MyTD

Dong Wang <wado18gb@student.ju.se>

A Project Work in Development of Server-side Solutions

Jönköping University 2019

Table of Contents

Introduction

Nowadays, more and more people are willing to go to the gym to keep their figure. But sometimes it is not convenient to record the information of each training activity and weight data, for the reason, maybe they forget to do or there is no suitable product to record these data. MyTD will be a good choice for people who always go to gym and want to record information about their training.

MyTD, short for My Training Diary, is a platform that can facilitate for users to stay in shape. On the platform, users can be able to log their training activities and weights, and, they can review this data and follow their progress. This will hopefully not only motivate users to keep training and to stay in shape, but the collected data can also be used for analyzing and finding patterns for illness (e.g. risk of training too much, risk of underweight, etc.) and to warn users about the illnesses before they are hurt.

Overview

MyTD uses SQLite database to store the data of accounts, trainings and weights. Users can use MyTD through a web application, with the website users can sign up for accounts. And then users sign in to their accounts, they can check the information of account and the training and weight data. In addition, users can upload new training or new weight.

Moreover, it is convenient for users to create new training activities on a smartphone application for MyTD. What's more, user can add new weights data after they measure their weights through a smart weighing scale. (*Figure-1*)

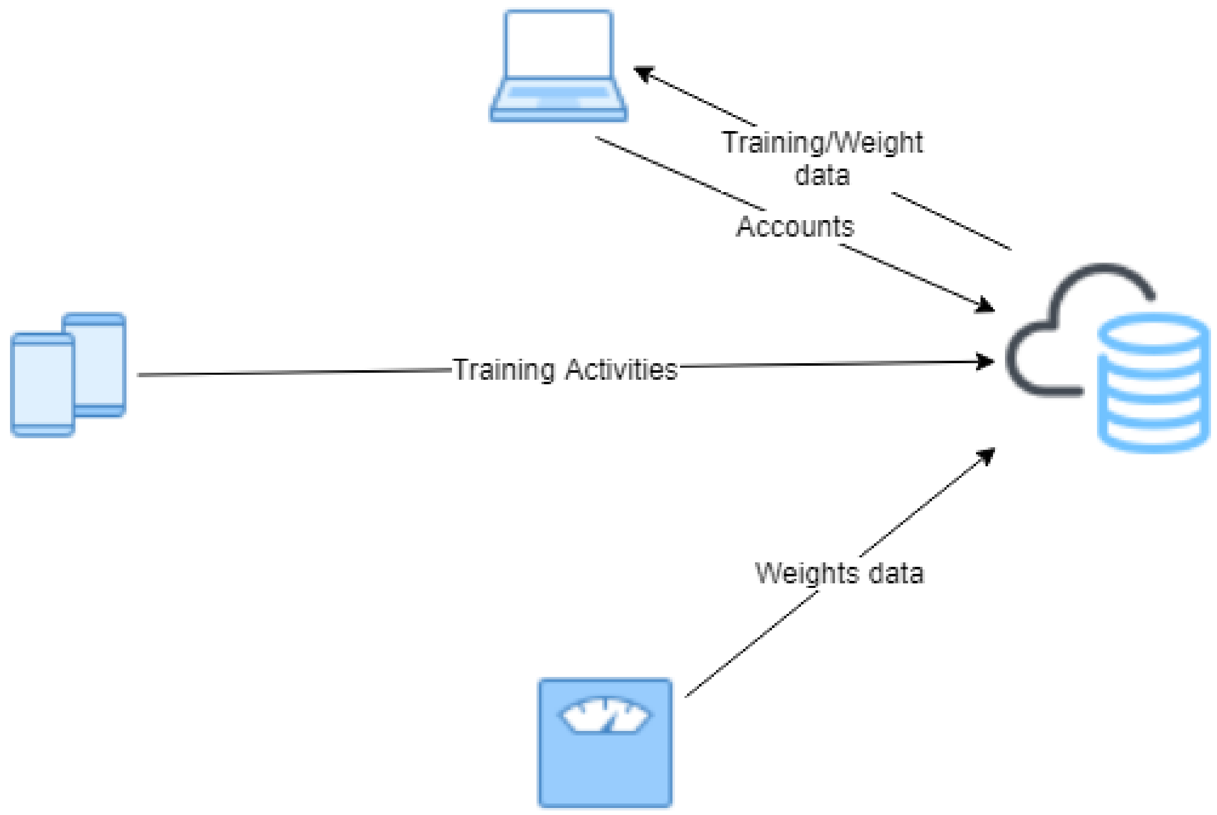


Figure-1 what can end user do?

Resources

MyTD uses SQLite to store data. There are two tables, one is accounts (*Table-1*), the other one is datas (*Table -2*). When the user registers the account, MyTD will store the user's basic information into the accounts table, and users can check it on their account page. Table datas stores the training activities and weights data that users created, users can also check it on MyTD. The relationship between the two tables is shown in *Figure-2.*

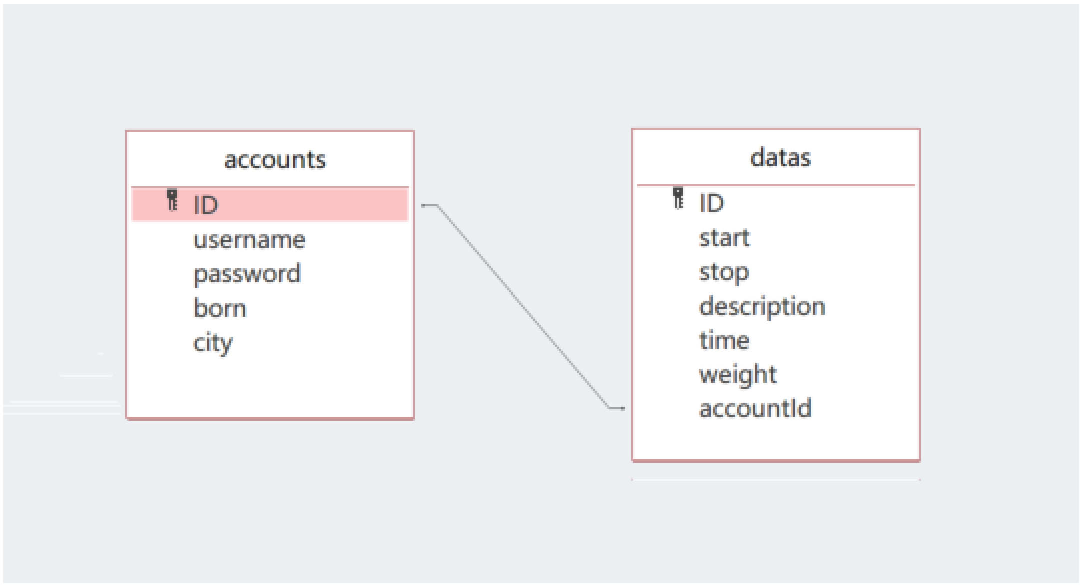


Figure-2 ER diagram

Field	Type	Primary Key	Null
id	int	P	no
username	text		no
password	text		no
born	int		no
city	string		no

Table -1 accounts

Field	Type	Primary Key	Null
id	int	P	no
start	int		no
stop	int		no
description	text		no
time	int		no
weight	int		no
accountId	int		no

Table -2 datas

Web Application

Graphical User Interface

After watching lots of website and according to users' usage habits, the wireframe of MyTD has been designed (*Figure-3*). Cause MyTD focus on to motivate users to keep training and to stay in shape, the theme color of MyTD is teal green. Teal green is associated with fluid communication and clarity. It is serene, calming, and, like other green blue hues, associated with nature and water. So, it can make users feel relaxed and confident to do the training (*Figure-4*). And the "Home", "Accounts" and "Sign In" buttons are on the left of navigation bar, the "Contact" and "About" buttons are on the footer of the website.

