

003: Basic Logging and Source Updates

I have the simplest web application I can create successfully deployed to Azure, but that's about it. There's a lot more I need to do in order to make this a production-grade deployment, so in this episode I'm going to take the first step on that journey by setting up automated deployments and logging.

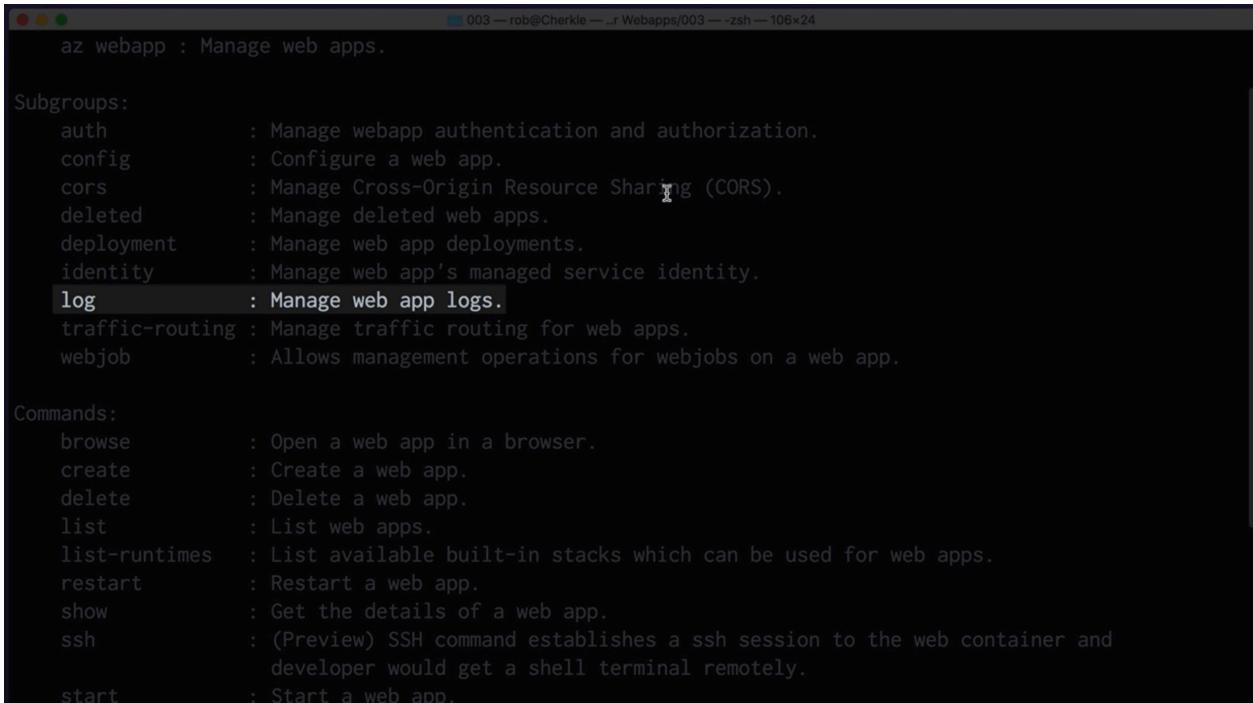
I've upgraded my single file Node web application to a bare bones website built using the Express generator - Express being a rather popular NodeJS web framework. As you can see I haven't changed the site at all - it's exactly what the generator created for me. I'll add more features later on - right now I want to be sure I understand any error that comes my way.

Setting Up Simple Logging

When it comes to logging I typically let an online service handle it for me - and there are plenty who are willing to take my cash. For instance Loggly, which I've used quite a few times, is a reliable service that I like a lot. There's also HoneyBadger and Skylight that step things up a bit and help you troubleshoot and tune your application as needed.

I'll get there eventually, but for right now I want to see what I can do with Azure. I'll be using the CLI one more time, interrogating it so I can learn how to setup what I need.

