Tunis Business School
University of Tunis

Project Report

Human Resources Management Tool

Yacine Montacer

January 15th 2023

IT325

Introduction

This project is a CRUD (Create, Read, Update, and Delete) Node.js API that can be used to manage employees, projects, and departments in an organization. It utilizes a MySQL database to store and retrieve data, and it also includes features such as user authentication and authorization, password hashing and salting, and email notifications.

One of the main problems that this project can solve is the need for a centralized and efficient way to manage employee information and their interactions with projects and departments. This can be especially useful for larger organizations that have multiple departments and need to keep track of which employees are working on which projects. By having a CRUD API in place, it allows for easy management of employee data, project assignments, and departmental information, making it easier for HR and management to keep track of the workforce and make informed decisions.

This project is a Node.js RESTful API that uses a MySQL database to manage employee information. It aims to provide an easy way for organizations to manage employee data, including personal information, projects and departments they are assigned to, and their login credentials. The API provides endpoints for performing CRUD operations on the employee data and also includes features such as password hashing and salting for added security and email notifications for important actions such as login and personal information updates. The project is built using popular technologies such as Express.js and bcrypt, and is designed to be easily extensible and adaptable to different use cases. In addition, the project uses OAuth2 to authenticate with Google's Gmail API to send emails upon successful login and personal information updates. The project also uses JWT tokens for authentication, and environment variables are used to manage sensitive data such as client secret, client id and redirect URL. The project is a solid foundation for building a comprehensive employee management system for any organization, and can be further developed to include additional features such as reporting and analytics, performance evaluations, and more.

Technical requirements

The technical requirements for this project include having Node.js and npm installed, which are necessary to run the server and install dependencies. A MySQL database management system, or a compatible system, is also required for storing employee, project, and department data. A web server is also necessary to host the application. For security measures, bcrypt and jwt are needed for password hashing and token generation. Email notifications can be sent using nodemailer. Postman can be used for testing the API endpoints. To authorize access to Google's Gmail API, an OAuth2 client and redirect URL is required. Knowledge of JavaScript, SQL, and RESTful API design principles is essential, as well as familiarity with Git and version control practices. It is also important to have a development environment set up, including a code editor and debugging tools. It is recommended to have a clear understanding of the problem that this project is trying to solve and be familiar with the use case scenarios.

Background

Node

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript on the server-side to build fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. It also has a built-in package manager (npm) that allows developers to easily download and use thousands of open-source libraries to perform various tasks such as making HTTP requests, connecting to databases, and more. Node.js is often used to build back-end services, real-time applications, and command-line tools.

RESTful API

An API is a set of definitions and protocols for building and integrating application software. It could be seen as a contract between an information provider and an information user. To interact with a system to retrieve data or perform a function, an API serves as an intermediary to and from the system. An API can serve to share resources and information while maintaining security, control and ensuring authentication.

REST stands for REpresentational State Transfer and was created by computer scientist Roy Fielding. It is a set of architectural constraints to transfer representations of the state of a resource through HTTP in different file formats, most popularly in JSON. For an API to be RESTful it has to fulfill certain criteria, such as that resources requested are identifiable and separate from the representations sent to the client.

CRUD

CRUD stands for Create, Read, Update, and Delete. These are the four basic operations that can be performed on data in a database. The Create operation refers to inserting new data into the database. The Read operation refers to retrieving existing data from the database. The Update operation refers to modifying existing data in the database. The Delete operation refers to removing data from the database. These are the fundamental operations that are supported by most databases and are commonly exposed through an API (Application Programming Interface) for developers to interact with the data stored in the database.

JWT

JWT stands for JSON Web Token. It is a compact, URL-safe means of representing claims to be transferred between two parties. JWT is often used to authenticate users by sending a token to the client after a user successfully logs in. The client can then include this token in the header of subsequent requests to the server, allowing the server to authenticate the user and authorize access to protected resources. JWT's are self-contained and can contain information such as user ID, expiration date and other useful data, allowing for a stateless authentication process. They are usually signed using a secret key to prevent tampering and are a common way to handle authentication and authorization in RESTful APIs.

