FACTORY TREE APPLICATION
PASSPORT.INC DEVELOPER CHALLENGE

DOCUMENT V1.0

PROJECT:

The objective of the project is to design a Live updating tree as a Full stack web application.

Requirements:

- The Tree should contain a group of nodes under a root node. Each node is a Factory.
- Each Factory can have a number of child nodes which are randomly generated with values within a certain range.
- Factories can be altered for name and range.
- Whenever child nodes are added the existing nodes are removed and replaced with the new nodes. There is a limit of 15 nodes to be added.
- When a factory range is changed the existing nodes are removed.
- The state of the Tree should be persistent.
- The tree should be live updated.

Tech Stack:

- Java 8 and Spring Boot with Embedded Tomcat for Middleware service. Using Maven for dependency management and building.
- Express + Node.js as client. Use Grunt as task runner.
- HTML5, CSS3, JQuery and Twitter Bootstrap for Frontend.
- Mongo DB as NoSQL persistent Data store.
- Redis for Pub/Sub & Socket.io for Real-time broadcasting & updating.

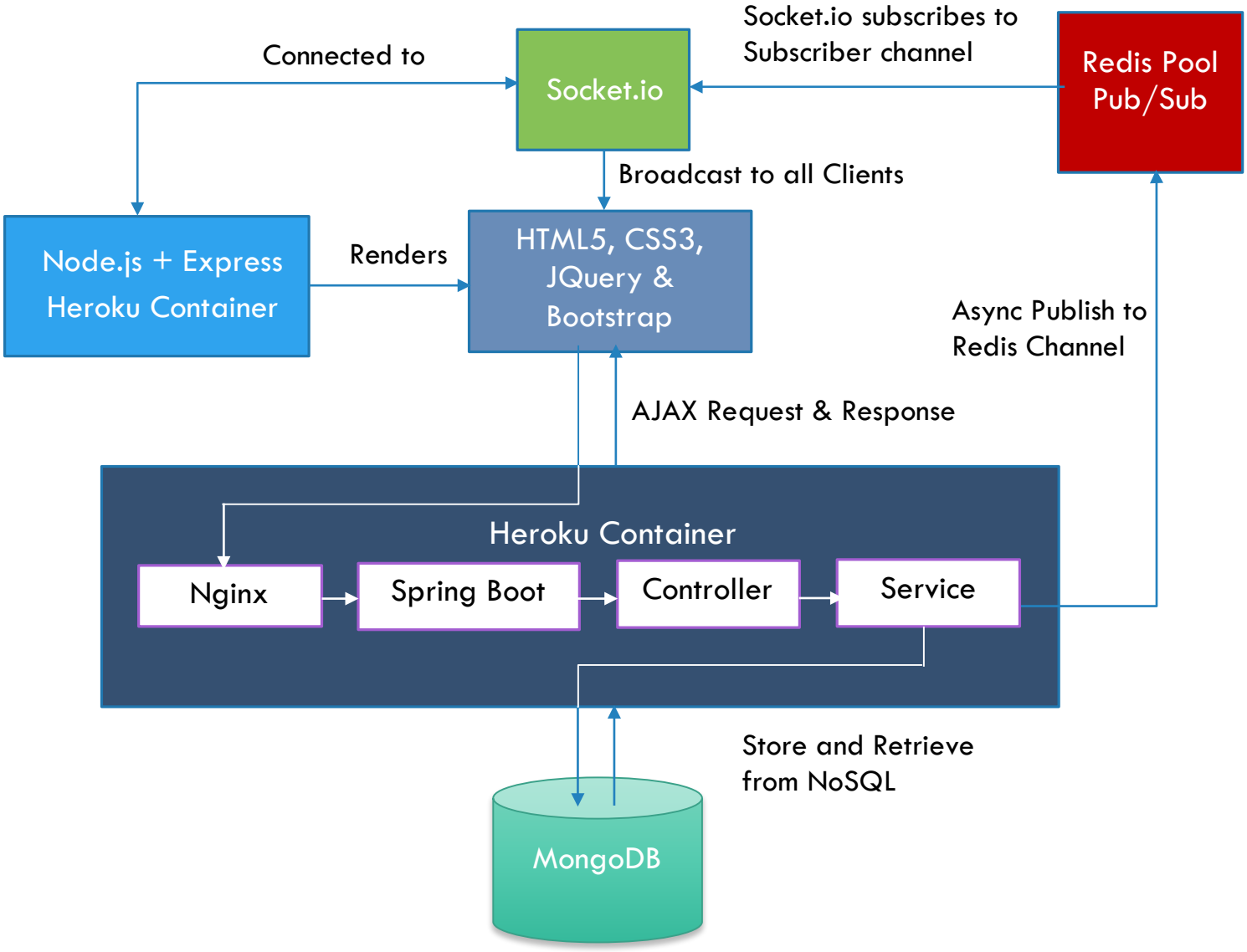
Infrastructure:

- Node.js + Express + Socket.io deployed to Heroku Dyno which uses Nginx routed to the node application.
- Spring Boot app deployed to Heroku Dyno which uses Nginx that forwards to the Tomcat application.
- MongoDB and Redis deployed to Amazon AWS running Ubuntu 14.04.

Architecture:

The architecture is highly distributed and scalable in future. The front-end Node.js application serves the static site which connects to the Spring service through Ajax. For scaling it is easy to add more instances of the Micro service. This can be configured using Eureka and ZUUL to support Micro Service Architecture. The advantage of using such an architecture is balance the load and handle high amount of web traffic.

The Real-time part is highly optimized by using Redis as a Pub/Sub service. Live updating can be achieved in several ways some involving web sockets. With Redis and Socket.io in place the real-time load is completely balanced thereby reducing the bottleneck on the web service. Also sharing socket connection between multiple services would be a problem when using a single web socket.



Node.js + Express Client:

The client application is a Node.js + Express app which serves static index.html along with the assets. It also is integrated with Socket.io and listens to the factory channels for updates. Express provides a structure to the node app which has minimal routing. Since Client is separated from the Server the load for rendering views and assets is split. The Middleware server acts purely as an API in this case.

Frontend:

Since this is a single page application that loads only once initially there is no need to include a client side MVC which would be an overkill. The frontend is HTML5, CSS3, JQuery and Twitter Bootstrap 3. For the Tree rendering a plugin called bootstrap-treeview.js is used. Since the library was constrained to just construction of the tree, there were some changes made to the design of the library.