I'll start as I did before, looking at the various subcommands for `az webapp`:



```
az webapp : Manage web apps.

Subgroups:
auth      : Manage webapp authentication and authorization.
config    : Configure a web app.
cors      : Manage Cross-Origin Resource Sharing (CORS).
deleted   : Manage deleted web apps.
deployment: Manage web app deployments.
identity  : Manage web app's managed service identity.
log       : Manage web app logs. (selected)
traffic-routing: Manage traffic routing for web apps.
webjob    : Allows management operations for webjobs on a web app.

Commands:
browse    : Open a web app in a browser.
create    : Create a web app.
delete   : Delete a web app.
list     : List web apps.
list-runtimes: List available built-in stacks which can be used for web apps.
restart  : Restart a web app.
show     : Get the details of a web app.
ssh      : (Preview) SSH command establishes a ssh session to the web container and
           developer would get a shell terminal remotely.
start    : Start a web app.
```

The subcommands are listed in alphabetical order so it's pretty easy to look under "l" for logs - and there they are. Diving in again with `az webapp log -h` and I see the subcommands:

```
003 — rob@Cherkle — .r Webapps/003 — zsh — 106x24
ssh      : (Preview) SSH command establishes a ssh session to the web container and
            developer would get a shell terminal remotely.
start    : Start a web app.
stop     : Stop a web app.
up       : Create and deploy existing local code to the web app, by running the command
            from the folder where the code is present. Supports running the command in
            preview mode using --dryrun parameter. Current supports includes Node,
            Python,.NET Core, ASP.NET, staticHtml. Node, Python apps are created as Linux
            apps. .Net Core, ASP.NET and static HTML apps are created as Windows apps. If
            command is run from an empty folder, an empty windows web app is created.
update   : Update a web app.

10:28
➔ 003 az webapp log -h

Group
az webapp log : Manage web app logs.

Commands:
config  : Configure logging for a web app.
download : Download a web app's log history as a zip file.
show    : Get the details of a web app's logging configuration.
tail    : Start live log tracing for a web app.

10:28
➔ 003
```

Seems straightforward. The one I'm interested in right now is `config` but `show` and `tail` will be useful to remember later on when I want to see the log output.

Let's ask the CLI how to use `config` by executing `az webapp log config -h`:

```
003 — rob@Cherkie — ..r Webapps/003 — ~zsh — 106x24
Command
az webapp log config : Configure logging for a web app.

Arguments
--application-logging      : Configure application logging to file system. Allowed values:
                             false, true.
--detailed-error-messages  : Configure detailed error messages. Allowed values: false, true.
--docker-container-logging : Configure gathering STDOUT and STDERR output from container.
                             Allowed values: filesystem, off.
--failed-request-tracing  : Configure failed request tracing. Allowed values: false, true.
--level                     : Logging level. Allowed values: error, information, verbose,
                             warning.
--slot -s                  : The name of the slot. Default to the productions slot if not
                             specified.
--web-server-logging       : Configure Web server logging. Allowed values: filesystem, off.

Resource Id Arguments
--ids                      : One or more resource IDs (space-delimited). If provided, no other
                             'Resource Id' arguments should be specified.
--name -n                  : Name of the web app. You can configure the default using 'az
                             configure --defaults web=<name>'.
--resource-group -g        : Name of resource group. You can configure the default group using
                             'az configure --defaults group=<name>'.
--subscription             : Name or ID of subscription. You can configure the default
```

That's a lot of arguments that I can set, but the naming is pretty clear and the details help explain things in a reasonable way. Since I'm only investigating right now, I think it's reasonable to basically turn *everything* on.

Let's double check the documentation to see if there are any gotchas.

I'll do a Google search using the term `az webapp log config` which will bring up the Microsoft documentation for the command. Right at the top I see something very useful: a command reference that I can copy and paste right into a script file! Scrolling down, however, there's not much in the way of extra information here. In fact the documentation is simply echoing the argument name, which isn't helpful.

