TITLE: RUFLEX: Revolutionizing User Engagement and Adoption with a User-Centric Utility on Rootstock

TEAM: Trinetra1627108

Here's the complete code with front-end and back-end integration:

Front-end:

- HTML
- CSS
- JavaScript (Client-side)

Back-end:

- Node.js
- Express.js
- MongoDB
- Mongoose (ODM)

HTML:

```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Your Platform Name</title>
  <link rel="stylesheet" href="styles.css">
</head>

<body>
```

```
<header>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Services</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</header>

<main>
  <section id="hero">
    <div class="container">
      <h1>Welcome to Your Platform</h1>
      <p>Discover a new way to manage your tokens.</p>
      <a href="#" class="cta-button">Get Started</a>
    </div>
  </section>

  <section id="features">
    <div class="container">
      <h2>Platform Features</h2>
      <div class="feature">
        <img src="icon-1.png" alt="Feature 1">
        <h3>Feature 1</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
      </div>
      <div class="feature">
        <img src="icon-2.png" alt="Feature 2">
        <h3>Feature 2</h3>
```

```
      <p>Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
    </div>
    <div class="feature">
      <img src="icon-3.png" alt="Feature 3">
      <h3>Feature 3</h3>
      <p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris.</p>
    </div>
  </div>
 </section>

 <section id="signup">
  <div class="container">
   <h2>Sign Up</h2>
   <form id="signup-form">
    <input type="text" id="name" placeholder="Full Name" required>
    <input type="email" id="email" placeholder="Email Address" required>
    <input type="password" id="password" placeholder="Password" required>
    <input type="password" id="confirm-password" placeholder="Confirm Password" required>
    <button type="submit">Sign Up</button>
   </form>
  </div>
 </section>
</main>

<footer>
 <div class="container">
  <p>&copy; 2023 Your Platform Name. All rights reserved.</p>
 </div>
</footer>

<script src="client.js"></script>
```

```
</body>

</html>
```

CSS (styles.css):

```css
/* Reset some default styles */
body, h1, h2, h3, p, ul, li {
  margin: 0;
  padding: 0;
}

/* Apply basic styles to the container */
.container {
  max-width: 960px;
  margin: 0 auto;
  padding: 20px;
}

/* Style the header and navigation */
header {
  background-color: #333;
  padding: 10px;
}

nav ul {
  list-style: none;
  display: flex;
  justify-content: center;
}
```

```
nav ul li {
  margin-right: 20px;
}

nav ul li a {
  color: #fff;
  text-decoration: none;
}

/* Style the hero section */
#hero {
  background-color: #f2f2f2;
  text-align: center;
  padding: 40px 0;
}

#hero h1 {
  font-size: 36px;
  color: #333;
  margin-bottom: 20px;
}

#hero p {
  font-size: 18px;
  color: #666;
  margin-bottom: 40px;
}

.cta-button {
  display: inline-block;
```

```
  padding: 12px 24px;
  background-color: #333;
  color: #fff;
  text-decoration: none;
  border-radius: 4px;
  transition: background-color 0.3s;
}

.cta-button:hover {
  background-color: #555;
}

/* Style the features section */
#features {
  padding: 60px 0;
  text-align: center;
}

#features h2 {
  font-size: 28px;
  color: #333;
  margin-bottom: 40px;
}

.feature {
  display: flex;
  flex-direction: column;
  align-items: center;
  margin-bottom: 40px;
}
```

```
.feature img {
  width: 80px;
  height: 80px;
  margin-bottom: 20px;
}

.feature h3 {
  font-size: 24px;
  color: #333;
  margin-bottom: 10px;
}

.feature p {
  font-size: 16px;
  color: #666;
}

/* Style the sign-up section */
#signup {
  background-color: #333;
  color: #fff;
  text-align: center;
  padding: 60px 0;
}

#signup h2 {
  font-size: 28px;
  margin-bottom: 40px;
}

#signup form input[type="text"],
```

```
#signup form input[type="email"],
#signup form input[type="password"] {
  width: 100%;
  padding: 12px;
  margin-bottom: 20px;
  border-radius: 4px;
  border: none;
}

#signup form button[type="submit"] {
  padding: 12px 24px;
  background-color: #f2f2f2;
  color: #333;
  text-decoration: none;
  border-radius: 4px;
  border: none;
  transition: background-color 0.3s;
  cursor: pointer;
}

#signup form button[type="submit"]:hover {
  background-color: #e0e0e0;
}

/* Style the footer */
footer {
  background-color: #f2f2f2;
  text-align: center;
  padding: 20px;
  color: #666;
  font-size: 14px;
```

