Email Service – Node JS Demo App

Contents

[**1.** **Email Service** 3](#_Toc475697153)

[**Problem Statement:** 3](#_Toc475697154)

[**Email Providers:** 3](#_Toc475697155)

[**2.** **Email Service Requirement Specs** 3](#_Toc475697156)

[**3.** **Technical Stack Selection** 4](#_Toc475697157)

[**Reasons** 4](#_Toc475697158)

[**Technology Stack** 4](#_Toc475697159)

[**Packages** 4](#_Toc475697160)

[**React UI – Email App** 4](#_Toc475697161)

[**Email Service Web API** 4](#_Toc475697162)

[**4.** **Node JS Server Application Architecture** 5](#_Toc475697163)

[**5.** **Application Design and Implementation** 6](#_Toc475697164)

[**Email App** 6](#_Toc475697165)

[**Email Service** 7](#_Toc475697166)

[**6.** **Install instructions** 7](#_Toc475697167)

[**Steps to Run NodeJS Service:** 7](#_Toc475697168)

[**Steps to run ReactJS application:** 7](#_Toc475697169)

[**7.** **Testing Methodology and Code Quality** 8](#_Toc475697170)

[**8.** **Deployment** 8](#_Toc475697171)

[**9.** **Demo** 8](#_Toc475697172)

[10. **Checklist** 10](#_Toc475697173)

[**11.** **Pending Features** 11](#_Toc475697174)

[**12.** **Limitation** 11](#_Toc475697175)

# **Email Service**

## **Problem Statement:**

Create a service that accepts the necessary information and sends emails. It should provide an abstraction between two different email service providers. If one of the services goes down, your service can quickly failover to a different provider without affecting your customers.

## **Email Providers:**

* SendGrid
* MailGun
* Mandrill
* Amazon SES

# **Email Service Requirement Specs**

|  |  |  |  |
| --- | --- | --- | --- |
| **REQID** | **Module** | **Features** | **Feature Detail** |
| EMS001 | UI | Compose Email |  |
| EMS002 | UI | Sent Email |  |
| EMS003 | UI | Config Settings |  |
| EMS004 | UI | Provider Priority |  |
| EMS005 | UI | Notifications | Display Notifications of email sent |
| EMS006 | UI | Inbox | Optional feature |
| EMS007 | Backend | Send Email using SendGrid |  |
| EMS008 | Backend | Send Email using MailGun |  |
| EMS009 | Backend | Send Email using Amazon SES |  |
| EMS010 | Backend | Select Email Provider | Based on Provider Priority |
| EMS011 | Backend | Error Handling |  |
| EMS012 | Backend | Error Notifications |  |
| EMS013 | Backend | Update Database with history |  |

# **Technical Stack Selection**

Full-stack - include both front-end and back-end.

## **Reasons**

* Easily have separable functions for managing the business layer and UI layer
* SPA is extremely good for very responsive sites. Server-side rendering is hard to implement for all the intermediate states - small view states do not map well to URLs
* Assume the client will have JavaScript enabled browsers

## **Technology Stack**

* UI
  + React JS,
  + Flux,
  + Bluebird,
  + HTML, CSS3
  + eslint
* Node JS Web Server
  + Express JS, Restify
  + Mongoose library for mongodb
  + bluebird
  + Interface to Email Service Providers
    - SendGrid, MailGun, Mandrill, AWS SES
* Database
  + Mongo DB
* Hosting
  + Internet facing server
* Testing
  + Mocha , Chai for asserts

