|  |
| --- |
| 1. Integration of Sendgrid to send emails |
| 2. Using nodemailer |
| 3. Sending OTP for Reset Password |
| 4. Sending Tickets on Email |

title : Creating the Link for forgot Password description : duration : 900 card\_type : cue\_card

In your login.js page add this one section of Forgot Password at the bottom and create a link for it so we can route

import React, { useEffect } from "react";  
import { Button, Form, Input, message } from "antd";  
import { Link, useNavigate } from "react-router-dom";  
import { LoginUser } from "../calls/users";  
  
function Login() {  
 const navigate = useNavigate();  
  
 useEffect(() => {  
 if (localStorage.getItem("token")) {  
 navigate("/");  
 }  
 }, [navigate]);  
  
 const onFinish = async (values) => {  
 try {  
 const response = await LoginUser(values);  
 if (response.success) {  
 message.success(response.message);  
 localStorage.setItem("token", response.token);  
 navigate("/");  
 } else {  
 message.error(response.message);  
 }  
 } catch (error) {  
 message.error(error.message);  
 }  
 };  
  
 return (  
 <>  
 <header className="App-header">  
 <main className="main-area mw-500 text-center px-3">  
 <section className="left-section">  
 <h1>Login to BookMyShow</h1>  
 </section>  
  
 <section className="right-section">  
 <Form layout="vertical" onFinish={onFinish}>  
 <Form.Item  
 label="Email"  
 htmlFor="email"  
 name="email"  
 className="d-block"  
 rules={[{ required: true, message: "Email is required" }]}  
 >  
 <Input id="email" type="text" placeholder="Enter your Email" />  
 </Form.Item>  
  
 <Form.Item  
 label="Password"  
 htmlFor="password"  
 name="password"  
 className="d-block"  
 rules={[{ required: true, message: "Password is required" }]}  
 >  
 <Input  
 id="password"  
 type="password"  
 placeholder="Enter your Password"  
 />  
 </Form.Item>  
  
 <Form.Item className="d-block">  
 <Button  
 type="primary"  
 block  
 htmlType="submit"  
 style={{ fontSize: "1rem", fontWeight: "600" }}  
 >  
 Login  
 </Button>  
 </Form.Item>  
 </Form>  
 <div>  
 <p>  
 New User? <Link to="/register">Register Here</Link>  
 </p>  
 <p>  
 Forgot Password? <Link to="/forget">Click Here</Link>  
 </p>  
 </div>  
 </section>  
 </main>  
 </header>  
 </>  
 );  
}  
  
export default Login;

title : Adding otp field at the Server Side description : duration : 900 card\_type: cue\_card

Add the otp field in your user Model at the server side

const mongoose = require("mongoose");  
  
const userSchema = new mongoose.Schema(  
 {  
 name: {  
 type: String,  
 required: true,  
 },  
 email: {  
 type: String,  
 required: true,  
 unique: true,  
 },  
 password: {  
 type: String,  
 required: true,  
 },  
 role: {  
 type: String,  
 enum: ["admin", "user", "partner"],  
 required: true,  
 default: "user",  
 },  
  
 // New Otp fields added  
 otp: {  
 type: String,  
 },  
 otpExpiry: {  
 type: Date,  
 },  
 },  
 { timestamps: true }  
);  
  
const User = mongoose.model("User", userSchema);  
  
module.exports = User;

In the Mongoose schema for the User model, the OTP (One-Time Password) part is managed by the otp and otpExpiry fields:

1. **otp**: This field is a string that stores the one-time password. It will be used for password reset. The OTP is typically generated and sent to the user, who then provides it back to the system for verification.
2. **otpExpiry**: This field is a date that specifies the expiry time for the OTP. It ensures that the OTP is only valid for a certain period, enhancing security by preventing reuse of the OTP after it has expired.

### How it works in practice:

1. **Generation**: When an OTP is needed (e.g., for account verification), the server generates a random OTP and sets the otp field with this value. It also sets the otpExpiry field to a future time, typically a few minutes ahead, indicating when the OTP will expire.
2. **Verification**: When the user provides the OTP back to the server, the server checks if the provided OTP matches the one stored in the otp field and if the current time is before the otpExpiry time.

We will now build these features

title : Creating Email Helper description : duration: 900 card\_type: cue\_card

create a folder by the name utils and add a file with the name emailHelper.js

emailHelper.js

Before creating this file we will understand two tools

1. Sengrid
2. nodemailer

### SendGrid

**SendGrid** is a cloud-based service that provides email delivery and marketing tools to help businesses communicate effectively with their customers. It handles the complexities of sending large volumes of email and ensures high deliverability rates. Key features of SendGrid include:

* **SMTP Relay**: Allows businesses to send emails through SendGrid’s servers by configuring their SMTP settings.
* **API**: Provides a RESTful API to send and manage emails programmatically.
* **Email Templates**: Offers tools to create and manage email templates.
* **Analytics**: Tracks email metrics such as opens, clicks, bounces, and spam reports.
* **Deliverability**: Implements best practices to ensure emails reach the inbox and not the spam folder.

**How SendGrid Works:**

1. **Account Setup**: Users sign up for a SendGrid account and verify their domain and sender identities to improve email deliverability.
2. **SMTP or API Configuration**: Users can configure their applications to send emails via SendGrid’s SMTP relay or using the SendGrid API.
3. **Sending Emails**: Emails are sent from the application to SendGrid, which then processes and delivers them to recipients’ inboxes.
4. **Tracking and Analytics**: SendGrid provides detailed analytics on email performance, helping users monitor and optimize their email campaigns.