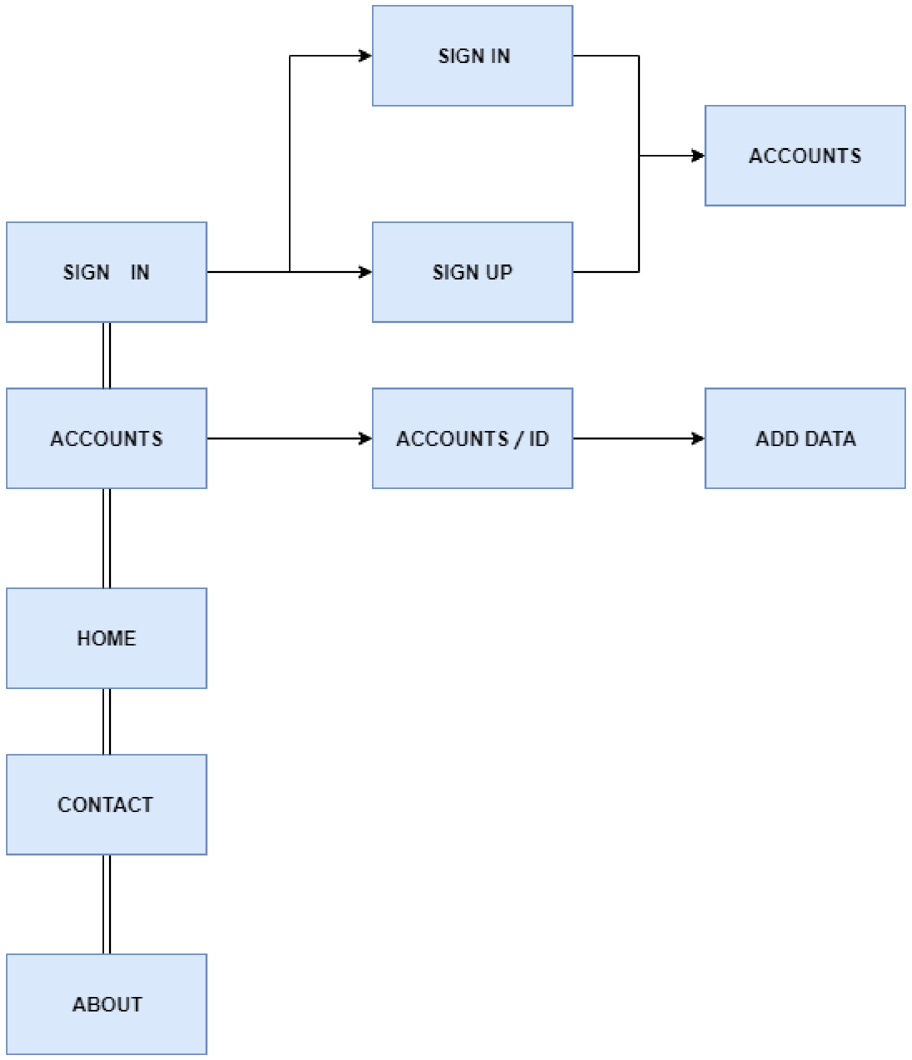
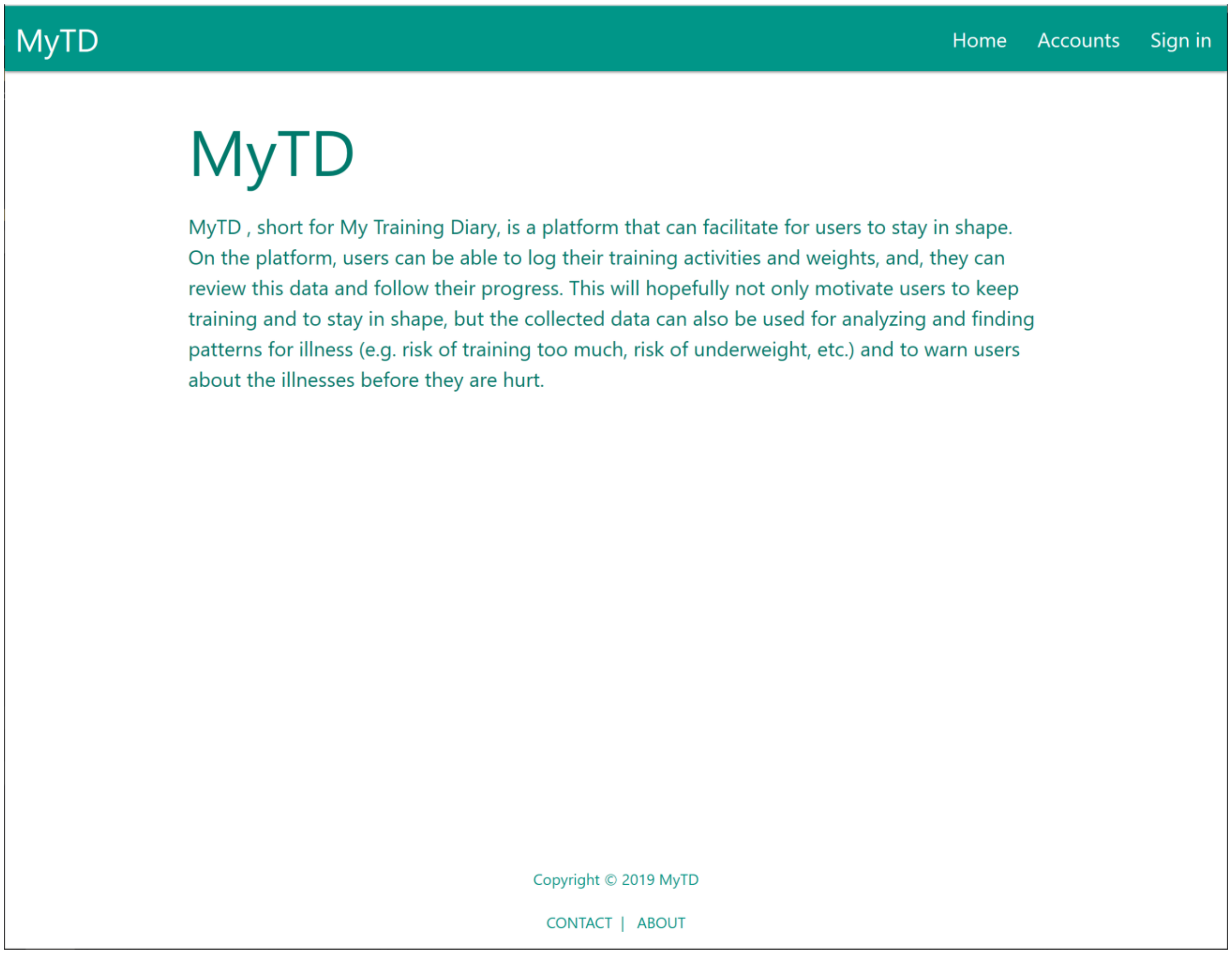


Figure-3 wireframe of MyTD


MyTD

Home Accounts Sign in

MyTD

MyTD , short for My Training Diary, is a platform that can facilitate for users to stay in shape. On the platform, users can be able to log their training activities and weights, and, they can review this data and follow their progress. This will hopefully not only motivate users to keep training and to stay in shape, but the collected data can also be used for analyzing and finding patterns for illness (e.g. risk of training too much, risk of underweight, etc.) and to warn users about the illnesses before they are hurt.

Copyright © 2019 MyTD

CONTACT | ABOUT

Figure-4 MyTD website

Code

Language: JavaScript, Node.js

Frameworks: Express

Libraries: SQLite

Structure: three-layered architecture

A three-layered architecture is a type of software architecture which is composed of three "layers" of logical computing. They are often used in applications as a specific type of client-server system. three-layered architectures provide many benefits for production and development environments by modularizing **the user interface, business logic**, and **data storage layers**.

MyTD used three-layered architecture to structure code (*Figure-5*). Different code is for different layer. In the user interface layer, MyTD used HTML, CSS and JavaScript to build the UI of the website (*Figure-6*). In the business logic layer, MyTD mainly used express to complete the operation of the data layer of the website and the processing of data business logic (*Figure-7*). In the data storage layer, MyTD used SQLite to directly operate the database, create, read, update, delete, on data (*Figure-8*).

