Serverless Single Page Web Apps, Part One

CSCI 5828: Foundations of Software Engineering
Lecture 20 — 10/27/2016
Goals

• Introduce our second textbook:
  • Serverless Single Page Web Apps by Ben Rady
  • Discuss what you need to do to use the example code in the book
Example Code (I)

- This book makes use of a "prepared environment"
  - A GitHub repo that contains software to help automate interactions with Amazon Web Services; *to allow us to focus on learning the content*
- The GitHub Repo is located here:
  - [https://github.com/benrady/learnjs.git](https://github.com/benrady/learnjs.git)
- **Do NOT clone this repository directly**
  - Instead, the book asks you to fork it
- Let's work our way through that process…
Example Code (II)

• To fork a repository, visit it in your web browser:
  • https://github.com/benrady/learnjs.git

• Click on the Fork button in the upper right (and follow any instructions)
  • This will create a copy of the repository in your user account

• Now, clone your own copy of the forked repository to your computer
  • So, for me, I would go to my laptop, and do something like
    $ cd Projects
    $ git clone git@github.com:kenbod/learnjs.git

• You should execute similar commands on your machine but use the URL from your own account NOT my account
Example Code (III)

• Now, you need to configure your repository such that you can get updates from the original repository (if and when they appear)

• To do that, on your local machine, go to the learnjs directory and type

  • $ git remote add upstream https://github.com/benrady/learnjs.git

• Then verify that the remote has been set-up correctly:

  • $ git remote -v

• You should see something like:

  • origin  git@github.com:kenbod/learnjs.git (fetch)
  • origin  git@github.com:kenbod/learnjs.git (push)
  • upstream https://github.com/benrady/learnjs.git (fetch)
  • upstream https://github.com/benrady/learnjs.git (push)

• Your origin URL will point to your own forked copy of the repo, however
Example Code (IV)

- We will be making changes to our local copy of the repository as we work through the chapters of this book
  - What should we do if the author publishes new commits to the original repository?
- The basic approach is the following
  - $ git stash
  - $ git fetch upstream
  - $ git checkout master
  - $ git merge upstream/master
  - $ git push
  - $ git stash apply
- What does this do?
  - (See next slide)
Example Code (V)

• Here's what the commands on the previous slide accomplish

  1. **git stash**: Save your changes and set them aside; your repository goes back to the state stored in the HEAD of the current branch

  2. **git fetch upstream**: Fetch the changes from the original repository; the changes are downloaded but NOT applied

  3. **git checkout master**: Make sure you're on the master branch

  4. **git merge upstream/master**: Merge the changes from the original repository to your local master branch; if you committed any changes to your local repository that conflict, you'll need to resolve the conflicts

  5. **git push**: Assuming no conflicts, this command pushes the changes from the original repository to your forked repository on GitHub

  6. **git stash apply**: Retrieve your changes saved in step 1 and apply them to the newly updated repository
Example Code (VI)

GitHub

forked repo .............. original repo

origin

upstream

local repo

Your Machine
Example Code (VII)

- Contents of the example code
  - A shell script called `sspa` (used to automate various tasks)
  - A public folder that contains our initial website
  - Various support and configuration folders
- Dependencies
  - To run the shell script, you need to have python 2.7
  - You will also need to have the Amazon Web Services CLI
    - To do that, make use of Python's package manager, pip
      - `pip install awscli` or `sudo pip install awscli`
      - If you don't have pip, try: `(sudo) easy_install pip`
Serverless Web Applications

• Our goal is to look at a class of web applications known as "serverless" apps
  • These apps stand in contrast to most traditional web application frameworks: Ruby on Rails, Django, etc.
    • These frameworks help you develop web applications that live on the server and generate HTML/CSS/Javascript that executes on a client machine in response to HTTP GET/PUT/POST/DELETE requests
  • With serverless apps, your first request to a server, downloads a set of HTML/CSS/Javascript that then handles all aspects of the web app **within the browser on the client machine**
    • The server is used initially to get those files and then may be used to respond to requests made on web services hosted on the server
    • Or not… we might use web services hosted on OTHER servers
How is this possible?

- Web applications used to be located on the server side (and the vast majority still are) to handle things like
  - user credentials, storage of data, ability to make calls on 3rd party services
- But now, you can avoid the traditional n-tier architecture of web apps
  - client browser => load balancer => web server => app server => database
  - and instead
    - use the web server as a delivery mechanism
    - all application logic lives in the browser
    - 3rd party web services handle everything else: user accounts, data, etc.
- All due primarily to the evolution of web standards: HTML5, ES6, CSS 3
Benefits of Serverless Design (I)

• Avoid having to understand a complex web application framework

• No more servers! :-)  
  • You no longer have to worry about maintaining physical servers; you will instead host your app’s files on a 3rd party service that simply delivers the app to the browser; someone else performs security updates, maintains file systems, etc.

