FAC4

Founders & Coders

Published
with GitBook

Table of Contents

This is a reference, of sorts, for Founders & Coders, a peer-led course in software development for the Web. It is the fifth course we have run since Dan ran *Coding the web* in January 2014 and the first course at our new home in Globe Town.

http://foundersandcoders.gitbooks.io/fac4/

History

- *Coding the web*, January-February 2014
- *FAC1*, March-May 2014
- *FAC2*, June-August 2014
- *FAC3*, September-November 2014
- *FAC4*, January-March 2015

Founders & Coders

We are a self-funded social enterprise run by its membership as a freelance co-operative. The membership is made up entirely of course mentors and alumni.

According to documents lodged at *Companies House*,

```
The company’s activities will provide benefit to our members,
to students of our study programmes, and to the wider public
```

And our activities include:

```
Providing programmes of study for the general public.

Providing a place of learning and work for our members.
```

And,

```
If the company makes any surplus it will be used for developing further
programmes of study and for developing further places of study and work.
```

That's it.

How do we intend to fund ourselves?

The short answer is, *We don't know*.

The longer answer is that we are trying different things and we expect one or more of them will eventually work. These include:

1. Finding work for our graduates and taking a proportion of the fees they earn from clients;
2. Acting as a recuitment agency for our graduates and taking fees from employers;
3. Developing training courses that we can charge for;
4. Accepting commercial sponsorship;
5. Applying for public funds;
6. Asking for support from successful alumni.

Please let us help you find work before you start looking yourself. If you see a job you like the look of, please let us contact the employer on your behalf.

Schedule

- January 26: Course starts
- March 23: bonus week - build your own MVP
- March 30: Client MVPs - Round 1 (over two short weeks)
- April 13: Client MVPs - Round 2
- April 20: Client MVPs - Round 3
- April 27: Start work on longer projects
- May 11: Summer cohort start.

Facilities

You are welcome to use the kitchen area, but please clean up after yourself and keep the surfaces clear.

We have a shower room, but we do not recommend using it until we get the ventiliation fixed.

We have one loo. Hopefully it will be enough, but this space is now shared between over 30 people, so it will get well used.

Timetable

Updated: 15 February, 2015

The space opens at around 9.30am and closes when the last person leaves, usually by around 9pm.

Classes start at 10am and end at 6pm Monday to Friday.

The provisional timetable is as follows.

- *10am* Morning coding challenge
- *10.30am* Morning scrums
- *2pm* Quick talks
- *5pm* Sprint reviews and invited speakers

Mondays

- *10.30am* New project for the week
- *5pm* Review of sprint goals

Tuesdays

- *2pm* individual catch ups
- *5pm* MVP pitch

Wednesdays

- *3pm Aim to get first sprint done
- *3pm Sprint retrospective and knowledge sharing
- *5pm* First sprint reviews*

Thursdays

- *5pm* MVP pitch

Fridays

- *2.30pm Aim to get first sprint done
- *2.30pm* Weekly retrospective: stop, start, continue
- *3pm Sprint retrospective and knowledge sharing
- *4.30pm* Employer talk
- *5pm* Final sprint reviews

Provisional syllabus

Don't take this syllabus too seriously. It is likely to change.

Week 1: development environments, source control, agile development, testing and building static websites

PivotalTracker, Git, GitHub, GitHub pages, Jekyll, HTML, CSS, Bootstrap, Sass, Mocha, Supertest. Setting up on OS X, Windows or Unix.

Week 2: client-side programming, JavaScript, linting, DOM, AJAX, APIs

Prerequisites:

- Discover DevTools course at Code School.

Weeks 3: server-side programming and APIs with Node.js

Prerequisites:

- Real-time Web with Node.js, Level 1

Week 4: Build tools and more Node.js

Week 5: MVC, Databases, REST APIs and yet more Node.js

Week 6: Data visualisation

Week 7: AngularJS

Week 8: More AngularJS

Staff

This is the most exciting change to the course this time. The first four versions of this course were taught more-or-less solo by Dan with a bit of invited help from outside speakers.

This time we have about 15 graduates from previous cohorts and two mentors who contributed to the last course all working on-site. They will all be helping with the course in some capacity and you will probably get to know all of them.

The afternoon seminar

We intend to have a seminar at 2pm every day.

On Mondays, we will get the ball rolling. From Tuesdays, the role groups take over.

We will decide the topic of each seminar for the week on Mondays, so you all have time to prepare.

Code reviews

Work in pairs. Treat each code review as a structured dialogue, involving a series of questions and answers. If you do not understand a question or the answer, seek clarification before moving on.

Suggested questions

1. Are you using sensible names for all your variables? Is it obvious at first glance what each of them is being used for?
2. Are you polluting the global namespace unnecessarily with variables that could be put into functions or objects?
3. Are you declaring all your variable names at the top of the block in which they are used (this is particularly important in a language like JavaScript)?
4. Is your code *DRY*: Have you factored out your code or are you repeating yourself?
5. Are you keeping structure (HMTL), styling (CSS), data and bahaviour separate?
6. Is your code nicely commented?
7. Is your code consistently indented?
8. What feedback to you get from running jshint or some other code linting tool over your code?

See Airbnb JavaScript Style Guide for more ideas.

Functions

1. When defining a new function are we using sensible names? Is it obvious at first glance what the function is for?
2. What are the inputs for the function? And are these inputs reflected in the arguments?
3. Do you expect the function to have a return value and if so what do you expect it to be?
4. Are your functions short?
5. Are your functions easy to test?
6. As you work through the problem, are you working from the outsides in or are you trying to solve the problem in a linear sequence from top to bottom?

Unit tests

1. Are you writing tests for your functions?
2. What's a sensible first test for the function? Is this the simplest possible test that you can think of?
3. As you work through the problem, are all your passing tests still passing? If not, why not?

Callbacks/Node.js

1. Are you checking for errors in callbacks?
2. Are you factoring out your nested code into separate functions? (i.e. trying to avoid callback hell)

Sprint reviews

We are aiming to split the week up into three parts:

Mondays are sprint planning days, when you will research your project, decide on your objectives for the week and turn them into user stories. We will end the day with a review of your sprint goals for the week.

Tuesdays and Wednesdays are the first sprint of the week. On Wednesday, we end the day with an initial sprint review.

Thursdays and Fridays are the second and final project sprint. On Fridays, we end the day with a final sprint review. We will aim to have an invited reviewer, too.

For the review of sprint goals, we will expect to see your user stories in Pivotal Tracker.

For the sprint reviews, each group will need to demonstrate:

- A functioning website
- Running tests
- Linted code
- The idioms, functions, libraries and tools you have used

Presentations will be followed by Q&A.

The groups

This course is group based.

You will be formed into project groups each containing four members, which you will maintain for the length of the course.

At the beginning of each week you will all be set a new open-ended project.

Each group will need to divide the work up and share it between them.

Roles

As well as working on different parts of the project together, there will be four specific roles within each group:

- Repo owner
- Tester
- DevOps engineer
- Librarian

The roles (but not the groups) will be rotated each week.

The client

Each of you will be allocated a *client* from upstairs who will act as the *product owner* for the week's project. They will respond to your user stories in Pivotal Tracker and will come and join us for the sprint reviews on Monday, Wednesday and Friday.

Pair programming

We are going to get you pair programming together, but we are still working out the best way to introduce pairing into the classroom.

Up until now, we have recommended it without mandating it or even explaining very well what we mean by it.

In the mean time, this is a good article on pair programming.

20-minute rule

Struggle is good, but not too much of it.

As a general rule, if you cannot solve a problem within 20 minutes, then stop and talk to somebody about it.

Roles

- Repo owner
- Tester
- DevOps engineer
- Librarian

Repo owner

Repo owners are expected to collaborate closely with each other and to present tutorials on all aspects of their work.

Responsibilities

- Run the scrums
- Ensure that sprint planning, review and restrospectives take place
- Maintain the group's code repository on GitHub
- Enforce a Git development strategy
- Present a tutorial on some aspect of their work
- Handover management of the repo on Friday

Tester

Testers are expected to collaborate closely with each other and to present tutorials on all aspects of their work.

