Up to Snuff

Preparing for production-scale programming

Agenda

- Goals of this training
- Attitudes for production-scale programming
- Fundamentals of languages
- Setting up the dev environment

Goals

- Understand what it takes to be a production-scale developer
- Build a full-stack to-do list app
    - You will write an app using *React Native* and a server in *Go*
    - If we have time, we will do the web app too

Production quality software

1. Great user experience (UX) - easy to learn, easy to use
    a. Minimize friction, frustration, wait-times
2. Great user interface (UI) - beautiful, good typography, good graphics
    a. Less important than UX
3. Offline first
    a. Whatever can be done offline, do it
4. User is in control
    a. always say what app is doing
    b. Don’t mysteriously do something in the background
    c. Provide a way to revert an operation (users make mistakes)

Attitudes

1. Rest well - this is a brain game, and the brain needs rest
2. Abstraction, Abstraction, Abstraction
3. Read carefully (please)
4. Write concisely
5. Be respectful
6. Be open, be honest
    - Very important to have this in a team to build a good quality product
7. See the good in your peers
8. ROOT CAUSE EVERYTHING!
    - A great developer root-causes everything!
    - Its not about the solution, its about the reason for the problem
    - Each time you root cause, you become faster and understand better

Fundamentals of Languages

- Standards
- How are languages made?
- Primitive and non-primitive types
- Memory representation
- Call-by-value, Call-by-reference

Standards - our whole world

- Javascript: https://www.ecma-international.org/publications/standards/Ecma-262.htm (12.2.5)
- Golang: https://golang.org/ref/spec
- The web: https://tools.ietf.org/html/rfc2616
  - The HTTP standard
- The Internet: https://tools.ietf.org/html/rfc793
  - The TCP/IP standard
- Wireless standards: http://www.3gpp.org/specifications
  - Phone to cell tower 2G, 3G, 4G, LTE. 5G is in progress
- Unix-like OS: http://standards.ieee.org/findstds/standard/1003.1-2017.html
  - The POSIX standard. System calls, terminal access characters, etc.

How are Languages Made?

- Write a grammar for your language (say AkshayLang)

  -
    ```
    <exp> ::= <exp> "+" <exp>
    <exp> ::= <exp> "*" <exp>
    <exp> ::= "(" <exp> ")"
    <exp> ::= "a"
    <exp> ::= "b"
    <exp> ::= "c"
    ```
- Valid program in AkshayLang: (a + b) * c
- Invalid program in AkshayLang: (x + b) * c

Generate the compiler

- Use a tool to generate a “lexer” (a.k.a tokenizer) from the grammar
  - Lex (defined in the POSIX standard): program to annotated token stream
- Use a tool to generate a “parser” (a.k.a compiler) from the lexer
  - Yacc - “Yet another compiler compiler” (also defined in the POSIX standard): token stream to Abstract Syntax Tree to machine/assembly code
- The compiler converts AkshayLang programs to assembly/machine code
- So, you now have a compiler for `AkshayLang!` You just invented a new language yay! (it really is that simple --- at a high level)
- Simply run
  - `AkshayLangCompiler firstprog.akshaylang`
  - `./firstprog.akshaylang`

Primitive and Non-primitive data types

- Primitive type: value with no associated operations (`int`, `bool`).
- Non-primitive: made up of primitive types and associated functions (`class`, `struct`)
- Variables with primitive type are used as if they contain values
- Variables with non-primitive type contain addresses

```
let myobj = {name: “Akshay”}
let addressOfObj = myobj // address of myobj (0xa42300)

let v = 5
let valOfInt = v // 5

let s = “hello”
let valOfString = s // “hello”
```

Pass by value, Pass by reference

- For any language, you can ask if the arguments to functions are passed by value or by reference.
- Pass by value: pass a copy of the value
- Pass by reference: pass a copy of the address
- Javascript: primitive types are call by value, objects/arrays are passed by reference
- Golang: everything is passed by value (full deep copy), unless you specifically pass a reference
- At the end of the day pass-by-value or reference is just a way of thinking. Reality is about memory addresses. Pass-by-value too is just an address...

How an array is stored

- Array example

```
Let myarray = [1, 2, 3, 4] // size in memory = 64 bits * 4

// at addr(myarray) is a 64 (or 32) bit number which is the
// address of the above array (MUST UNDERSTAND THIS). So the
// value of myarray is a 64 bit or 32 bit address depending
// on if you have a 32-bit or 64-bit machine

passAnArray(myarray)

Function passAnArray(a) { // a contains copy of myarray (an address)
    console.log(a[2]) // print 64bit number at a + 2 * 64 bits
}
```

How a primitive type is stored

- Number example