### Nodemailer

**Nodemailer** is a module for Node.js applications that simplifies the process of sending emails. It supports various transport methods, including SMTP, to deliver emails. Nodemailer is widely used because of its ease of use and flexibility.

**Key Features of Nodemailer:**

* **SMTP Support**: Can send emails using the SMTP protocol.
* **Transport Plugins**: Supports various transport mechanisms, including SMTP, SES, and others.
* **HTML Content**: Allows sending emails with HTML content and attachments.
* **Secure Transport**: Supports secure connections using SSL/TLS.

**How Nodemailer Works:**

1. **Installation**: Nodemailer is installed via npm (Node Package Manager) in a Node.js project.
2. **Transport Configuration**: A transport method is configured, specifying the email service provider’s details (e.g., SMTP server, port, authentication).
3. **Email Composition**: Email details, including sender, recipient, subject, and content, are composed.
4. **Sending Emails**: The composed email is sent using the configured transport method.

### Summary

* **SendGrid** provides the infrastructure for sending emails, ensuring high deliverability and providing analytics.
* **Nodemailer** simplifies the process of composing and sending emails in Node.js applications.
* In the example, Nodemailer is configured to use SendGrid’s SMTP server to send templated emails. The email template is read from a file, placeholders are replaced with actual values, and the email is sent using SendGrid’s SMTP server. This integration leverages SendGrid’s robust email delivery infrastructure while utilizing Nodemailer’s simplicity for email composition and sending.

First of all Install Nodemailer by the command npm intsall nodemailer , then login to sengrid’s website and just generate an API key.

After this add this code in your EmailHelper.js file

const nodemailer = require("nodemailer");  
const fs = require("fs");  
const dotenv = require("dotenv");  
const path = require("path");  
dotenv.config();  
  
const { SENDGRID\_API\_KEY } = process.env;  
  
function replaceContent(content, creds) {  
 let allkeysArr = Object.keys(creds);  
 allkeysArr.forEach(function (key) {  
 content = content.replace(`#{${key}}`, creds[key]);  
 });  
  
 return content;  
}  
async function EmailHelper(templateName, reciverEmail, creds) {  
 // console.log(templateName, reciverEmail, creds)  
 try {  
 const templatePath = path.join(\_\_dirname, "email\_templates", templateName);  
 let content = await fs.promises.readFile(templatePath, "utf-8");  
 const emailDetails = {  
 to: reciverEmail,  
 from: "mrinal.bhattacharya@scaler.com", // Change to your verified sender  
 subject: "Mail from ScalerShows",  
 text: `Hi ${creds.name} this your reset otp ${creds.otp}`,  
 html: replaceContent(content, creds),  
 };  
 const transportDetails = {  
 host: "smtp.sendgrid.net",  
 port: 587,  
 auth: {  
 user: "apikey",  
 pass: SENDGRID\_API\_KEY,  
 },  
 };  
  
 const transporter = nodemailer.createTransport(transportDetails);  
 await transporter.sendMail(emailDetails);  
 console.log("email sent");  
 } catch (err) {  
 console.log(err);  
 }  
}  
  
module.exports = EmailHelper;

let’s break down the code step by step:

### Step-by-Step Explanation:

1. **Import Required Modules:**

const nodemailer = require("nodemailer");  
const fs = require("fs");  
const dotenv = require("dotenv");  
const path = require("path");

* nodemailer: A module for Node.js applications to send emails.
* fs: File system module to read and write files.
* dotenv: Module to load environment variables from a .env file.
* path: Module to handle and transform file paths.

1. **Load Environment Variables:**

dotenv.config();

* Loads environment variables from a .env file into process.env.

1. **Extract SendGrid API Key from Environment Variables:**

const { SENDGRID\_API\_KEY } = process.env;

* Extracts the SENDGRID\_API\_KEY from the environment variables.

1. **Define Function to Replace Content Placeholders:**

function replaceContent(content, creds) {  
 let allkeysArr = Object.keys(creds);  
 allkeysArr.forEach(function (key) {  
 content = content.replace(`#{${key}}`, creds[key]);  
 });  
 return content;  
}

* replaceContent(content, creds): This function replaces placeholders in the email template with actual values from the creds object.
  + content: The email template content.
  + creds: An object containing key-value pairs to replace in the template.
  + Iterates over each key in the creds object and replaces the placeholder in the content with the corresponding value.

1. **Define the EmailHelper Function:**

async function EmailHelper(templateName, reciverEmail, creds) {  
 try {  
 const templatePath = path.join(\_\_dirname, "email\_templates", templateName);  
 let content = await fs.promises.readFile(templatePath, "utf-8");

* EmailHelper(templateName, reciverEmail, creds): This asynchronous function handles reading the email template, replacing placeholders, and sending the email.
  + templateName: Name of the email template file.
  + reciverEmail: Recipient’s email address.
  + creds: Object containing key-value pairs for the email content.
  + Constructs the path to the email template using path.join and reads the template file asynchronously using fs.promises.readFile.