In addition to this the whole app is Modularized with a number of design patterns implemented to provide better maintainability and structure to the app. The following design patterns are Implemented:

1. Prototype Pattern
2. Module Pattern
3. Revealing Module Pattern
4. Proxy Pattern

The whole website is completely responsive to fit across a variety of screen sizes. It also uses Alertify.js plugin to enable notifications.

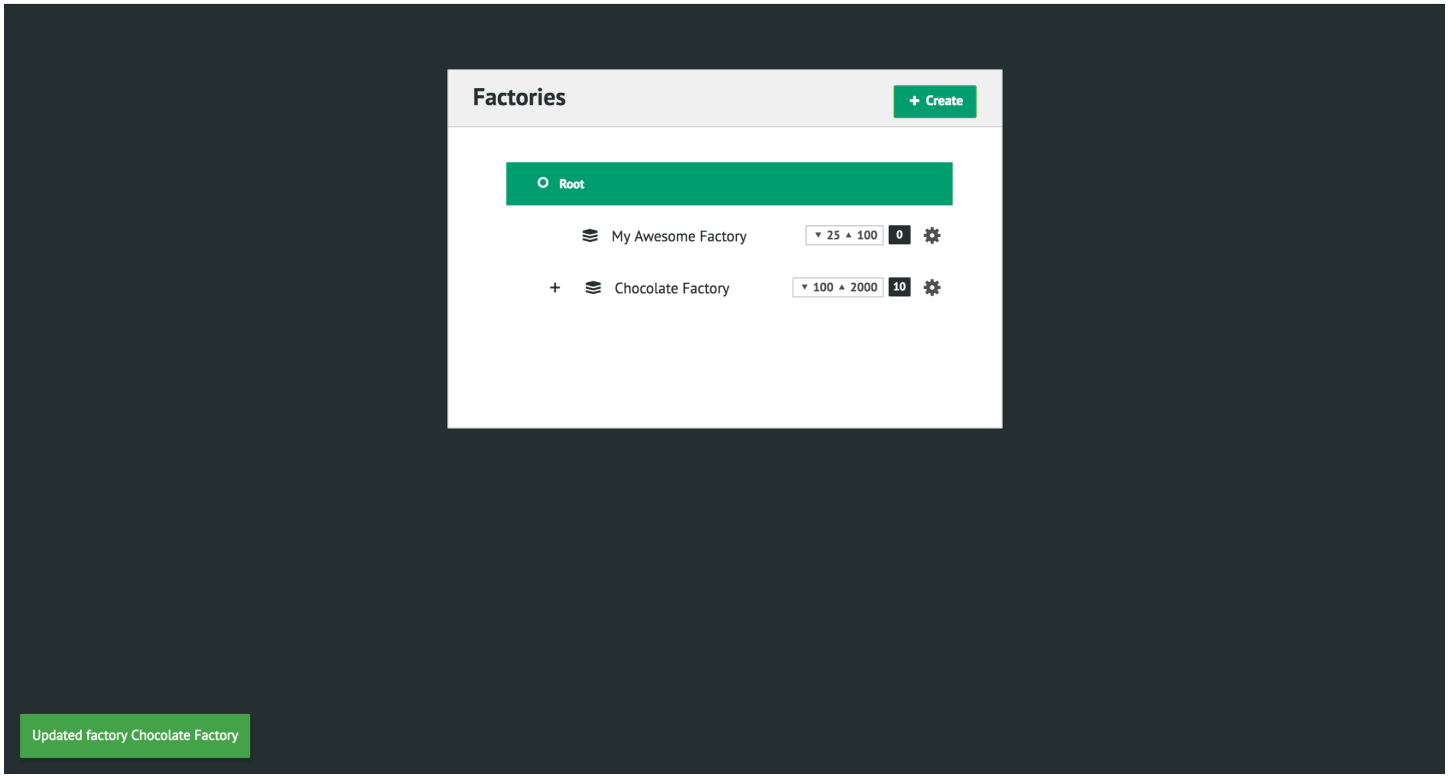
The assets of the app are combined, minified and gzipped for production. This is made available through Task Runner Grunt. Also configurations for each environment are setup separately.

You can check out the complete source code for the client-app here:

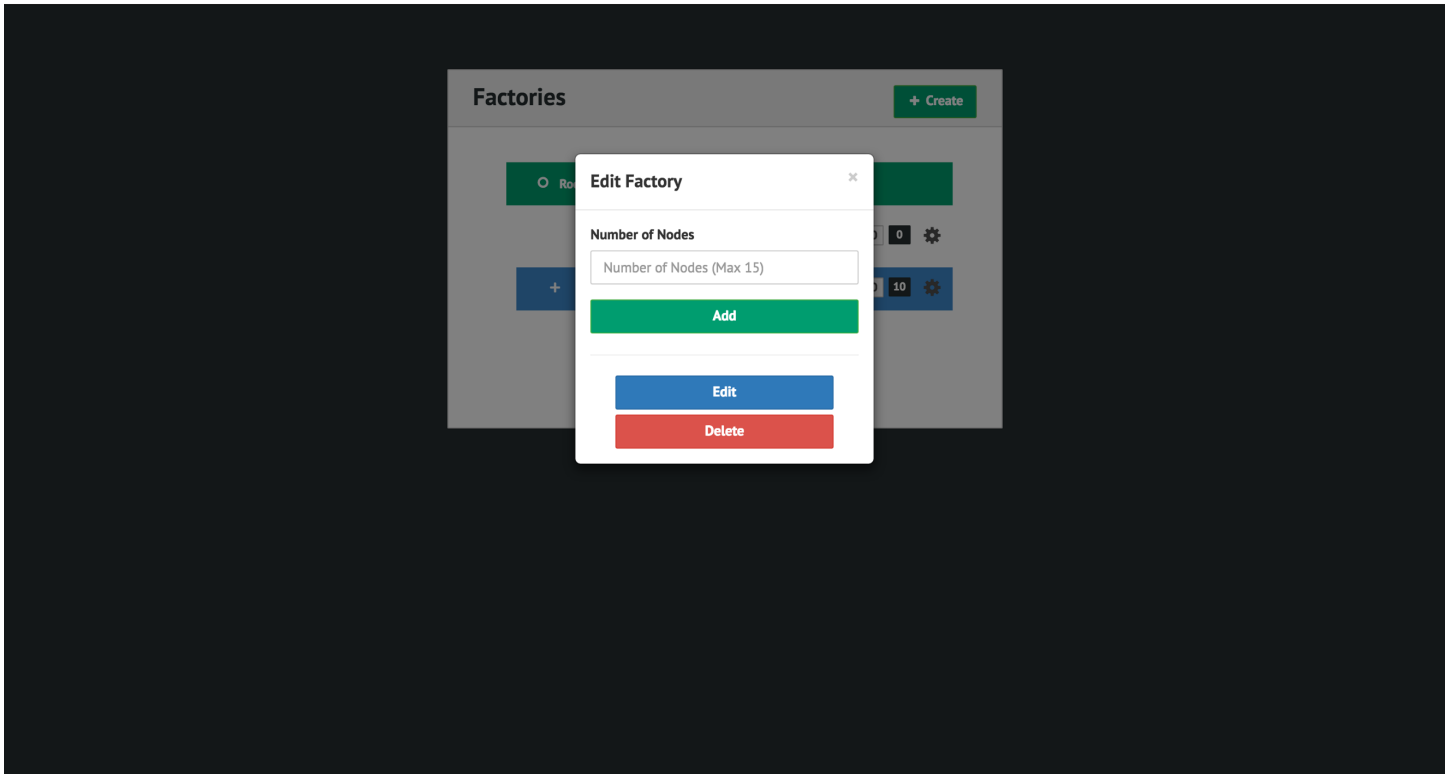
GitHub: https://github.com/sudharti/passport-developer-challenge/passport-tree-client

