Python Software Foundation

Sub-org: EOS Design System

Project: EOS-icons - feature/icon request [backend]

By - Aditya Sharma

Contact Information

Name: Aditya Sharma
Email: sharmaaditya570191@gmail.com
Github Username: sharmaaditya570191
Gitlab Username: sharmaaditya570191
Mobile: +91-8491958379
Country: India
Time-Zone: UTC +5:30
Primary Language: English
Linked-In: Aditya Sharma

University Information

University name: Shri Mata Vaishno Devi University
Major: Computer Science and Engineering
Current year: 2nd year
Expected Graduation Date: August 2022
Degree: Bachelor of Technology (B.Tech)

Project Abstract

EOS is the first open source and customizable design system to help open source, SMEs, and all sizes of organizations deliver outstanding user interfaces and consistent user experience.

Users can design a custom iconic font on the eos-icons landing page by choosing from a wide variety of EOS icons. These can be used in any open source project by following the guides in the documentation page. Currently people can suggest and request new icons and features by opening an issue on Gitlab.
The problem here is that most people do not want to sign in to Gitlab to send an issue and want something faster within the same interface of EOS. Going to Gitlab every time to monitor and manage a large number of requests and responses is not a user friendly option either.
The aim of this project is to design a scalable backend infrastructure that delivers a web interface allowing users to request new icons in an easy and intuitive way. Users can attach files with the request to explain what they want. The admins or EOS maintainers can then resolve, close and revise these requests. Even users can comment and vote for the existing requests. This can also serve as an efficient feedback and response mechanism which is critical for any organization to improve and make progress. Thus, it potentially becomes another reusable open source project for EOS.

Code Contribution

Pull Requests

Target Branch**:** master

- Status: Merged
  View Pull Requests

- Status: Open, Approved
  View Pull Requests

Target Branch: ops/react-migration

- Status: Merged
  View Pull Requests

Issues

- Status: All
  View Issues

Repository

This was the initial repository structure I made to experiment with different toolchains used with modern day React and decide the best tools we can use at EOS to migrate the existing website to React.View Repository

- Status: Used Create-React-App for a modern build setup with zero configuration

---

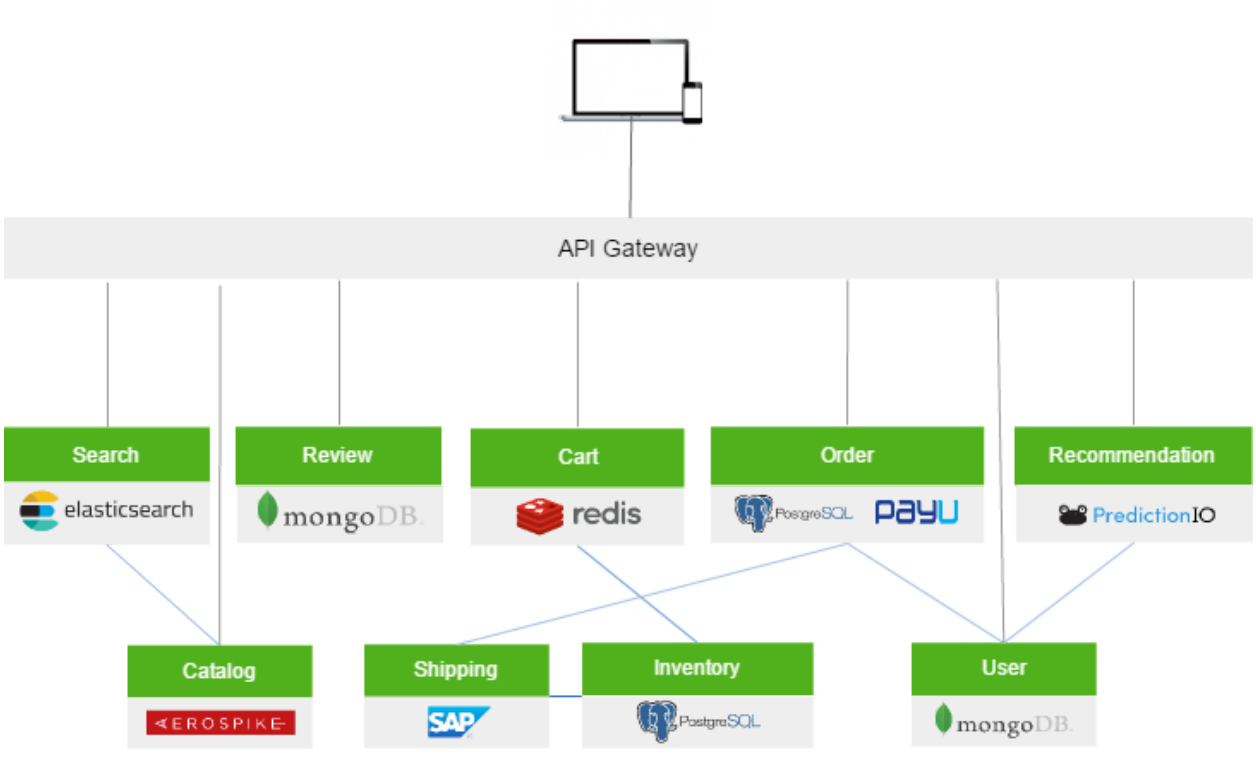
Core Backend Functionality

The server runs an app that contains logic about how to respond to various requests based on the HTTP verb and the Uniform Resource Identifier (URI). The pair of an HTTP verb and a URI is called a *route* and matching them based on a request is called *routing*. Some of these handler functions will be *middleware*. In this context, middleware is any code that executes between the server receiving a request and sending a response.
These middleware functions might modify the request object, query the database, or otherwise process the incoming request. Middleware functions typically end by passing control to the next middleware function, rather than by sending a response. Eventually, a middleware function will be called that ends the request-response cycle by sending an HTTP response back to the

client. Each route can have one or many handler functions that are executed whenever a request to that route (HTTP verb and URI) is matched.