• Easy to Scale
  • You can rely on cloud service providers to scale your application; our textbook looks at how AWS can help us scale up to large amounts of data and users if we need to
Benefits of Serverless Design (II)

• **Highly Available**
  - You don't have to bring your system down to upgrade it
  - You simply deploy a set of static files to the hosting service
    - Your users will see the update on the next full refresh on the client

• **Low Cost**
  - For small applications, your computational demands will typically stay in the range of a service's "free tier"; if you're at that level and your trial period expires, your costs are often "pennies per day"
    - When your needs go up, this approach still scales nicely; the book claims that its example app could scale to 1M users and still costs only "dollars per day"
Benefits of Serverless Design (III)

• **Microservice Friendly**

  • By running all your code in the browser, you can easily integrate new microservices or web services into your app

    • These services will follow OAuth workflows to "login" as a particular user and then be able to make calls on behalf of that user

    • You're not in the business of storing any of that data; the data is stored on the web service's computers

    • Your app might store some client-specific data but in the form of cookies on a client's machine or in a 3rd party service

  • So, once again, you avoid the headaches that come with server side frameworks: how do I store client data in a safe and secure manner?
Benefits of Serverless Design (IV)

• **Less Code**
  
  • There is often a duplication of code that exists when using traditional web application frameworks

  • There's the HTML/CSS/Javascript in the client; it has logic about how to interoperate with the code on the server

  • There's framework code on the server that has to respond to those interactions

  • Change one, you need to change the other

  • With serverless apps, all of this logic resides in one place: the client
Limitations

• *It's not all rainbows and butterflies*
  
  • adopting serverless web apps brings limitations and new techniques that are likely unfamiliar

• **Vendor Lock-In**
  
  • Our textbook makes use of AWS services; migrating to Google Compute Engine would not be trivial

• **Logging**
  
  • With traditional frameworks, all of your logging is done in one place; with serverless apps, your logs might be distributed across multiple services

• **Security and Identity Models**
  
  • Validating data becomes tricky with serverless apps; Identity/Logins makes use of 3rd party services that are initially unfamiliar

• **Big Money**: Usage spikes could impose BIG charges; you have to plan ahead
The Example App

• Our textbook is going to spend its time developing an application called LearnJS

• It will be a quiz application that provides simple JavaScript questions to users who can submit answers and then see the results

Problem #1

What is truth?

```javascript
function problem() { return __; }
```

CHECK ANSWER
Three Main Files

• Our app consists of three main files
  • index.html — content
  • app.js — logic
  • app_spec.js — tests

• In index.html, we can review the libraries that we depend on
  • Normalize, Skeleton, jQuery
  • vendor.js references JavaScript libraries that we need for AWS

• To run this on our local machine, head to the learnjs directory and type
  • $ ./sspa server
  • Then visit http://localhost:9292
First Change

• Let's replace the boilerplate in index.html with the start of a landing page

```html
<body>
  <div class='container'>
    <div class='row'>
      <div class='one-half column'>
        <h3>Learn JavaScript, one puzzle at a time.</h3>
        <a href='' class='button button-primary'>Start Now!</a>
      </div>
      <div class='one-half column'>
        <img src='/images/HeroImage.jpg'/>
      </div>
    </div>
  </div>
</body>
```

• We will learn about HTML, JavaScript, and CSS via osmosis
Let's Deploy

• Just like your semester projects, the first issue the book wants to tackle is deployment!
  • Since we don't need a server—just a place to deploy static files—we can get away with Amazon's S3 file service (Simple Storage Service)
• To make use of that, we need to create an account on Amazon.
  • What I did was I created a new e-mail account on Google
    • I then used that account to sign up for AWS at
      • https://console.aws.amazon.com
• Page 16 and 17 of the textbook step you through the process of then creating a user via the Identity & Access Management service
  • We use that "user" to generate the tokens we need to access AWS and to assign that user the rights to make use of those services
What will this cost?

• For the start of this project, we won't be taxing AWS in any way. We will stay within their "free tier" for quite a while.

• If/when our free tier expires, the size of the files that we're putting on S3 will only cost a few pennies per month
  • Nothing that will break the bank
Create a "bucket"

- Amazon S3 has the notion of a "bucket"
  - You create a bucket and then store things inside of it
  - Those items get referenced by things that look like file system paths
  - Those paths can then be combined with an "http" prefix and suddenly you have a URL that provides you with access to the data you put into the bucket
- To create the bucket, we use the sspa script:
  - `$ ./sspa create_bucket <bucket name>`
- I used:
  - `$ ./sspa create_bucket csci5828-f16-kena`
- You'll get back a URL like this:
  - `http://<bucket_name>.s3-website-us-east-1.amazonaws.com`
  - `http://csci5828-f16-kena.s3-website-us-east-1.amazonaws.com/`
Finally: Deploy the website

- We deploy our website to S3 using the `sspa` script
  - `$ ./sspa deploy_bucket <bucket_name>`
- Or, in my case,
  - `$ ./sspa deploy_bucket csci5828-f16-kena`
- You can then visit the URL on the previous slide to see your deployed website
- This initial work sets us up to explore the mechanics of single-page web apps next week!
Summary

- Introduced the notion of serverless single page web applications
  - Discussed benefits and limitations
  - Retrieved the "prepared workspace" and configured it
  - Took our first steps in developing and deploying our application

- Next week:
  - Chapter 2: Routing Views with Hash Events