_Input_Template:
    a. id: registry id
    b. author_id: user PK
    c. name: name of the Input Template
    d. input_schema: schema of the input template
    e. created_at: when was created
    f. updated_at: when was updated
    g. status: status id for the input template
4. CF_Form_Status:
    a. id: registry id
    b. name: name of the status
    c. created_at: when was created

API

To describe our API, we will use OpenAPI Specification because of its completeness.
The entire yaml could be found here: https://github.com/anibal21/wanderjaunt/blob/main/spec_doc/customFieldsSwagger.yaml

The available endpoints are:

1. api/login:
    a. Description: endpoint to authenticate and authorize user
    b. Method: POST
    c. Body: credentials
    d. Response: JWT

2. api/form:
    a. Description: create a form
    b. Method: POST
    c. Body: the form schema
    d. Response: the forms available for me

3. api/form:
    a. Description: obtain my created forms and those that I can see
    b. Method: GET
    c. Response: the forms available for me

4. api/form/{id}:
    a. Description: obtain the form with id {id}
    b. Method: GET
    c. Response: the form with {id}

5. api/form/{id}:
    a. Description: update a form with id {id}
    b. Method: PUT
    c. Body: the updated data
    d. Response: a updated status

6. api/form/{id}:
    a. Description: delete a form with id {id}
    b. Method: DELETE
    c. Response: a delete status

7. api/input:
    a. Description: obtain all the input schemas
    b. Method: GET
    c. Response: a list of input schemas

8. api/form/data/{id}:
    a. Description: we use this to create a data instance of a form id {id}
    b. Method: POST
    c. Body: the form data
    d. Response: a response of the created form data

9. Api/form/data/{id}:
    a. Description: we use this to update a data instance of a form id {id}
    b. Method: PUT
    c. Body: the form data
    d. Response: a update status