Research and Implementation

I have been researching this topic for the last month and came across the following ideas to implement this project.
Our main goal is to keep the API design simple and reusable for all EOS projects. In order to achieve this we must expose all our data and functionality through service interfaces and all interprocess communication should be allowed only through these interfaces. These interfaces can then be used by various EOS projects for feedback or discussion regarding existing products at EOS.
An intuitive way to do this is to hide these services behind a new service layer and provide an API that is tailored to each client. This aggregator service layer is also known as **API Gateway** and is a common way to tackle this.



This is an example of the microservices architecture discussed above which tackles the problem by dividing it into smaller subproblems aka divide and conquer in developers world.
A great, simple and effective way to implement this is by using **Strapi,** a flexible open source Node.js headless CMS currently preferred at EOS.
The Strapi's admin panel gives us an intuitive interface to create, edit and delete our content. We can generate the admin panel in just a few clicks and get our whole CMS setup in a few minutes. We can fetch all our data with a REST API that Strapi will provide us.
Strapi gives us the option to choose the most appropriate database for our project. **MongoDB** is the preferred database at EOS.
MongoDB must already be running in the background if we want to create a Strapi project locally using the MongoDB database. This can be done and verified as follows.

```
 ✝  ~  sudo service mongod start
[sudo] password for sharmaaditya570191:
 ✝  ~  service mongod status
● mongod.service - MongoDB Database Server
   Loaded: loaded (/lib/systemd/system/mongod.service; disabled; vendor preset:
   Active: active (running) since Mon 2020-03-09 07:00:28 IST; 4s ago
     Docs: https://docs.mongodb.org/manual
 Main PID: 11314 (mongod)
   CGroup: /system.slice/mongod.service
           └─11314 /usr/bin/mongod --config /etc/mongod.conf

Mar 09 07:00:28 sharmaaditya systemd[1]: Started MongoDB Database Server.
lines 1-9/9 (END)
```

Moving ahead with a custom installation type we can simply create our project with:

```
 ✝  ~  yarn create strapi-app eos
```

This will generate a project with the following file structure:

```
 ✝  ~/eos  tree -L 1
.
├── api
├── build
├── config
├── extensions
├── favicon.ico
├── node_modules
├── package.json
├── public
├── README.md
└── yarn.lock

6 directories, 4 files
```

Strapi also comes with a full featured command line interface (CLI) which will let us scaffold and manage our project in seconds. This allows us to perform really fast operations without even opening the user interface. The available scripts can be seen as follows:

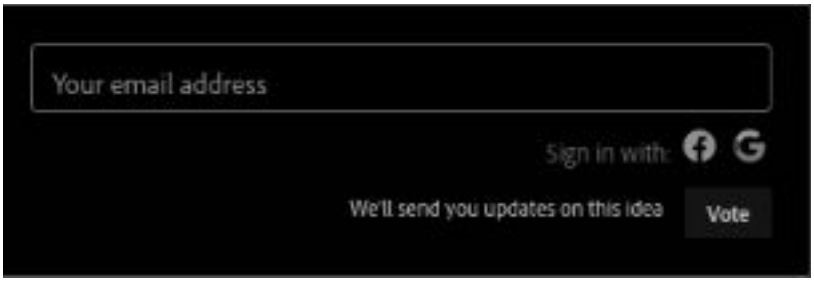
```
 ✝  ~/eos  yarn strapi
yarn run v1.21.1
$ strapi
Usage: strapi [options] [command]

Options:
  -v, --version                                output the version number
  -h, --help                                   output usage information

Commands:
  version                                      output your version of Strapi
  console                                      open the Strapi framework console
  new [options] <directory>                    create a new application
  start                                        Start your Strapi application
  develop|dev [options]                        Start your Strapi application in development mode
  generate:api [options] <id> [attributes...]  generate a basic API
  generate:controller [options] <id>           generate a controller for an API
  generate:model [options] <id> [attributes...] generate a model for an API
  generate:policy [options] <id>               generate a policy for an API
  generate:service [options] <id>              generate a service for an API
  generate:plugin [options] <id>               generate a basic plugin
  build [options]                              Builds the strapi admin app
  install [plugins...]                         install a Strapi plugin
  uninstall [options] [plugins...]             uninstall a Strapi plugin
  watch-admin                                  Starts the admin dev server
  help                                         output the help
Done in 0.50s.
```

Next we start our application in development or watch mode to create an Administrator (or "root user") for our project. An Administrator has all administrator privileges and access rights.

Now we create the schema of the data structure ie. the Content Type. A Content Type can be considered a sort of blueprint for the data created.

