Avaya Inventory Management System

June 15, 2015

Jordan Moser
Jin Oh
Erik Ponder
Gokul Natesan

Table of Contents

1. Introduction 1
2. Requirements 2-3
3. System Architecture 4
4. Technical Design 5-6
5. Design and Implementation Decisions 7
6. Results 8
7. Appendix I 9-10

Introduction

Client Description

Avaya is an internationally recognized company that specializes in business solutions. With offices set up worldwide, its headquarters are located in Santa Clara, California. These business solutions are broken down into three categories: Team Engagement, Customer Engagement, and Fabric Networking. Avaya provides technologies and services to large enterprises, mid market companies, small businesses, and government organizations. Through the help of Avaya's services and consultations, customers are able to manage risks and maximize performance while dealing with their business requirements.

The Avaya global demo team currently presents demos at trade shows and to its customers around the world. This particular team at Avaya is in charge of ensuring the correct equipment is sent to various locations for trade shows. As the demand for the global demo team has increased, they have discovered that it is difficult to maintain the precise location for their equipment at all times. Therefore, they need a system for tracking their products as they send them out to various trade shows.

Product Vision

The primary goal of this summer field session project is to create a web application that can keep track of various assets when they are sent out from Avaya. The product is intended to be used by an IT related employee within the company. It is critical for the success of our software that each asset has status and location attributes that can regularly be updated. Additionally, this system needs to allow the administrators to scan QR codes and link to the site to find the current location of the asset or change its status. As well as implementing the tracking API, we need to design a user interface for interacting with the database where this information will be stored. This target time frame for which we intend to complete this project is six weeks.

Requirements

We need a web application that would satisfy the following requirements:

Functional

Assets Page:

- List all assets that have been created
- Show complete asset information after clicking on the asset name
- Enable each asset to be edited if it does not belong to a case
- Enable each asset to be deleted but not removed from database
- Include a quick add to place an asset into a case
- Allow for creating a new asset

Cases Page:

- List all current cases and number of assets in the case
- Show all assets in a case after clicking on a case
- Enable each case to be edited, either adding or removing assets from a particular case
- Enable each case to be deleted, completely removing the case from the database
- Allow for creating a new case and adding assets to that case

Histories Page:

- Display the asset name, status, location, and date the asset is created/updated
- Sort the assets by date created/updated

Miscellaneous:

- Allow for assets and cases to be loaned, include renter's information for the loaned item
- Send email alerts to borrower when return date is approaching
- Include paginations for assets, histories, and cases pages

Non-Functional

- The web application user interface will be written in HTML
- The API, which maintains information regarding about each item, will be written in javascript
- Document the API portion of the project
- The final project will be submitted in a GitHub repository
- When demoing the project, be able to pull up an asset by its QR code