That's OK - I think I have what I need. I'll copy and paste the the command from the documentation and then set the arguments as I need.

```
scripts.sh — 003
1  #setup logging
2  # az webapp log config [--application-logging {false, true}]
3  #
4  #          [--detailed-error-messages {false, true}]
5  #
6  #          [--docker-container-logging {filesystem, off}]
7  #
8  #          [--failed-request-tracing {false, true}]
9  #
10 #
11 #          [--ids]
12 #
13 #          [--level {error, information, verbose, warning}]
14 #          [--name]
15 #
16 #          [--resource-group]
17 #
18 #          [--slot]
19 #
20 #          [--subscription]
21 #
22 #          [--web-server-logging {filesystem, off}]
```

Ln 1, Col 1 Spaces: 4 UTF-8 LF Shell Script ☺ 🔔

I've turned everything on here as I want to see application logging and I want those logs to go to the filesystem as opposed to Azure storage. I want detailed error messages and failed request tracing so I can troubleshoot things if my application becomes unresponsive.

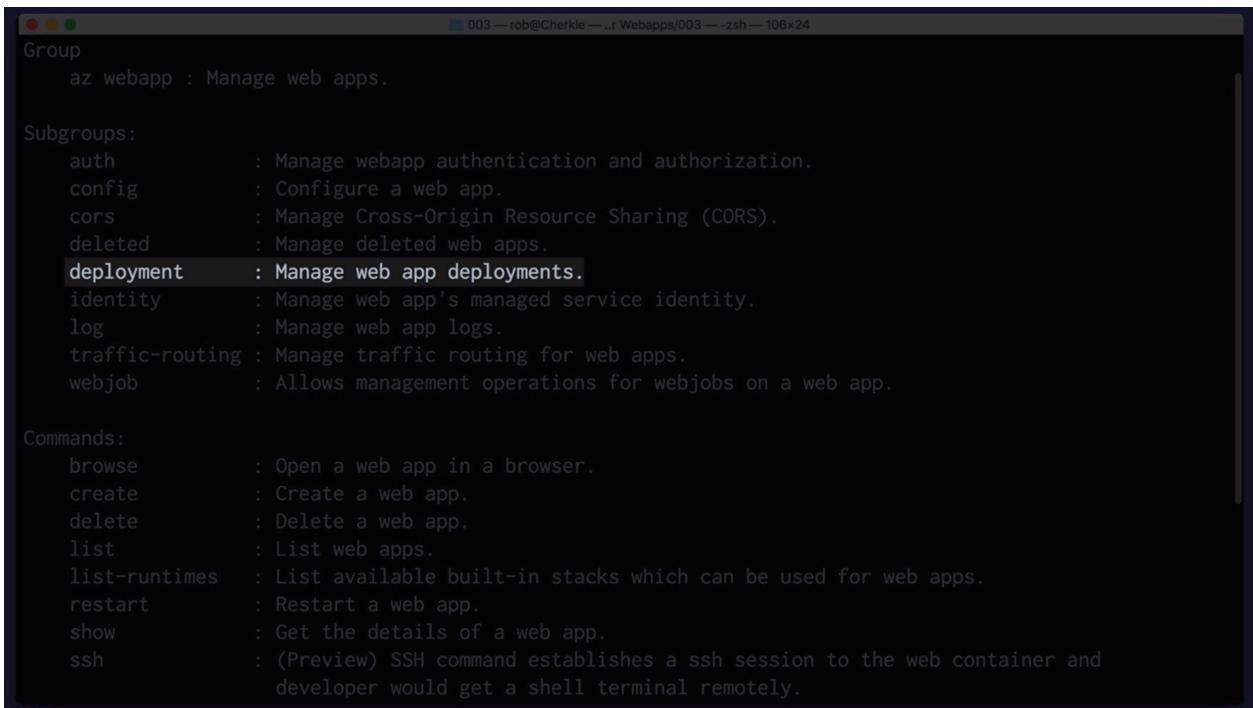
Finally, I'll be adding logging the application I created in the first episode - **velzyapp** - so I'll specify the name as you see here.

