# Contents

[1 Contents 1](#_Toc113004994)

[1 Functional specifications 2](#_Toc113004995)

[1.1 User management 2](#_Toc113004996)

[1.1.1 Super admin user: applicable for whole system: only one per system 2](#_Toc113004997)

[1.1.2 Admin users: applicable at tenant level: one per tenant 2](#_Toc113004998)

[1.1.3 Business users: applicable at branch level: one user can have many branches 2](#_Toc113004999)

[2 Technical specifications 2](#_Toc113005000)

[2.1 Database 2](#_Toc113005001)

[2.2 Server side 3](#_Toc113005002)

[2.2.1 GraphQL used for API 3](#_Toc113005003)

[2.2.2 App server: Python Flask used 4](#_Toc113005004)

[2.2.3 Logging 4](#_Toc113005005)

[2.2.4 Error handling at server 5](#_Toc113005006)

[2.3 Client side 5](#_Toc113005007)

[3 Tips and tricks 5](#_Toc113005008)

[3.1 Client side 5](#_Toc113005009)

[3.1.1 Material-ui 5](#_Toc113005010)

# Functional specifications

## User management

### Super admin user: applicable for whole system: only one per system

* Creates new tenant. This in turn creates new database schema in PostgreSQL internally. Inputs tenant code, tenant name. CRUD available. When deletes a tenant, the schema is not removed from database, you need to remove it manually
* Creates admin users for the tenants. One tenant can have only one admin user
* Username and password for super admin user is fixed by system. It can be changed only by changing the config file at server
* Does CRUD operations on “Control permissions”. There are some controls in user interface which can be made disabled based on user login. Super admin maintains the control master for such controls

### Admin users: applicable at tenant level: one per tenant

* Has all the permissions at tenant level
* Creates business users (CRUD operations)
* Can change its own username and password
* Creates permissions
* Associates permissions with business users
* Associates branches with business users

### Business users: applicable at branch level: one user can have many branches

* Controls are visible based on permissions set by admin user

# Technical specifications

## Database

* At app.cloudjiffy.com database created

{

"user":"webadmin",

"password":"in last email",

"host":"node113022-kater-db.cloudjiffy.net",

"port":"11301",

"database":""

}

* Database tables
* Backup strategy

## Server side

app.cloudjiffy.com cloud server used. User name, password is given to customer. App server, database etc. are used as docker containers. There are three app servers. 1) dev, 2) stage, 3) production named as kater-dev, kater-stage and kater respectively. All use the same database. There is a demo schema in the database for demo purposes

### GraphQL used for API

* Used Ariadne schema first framework with Flask
* Steps for using Ariadne with Flask
  + Use blueprint to keep all GraphQL stuff together
  + Create a .graphql file which defines all types. Use scaler type which is same as any in TypeScript
  + Complete stuff here. This is sample for query. Similarly, mutation is to be implemented. Key thing is keep all graphql stuff in one folder.

graphQlArtifacts = Blueprint('graphQlArtifacts', \_\_name\_\_)

@graphQlArtifacts.route('/graphql', methods=['GET'])

def graphql\_playground():

    return PLAYGROUND\_HTML, 200

@graphQlArtifacts.route("/graphql", methods=["POST"])

def graphql\_server():

    data = request.get\_json()

    success, result = graphql\_sync(

        schema,

        data,

        context\_value=request,

        # debug=app.debug

    )

    status\_code = 200 if success else 400

    return jsonify(result), status\_code

# type\_defs = gql("""

#     type Query {

#         kater: KaterQuery

#     }

#     type KaterQuery {

#         genericView: Generic

#     }

#     scalar Generic

# """)

# or following

type\_defs = load\_schema\_from\_path('graphql\_container')

query = QueryType()

@query.field('kater')

def resolve\_kater(\*\_):

    return {}

katerQuery = ObjectType('KaterQuery')

@katerQuery.field("genericView")

def resolve\_people(\*\_):

    return [

        {"firstName": "Sushant", "lastName": "Agrawal", "age": 58},

        {"firstName": "Prashant", "lastName": "Agrawal", "age": 58}

    ]

schema = make\_executable\_schema(type\_defs,katerQuery, query)

### App server: Python Flask used

Following libraries are to be installed at cloudjiffy server by pip install

flask, ariadne, flask\_cors

* Used Flask as App server
* At app.cloudjiffy .com created python environment 3.10.6
* Created wsgi.py file
* Uploaded KaterServer.zip file to unzip and create KaterServer folder. Initially tested with Hello, which worked. [https://kater-server.cloudjiffy.net](https://kater-server.cloudjiffy.net/) returned Hello
* Implemented Flask BluePrint in the app server for routing

### Logging

* Implemented monthly logger as logger.py. Every month of year a new log file in logs folder

## Client side

* Used React version 18.2
* Used hookstate as global state management library [Hookstate: supercharged React.useState hook | Hookstate](https://hookstate.js.org/)
* Error handling strategy

# Tips and tricks

## Client side

### Material-ui

#### Customizing theme

* Buttons

To change button style globally use styleOverrides. To create a new variant use variants

const theme: any = createTheme({

        components: {

            MuiButton: {

                styleOverrides: {

                    root: {

                        textTransform:'none',

                    },

                },

                variants: [

                    {

                        props: { variant: 'menuButton' },

                        style: {

                            border: '2px solid transparent',

                            fontSize:'1.2rem',

                            '&&:hover': {

                                border: `2px solid ${globalTheme.palette.background.default}`,

                            },

                        },

                    },

                ],

            },

        },

    })

* Change style of selected item in MenuItem. In documentation in classes of root you find a class .Mui-selected

Now in the parent element in sx create a class with name .Mui-selected. This class actually overrides the built-in class.

sx={{

                width: drawerWidth,

                flexShrink: 0,

                '& .MuiDrawer-paper': {

                    width: drawerWidth,

                    boxSizing: 'border-box',

                },

                '& .Mui-selected': {

                    // for selected item to show red

                    color: 'red',

                    backgroundColor: `${theme.palette.grey[200]}!important`,

                },

            }}

# Technical understanding

## Flask

### Response

If you return dict from a router jsonify() is automatically called and it creates a response with corresponding json. If you return a list or dict it will be converted to json.

### Error handling at flask

The http related errors such as 3xx, 4xx are handled through

@app.errorhandler(exceptions.HTTPException)

def handle\_http\_exception(e):

    return jsonify(code=e.code, name=e.name, descr=e.description), e.code

The unresolved errors are handled through

@app.errorhandler(Exception)

def handle\_http\_exception(e):

    return jsonify(code=e.code, name=e.name, descr=e.description), e.code

It is recommended to handle all other errors through try catch block and in catch block (except block) raise GenericException with details of error in code, description and name. Separately created a GenericException Python class. Error is handled in this manner

@app.errorhandler(GenericException)

def handle\_generic\_exception(e):

    return jsonify(code=e.code, name=e.name, descr=e.description), 500

Error code is kept at 500 so the browser understands it as ‘internal server error’. Of course logging is also done in real code.