

Billable hours App Documentation

Base-URL: <localhost:8080/billablehour/v1>

To run this app, clone from the repo and open it in an IDE and run, the necessary dependencies will be download since is a maven project.

Repo: <https://github.com/elderjames314/billableHourApp.git>

Swagger Api documentation URL: <localhost:8080/billablehour/v1/ui-swagger>

Database URL: <localhost:8080/billablehour/v1/h2-console> (password=password)

Hint on the database: refer to application.properties

Introduction

Billable hour's app is all about management of bill allocated to company lawyers for the job done, for every lawyer, there is billable rate as per their grade and work did.

For every project they worked on, they have to send the total hours spent to the finance team and in turn, finance can generate receipts to the client accordingly.

The lawyers are to send their timesheet in the following format

- ✓ His employee ID; eg 1004
- ✓ His billable rate per hour as assigned by the company: eg 300/hr
- ✓ Project name worked on eg. MTN
- ✓ The date that the work was carried out.
- ✓ The start time of the project
- ✓ The end time of the project

Table

EMPLOYEES

ID (INT) PK, Name VARCHAR (70), Phone VARCHAR(70)

BILLS

ID (INT) PK, employee_id INT FK, Grade VARCHAR (20), Hours INT (4), Amount DOUBLE,
Remarks TEXT