Demo Link: https://passport-factory.herokuapp.com

Factory Tree



Factory Tree



Middleware:

Middleware is Spring Boot application which acts as an API to interface between the client and the database. The factory CRUD operation is written as a Microservice and the output follows the REST API conventions. The output JSON is consumed by the Frontend application. Before saving the data it validates using a Validator service. It also uses an Async mechanism to update Redis subscriber channel with the updated object. This is used for Broadcasting to all client applications.

CORS is an important feature that is being handled by the application. In production environment it accepts Requests only from the client domain. This ensures the security of the API by confining it to the client alone. For development purposes CORS is disabled in development mode.

Spring Boot uses a Model View Controller architecture. The View is basically just JSON content transferred over HTTP. The Controller layer handles the incoming request and passes over to the Model/Service layer where it is validated and stored in the database. The Response is rendered by the Controller.

Exception Handling:

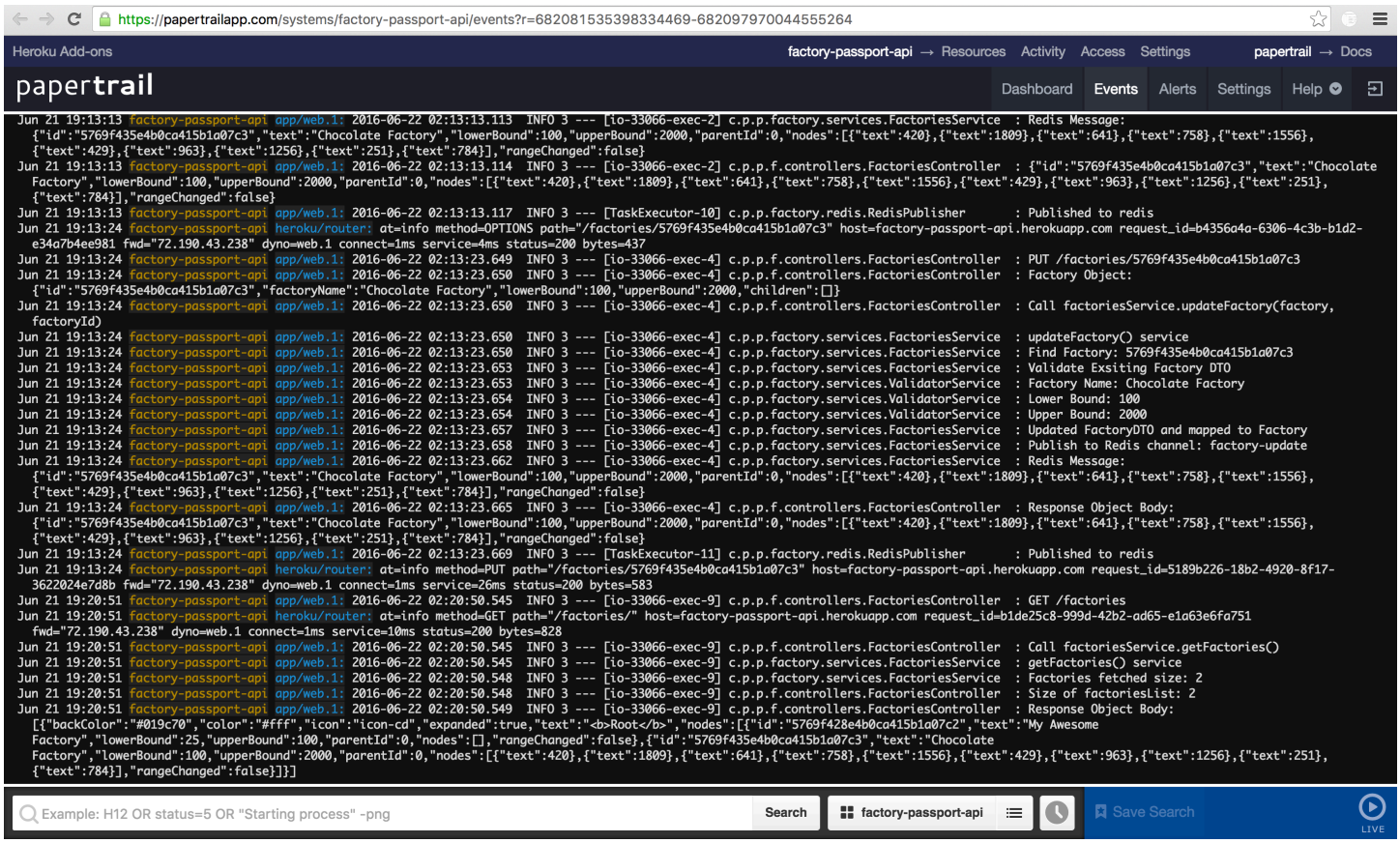
The Middleware application uses a common Exception handler. Whenever an exception is thrown in any level is transcends above to the Application where it is handled by the ControllerAdvice. The ControllerAdvice renders the appropriate error message and status.