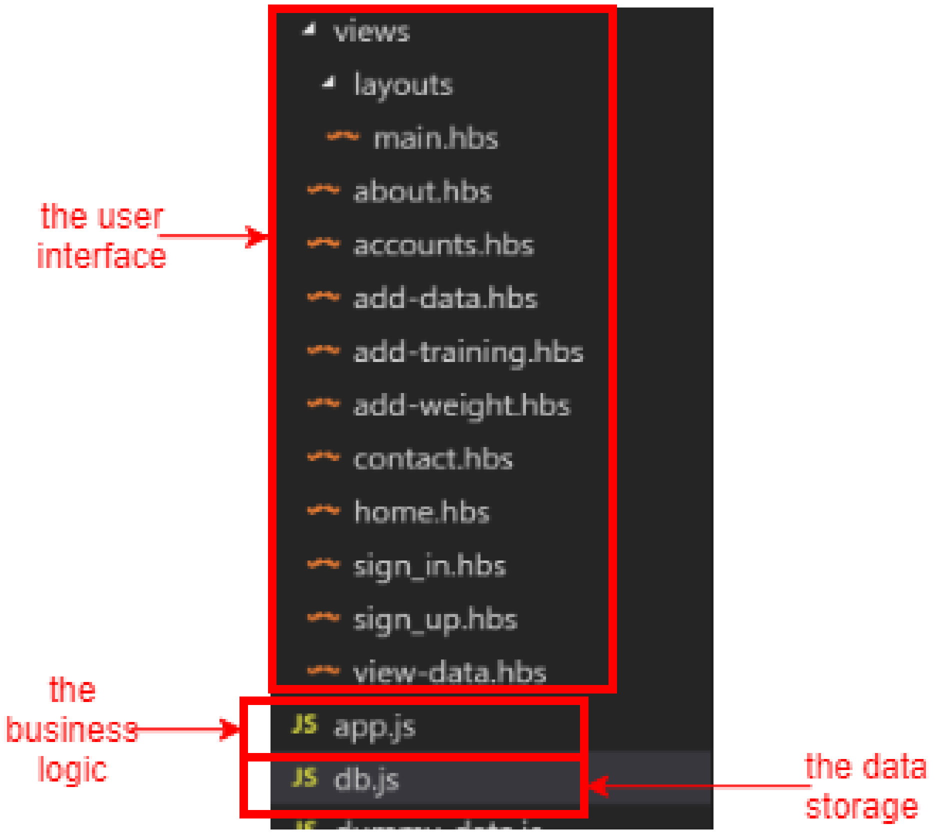
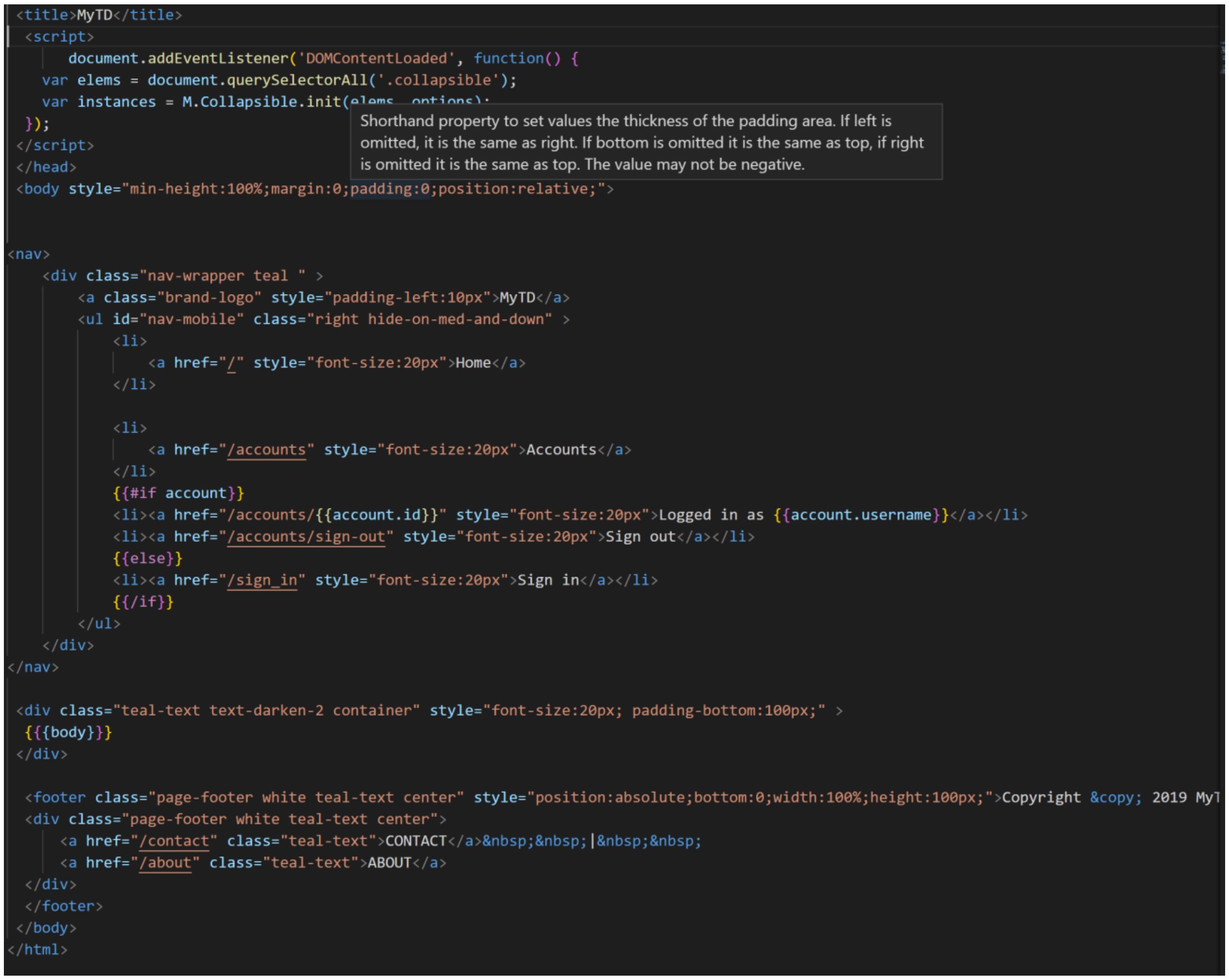


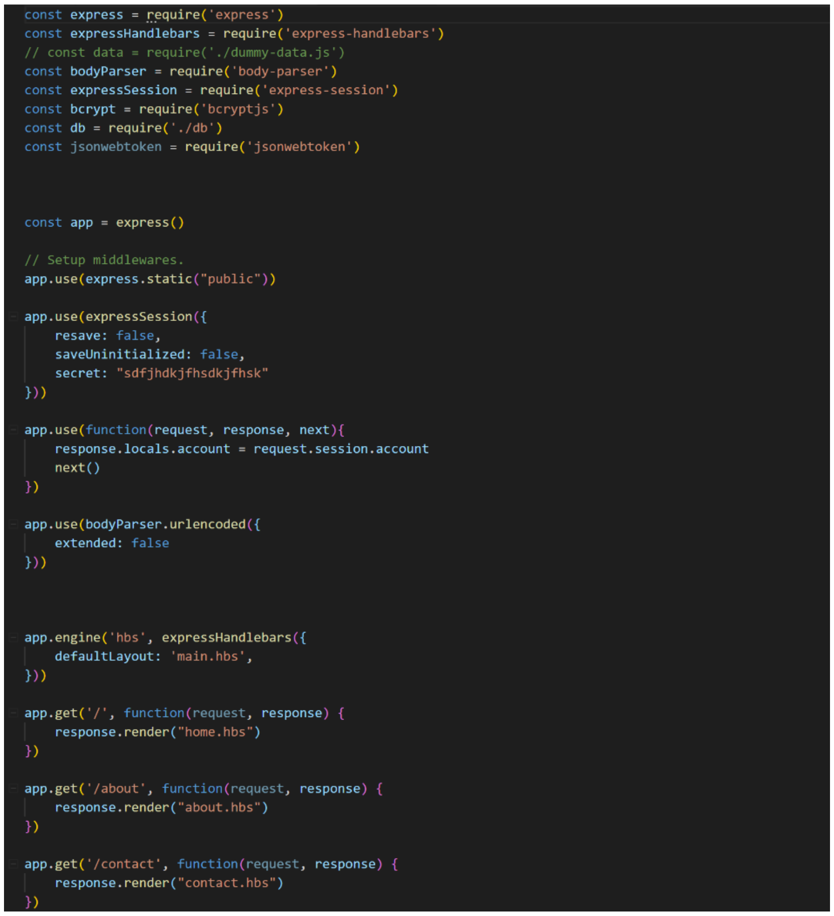
Figure-5 three-layered architecture



```
 <title>MyTD</title>
  <script>
      document.addEventListener('DOMContentLoaded', function() {
    var elems = document.querySelectorAll('.collapsible');
    var instances = M.Collapsible.init(elems, options);
  });
 </script>
 </head>
 <body style="min-height:100%;margin:0;padding:0;position:relative;">


<nav>
    <div class="nav-wrapper teal " >
        <a class="brand-logo" style="padding-left:10px">MyTD</a>
        <ul id="nav-mobile" class="right hide-on-med-and-down" >
            <li>
                <a href="/" style="font-size:20px">Home</a>
            </li>

            <li>
                <a href="/accounts" style="font-size:20px">Accounts</a>
            </li>
            {{#if account}}
            <li><a href="/accounts/{{account.id}}" style="font-size:20px">Logged in as {{account.username}}</a></li>
            <li><a href="/accounts/sign-out" style="font-size:20px">Sign out</a></li>
            {{else}}
            <li><a href="/sign_in" style="font-size:20px">Sign in</a></li>
            {{/if}}
        </ul>
    </div>
</nav>

 <div class="teal-text text-darken-2 container" style="font-size:20px; padding-bottom:100px;" >
  {{{body}}}
 </div>

  <footer class="page-footer white teal-text center" style="position:absolute;bottom:0;width:100%;height:100px;">Copyright &copy; 2019 MyT
  <div class="page-footer white teal-text center">
      <a href="/contact" class="teal-text">CONTACT</a>&nbsp;&nbsp;|&nbsp;&nbsp;
      <a href="/about" class="teal-text">ABOUT</a>
  </div>
  </footer>
 </body>
</html>
```