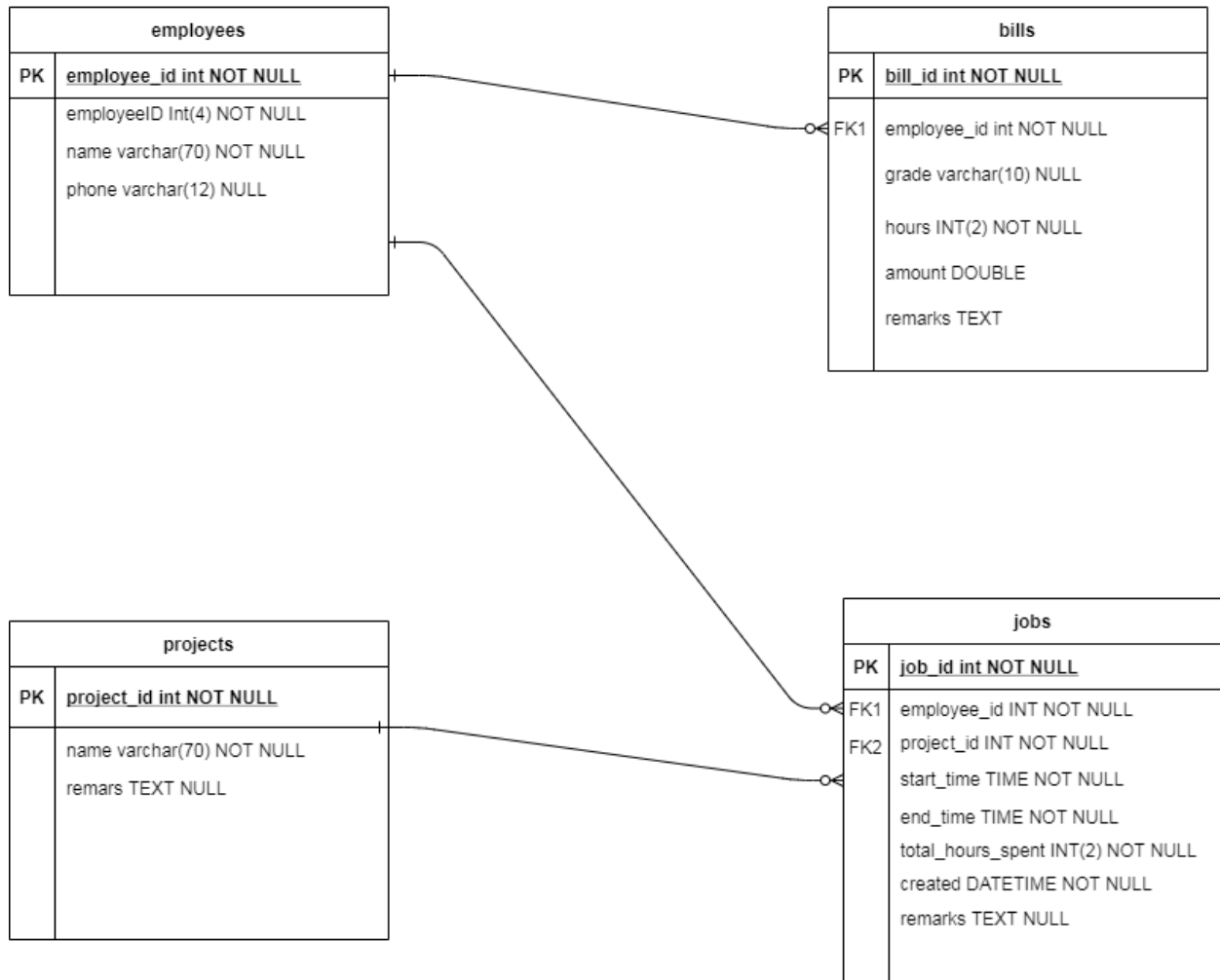
PROJECTS

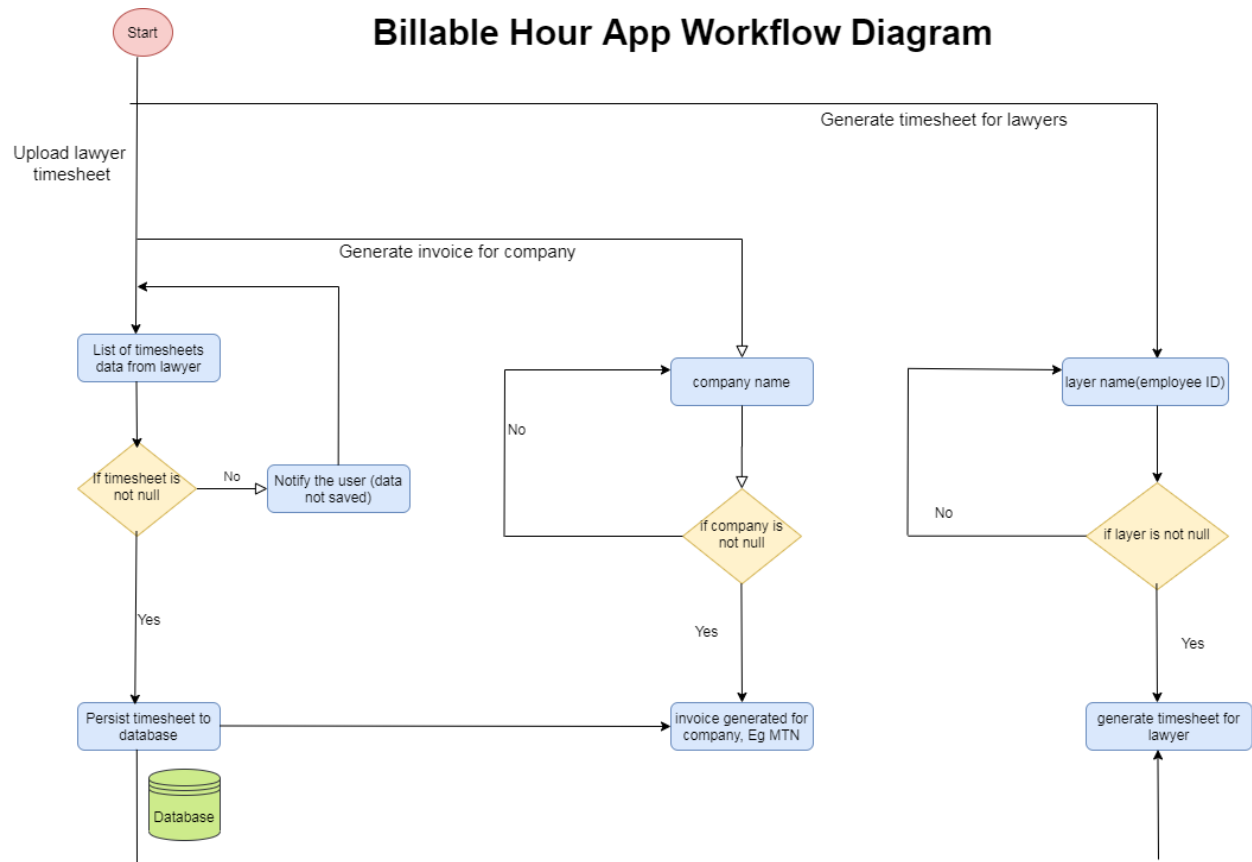
ID (INT) PK, Name VARCHAR (50), Remark TEXT