Now we want to generate a request system where users can create an account, login and then submit icon or feature requests and comment and vote on existing requests. Admins can comment on the request to update the progress made and then close them on completion. This would include functionality to register new users to Strapi and provide authentication, password reset and email validation. Following is an sample interface this system will provide:

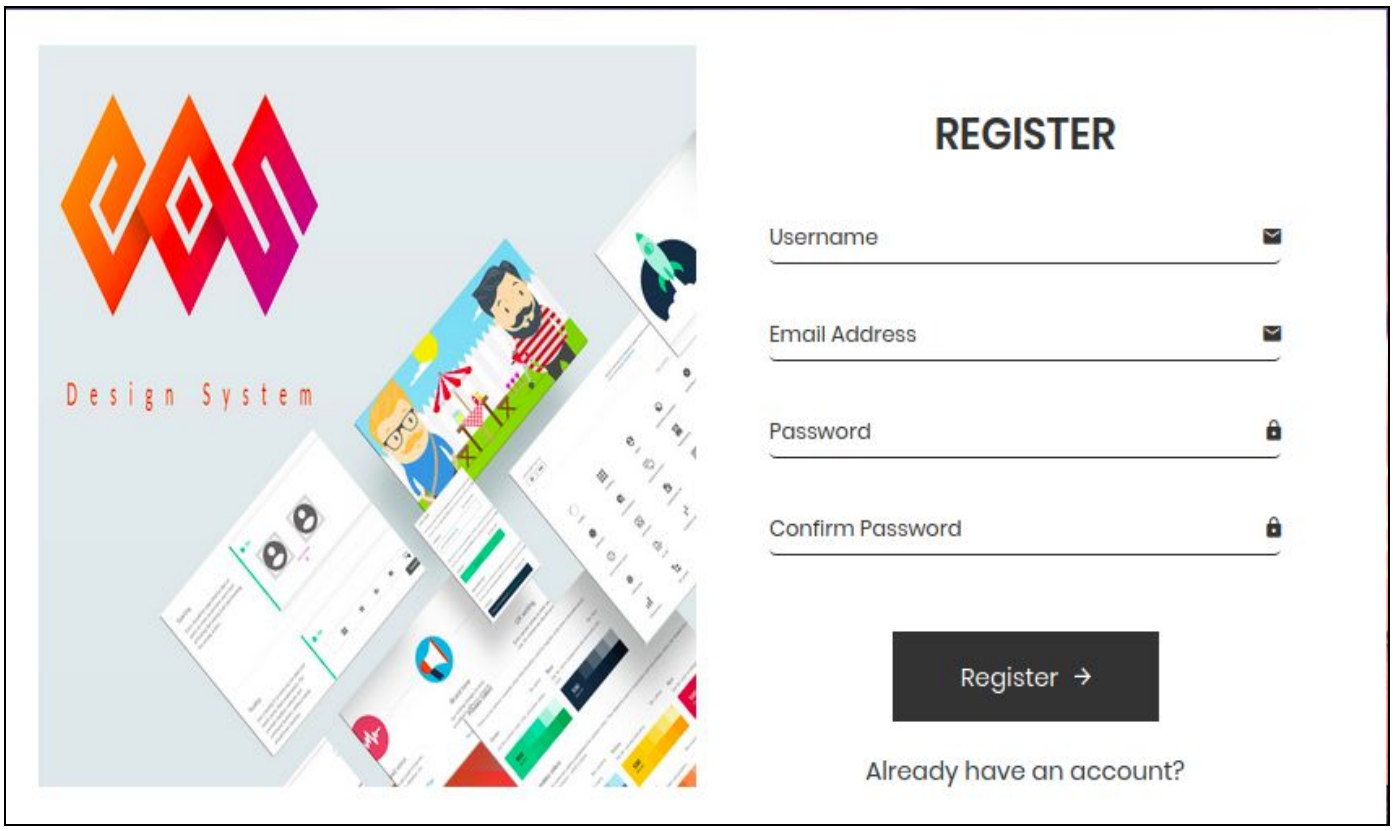
The system shall enable users to vote for a existing request like this:



Admins may also assign a particular icon request to a designer in the EOS team which can be implemented by adding the assignees to an array. Admins can also add some labels to the existing requests for a quick status update. We can also use and deploy bots for adding relevant labels to the request like:

Comments to the icon request can be added like this:

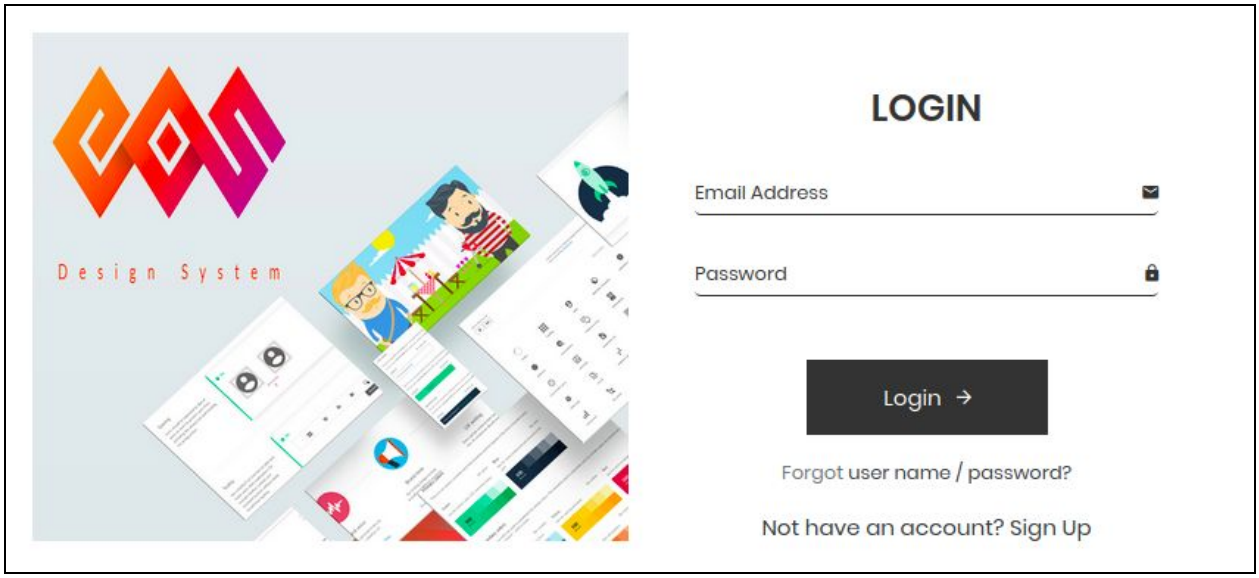
The initial functionality of registering new users can be achieved by:



Now we have to persist the user data to our database after checking that the user has filled in all credentials correctly. This interface can be connected with the MongoDB database via our Strapi instance as follows:

```
if(username != "" && email != "" && password != "" && cPassword != "") {
    if(password == cPassword) {
        axios
            .post('http://localhost:1337/auth/local/register', {
                username: username,
                email: email,
                password: password,
                })
            .then(response => {
            // Handle success.
            console.log('Well done!');
            console.log('User profile', response.data.user);
            console.log('User token', response.data.jwt);
            })
            .catch(error => {
            // Handle error.
            console.log('An error occurred:', error);
            });
    }
    else {
        window.alert("Password does not match with the Confirm Password.")
    }
}
else {
    window.alert("Form is incomplete. Please fill out all fields.")
}
```

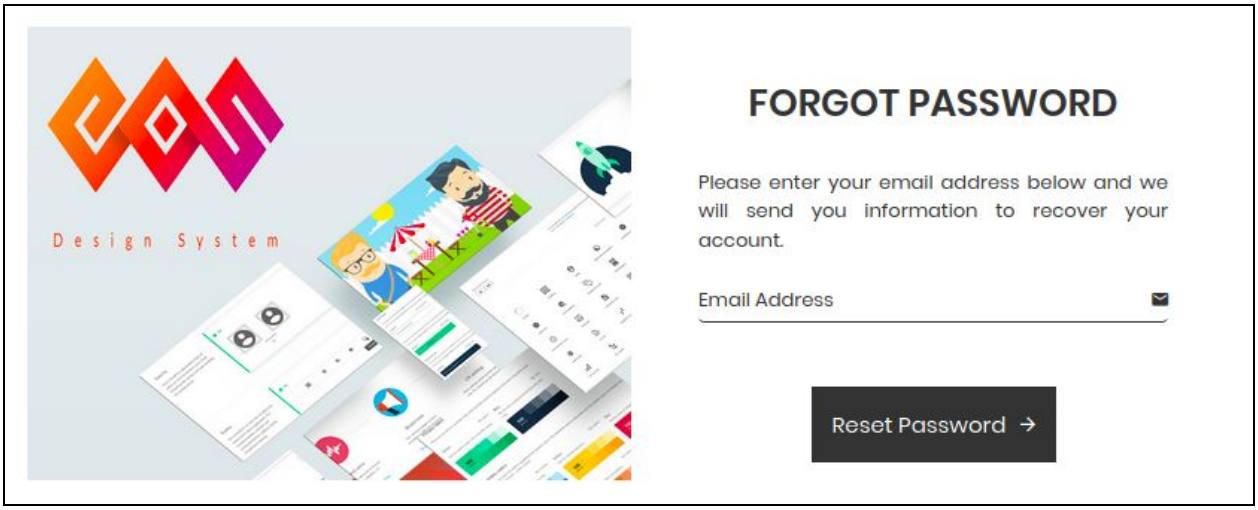
Now we can give users a sign in option after they have registered in the icon request system. This can be achieved by:



This uses our Strapi API after performing basic form checks as follows:

```
if(email != "" && password != "") {
    // Request API.
    axios
        .post('http://localhost:1337/auth/local', {
        identifier: email,
        password: password,
        })
        .then(response => {
        // Handle success.
        console.log('Well done!');
        console.log('User profile', response.data.user);
        console.log('User token', response.data.jwt);
        })
        .catch(error => {
        // Handle error.
        console.log('An error occurred:', error);
        });
}
else {
    window.alert("Form is incomplete. Please fill out all fields.")
}
```

Users can recover their account with the password reset functionality where they will be sent information via email to reset their password:



This consumes the API like:

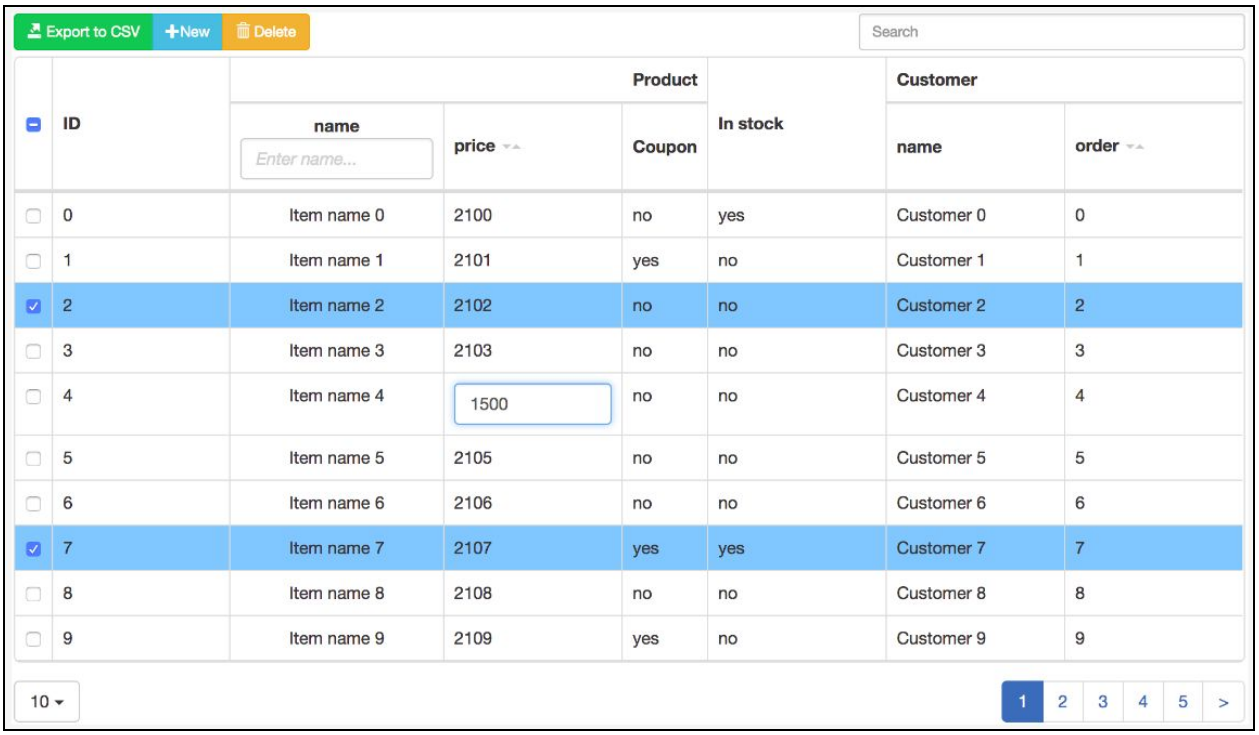
```
if(email != "") {
    axios
        .post('http://localhost:1337/auth/forgot-password', {
            email: email
        })
        .then(response => {
            // Handle success.
            console.log('Your user received an email');
        })
        .catch(error => {
            // Handle error.
            console.log('An error occurred:', error);
        });
}
else {
    window.alert("Please provide your email first.")
}
```

Users can select from a variety of icon categories while making the request like regular, bold, animated etc. to name a few.

A live demo of the initial API design can be seen here:

Click here to view live demo

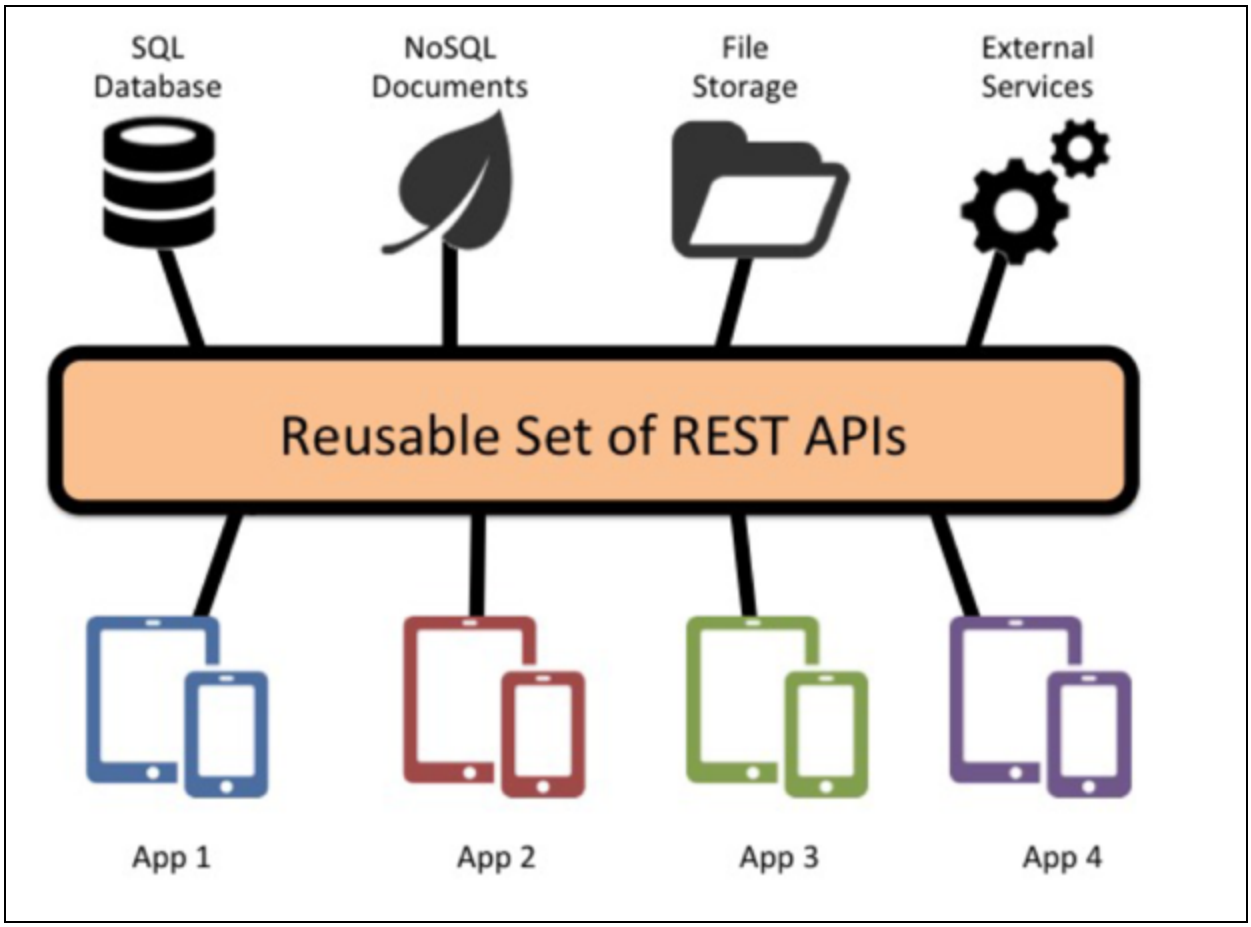
To display all icon requests we can organize all the data in the form of an intuitive table design. This table would include features like searching, sorting, pagination among many and would fetch requests data from our Strapi API. To implement this we can use react-bootstrap-table2 or design a custom made table from scratch in **React.js**.



