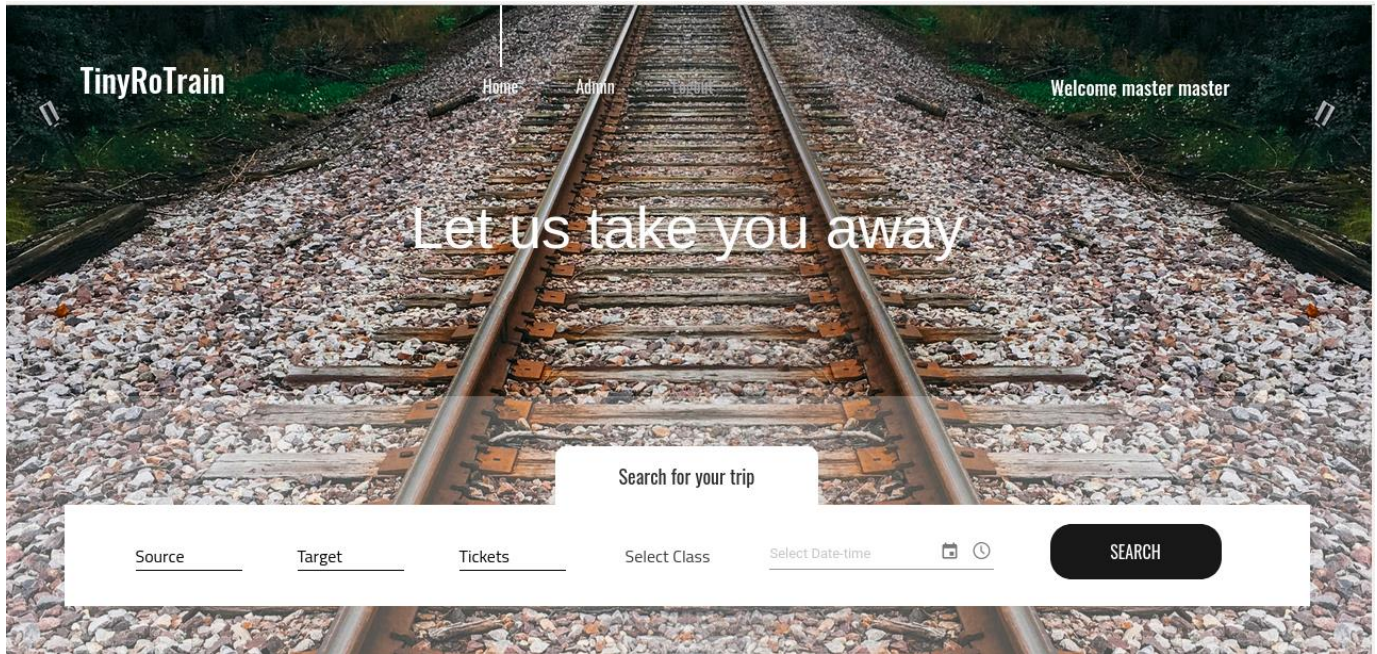


TheTinyROTrain documentation



Content

1. Introduction
2. Description
3. Software design
4. Implementation
5. Deployment information

1. Introduction

1.1 Name of the project/team

TheTinyROTrain

1.2 Team

- Iorga Adina
- Melciu Bogdan-Sabin
- Mema Mihai-Raul
- Petrisor Paul-Andrei

1.3 Contribution

- Frontend part: Iorga Adina and Petrisor Paul-Andrei
- Backend part: Melciu Bogdan-Sabin and Mema Mihai Raul

Both backend and frontend tasks were split equally.

1.4 Github repository

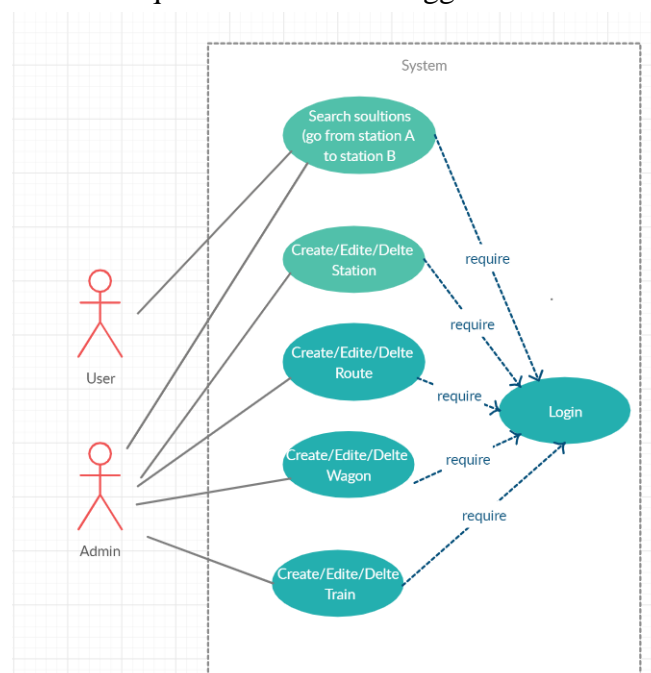
<https://github.com/BogdanSabin/DP-TheTinyRoTrain.git>

