Starting a Project with a Developer's Mindset

A brief break from coding

Planning a Project

- When we start projects, we're often tempted to start coding immediately
- After all, we're developers right? We work with code.
- However, your project should start long before you write your first line of code
- We're going to talk about how you should think about, and plan your projects, before you even open your editor

Step 1: Define Your Project

- Before you start working, you need to know what you're working on
- A brief outline of what you're building is usually a good idea (This is typically provided with your assignments)
- You'll want to expand on that with a list of goals or features you'll need to make for your app to be complete
- While grading criteria is often provided, sometimes they don't cover all of the features in the description. You may need to add some of your own criteria as well.

Online Contact Book

This is the outline for an application that stores your contacts on a website. It's only meant to hold the contacts for one individual, so there is no concept of accounts or different lists. This app only manages one set of contacts.

Goals

- Provide a list of all of a users contacts that have been entered manually
- Allow the user to inspect and alter all of the fields of each contact individually:
  - Ability to add new contacts
  - Ability to alter existing contacts
  - Ability to delete unwanted contacts
- Contacts should have the following information:
  - Name
  - Phone number
  - Address
  - Email
- Provide a way for the user to search for contacts by name

Step 2: Outline Your Models

- From your goals, you'll want to define every entity you'll need for your application
- Defining your data models upfront is really important, as you don't want to have to change them along the way
- It's best to define what their databse schema will look like, and provide an example JSON object of one of them so you'll know what it should look like

Data Models

`contacts`

Column	Type
`id`	SERIAL (PRIMARY KEY)
`name`	VARCHAR (NOT NULL)
`phone`	VARCHAR
`email`	VARCHAR
`address`	VARCHAR

```
{
  "id": 12345,
  "name": "John Smith",
  "phone": "123-555-6789",
  "email": "example@email.com",
  "address": "42 Wallaby Way, Sydney AUS"
}
```

Step 3: Outline your Routes

- Now that you know the data, and have goals in mind, you'll want to outline what your site or app looks like
- You should define each page, if it takes any arguments, and what its responsibilities should be
- Everything you outlined in Step 1 should be reflected in at least one of these pages
- Each route should be a bite-sized chunk that you can work on individually
- If any one page / route seems overly complicated, see if you can split it out in to multiple

Routes

GET `/`

Argument	Description
`message`	What message to display. Optional.

- The main list page of all contacts
- Each contact will be a link to their individual contact page
- It'll have a link for going to the create a new contact page
- It'll have a search bar at the top for searching through contacts

GET `/contact/new`

No arguments

- Displays a form for creating a new contact
- Inputs for all contact fields except id, which is auto-generated
- Upon submitting, POSTs to `/contact` with data from the form

GET `/search`

Argument	Description
`name`	What name we're searching for

- This page will look like and behave like the main page, but only show searched contacts
- It will show anyone whose name includes the string in `name`
- The search form at the top will have its value pre-populated with `name`

GET `/contact/:contactid`

Argument	Description
`:contactid`	Primary key for the contact

- Individual contact's page
- Will display all data about the contact
- Contact info is displayed as editable fields that can be saved
- Has a button to delete the contact as well

POST `/contact`

Argument	Description
`id`	Primary key for the contact, if one exists. Optional, new contact is made if left out.
`name`	Name to assign to the contact. Optional, unless creating a new contact.
`phone`	Phone number to assign to the contact. Optional.
`email`	Email address to assign to the contact. Optional.
`address`	Physical address to assign to the contact. Optional.
`delete`	Whether or not to delete the contact at `id`. Optional.

- Endpoint for submitting the contact form
- Provides ability to create, alter, or delete a contact
- Upon success, redirects back to `/` with a success message saying which of the 3 happened.
- Upon error, redirects back to `/contact/:contactid` or `/contact/new` with an error message.