Responsibilities

- Ensure that *user stories* in *Pivotal Tracker* are written in a form that can be used as acceptance tests
- Ensure that your team are all using *Pivotal Tracker* correctly
- Write tests (initially using *Mocha* and *Supertest*)
- Make sure that your team are all constantly running tests against their code
- Make sure that your team are all constantly using JSHint
- Present a tutorial on some aspect of their work
- Handover to the new tester on Friday

See the notes on *BDD* in Week 1.

DevOps engineer

The DevOps engineer is responsible for understanding the dev environment, setting up new environments and services and introducing new tools to assist the team's build and monitoring processes.

DevOps engineers are expected to collaborate closely with each other and to present tutorials on all aspects of their work.

The dev environment

Environments that DevOps engineers should get familiar with, include:

- Dev invironments on Unix, Mac and PC
- The Unix command line and essential commands
- Installing software on Unix
- The `sudo` command
- Unix file permissions (`ls -l` and `chmod`)
- Working with Unix paths (starting with `echo $PATH`)
- Using homebrew on a mac
- Using Cygwin on Windows
- Installing Unix on a PC or ChromeBook, with e.g. Crouton
- Code editors, such as Sublime Text
- Unix-based editors: Emacs and Vi

Production environments

- Application hosting, such as Heroku
- Database hosting, such as Compose.io
- General-purpose cloud-based services, such as AWS

Build tools

The DevOps engineers should concentrate on introducing one new tool at a time, they should agree with the other DevOps engineers on which tool to introduce, and they should contribute to a whole-class tutorial on their chosen tool before moving on to the next tool.

Suggested list of build tools:

- CSS preprocessors (SCSS and Sass using Jekyll)
- Shims, fallbacks, and polyfills (HTML5-boilerplate, Modernizr...)
- Package managers (Bower and npm)
- Task runners (Gulp and Grunt)
- Scaffolding (Yeoman)
- Containers (Docker, Fig)
- dotfiles
- make

Librarian

Librarians are expected to collaborate closely with each other in researching and presenting new libraries.

Each week the librarians should introduce at least one new library, implement it somewhere in their team's codebase and present it to the whole class.

Week 1

We have a lot of ground to cover this week. We will add more stuff here as we go.

Week 1

Build a blog using GitHub Pages

The first task is to build a group blog. This will be a permanent record of your work over the next seven weeks. You will use it to share with each other and to showcase Founders & Coders to the world.

In order to complete this project, you are going to first have to get familiar with a very wide range of environments, languages and tools. Your task is to identify how they all fit together, to divide the work up between you, record your efforts, and to complete the task.

This is an open-ended project on which you will be working for exactly one week. You can make the blog as elaborate as you like and use it as a playground for as many technologies as you are able to research and implement.

As a minimum requirement, your blog should have profiles of each of your group members and each profile should include links to your *linkedin*, *github* and *codewars* accounts.

What's the point?

- Get familiar with a wide range of web technologies and tools;
- Get used to working together in groups;
- Produce a lasting record of your experience at Founders & Coders.

Topics

- Internet & Web
- Development environments
- Command line
- Git & GitHub
- GitHub Pages
- Jekyll and Markdown
- HTML & CSS
- CSS Frameworks
- Social Media

Agile development

Right from the beginning, we are going to take an *agile* approach to development.

What this means in practice is:

- Story-based project planning
- Behaviour-driven development
- Paired programming
- A product owner
- Two-day sprints
- Daily scrums
- Sprint planning
- Sprint review and retrospective
- Timeboxing (see Pomodoro)
- Stop, go, continue

Reading

The agile manifesto

Behaviour-driven development

We are going to introduce testing early in this course and we are going to use an approach to testing that fits in well with the agile approach and with the story-based project management tool that we are using.

Before we write any other software, we are going to write code to test the *expected behaviour* of our software. This approach is known as ***behaviour-driven development***, or ***BDD***.

BDD

In BDD, we start with ***acceptance criteria***. These are the criteria that a non-technical user can use to judge whether an application works as intended. These criteria should map to *user stories*.

User stories

A *user story* is a description of some desired application behaviour that must include a *scenario* and an *outcome*. An example might be:

```
When a user goes to the home page, it should display the text, "Hello, Globe Town"
```

Stories should generally be of this *"When... should..."* form. (There is another way of writing these stories, which uses the structure *"Given... When... Should"*. We might run into it later.)

These *acceptance criteria*, expressed as *user stories* are then turned into ***acceptance tests***.

Acceptance tests

Acceptance tests are code that tests the outcome of a given user story. If a user story is carefully composed, then translating it into code is relatively simple.

In pseudo-code, a test might look something like this:

```
describe 'When a User goes to the home page':
    it should actually exist
    it should display the message, 'Hello, Camden Town'
```

The `describe` block defines some scenario and each of the enclosed `it` blocks describe some desired outcome.

Mocha

For testing we are going to use Mocha, a JavaScript testing library.

Install Mocha as follows:

```
npm install -g mocha
```

In a new directory, create a file `test.js` and add the following:

```
var assert = require('assert');

describe('The number one', function() {
```

```
    it('should equal 1', function() {
        assert.equal(1, 1);
    });

    it('should not equal 2', function() {
        assert.notEqual(1, 2);
    });

});
```

Then run:

```
mocha
```

And your tests should pass, like this:

```
The number one
  ✓ should equal 1
  ✓ should not equal 2

2 passing
```

Supertest

As well as *mocha*, we are going to use *Supertest*, a JavaScript library for testing HTTP servers. This will allow us to test actual web pages, making GET and POST requests and checking the response.

Install Supertest as follows:

```
npm install -g supertest
```

Using Supertest, we can easily map user stories to acceptance tests for web applications.

A test script might look like this:

```
var request = require('supertest');
request = request('http://foundersandcoders.org/');

describe('When a user goes to the home page', function() {

    it("should return status code 200 OK", function(done) {
        request.get('/')
            .expect(200, done);
    });

    it("should contain the text 'Camden Town'", function(done) {
        request.get('/')
            .expect(/Camden Town/, done);
    });

});
```

Run `mocha` and these should pass as follows:

```
When a user goes to the home page
  ✓ should return status code 200 OK (207ms)
  ✓ should return some text referencing 'Camden Town' (243ms)

2 passing (459ms)
```

(It is possible the first test will fail with `Error: expected 200 "OK", got 302 "Moved Temporarily`. If that happens, just run the test again.)

These examples are also on GitHub.

Note

Testing is a big topic and we have just scratched the surface. There are also different ways to do BDD (see Wikipedia). The approach used here is a good jumping off point, but is not close to being exhaustive.

The command line

Before we start

You may want to set up a remote development environment (particulary if you are not running OS X or Linux).

Sign up for Nitrous.IO, have a quick read through the Getting Started guide and set up a Node.js box. You can then login and open up a console window.

Or, if you prefer, just open a terminal window on your computer.

Additional learning resources

After you have worked through the quick summary below, you can take a look at The Command Line Crash Course.

And to get the commands into your head, have a go at this course on Memrise.

A quick step-by-step summary of the basics

Where am I?

```
pwd
```

"Print Working Directory"

What's in my current directory?

```
ls
```

"LiSt"

Show me hidden files, as well:

```
ls -a
```

And show me details of each file:

```
ls -l
```

Show me both details and hidden files:

```
ls -al
```

Make a new directory called *things*:

```
mkdir things
```

"MaKe DIRectory"

Change to the *things* directory:

```
cd things
```

"Change Directory"

Go up a directory level:

```
cd ..
```

Return to my home directory:

```
cd
```

Create a new file with nothing in it:

```
touch one.txt
```

`touch` changes the access and modification time of a file (i.e. it *touches* it), but this also creates a new file if none exists with the name given.

Make a copy of it:

```
cp one.txt two.txt:
```

"CoPy"

Change its name:

```
mv one.txt three.txt
```

"MoVe"

And delete it:

```
rm three.txt
```

"ReMove"

Remove all files and folders without prompting and with no chance of recovery:

```
rm -rf .
```

"ReMove Recursively and Forcefully" everything in your current working directory. Don't do this. Instead, `mkdir ~/tmp` and `mv` anything you no longer want to there.