2. Description

TheTinyROTrain is a web application which has the main goal to manage a train system. As admin or master you can create stations, routes, wagons and trains. A station represents a place where a train can be at a certain time. A route is a collection of stations, each station in a route has an arrival date (which represents the moment when the train will be in that station). A wagon is of certain type (e.g : class I, class II, restaurant, etc.), and has a number of seats and a price for each seat (varies by type). A train has a name, and a route and a collection of wagons. After a resource is created it can be deleted or edited.

As user (admin/normal user) you could search for trains that could take you from station A to station B.

All the above actions require the user to be logged in.



3. Software Design

The application is formed from two main components: the client part and the server part. The client part is an angular application and the server part is developed with Node.js (12.16.2), as storage unit, the server uses a MongoDB database. MongoDB is a NoSql database which uses documents (JSON structures) as entries.

The communication between client and server is realized via HTTP calls. The server is build around the framework Express.js and exposes RESTful routes that the customer can use.

```
app.use('/api/authentication', routes.authentication);
app.use('/api/resource/user', routes.user);
app.use('/api/resource/station', routes.station);
app.use('/api/resource/route', routes.route);
app.use('/api/resource/wagon', routes.wagon);
app.use('/api/resource/train', routes.train);
app.use('/api/resource/image', routes.images);
app.use('/api/booking', routes.booking);
```

Routes organization

The communication between server and Mongo database is realized using the mongoose package. For each resource there is created a mongoose schema and a Model. All models uses a single connection to the database singleton pattern.

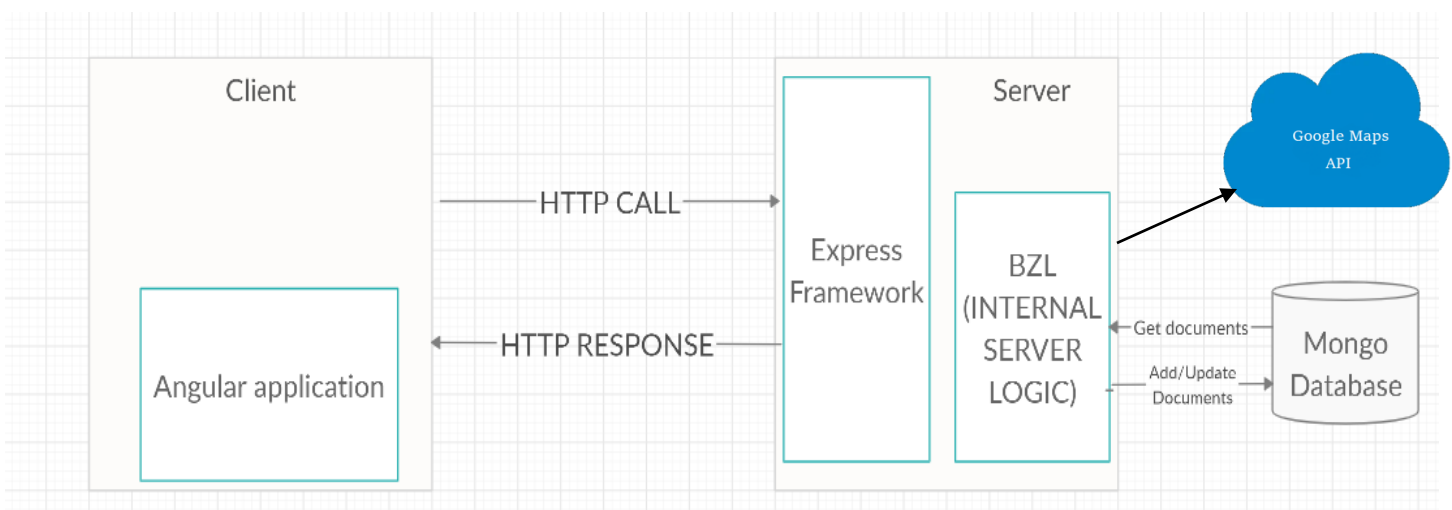
```
var trainSchema = mongoose.Schema({
  name: String,
  route: { type: mongoose.Types.ObjectId, ref: 'Route' },
  wagons: [{ type: mongoose.Types.ObjectId, ref: 'Wagon' }]
},{
  collection: 'trains'
});

mongoose.model('Train', trainSchema);

module.exports = function(){
  return getConnection('connection').model('Train');
}
```