Environment Variables

Environment variables are values that can be passed to the runtime of a program to configure its behavior. They are typically used to store sensitive information such as passwords, keys, and configuration settings that should not be hard-coded in the source code. Environment variables can be set in the operating system or in the runtime environment of the program. They are accessed by the program using predefined keys or names and are often used to change the behavior of the program without requiring a code change. They are also useful for configuring different environments like development, test, and production.

Salting Hashing

Salting and hashing are two techniques used for securing sensitive data, such as passwords. A salt is a random string of characters that is generated and added to a password before it is hashed. The salt is then concatenated with the password and run through a hashing algorithm, which produces a fixed-length string of characters, also known as the hash.

The purpose of salting is to add an extra layer of security to the password by making it harder for attackers to crack the password, even if they have access to the hash. Because the salt is unique for each password, even if two users have the same password, the hash will be different, making it more difficult for an attacker to use precomputed tables of common passwords to crack the password.

Hashing, on the other hand, is a one-way function that takes an input (or 'message') and returns a fixed-length string of characters, which is the 'message digest'. The same input will always produce the same hash, but even a small change to the input will produce a very different hash.

bcrypt is a popular hashing algorithm that's designed specifically for storing passwords. It's an adaptive algorithm, which means that it's designed to be slow, making it harder for attackers to crack a password using brute force attacks. It also automatically handles the salting process, so you don't have to worry about adding the salt to the password yourself.

System Architecture

SQL DBMS

employeeDB1 contains three tables: employee, project, and department. The employee table has fields such as EmpID, FirstName, LastName, email, phone, and password which are used to store and manage information about employees such as their identification number, name, email address, phone number, and password. The project table has fields such as ProjectID, ProjectName, StartDate, EndDate, and EmpID which are used to store and manage information about projects such as their identification number, name, start date, end date and the identification number of the employee who is responsible for the project. The department table has fields such as DeptID, DeptName, and ManagerID which are used to store and manage information about departments such as their identification number, name and the identification number of the manager who is responsible for the department.

The tables in the database are linked together through the use of foreign keys. The foreign keys are used to establish a link between the data stored in one table and the data stored in another table. For example, the 'employee' table has a foreign key 'EmpID' that references the 'EmpID' primary key in the 'project' table. This establishes a link between the employee and the projects they are assigned to. Similarly, the 'project' table has a foreign key 'DeptID' that references the 'DeptID' primary key in the 'department' table. This establishes a link between the project and the department it belongs to. By using these foreign keys, the database can easily retrieve and manage the related data stored in different tables, allowing for efficient data management.

Administrator Token

The login token mechanism for the administrator works by verifying the user's credentials when they attempt to log in. When the administrator makes a request to the login endpoint with their email and password, the server checks the database to see if a user with that email exists and if their provided password matches the hashed version stored in the database. If the email and password match, the server creates a JSON Web Token (JWT) that contains the user's ID and a secret key. This token is then sent back to the administrator in the response and is used to authenticate future requests. The administrator must include this token in the headers of any requests that require authentication. The server then verifies the token by checking if it is valid and has not expired. If the token is valid, the request is processed, otherwise, the server sends an error message.

Employee Token

When an employee attempts to log in, their email and plain text password are sent as a request to the server. The server then retrieves the employee's hashed password from the database using the provided email. The plain text password is then hashed using bcrypt, and the resulting hash is compared to the hashed password retrieved from the database. If the two hashes match, the employee's login is considered successful. The server then generates a JSON Web Token (JWT) using the employee's EmpID as the payload and a secret key as the signing key. This JWT is then sent back to the client as a response, and the client can use this token for subsequent requests to the server to authenticate the employee.

CRUD operations

The index.js file is the main entry point of the application and it contains the logic for all the routes that have been defined for the CRUD operations on the employee, project, and department data. The file starts by requiring all the necessary modules and dependencies, including the Express.js framework for creating the server, the MySQL module for connecting to the database, the bcrypt and jwt modules for password hashing and token generation, the nodemailer module for sending email notifications, and the OAuth2 and googleapis modules for authorizing access to the Gmail API.

The file defines the various routes for the CRUD operations on the employee, project, and department data. For example, the route for creating a new employee uses the Express.js post method to handle the HTTP post request, and it uses the MySQL query method to insert the new employee data into the database. Similarly, the route for updating an existing employee uses the put method to handle the HTTP put request and the MySQL query method to update the existing employee data in the database.