JOBS

ID(INT) PK, project_id INT FK, employee_id INT FK, Start_time TIME, End_time TIME,
Total_hours_spent INT(4), created DATETIME, Remarks TEXT

Billable Hours App Entity Relationship





Workflow flow is a visual of the process. Can always be the first place to start. Timing of when things happen can tell a lot about the data.

Methodology/Implementation

The Billable hours' App is implemented with Kotlin/SpringBoot. In this project, Singleton design and dependency injection was used which happen to be a default design for springboot framework. For the sake of simplicity, the H2 In-memory database was used however it can be changed to Mysql or Postgres with a line of code in the application.properties, though springboot is an opinionated framework, still, there is room for hands-on configuration.

Project property

- Maven project
- Springboot 2.3.2
- Packaging: Jar
- JDK 11
- Kotlin 1.3.2

Project major dependencies in pom.xml

- Spring web -> To access web goodies
- H2 Database -> the persist store for the application
- Spring Data JPA -> this will handle all the persist logic such as CRUD(Create, Read, Update and Delete) operations
- Thymeleaf -> This is our template engine -> for frontend web app
- Spring DevTool -> this is a dev tool and it serves the purpose or restarts the server upon saving.

Project minor dependencies in pom.xml

- Bootstrap – for front-end web
- JQuery -> javascript framework.
- Swagger – for API documentation
- Rest-Assured – for writing test for Apis endpoints
- JUnit Jupiter – for writing unit test

The project design

The project structure was arranged with the help of packages. Since it is REST API, it will be nice if the first version is put in v1 package, and in the future, there is another version, then it will be created accordingly and there will be a total separation of concern satisfying the S and O of S.O.L.I.D principle. Project or class should have a single responsibility and should be close for modification and open for extension

Project tree:

V1

Backend

Controller

Model

Repository

Service

Resource

Exception

Frontend

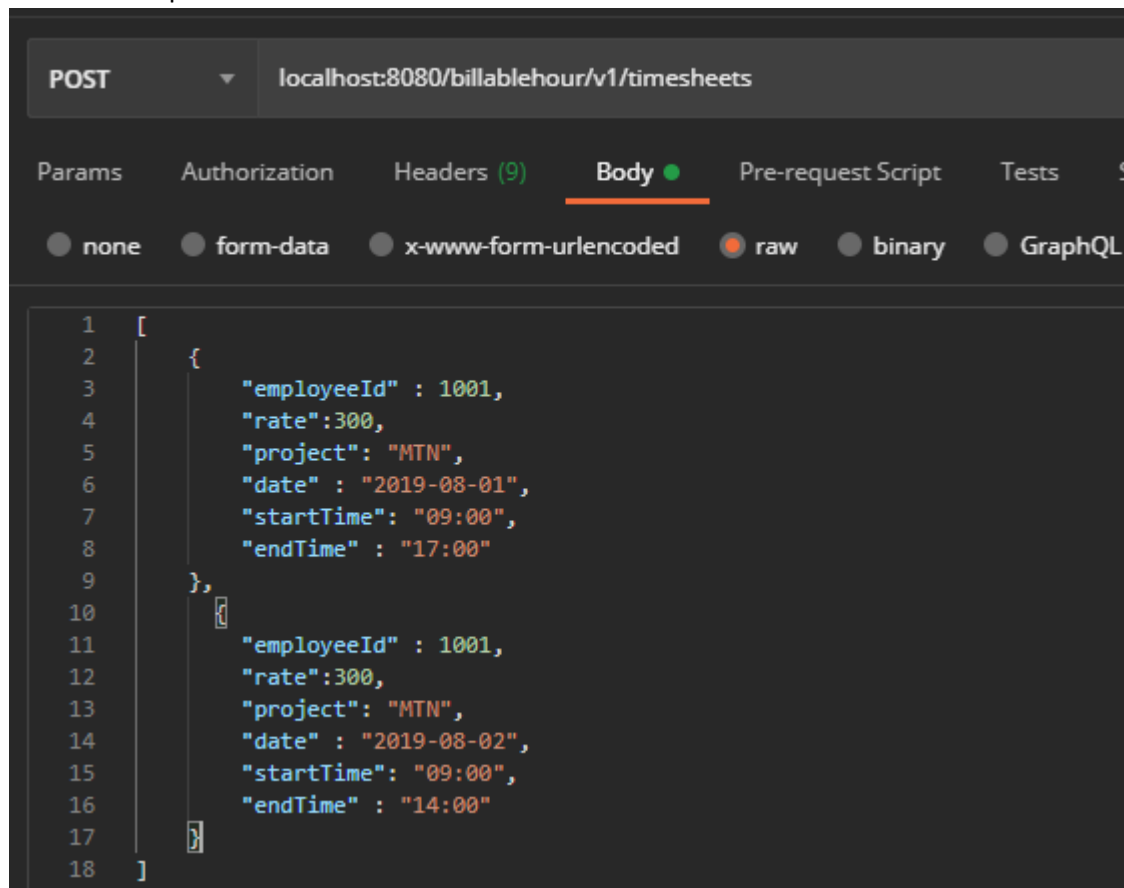
Controller

Controller

Due to the portability of the project, there is an only controller (TimesheetController) as the time of writing this documentation. And it handles four endpoints which are:

1. *POST: localhost:8080/billablehour/v1/timesheets*

The method that handles this request is addTimesheet() and it takes array/list of employee timesheet as parameters.

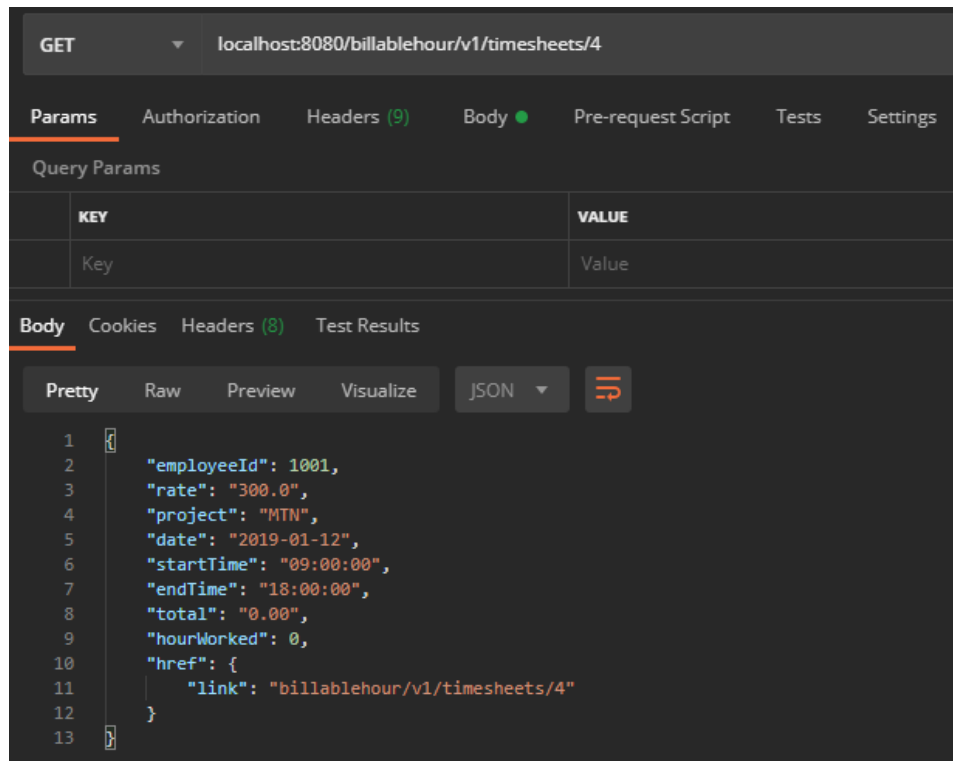


It checks if all the timesheet attributes are given, then it tries to save employee to employee table if new employee, same as bill and project before finally committing to the job table.