Schema Example for train resource

The communication between the express framework and the server's internal logic is done via a middleware where the routes are interpreted (parameters are extracted from route params or body) and methods from bzl are called (in the same process).



4. Implementation

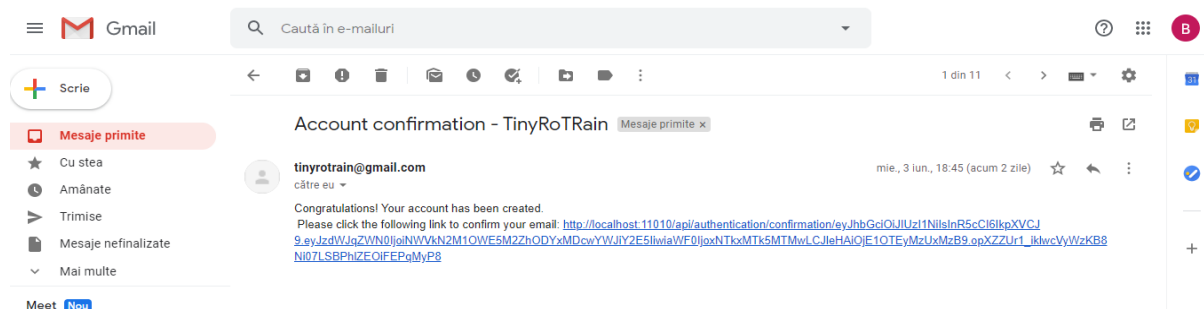
4.1 Server

The server exposes routes for login and register and for each resource (station, route, train, wagon) routes for CRUD operations. The login system is based on JWT (JsonWebToken) and email confirmation. After a user has registered he must confirm his email account in order to log in. After the email confirmation the user is saved in database with his password encrypted by the server. We used the *nodemailer* module from npm to send the emails and the *jsonwebtoken* module to create and verify tokens. The emails were sent by a gmail account created by us (tinyrotrain@gmail.com).

After a user has logged in a jwt is sent from the server that contains (as secret) the mongo id and the role of that user. For each request to the server the token received at login is sent in the headers under Authorization. On the server the token is checked to see if the user can access that route. This is how the differentiation is made between the routes that a normal user and an admin can access.

For each resources (station, route, wagon, train, user) we have implemented the same CRUD operations, and in order to not have duplicated code we created a template method for each of those operations (create, delete, update, findOne, findAll). The template methods could be found in /server/src/lib/helper. For working with Mongo documents we used Model from Mongoose.

The algorithm for the search of trains which go from station A to station B we use the API for distances from Google which tells us the direct distance from point A to point B. we know precisely that the two points exist on the map because on the client side where we create stations we used the API for finding places from Google. We use these two APIs with an API key generated by Google Maps Platform. On the server side the API for getting distances is encapsulated under the *google-distance* module from npm.



Email confirmation

```
createTrain: function (data, token, next) {
  return authorize(token, 'admin', function (error, ok) {
    if(error)
      return next(error);
    if(ok)
      return helper.createResource({
        data: data,
        Model: lib.trainModel,
        createFilter: lib.createFilter(data),
        responseFilter: lib.responseFilter,
        afterCreate: libWagon.attachedToTrain
      }, next);
  });
},
```

Only admins can create trains

Route Authorization

```

module.exports.getResourceById = function(options, next){
  if(!options.id)
    return next(serverError.MissingArgument('Data in function getResourceById'));
  if(!options.Model)
    return next(serverError.MissingArgument('Model in function getResourceById'));
  let Model = options.Model;
  if(!options.populate)
    options.populate = function(data, next){
      return next(null, data);
    }
  let populate = options.populate;
  if(!options.responseFilter)
    options.responseFilter = function(data, next){
      return next(null, data);
    }
  let responseFilter = options.responseFilter;
  Model.findOne({_id: options.id}, function(error, doc){
    if(error)
      return next(serverError.InternalError(error));
    if(!doc)
      return next(serverError.NodataFound());
    populate(doc, function(error, populated){
      if(error)
        return next(serverError.InternalError(error));
      if(!populated)
        return next(serverError.NodataFound());
      responseFilter(populated, function(error, data){
        if(error)
          return next(serverError.InternalError(error));
        if(!data)
          return next(serverError.NodataFound());
        return next(null, data);
      });
    });
  });
};