The login route is used for employee authentication, it takes the email and password from the request body, it then uses the MySQL query method to retrieve the hashed password from the database and compares it to the plain text password entered by the user. If the comparison is successful, the route generates a JSON web token and sends it as the response. If the comparison is not successful, the route sends a 401 Unauthorized status code with an error message.

The index.js file also includes a middleware function for verifying the JSON web token for routes that require authentication. This function uses the jwt.verify method to decode the token and check that it has not expired and that it was signed with the correct secret key. If the token is valid, the request is allowed to continue to the next middleware function or route handler, otherwise, the function sends a 403 Forbidden status code with an error message.

In summary, the index.js file is the main file of the project and it contains all the routes and logic for the CRUD operations on the employee, project, and department data. It also includes the login and token verification mechanism, as well as the email notifications mechanism.

Implementations

Admin Login

```
//Administrator login, Access token generation and refreshment
const username = USERNAME_ADMIN;
const password = PASSWORD_ADMIN;
//const username = "yacine_montacer";
//const password = "kScJM2Hf5_TV?hN-";
app.post("/", (req, res, next) => {
  let p_username = req.body.username;
  let p_password = req.body.password;
  if (p_username == username && p_password == password) {
    var token = jwt.sign(
      { username: username },
      "secretkey",
      { expiresIn: "15m" },
      (err, token) => {
        res.send({
          ok: true,
          token: token,
          message: "Login successful",
        });
      }
    );
    // Refresh token after 10 minutes
    setTimeout(() => {
      var newToken = jwt.sign(
        { username: username },
        "secretkey",
        { expiresIn: "15m" }
      );
      res.send({
        ok: true,
        token: newToken,
        message: "Token refreshed",
      });
    }, 600000);
  } else {
    res.send({
      ok: false,
      message: "Username or password incorrect",
    });
  }
});
}
```

Days off endpoint

```
//Book days off for an employee
app.get('/check-days-off', (req, res) => {
  const EmpID = req.body.EmpID;
  const daysOff = req.body.daysOff;

  // Check if employee has enough days off available
  mysqlConnection.query(`SELECT DaysOff FROM employee WHERE EmpID = ${EmpID
    ↪ }`, (error, results) => {
    if (error) {
      res.status(500).send(error);
    } else {
      const employeeDaysOff = results[0].DaysOff;
      if (employeeDaysOff >= daysOff) {
        // Employee has enough days off, so deduct days off taken
        mysqlConnection.query(`UPDATE employee SET DaysOff = DaysOff - ${
          ↪ daysOff} WHERE EmpID = ${EmpID}`, (error) => {
          if (error) {
```

```
          res.status(500).send(error);
        } else {
          res.send(`Successfully deducted ${daysOff} days off from employee
            ↪  with EmpID ${EmpID}.`);
        }
      });
    } else {
      // Employee does not have enough days off
      res.send(`Employee with EmpID ${EmpID} does not have enough days off
        ↪  available.`);
    }
  }
 });
});
}
```