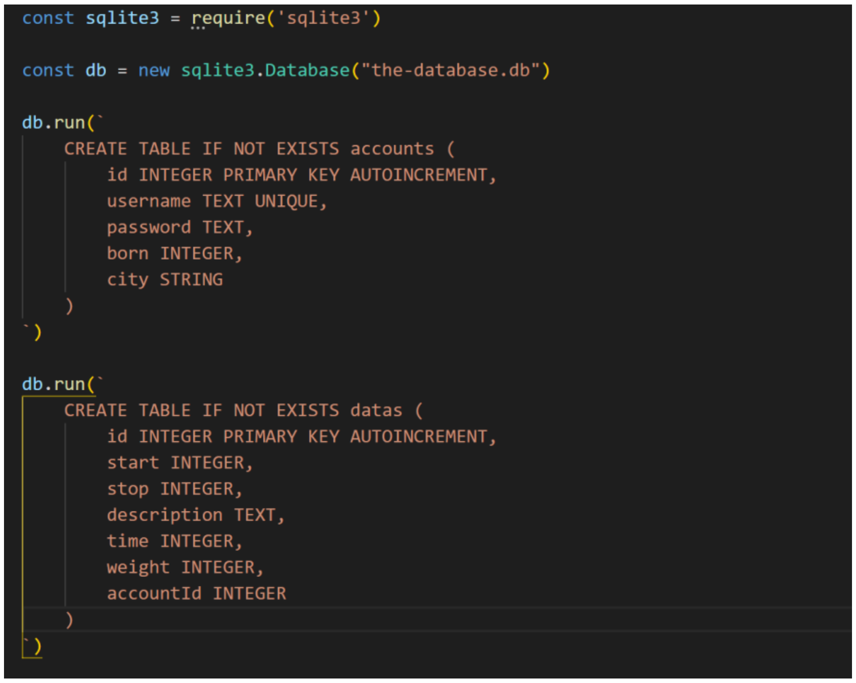
Shorthand property to set values the thickness of the padding area. If left is omitted, it is the same as right. If bottom is omitted it is the same as top, if right is omitted it is the same as top. The value may not be negative.

Figure-6 the user interface layer



```
const express = require('express')
const expressHandlebars = require('express-handlebars')
// const data = require('./dummy-data.js')
const bodyParser = require('body-parser')
const expressSession = require('express-session')
const bcrypt = require('bcryptjs')
const db = require('./db')
const jsonwebtoken = require('jsonwebtoken')



const app = express()

// Setup middlewares.
app.use(express.static("public"))

app.use(expressSession({
    resave: false,
    saveUninitialized: false,
    secret: "sdfjhdkjfhsdkjfhsk"
}))

app.use(function(request, response, next){
    response.locals.account = request.session.account
    next()
})

app.use(bodyParser.urlencoded({
    extended: false
}))



app.engine('hbs', expressHandlebars({
    defaultLayout: 'main.hbs',
}))

app.get('/', function(request, response) {
    response.render("home.hbs")
})

app.get('/about', function(request, response) {
    response.render("about.hbs")
})

app.get('/contact', function(request, response) {
    response.render("contact.hbs")
})
```

Figure-7 the business logic layer

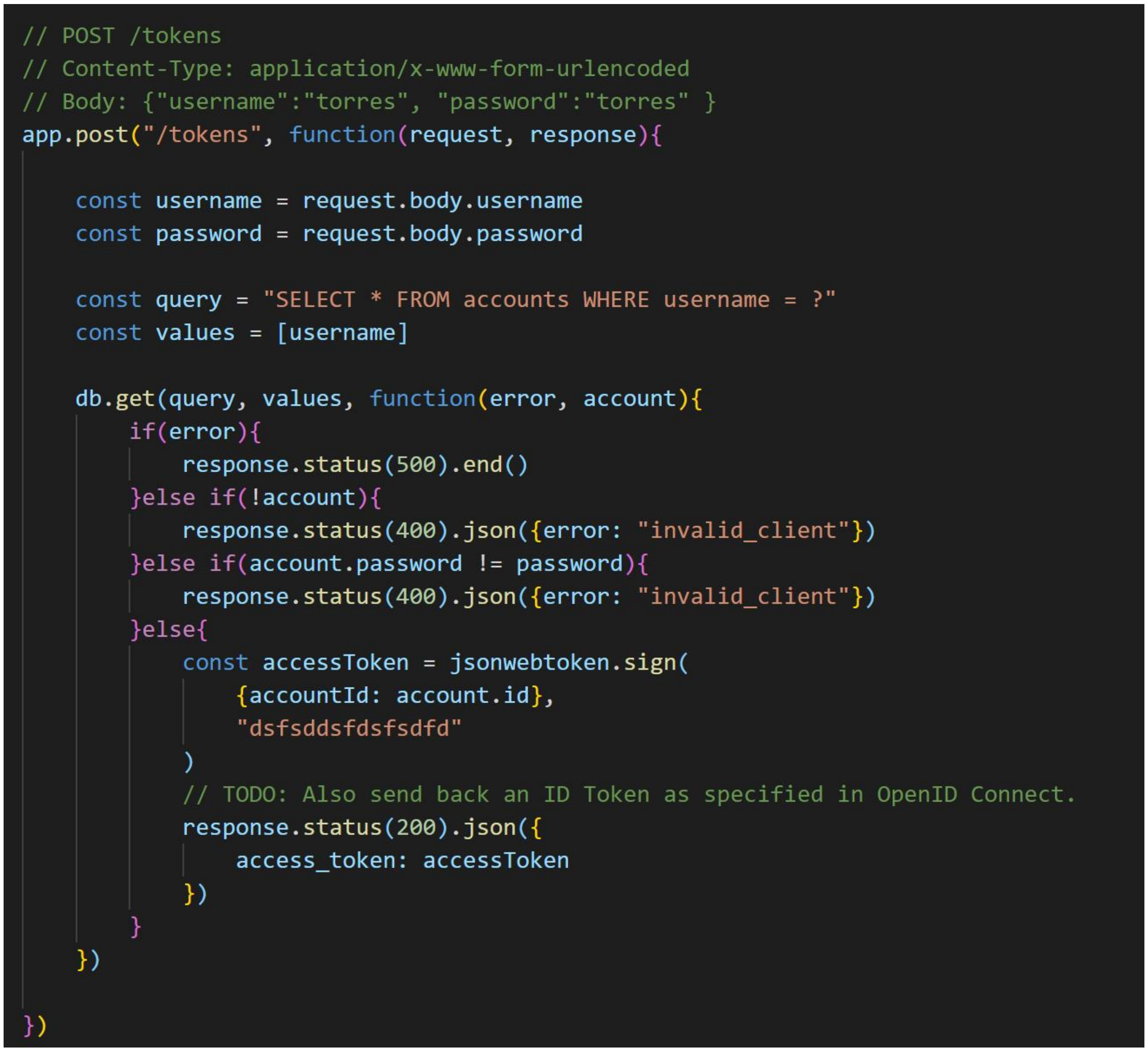


```
const sqlite3 = require('sqlite3')

const db = new sqlite3.Database("the-database.db")

db.run(`
    CREATE TABLE IF NOT EXISTS accounts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE,
        password TEXT,
        born INTEGER,
        city STRING
    )
`)

db.run(`
    CREATE TABLE IF NOT EXISTS datas (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        start INTEGER,
        stop INTEGER,
        description TEXT,
        time INTEGER,
        weight INTEGER,
        accountId INTEGER
    )
`)
```

Figure-8 the data storage layer

REST API

The smart products can communicate with the platform using the web application's REST API. When smart product access server, it can receive the JSON Web Token（JWT） returned by the server, which can be stored in the cookie or stored in local storage. After that, the client will bring this token every time it communicates with the server. And the token is always put in the header information Authorization field of the HTTP request (*Figure-9*).

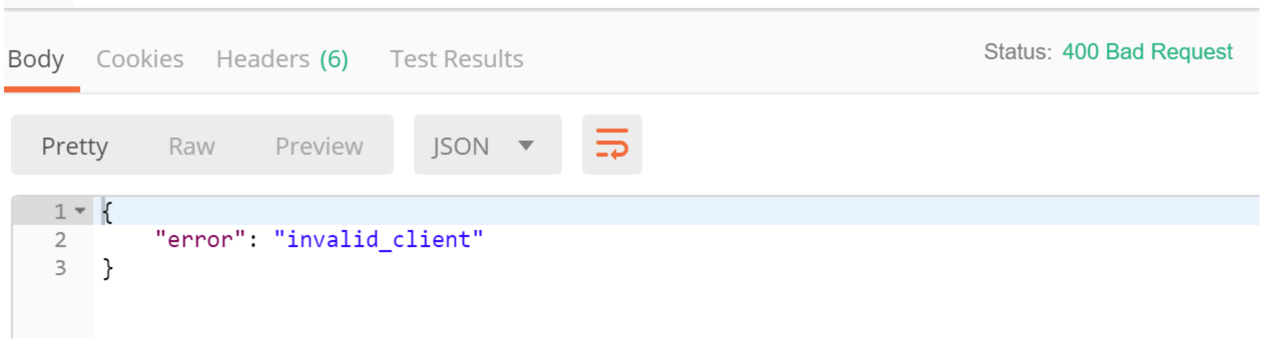


```
// POST /tokens
// Content-Type: application/x-www-form-urlencoded
// Body: {"username":"torres", "password":"torres" }
app.post("/tokens", function(request, response){

    const username = request.body.username
    const password = request.body.password

    const query = "SELECT * FROM accounts WHERE username = ?"
    const values = [username]

    db.get(query, values, function(error, account){
        if(error){
            response.status(500).end()
        }else if(!account){
            response.status(400).json({error: "invalid_client"})
        }else if(account.password != password){
            response.status(400).json({error: "invalid_client"})
        }else{
            const accessToken = jsonwebtoken.sign(
                {accountId: account.id},
                "dsfsddsfdsfsdfd"
            )
            // TODO: Also send back an ID Token as specified in OpenID Connect.
            response.status(200).json({
                access_token: accessToken
            })
        }
    })

})
```