Remove a directory:

```
rmdir things
```

"ReMove DIRectory" (you need to empty it first)

Put some text into a file:

```
echo "hello" > greet.txt
```

And display the contents:

```
cat greet.txt
```

"conCATenate and list"

Complete the name of a file without having to type it all:

```
touch afilewithalongname
cat af<tab>
```

(i.e. use the `tab` key)

Create a file within a folder within a folder:

```
mkdir one
mkdir one/two
echo "hello" > one/two/three.txt
```

Read its contents:

```
cat o<tab><tab><tab>
```

(i.e. use the `tab` key three times in succession)

Cycle through previously-entered commands:

Use the up and down arrow keys

Edit a previously-entered command:

Use the left and right arrow keys

Git and GitHub

The importance of learning version control cannot be exaggerated. These days, that generally means Git and GitHub.

Create a GitHub project: step-by-step instructions

First, create a GitHub account.

Also, if you are not using a development environment like Nitrous.IO, you will need to install git on your own machine. If you are not sure how to do this, you may want to set up a Nitrous account now. If you do use Nitrous, you will need to connect your newly-created box to your GitHub account.

We are going to start by creating a project. Once we have done this, please follow the videos and online tutorial below, in your own time.

- Go to the **new** page on GitHub.
- Select the `Initialize this repository with a README` option.
- Add `.gitignore: Node`, if you like
- Click on `Create repository`

When redirected to the new repo page, go to `HTTPS clone URL` on the right of the page. Where it says `You can clone with HTTPS, SSH, or Subversion`, select `SSH`. This allows you `push` changes to your repo without having to enter your username and password every time you do so. Then click on the `Copy to Clipboard` icon.

Next, return to the command line on your development server:

- Navigate to the directory where you want to put your repo.
- Type `git clone` and paste the URL copied from the GitHub repo page.

(The GitHub help pages are also worth a read)

If you have not set up your git identity on your dev box, you will need to do so.

```
git config --global user.name "your name"
git config --global user.email your-email-address
```

Check the status of the new repository

```
cd <name of cloned project>
git status
```

You should see:

```
# On branch master
nothing to commit, working directory clean
```

Edit the README file

Use any code editor.

Check the status of the new repository:

```
git status
```

You should see:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.md
```

Commit the changes:

```
git commit -am 'README edit'
```

`-a` automatically stages files that have been modified and deleted. `-m` adds a message.

Check the status again:

```
git status
```

Push the commited changes back to GitHub:

```
git push
```

The first time you do this on your dev box, you will be asked to enter your GitHub credentials

Create a new file:

```
touch new.txt
```

Add the new file to the repository:

```
git add new.txt
```

Check the status again, if you like:

```
git status
```

Commit the added files:

```
git commit -a -m "a new file"
```

And push them:

```
git push
```

Check the status again:

```
git status
```

Keep `git add` ing files, `git commit` ing them and `git push` ing the changes up to GitHub.

Questions

How do I add some existing files to a new repository?
Just move them in to the repository and `git add` them.

How do I add an existing git repo to a new GitHub project?
Assuming you have already created a project on GitHub, then:

```
git remote add origin <remote repository URL>
git remote -v
git push origin master
```

See Adding an existing project to GitHub using the command line.

Introductory videos

Having set up a GitHub project, now is a good time to learn more about both Git and GitHub.

- Git Basics
- GitHub & Git Foundations

Online tutorials

Start now with *Getting Started* and *Basics* in Git Immersion.

Complete the *Introduction Sequence* and *Push & Pull -- Git Remotes!* in learnGitBranching.

Both Git Immersion and LearnGitBranching cover similar ground. Try them both out. See which one you find most useful.

Acceptance Criteria

Ron Jeffries 3 Cs

Card (User Stories) – Stories are traditionally written on notecards, and these cards can be annotated with extra details. Develope user stories through coversations with the product owner.

Conversation (Developing Acceptance Criteria) – details behind the story come out through conversations with the Product Owner.

Confirmation (Testing the AC) – acceptance tests confirm the story is finished and working as intended.

Card/User Stories

User stories - description of an objective a person should be able to achieve when using your website

Format:

```
As a who, I want what, so that why
```

For example:

```
“As a Flickr member I want to be able to assign different privacy levels to my photos so I can control who I share whic
```

Converstation/AC:

Meeting with product owner Acceptance Criteria develop from QA from discussing user stories “As a Flickr member I want to be able to assign different privacy levels to my photos so I can control who I share which photos with.”

How many different privacy levels do they need? Should changes apply to collections/individual photos? Should they be sent a confirmation email? What should the default setting be?

Confirmation/Testing:

Acceptance criteria define the bounds of a user story and are used to confirm when a story is complete and working.

You demonstrate functionality by showing how each criterion raised in conversation is satisfied from the tests.

Removes ambiguity and encourages thinking from user perspective

As mentioned above, the Product Owner decides on the priority for all remaining work. This forms the “product backlog”. Items that are at the top of this queue will have a lot more detail than items towards the bottom.

There is little value adding a lot of detail to stories that are at the bottom of the backlog because a lot might change before you get around to working on them.

Acceptence Criteria is only needed for current user stories, and shouldn't be done for everything.

Acceptence Criteris is what you test.

Week 2

Week 2: Client-side developer challenge

The Challenge will be posted on gitter.

Time: one week

An experienced frontend developer could perhaps produce working code within a few hours, but you will have a week to meet--and exceed--the brief. You will be expected to add the pages for this project to your existing blog, hosted on GitHub.

Your code will be reviewed at 5pm on Thursday by an external assessor. You will then have time on Friday to respond to his feedback and produce a final version of the challenge.

What’s the point?

- Get more familiar with responsive web design;
- Get familiar with following style guidelines;
- Get familiar with the Chrome DevTools;
- Start using JavaScript;
- Get familiar with fetching data from an API;
- Get familiar with DOM manipulation.

Languages

- HTML/CSS;
- JavaScript.

Libraries

- jQuery;

Stretch Goals

Feel free to complete the following stretch goals asynchronously by dividing up work load.

- So you've managed to make a cool client-side app using the various jQuery ajax methods. Time to go a bit wild and try creating your own ajax/ajaj methods using vanilla js.

- You've created your own ajax methods using vanilla js and your feeling like a bit of a DON!! Time to get creative and add some sexy features to your client-side app.

- You must attempt to style your page using SASS rather than CSS - It is the job of the librarians to organise this.

Tools

- Chrome DevTools.

Agile Practices

Agile Manifesto

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

There are a lot of development processes that people have created to reflect the values of the agile manifesto. These include Scrum, Kanban, Crystal Clear, Extreme Programing and many more, which all include a certain selection of agile practices.

Agile Practices

Key Agile Practices.

Scrum

Headed by an appointed scrum master, scrums are daily standing meetings where the work load is decided upon and divided based on the product owners demand and priorities. Main scrum questions:

1. What have you done since yesterday?
2. What are you planning to do today?
3. What are any impediments?

Any impediments raised are resolved by the scrum master outside of the scrum, no detailed discussions in scrums.

Morning scrums must be of the highest priority and can not be delayed no matter the circumstance. Everything must be dropped and they must occur at the exact same time every day!!

Sprints

Work loads are executed in sprints (decided in a sprint planning meeting). Sprints are of a predetermined length with a clearly specified endpoint. The first sprints should always be designed to complete the most basic functioning product. and subsequent sprints should add to the functionality and always end with a working and deployable product.

End Meetings

Sprint review - Presentation of completed work to clients and stake-holders. Sprint retrospective - A stop start continue evaluation headed by the scrum master.

info sourced from: [http://www.agileproductdesign.com/]

Week 3: use Node.js to create a social media feed

Time: one week

We will be building a Node.js application to access various social media APIs to collect posts for displaying on each of your blogs.

The initial task will be to create a gallery of tweeted images with the hashtag #FoundersandCoders on your blog, using the Twitter API and an application that responds to AJAX requests by serving JSON.

If you successfully complete this task, there are a number of possible extension tasks that you can attempt:

- Create a real-time feed that automatically updates your page when new data is posted to Twitter;
- Store older tweets locally, so that when they are too old to be retrieved directly from the API, they can be retrieved locally;
- Use the Instagram API and display images from two different sources in the same page;

There are several questions that you will have to answer to successfully complete this week's task:

- What is Node.js and how does it work;
- What Node.js modules might there be to help us with this task;
- How can we access the Twitter API;
- How can we ensure that our API credentials are not shared publicly on GitHub;
- How can we extract image URLs from the Twitter feed;
- How can we repurpose the Twitter data to create our own data feed;
- How can we extract that data to display on the page;
- How can we make a nice-looking gallery page;
- How far back do tweets served by the Twitter API go;
- How can we store and extract data locally;
- How can we extract data from the Instagram API;
- How can we nicely combine data from two different sources;
- Where can we serve our application from?

The complexity of this task is greater than the client-side developer challenge we did in Week 2, but none of the pieces are necesarily harder than anything you have already tackled. The trick will be to understand how all the pieces fit together and to break the task down into manageable chunks.

Project planning is going to be essential and you may want to understand how to solve this problem on paper before you start writing any code.

Homework