Password Reset

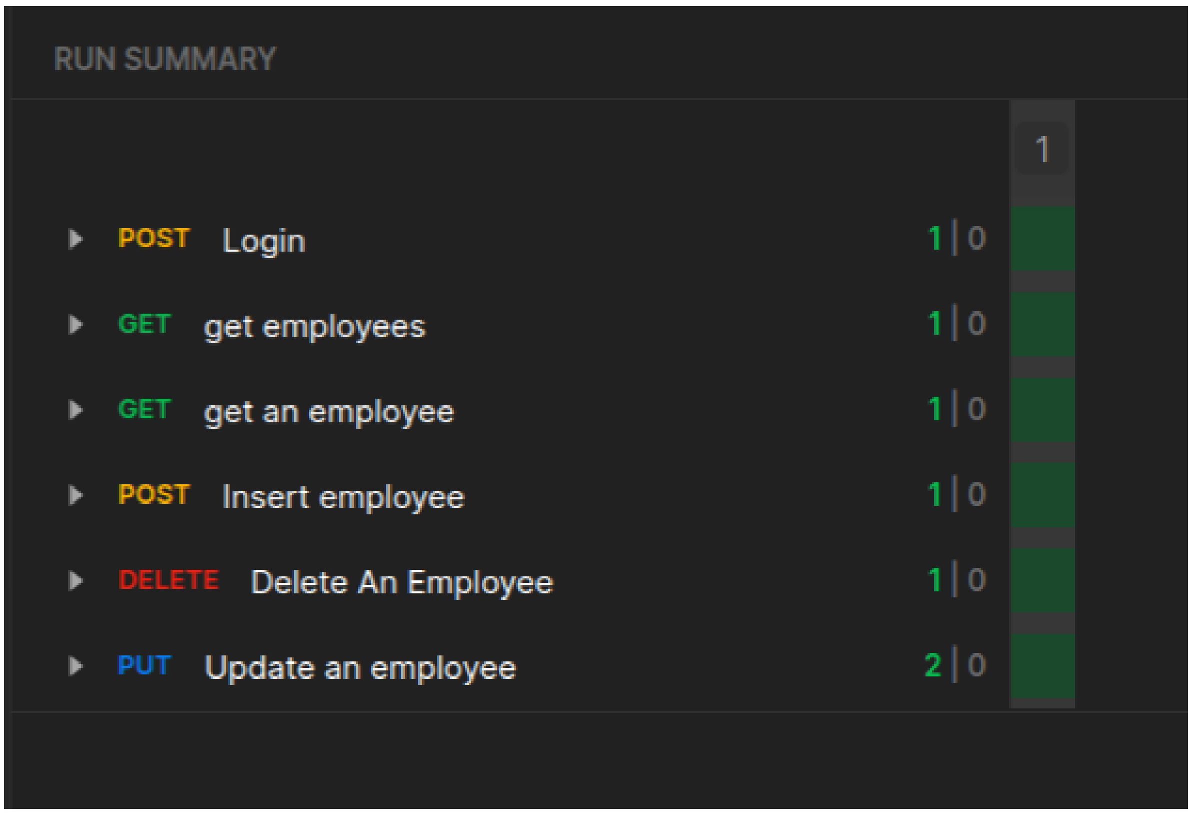
```
//Change password for an employee: Store hashed and salted version on the
   ↪ database
app.post('/set-password', (req, res) => {
  const EmpID = req.body.EmpID;
  const plainPassword = req.body.password;
  const saltRounds = 10;

  bcrypt.genSalt(saltRounds, function(err, salt) {
    bcrypt.hash(plainPassword, salt, function(err, hashedPassword) {
      mysqlConnection.query(`UPDATE employee SET password = '${
        ↪ hashedPassword}' WHERE EmpID = ${EmpID}`, (error, result) => {
        if (error) {
          res.status(500).send(error);
        } else {
          res.status(200).send(`Password for employee with EmpID ${EmpID} has
            ↪  been set successfully!`);
        }
      });
    });
  });
});
}
```