Figure-9 JSON Web Token

POST/TOKENS

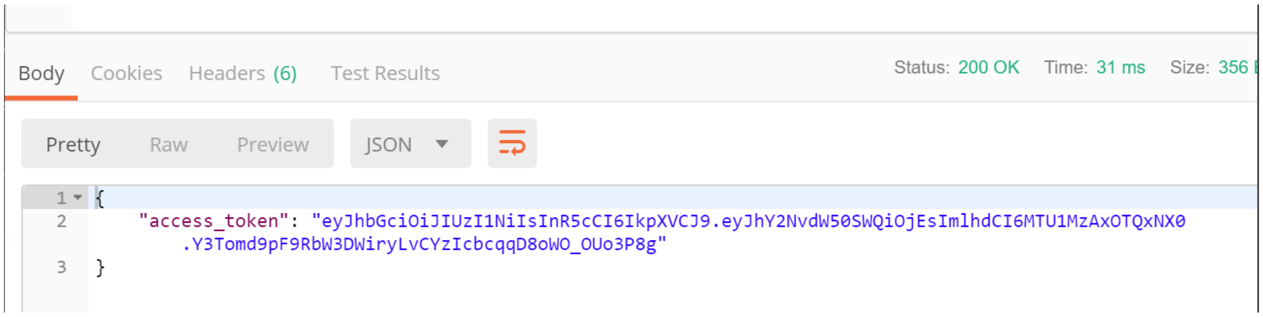
If the username or the password of the account is not correct, then the server will return 400 bad request (*Figure-10*). When the account is correct, then execute "post/tokens" order, and then the server will response 200 ok, and client can get the tokens (*Figure-11*).



Status: 400 Bad Request

```
{
    "error": "invalid_client"
}
```

Figure-10 400 bad request



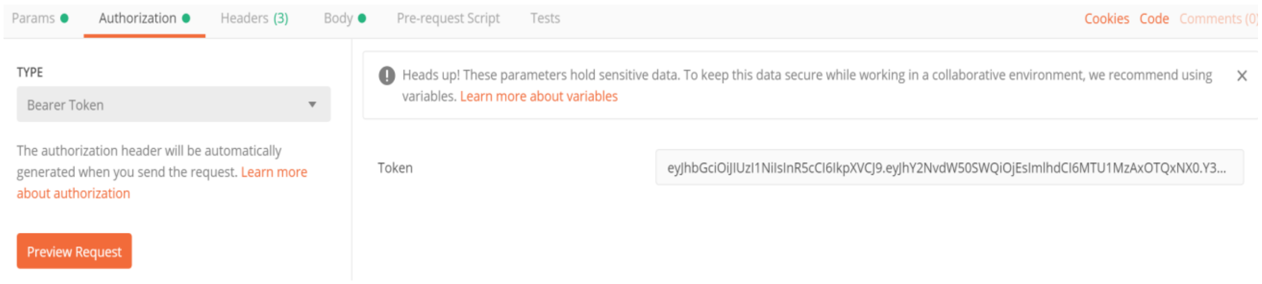
Status: 200 OK Time: 31 ms Size: 356

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50SWQiOjEsImlhdCI6MTU1MzAxOTQxNX0
        .Y3Tomd9pF9RbW3DWiryLvCYzIcbcqqD8oWO_OUo3P8g"
}
```

Figure-11 200 ok

POST/DATAS

After getting the token, put it in the header information Authorization field (*Figure-12*). Then put what user want to post to the server, and if the token is not correct, the server will return 401 unauthorized (*Figure-13*). If a field is missing from the data uploaded by the user, the server will return 400 bad request and the error (*Figure- 14*). If everything is ok, then the server will return 201 created (*Figure-15*).



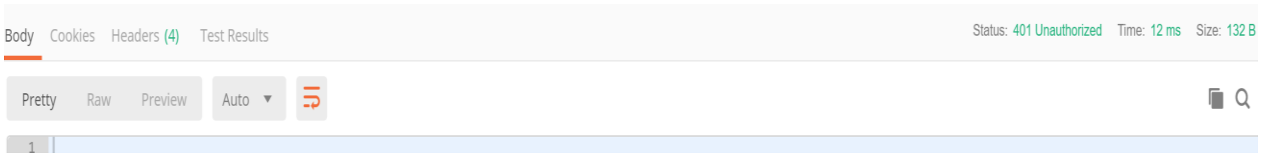
TYPE: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

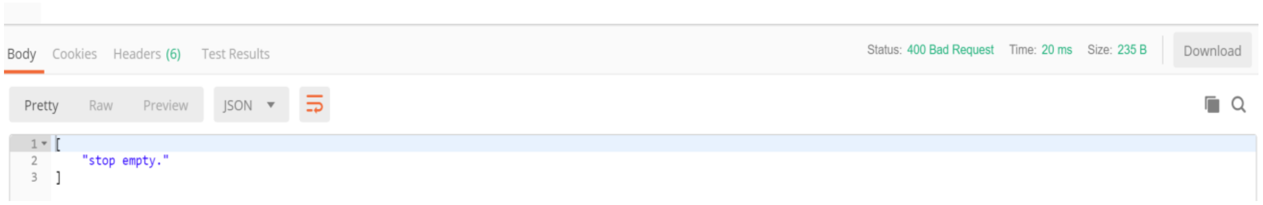
Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50SWQiOjEsImlhdCI6MTU1MzAxOTQxNX0.Y3...

Figure-12 Authorization field



Status: 401 Unauthorized Time: 12 ms Size: 132 B

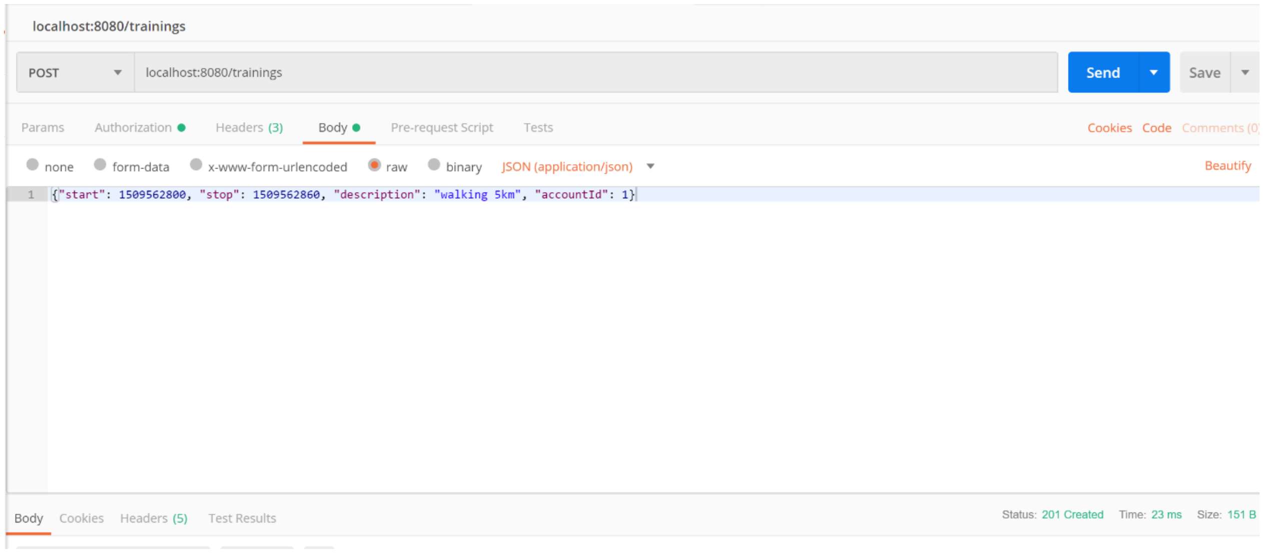
Figure-13 401 unauthorized



Status: 400 Bad Request Time: 20 ms Size: 235 B