In addition to inbuilt exceptions a few User defined exceptions are also thrown and handled in the application.

Logging:

To enable end-to-end monitoring of the Application workflow Logging is enabled in all the layers. The log level is set to info to enable workflow tracking. For Logging Apache Library Log4J is used. For the error logging the logs are errored out in the Exception Handler. The Live logs can be viewed in production with the help of Papertrail addon for heroku. But it can be viewed even if deployed separately.

Here is a screenshot of the Papertrail live logging requests. Everything from class name and object json is logged for a detailed monitoring in Production.



Mongo DB:

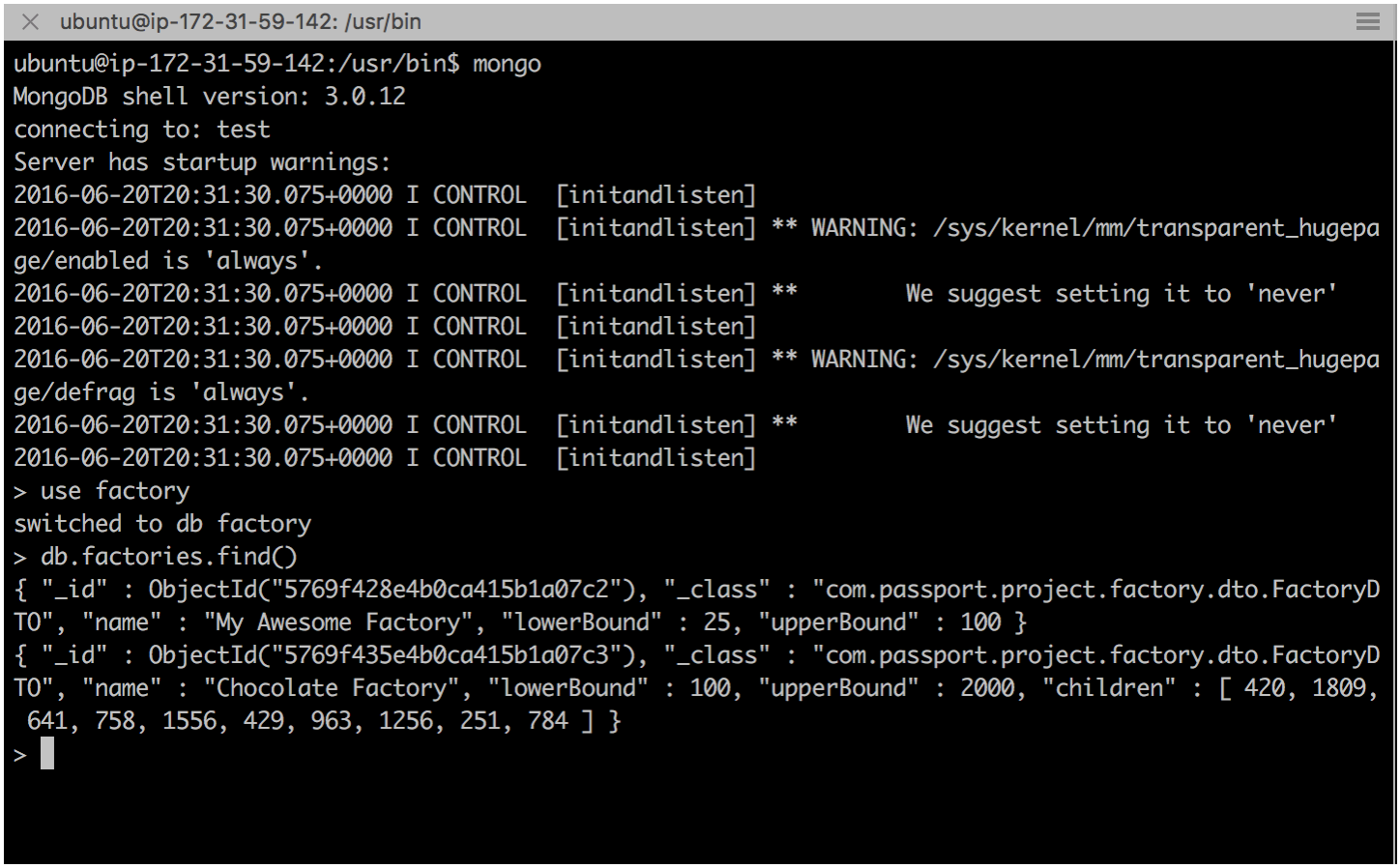
MongoDB version v3.0.12 is deployed to EC2. The reason for using MongoDB as the backend datastore for this application is for Speed primarily. And since the App functionality is just a CRUD of Factory model and childNodes creation this is a perfect fit. MongoDB is schema-less and uses a JSON like document to store the data. And the whole thing is indexed to ensure speed. So the app feels fluidic to use.

For creation of childNodes there is a flattened model that is used. In MongoDB it is best practice to flatten the data instead of creating a relational document, which eliminates the need for N+1 Queries. Since childNode is this case is just a random number generation this works best and is stored as an Array which is serialized and deserialized.

Collection: factories

Document Fields: id(Auto generated), name, lowerBound, upperBound, children, created_at, updated_at

Here is the Mongo shell in production running on EC2. Command for displaying all the records in the collection factories.



{ "_id" : ObjectId("5769f428e4b0ca415b1a07c2"), "_class" : "com.passport.project.factory.dto.FactoryDTO", "name" : "My Awesome Factory", "lowerBound" : 25, "upperBound" : 100 }

{ "_id" : ObjectId("5769f435e4b0ca415b1a07c3"), "_class" : "com.passport.project.factory.dto.FactoryDTO", "name" : "Chocolate Factory", "lowerBound" : 100, "upperBound" : 2000, "children" : [420, 1809, 641, 758, 1556, 429, 963, 1256, 251, 784] }

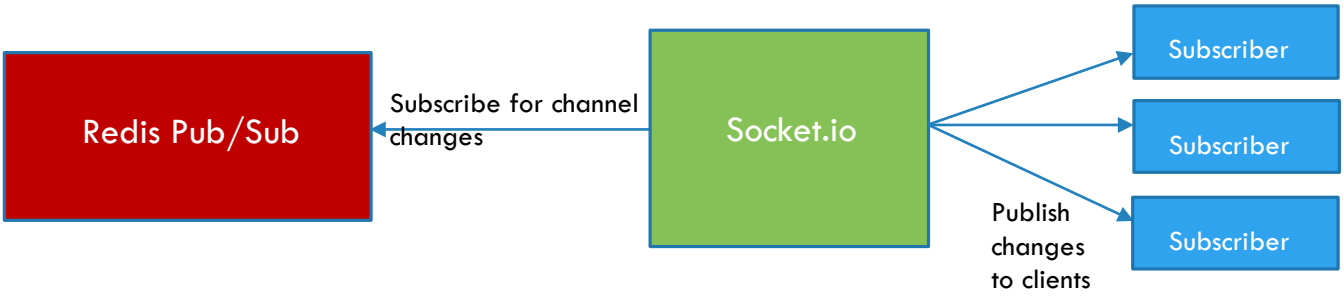
MongoDB running on EC2 is connected to from the Spring Boot app in production. For this to work the ports should be enabled for inbound requests on Amazon AWS. The ports are enabled for both MongoDB and Redis.