System Architecture

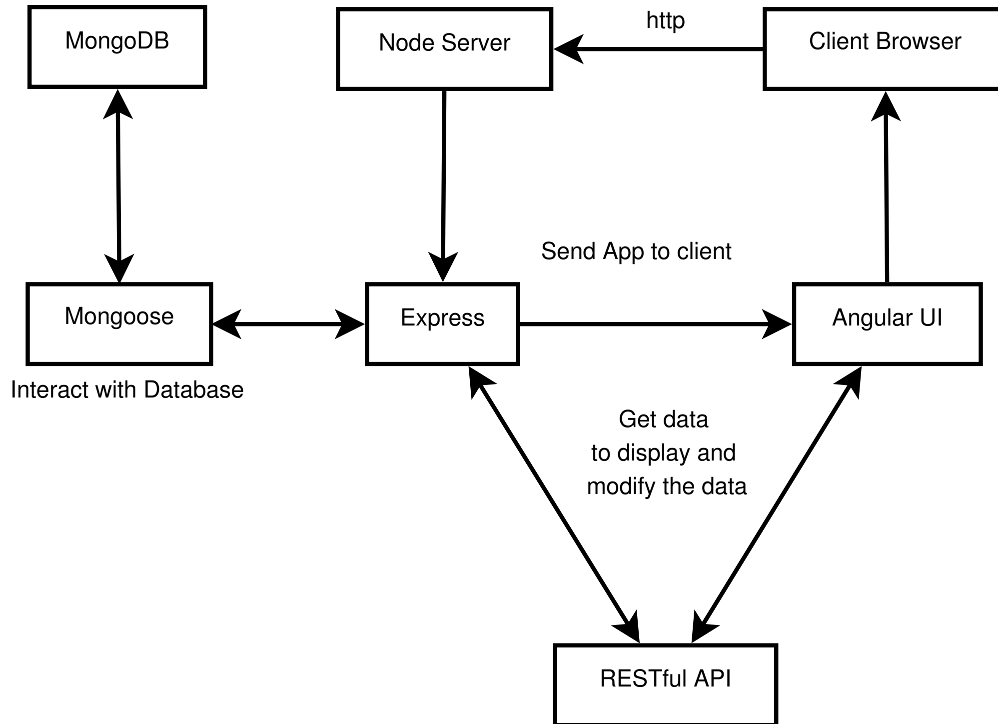

Figure 1: Design

Figure 1 depicts the overall design of our application. The client will browse to our website which will be running on a node.js platform. Express.js will be our backend application that handles all of the servers routes. Our angular application will be the front-end of our application and it will get sent to the clients browser from the express route. The client will navigate the Angular application to create new assets and cases. The Angular app will then send the new data or updates to our Express application via the RESTful routes we have created. The Express app will then use mongoose.js to interact with our Mongo database to update the records for persistence.

Technical Design

Our database will be created with the following schema to facilitate our needs for this application.

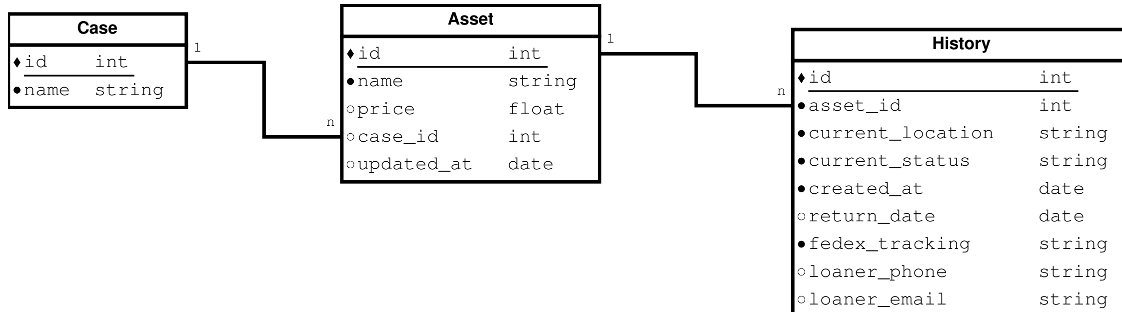

Figure 2: Schema

Our schema has three main models which are important to the database: case, asset, and history. Figure 2 helps depict their individual attributes as well as the relationship between these different models. Each asset has a unique id number and a name. Additionally, it can come with a price and if it belongs to a case, a case id. An asset can also have a return date depending on its status. The cases model is extremely simple. A case will consist of a case id and a name. Cases are related to assets such that a case can contain multiple assets. Finally, there is a history model designed so that each asset will have a collection of history records. The history will contain its own id number that will not be shown. This attribute is strictly for database purposes. An asset's history holds the asset id and records the fedex tracking number, current location of the asset, current status of the asset, as well as loaner contact information, and a timestamp for when the history record was created.