```

Template method for getting a resource by id

```

getTrain: function (trainid, token, next) {
  return authorize(token, 'admin', function (error, ok) {
    if(error)
      return next(error);
    if(ok)
      return helper.getResourceById({
        id: trainid, Add the corresponding id (mongo id)
        Model: lib.trainModel, Add the corresponding Model
        populate: lib.populate, Add the populate func. for Train
        responseFilter: lib.resposeFilterGet
      }, next);
    Add the response filter for trains
  });
},

```

The call of the template method getResourceById for trains

ProjectDB localhost:27017 tinytrain

db.getCollection('users').find({})

users 0.021 sec.

Key	Value	Type
(1) ObjectId("10101010101010101010101010101010")	{ 8 fields }	Object
_id	ObjectId("10101010101010101010101010101010")	ObjectId
role	master	String
emailConfirmation	true	Boolean
firstName	master	String
lastName	master	String
email	master@email.ro	String
password	a62b07f973a477b655c989016d8cc442	String
_v	0	Int32

Hashed password

```

module.exports.findSolution = function(data, next){
  let solutions = [];
  ModelRoute.find({ "$and": [
    { "stations.name": data.stationStart },
    { "stations.name": data.stationEnd }
  ]})
    .then(function(routes){
      //get Routes that contains the 2 stations and meet the time criteria
      return getRoutesByDate(data.departureData, routes, data.stationStart)
    })
    .then(function(filtredRoutes){
      let finalRoutes = filtredRoutes;
      //get trains that have those routes
      return getTrainsByRoutes(finalRoutes)
    })
    .then(function(trains){
      //get wagons that have free space and the class required by user
      return filterByWagons(trains, data.wagonClass, data.numberOfTickets)
    })
    .then(function(results){
      solutions = results;
      //get distance from Google API
      return getDistance(data.stationStart, data.stationEnd)
    })
    .then(function(distance){
      solutions.forEach(f =>{
        f.distance = distance;
        f.class = data.wagonClass;
        f.departureData = data.departureData;
        f.price = (distance * pricekm) + f.priceW; //calculcate price for ticket
      })
      return next(null, solutions);
    })
    .catch(function(reason){
      console.log("Error search: " + reason);
      return next(null, reason);
    });
};

```

Algorithm for finding solutions

4.2 Client

The Client part consists of the **Main, Login & Registration, Admin, User Profile** and **Results** pages (components).

Main page(components)

Contains:

- information about the company and team
- links to other pages such as login in case we are not logged in or logout in case we are logged in and admin page which is visible only for users that have master or admin role.
- Also if you are logged in the user name will be displayed on the header (received from login - local storage). The display name is also an link that if pressed we will be redirected to user profile page.
- Search form that must be filled in order to search, at button press all data is stored and sent through a post request which will take as parameters the object that has the data and also the authorization token which we will get from login (local storage), to the server . As a response we will get an object which will contain the results used in results page.
- Cannot search if you are not logged in !

Results page

Contains a table which displays the data (object) response from search service what we get from the server.

- In the table we also have the opportunity to buy a ticket on desired route through a button that makes a post request to the server sending the current logged id (user), received from login (local storage) added to the server route url, the data of the desired route on which you want to buy the ticket and the authorization token of current logged user. At a successfully response from the server user will be redirected to the user profile page to see their tickets.

User Profile page

- Contains links as in main page to navigate to different pages.
- Profile page also contains a user profile image that can be changed and information of current user logged in received at login (local storage) as role, name.
- Every time we access/refresh the page, it does a get request to the server which has as parameters the server route url on which we append the current user id (received at login)

and the authentication token, as a response to this request we will get an object containing the data of the route on which we have bought tickets.