Alternative Approach

Another way of implementing the feature and icon request system is by creating a custom backend from scratch using **Express.js**, a framework based on **node.js**. This involves identifying data sources that need to be mobilized, then creating a comprehensive and reusable REST API back end that supports general-purpose application development.

There are some basic characteristics that any reusable REST API needs to have. The API needs to support both HTML5 and native mobile access patterns. Requests and responses should include JSON or XML with objects, arrays, and subarrays.

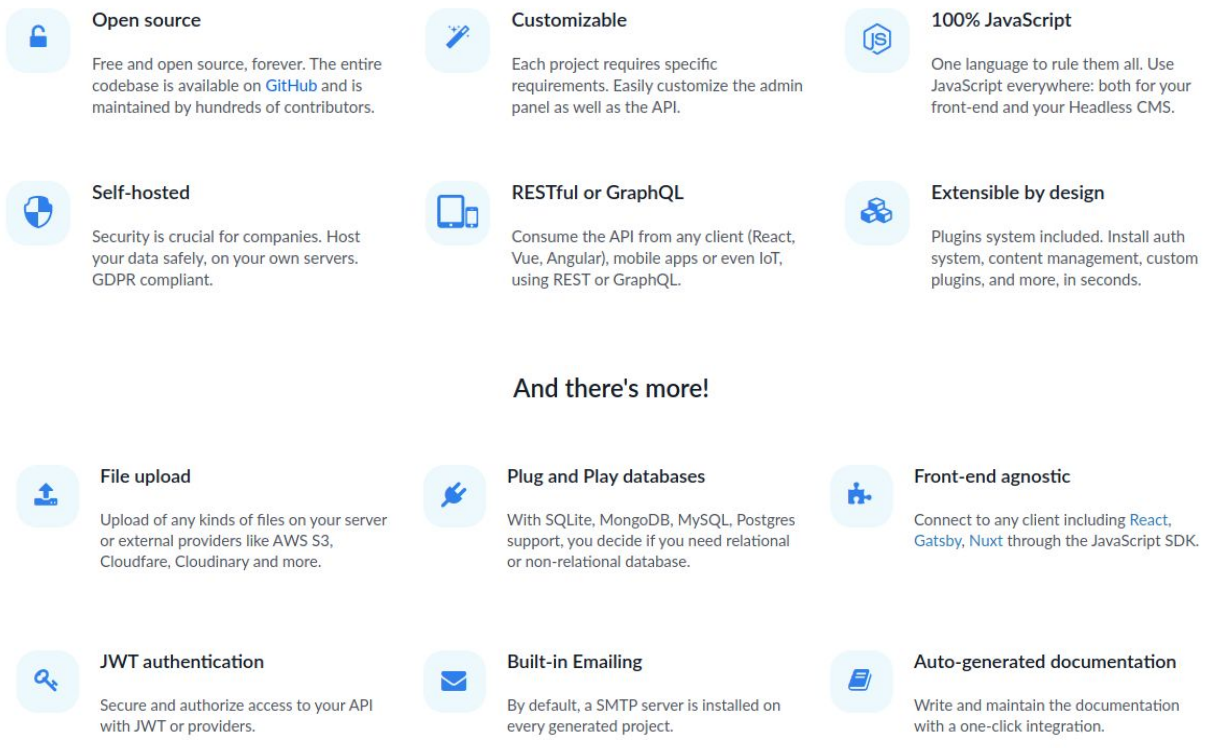


Noun-based endpoints should be automatically generated depending on the database schema. All HTTP verbs (GET, PUT, DELETE, and so on) need to be implemented for every use case. Support for Web standards like OAuth, CORS, GZIP, and SSL is also important.

There needs to be a consistent URL structure for accessing any back-end data source. Parameter names should be reused across services where possible. This presents developers with a familiar interface for any data source. The API should include interactive documentation that allows developers to quickly understand and try an API as well as experiment with different parameters.

Decision

Why hurt yourself by creating again and again a backend when we can use a product that is simply understandable by a JavaScript developer proven by its community and focus on the part that we prefer in order to deliver a final product very quickly.
It is better to use Strapi CMS because of the following advantages:



Integrating Strapi with React

React is amazing not only because it's blazing fast but also most importantly for its component-based architecture.
When using a CMS, React unleashes all its power mainly because of this architecture since the developer will design and develop really reusable components.
Growing and maintaining the codebase is, therefore, easier since no content is hard-coded.
The benefit to this approach is that we **end up doing less duplicate work** by **managing the components** rather than managing duplicate content across different pages. This concept has become a **real success**.