```
// Write a small program which runs in node.
// This program will allow you to run the following command:

>$ node program.js hello

// and get the following result:

>$ node program.js HELLO
> H
> E
> L
> L
> O

// For this challenge you will need to install node and npm.
```

```
// HINT: http://nodejs.org/api/process.html
```

Some references: https://www.codeschool.com/courses/real-time-web-with-node-js

Here you can find a lot of material: https://github.com/FilWisher/node-books

TESTERS

- Conceptually understand unit testing
- How do unit tests fit in to the developer's toolbox/workflow?
- Communicate with repo owners about modularizing code (this is ESSENTIAL for unit testing)
- Towards the end of the week, write some unit tests using tap and tape.

REPO OWNERS

- Look at "module.exports" and "require".
- Modularizing code and separating functionality
- How should we split our code apart? How should we structure it?
- Not sharing passwords and private keys on github (and not sharing node - modules too).
- Communicate with testers about modularizing code, they will need it to test.

DEV-OPS

- Deploying to Heroku
- Environmental variables
- How to structure branches to make sure Heroku only builds working projects
- How does the twitter API work?

LIBRARIANS

- npm & npm modules
- What modules will make this projects a bit easier?
- What cool modules can you find?
- npm init and creating modules yourselves

ALL

- You will each be assigned a mentor/guide
- They will share some resources with you and give you some gentle prodding in the right direction.

Week 4: Social Media Feed v2.

Time: one week

The project this week is similar to last week.

Again, we will be using Nodejs to build an application to access the Instagram API, collect pictures and display them.

The components to this task will be similar to last week:

- the instagram API
- a server that responds to requests with JSON
- a separate front end that makes ajax requests to your server

The challenge this week will come from the tools you will use to build this.

- TravisCI will run tests and deploy your application from Github
- Heroku will host your application online
- Codeclimate will tell you about your code quality
- Make or Gulp will run tasks as you develop
- Tap/Tape will unit test your code and tell you about code coverage
- LevelDB will be a database to store your JSON responses on request.
- Precommit hooks and JSHint to improve code quality

The trick will be to understand how all the pieces fit together and to break the task down into manageable chunks. Make use of the role groups, it will be essential for dividing up the workload.

The best apps this week will not be those that look the coolest or have the flashiest features. They will be the ones with as close to 100% test coverage (with GOOD tests), as close to 4.0 on Codeclimate, and a fully updated Pivotal Tracker board.

We want to see at least 1 deployment per day. That means you have to have something, no matter how incomplete, deployed by MONDAY EVENING!

Testers:

Managing Userstories Codeclimate

Devops:

Heroku Travis

Repo:

Git Understanding which branches to deploy from and when to merge into them. Continuous integration workflows

Librarians:

LevelDB

Week 5: D3

D3 is an extraordinary library written by Mike Bostock (see some of his work for the New York Times here). You can use it for displaying all sorts of graphical data.

HOMEWORK

Please read chapters 2, 3 (only SVG part) and 5.

Optional - You can also work through Mike's tutorial, *Let's Make a Map*, starting with *Loading data*. This is just to get an idea of the potential of D3. If you like, you can use this minimal example of a map of the world drawn using D3 as a starting point.

MONDAY

Short introduction to D3 and a quick discussion of your homework reading. Then you will be working through chapters 5, 6 and 7. After reading through the chapters we will discuss the chapter as a group, randomly picking someone to explain key aspects. We will also be looking at the interactive examples (js bins) together.

At the end of the day you will be given your challenge for the rest of the week and assigned roles.

References

- D3
- The D3 gallery for ideas
- More examples of data visualisation from Mike Bostock
- Learn D3 on docdis
- A minimal example of a map of the world drawn using D3
- Let's Make a Map by Mike Bostock, uk.json file.

Tuesday: Jigsaw Classroom

The project: Build a *Stop Go Continue* dashboard.

1. A user tweets to @founderscoders a suggestion with the hashtag *#stop*, *#go* or *#continue*
2. Other users can retweet a suggestion to vote on it;
3. Users should be able to go to a dashboard and see one or more visualisations of all the votes for all the suggestions

The project is going to be broken down into four parts:

1. A node client to pull all relevent tweets and stores the relevent data from each tweet in a file as a JSON object;
2. A node server that responds to requests by reading the file and sends it as a JSON response;
3. A web page that polls the server for data at intervals and updates the page when the data changes;
4. A dashboard that shows incoming votes for different suggestions.

STEP 1

If streaming: Connect to the Twitter API using a streaming connection. Listen for events of data being emitted in the form of tweets to founders and coders that also contain one of the relevant hashtags.

If using the REST API: create a request to the Twitter API that is triggered every to retrieve relevant tweets.

STEP 2

When we query the Twitter API, it will return the tweet results as objects.

The key information we want to extract and save for later is:

text-body: string hashtag: #stop/#go/#continue num-retweets: number original tweeter: string voters : array Twitter will give us the whole tweet as one string. Therefore, in order to separate out the text body from the hashtags, we will use regex.

Number of retweets, original tweeter and voters (i.e. the retweeters) are provided in the object Twitter gives us, so are simpler to access.

A key job is to distinguish between new tweets and retweets. When there is a new tweet, we want to create a new object to store that suggestion in, whereas if there is a retweet, we want to update an existing suggestion object with an increased retweet count, and the user name of the person who retweeted will be added to the list of tweeters or 'voters' of the original tweet. Luckily, the structure of retweets is different to tweets. The property 'retweeted_status' only exists for retweets. So we can use an 'if' statement with this property and separate the two. Within retweet objects, there is a property 'retweet-count'. This is what we will use to increase tell how many votes a suggestion gets.

if (retweeted_status exists) { fetch the retweet-count }; else (retweeted_status doesn't exist){ create new object };

We can also check the ID of the tweeter to make sure the same person has not tweeted the same text in order to cheat the voting system.

STEP 3

In order to access the data properties of the JSON object we will .stringify it.

We will have to use string manipulation with regular expressions to sort out of the tweeted text e.g. remove "@founderscoders".

We will write it to the filesystem in the form of a JSON object as a series of name: value pairs. Each tweet will be written as a new file.

There will either be a timestamp on each tweet or each file will be written in an order that easily allows whoever is reading the filesystem to easily identify which tweets are most recent and which they have already received the data from. The filename should also contain the id number of the tweet so we can easily find the right file to update if we have a retweet.

```
{ created_at: 'Tue Feb 24 12:24:51 +0000 2015',
  id: 570197663351746560,
  id_str: '570197663351746561',
  text: 'RT @RorySedgwick: @founderscoders testing #STOP',
  source: '<a href="http://twitter.com" rel="nofollow">Twitter Web Client</a>',

  user:
   { id: 1613170598,
     id_str: '1613170598',
     name: 'Greg Aubert',
     screen_name: 'gregaubs',
     location: 'London',
     url: 'http://gregaubert.com/',
     description: 'New Entrepreneurs Foundation 2014   |   Travel, tourism & startups',
     protected: false,
     verified: false,
```