Redis + Socket.io for Real-time Processing:

Redis is an In-Memory store that is primarily used for caching due to it's faster Map access. It also has a less used feature of Pub/Sub by creating channels. Whatever that is published to Redis is Published back to the subscriber channel. Request comes in from Publisher channel and clients connected Subscriber channel consume them. This basically acts more like a Queue in this case. Redis also stores a pool of connections. We can also monitor the Redis requests and responses using the redis-cli.

```
52.205.1.181:6379> MONITOR
OK
1466565571.786490 [0 54.158.25.177:43221] "PUBLISH" "factory-create" "{\"id\":\"576a03c4e4b0ca415b1a07c4\",\"text\":\"New Awesome Factory\",\"lowerBound\":35,\"upperBound\":100,\"parentId\":0,\"nodes\":[],\"rangeChanged\":false}"
1466565577.551219 [0 54.158.25.177:43221] "PUBLISH" "factory-child" "{\"id\":\"576a03c4e4b0ca415b1a07c4\",\"text\":\"New Awesome Factory\",\"lowerBound\":35,\"upperBound\":100,\"parentId\":0,\"nodes\":[{\"text\":63},{\"text\":78},{\"text\":63},{\"text\":76},{\"text\":53},{\"text\":99},{\"text\":100},{\"text\":78},{\"text\":76},{\"text\":59},{\"text\":48},{\"text\":88},{\"text\":56},{\"text\":97},{\"text\":57}],\"rangeChanged\":false}"
1466565588.326870 [0 54.158.25.177:43221] "PUBLISH" "factory-update" "{\"id\":\"5769f428e4b0ca415b1a07c2\",\"text\":\"My Awesome Factory\",\"lowerBound\":25,\"upperBound\":100,\"parentId\":0,\"nodes\":[],\"rangeChanged\":false}"
```

Now to enable the Live tree processing there is one more important library to use, Socket.io. Now Socket.io is really interesting. It works through polling the channel for changes and when it happens it gets pushed to socket rooms/channels. The client app has socket that listens to changes to these rooms.



Functionality:

1. **Create Factory:**

   Creates new Factory given the payload and stores it in the backend Database. Once created returns the object json as reponse with 201.

   URL: https://factory-passport-api.herokuapp.com/factories

   Method: POST

   URL Params: none

   Data Params:

   { "factoryName":"Test",

   "lowerBound":50,

   "upperBound":500,

   "children":[]

   }

   Success Response:

   Code: 201

   Response:

   {"id":"576a10e5e4b0df303bb08ee0","text":"Test Factory","lowerBound":50,"upperBound":500,"parentId":0,"nodes":[],"rangeChanged":false}

   Error Response:

   Code: 400, 422

   Response:

   {"error": "Bad Request", "message": "Factory name is blank.","status": "400"}

2. Update Factory

Updates existing Factory with the given payload. The Factory name, lowerBound and upperBound values are updated. Whenever the range is updated the flag rangeUpdated is set to true and then the existing childNodes are removed. Otherwise the params alone are updated and childNodes are retained.

URL: https://factory-passport-api.herokuapp.com/factories/{factoryId}

Method: PUT

URL Params: none

Data Params:

{ "factoryName":"Test",
"lowerBound":50,
"upperBound":500,
"children":[]
}

Success Response:

Code: 200

Response:

{ "id": "576a10e5e4b0df303bb08ee0","text": "Test", "lowerBound": 50,
"upperBound": 500,"parentId": 0,"nodes": [],"rangeChanged": false }

Error Response:

Code: 400, 404, 422

Response:

{ "error": "Not Found", "message": "Factory not found.", "status": "404" }

3. Delete Factory

Deletes the existing Factory given the factoryId in the path variable.

URL: https://factory-passport-api.herokuapp.com/factories/{factoryId}

Method: DELETE

URL Params: none

Data Params: none

Success Response:

Code: 200

Response: No Content

Error Response:

Code: 404, 422

Response:

{ "error": "Not Found", "message": "Factory not found.", "status": "404" }

4. Add ChildNodes

Add childNodes to a particular factory. Maximum of 15 nodes can be added. The node values are Random and within the range [lowerBound – upperBound]. When added the existing childNodes are replaced.

URL: https://factory-passport-api.herokuapp.com/factories/{factoryId}/childNodes

Method: POST

URL Params: numChilds [Integer < 15]

Data Params: none

Success Response:

Code: 200

Response:

{ "id":"5769f428e4b0ca415b1a07c2","text":"My Awesome Factory","lowerBound":25,"upperBound":100,"parentId":0,"nodes":[{"text":92},{"text":57},{"text":98},{"text":63},{"text":98},{"text":56},{"text":93},{"text":58},{"text":92},{"text":47}],"rangeChanged":false }

Error Response: 400, 404, 422

Response:

{ "error": "Bad Request","message": "The Number of childNodes should be within 1 and 15.", "status": "400" }

5. **Fetch Factories**
   Fetches the existing factories in the database. This happens when the app is first loaded.