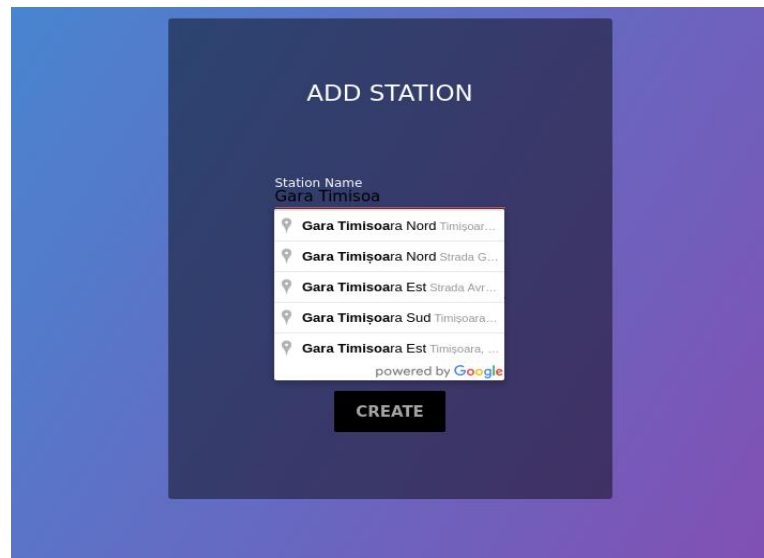
Login & Registration pages (components)

A form must be filled up on both pages in order to login or register. (email verification first)

- The client, communicate with server through a post request which will sent the data stored from the form, through a route url.
- As a successfully response from the server on login we will get the login token, role, id and name which will be stored on local storage. After response we check if the role is a master or admin, we redirect to admin page otherwise we redirect to main page
- As a successfully response from the server on register we will be redirected to login page
- Both pages contain navigation links between them, which is done through a router service which will redirect to desired page.

Admin page

- Cannot access page if you are not logged in as an admin or master
- Has a side panel in which we can chose to create(add) or view all stations, routes, wagons, trains, users.
- For the create choice all are the same, the data which is stored from the forms along with the authorization token is sent through a post request to the server (server route url). Before making the post request some of the create options such as create route, wagon or train have select boxes in which you can select the added stations/wagons/routes, in order to do this when we choose the desired create option the client calls a get request method that receives from the server as a response, objects containing the specific stations/wagon & wagon classes)/ route data. Also to create a station you can use Google's autocomplete api for location.



```

createRoute(){
  return this.http.post(this._createUrl, this.Routes, this.httpOptions)
    .subscribe(
      res=> {
        console.log(this.Routes.name)
        this.showSuccess()
        this.refresh()
      },
      (err) => {console.log(err)
        this.showError();
      }
    )
}
}

```

- The second option that we can choose is to display separately all stations/routes/etc..
When we choose the desired option an get request is called to the server , as a response to these requests we get different objects (depends on the case) which will contain all stations/wagons/etc... which will be displayed on tables on different pages (depends on case).

VIEW ALL STATIONS

Name	Time to Wait	Update	Delete
North Railway Station Neighborhood	2	UPDATE	DELETE
Timișoara Nord	10	UPDATE	DELETE
Gara Centrală Arad	1	UPDATE	DELETE

```

getAllStations(){
  return this.http.get<any>(this._getAllStationsURL, this.httpOptions)
    .subscribe(
      data => {this.responseStations = data},
      res => {console.log(res);
    }
  )
}

```


- For each table you can choose two options (you have two buttons) one for updating or deleting the current station/route/etc.

If we choose to update the current station for example a get request is called which will send to the server the current id of the station we want to update and the server will response us with the desired station , object containing all the data for that station, this data will be displayed on new page in forms where we have an button that calls a post request with the updated fields of the current station/wagon/etc... id.

```
getOneStation(id){
  return this.http.get<any>(this._getOneStationURL+id, this.httpOptions)
    .subscribe(
      res => {
        this.stations = res,
        this.current_id = id
      }
    )
}

getOneRoute(id){
  return this.http.get<any>(this._getOneRouteURL+id, this.httpOptions)
    .subscribe(
      res => {
        this.Routes = res,
        this.current_id =id
      }
    )
}
```

The Delete option in the table calls a delete request to the current id of the station and the server will delete the current station sent as id (table view is refreshed after deletion).

- The admin page also has the option to see all registered users and can delete or update their data either first name, last name, email or role. This is also done through post/request methods as explained above.

UPDATE USER

User Role
admin

CHANGE ROLE

First name
Paul

Last name
Andrei

UPDATE

5. Deployment information

I did **not** implement a deployment system in time for the presentation but the idea was to create two docker images. First docker image will have the client (Angular Project) and the second image will have the server side and a server for MongoDB. Having this two images on the docker hub repository we could make a container for each component, and those containers will be deployed on any server, or local machine.