1. **Prepare Email Details:**

const emailDetails = {  
 to: reciverEmail,  
 from: "mrinal.bhattacharya@scaler.com", // Change to your verified sender  
 subject: "Mail from ScalerShows",  
 text: `Hi ${creds.name} this your reset otp ${creds.otp}`,  
 html: replaceContent(content, creds),  
};

* emailDetails: Object containing the email details:
  + to: Recipient’s email address.
  + from: Sender’s email address (must be verified with the email service provider).
  + subject: Subject of the email.
  + text: Plain text version of the email.
  + html: HTML version of the email with placeholders replaced using replaceContent.

1. **Configure Transport Details for SendGrid:**

const transportDetails = {  
 host: "smtp.sendgrid.net",  
 port: 587,  
 auth: {  
 user: "apikey",  
 pass: SENDGRID\_API\_KEY,  
 },  
};

* transportDetails: Object containing the transport details for SendGrid:
  + host: SMTP server address for SendGrid.
  + port: Port number (587 for TLS).
  + auth: Authentication details:
    - user: Username for SendGrid (fixed as “apikey”).
    - pass: SendGrid API key for authentication.

1. **Create Transporter and Send Email:**

const transporter = nodemailer.createTransport(transportDetails);  
 await transporter.sendMail(emailDetails)  
 console.log("email sent")  
 } catch (err) {  
 console.log(err)  
 }  
}

* Creates a transporter object using nodemailer.createTransport with the specified transport details.
* Uses the transporter to send the email with transporter.sendMail.
* Logs “email sent” if successful, and catches and logs any errors.

1. **Export EmailHelper:**

module.exports = EmailHelper;

* Exports the EmailHelper function so it can be used in other modules.

### Summary:

* This code sets up an email sending utility using nodemailer and SendGrid.
* It reads an email template from the file system, replaces placeholders with actual values, and sends the email.
* Environment variables are used to securely store the SendGrid API key.
* The EmailHelper function handles the entire process asynchronously, providing a streamlined way to send templated emails.

In your userRoute.js file in the server side add these two routes for forget password and reset Password

title : Email Template For OTP description : duration : 900 card\_type : cue\_card

Now let’s create our Email Template for OTP

Create a email\_templates folder in your utils folder and inside it create a file by the name otp.html

otp.html

<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <meta charset="UTF-8" />  
 <title>Email Verification OTP</title>  
 <style>  
 /\* Inline styles for email compatibility \*/  
 body {  
 font-family: Arial, sans-serif;  
 margin: 0;  
 padding: 0;  
 background-color: #f4f4f4;  
 }  
  
 .container {  
 width: 100%;  
 max-width: 600px;  
 margin: 0 auto;  
 padding: 20px;  
 background-color: #ffffff;  
 border-radius: 8px;  
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);  
 }  
  
 .header {  
 text-align: center;  
 margin-bottom: 20px;  
 }  
  
 table {  
 width: 100%;  
 border-collapse: separate;  
 border-spacing: 0;  
 margin-bottom: 20px;  
 background-color: #ffffff;  
 border: 2px dashed #333333;  
 border-radius: 6px;  
 }  
  
 .otp {  
 font-size: 24px;  
 font-weight: bold;  
 text-align: center;  
 padding: 10px;  
 }  
  
 .note {  
 font-size: 14px;  
 color: #666666;  
 text-align: center;  
 }  
 </style>  
 </head>  
  
 <body>  
 <div class="container">  
 <div class="header">  
 <h2>RESET PASSWORD Verification OTP</h2>  
 </div>  
 <h2>Hi #{name}</h2>  
 <div class="content">  
 <p>Your One-Time Password (OTP) for reseting the password :</p>  
 <table>  
 <tr>  
 <td  
 style="border-radius: 6px 6px 0 0; background-color: #f9f9f9; padding: 10px;"  
 >  
 <!-- Top rounded corners -->  
 <div class="otp">  
 <!-- OTP placeholder -->  
 #{otp}  
 <!-- Replace this with the actual OTP generated -->  
 </div>  
 </td>  
 </tr>  
 <tr>  
 <td  
 style="border-radius: 0 0 6px 6px; background-color: #f9f9f9; padding: 10px;"  
 >  
 <!-- Bottom rounded corners -->  
 <p class="note">Please use this OTP to reset your password.</p>  
 </td>  
 </tr>  
 </table>  
 </div>  
 </div>  
 </body>  
</html>

this is a basic tempalte in this format the otp email will be sent

title : Creating Routes for Forget and reset Password description : duration : 900 card\_type : cue\_card

Now in your userRoutes at the backend we will create two routes userRoutes.js

// forgot password  
  
//Function for otp generation  
  
const otpGenerator = function () {  
 return Math.floor(Math.random() \* 10000 + 90000);  
};  
  
router.patch("/forgetpassword", async function (req, res) {  
 try {  
 /\*\*\*\*  
 \* 1. You can ask for email  
 \* 2. check if email is present or not  
 \* \* if email is not present -> send a response to the user(user not found)  
 \* 3. if email is present -> create basic otp -> and send to the email  
 \* 4. also store that otp -> in the userModel  
 \* 5. to avoid that collison  
 \* response -> unique url with id of the user and that will form your reset password  
 \*  
 \* \*\*\*/  
 if (req.body.email == undefined) {  
 return res.status(401).json({  
 status: "failure",  
 message: "Please enter the email for forget Password",  
 });  
 }  
 // find the user -> going db -> getting it for the server  
 let user = await User.findOne({ email: req.body.email });  
 if (user == null) {  
 return res.status(404).json({  
 status: "failure",  
 message: "user not found for this email",  
 });  
 }  
 // got the user -> on your server  
 const otp = otpGenerator();  
 user.otp = otp;  
 user.otpExpiry = Date.now() + 10 \* 60 \* 1000;  
 // those updates will be send to the db  
 await user.save();  
 res.status(200).json({  
 status: "success",  
 message: "otp sent to your email",  
 });  
 // send the mail to there email -> otp  
 await EmailHelper("otp.html", user.email, {  
 name: user.name,  
 otp: otp,  
 });  
 } catch (err) {  
 res.status(500).json({  
 message: err.message,  
 status: "failure",  
 });  
 }  
 // email  
});  
  