```
[
    "stop empty."
]
```

Figure- 14 some field empty



POST localhost:8080/trainings

raw: JSON (application/json)

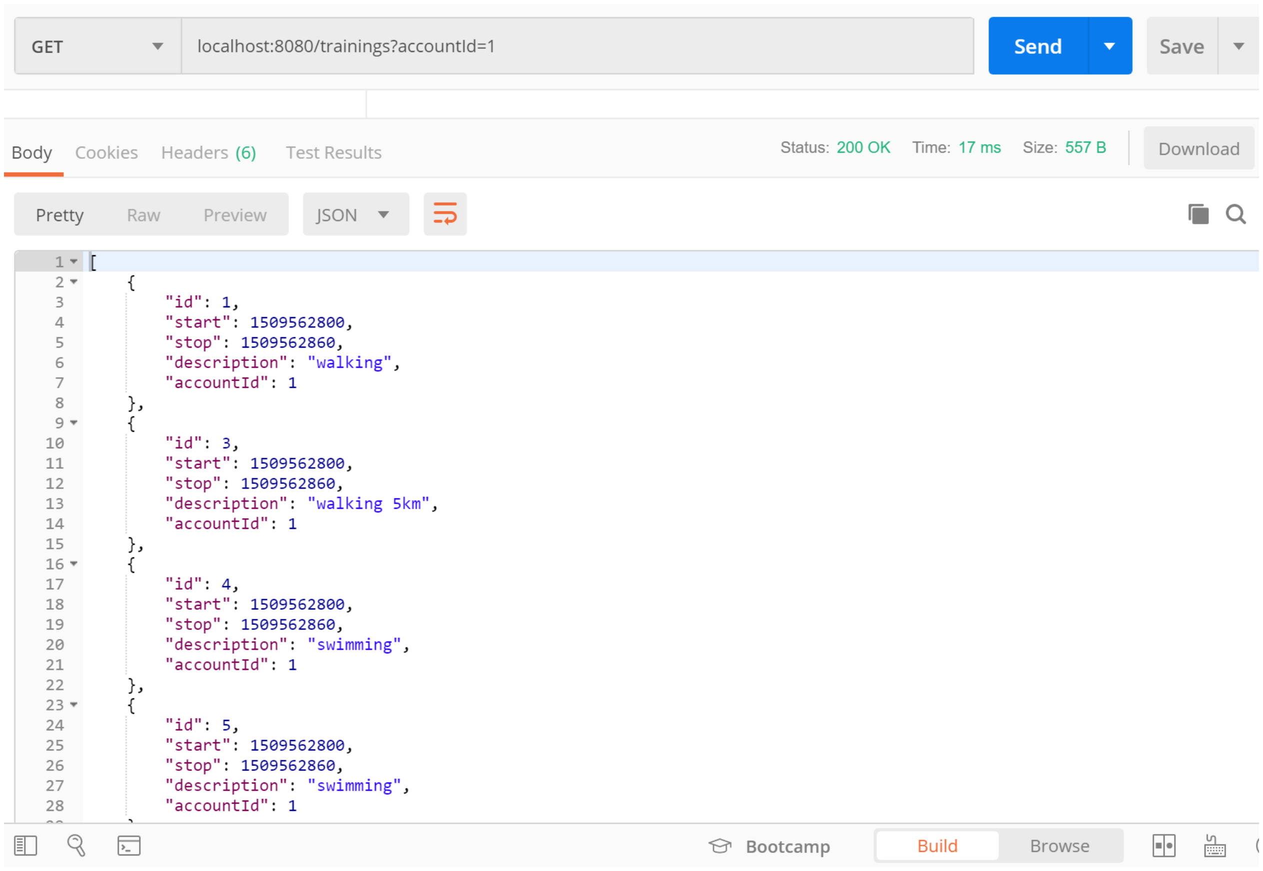
```
{"start": 1509562800, "stop": 1509562860, "description": "walking 5km", "accountId": 1}
```

Status: 201 Created Time: 23 ms Size: 151 B

Figure-15 201 created

GET/DATAS

If the user wants to check the data he/she just post, use get/datas order to complete it. And if the token is correct, the server will return 200 ok and send back the data (*Figure-16*).



```
localhost:8080/trainings?accountId=1
```

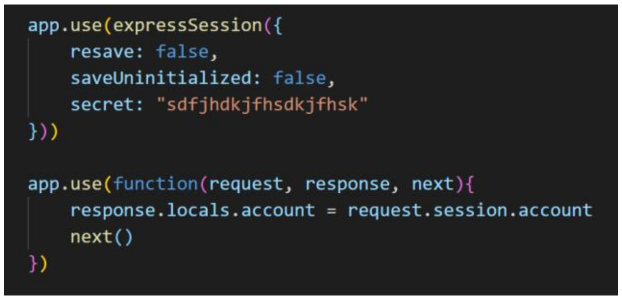
```
[
    {
        "id": 1,
        "start": 1509562800,
        "stop": 1509562860,
        "description": "walking",
        "accountId": 1
    },
    {
        "id": 3,
        "start": 1509562800,
        "stop": 1509562860,
        "description": "walking 5km",
        "accountId": 1
    },
    {
        "id": 4,
        "start": 1509562800,
        "stop": 1509562860,
        "description": "swimming",
        "accountId": 1
    },
    {
        "id": 5,
        "start": 1509562800,
        "stop": 1509562860,
        "description": "swimming",
        "accountId": 1
```

Figure-16 get/datas

Security

SESSION

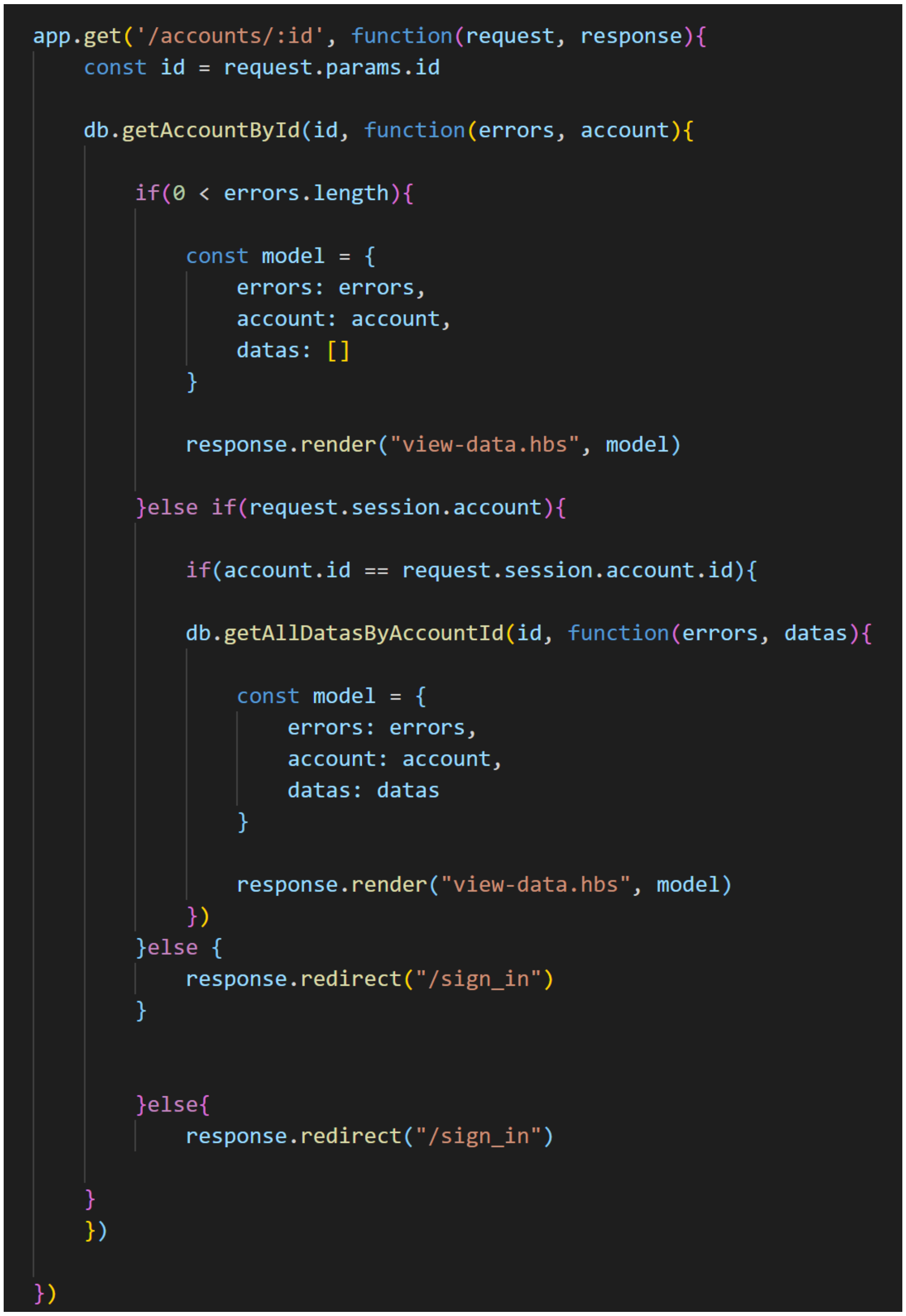
The user can log in MyTD with his/her account if he/she already has one. However, HTTP is stateless. The next time it receives an HTTP request from the same user, it has no memory of that the user has already authenticated itself as the owner of an account in the previous request. The application somehow needs to be remembered this, so MyTD used session to complete it (*Figure-17*).