## **Packages**

Node JS 6.4.1

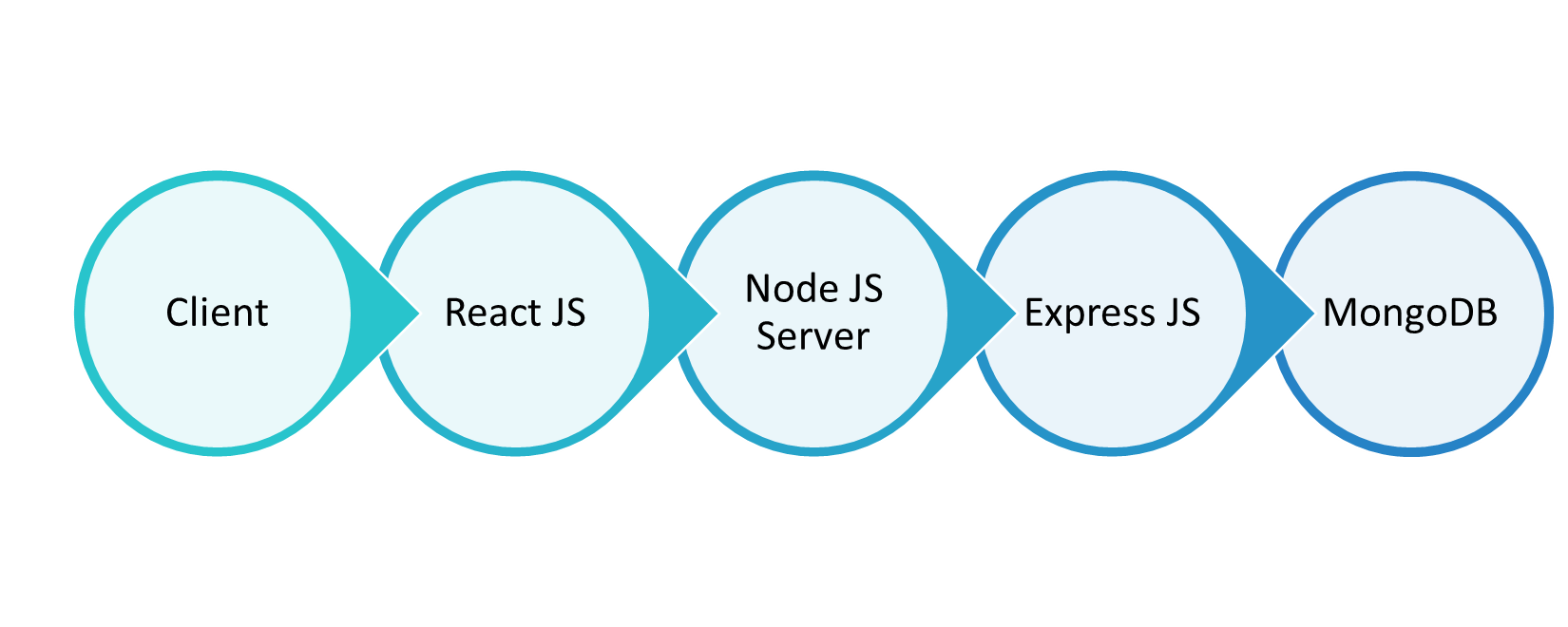
### **React UI – Email App**

* "bluebird": “3.4.7” – promises implementation
* "flux": "^3.1.2" – An application architecture for React utilizing a unidirectional data flow
* "react": "^15.4.2" - A javascript library for building user interfaces from facebook
* "react-dom": "^15.4.2" – React’s Document object model to directly update the DOM as an object
* "superagent": "^3.4.1" **–** to consume promises from bluebird