router.patch("/resetpassword", async function (req, res) {  
 // -> otp  
 // newPassword and newConfirmPassword  
 // -> params -> id  
 try {  
 let resetDetails = req.body;  
 // required fields are there or not  
 if (!resetDetails.password == true || !resetDetails.otp == true) {  
 return res.status(401).json({  
 status: "failure",  
 message: "invalid request",  
 });  
 }  
 // it will serach with the id -> user  
 const user = await User.findOne({ otp: req.body.otp });  
 // if user is not present  
 if (user == null) {  
 return res.status(404).json({  
 status: "failure",  
 message: "user not found",  
 });  
 }  
 // if otp is expired  
 if (Date.now() > user.otpExpiry) {  
 return res.status(401).json({  
 status: "failure",  
 message: "otp expired",  
 });  
 }  
 user.password = req.body.password;  
 // remove the otp from the user  
 user.otp = undefined;  
 user.otpExpiry = undefined;  
 await user.save();  
 res.status(200).json({  
 status: "success",  
 message: "password reset successfully",  
 });  
 } catch (err) {  
 res.status(500).json({  
 message: err.message,  
 status: "failure",  
 });  
 }  
});  
  
module.exports = router;

Let’s break down the two routes in detail.

### Route 1: Forget Password (/forgetpassword)

#### Purpose:

This route handles the “Forgot Password” functionality, which generates an OTP (One-Time Password) and sends it to the user’s email to verify their identity.

#### Code Explanation:

1. **Check Email**:
   * The route expects an email in the request body.
   * If the email is not provided, it sends a 401 status with an error message.
2. **Find User**:
   * The route queries the database to find the user associated with the provided email.
   * If the user is not found, it sends a 404 status with an error message.
3. **Generate OTP**:
   * If the user is found, an OTP is generated using the otpGenerator function.
   * This OTP is stored in the user’s database record along with an expiration time (10 minutes from the current time).
4. **Save OTP**:
   * The user record is updated in the database with the new OTP and expiration time.
5. **Send OTP**:
   * An email is sent to the user with the OTP using an email helper function.
   * A success response is sent to the client indicating that the OTP has been sent.
6. **Error Handling**:
   * If any error occurs during this process, a 500 status is returned with the error message.

router.patch("/forgetpassword", async function (req, res) {  
 try {  
 if (req.body.email == undefined) {  
 return res.status(401).json({  
 status: "failure",  
 message: "Please enter the email for forget Password",  
 });  
 }  
  
 let user = await User.findOne({ email: req.body.email });  
 if (user == null) {  
 return res.status(404).json({  
 status: "failure",  
 message: "user not found for this email",  
 });  
 }  
  
 const otp = otpGenerator();  
 user.otp = otp;  
 user.otpExpiry = Date.now() + 10 \* 60 \* 1000;  
 await user.save();  
  
 res.status(200).json({  
 status: "success",  
 message: "otp sent to your email",  
 });  
  
 await EmailHelper("otp.html", user.email, {  
 name: user.name,  
 otp: otp,  
 });  
 } catch (err) {  
 res.status(500).json({  
 message: err.message,  
 status: "failure",  
 });  
 }  
});

### Route 2: Reset Password (/resetpassword)

#### Purpose:

This route handles the resetting of the user’s password using the OTP sent to their email.

#### Code Explanation:

1. **Check Request Body**:
   * The route expects the OTP and the new password in the request body.
   * If either is missing, it sends a 401 status with an error message.
2. **Find User**:
   * The route searches for the user in the database using the provided OTP.
   * If no user is found with the given OTP, it sends a 404 status with an error message.
3. **Verify OTP**:
   * If a user is found, the route checks if the OTP has expired.
   * If the OTP is expired, it sends a 401 status with an error message.
4. **Update Password**:
   * If the OTP is valid, the user’s password is updated with the new password provided in the request body.
   * The OTP and its expiration time are removed from the user’s record.
5. **Save Changes**:
   * The user record is updated in the database with the new password and without the OTP.
6. **Success Response**:
   * A success response is sent to the client indicating that the password has been reset successfully.
7. **Error Handling**:
   * If any error occurs during this process, a 500 status is returned with the error message.

router.patch("/resetpassword", async function (req, res) {  
 try {  
 let resetDetails = req.body;  
 if (!resetDetails.password || !resetDetails.otp) {  
 return res.status(401).json({  
 status: "failure",  
 message: "invalid request",  
 });  
 }  
  
 const user = await User.findOne({ otp: req.body.otp });  
 if (user == null) {  
 return res.status(404).json({  
 status: "failure",  
 message: "user not found",  
 });  
 }  
  
 if (Date.now() > user.otpExpiry) {  
 return res.status(401).json({  
 status: "failure",  
 message: "otp expired",  
 });  
 }  
  
 user.password = req.body.password;  
 user.otp = undefined;  
 user.otpExpiry = undefined;  
 await user.save();  
  
 res.status(200).json({  
 status: "success",  
 message: "password reset successfully",  
 });  
 } catch (err) {  
 res.status(500).json({  
 message: err.message,  
 status: "failure",  
 });  
 }  
});