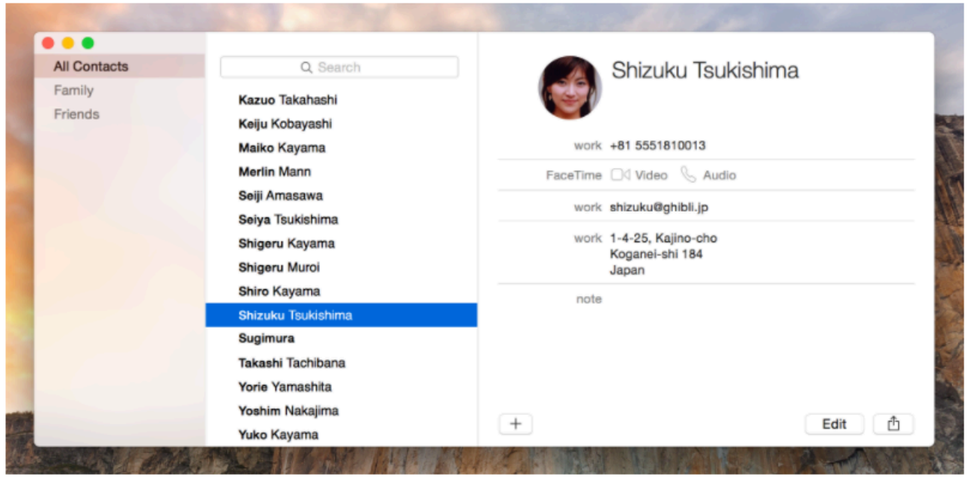
Step 4: Find Inspiration

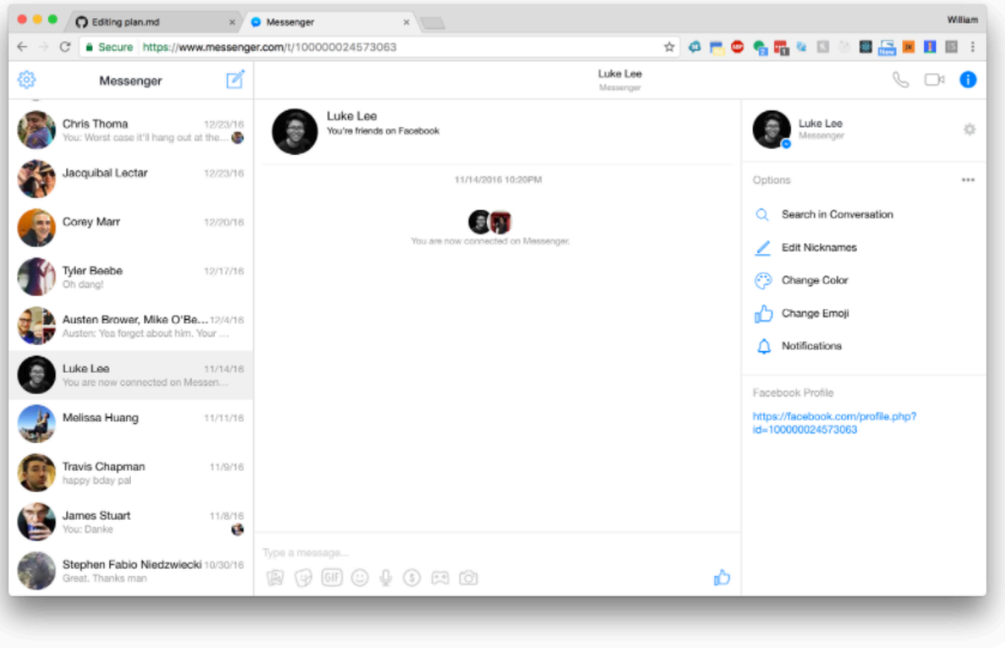
- It's very rare that any application you're working on is 100% original, so you should try to find inspiration from other apps
- That's not to say you should just *copy* other work, but you can see how other people have solved similar problems
- Make a list of things you can refer to, and why you like them
- You could also take some screenshots or other imagery that you want to base your design off of

Inspiration

I'm using the following apps for deisgn / functionality inspiration:

- Contacts+ - Exactly what I want to build. I really like their use of conact's images.
- Facebook Messenger - Even though it's a chat app, it has contacts. Great design!
- OSX Contacts - It's a desktop app, but its simplicity is fantastic.





This planning doc is available at:

https://gist.github.com/wbobeirne/65eb451f3b62c1c482fb896ca4b18400

Beginning to Code

- So now we have a kick-ass, thorough plan. We're should be ready to code, right?
- But where do we start? It may be overwhelming to try to tackle this all at once.
- We'll want to employ some techniques to make this project more manageable.

Technique 1: Bite Sized Pieces

- Even a small project like our contact app results in a ton of code and scaffolding
- If we try to write it all at once, we'll run in to dozens of errors
- And what if our plan changes? Or we find out our model needs more / differently structured data?
- Writing code in smaller pieces keeps us from feeling overwhelmed and lost, and makes changing core parts of our project (like the data model) less of a daunting task
- If a function is more than **15 lines of code**, it's probably too long and you should break it out in to more functions
- If a file is more than **80 lines of code**, it's probably too big and you should make isolated, reusable modules that it imports

Technique 2: Run Early and Often

- Writing code in smaller pieces doesn't do you much good if you're not running it frequently. You should run our project after *every single change*.
- If you run in to an error when you run, *resolve it immediately before adding more code*. Leaving that error will only make it harder to solve later.
- If you can't run your code after making a small change because it requires more changes elsewhere, *this means your code isn't bite-sized and modular enough*.
- **You shouldn't write more than 20 lines of code without running it.**

Technique 3: Write Independent Modules

- Every part of your project can be turned in to individual modules that have everything they need either passed in via functions, or imported via other modules
- This also means you should be able to test out these modules without any other setup
- Having files split out makes it easier to debug issues, and reuse that code for future projects
- It may seem like a lot of overhead to do the importing and exporting, but it will save you time in the long run

Technique 4: Fake It 'Till You Make It

- Most things in your project depend on something else
- Your views need data, your data needs a database, your database needs views to enter data
- Rather than try to build all 3 at once, simply fake pieces of it until you finish one part
  - Create json files that have fake data in them to simulate getting data from a database
  - Manually insert items into your database using your Postgres GUI to test querying against in node
  - Leave console.logs in functions you haven't finished yet, so that you can see it getting called (Especially for things like saving to the database)
- Don't be afraid to commit these faked pieces, just leave a log saying it's not real yet

Technique 5: Commit Individual Pieces

- Many assignments only have 1-3 git commits total, which is far too few
- Every time you complete a feature, you should commit it and describe the feature in your commit message
- But don't be afraid to commit things that aren't complete yet!
- Whenever you finish coding, even if it's incomplete, leave a commit describing what you're in the middle of
- That way, when you come back, you can read your last message and see what you were working on
- Given the size of your homework projects, you should easily have **6+** commits, and commits shouldn't be more than **~80 lines of code**

Technique 6: Manage Your Time

- The developer's mindset isn't one that we sit in all the time
- Don't try to take on projects 20 minutes at a time, the time it takes to get yourself setup and thinking about your current problem is pretty sizable
- Set aside at least an hour when working, so that you can really get in to the task at hand
- But on the flip, taking breaks can be extremely helpful, you may find your issues resolve much more easily if you get up and take a walk

Additional Reading

- What is Code? (2015) - A long, strange article about what it is to write code. I don't know that I've ever read anything that summed it up better.
- The Best Websites a Programmer Should Visit - A big index of a ton of helpful sites you should visit, whether you're looking to learn something new, practice something you know, or follow news and trends.