```
    followers_count: 80,
    friends_count: 65,
    listed_count: 9,
    favourites_count: 1,
    statuses_count: 84,
    created_at: 'Mon Jul 22 16:21:53 +0000 2013',
    utc_offset: null,
    time_zone: null,
    geo_enabled: false,
    lang: 'en-gb',
    contributors_enabled: false,
    is_translator: false,
    profile_background_color: '676B67',
    profile_background_image_url: 'http://pbs.twimg.com/profile_background_images/378800000032343247/a7cbb5fd198fc0794
    profile_background_image_url_https: 'https://pbs.twimg.com/profile_background_images/378800000032343247/a7cbb5fd19
    profile_background_tile: false,
    profile_link_color: '6595A3',
    profile_sidebar_border_color: 'FFFFFF',
    profile_sidebar_fill_color: 'DDEEF6',
    profile_text_color: '333333',
    profile_use_background_image: false,
    profile_image_url: 'http://pbs.twimg.com/profile_images/378800000171420897/763c79ed2ed13af097ca4e7bf382430e_normal
    profile_image_url_https: 'https://pbs.twimg.com/profile_images/378800000171420897/763c79ed2ed13af097ca4e7bf382430e
    default_profile: false,
    default_profile_image: false,
    following: null,
    follow_request_sent: null,
    notifications: null },
 geo: null,
 coordinates: null,
 place: null,
 contributors: null,


 retweeted_status:
  { created_at: 'Tue Feb 24 12:19:48 +0000 2015',
    id: 570196391567466500,
    id_str: '570196391567466496',
    text: '@founderscoders testing #STOP',
    source: '<a href="http://twitter.com" rel="nofollow">Twitter Web Client</a>',
    truncated: false,
    in_reply_to_status_id: null,
    in_reply_to_status_id_str: null,
    in_reply_to_user_id: 971846516,
    in_reply_to_user_id_str: '971846516',
    in_reply_to_screen_name: 'founderscoders',
    user:
     { id: 2447724711,
       id_str: '2447724711',
       name: 'Rory Sedgwick',
       screen_name: 'RorySedgwick',
       location: 'London',
       url: null,
       description: null,
       protected: false,
       verified: false,
       followers_count: 15,
       friends_count: 62,
       listed_count: 1,
       favourites_count: 1,
       statuses_count: 21,
       created_at: 'Sun Mar 30 18:23:45 +0000 2014',
       utc_offset: null,
       time_zone: null,
       geo_enabled: false,
       lang: 'en',
       contributors_enabled: false,
       is_translator: false,
       profile_background_color: '000000',
       profile_background_image_url: 'http://abs.twimg.com/images/themes/theme1/bg.png',
       profile_background_image_url_https: 'https://abs.twimg.com/images/themes/theme1/bg.png',
       profile_background_tile: false,
       profile_link_color: '4A913C',
       profile_sidebar_border_color: '000000',
       profile_sidebar_fill_color: '000000',
       profile_text_color: '000000',
       profile_use_background_image: false,
       profile_image_url: 'http://pbs.twimg.com/profile_images/511502370375618560/bVwVBWH2_normal.jpeg',
       profile_image_url_https: 'https://pbs.twimg.com/profile_images/511502370375618560/bVwVBWH2_normal.jpeg',
       default_profile: false,
       default_profile_image: false,
       following: null,
```

```
          follow_request_sent: null,
          notifications: null },
       geo: null,
       coordinates: null,
       place: null,
       contributors: null,
       retweet_count: 1,
       favorite_count: 0,
       entities:
        { hashtags: [Object],
          trends: [],
          urls: [],
          user_mentions: [Object],
          symbols: [] },
       favorited: false,
       retweeted: false,
       possibly_sensitive: false,
       filter_level: 'low',
       lang: 'en' },


    retweet_count: 0,
    favorite_count: 0,
    entities:
     { hashtags: [ [Object] ],
       trends: [],
       urls: [],
       user_mentions: [ [Object], [Object] ],
       symbols: [] },
    favorited: false,
    retweeted: false,
    possibly_sensitive: false,
    filter_level: 'low',
    lang: 'en',
    timestamp_ms: '1424780691360' }
```

Q) What is a Server?

A) A server accepts incoming responses from the client and in turn sends a request to a data source. The data source sends a response back to the server which then responds with the data requested from the client.

Q) How do you read JSON content?

A) We use a readFile function with the parameters being :

first one is the path to the file you are reading.

second is the character set you are reading.

third is a function with two parameters, the first is for an error message and the second is the actual content you are reading from the specified file.

```
var obj;

fs.readFile('./filename.json', 'utf8', function (err, data) {

    if (err) throw err;

    obj = JSON.parse(data);
});
```

Q) How do we send the data received from the client to the browser?

A) `Response.write(JSON.stringify(obj.user.username));`

We use stringify to convert the objects in our file to strings to send to the browser. In this example we have referenced our file as obj and then specified what data we want in that object.

Task: A web page that polls the server for data at intervals and updates the page when the data changes:

Assuming that we receive the data in 3 different objects, we will be focusing on two major functions:

- A function to request the data from the server. We will be using a jQuery getJSON function to do this. However, it is possible to use an open connection but we will not be doing this today due to time constrictions.
- A function to search and compare the tweets' content (text) and the number of retweets (retweet count).

We are also considering including the favourited count depending on time.

We will be presenting to group 4, three separate objects corressponding to the hashtags #stop #go #continue. Every new tweet will be saved in an object and held within the parent object with the appropirate hashtag.

Group three is awesome!

Graph:

- Bar chart - vertical
- Height & colour - proportional to number of tweets
- 3 bar charts - one for each hashtag
  - One svg for bar chart
- 3 radio buttons to toggle between charts
  - Button switches dataset and redraws chart
- Suggestions in svg text element below chart
- Hover over bar displays suggestion text
- Optional sort button to sort

Questions:

- What should it look like: see above
- What data do we need:
  - Suggestion text
  - Retweet count
  - Which hashtag
- How often are we getting updates?
  - We are assuming we will get data and populate graph in the response callback - We will define our drawing function elsewhere then pass it as a callback to the response processing function.
- How are we going to store data?
  - Each suggestion type will be stored in a separate dataset.

Stretch:

- Instead of radio buttons, use a carousel or tabs, etc
- Change how suggestion is displayed:
  - have vertical/horizontal bars
  - have a suggestion text element below the chart
  - hovering over bar will highlight the suggestion text element and vice versa
- Timer bar on top
- Pie chart
- Force diagram

Week 6: React

Weekend assignments

1. Get an overview by following the links on React at docdis.

2. Have a go at creating some screencasts. Next week, we are going to start making video tutorials. Be prepared.

The project: build a note-taking app

Build a note-taking application in React. Any user should be able to take notes and those notes should appear in a list on the page.

Here are a number of issues you might face:

- Persisting notes locally between browser sessions;
- storing notes on a remote server so that they can be shared between users;
- Editing notes.

There are a number of ways that this project could be extended, including adding access control, assigning notes to specific users and allowing commenting on notes. We probably won't have time to explore these areas in detail this week, but we may want to start thinking about them for future weeks.

Concepts group

What vocabulary do you need to understand and what questions do you need to ask in order to understand React?

Some vocabulary: *declarative programming, rendering, mutability, reusability, composition, components, templates, coupling, cohesion, separation of concerns, scope, binding, data binding, virtual DOM, HTML attributes, string concatenation, XSS injection.*

Some questions:
Why might components encourage a better separation of concerns than templates?
Why might you want to re-render rather than mutate the DOM?
How does React tackle the issue of XSS injection?
Why might you describe React as "version control for the DOM"?
What is meant by two-way data binding and why might it be problematic in the context of a web app?

UI group

Design and mock up the user interface in HTML and CSS. Break the UI into a component hierarchy and give the components sensible names. Create a JSON object of some fake data that can be used to test the application without having to read an actual API. Present the UI and the components.

If you have time, mock up interfaces for extending the application, perhaps adding user management (logging in and so forth) to allow users to comment on each others' notes.

Reference: Thinking in React (Step 1).

Library group

What are the main methods and attributes needed to build React components and how would you use them?

Some methods and attributes:

```
props
state
createClass()
render()
renderComponent()
setState()
```

Go and read the React library documentation and become familiar with as many methods and idioms as you think will be useful.

Testing group

Take a look at: http://facebook.github.io/jest/docs/tutorial-react.html and then start working on tests for the components being designed by the UI group. Be prepared to take the lead in taking a TDD approach when you return to your teams on Wednesday.

Some questions:
Why might react components be particularly testable?
Why might testing against the virtual DOM make your life easier?

Tuesday

10:00 Morning challenge
10:30 Start work in role groups to deliver a set of recommendations.
12:00 Catchups with mentors
15:00 Catchups with mentors
17:00 Screencast presentation(s) from all role groups and pull request to gitbook.

Wednesday

10:00 Morning challenge
10:30 50-min pairing session
11:30 50-min pairing session
12:30 50-min pairing session
14:00 Table-based code reviews with mentors
14:30 Continue to work on projects in teams

Concepts

Declarative Programming:

Allows us to describe what we want, and let the underlying software/computer/etc deal with how it should happen

Imperative Code:

```
var numbers = [1,2,3,4,5]
var doubled = []

for(var i = 0; i < numbers.length; i++) {
  var newNumber = numbers[i] * 2
  doubled.push(newNumber)
}
console.log(doubled) //=> [2,4,6,8,10]
```

Declarative Code:

```
var numbers = [1,2,3,4,5]

var doubled = numbers.map(function(n) {
  return n * 2
})
console.log(doubled) //=> [2,4,6,8,10]
```

Rendering:

The process of converting/translating a form of data into another form of data usually visual

Mutability:

Its changeable unlike strings, numbers which are immutable

Arrays, Objects are mutable

reusability:

How suitable it is for reuse.
Some factors that influence how reusable the code is:

- How dry the code is
- How independant the code is
- How extensible the code is
- How Modular the code is

composition:

Composition is simply when a class is composed of other classes; or to say it another way, an instance of an object has references to instances of other objects.

components:

React components are very simple. You can think of them as simple functions that take in props and state and render HTML.

templates:

A template is a generic 'structure' for a web page, data is then injected into it and functionality (javascript) added to make a final page.

"Templates separate technologies not concerns" e.g. Templates separate html from javascript but don't separate up a page with concern for different component areas (aka functional areas)

Think of Jekyll and the _layout directory. We used liquid and YaML to do this templating.

Javascript libraries that deal with setting up templates: Underscore, Mustache, Handlebars, etc

coupling:

Coupling is the act of joining two things together. In software development, coupling refers to the degree to which one software component is dependant upon another.

- tightly-coupled architecture - when each component and its associated components must be present in order for code to be executed or compiled.
- loosely-coupled architecture - when components can remain autonomous and allow middleware software to manage communication between them.
- decoupled architecture - the components can operate completely separately and independently.

cohesion:

refers to the degree to which the elements of a module belong together.
Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is.
In Objects - all of the functions and attributes will have a high level of cohesion

separation of concerns:

separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate

Concerns are the different aspects of software functionality. For instance, the "business logic" of software is a concern, and the interface through which a person uses this logic is another.

The separation of concerns is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa.

scope:

Scope is where code lives. Scope in JS is created by functions. Code can only see and manipulate other code withing the same scope or higher scope.

binding:

Binding in this sense refers to the javascript method 'bind()'. myFunc.bind(x) forces x to be treated as the 'this' value in the myFunc function. The reason for this is beyond the scope of this brief definition. An alternative usage of 'bind' follows.

data binding:

As was alluded to in the previous definition, an alternative usage of bind is in the context of data-binding. Two-way binding refers to binding changes to an object's properties to changes in the UI and vice-versa. If the data changes, the UI should

correspondingly change, and vice-versa. Consider D3's data-binding.

virtual DOM:

The virtual DOM, or vDOM as its friends call it, is React's way of getting around the age-old problem of perfomance. React generates a representation of the DOM (vDOM - how it should look), against which it compares changes in the actual DOM (how it actually looks), in order to minimise DOM manipulation.

In other words, it compares the old-version of the DOM to a new version of the DOM and calculates the minimum number of changes required to bring the old in line with the new (i.e. to render the UI).

HTML attributes:

Attributes provide additional information about HTML elements. e.g. Width, Height, Type, Href, Src, Hidden

string concatenation:

Joining strings together to form a single string

XSS injection:

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in Web applications. XSS enables attackers to inject client-side script into Web pages viewed by other users

A type of hacker attack where the hacker injects some malicious code in the form of a string which is run when the program executes some function related to the data.

UI GROUP

Our main focus for today was understanding the basic User Interface, composing various components and producing the component hierarchy.

Check out our screencast where we look at Apple Notes and describe where each component lives and what sub-component(s) they contain:

https://www.youtube.com/watch?v=iZxAMXLUjX0&feature=youtu.be

You should also check out these docs in order to get a better understanding of React and what we are talking about in our video (basically Step 1 of the docs):

http://facebook.github.io/react/docs/thinking-in-react.html

We also had a go at making a very basic Note-Taking App. It's not complete, but check out the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello world</title>
    <script src="http://fb.me/react-0.12.1.js"></script>
    <script src="http://fb.me/JSXTransformer-0.12.1.js"></script>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js"></script>

<style type="text/css">
.NoteApp{
    margin: 30px auto;
}


.SearchField{
    width: 100%;
    padding-right: 0px;

}
.textarea{
    width: 100%;
    height: 300px;
}

.NoteList{
    margin-top: 10px;
    padding-left: 0px;
    list-style-type: none;
}
</style>
</head>
<body>

<script type="text/jsx">



var NoteApp = React.createClass({
    render: function(){
        return (
            <div className="NoteApp col-md-6 col-lg-6 col-sm-8 col-xs-10">
                <SearchableTable notes={this.props.notes}/>
                <TextEditor notes={this.props.notes}/>

            </div>
        )
    }
});

var TextEditor = React.createClass({
    render: function(){
        return(
            <div className="TextEditor col-md-8 col-lg-8 col-sm-8 col-xs-12">
```

```
            <textarea className="textarea"/>
            </div>
            )
    }
});

var SearchableTable = React.createClass({
    render: function(){
        return (
            <div className="SearchableTable col-md-4 col-lg-4 col-sm-4 col-xs-4">
                <SearchField/>
                <NoteList notes={this.props.notes}/>
            </div>
            )
    }
});

var SearchField = React.createClass({
    render: function(){
        return(
            <input type="text" className="SearchField" />
            )
    }
});


var NoteList = React.createClass({
    render: function(){
        var titles = [];
        this.props.notes.forEach(function(note){
            titles.push(<IndivNote note={note} />)
        });

        return (
            <ul className='NoteList' > {titles} </ul>
            )

    }
});

var IndivNote = React.createClass({
    render: function(){
        return (
            <li>
                <p> {this.props.note.title} </p>
            </li>
        )
    }
});


var NOTES = [
     {author: 'Amil', title: 'Amils Text1', text: 'This is Amil here and this is my text'},
     {author: 'Per', title: 'Pers Text1', text: 'This is Per here and this is my text'},
     {author: 'Ron', title: 'Rons Text1', text: 'This is Ron here and this is my text'},
     {author: 'Amil', title: 'Amils Text2', text: 'This is Amil here and this is my text'},
     {author: 'Per', title: 'Pers Text2', text: 'This is Per here and this is my text'},
     {author: 'Ron', title: 'Rons Text2', text: 'This is Ron here and this is my text'}
];

React.render(<NoteApp notes={NOTES}/>, document.body);



</script>


</body>

</html>
```

You might want to copy and paste this code into your own text editor, save it and drag and drop the file into your browser, just so you can get a taste of what it looks like!

Hello World

```
<html>
    <head>
        <script src="http://fb.me/react-0.12.2.js"></script>
        <script src="http://fb.me/JSXTransformer-0.12.2.js"></script>
    </head>
    <body>
        <div id="mydiv"></div>
        <script src="app.js" type="text/jsx"></script>
    </body>
</html>
```

```
var Hello = React.createClass({ //here we are creating the virtual DOM component, the standard naming convention is to
render: function(){ //using React's render method, we are creating the contents which will LATER be rendered, here we a
    return (
        <h1>hello world</h1>
        );
   }
});
React.render( //here we are calling the render method using dot notation, it is here that the contents i.e hello world
    <Hello/>, //the 1st argument is the component we are calling, this is variable Hello which will appear in the virtu
    document.getElementById('mydiv') //the second argument places the content of the render method into our div.
    );
```

Hello Me

using props(properties):

```
var Me = React.createClass({
   getDefaultProps: function(){ //getDefaultProps is another method in the React library
      return {
            name: "Sam"
      }
   },
  render: function(){
    return (
        <h1>Hello, my name is {this.props.name}</h1> //this refers to what component you are currently in i.e. Me, and
        );
   }
});
React.render(
    <Me/>,
    document.getElementById('mydiv')
);
```