### Summary:

* **Forget Password Route**: Validates the email, generates and stores an OTP, and sends the OTP to the user’s email.
* **Reset Password Route**: Validates the OTP and new password, updates the user’s password, and clears the OTP from the user’s record.

title : Axios Instance for forgot Password and Reset Password description : duration : 900 card\_type : cue\_card

After this we will create two more Axios Calls for the Reset and Forgot Password Routes

const { axiosInstance } = require("./index");  
  
//Register new User  
  
export const RegisterUser = async (value) => {  
 try {  
 const response = await axiosInstance.post("api/users/register", value);  
 return response.data;  
 } catch (error) {  
 console.log(error);  
 }  
};  
  
// login user  
  
export const LoginUser = async (value) => {  
 try {  
 const response = await axiosInstance.post("api/users/login", value);  
 return response.data;  
 } catch (error) {  
 console.log(error);  
 }  
};  
  
// get current user from the frontend  
  
export const GetCurrentUser = async () => {  
 try {  
 const response = await axiosInstance.get("api/users/get-current-user");  
 return response.data;  
 } catch (error) {  
 console.log(error);  
 }  
};  
  
/// New Axios Instnace for Forget and Reset Password  
  
export const ForgetPassword = async (value) => {  
 try {  
 const response = await axiosInstance.patch(  
 "api/users/forgetpassword",  
 value  
 );  
 return response.data;  
 } catch (error) {  
 console.log(error);  
 }  
};  
  
export const ResetPassword = async (value) => {  
 try {  
 const response = await axiosInstance.patch(  
 "api/users/resetpassword",  
 value  
 );  
 return response.data;  
 } catch (error) {  
 console.log(error);  
 }  
};

title : Creating the Pages for forgot Password and reset Password description : duration : 900 card\_type : cue\_card

Now create a Page for this ForgotPassword Page in the client side

and also create for a reset Password Page

Forget.js component

import React, { useEffect } from "react";  
import { Button, Form, Input } from "antd";  
import { Link, useNavigate } from "react-router-dom";  
import { ForgetPassword, LoginUser } from "../calls/users";  
import { message } from "antd";  
  
function Forget() {  
 const onFinish = async (values) => {  
 console.log(values);  
 try {  
 const response = await ForgetPassword(values);  
 if (response.status === "success") {  
 message.success(response.message);  
 alert("OTP sent to your email");  
 window.location.href = "/reset";  
 } else {  
 message.error(response.message);  
 }  
 } catch (error) {  
 message.error(error.message);  
 }  
 };  
 const navigate = useNavigate();  
 useEffect(() => {  
 if (localStorage.getItem("token")) {  
 navigate("/");  
 }  
 }, []);  
 return (  
 <>  
 <header className="App-header">  
 <main className="main-area mw-500 text-center px-3">  
 <section className="left-section">  
 <h1>Forget Password</h1>  
 </section>  
 <section className="right-section">  
 <Form layout="vertical" onFinish={onFinish}>  
 <Form.Item  
 label="Email"  
 htmlFor="email"  
 name="email"  
 className="d-block"  
 rules={[{ required: true, message: "Email is required" }]}  
 >  
 <Input  
 id="email"  
 type="text"  
 placeholder="Enter your Email"  
 ></Input>  
 </Form.Item>  
  
 <Form.Item className="d-block">  
 <Button  
 type="primary"  
 block  
 htmlType="submit"  
 style={{ fontSize: "1rem", fontWeight: "600" }}  
 >  
 SEND OTP  
 </Button>  
 </Form.Item>  
 </Form>  
 <div>  
 <p>  
 Existing User? <Link to="/login">Login Here</Link>  
 </p>  
 </div>  
 </section>  
 </main>  
 </header>  
 </>  
 );  
}  
  
export default Forget;

Let’s break down the React component step-by-step:

### Import Statements

import React, { useEffect } from "react";  
import { Button, Form, Input, message } from "antd";  
import { Link, useNavigate } from "react-router-dom";  
import { ForgetPassword } from "../calls/users";

### Functional Component: Forget

function Forget() {

This defines the functional component named Forget.

### onFinish Function

const onFinish = async (values) => {  
 console.log(values);  
 try {  
 const response = await ForgetPassword(values);  
 if (response.status === "success") {  
 message.success(response.message);  
 alert("OTP sent to your email");  
 window.location.href = "/reset";  
 } else {  
 message.error(response.message);  
 }  
 } catch (error) {  
 message.error(error.message);  
 }  
};

* **Parameters**: values contains form data.
* **Logging**: Logs form data to the console.
* **API Call**: Calls ForgetPassword with form data.
* **Success Handling**: Displays success message, shows alert, and redirects to the /reset page.
* **Error Handling**: Displays error message using Ant Design’s message.

### Navigation with useNavigate

const navigate = useNavigate();  
useEffect(() => {  
 if (localStorage.getItem("token")) {  
 navigate("/");  
 }  
}, []);

* **Hook**: useNavigate for programmatic navigation.
* **Effect**: Runs once on component mount to check for a token in local storage. If a token is found, it redirects to the home page.