### **Email Service Web API**

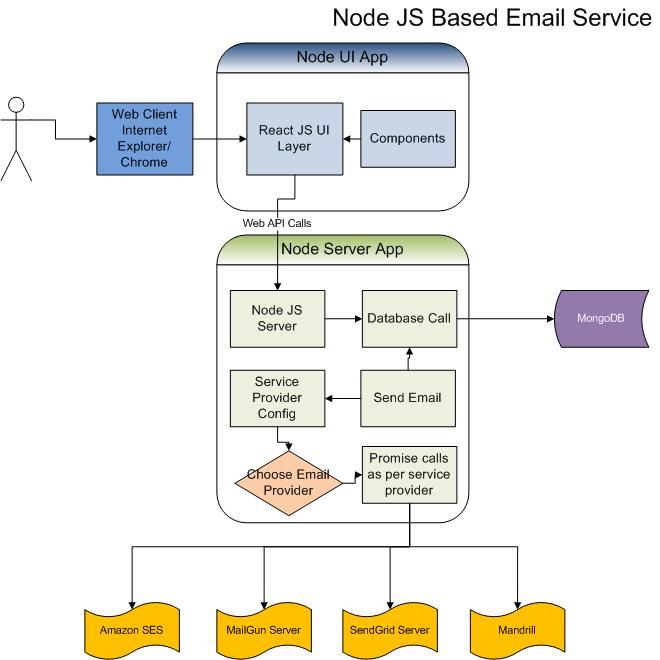
* "bluebird": "^3.4.7" - promises implementation
* "express": "^4.14.1" – a minimal Web application framework
* "log4js": "^1.1.0" – to enable logging in log files
* "mailgun-js": "^0.8.1"- email service provider
* "mongoose": "^4.8.1" – package to connect to mongodb
* "nconf": "^0.8.4" - nconf. Hierarchical node.js configuration with files
* "restify": "^4.3.0" - restify is a smallish framework, similar to express for building REST APIs.
* "sendgrid": "^4.7.1"- email provider,
* "sendgrid-mailer": "^1.1.2 – send grid email provider
* “eslint”: - For coding conventions

# **Node JS Server Application Architecture**



* **Client**: Is the Web client e.g. browser
* **React JS:** is the UI framework for developing UI components
* **Node JS Server**: hosts the application UI and update DOM based on the client interactions by raising the events.
* **Express JS** framework for creating Rest services.
* **Mongo dB** is database calls from the Web API methods (e.g. GET and POST)

Above diagram is a generic application architecture used in typical node JS. Below is a tiered representation of the various services that will be developed to implement the email service application.



# **Application Design and Implementation**

Application uses mongo DB and has 2 main projects Email App and Email Service

## **Email App**

Web UI is developed using flux: application has the following folder structure:

Src\

Components 🡪 ComposeMailForm,ConfigureProvider, ConfigureProviderItem, Footer, Header, InBox, InBoxItem, Logo, MailDetails

Dispatcher 🡪 MailDispatcher

Stores 🡪 MailStore

Utils 🡪 MailAPI

In general application architecture is as shown:



## **Email Service**

This uses ExpressJS App framework was used for application created and is helpful testing web services and based on the template it can be further extended. Testing is also doable

Only following folders are relevant:

Routes 🡪 index – has the routes for GET ad POST for each providers

Services🡪 mailerService

Email Service also connects to MongoDB to store data such as configuration.

# **Install instructions**

## **Steps to Run NodeJS Service:**

1. Open command prompt. Navigate to root directory of NodeJS application.
2. Run command “npm install” to install all dependencies from package.json file.
3. Node server.js
4. Service will start on port 1337.
5. By default priority can be set. If email sending fails on one it will get sent using another.

