Why use Microservices? When businesses are growing, there are 2 ways, scaling up and scaling out. Scaling up means upgrading hardware. New memory, faster cpu, means it can handle more demands. At some point however, there comes a time when the cost becomes too much or you get stuck in Vendor Lock- you sign a contract to use a company’s hardware but you are stuck using only that. If it's too expensive you can scale out. Means getting more servers and breaking up the application. This is where Microservices come in. When considering scaling software it's important to consider user perspective. Ideally the user should not notice a difference between Monolithic and Micro. Mono is made like a puzzle put together. All pieces work to make one application. It won't work if a piece is missing. Micro is like Legos.  
Mono can be put on 1 server. But if there are more people, than a bigger comp is needed to make it work. Eventually there won't be a server that will hold it.   
A good example of Micro is an Amazon website. There are many little things needed to make the site run. The pictures, the multiple links, the worded descriptions. (NOTE the amazon Manifesto) If one part of the website isn't working, the web page doesn't crash, only one thing doesn't work.

**Microservices**- architectural style that structures an application as a collection of services that are: Highly maintainable, loosely coupled, independently deployable, organized around business capabilities, owned by a small team. The code inside each Micro can be whatever but the API needs to be able to understand it.

MONO vs MICRO  
Mono is built as a single unit with 3 parts: Database consisting of many tables, client-side user interface, server-side application. It runs a single logical executable. Alterations require a dev to build and deploy an updated ver of the parts. Mono are used for small jobs. Mono servers are hard to replace and losing one can be devastating (pet). Micro should act like cattle.   
Service Oriented Architecture (SOA) came out before Micro. A middle stepping point.

MICRO vs CLOUD or vs SEVERLESS  
the cloud means someone else's comp. Serverless just focuses on running programs without a specific server needed. Micro uses containers like Docker. Docker makes containers but is not the only one that can much like Google is not the only search engine. Micro is not Docker. Micro does not require containers, but they are needed to get the best of Micro. Kubernetes is a deployment tool that launces and manages containers. Cloud Native Computing Foundation (CNCF)- launched alongside Kuber. Started by Google, Twitter and many other tech companies to help build container technology.

APPLICATIONS   
**Applications**- software designed to carry out a specific task. Program(s) that are for the end user. Code that gives specific functionality. [12factor.net](https://12factor.net/)   
1. **Codebase** GitLab/GitHub/Bitbucket -any repo or any set of repos that share a root commit. One codebase tracked in revision control, many deploys. There’s always a one-to-one correlation between the codebase and the app. If there are multiple codebases it’s a distributed system. **Deploy**- running instance of the app.  
2. **Dependencies**- EXPLICITLY declare and isolate dependencies- a program may need one or more programs to run. **A twelve-factor app never relies on implicit existence of system-wide packages.** It declares dependencies using a dependency declaration manifest. It uses a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system. Both tools must always be together, one or the other is not sufficient. One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. Twelve-factor apps also do not rely on the implicit existence of any system tools.  
3. **Configuration-** store config in the environment- An app’s config is everything that is likely to vary between deploys. Config should not be part of the code. It can vary greatly each deploy but the code does not. **The twelve-factor app stores config in environment variables** (often shortened to env vars or env).

These are easy to change between deploys without changing any code.

4. **Backing** **Services-**treat backing services as attached resources- A backing service is any service the app consumes over the network as part of its normal operation. **Service**- standalone independently deployable software component which performs a specific function for a business. **The code for a twelve-factor app makes no distinction between local and third-party services.** A deploy of the 12 factor app should be able to swap a local MySQL database with one run by a third-party without any changes to the apps code.

5. **Build, Release, Run-** strictly separate build and run stages- A codebase is transformed into a (non-development) deploy through three stages:

The build stage is a transform which converts a code repo into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors dependencies and compiles binaries and assets.

The release stage takes the build produced by the build stage and combines it with the deploy’s current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.

The run stage (also known as “runtime”) runs the app in the execution environment, by launching some set of the app’s processes against a selected release. **The twelve-factor app uses strict separation between the build, release, and run stages.** Builds are initiated by the app’s devs whenever new code is deployed. Build stage can be more complex since errors are always in the foreground. In contrast runtime execution can happen automatically and should have as few moving parts as possible in case of something breaking without a dev around.

6. **Processes-** execute the app as one or more stateless processes- **Twelve-factor processes are stateless and share-nothing.** Any data that needs to persist must be stored in a stateful backing service, typically a database.

7. **Port Binding-** export services via port binding- **The twelve-factor app is completely self-contained** and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app **exports HTTP as a service by binding to a port**, and listening to requests coming in on that port.

8. **Concurrency-** doing more than one thing at a time; scale out via the process model- **In the twelve-factor app, processes are a first-class citizen.** Processes in the twelve-factor app take strong cues from the Unix process model for running service daemons. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. Any computer program, once run, is represented by one or more processes. The process model truly shines when it comes time to scale out. The share-nothing, horizontally partitionable nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation. The array of process types and number of processes of each type is known as the **process formation.**

9. **Disposability-** maximize robustness with fast startup and graceful shutdown- **The twelve-factor app’s processes are *disposable*, meaning they can be started or stopped at a moment’s notice.**  Processes should strive to minimize startup time. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs. twelve-factor app is architected to handle unexpected, non-graceful terminations.