```
Let mynum = 6 // mynum is an address where “6” is stored
console.log(mynum) // print 64 bit number at mynum
```

- Now the compiler could already generate code where all occurrences of `mynum` are replaced with a “6”, but still, at the lowest level an address where “6” is stored is read and then printed using `console.log`

How an Array is stored in Golang

- In golang, Arrays are more like primitive types. When passed into a function, a full deep copy of the array is passed with a brand new starting address.
- However, we hardly use Arrays in Golang. Instead we use “Slices”
- A slice is a view of an array in memory. It is used exactly like an array.
  - It contains the starting address of the array in memory, and its length (and some other things).
- Copying a slice means copying a view into the same array. So altering the array using any copy of the slice, alters the same array.
- Let’s see more here: https://blog.golang.org/go-slices-usage-and-internals

Again, pass-by-value or pass-by-reference

- Just a way of thinking. What’s passed around is the address of a variable
- The type of the variable determines the size (in bytes/bits) of the value at that address and how it will be interpreted
  - Don’t forget that the type of a variable is also stored somewhere along with its address. So each variable has all this metadata that is managed transparently by the compiler, linker, and the runtime
- Its. All. About. The. Address. (of a series of bytes in memory)

A little more about memory

- Can you have a 32-bit machine with 8 GB of RAM?
- No. If memory address of each byte is stored in 32 bits than there are maximum 2^32 addresses = 4 GB addresses = 4GB of max RAM
- Which is why machines these days are mostly 64-bit to satisfy the need to have more addressable RAM

Web, REST, what?

- Web = HTTP 2.0 requests and responses with servers
- REST (representational state transfer): bad acronym, I still don't understand what it means, but, it implies a few things
  - Stateless transactions: server does not store any state, only client does (e.g. cookies)
  - CRUD (Create, Read, Update, Delete) operations on resources (files, data) only
- REST API is a set of
  - (HTTP URL + query params + headers + body + response specification)
- http://yourapp.com/login - not RESTful, URL must access resource not perform action like login. How about http://yourapp.com/credentials?

High-level HTTP request

- Base URL - http://yourapp.com
- Endpoint - http://yourapp.com/todoitem
- Query Parameter - http://yourapp.com/todoitem?due=today&complete=false
- Method: GET http://yourapp.com/todoitem?due=today&complete=false
  - A GET request cannot have a body according to the HTTP standard
- Method: POST http://yourapp.com/todoitem
  - Body (of POST message) in JSON format: {“task”: “buy milk”, “complete”: false}
  - Headers: Content-Type: application/json
- Similar specifications for each endpoint your server handles makes up the API of your server

How to write an API specification?

- This is a design of your WEB API
- A design can be written with pen and paper. Does not need any implementation or code
- The design is what a developer eventually implements (very similar to a standards document)
- Once you are familiar with writing REST APIs by hand, you can use standards-based tools (yes, standards again!). They provide formal notation to write a REST API.
  - Swagger
  - Raml
- Use Restlet Studio to make it even easier

Quick notes about Git

- Create a Git repository on GitLab
- Add an SSH key to GitLab (ssh-keygen) (~/.ssh/id_rsa.pub)
- Git clone <git://repo you just created>
- Change some files
- Git add -A :/
- Git commit -m “describe the changes you made”
- Git push

Setting up the dev environment (server)

- Install VSCode, Golang (https://golang.org/doc/install), Git
- Create an account on GitLab. Create repo in GitLab (project). Add SSH key.
- Create Heroku account. Follow Heroku's quick start guide for Go (https://devcenter.heroku.com/articles/getting-started-with-go#introduction)
    - Read up to section "Run the app locally"
    - Skip sections "Define a Procfile" all the way to and including section "Declare app dependencies"
    - After that, skip everything and go straight to section "Use a database". To connect to the DB do the following

```
db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))
    if err != nil {
        log.Fatalf("Error opening database: %q", err)
    }
```

Homework

- Write a document with your API specification (in a text file)
  - Requirements: CRUD a todo item. A todo item has a “task” and is “complete” or not.
  - **Once this document is approved,** move this API to restlet studio
- Follow this Git guide: http://rogerdudler.github.io/git-guide/
  - In your own GitLab repo add/modify a README.md file
- Learn about Go: https://tour.golang.org/list (do tour until “Methods and Interfaces”)
- Run your own Gin server in your heroku account and **send me the URL** to it
  - You will have to install gin as described here
  - This web server should have one endpoint called “/ping”
  - When I make a GET request to /ping it should send a message back
  - The sample code is already in the Gin README so all you have to do is copy and paste
  - Test using the restlet client Google chrome extension

The End