Blizzard Kicker

The Apple/Peeler/Corer/Slicer Team

Kamiar Coffey, Ryan Davis, Chad Di Lauro, Asher Farr,
John Henry Fitzgerald, Matt Skogen

Index

- Project Overview
- Tools Used
- Live Demonstration

Project Overview

Problem Statement

The issue: During winter in Colorado, one of the biggest decisions people are forced to make on a daily basis is deciding which ski resort would be the most fun to go to on any given day. Snow conditions, driving times, weather, ski pass availability, and other factors can make this extremely complicated.

Team Apple Peeler Corer Slicer (APCS) will create a web app product that will make all its users' ski trip experiences better in ways that specifically tailor to each user's preferences.

Blizzard Kicker

Blizzard kicker is a web app by APCS that gives its users recommendations on the best ski resort to maximize their time on the mountain:

- Navigate to site and log in or register
- Set preferences for the account
  - HTML buttons have preset criteria and values for users to choose from
  - Preferences are saved to MongoDB database
- Navigate to your user cave to get a recommendation based on preferences
  - Data is pulled into database using an API and scraped from snow report
  - Updates when “view ski data” is clicked
- Algorithm compares user preferences and current conditions to pick best option
- Display shows best resort based on preferences and a map

Tools Used

Database: MongoDB using Mongoose

- MongoDB can store JSON documents, but we defined our own Schema and parsed Strings/Numbers for input
- Handles asynchronous processes with *Promises* .
- Mongo structure:
  - Database
    - Collections
      - Models
        - Schema

```
var UserSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  username:{
    type: String,
    unique: true,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
```

Handling Asynchronous Queries Updates To MongoDB

```
//update user's resort list
UserSchema.statics.updateResortList = function(email){
  User.findOne({email: email}).exec()
  .then((user)=>{
    user.loadResorts();
    return User.updateOne({email:email}, user).exec()
  })
  .then((info)=>{
    console.log(info);
  }).catch((err)=>{
    console.log(err);
  })
}

var User = mongoose.model('User', UserSchema);
module.exports = User
```

User.findOne.exec()
// Returns a asynchronous promise

Promise *must* be resolved

Templating: Pug

Why We Chose Pug:

- Check user input validity on the front end
- Ability to customize pages without reloading HTML files
- Cleaner syntax

```
cextends ../layouts/layout
block append head
  link(href='/css/register.css', rel='stylesheet')
block vars
  - var currentPage = "register"
block content
  //highlight current
  .col-md-5.mx-auto.mt-5
    form.form-container.form-signin(action='/register' method='POST')
      .form-body
        h1.mb-3.font-weight-normal Register
        label(for='inputEmail') Email address
        input#inputEmail.form-control.mb-3(name='email'
          type='email'
          placeholder='Email address'
          required
          autofocus='')

        label(for='inputUser') Username
        input#inputUser.form-control.mb-3(name='username'
          type='text'
          placeholder='Username'
          required
          )
        //- single lowercase, single uppercase, single digit, single symbol, at least 8 long
        -var passPatern= '(?=.*\\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[!-\/:-@[-`~])\\S{8,}'
```

Routing/Login: Node.js with ExpressJs

Cookies:
When a user logs in a unique session cookie is created which stores their email as a user ID. This is then included with HTML requests and so is shared across JavaScript files to access the correct table in the database.

```
// in login.js
req.session.userId = user.email;
req.session.userName = user.username;

// in cave.js
currentUser = req.session.userId
```

Encryption with bcrypt Hashing:
Keep passwords secure

```
// only hash the password if it has been modified (or is new)
  if (!user.isModified('password')) return next();
  // generate a salt
  bcrypt.genSalt(SALT_WORK_FACTOR, function(err, salt) {
      if (err) return next(err);
      // hash the password using our new salt
      bcrypt.hash(user.password, salt, function(err, hash) {
          if (err) return next(err);
          // override the cleartext password with the hashed one
          user.password = hash;
          next();
      });
  });
```

Testing Tool: Travis CI

★★★☆☆

Continuous Integration:

- Integrates with MongoDB, Node.js, and JavaScript
- Check dependencies, run unit tests, and test functionality
- Experience from lab 12

Deployment Environment: Docker

- Docker is a program that allows Blizzard Kicker to run on an identical environment, no matter what machine it runs from
- Works by creating a “container”, then running startup commands on that container
  - Container can be thought of VM specifically configured to our needs
- Eliminates need to run different ways for different machines
- Creates mysterious errors on some machines, which nearly eliminates its purpose

Project Management Style: Agile

- Splits entire project into smaller steps called sprints
- Each sprint completed over ≅ 2-4 weeks
- Completed sprints add increased functionality to current product
- Blizzard Kicker split into four different sprints:
  - Initial frontend/backend setup and deployment with npm
  - Add in database functionality and API functionality
  - Include ability for users to set preferences and store necessary API data (both in database); change deployment environment to docker
  - Finalize selection algorithm and perform testing

Demo Plan: See Video

- Launch web app
- Create account
- Set preferences
- Find best resort
- Change preferences
- See results
- Log out

Questions?