Hello You

Now let's use props to get data from an object outside of our function:

```
var dataset = {
        name1: "Dec",
        name2: "Dave",
        name3: "Anita",
        name4: "Jason"
}

var HelloYou = React.createClass({
    render: function(){
        return (
```

```
            <h1>hello, {this.props.data.name1}</h1> //data here is just a placeholder for our dataset object, we could
            );
        }
});
React.render(
    <HelloYou data={dataset}/>, //here data becomes the attribute in the virtual DOM, to which dataset is assigned.
    //Download the chrome extension and have a look at the React DOM in your browser to see this!
    document.getElementById('mydiv')
);
```

Hello ...?

```
var BoringComponent = React.createClass({
  getInitialState : function() { //getInitialState is another method from the React library - this sets the value upon
    return {
      name : "chris"
    };
  },
  handleClick : function() { //handleClick is also another method from the React library - it is the state after the cl
    this.setState({
      name : "bob"
    });
  },
  render : function() {
    return <div onClick={this.handleClick}> //onclick is a HTML event handler
      hello {this.state.name} //here chris will be replaced by bob
    </div>;
  }
});

React.render(
  <BoringComponent />,
  document.getElementById('mydiv')
);
```

Setup

Framework

In accordance with the React devs suggestion, we're going to use Jest to run our tests. However, if you want to try a different test framework, there's nothing stopping you! Don't get confused between React and Jest -they're entirely different and independent. If you like Jest, it could be your go to all purpose test suite..

Warning

Jest doesn't play well with version 0.12 of Node. If you experience errors (e.g. Segmentation Fault), you may need to install npm packages `n` or `nvm` to roll back your version of node to 0.10.x. Version 0.10.33 seems to work well.

If using an older version of node, you may get an error message requiring you to use `--harmony` command line flag when running your tests. You can get around this by installing npm `harmonize` Instructions here.

Get started with Jest

Getting Started

There are plenty of complications to come, but here are the basics of getting a Jest test suite up and running.

1. Create a `__test__` directory in your project directory.
2. Create a package.json (e.g. use `npm init`).
3. Install jest-cli and add it to your package.json:

```
npm install jest-cli --save-dev
```

4. In your package.json, set the 'test' script to 'jest':

```
{
...json
"scripts": {
"test": "jest"
}
...
}
```

5. Run your tests with `npm test`. Jest will find and run all tests in your **test** dir.

Run a basic test

1. Create a file with a function you want to test in your project directory, and make sure you export the function. e.g.

```
// sum.js
function sum(value1, value2) {
  return value1 + value2;
}
module.exports = sum;
```

1. Create a file for your tests in your `__tests__` dir.

2. Require the file you're testing. Jest different syntax for this, to the usual node 'require'. Jest mocks functions by default, but we don't want it to mock the code we're actually **testing**. So we write:

```
jest.dontMock('../fileToTest.js');
```

1. Write your tests with in the Jasmine 'describe' style (Jest is built on Jasmine). e.g.

```
// __tests__/sum-test.js

jest.dontMock('../sum');

describe('sum', function() {
 it('adds 1 + 2 to equal 3', function() {
   var sum = require('../sum');
   expect(sum(1, 2)).toBe(3);
 });
});
```

Jasmine

A typical Jasmine test takes this form:

```
describe("A test suite name", function() {

  it("A description of your expectation", function() {
    expect(true).toBe(true);
  });
});
```

1. **describe()** opens a new test block. *The first parameter is a string describing the suite of tests. It's for your benefit, it doesn't e.g. need to match any function names.* The second parameter is a callback which will hold your tests.

2. **it()** starts a new test. *The first parameter is a string describing your expectation for this test.* The second parameter is a callback which will hold **this** test (or 'expectation').

3. **expect()** takes the function call you are testing as an argument.

4. **toBe()** is a **matcher**. *It takes your expected results as an argument.

5. **Matchers**. Jasmine has a range of these. Consult the docs, but key examples are:

    - toBe()
    - toEqual()
    - **not**.anyMatcher() => negation

Setting up Jest to test React

At last!

1. You will need to require React, and specifically the React TestUtils in your test file.

```
var React = require('react/addons'),
    TestUtils = React.addons.TestUtils;
```

1. If you're using JSX, you will need to add a file with a helper function to convert it. e.g. Create a file called `preprocessor.js` containing the following code:

```
var ReactTools = require('react-tools');

module.exports = {
  process: function(src) {
    return ReactTools.transform(src);
  }
};
```

1. To use this preprocessor, you will need to add a reference to it to your package.json. This example also stops Jest mocking React, and assumes that preprocessor.js is an a `support` dir.

```
"jest": {
  "scriptPreprocessor": "<rootDir>/support/preprocessor.js",
  "unmockedModulePathPatterns": [
    "<rootDir>/node_modules/react"
  ]
}
```

Testing React!

TestUtils

React has a built in library of test methods that you can plug in to your Jasmine style describe/it statemnts. Have a look at the methods here.

The key method is 'render into document', which will create a component instance in the virtual DOM of your component class, and allow you to run tests on it in it's instantiated state

```
ReactComponent renderIntoDocument(ReactComponent instance)
```

Tutorials

Building a test suite in React

Offical Jest React tutorial

Week 7

Draft timetable

Monday

10:00 Morning challenge
10:30 This week's project
10:45 Split into role groups for the day
16:30 Role group presentations
17:30 MVP pitch

Tuesday

10:00 Morning challenge
10:30 50-min pairing session
11:30 50-min pairing session
12:30 50-min pairing session
13:30 Continue to work on projects in teams

Wednesday

10:00 Morning challenge
10:30 Continue to work on projects in teams
17:00 Project presentations

Thursday

10:00 Morning challenge
10:30 Split into role groups for half a day *
14:00 Pairing sessions

* depending on progress

Friday

10:00 Morning challenge
10:30 Continue to work on projects in teams
17:00 Project presentations

Week 7: build a blog using node.js

Clone the Starter kit.

The blog should have the following elements:

- Submission of new blog entries
- Viewing of existing entries
- Editing of existing blog entries
- Listing of all blog entries

You should build your blog step by step, as follows:

- Write acceptance tests
- Build a JSON API using RESTful routes
- Save and retrieve entries using a database
- Add HTML pages using templates

As stretch goals, you should look at adding:

- Editing and deleting of entries
- User registration
- Caching
- Logging
- Rich-text editing

Week 7 stretch goal

Add user registration and authentication to your blog using JSON web tokens.

Resources

- Learn to use JSON Web Tokens (JWT) for Authentication

Week 7: Code Reviews

This week, we are going to introduce a new style of code review.

At times during the week, starting on Wednesday, we will start reviewing your code by viewing it on GitHub and attempting to run it.

Please always have running code on your *master* branch and update the *README* to reflect any recent changes. Make sure you have clear instructions on how to run tests and run the application.

We will leave comments as open issues in your GitHub repo.

Week 7: Roles

The role groups will be expected to create a git repository with the following elements:

- working code;
- a README with instructions for how to run the code and an explanation of what it demonstrates;
- A link to a 2-minute video summarising your findings.

Testers

Starting with the user stories, write a full set of acceptance tests using *mocha* and *shot*.

API builders

Research RESTful routes and design a complete RESTful CRUD API for the application in the *handlers.js* file.

Database administrators

Decide on how to store and retrieve data, and create working database connexions in the *model.js* file.

UI designers

Research templating libraries, including *Swig* and *Jade*, and implement outline templates for viewing, editing and listing blog entries.

TESTING

We are using shot to inject requests to the handler. This means we do not have to start the server or make real HTTP requests through the network stack. Instead, we inject mock requests directly to the handler and receive mock responses that we can then test. This saves us A LOT of time (real HTTP requests are slow).

Tools

- We are writing acceptance tests using Mocha and Shot.
- We are using node's core Assert module for assertions.
- We are writing HTTP servers using node.

Commands

- Install dependencies with `npm install`
- Start the server with `make s` or `node server.js`
- Run the tests with `make t` or `npm test`

USER STORIES

- As a user I would like to see an excerpt of the 'n' most recent blog posts when I first open the page
- As a user I would like to see a side pane list of past blog posts titles
- As a user I would like to be able to view an entire blog post in a separate page when I select 'read more'
- As a user I would like to be able to select one of the older posts from the list and have an excerpt displayed on the main page
- As a user I would like to add a new post
- As a user I would like to edit a post
- As a user I would like a separate editing/new post page.
- As a user I would like to delete a post

TESTING HTML AND JSON

- We need to test that a client request(GET, POST, DELETE) is met with a successful response, however it is trickier to extract the data we want to check from html than JSON.
- Therefore, it may be easier to test that the content of a server rendered page contains what we expect it to by looking at the data in it's json form.