2. *GET: localhost:8080/billablehour/v1/timesheets/{employeeId}*

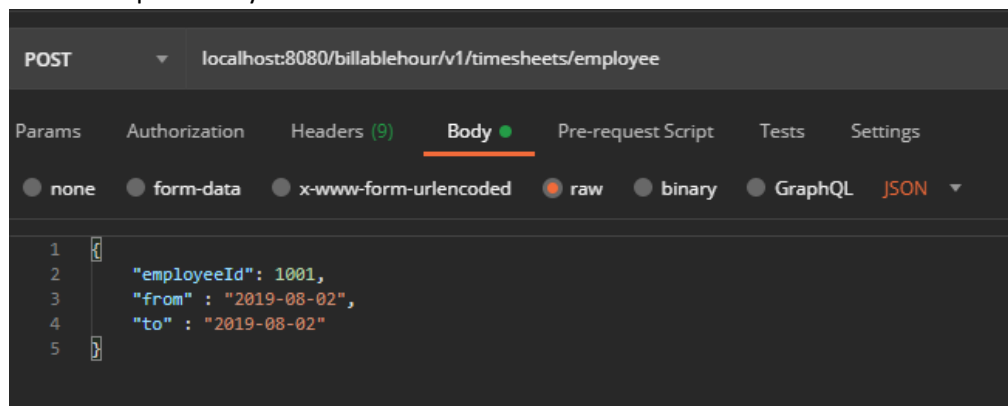
This is an endpoint that responsible for fetching single timesheet/job requests. And it is being handled by getTimesheetById(): it takes employee ID: 1002

Its job is simple, it checks with jobs table as per employee ID passed in and fetched record accordingly if found.



3. *POST: localhost:8080/billablehour/v1/timesheets/employee*

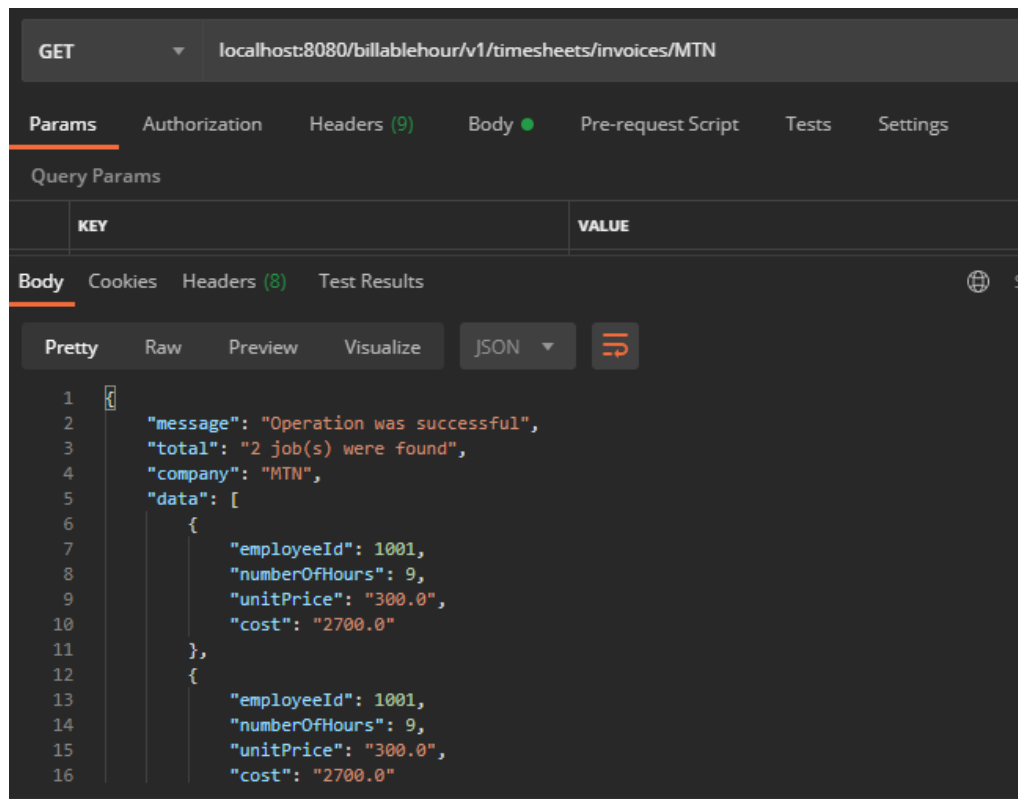
This is the endpoint that handles employee weekly timesheet request. It is a post request and it takes request body as follows:



The request will return all the jobs that this particular employee has done throughout date range specified, technically it should be 7days range (weekly) however it is flexible as it can also take any date range in months and even years.