   URL: https://factory-passport-api.herokuapp.com/factories/

   Method: GET

   URL Params: none

   Data Params: none

   Success Response:

   Code: 200

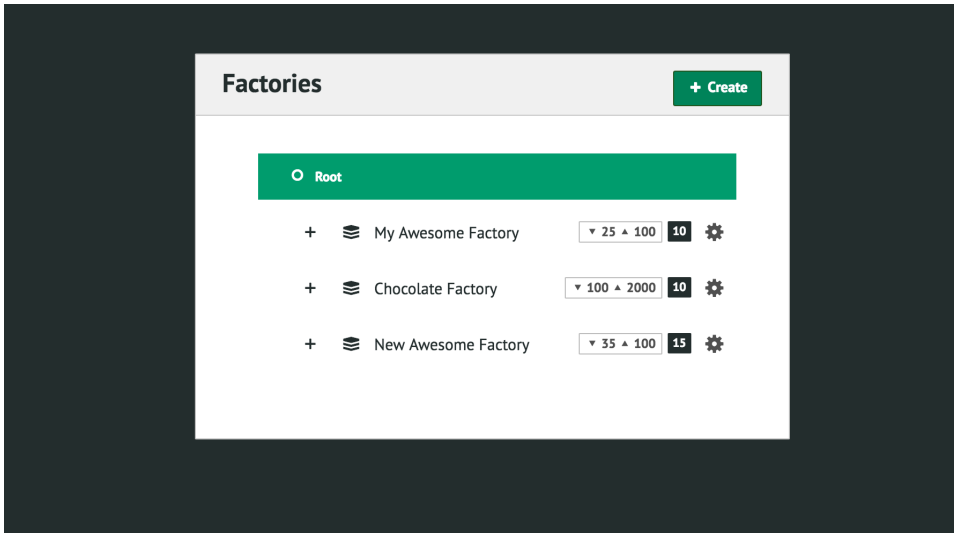
   Response:

[{"backColor":"#019c70","color":"#fff","icon":"icon-cd","expanded":true,"text":"Root","nodes":[{"id":"5769f428e4b0ca415b1a07c2","text":"My Awesome Factory","lowerBound":25,"upperBound":100,"parentId":0,"nodes":[],"rangeChanged":false},{"id":"5769f435e4b0ca415b1a07c3","text":"Chocolate Factory","lowerBound":100,"upperBound":2000,"parentId":0,"nodes":[{"text":420},{"text":1809},{"text":641},{"text":758},{"text":1556},{"text":429},{"text":9

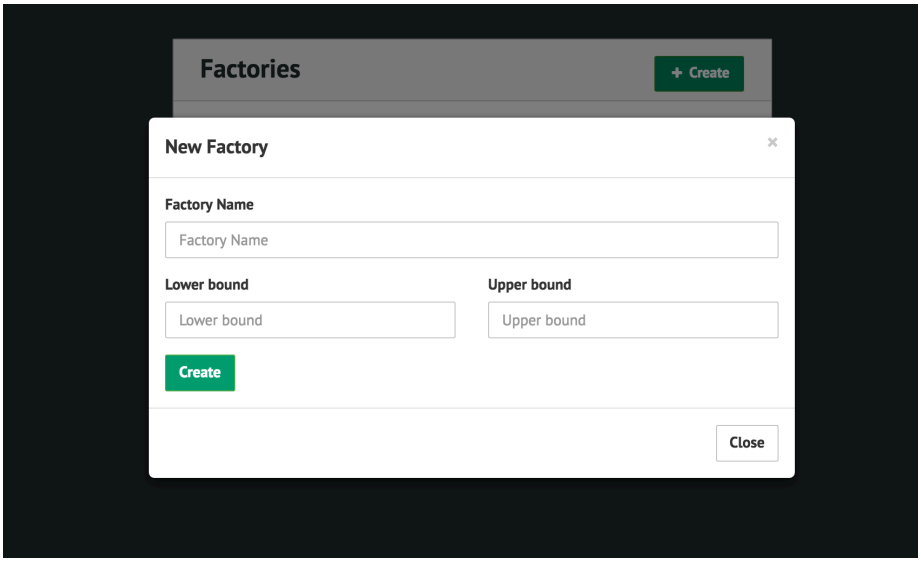
63},{"text":1256},{"text":251},{"text":784}],"rangeChanged":false},{"id":"576a03c4e4b0ca415b1a07c4","text":"New Awesome Factory","lowerBound":35,"upperBound":100,"parentId":0,"nodes":[{"text":63},{"text":78},{"text":63},{"text":76},{"text":53},{"text":99},{"text":100},{"text":78},{"text":76},{"text":59},{"text":48},{"text":88},{"text":56},{"text":97},{"text":57}],"rangeChanged":false}]}]

Client Functionality:

1. The existing Factories are fetched from the database when client is loaded.

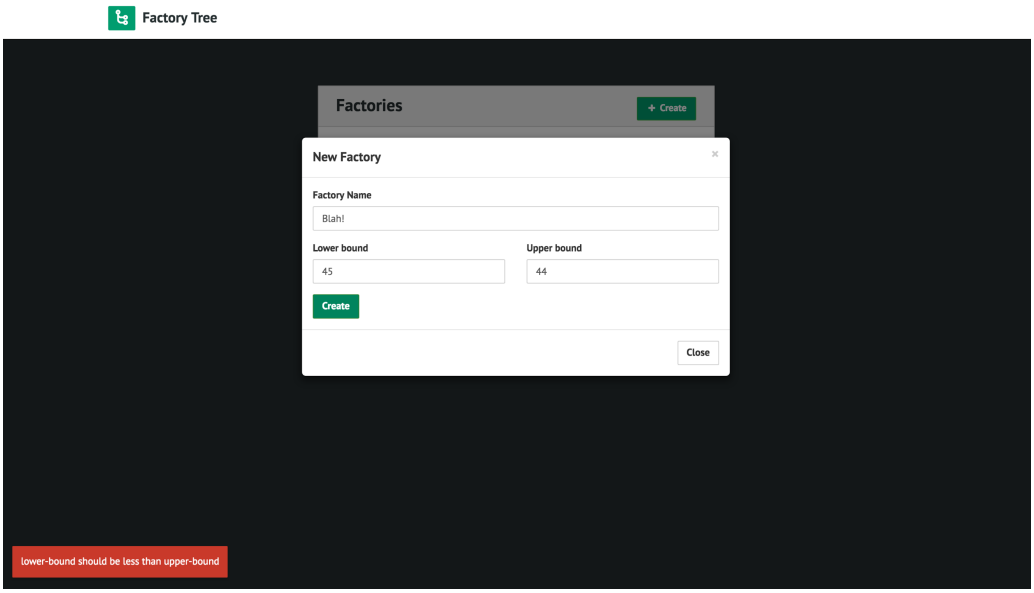


2. Factories can be created by clicking the Create button on the top right. This opens up a Modal window where the factory details are entered.