10. **Dev/Prov Parody-** keep development, staging, and production as similar as possible- **The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.**  Historically, there have been substantial gaps between development (a developer making live edits to a local deploy of the app) and production (a running deploy of the app accessed by end users). These gaps manifest in three areas:

* **The time gap**: A developer may work on code that takes days, weeks, or even months to go into production. /Make the gap small. Write code that can be deployed hours or even minutes later.
* **The personnel gap**: Developers write code, ops engineers deploy it. /Make this gap small. Devs who wrote the code should be closely involved in the deployment and watch the production.
* **The tools gap**: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux./Keep development and production as similar as possible.

**The twelve-factor developer resists the urge to use different backing services between development and production**, even when adapters theoretically abstract away any differences in backing services. Differences between backing services mean that tiny incompatibilities crop up, causing code that worked and passed tests in development or staging to fail in production.

11. **Logs-** treat logs as even streams- **Logs** provide visibility into the behavior of a running app. Logs are the **stream**- no fixed beginning or end, but flow continuously as long as the app is operating- of aggregated, time-ordered events collected from the output streams of all running processes and backing services. **A twelve-factor app never concerns itself with routing or storage of its output stream.** During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

12. **Admin Process-** run admin/management tasks as one-off processes- Twelve-factor strongly favors languages which provide a REPL shell out of the box, and which make it easy to run one-off scripts. In a local deploy, developers invoke one-off admin processes by a direct shell command inside the app’s checkout directory. In a production deploy, developers can use ssh or other remote command execution mechanism provided by that deploy’s execution environment to run such a process. One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

Micro composed of 3 parts. Data Store, API, Software Modules. One Micro cannot communicate with another Micro unless they work with an API. Application Program Interfaces. Micro are designed around business capability. Microservices are never considered to be "done" unless the services is going to be killed. There are always things to improve or change.  
**Idempotent Operations**- no matter how many times you attempt to deploy, provision, or update. infrastructure, or what state it started at, it will always end at the same state. **Business Capability**- something business does to generate value.

DAY 2 Good job Sam on adapting to less people. Check out Hub.docker.com    
Questions: what is a heartbeat? A way to check to see if a program is still alive. What exactly is the 12 factor app process? A guideline for building the best working app. You do not have to build apps this way, but it is the best practice.              
Developing a service registry  
Curly braces/mustache brackets are representational of variables.   
Connecting Services to a Service Registry  
os.getenv gets environment variables. Variables do not move from tmux pane to a different pane. They must be exported.   
Creating First Micro lab 10  
Using Containers for Microservices  
Docker- a containerization tool; start stop and interact with containers; build container images; automatically download container images. Image is a blueprint for what should be running.   
When creating containers, you want a streamlined version. Only what is needed to make the app run. In the real world though, it usually includes much unnecessary (bloatware) data because it is faster to develop that way. **Hypervisor**- OS that acts as the main OS that let other OS operate on it.   
Integrating Microservice Lab 12  
Making More Microservices Lab 13  
Integrating More Micro Lab 14

Day 3  
HTTP Requests and Responses  
when using a GET request, we should get a response. A browser is called an 'agent' that is what makes the requests. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status This website lists information responses. Will be a 3-digit code. We want a response in the 200's.   
Procedure MITM (Man In The Middle) Proxy  
Feedback Loop: OODA Observe, Orient, Decide, Act, cont.  Created by John Boyd. Starting with observe, gather info from what is around. What can be seen, heard. Orient is sorting through the info and finding meaning. Decide is a hypothesis based off of observed info. They need tested to form more info. \* I intend to do (action) to form (result). If I move this way, the person will move another way. From here, new feedback can restart the loop. Or it can move forward to Act. Testing the decision made. Moves back to Observation to see if decision was correct or to gather new info.   
Jenkins is an automation tool. Allows build of continuous delivery pipeline, testing, and deployment of corrected code. Lab 17 Installing Jenkins  
Docker-Compose easy way to launch docker containers and specify how many instances of contains. It is an add-on to regular Docker. Works best with small apps. One server. This is also a drawback.   
YAML - Yaml Aint Markup Language  
PUT YAML INFO HERE   
Kubernetes was designed for management of containers. Can check if containers are running, healthy, the right number, if they are reachable, etc. Works on deploying and networking to automatically install updates and can scale accordingly. Typically, there are 3 master nodes and multiple worker nodes. Worker nodes live inside a pod- a group of containers. Pods are what Kuber puts onto a node. There can be up to 5,000 nodes or 150,000 pods. Pods are the smallest component in K8s (Kubernetes). Deployment: pods are rarely launched on a d cluster by themselves, they are often launched as deployment. Knows how many pods should be running at a time and updates to new versions.    
Lab 21 Deploying the Micro with Kuber  
Conway's Law- any org that designs a sys will produce a design whose structure is a copy of the org's communication structure.