Testing

Postman

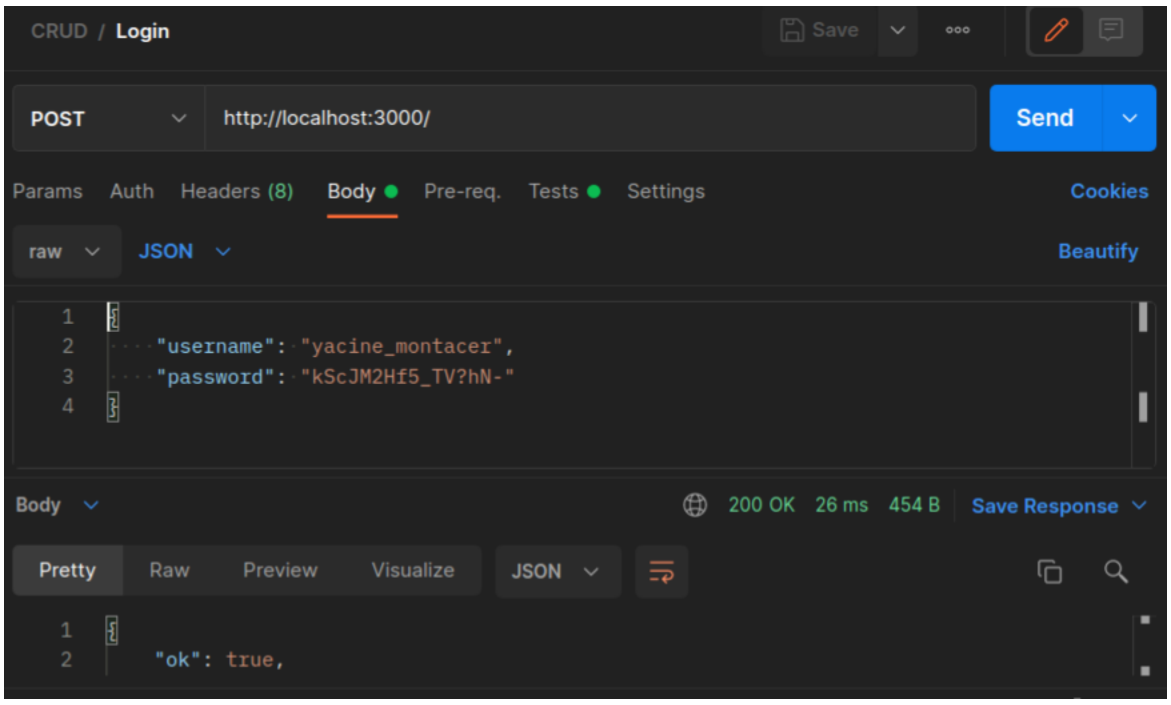
API testing with Postman is a way to ensure that an application's API is working as expected. It involves sending various requests to the API endpoints and asserting that the response received is as expected. One can test the different HTTP methods such as GET, POST, PUT and DELETE to check if the API is able to handle these requests correctly. Additionally, one can test for expected error codes, such as a 404 or 500 status code, to ensure that the API is handling errors correctly.

In the project, testing is done using postman to test the different API endpoints. This includes testing for successful requests and error handling. Additionally, some test cases for the login and change password functionality were also added to ensure that the API was properly handling the creation and validation of tokens. Furthermore, the test cases for the API endpoints that handle CRUD operations on the employee, project, and department data were also added to ensure the API is able to handle these requests correctly.



RUN SUMMARY

			1
POST	Login	1 | 0	🟩
GET	get employees	1 | 0	🟩
GET	get an employee	1 | 0	🟩
POST	Insert employee	1 | 0	🟩
DELETE	Delete An Employee	1 | 0	🟩
PUT	Update an employee	2 | 0	🟩