4. *GET: localhost:8080/billablehour/v1/timesheets/invoices/{company}*

The last but not the least, this endpoint is responsible for generating invoices for company/client worked for. It takes company name as parameter eg MTN and it will return invoices for only MTN.



Repository

This is a data center and each model has an interface that extends JpaRepository (Java persistence) which allows us to perform CRUD (Create, Read, Update, and Delete) operations.

Aside from CRUD operations, there are customized methods that were added.

In employeeRepository, there is findEmployeeByEmployeeId: this return employee record as per employee passed in.

In JobRepository, there following customized methods were added:

1. findJobBetweenCreatedDate: this is methods that fetched data from the jobs table base on the date range given and the employee ID.
2. finJobByProject: this is a method that returns all jobs done for a client base on the client's name passed in.

And in ProjectRepository: there is findProjectByName: which return project entity base on the project name passed in

Service

This is the engine room of the app. technically the number of services depends on how large and complex the project is. Since this is a portable project, there is only one service for this project and is called TimesheetService.

The TimesheetService methods are:

1. AddNewTimesheet() : this take timesheet as argument. When uploading timesheet into the database, the service called this method and in turn, this method called the following:
 - a. saveEmployee(): take the employee as an argument, responsible to persist employee to the database
 - b. saveProject() : take project object as argument. Responsible to save a project to the database
 - c. saveJob(): takes project and timesheet object and it persists both accordingly

And finally, return void (unit) i.e it does not return anything.

2. getSingleTimesheet(): this service take job Id and return job accordingly
3. getWeeklyTimesheetForEmployee() : this method take employeeTimesheetRequest resource which contain employeeId, FromDate and ToDate attributes. And is responsible for fetching employee weekly timesheet.
4. getCompanyInvoice(): this service method takes company name as a parameter and return all jobs done for this company

Model

EmployeeModel

Employee entity that will persist to the database and the attributes are:

- ID: the auto-increment,
- Employee ID: this is the company-defined employee code. It could be 3 or 4 digit eg 3445 as the company policy defined. Please note that employee ID can also be string since it is stated as an integer in the task that is why I used integer.
- Name: Name of the employee
- phone if there is any -> Optional field

BillModel

Bill model, for every company lawyer, there should be a bill allocated to him/her, employee entity will be mapped to bill object, and vice versa. And the attributes are:

- ID: auto increment, this is generated by the database

- Employee object: this will be mapped to the employee entity. That is one to one relationships with the employee entity. An employee has a bill object and a bill belongs to the employee.
- Grade: employee grade or level, this is optional
- Employee hour: hours charged for employee, default will be one hour. If not fill, the system will assume it is one hour.
- Amount: This unit amount concerning hour filled. If two hours are entered, you are expected to fill the amount charge of two hours. This is a compulsory field, as there is no way system can guess that.
- Remark: this attribute is optional, sometimes, you might need to put a note, for instance, “ he is a returnee employee”

JobModel

For every job (that is a lawyer working for clients) the system must capture the details of the job, this is the reason why the billable hour app is created in the first place. The jobs have the following attributes:

- ✓ ID: auto-generated Id
- ✓ Employee: the lawyer that worked on the job. This is one to many relationships. Jobs has many employees and a job belong to the employee
- ✓ Project: Of course we also need to capture the client that the lawyer worked for. Also, the job has many to one relationship with the project entity. The project belongs to a job and job has many projects
- ✓ Start time: the lawyer is expected to fill in the exact time he/she started work
- ✓ End time: also the time she finished the job
- ✓ Hours’ work done: this will be auto-generated by the system. It will deduct start time from the end time and get the difference in hours
- ✓ Created: It is also necessary to capture the date that the job was done for report sake.