Testing

Strapi's test suite is written using mocha although Strapi doesn't impose any testing framework for our apps.
We are going to set up *bootstrap.js* with *before* and *after* hooks to perform any actions before and after our tests. In this example, the app server is started before running any tests and stops the server after tests are completed.

./test/bootstrap.js

```
const strapi = require('strapi');

before(function (done) {
  strapi.start({}, function(err) {
    if (err) {
      return done(err);
    }

    done(err, strapi);
  });
});

after(function (done) {
  strapi.stop(done());
});
```

Once we have set up our directory structure we can use co-supertest, a co and Supertest integration library. Supertest provides several useful methods for testing HTTP requests.
If we want to test an api endpoint, we can do it like this:

./test/integration/controllers/my_endpoint.js

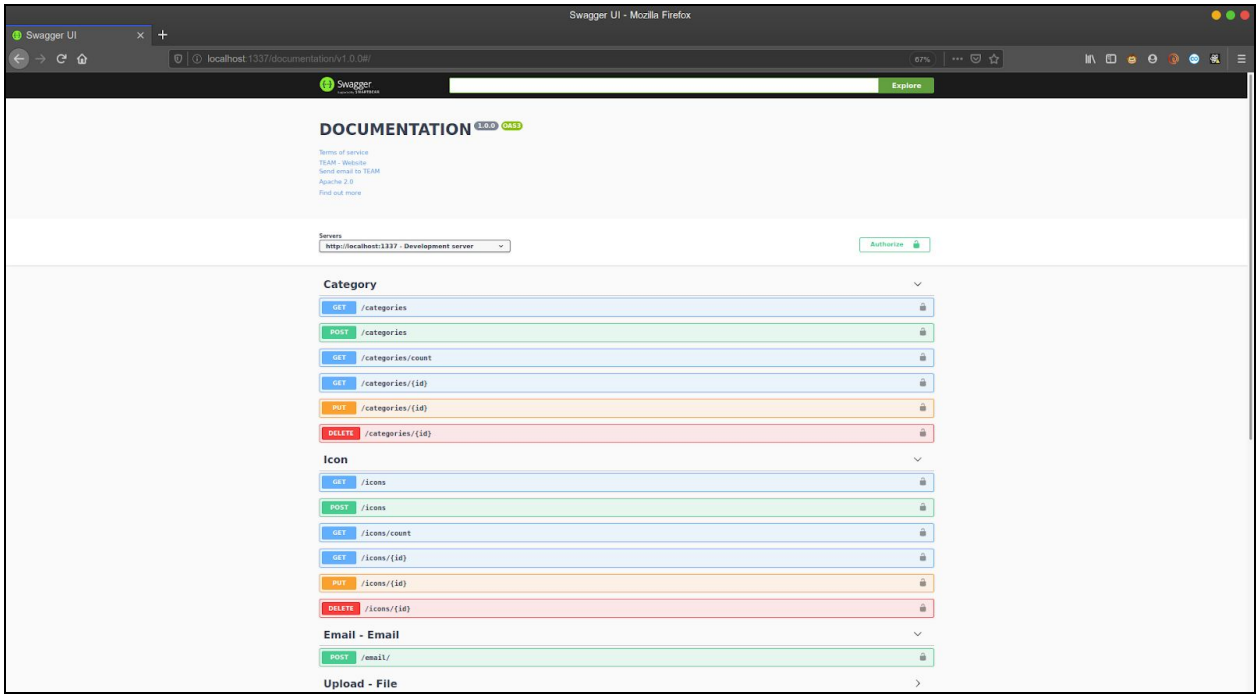
```
const request = require('co-supertest');

describe('MyEndpoint Controller Integration', function() {
  describe('GET /my_endpoint', function() {
    it('should return 200 status code', function *() {
      yield request(strapi.config.url)
        .get('/my_endpoint')
        .expect(200)
        .expect('Content-Type', /json/)
        .end();
    });
  });
});
```

Documentation

Now that we have created our API it's really important to document its available end-points. The documentation plugin takes out most of our pain to generate our documentation. This plugin uses SWAGGER UI to visualize our API's documentation.
A detailed API documentation can be generated like this:

```
 ✝  ~/eos  sudo yarn strapi install documentation
[sudo] password for sharmaaditya570191:
yarn run v1.21.1
$ strapi install documentation
✔ Installing dependencies
✔ Rebuilding admin UI
Done in 467.25s.
```

The documentation can now be viewed via the admin panel:



Deployment

Strapi gives us many possible deployment options for our project or application. Strapi can be deployed on traditional hosting servers or services such as Heroku, AWS, Azure and others. As EOS has previously used Heroku to deploy API's we can follow the detailed guide here to achieve this.

Search Engine Optimization (SEO)

So how do I get on the first page of Google?
Everyone wants to know the answer to that million-dollar question. Without going into great detail I would like to discuss the methods I found to optimize our app for search engines.

- Reach Router: Gives us a unique URL for each of the views in our app.

- React Helmet: Allows us to set title, description and other header tags.
- Fetch as Google (EDIT: Now URL inspection) : Helps troubleshoot Google's ability to view our content.
- React-SEO: The crawlers will wait for an AJAX call to be returned if the call was made before the page render. react-seo will allow us to do exactly that via a configurable API.

Just using these four essential tools, we will be able to boost our app's ranking in Google and other search engines.