### JSX: Rendered Component

return (  
 <>  
 <header className="App-header">  
 <main className="main-area mw-500 text-center px-3">  
 <section className="left-section">  
 <h1>Forget Password</h1>  
 </section>  
 <section className="right-section">  
 <Form layout="vertical" onFinish={onFinish}>  
 <Form.Item  
 label="Email"  
 htmlFor="email"  
 name="email"  
 className="d-block"  
 rules={[{ required: true, message: "Email is required" }]}  
 >  
 <Input  
 id="email"  
 type="text"  
 placeholder="Enter your Email"  
 ></Input>  
 </Form.Item>  
  
 <Form.Item className="d-block">  
 <Button  
 type="primary"  
 block  
 htmlType="submit"  
 style={{ fontSize: "1rem", fontWeight: "600" }}  
 >  
 SEND OTP  
 </Button>  
 </Form.Item>  
 </Form>  
 <div>  
 <p>  
 Existing User? <Link to="/login">Login Here</Link>  
 </p>  
 </div>  
 </section>  
 </main>  
 </header>  
 </>  
);

* **Layout**: Consists of a header and main area divided into sections.
* **Form**: Uses Ant Design’s Form component with vertical layout. onFinish handles form submission.
* **Form.Item for Email**: A required email input field.
* **Form.Item for Button**: Submit button that sends the OTP.
* **Link**: Provides navigation to the login page for existing users.

### Component Export

export default Forget;

* **Export**: Makes the Forget component available for import in other files.

### Summary

This component handles a “Forget Password” feature:

* It displays a form for users to enter their email.
* On form submission, it calls an API to send an OTP to the user’s email.
* If successful, it shows a success message and redirects the user.
* It also checks if the user is already logged in (via a token in local storage) and redirects to the home page if so.

Reset.js

import React, { useEffect } from "react";  
import { Button, Form, Input } from "antd";  
import { useNavigate } from "react-router-dom";  
import { ResetPassword } from "../calls/users";  
import { message } from "antd";  
  
function Reset() {  
 const onFinish = async (values) => {  
 try {  
 const response = await ResetPassword(values);  
 if (response.status === "success") {  
 message.success(response.message);  
 window.location.href = "/login";  
 } else {  
 message.error(response.message);  
 }  
 } catch (error) {  
 message.error(error.message);  
 }  
 };  
 const navigate = useNavigate();  
 useEffect(() => {  
 if (localStorage.getItem("token")) {  
 navigate("/");  
 }  
 }, []);  
 return (  
 <>  
 <header className="App-header">  
 <main className="main-area mw-500 text-center px-3">  
 <section className="left-section">  
 <h1>Reset Password</h1>  
 </section>  
 <section className="right-section">  
 <Form layout="vertical" onFinish={onFinish}>  
 <Form.Item  
 label="OTP"  
 htmlFor="otp"  
 name="otp"  
 className="d-block"  
 rules={[{ required: true, message: "OTP is required" }]}  
 >  
 <Input  
 id="otp"  
 type="number"  
 placeholder="Enter your otp"  
 ></Input>  
 </Form.Item>  
  
 <Form.Item  
 label="Password"  
 htmlFor="password"  
 name="password"  
 className="d-block"  
 rules={[{ required: true, message: "Password is required" }]}  
 >  
 <Input  
 id="password"  
 type="password"  
 placeholder="Enter your Password"  
 ></Input>  
 </Form.Item>  
 <Form.Item className="d-block">  
 <Button  
 type="primary"  
 block  
 htmlType="submit"  
 style={{ fontSize: "1rem", fontWeight: "600" }}  
 >  
 RESET PASSWORD  
 </Button>  
 </Form.Item>  
 </Form>  
 </section>  
 </main>  
 </header>  
 </>  
 );  
}  
  
export default Reset;

The Reset component is a functional React component that allows users to reset their password by submitting a form with an OTP (One-Time Password) and a new password. Here is an explanation of its key functionalities:

### Key Functionalities

1. **Form Submission Handling (onFinish function)**:
   * When the user submits the form, the onFinish function is triggered. This function is asynchronous and handles the password reset logic.
   * It calls the ResetPassword function with the form values (OTP and new password).
   * If the password reset is successful (response.status === "success"), a success message is displayed using message.success from antd, and the user is redirected to the login page.
   * If the reset is not successful, an error message is displayed using message.error.
2. **Navigation Handling (useNavigate hook)**:
   * The useNavigate hook from react-router-dom is used to programmatically navigate the user.
   * Inside a useEffect hook, it checks if there is a token in local storage (localStorage.getItem('token')). If a token is found, it navigates the user to the home page ("/"), preventing access to the reset password page if the user is already logged in.
3. **Form Rendering**:
   * The component renders a form using Form from antd with a vertical layout.
   * It contains two input fields wrapped in Form.Item:
     + **OTP Input**: A number input field with a label “OTP” and validation rule requiring the OTP to be filled.
     + **Password Input**: A password input field with a label “Password” and validation rule requiring the password to be filled.
   * There is also a submit button styled and configured to submit the form.
4. **Styling and Structure**:
   * The component is structured with a header and main section containing two sections: a left section with a title and a right section with the form.
   * The form is centered within the page using CSS classes.