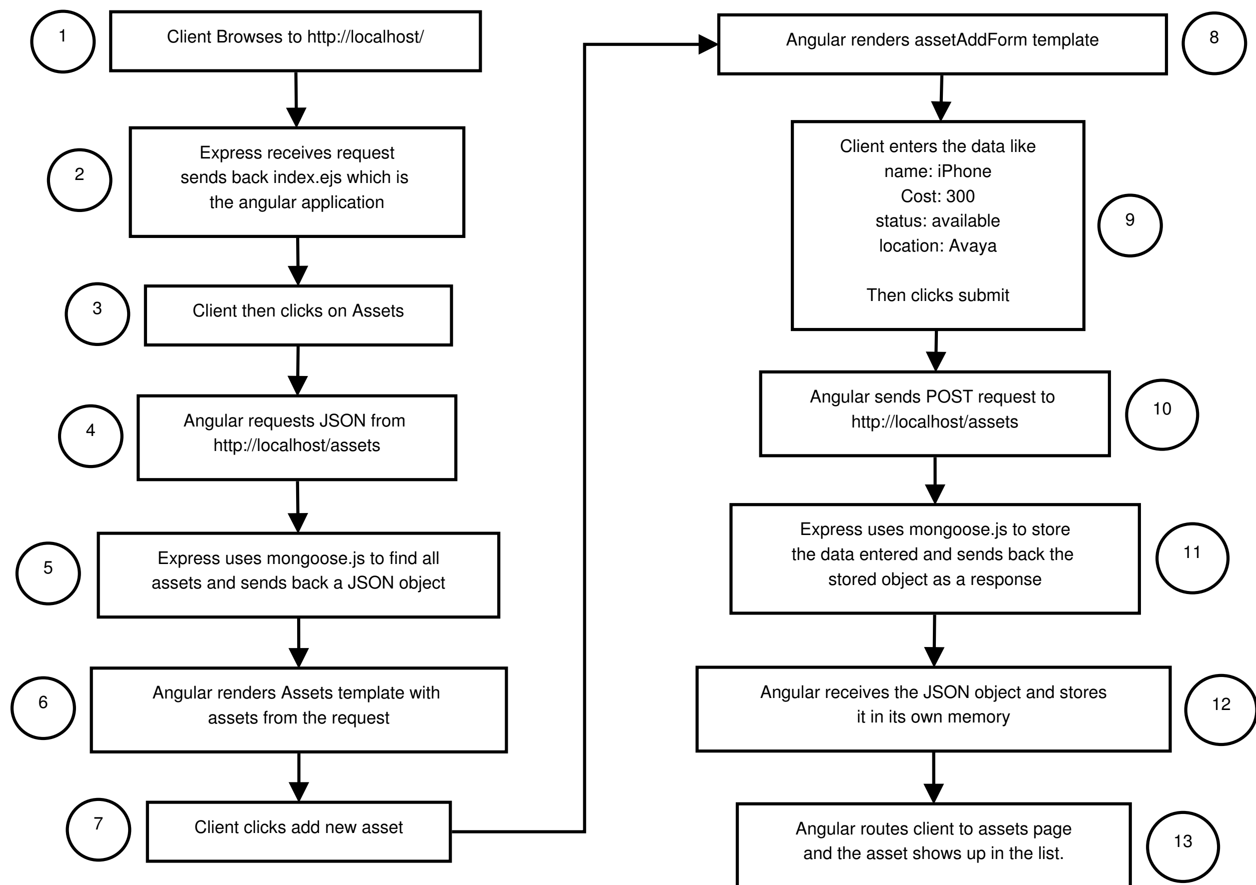

Figure 3: Process for Adding an Asset

Figure 3 depicts a simple process of adding an asset to our database and the interactions that Angular has with Express. Step 1 begins with the user navigating to the application like they would to any other website. Step 2 is where we introduce Node.js and Express.js. Node is the javascript runtime running the application and Express provides the framework and routing for this application it is similar to Ruby on Rails or Laravel. Angular is a javascript framework that runs on the client browser and generates all the pages that the user sees. Step 3 and 4 illustrate our angular application interacting with our RESTful routes we designed in Express. Step 5 and 6 document the exchange of information from our Express application and Mongo with our Angular application. Angular sent a request for data in step 4 which is handled by the Express route. In this route, mongoose.js and our asset model is used to pull data from the Mongo database. This data is then packaged into a JavaScript Object Notation (JSON) format which is the response that Express sends back to angular. Angular then injects the data received into an HTML template via data binding that is a feature of Angular. The figure then continues to illustrate how a POST request submitted via a form can then be stored in the Mongo database via mongoose and the asset model (steps 7-12). Finally when the data has been persisted to the database, Angular redirects the user to the assets page once again where a request is made for all the assets like in step 4. This process of gathering data and storing data via HTTP requests from the Angular framework to the backend is what makes up the MEAN stack utilized in this application.