## **Steps to run ReactJS application:**

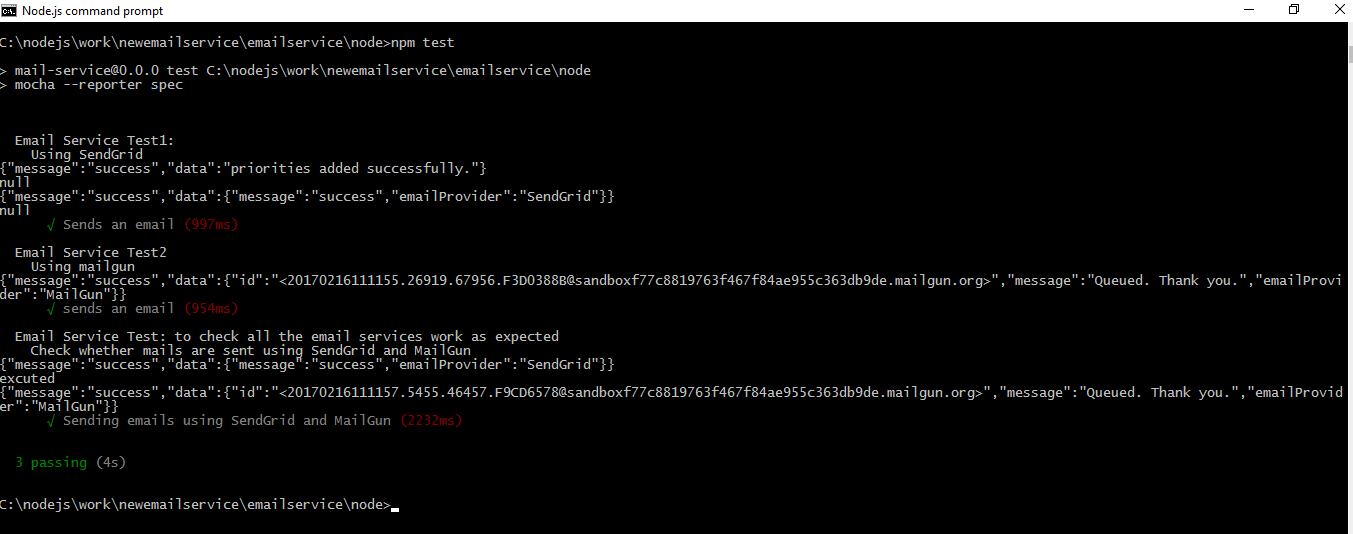
1. Open command prompt. Navigate to root directory of ReactJS application.
2. Run command “npm install” to install all dependencies from package.json file.
3. Run command “npm start” to start application.
4. Service will start on port 3002.
5. Go to browser and type URL <http://localhost:3002/> to run application.

# **Testing Methodology and Code Quality**

Unit testing is implemented in checking whether web service work properly or not.

Use npm test to run the tests the results of 3 tests for each email provider are shown as below;

1. Test SendGrid
2. Test MailGun
3. Test priority – this test sends email by changing the priority test will fail if not sent by set priority and get sent by provider due to error.



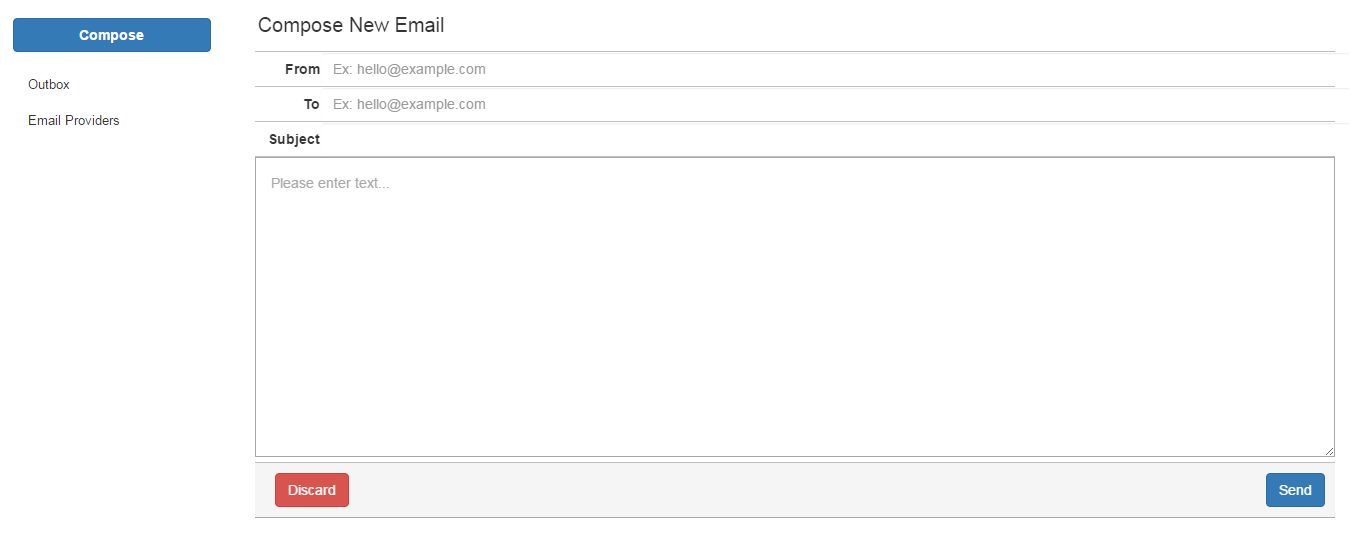
# **Deployment**

Application is deployed live in the internet public IP and can be accessed at:

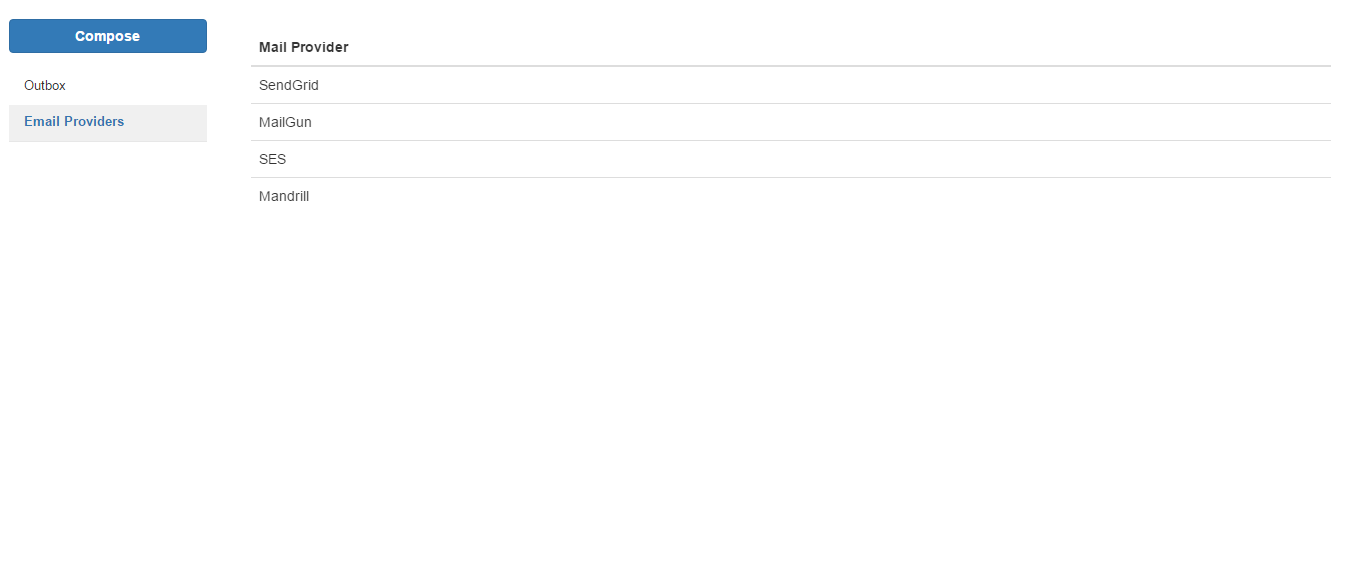
http://114.143.30.179:3002/

# **Demo**

After login to above URL you will see the following:



Email providers and their priority can be set as shown below:



# **Checklist**

|  |  |  |
| --- | --- | --- |
| Index | Description | Checked (Yes/No) |
| 1 | Code Commenting | Yes |
| 2 | Object oriented principle followed | Yes |
| 3 | Security Testing | Limited |
| 4 | Coding Style | Variable names and file names are given such that it makes easier to understand and locate. Also structure is modular as per node.js standards. |
| 5 | Variable naming convention | Yes |
| 5 | Scalability | Application is scalable. Email text and number of email address and are unlimited to the limit provided by the email service providers |
| 6 | Production readiness (deployed) | Yes deployed on WWW |

# **Pending Features**

* Priorities setting is done at config file Level and require an application restart to make changes come into effect.
* Mandrill is paid version and hence not configured
* Amazon SES require EC2 instance and hence will give an error if used.

# **Limitation**

Bluebird promise is being used to send email. It should be noted that if there are errors then it will not get resent.

**Solution:** The solution is to use a node package for queuing and managing the queue.

The queue task is executed as a job and one at a time. The queue is fed into worker. The completed job i.e. email that got sent is removed from the queue and remaining items are attempted again.

**NodeJS package** :  We can use RabbitMQ to implement workers.