```
app.use(expressSession({
    resave: false,
    saveUninitialized: false,
    secret: "sdfjhdkjfhsdkjfhsk"
}))

app.use(function(request, response, next){
    response.locals.account = request.session.account
    next()
})
```

Figure-17 session (1)

Cause different user has different account, if one user wants to check other accounts, MyTD will check if the account stored in the session belongs to this user or not, if it is, the user can check the account, otherwise, it will go to the Sign In page, and he/she should sign in (*Figure-18*).

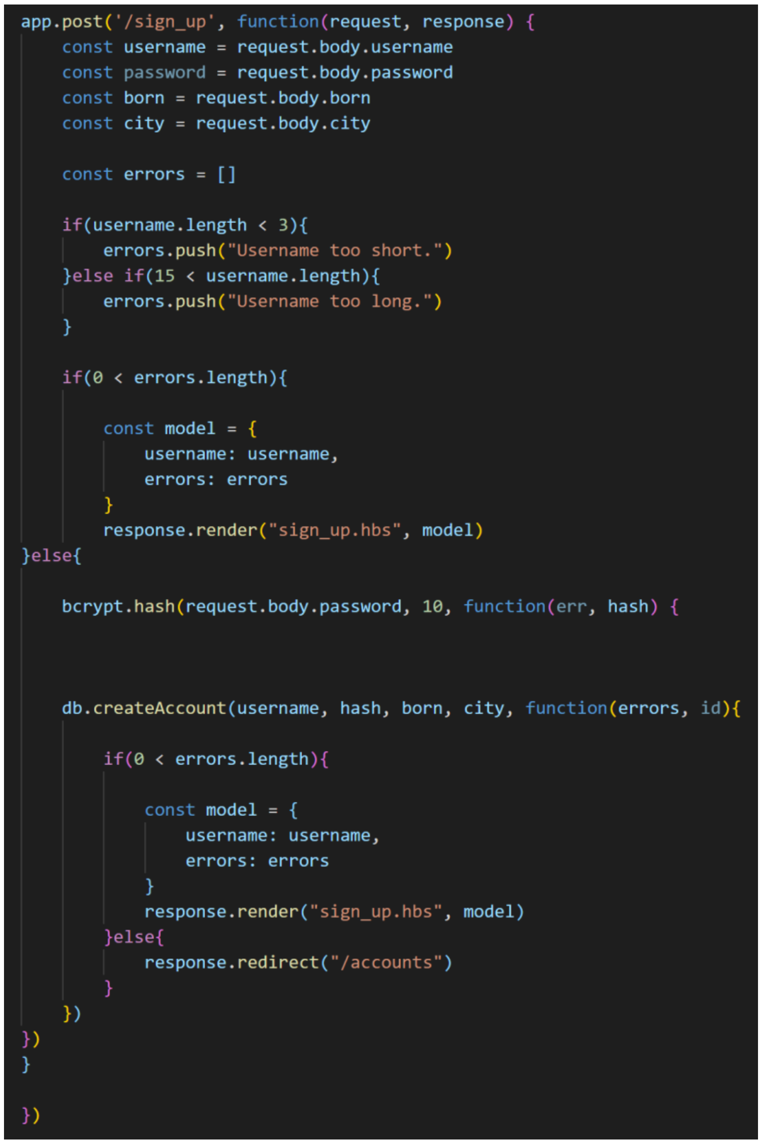


```
app.get('/accounts/:id', function(request, response){
    const id = request.params.id

    db.getAccountById(id, function(errors, account){

        if(0 < errors.length){

            const model = {
                errors: errors,
                account: account,
                datas: []
            }

            response.render("view-data.hbs", model)

        }else if(request.session.account){

            if(account.id == request.session.account.id){

            db.getAllDatasByAccountId(id, function(errors, datas){

                const model = {
                    errors: errors,
                    account: account,
                    datas: datas
                }

                response.render("view-data.hbs", model)
            })
        }else {
            response.redirect("/sign_in")
        }


        }else{
            response.redirect("/sign_in")

    }
    })

})
```

Figure-18 session (2)

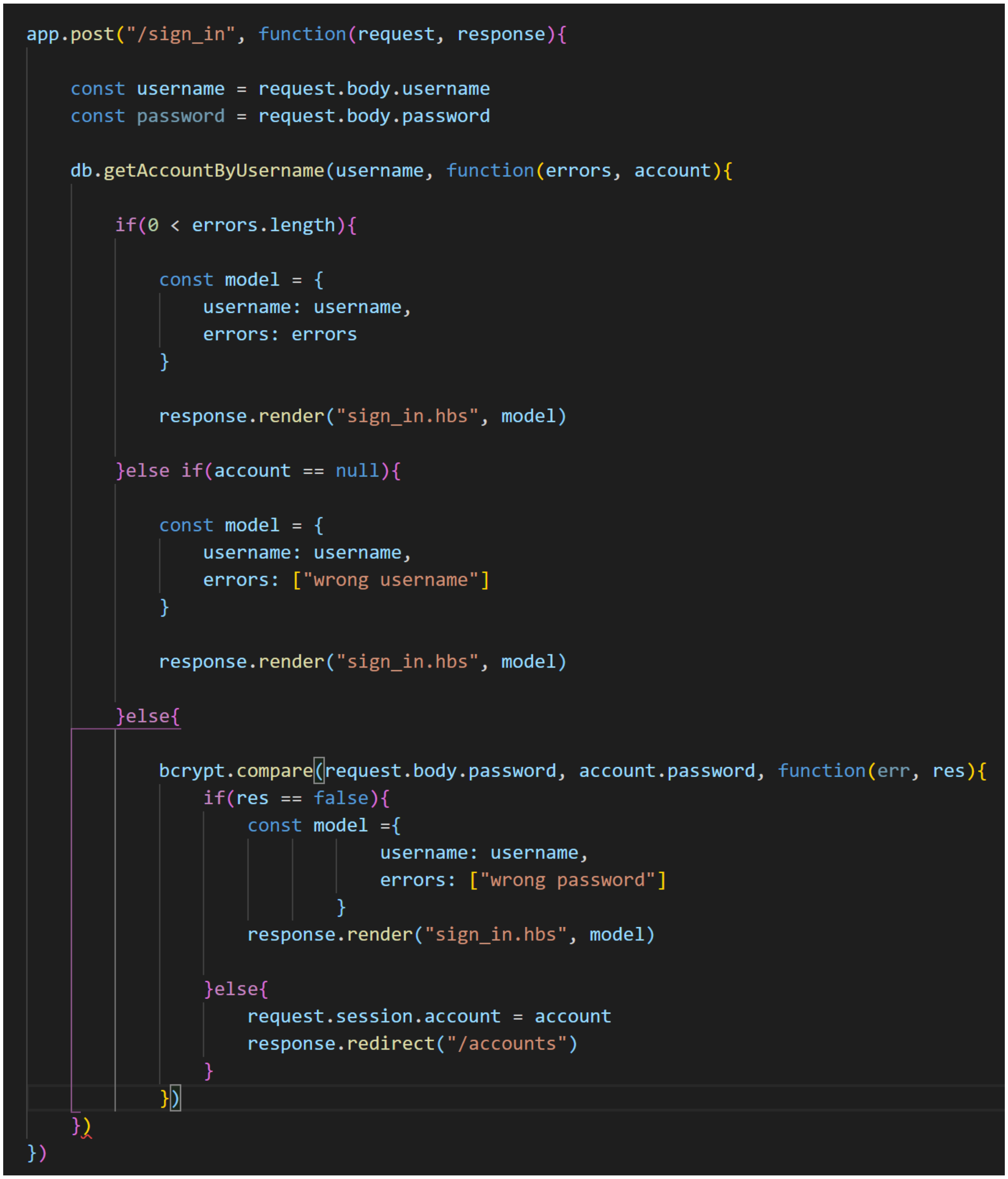
Securely storing the passwords

If the users' passwords are stored in plaintext in the database, that could be a security issue for them. For example, if a hacker comes across your database, she can read the users' passwords in plain text, and since users are stupid and use the same password on different websites, she can now login as them on other websites. Not good! To avoid this, the passwords need to be hashed, preferably with dynamic salt, and we then store the hash of the passwords. This way, if a hacker comes across our database, he/she can't easily figure out the users' passwords. MyTD used npm package *bcryptjs* to encrypt users' password when the user signs up an account (*Figure-19*). And MyTD will compare the password that the user inputs when he/she signs in with the password stored in the database (*Figure-20*).