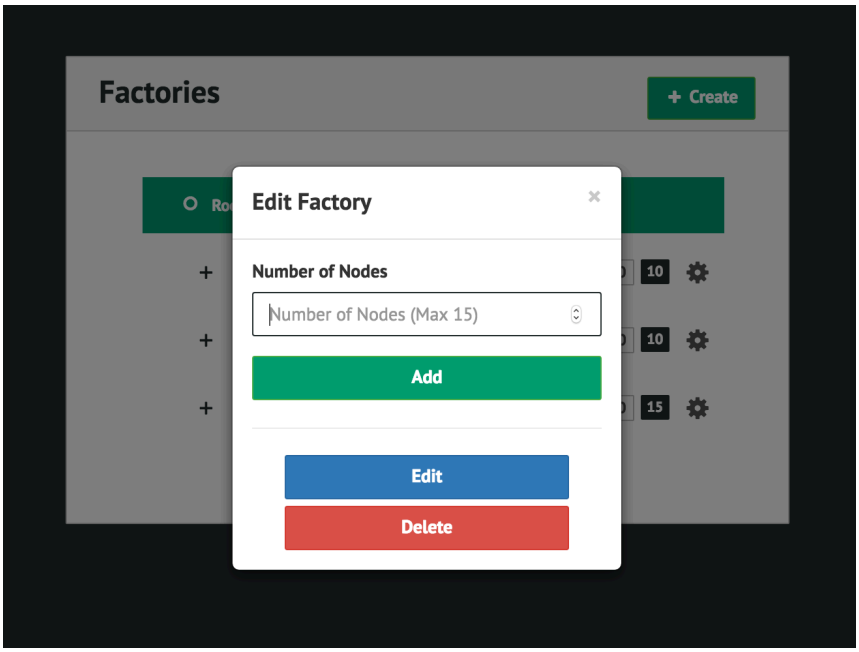


Validated Fields:

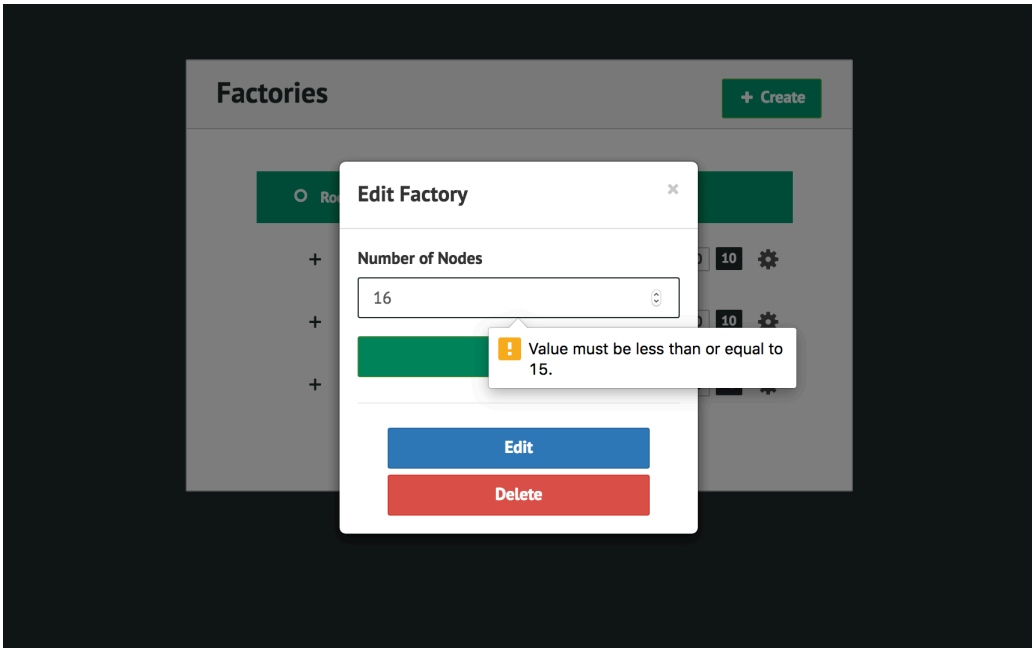
- Factory Name – Text, Max 50 characters, Not Empty
- LowerBound – Numeric, Range 1 to 99999999, Not Empty
- UpperBound – Numeric, Range 1 to 99999999, Not Empty
- LowerBound < UpperBound



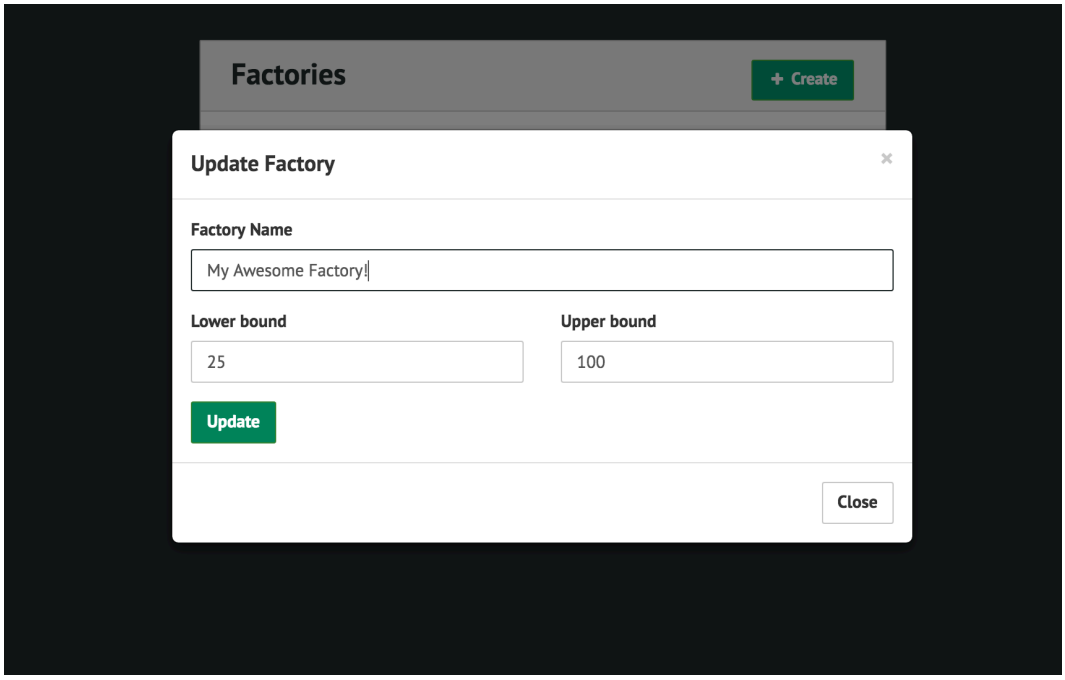
3. On Right clicking the list item or pressing the Settings icon the edit Factory popup comes up. ChildNodes can be added, Factory can be edited or deleted from there.