}

```

**Client-side JavaScript (client.js):**

```javascript
document.getElementById('signup-form').addEventListener('submit', async (e) => {
  e.preventDefault();

  const name = document.getElementById('name').value;
  const email = document.getElementById('email').value;
  const password = document.getElementById('password').value;
  const confirmPassword = document.getElementById('confirm-password').value;

  // Perform form validation
  if (!name || !email || !password || !confirmPassword) {
    return alert('Please fill in all fields');
  }

  if (password !== confirmPassword) {
    return alert('Passwords

do not match');
  }

  try {
    // Send form data to the server for registration
    const response = await fetch('/users/register', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ name, email, password, confirmPassword })
    });
```
```

```
    if (response.ok) {
      alert('User registered successfully');
      // Reset form fields
      document.getElementById('name').value = '';
      document.getElementById('email').value = '';
      document.getElementById('password').value = '';
      document.getElementById('confirm-password').value = '';
    } else {
      const data = await response.json();
      throw new Error(data.error);
    }
  } catch (error) {
    alert(error.message);
  }
});
```

Server-side JavaScript (server.js):

```javascript
const express = require('express');
const path = require('path');
const connectDB = require('./db');
const User = require('./models/User');

const app = express();

// Connect to MongoDB
connectDB();

// Serve static files
```

```
app.use(express.static(path.join(__dirname, 'public')));

// Parse JSON bodies
app.use(express.json());

// User registration endpoint
app.post('/users/register', async (req, res) => {
  const { name, email, password, confirmPassword } = req.body;

  try {
    // Validation and registration logic
    if (!name || !email || !password || !confirmPassword) {
      return res.status(400).json({ error: 'Please fill in all fields' });
    }

    if (password !== confirmPassword) {
      return res.status(400).json({ error: 'Passwords do not match' });
    }

    // Save user to the database
    const user = new User({ name, email, password });
    await user.save();

    return res.status(200).json({ message: 'User registered successfully' });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Server error' });
  }
});

// Handle other routes
```

```
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

To run the application, you'll need to install the required dependencies using npm or yarn. Run the following commands in your project directory:

```
npm install express mongoose
```

Make sure you have MongoDB installed and running.

Then, start the server:

```
node server.js
```

Open the web application in your browser, and you should be able to register a user using the sign-up form. The form data will be sent to the server, and if the registration is successful, you'll receive a success message.

Please note you may need to adapt the code to fit your specific requirements. Additionally, you can further enhance the application with additional features and validations as needed.

Here's a step-by-step explanation with (BACKEND)code for setting up the project

Step 1: Set up the project.

1. Create a new directory for your project. Open your command line or terminal and navigate to the desired location where you want to create the project directory.

```
mkdir myplatform
```

2. Change into the newly created project directory.

```
cd myplatform
```

3. Initialize a new Node.js project using the `npm init` command. This will create a `package.json` file and prompt you to enter details about your project.

```
npm init
```

Follow the prompts and provide the necessary information.

4. Install the required dependencies: Express.js and Mongoose. Run the following command to install the dependencies and save them to your `package.json` file.

```
npm install express mongoose
```

This command will download the required packages and create a `node_modules` directory in your project, which will contain the dependencies.

Your project is now set up with the necessary dependencies installed. You can proceed to the next steps for creating the server file and connecting to MongoDB.

Note: Make sure you have Node.js and npm (Node Package Manager) installed on your system before running these commands.

Step 2: Create the server file.

1. Create a new file named `server.js` in the project directory.

2. Open `server.js` and require the necessary modules:

```javascript
const express = require('express');

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

The `express` module is required to create an instance of the Express app, and the `mongoose` module is required to connect to the MongoDB database.

3. Create an instance of the Express app:

```javascript
const app = express();
```

This creates an instance of the Express application, which allows you to define routes, middleware, and handle HTTP requests.

4. Set up middleware to parse incoming JSON data:

```javascript
app.use(express.json());
```

The `express.json()` middleware parses incoming requests with JSON payloads. It allows you to access the request body as a JavaScript object.

At this point, the `server.js` file should look like this:

```javascript
const express = require('express');
const mongoose = require('mongoose');

const app = express();
app.use(express.json());
```

In the next steps, we will connect to the MongoDB database and define routes for your application.