### Component Breakdown

* **Header**: Contains the title “Reset Password”.
* **Main Area**: Contains two sections:
  + **Left Section**: Displays the title.
  + **Right Section**: Contains the form with OTP and password inputs, and a submit button.
* **useEffect**: Runs on component mount to redirect logged-in users.
* **onFinish**: Handles the form submission and password reset logic.

This component combines form handling, API calls, user feedback, and conditional navigation to provide a complete reset password functionality.

Add both these pages in app.js as Routes

`app.js

import "./App.css";  
  
import { BrowserRouter, Routes, Route } from "react-router-dom";  
import Home from "./pages/Home";  
import Login from "./pages/Login";  
import Register from "./pages/Register";  
import ProtectedRoute from "./components/ProtectedRoute";  
import { useSelector } from "react-redux";  
import Profile from "./pages/Profile";  
import Admin from "./pages/Admin";  
import BookShow from "./pages/BookShow";  
import SingleMovie from "./pages/SingleMovie";  
import Partner from "./pages/Partner";  
import Forget from "./pages/Forget";  
import Reset from "./pages/Reset";  
  
function App() {  
 const { loading } = useSelector((state) => state.loader);  
  
 return (  
 <div className="App">  
 {loading && (  
 <div className="loader-container">  
 {" "}  
 <div className="loader"> </div>{" "}  
 </div>  
 )}  
 <BrowserRouter>  
 <Routes>  
 <Route  
 path="/"  
 element={  
 <ProtectedRoute>  
 <Home />  
 </ProtectedRoute>  
 }  
 />  
 <Route  
 path="/admin"  
 element={  
 <ProtectedRoute>  
 <Admin />  
 </ProtectedRoute>  
 }  
 />  
 <Route  
 path="/profile"  
 element={  
 <ProtectedRoute>  
 <Profile />  
 </ProtectedRoute>  
 }  
 />  
 <Route  
 path="/partner"  
 element={  
 <ProtectedRoute>  
 <Partner />  
 </ProtectedRoute>  
 }  
 />  
 <Route path="/login" element={<Login />} />  
 <Route path="/register" element={<Register />} />  
 <Route path="/forget" element={<Forget />} />  
 <Route path="/reset" element={<Reset />} />  
 <Route  
 path="/movie/:id"  
 element={  
 <ProtectedRoute>  
 <SingleMovie />  
 </ProtectedRoute>  
 }  
 />  
 <Route  
 path="/book-show/:id"  
 element={  
 <ProtectedRoute>  
 <BookShow />  
 </ProtectedRoute>  
 }  
 />  
 </Routes>  
 </BrowserRouter>  
 </div>  
 );  
}  
  
export default App;

title : Sending Tickets in Email description : duration : 900 card\_type : cue\_card