```
app.post('/sign_up', function(request, response) {
    const username = request.body.username
    const password = request.body.password
    const born = request.body.born
    const city = request.body.city

    const errors = []

    if(username.length < 3){
        errors.push("Username too short.")
    }else if(15 < username.length){
        errors.push("Username too long.")
    }

    if(0 < errors.length){

        const model = {
            username: username,
            errors: errors
        }
        response.render("sign_up.hbs", model)
}else{

    bcrypt.hash(request.body.password, 10, function(err, hash) {



    db.createAccount(username, hash, born, city, function(errors, id){

        if(0 < errors.length){

            const model = {
                username: username,
                errors: errors
            }
            response.render("sign_up.hbs", model)
        }else{
            response.redirect("/accounts")
        }
    })
})
}

})
```

Figure-19 encrypt password



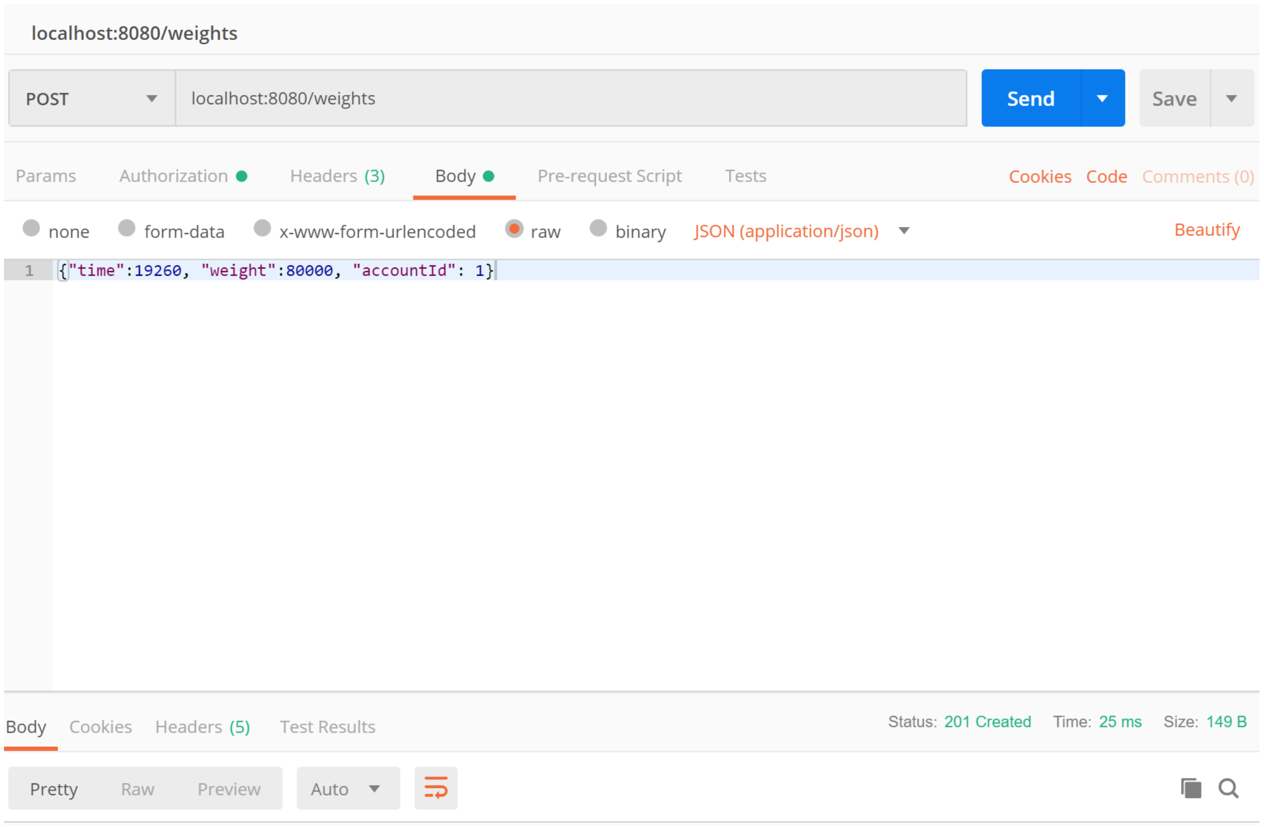
```
app.post("/sign_in", function(request, response){

    const username = request.body.username
    const password = request.body.password

    db.getAccountByUsername(username, function(errors, account){

        if(0 < errors.length){

            const model = {
                username: username,
                errors: errors
            }

            response.render("sign_in.hbs", model)

        }else if(account == null){

            const model = {
                username: username,
                errors: ["wrong username"]
            }

            response.render("sign_in.hbs", model)

        }else{

            bcrypt.compare(request.body.password, account.password, function(err, res){
                if(res == false){
                    const model ={
                                username: username,
                                errors: ["wrong password"]
                            }
                    response.render("sign_in.hbs", model)

                }else{
                    request.session.account = account
                    response.redirect("/accounts")
                }
            })
    })
})
```

Figure-20 compare password

Smart Products

Smart Weighing Scale

When the user uses the smart weighting scale to measure his/her weight, the smart weighting scale will send request to MyTD server and get a token, and then the weighting scale will send weight data to MyTD server by using REST API (*Figure-21*).

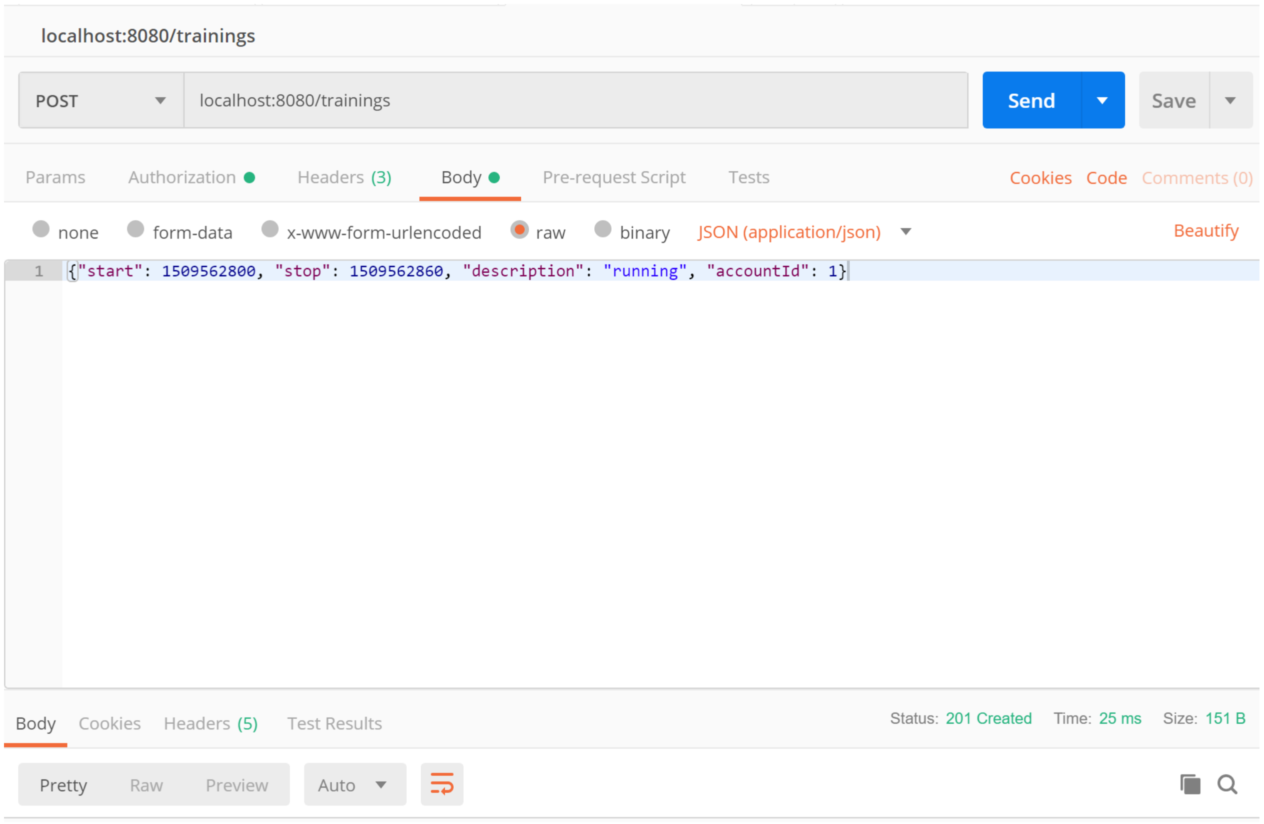


```
{"time":19260, "weight":80000, "accountId": 1}
```

Figure-21 smart weighting scale

Smartphone App

When the user uses the smartphone app to create a new training activity, the smartphone app will send request to MyTD server and get a token, and then the smartphone app will send the new training activity to MyTD server by using REST API (*Figure-22*).


localhost:8080/trainings

POST localhost:8080/trainings

```
{"start": 1509562800, "stop": 1509562860, "description": "running", "accountId": 1}
```

Status: 201 Created Time: 25 ms Size: 151 B

Figure-22 smartphone app