ProjectModel

Project entity stores the details of every client that the company has worked for and he has the following attributes:

- ✓ ID: auto-generated fields
- ✓ Name: the name of the clients
- ✓ Remarks: remarks about the project, please note this an optional field.

Resources

I called these resources because I use them to transform request or response data as dim fit. And sometimes if I am working on a large application, I normally group them into packages, but since this is a small project it is okay if there were put in one package. This app has the following resources:

1. CompanyInvoiceDataResource: this is a template for company invoice its attributes are message, total, company, and data(the invoice itself) -> collection of invoice
2. CompanyInvoiceResource: this is a single invoice
3. EmployeeTimesheetRequestResource: this template for fetching weekly employee timesheet and it has employeeID, from date, and to date attributes
4. IncomingRequestTimesheetRequest: this template resource is for single jobs/timesheet transformation
5. And a couple of more, basically they are pretty explanatory as at what they do.

Exception: Springboot exception is good, no doubt but for only development, in production, I like to override some specific exceptions like InvalidConfigurationPropertyValueException(responsible for NotFound), etc and also global Exception. The idea is that in case if anything goes wrong, I will like the application to return a friendly message to the user instead of verbose message generated by springboot.

Application.properties

This is where application settings are been configured, eg database. The default server port is used in this app. Port 8080. This can be changed here. As stated before, the H2 database was used. Also, this can be changed here to any database and the app will still work as expected.

And after running the app, if you want to see the database, it can be accessed in

Localhost: <localhost:8080/billablehour/v1/h2-console>

JDBC: URL: jdbc:h2:~/data/billableDemoData

Username: sa

Password: password

Please note that the above settings can be changed in the application.properties (this file can found in the root directory of the app)

Api Endpoint

1. POST: <localhost:8080/billablehour/v1/timesheets> -> Upload employee timesheet
2. GET: <localhost:8080/billablehour/v1/timesheets/{id}> -> Get single Timesheet
3. POST: <localhost:8080/billablehour/v1/timesheets/employee> -> Get employee weekly timesheet
4. GET: <localhost:8080/billablehour/v1/timesheets/invoices/{companyName}> -> Get company invoice

FRONTEND WEB APP

Html, CSS, and Javascript will be okay for this kind of project if it were to be a large app, would have chosen to use Reactor Vue Frontend Framework. I always like to use the right tool for the right project.

The app contains a Home controller, which has one action method, the method load the index page along with employee collections, projection collection, and today's date.

The frontend web app has two events, get employee timesheet and company invoice. Since their APIs have been written from backend, the frontend consumes those APIs accordingly using Ajax.


1. Get timesheet for employee

To fire this event, select employee name, the start date and since it is a weekly timesheet according to the task, therefore the corresponding weekend date will be automatically generated by the system

Billable Hour App.

The end week date will automatically updated when you select start date.

name1001 ▾

01/12/2019 

2019-01-19

Generate Employee Weekly Timesheet

There are default dataset loaded on upon starting the application. this is intentional however, you can upload more employee timesheets via this post request to: localhost:8080/billablehour/v1/timesheets

Timesheets for : 1001 between 2019-01-12 and 2019-01-19

Total timesheet found: 4 timesheet(s) were found

Employee ID	Billable Rate (per hour)	Project	Date	Start Time	End Time	Hours worked	Total
1001	300.0	MTN	2019-01-12	09:00:00	18:00:00	9	2700.00
1001	300.0	MTN	2019-01-12	09:00:00	18:00:00	9	2700.00
1001	300.0	Fidelity	2019-01-12	09:00:00	14:00:00	5	1500.00
1001	300.0	Fidelity	2019-01-12	09:00:00	14:00:00	5	1500.00

2. Generate company invoice.

To run this event, you will need to select the company name from dropdown control and hit the generate company invoice button. This will connect with backend endpoint (/billablehour/v1/timesheets/invoice) using Ajax and generate company invoice details accordingly