Now we will also implement a freature to send tikcets via email so for that we will create one more template for ticketEmail by the name

```html  
<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <meta charset="UTF-8" />  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
 <title>Movie Ticket</title>  
 <style>  
 body {  
 font-family: Arial, sans-serif;  
 background-color: #f4f4f4;  
 margin: 0;  
 padding: 0;  
 }  
 .email-container {  
 max-width: 600px;  
 margin: 20px auto;  
 background-color: #fff;  
 padding: 20px;  
 border-radius: 10px;  
 box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);  
 }  
 .header {  
 text-align: center;  
 }  
 .header img {  
 max-width: 100%;  
 height: auto;  
 border-radius: 10px;  
 }  
 .ticket-details {  
 margin-top: 20px;  
 }  
 .ticket-details h2 {  
 color: #333;  
 }  
 .details {  
 list-style: none;  
 padding: 0;  
 }  
 .details li {  
 margin: 10px 0;  
 color: #555;  
 }  
 .details li span {  
 font-weight: bold;  
 }  
 .footer {  
 text-align: center;  
 margin-top: 20px;  
 font-size: 12px;  
 color: #999;  
 }  
 </style>  
 </head>  
 <body>  
 <div class="email-container">  
 <div class="ticket-details">  
 <h2>Movie Ticket</h2>  
 <ul class="details">  
 <li><span>Name:</span> #{name}</li>  
 <li><span>Movie:</span> #{movie}</li>  
 <li><span>Theatre:</span> #{theatre}</li>  
 <li><span>Date:</span> #{date}</li>  
 <li><span>Time:</span> #{time}</li>  
 <li><span>Seats NOs:</span> #{seats}</li>  
 <li><span>Amount:</span> Rs. #{amount}</li>  
 <li><span>Transaction ID:</span> #{transactionId}</li>  
 </ul>  
 </div>  
 <div class="footer">Enjoy your movie!</div>  
 </div>  
 </body>  
</html>

Now in our BookingRoutes.js import your Email helper and in the /book-show route add the email sending logic with sending the necessary details

const router = require("express").Router();  
const stripe = require("stripe")(process.env.stripe\_key);  
const authMiddleware = require("../middlewares/authMiddleware");  
const Booking = require("../models/bookingModel");  
const Show = require("../models/showModel");  
const EmailHelper = require("../utils/emailSender");  
  
router.post("/make-payment", async (req, res) => {  
 try {  
 const { token, amount } = req.body;  
 const customer = await stripe.customers.create({  
 email: token.email,  
 source: token.id,  
 });  
  
 const paymentIntent = await stripe.paymentIntents.create({  
 amount: amount,  
 currency: "usd",  
 customer: customer.id,  
 payment\_method\_types: ["card"],  
 receipt\_email: token.email,  
 description: "Token has been assigned to the movie!",  
 });  
  
 // const charge = await stripe.charges.create({  
 // amount: amount,  
 // currency: "usd",  
 // customer: customer.id,  
 // receipt\_email: token.email,  
 // description: "Token has been assigned to the movie!"  
 // });  
  
 const transactionId = paymentIntent.id;  
  
 res.send({  
 success: true,  
 message: "Payment Successful! Ticket(s) booked!",  
 data: transactionId,  
 });  
 } catch (err) {  
 res.send({  
 success: false,  
 message: err.message,  
 });  
 }  
});  
  
// Create a booking after the payment  
router.post("/book-show", async (req, res) => {  
 try {  
 const newBooking = new Booking(req.body);  
 await newBooking.save();  
  
 const show = await Show.findById(req.body.show).populate("movie");  
 const updatedBookedSeats = [...show.bookedSeats, ...req.body.seats];  
 await Show.findByIdAndUpdate(req.body.show, {  
 bookedSeats: updatedBookedSeats,  
 });  
  
 const populatedBooking = await Booking.findById(newBooking.\_id).populate("user")  
 .populate("show")  
 .populate({  
 path: "show",  
 populate: {  
 path: "movie",  
 model: "movies",  
 },  
 })  
 .populate({  
 path: "show",  
 populate: {  
 path: "theatre",  
 model: "theatres",  
 },  
 });  
  
  
 console.log("this is populated Booking", populatedBooking);  
 // console.log(populatedBooking.user.email);  
  
 res.send({  
 success: true,  
 message: "New Booking done!",  
 data: populatedBooking,  
 });  
  
 await EmailHelper("ticketTemplate.html", populatedBooking.user.email, {  
 name: populatedBooking.user.name,  
 movie : populatedBooking.show.movie.title,  
 theatre : populatedBooking.show.theatre.name,  
 date:populatedBooking.show.date,  
 time:populatedBooking.show.time,  
 seats : populatedBooking.seats,  
 amount : populatedBooking.seats.length \* populatedBooking.show.ticketPrice,  
 transactionId : populatedBooking.transactionId,  
  
  
  
 });  
 } catch (err) {  
 res.send({  
 success: false,  
 message: err.message,  
 });  
 }  
});  
  
router.get("/get-all-bookings", authMiddleware, async (req, res) => {  
 try {  
 const bookings = await Booking.find({ user: req.body.userId })  
 .populate("user")  
 .populate("show")  
 .populate({  
 path: "show",  
 populate: {  
 path: "movie",  
 model: "movies",  
 },  
 })  
 .populate({  
 path: "show",  
 populate: {  
 path: "theatre",  
 model: "theatres",  
 },  
 });  
  
 res.send({  
 success: true,  
 message: "Bookings fetched!",  
 data: bookings,  
 });  
 } catch (err) {  
 res.send({  
 success: false,  
 message: err.message,  
 });  
 }  
});  
  
module.exports = router;  
```The email part in this component is handled using the `EmailHelper` function. Here’s a detailed explanation of how it works and what it does:  
  
1. \*\*Email Template\*\*: The `EmailHelper` function is designed to send an email using a specified template file. In this case, the template is `"ticketTemplate.html"`.  
  
2. \*\*Recipient Email Address\*\*: The email is sent to the user who made the booking. The recipient's email address is obtained from `populatedBooking.user.email`.  
  
3. \*\*Dynamic Content for Email\*\*: The email includes various pieces of dynamic content related to the booking. These pieces of content are provided as an object in the third argument of the `EmailHelper` function. Specifically, the content includes:  
  
 - \*\*Name\*\*: The name of the user who made the booking (`populatedBooking.user.name`).  
 - \*\*Movie\*\*: The title of the movie for which the ticket was booked (`populatedBooking.show.movie.title`).  
 - \*\*Theatre\*\*: The name of the theatre where the movie will be shown (`populatedBooking.show.theatre.name`).  
 - \*\*Date\*\*: The date of the show (`populatedBooking.show.date`).  
 - \*\*Time\*\*: The time of the show (`populatedBooking.show.time`).  
 - \*\*Seats\*\*: The seats booked by the user (`populatedBooking.seats`).  
 - \*\*Amount\*\*: The total amount paid for the booking, calculated as the number of seats multiplied by the ticket price (`populatedBooking.seats.length \* populatedBooking.show.ticketPrice`).  
 - \*\*Transaction ID\*\*: The transaction ID of the payment (`populatedBooking.transactionId`).  
  
4. \*\*Sending the Email\*\*: The `EmailHelper` function is an asynchronous function that likely uses a service like Nodemailer or a similar library to send the email. It sends the email using the specified template and fills in the placeholders in the template with the dynamic content provided.  
  
Here is the relevant part of the code for reference:  
  
```javascript  
await EmailHelper("ticketTemplate.html", populatedBooking.user.email, {  
 name: populatedBooking.user.name,  
 movie: populatedBooking.show.movie.title,  
 theatre: populatedBooking.show.theatre.name,  
 date: populatedBooking.show.date,  
 time: populatedBooking.show.time,  
 seats: populatedBooking.seats,  
 amount: populatedBooking.seats.length \* populatedBooking.show.ticketPrice,  
 transactionId: populatedBooking.transactionId,  
});

This ensures that once a booking is successfully made, the user receives a confirmation email with all the relevant details about their booking, enhancing the user experience by providing immediate confirmation and a summary of their booking.