Step 3: Connect to MongoDB

1. Use Mongoose to connect to your MongoDB database. Add the following code to `server.js`:

```javascript
mongoose.connect('mongodb://localhost/myplatform', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => {
    console.log('Connected to MongoDB');
  })
  .catch((error) => {
    console.error('MongoDB connection error:', error);
  });
```

Replace `'mongodb://localhost/myplatform'` with your actual MongoDB connection string. This example assumes you are connecting to a local MongoDB instance and the name of the database is "myplatform". Adjust the connection string according to your specific MongoDB setup.

The `mongoose.connect()` function establishes a connection to the MongoDB database using the provided connection string and options. The options `useNewUrlParser: true` and `useUnifiedTopology: true` are used to ensure compatibility with the latest version of MongoDB.

The `then()` method is called if the connection is successful, and the `catch()` method is called if there is an error connecting to the database. In the example code, a success message is logged to the console if the connection is successful, and an error message is logged if there is an error.

At this point, the `server.js` file should look like this:

```javascript
const express = require('express');
const mongoose = require('mongoose');

const app = express();
app.use(express.json());

mongoose.connect('mongodb://localhost/myplatform', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => {
    console.log('Connected to MongoDB');
  })
  .catch((error) => {
    console.error('MongoDB connection error:', error);
  });
```

Step 4: Define the data models.

1. Create a new file named `models/User.js` to define the user schema and model.
2. In `User.js`, require `mongoose` and create the user schema:

```javascript
const mongoose = require('mongoose');
```

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
});

module.exports = mongoose.model('User', userSchema);
```

The `mongoose.Schema()` function is used to define the structure of the user document. In this example, we have defined three fields: `name`, `email`, and `password`. Each field has a specified type (`String`) and a `required` flag set to `true`, indicating that these fields are required when creating a new user.

The `mongoose.model()` function is used to create a model from the user schema. We pass the name ``'User'`` as the first argument, which will be used as the collection name in the MongoDB database. The second argument is the user schema (`userSchema`).

By exporting the model, we make it accessible in other parts of the application.

At this point, the `models/User.js` file should look like this:

```javascript

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
```

```
  },
  email: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
});

module.exports = mongoose.model('User', userSchema);
```

Step 5: Implement the API endpoints.

1. Create a new file named `routes/users.js` to define the user-related API endpoints.
2. In `users.js`, require `express` and `User` model:

   ```javascript
   const express = require('express');
   const User = require('../models/User');
   ```

3. Create an instance of the Express Router and define the API endpoints:

   ```javascript
   const router = express.Router();

   router.post('/register', async (req, res) => {
     try {
       const { name, email, password } = req.body;
       // Check if the user already exists
       const existingUser = await User.findOne({ email });
       if (existingUser) {
         return res.status(400).json({ error: 'User already exists' });
```

```
    }

    // Create a new user
    const user = new User({ name, email, password });
    await user.save();
    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    res.status(500).json({ error: 'Internal server error' });
  }
});

module.exports = router;
```

In this code snippet, we define a `POST /register` endpoint for user registration. Inside the endpoint, we retrieve the `name`, `email`, and `password` from the request body using `req.body`.

We then check if a user with the provided email already exists in the database by querying the `User` model with `User.findOne({ email })`. If an existing user is found, we return a 400 status code with an error message.

If the user doesn't exist, we create a new instance of the `User` model with the provided data and save it to the database using `user.save()`. We respond with a 201 status code and a success message.

In case of any errors, we catch them and return a 500 status code with an error message.

At this point, your `routes/users.js` file should look like this:

```javascript

```
const express = require('express');
const User = require('../models/User');

const router = express.Router();

router.post('/register', async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const existingUser = await User.findOne({ email });
```

```
    if (existingUser) {
      return res.status(400).json({ error: 'User already exists' });
    }
    const user = new User({ name, email, password });
    await user.save();
    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    res.status(500).json({ error: 'Internal server error' });
  }
});

module.exports = router;
```

Step 6: Mount the routes in the main server file.

1. In `server.js`, require and mount the user routes:

   ```javascript
   const userRoutes = require('./routes/users');
   app.use('/users', userRoutes);
   ```

   In this code snippet, we require the `userRoutes` from the `./routes/users` file and mount them under the `/users` route prefix using `app.use()`. This means that all the user-related routes defined in `routes/users.js` will be accessible under the `/users` URL path.