The screenshot shows the 'Billable Hour App' interface. At the top, there's a header with the app name, a company dropdown set to 'CBN', and a 'Generate Company Invoice' button. Below the header, the app title 'Billable Hour App.' is displayed, followed by a note: 'The end week date will automatically updated when you select start date.' There are input fields for 'name' (set to '1001'), a date picker (set to '01/12/2019'), and a date field (set to '2019-01-19'). A 'Generate Employee Weekly Timesheet' button is also present. A message states: 'There are default dataset loaded on upon starting the application, this is intentional however, you can upload more employee timesheets via this post request to: <localhost:8080/billablehour/v1/timesheets>'. Below this, the heading 'Company invoice for : CBN' is shown, followed by the text 'Total invoice found: 2 job(s) were found'. A table displays the invoice details:

Employee ID	Number of hours	Unit price	Cost
1001	9	300.0	2700.0
1001	9	300.0	2700.0
1002	7	350.0	2450.0
1002	7	350.0	2450.0

At the bottom, it says 'Powered by TechHustle Inc'.

The javascript script file responsible for the frontend app can be found in static/main.js all the static files(CSS, js, etc) are located in a static folder under the resources directory

TEST DOCUMENTATION

Tests were arranged and structured as the main project for easy maintenance. JUnit Jupiter and REST – Assured was used. Junit5 for unit testing and REST-Assured for API endpoints testing and Mockito to mock the data.

API endpoints:

1. *CompanyInvoiceTests*: This is a test that handles generating invoices for companies and there are two test cases in this class.
 - a. *getCompanyInvoiceByCompanyName*: Assert the accuracy of the returned data if the actual company name is provided.
 - b. *getCompanyInvoiceByUnknowCompanyNameResultIn404*: Assert that the response returns 404 status code if the company name provided is not found.
2. *GetEmployeeTimesheets*: Test that handles generating of timesheets and it consists of three test cases. The request is employeeID, fromDate, and toDate
 - a. *getEmployeeWeeklyTimesheet*: Asserts that the responses are accurately provided the request is provided.
 - b. *getEmployeeWeeklyTimesheetWithUnknowEmployeeID*: Asserts that the response is null if employee ID is not provided.
 - c. *getEmployeeWeeklyTimesheetWithIncorrectDates*: Assert that the response return 500 status code if dates are incorrect or missing in the parameter.

3. *GetSingleTimesheetDetails*: Response for getting a timesheet details base on the jobID provided. It has two test cases.
 - a. *getSingleTimesheetByJobId*: Assert the accuracy of the data is correct given a jobID.
 - b. *getSingleTimesheetByJobIdNotPresentDatabaseResultIn404*: Return 404 status code if jobID is not found in the database.
4. *UploadLawyerTimesheets*: Testing the uploading timesheets to the database and has two test cases:
 - a. *uploadTimesheetTest*: Assert that response is correct when uploading timesheets given all the necessary parameters
 - b. *uploadTimesheetWithMissingParameterWillResultToExceptionTest*: Ensure that the responses return 500 status code provided that if one parameter is missing or not provided.

Controller

TimesheetControllerTest: Test class that will do unit testing of all the public functions in the controller, and this controller has four functions and all be tested accordingly:

1. *getTimesheetByIdTest* : Assert that the correct data is return given a jobID
2. *addTimesheetTest*: Ensure that if the accurate request is given, timesheets will be uploaded accordingly return the appropriate response.
3. *generateTimesheetForLawyerTest*: Given a request(employeeID, fromDate, and toDate), it will mock data and return appropriate data, this assert all is well provided the request parameter is correct.
4. *generateCompanyInvoiceTest*: unit testing on the function that returns company details given a company name.

Service

TimesheetServiceTests: further unit testing on the service functions, the idea is, there is at least 90% test coverage.

1. *saveProjectTest*: this test cases mock the projectRepository and assert the project is save given the project object.
2. *saveJobTest*: Mock the jobRepository and assert that job save successfully provided job/timesheet, employee and project object are given
3. *saveEmployeeTest*: Test case that responsible for the saving of employees. Assert that employee will be saved given TimeResource object.
4. *saveBillTest*: unit testing on saving bill, assert it is save given rate parameter and employee object
5. *getWeeklyTimesheetForEmployeeTest*: unit testing on getting employee weekly timesheet, assert the right data is return given employeeID, from date, and todate parameter.

6. `getTimesheetTest`: Assert that the correct timesheet details will return given jobID that is in the database
7. `getCompanyInvoiceTest`: Assert the company invoice generated is correct given a company name that is in the database.