Timeline

Community Bonding (May 4 - June 1)

I will utilize every moment of this period to understand the workflow at EOS in detail. This will help me a lot to make further progress in the project. I will use this time to plan the API design and integration of the front-end and back-end for the icon request system. I will also give some time to my university exams during this period along with getting familiar with all community members at EOS.

Output of this period:

- API planning and design
- Enhance knowledge of Strapi CMS

Week 1 (June 2 - June 8)

During this time I will create an administrator or root user who has access and rights to the Administration Dashboard (or backend) of Strapi. Administrators can, for example, add content, add plugins, and upload images using Strapi. This includes starting a new project with Strapi and connecting it with an instance of MongoDB locally.

Output of this period:

- Custom Strapi instance plugged with MongoDB
- Admin panel for the app

Week 2 (June 9 - June 15)

This time will be used to create a front-end user on Strapi. A front-end user is someone who interacts with your project through the front-end. A front-end user can, for example, be an "Author" of an article, make a purchase, have an account, leave a review, or leave a comment.

Output of this period:

- Create front-end users
- Set roles and permissions for the front-end users

Week 3 (June 16 - June 22)

I will use this time to add functionality to sign up new users to the Strapi instance by consuming the REST API exposed by Strapi. This would involve integrating functionality with React on the client side.

Output of this period:

- Add sign-up functionality to app
- Integrate sign-up and authentication with client

Week 4 (June 23 - June 29)

I will use this period to implement sign-in or login functionality once the user is registered and authenticated via email or other providers. This would include implementation of requests via axios library from client side to connect to the backend.

Output of this period:

- Add sign-in functionality to app
- Integrate React sign-in component with backend

Week 5 (June 30 - July 6)

This is the time for first evaluations. I will work together with the mentors to get feedback and improve and work on code reviews by the mentor. I will also add user authentication functionality using JWT tokens. My main aim will be to get my patches merged into the main tree during the evaluation period.

Output of this period:

- Add user authentication using JWT tokens
- Modify the permissions of each user's role in admin dashboard
- Check for 401 authentication errors
- Submit feedback for evaluations

Week 6 (July 7 - July 13)

This involves working on the password reset feature of the app which will help in easy user account recovery in case of loss of credentials by the user.

Output of this period:

- Add password reset functionality to the API
- Use axios request to API auth endpoints in Strapi
- Integration with React component on client side

Week 7 (July 14 - July 20)

I will add more OAuth and OAuth2 providers for authentication to enhance the accessibility of the system.

Output of this period:

- Integrate Facebook, Github, Google, Twitter and Discord with the help of ngrok package to work with providers that don't allow localhost redirect URIs

Week 8 (July 21 - July 27)

This time will be used to create routes for the */icons* which will have the details of the icon requested by the user like name of the icon, description and some sample image file depicting the product needed.

Output of this period:

- Complete */icons* route to serve JSON response at the endpoints exposed by the Strapi API

Week 9 (July 28 - August 3)

This is the time for second evaluations and I will work closely with the mentors to solve issues with my existing code and work on code reviews. I would also integrate the JSON response sent by the API at */icons* route with the client side using axios requests.

Output of this period:

- Final submissions for the evaluations
- Complete the API route and React integration
- Feedback submission for evaluations

Week 10 (August 4 - August 10)

This time will be used to create the */categories* route using Strapi API which defines the various categories of icons that the user can request or a custom entry by the user.

Output of this period:

- Complete */icons* route to serve JSON response at the endpoints exposed by the Strapi API
- Integrate the API with the client side managing state of the app on React side
- Initial unit tests using chai and mocha

Week 11 (August 11 - August 17)

Work on integrating the various users generated on the platform via */users* route to organize the users data in the form of a responsive table on the client side.

Output of this period:

- Organize users data via */users* data
- Code enhancement and cleaning
- Unit tests for the API

Week 12 (August 18 - August 24)

Work on completing unit tests for the API and ensuring that the API passes all tests for various use cases.

Output of this period:

- Complete unit tests
- Cleaning code and final API routes

Week 13 (August 25 - August 31)

I will use this time to generate final API documentation using the documentation plugin available in Strapi.

Output of this period:

- Complete documentation for the API to help developers
- Fix minor issues and code cleaning
- Final project submission

Previous Experience

I am a web, linux and open source enthusiast and I have contributed to various open source organizations like Mozilla and Creative Commons to name a few. I have made various personal web based full stack projects which are showcased on my github profile. I gained knowledge in the web development area using various courses and workshops available via the github student developer pack and technical blogs written by highly skilled industry experts.
Apart from web development I lead the code club at my university, guiding students in their open source journey.
I have also mentored around 250 students in open source programs like Kharagpur Winter of Code and Jaypee Month of Code. I was part of Girlscript Summer of Code under Girlscript foundation as a project administrator. I have even participated and won in team events like university hackathons.

About Me

I am a typical geek who loves programming and enjoys problem solving and making side projects as a part of hobby coding.

Open source contributions give me satisfaction as I solve real world problems with my code patches. I love to go for short nature walks when I take breaks and enjoy interacting with pets around me.
I am confident about completing this project efficiently as I am contributing to EOS from quite some time and I am familiar with the codebase. Also I like the wonderful and supportive community members at EOS and would love working with them.

Stretch goals

Command Line Interface
Short of time and cannot navigate to the eos-icons interface?
Use the command line interface via your terminal to submit an icon or feature request in a second. We can consume the REST API exposed by Strapi using some of the best command line REST clients like resty and HTTPie.
Developers can use the CLI via http-prompt which is built on top of HTTPie.

Note: This idea is not a part of the original project and can be implemented once the main project work is complete.