   By mounting the routes in this way, we can keep our server file organized and handle user-related routes separately.

   the `server.js` file should now look something like this:

   ```javascript

   ```
   const express = require('express');
   const mongoose = require('mongoose');
   const userRoutes = require('./routes/users');

   const app = express();
```

```
app.use(express.json());

mongoose.connect('mongodb://localhost/myplatform', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => {
    console.log('Connected to MongoDB');
  })
  .catch((error) => {
    console.error('MongoDB connection error:', error);
  });

app.use('/users', userRoutes);

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

In the next step implementing additional API endpoints for user authentication, such as login and token generation.

Step 7: Start the server.

1. Add the following code at the end of `server.js` to start the server listening on a specific port:

```**javascript**
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
});
```

```

In this code snippet, we define the `PORT` variable to either use the value from the `process.env.PORT` environment variable (if it exists) or default to port `3000`. We then call the `app.listen()` method to start the server and log a message indicating the port on which the server is running.

Now, when you run `node server.js` in your command line or terminal, the server will start and listen for incoming requests on the specified port.

the `server.js` file should now look something like this:

**```javascript**

```
const express = require('express');
const mongoose = require('mongoose');
const userRoutes = require('./routes/users');

const app = express();
app.use(express.json());

mongoose.connect('mongodb://localhost/myplatform', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => {
    console.log('Connected to MongoDB');
  })
  .catch((error) => {
    console.error('MongoDB connection error:', error);
  });

app.use('/users', userRoutes);

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
```
```

```
  console.log(`Server started on port ${PORT}`);
});
```

now we have set up the server, connected to the MongoDB database, defined the user schema, implemented the user registration endpoint, and started the server. You can continue to add more API endpoints and functionality as needed for application.

Step 8: Test the server.

1. In the command line or terminal, run `node server.js` to start the server.

2. Test the API endpoints using a tool like Postman or by making HTTP requests from your front-end code.

With the server running, you can now test the API endpoints to ensure they are working correctly. Here are the steps to test the registration endpoint using Postman:

- Open Postman or any similar API testing tool.

- Make a POST request to `http://localhost:3000/users/register` (replace `3000` with your actual server port if different).

- Set the request body to JSON format and provide the required user data, such as name, email, and password.

- Send the request and check the response from the server.

- Verify that the user is successfully registered, and the appropriate response is received.

You can also test the API endpoints using other HTTP clients or by making requests from your front-end code.

Testing the server helps ensure that the endpoints are functioning correctly, and that data is being processed and stored as expected. It is an essential step to validate the backend functionality before integrating it with the front-end application.

Once you have tested the server and verified that the API endpoints are working as intended, you can proceed with further development or deployment of your application.

Step 9: Deployment

1. Choose a cloud hosting platform or service for deployment, such as Heroku, AWS, or Azure.

2. Follow the deployment instructions specific to your chosen platform to deploy your Node.js application.

Deployment is the process of making your application available on the internet for users to access. Here are the general steps for deploying a Node.js application:

- Choose a cloud hosting platform or service: There are various cloud hosting platforms available, such as Heroku, AWS (Amazon Web Services), Azure, and others. Choose a platform that suits your requirements and sign up for an account.
- Set up your deployment environment: Follow the platform-specific instructions to set up your deployment environment. This typically involves creating a new application or project, configuring any necessary settings, and connecting to your version control system.
- Configure environment variables: Depending on your application's requirements, you may need to set up environment variables for configuration, such as database connection strings, API keys, and other sensitive information. Configure these environment variables in your deployment environment.
- Build and deploy your application: Use the deployment platform's tools or command-line interface to deploy your application. This usually involves pushing your code repository to the hosting platform, which triggers the deployment process. The platform will build and run your application in the deployment environment.
- Monitor and test your deployed application: Once deployed, monitor your application to ensure it is running correctly. Test all the functionality and API endpoints to verify that everything is working as expected. Use the logging and monitoring tools provided by the hosting platform to track any errors or issues.
- Update and maintain your deployed application: As you continue developing your application, you may need to deploy updates or bug fixes. Follow the platform's guidelines for deploying updates, and make sure to thoroughly test your changes before deploying them to the live environment.

The specific deployment steps and procedures may vary depending on the chosen hosting platform or service. Refer to the documentation and resources provided by the platform for detailed instructions on deploying a Node.js application.

It's also important to ensure that your deployment includes proper security measures, such as enabling SSL certificates, securing database connections, and implementing authentication and authorization mechanisms, depending on your application's requirements.

By following the deployment process and best practices, you can make your application accessible to users and provide a reliable and secure experience.