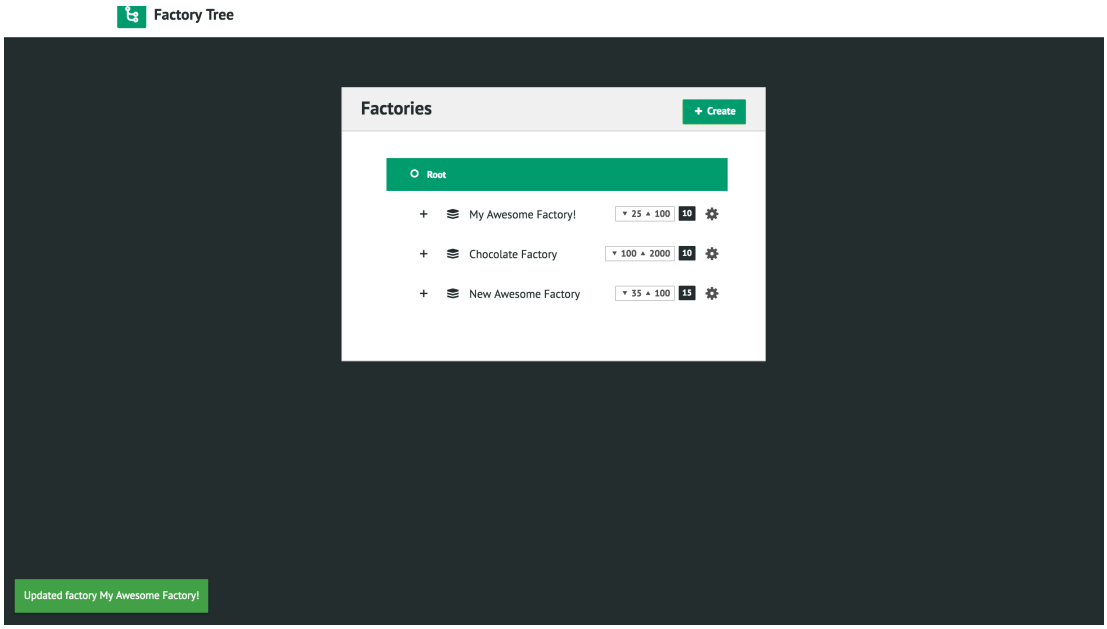


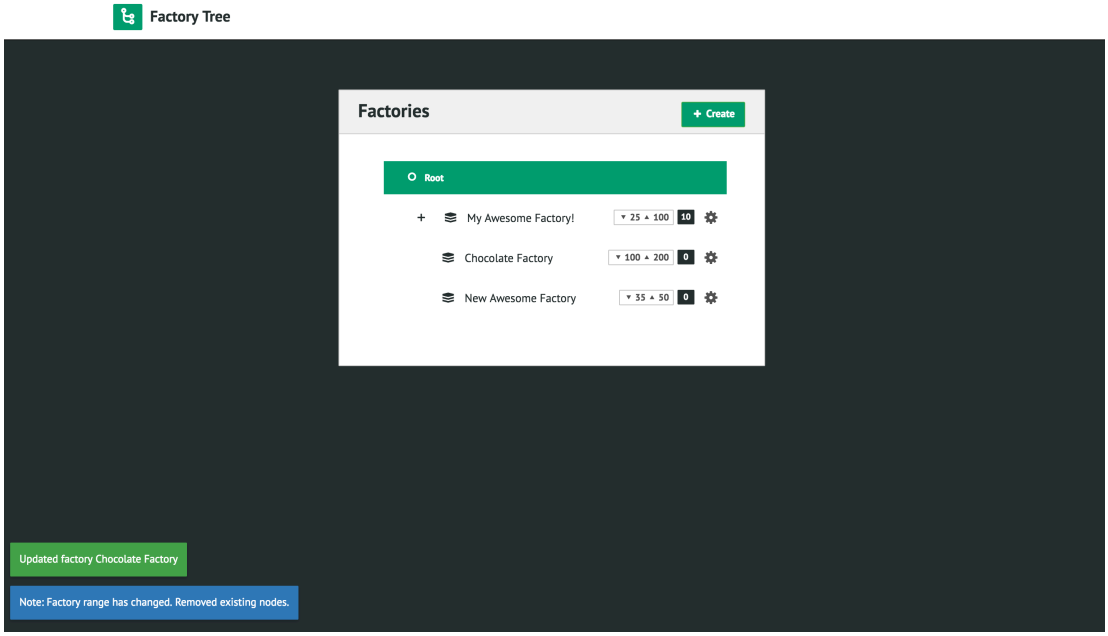
Validations:
Maximum of 15 nodes can be generated. The existing nodes are replaced by the new nodes.



4. Clicking on Edit opens the Update factory modal. The existing values are filled in the form. The same validations as creation is done. If the range values are updated then the existing nodes are removed and user is notified.







5. Clicking on the Delete button removes the Factory.
6. The Plus button expands a particular Factory and shows the existing nodes under it.
7. The Factory Tree is updated in real-time. So anyone who opens this up on a browser will see the changes happening in real-time without browser refresh.

Some Notable Features for the Dev Environment:

- Spring Boot application has properties for multiple Environments – dev and prod.
- The same configurations are added to the Node.js Application as well.
- Using Grunt the asset files are minified and gziped for production. Added the Grunt tasks in Gruntfile.js.
- Works both Locally as well as in Prod based on the configuration provided. Automatically picks up local/remote mongo and redis servers.

Links and ReadME:

1. Source Code: https://github.com/sudharti/passport-developer-challenge
2. Demo: https://passport-factory.herokuapp.com/
3. API: https://factory-passport-api.herokuapp.com/factories

   [Protected by CORS]
4. For setting up check README file on Github https://github.com/sudharti/passport-developer-challenge/blob/master/README.md