Design and Implementation Decisions

- Our first design decision was to determine what framework to use to create our application. We had considered frameworks like Ruby on Rails, PHP frameworks, and the MEAN stack. We were informed by our client that he is currently using the MEAN stack. This allowed us to implement our entire application using just javascript, HTML, and some CSS.

- Instead of implementing our own CSS, we decided to use the bootstrap framework. This made styling our application significantly easier and allowed us to focus our intentions on the logic of the application rather than the aesthetics. However, we did not need to sacrifice the look of our application.

- Our next design decision was to pick a database for our application. Conveniently the MEAN stack provides us with Mongo DB. Mongo is not like a MySQL database and as such we needed to figure out how to connect to the database. There is a javascript library called mongoose.js that allows the application to communicate with the mongo database seamlessly.

- Since our client wanted to be able to audit their assets, we needed to keep a permanent history of all the assets. Our next design decision was to "soft delete" items from the database. This way we can keep a history of the asset without permanently removing it from our database. If there is any need for an audit, the client can easily find assets they currently own and items that they loaned.

Results

Here are the results of our application:

- All functional and nonfunctional requirements of the project satisfied
- Works on various web browsers such as Chrome, Firefox, and Safari. Not tested on Internet Explorer
- Asset page can be pulled up by scanning the item's QR code

For the duration of this 6 week session, we gained valuable experience by working on an application that would be used in an actual industry. Here are some of the lessons we have learned through this project.

- Don't underestimate the complexity of a project. After reading the description of the project, we expected this project to take no longer than a couple weeks. But not having any previous knowledge of the frameworks added more complexities to the project. It took our group the entire 5 weeks to finish the project.
- Agile development is an effective method to work in a group environment. Communication is essential to the success of a group through scrums, retrospectives, and planning.
- Web hooks are very useful for intercepting events in your web application and allowing you to perform an action. However, we didn't understand the concept of web hooks and their uses until we implemented them. They allow the client to receive updates when an asset is shipped or when a loaned item is expiring and they need to contact the borrower to get the item back.

Appendix I

API Documentation

Routes

METHOD	ROUTE	EFFECT
GET	/	The main route of the application, Angular lives here
GET	/possible_statuses	Fetches all enumerated statuses
GET	/assets	Gets all the assets from Mongo
POST	/assets	Creates a new asset
GET	/assets/:asset	Gets a particular asset by _id
PUT	/assets/:asset	Updates a particular asset by _id
DELETE	/assets/:asset	Deletes a particular asset by _id
GET	/recent	Gets 10 assets with the most recent updated time
GET	/histories	Gets all the history records
POST	/assets/:asset/histories	Create a history and attach it to the asset
GET	/cases	Get all the cases
GET	/cases/:case	Get a particular case by _id
PUT	/cases/:case	Update the name of the case
POST	/cases	Create a case
DELETE	/cases/:case	Delete a case and remove all the assets from that case
PUT	/cases/:case/assets/:asset	Add asset to a case by _id
DELETE	/cases/:case/assets/:asset	Remove an asset from the case
GET	/locations	Gets all the locations that have been entered by users
POST	/locations	Adds new location to list of locations
POST	/hooks	Add web hook to database

Models

Asset Model

```
name: String,
price: Number,
updated_at: Date,
```

Case Model

```
name: String,
```

History Model

```
fedex_tracking: String,
current_location: String,
current_status: {type: String,
  enum: ["Available", "On Route",
  "Checked Out", "Lost", "Loaned",
  "Deleted"]},
return_date: Date,
created_at: Date,
asset: {type: mongoose.Schema
  .Types.ObjectId, ref: 'Asset'},
loaner_phone: String,
loaner_email: String
```

Location Model

```
name: String
```

Hook Model

```
server: String,
post_url: String,
status: {type: String,
  enum: ["Available", "On Route",
  "Checked Out", "Lost", "Loaned",
  "Deleted"]}
```

Error Model

```
message: String,
stack: String,
status: String,
headers: String,
body: String
```