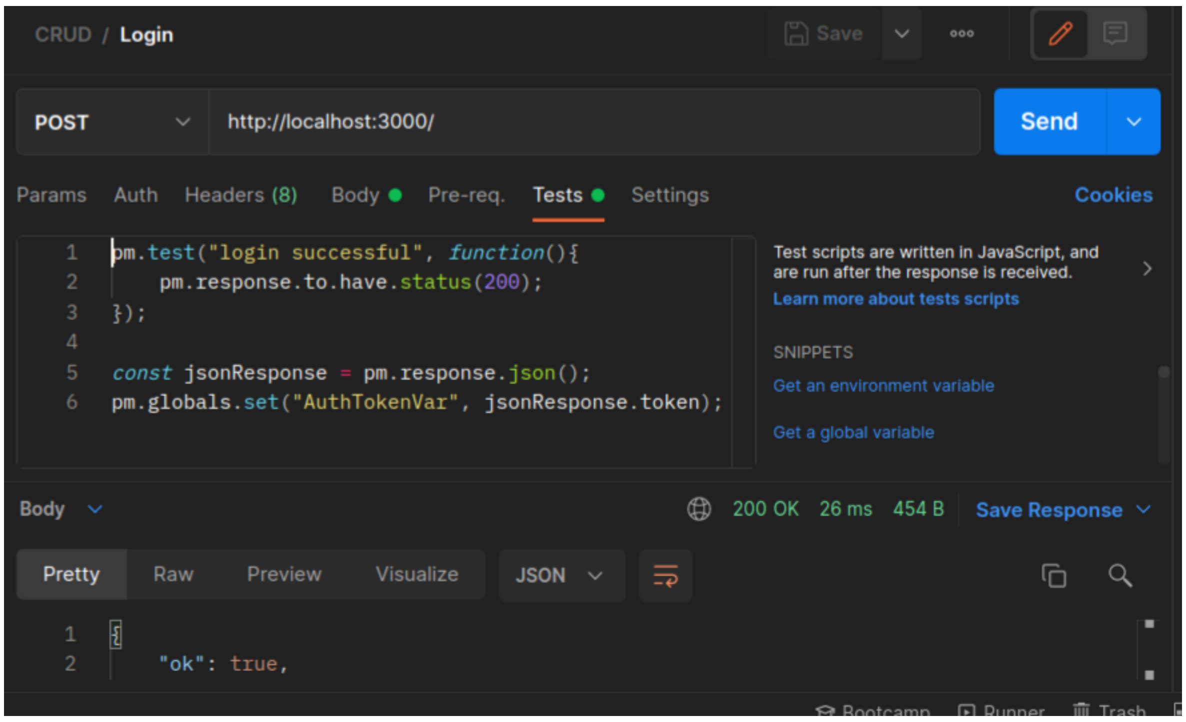

CRUD / Login
Save
POST
Send
Params Auth Headers (8) Body Pre-req. Tests Settings
Cookies
raw JSON
Beautify
Body
Save Response
Pretty Raw Preview Visualize JSON

```
http://localhost:3000/
```

```
{
    "username": "yacine_montacer",
    "password": "kScJM2Hf5_TV?hN-"
}
```

200 OK 26 ms 454 B

```
{
    "ok": true,
```


CRUD / Login
Save
POST
Send
Params Auth Headers (8) Body Pre-req. Tests Settings
Cookies
Learn more about tests scripts
SNIPPETS
Get an environment variable
Get a global variable
Body
Save Response
Pretty Raw Preview Visualize JSON
Bootcamp Runner Trash

```
http://localhost:3000/
```

```
pm.test("login successful", function(){
    pm.response.to.have.status(200);
});

const jsonResponse = pm.response.json();
pm.globals.set("AuthTokenVar", jsonResponse.token);
```

Test scripts are written in JavaScript, and are run after the response is received.

200 OK 26 ms 454 B

```
{
    "ok": true,
```

Version Control

Git

Version control is a method of keeping track of the changes made to a project over time. It allows multiple developers to work on the same project simultaneously while also providing a way to easily revert to previous versions of the project if necessary. In this project, Git is used as the version control system. Git allows developers to create branches for different features or bug fixes, and then merge those branches back into the main branch when the changes are ready to be released. This allows for a streamlined development process and makes it easy to roll back to a previous version of the project if necessary. Additionally, Github is used as the remote repository for the project, allowing for easy collaboration and sharing of the code among team members. The use of version control and Github also facilitates continuous integration and deployment.

Conclusion

In conclusion, this project is a CRUD (Create, Read, Update, and Delete) API for managing employee, project, and department data. It utilizes Node.js, npm, and MySQL for the server and database components, as well as bcrypt for password hashing, jwt for token generation, and nodemailer for sending email notifications. The API has been developed with RESTful principles and has been tested using postman. Additionally, the API uses OAuth2 client credentials and a redirect URL to authorize access to Google's Gmail API for sending email notifications. This project also includes version control practices using Git. Overall, this project demonstrates a solid understanding of API design and development, as well as database management, and it can be used as a starting point for a more extensive HR management software.

Potential future implementation and features

This project is a basic implementation of a CRUD (Create, Read, Update, Delete) API for managing employee, project, and department data using Node.js, MySQL and Express.js. While it serves as a foundation for a more complex and feature-rich employee management system, there are several areas where it could be expanded upon. Some potential features that could be added to this project include:

- Authentication and authorization: The current implementation only has a basic login mechanism for employees and does not include any form of role-based access control. A more robust authentication and authorization system could be implemented using a library such as Passport.js.
- Advanced search and filtering: The current implementation only supports basic filtering by ID. More advanced search and filtering capabilities could be added to allow users to search for employees based on different criteria such as name, email, and department.
- Employee self-service portal: A self-service portal could be added to allow employees to view and update their own personal information, request time off, and view their schedules.
- Reporting and analytics: The current implementation does not include any reporting or analytics capabilities. A reporting and analytics module could be added to allow managers to view data such as employee attendance, project progress, and department performance.
- Mobile App: The current implementation is a web-based application, a mobile app could be added to make it more accessible for employees and managers to access it on the go.

Additionally, the project could be integrated with other systems such as payroll, time tracking, and performance management systems to create a more comprehensive employee management system.