TESTS USING SHOT

```
var shot = require("shot");
var server = require("../handler");
var assert = require("assert");
var request;

describe("Main page (reading view)", function () {
    request = {
            method: "GET",
            url: "
    };
    it("should respond with an OK status code", function (done) {
        shot.inject(server, request, function (res) {
            assert.equal(res.statusCode, 200);
            done();
        });
    });
    it("should show a list of blog entry titles", function (done) {
```

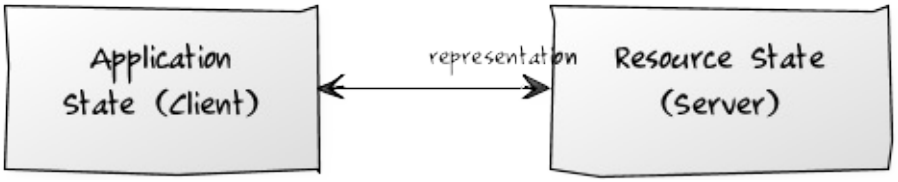
```
        shot.inject(server, request, function (res) {
            // Placeholder test - looking for evidence of 'blog post' html
            assert.equal(res.payload.match(/blog/), true);
            done();
        });
    });
    it("should show excerpts of the most recent blog posts", function (done) {
        shot.inject(server, request, function (res) {
            // Placeholder test - looking for evidence of 'blog post' html
            assert.equal(res.payload.match(/blog/), true);
            done();
        });
    });
});

describe('Edit Page', function(){
     it("should respond with an OK status code", function (done) {
        request = {
            method: "GET",
            url: "/edit"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.statusCode, 200);
            done();
        });
    });
    it("should show a text editor for writing your blog post", function (done) {
        request = {
            method: "GET",
            url: "/edit"
        };
        shot.inject(server, request, function (res) {
            // Placeholder test looking for 'input' field
            assert.equal(res.payload.match(/input/), true);
            done();
        });
    });
    it("should select a post from list in edit page for editing", function (done) {
        request = {
            method: "GET",
            url: "/edit/<blog_post_id_number>"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.payload, "");
            done();
        });
    });
    it("should add new blog post", function (done) {
        request = {
            method: "POST",
            url: "/edit"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.payload, "");
            done();
        });
    });
    it("should delete selected blog post", function (done) {
        request = {
            method: "DELETE",
            url: "/edit/<blog_post_id_number>"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.payload, "");
            done();
        });
    });
    it("should update selected blog post", function (done) {
        request = {
            method: "PUT",
            url: "/edit/<blog_post_id_number>"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.payload, "");
            done();
        });
    });
});

describe('Individual Post Page', function(){
    it("should retreive a and display single blog post", function (done) {
        request = {
```

```
            method: "GET",
            url: "/blog/<blog_post_id_number>"
        };
        shot.inject(server, request, function (res) {
            assert.equal(res.payload, "");
            done();
        });
    });
});
```

The API

What is REST?

- REST stands for **RE**presentational **S**tate **T**ransfer. This refers to transferring "representations". You are using a "representation" of a resource to transfer resource state which lives on the server into application state on the client. See image below:



- It is a software architecture style consisting of guidelines and best practices for creating scalable web services
- The use of REST is often preferred over the more heavyweight SOAP (Simple Object Access Protocol) style because REST does not leverage as much bandwidth, which makes it a better fit for use over the Internet

---

What is CRUD?

- CRUD stands for **C**reate, **R**ead, **U**pdate and **D**elete
- Sometimes known as SCRUD - the S being Search
- For each letter in the CRUD acronym the HTTP Methods are as follows:
    - Create - POST (201 (Created), 'Location' header with link to /customers/{id} containing new ID.)
    - Read - GET (200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.)
    - Update - PUT (404 (Not Found), unless you want to update/replace every resource in the entire collection.)
    - Delete - Delete (404 (Not Found), unless you want to delete the whole collection—not often desirable.)

What is Routing?

- "Routing" means, we want to handle requests to different URLs differently
- Making different HTTP requests point at different parts of our code is called "routing"

The server.js and handler.js files - a basic skeleton for the RESTful CRUD API

server.js

```
var http = require("http");
var handler = require("./handler");
var port = 4000;


//*** List of Routes and Associated Handler Functions ***//
var routes = {}
routes["/"] = handler.home;
routes["/create"] = handler.create;
routes["/update"] = handler.update;
```

```
//*** Invokes the right handler or throws error ***//
var router = function(req, res){
    var url = req.url;
    console.log("request received for ", url);


    if (typeof routes[url] === 'function'){
        routes[url](req, res);
    } else {
        console.log('Error, route for ', url, 'does not exist');
        res.writeHead(404, {"Content-Type": "text/plain"});
        res.end(" ERROR!!");
    }
}


http.createServer(router).listen(port);

console.log('Server running on port', port);
```

handler.js

```
module.exports = {


    home: function handler(req, res) {
        // console.log("Request for " + pathname + " received.");

        res.writeHead(200, {"Content-Type": "text/plain"});
        res.write("Welcome");
        res.end(" to Testing");

    },

    create: function handler(req, res) {
        // console.log("Request for " + pathname + " received.");

        res.writeHead(200, {"Content-Type": "text/plain"});
        res.write("Welcome");
        res.end(" to create article");

    },

    update: function handler(req, res) {
    // console.log("Request for " + pathname + " received.");

    res.writeHead(200, {"Content-Type": "text/plain"});
    res.write("Welcome");
    res.end(" to update article");

    },

    //createArticle

    //updateArticle

    //deleteArticle

    //viewArticle


}

};
```

Helpful Links:

- The Node Beginner's Book - http://www.acfo.org/www/uploads/documents/33e2d962a6da1d1124e7c23a9fb23972.pdf

- Vanilla JS Node Routing Demo - http://www.learnallthenodes.com/episodes/3-beginning-routing-in-nodejs
- Restful API Design With Node JS Restify (more helpful next week when we introduce frameworks) - http://code.tutsplus.com/tutorials/restful-api-design-with-nodejs-restify--cms-22637

The Database

The Templates

Week 8

This is the final week of the shared syllabus. Next week we start on MVP projects.

Draft timetable

Monday

10:00 Morning challenge
10:30 This week's project
10:45 Split into role groups for the day
16:30 Role group presentations
17:30 MVP pitch for our own projects

Tuesday

10:00 Morning challenge
10:30 Team reviews
11:30 Continue to work on projects in teams
17:30 MVP pitch for our own projects

Wednesday

10:00 Morning challenge
10:30 Continue to work on projects in teams
17:00 Project presentations (optional)
17:30 MVP pitch for our own projects

Thursday

10:00 Morning challenge
10:30 Continue to work on projects in teams
17:30 MVP pitch for our own projects

Friday

10:00 Morning challenge
10:30 Continue to work on projects in teams
14:30 Project presentations
17:30 End of course drinks

Week 8: build a blog using hapi

This week we are going to re-build our blog using hapi.

Your jumping off point is Nelson's learn Hapi repo and the hapi tutorials.

We recommend that your start with the following elements:

- a JSON API
- Submission of new blog entries
- Listing of all blog entries
- Viewing of existing entries
- User registration

Your blog will have the following elements:

Routing

Using the hapi package itself (you may also want to follow Rails routing conventions).

Testing

Using lab (we actually covered a lot of this ground last week with *shot*).

Validation and data storage

Using joi. For storing and retrieving your data you can start by storing your data in a *json* file or you can jump straight into using MongoDB (with e.g. mongojs or mongoose) or LevelDB, or some other database.

User authentication

Using bell or hapi-auth-cookie.

Roles

Split up into four different roles to explore routing, testing, validation and authentication.

Stretch goals

If you have time, you can add:

- HTML pages using views and any templating engine you like.
- Editing and deleting of existing entries

And you can explore the following packages:

- boom for error handling
- good for logging
- catbox for caching
- reply.file or inert for static file handling

Routing

Testing

Validation

Authentication

DevOps

Libraries

Source Code Management (Git)

Testing