I can execute this script by loading it into my shell, which I can do using `source ./scripts.sh`. Once again Azure returns a JSON dump telling me that things were successful - my application is all set up for logging.

Continuous Container Deployment

Now I want to enable continuous deployment. Just so it's completely clear: I'm not talking about DevOps here or any kind of orchestration. What I want to enable is a simple pull of an updated image from DockerHub whenever I make changes. Let's see if Azure can handle that scenario.

Once again, I'll have a look around the `az webapp` subcommands and - there it is - `deployment`:



```
003 — rob@Cherkle — ..r Webapps/003 — ~zsh — 106x24
Group
az webapp : Manage web apps.

Subgroups:
auth      : Manage webapp authentication and authorization.
config    : Configure a web app.
cors      : Manage Cross-Origin Resource Sharing (CORS).
deleted   : Manage deleted web apps.
deployment : Manage web app deployments. (This line is highlighted)
identity  : Manage web app's managed service identity.
log       : Manage web app logs.
traffic-routing : Manage traffic routing for web apps.
webjob    : Allows management operations for webjobs on a web app.

Commands:
browse    : Open a web app in a browser.
create    : Create a web app.
delete   : Delete a web app.
list     : List web apps.
list-runtimes : List available built-in stacks which can be used for web apps.
restart   : Restart a web app.
show     : Get the details of a web app.
ssh      : (Preview) SSH command establishes a ssh session to the web container and developer would get a shell terminal remotely.
```

Let's see what kind of things this subcommand can do. I'll execute `az webapp deployment -h` and this time, thankfully, there are fewer commands to wade through:

```
→ 003 az webapp deployment -h

Group
  az webapp deployment : Manage web app deployments.

Subgroups:
  container          : Manage container-based continuous deployment.
  slot               : Manage web app deployment slots.
  source              : Manage web app deployment via source control.
  user                : Manage user credentials for deployment.

Commands:
  list-publishing-credentials : Get the details for available web app publishing credentials.
  list-publishing-profiles    : Get the details for available web app deployment profiles.
```

I have a container-based deployment so let's see what the deal is with the `container` sub-subcommand using `az webapp deployment container -h`:

```
→ 003 az webapp deployment container -h

Group
  az webapp deployment container : Manage container-based continuous deployment.

Commands:
  config      : Configure continuous deployment via containers.
  show-cd-url : Get the URL which can be used to configure webhooks for continuous deployment.
```

Paydirt! Looks like the `config` sub-sub-subcommand (😅) is exactly what we're looking for. Right below that we have a command that will give us a URL to plugin to a webhook. We'll need that - so let's remember it's here.

OK let's jump into `az webapp deployment container config -h` and see what we can learn:

```

003 — rob@Cherkie — .r Webapps/003 — zsh — 106x24
→ 003 az webapp deployment container config --help

Command
  az webapp deployment container config : Configure continuous deployment via containers.

Arguments
  --enable-cd -e [Required] : Enable/disable continuous deployment. Allowed values: false, true.
  --slot -s                : The name of the slot. Default to the production slot if not specified.

Resource Id Arguments
  --ids                   : One or more resource IDs (space-delimited). If provided, no other 'Resource Id' arguments should be specified.
  --name -n               : Name of the web app. You can configure the default using 'az configure --defaults web=<name>'.
  --resource-group -g     : Name of resource group. You can configure the default group using 'az configure --defaults group=<name>'.
  --subscription          : Name or ID of subscription. You can configure the default subscription using 'az account set -s NAME_OR_ID'.

```

To turn on continuous deployment, I simply need to set the `--enable-cd` argument to `true`. I don't know what slots are just yet so I'll ignore that for now.

As with most `webapp` commands, I also need to send in the name and resource group arguments, so I'll do that now... and I get an error because I need two dashes... fixing that and I get some JSON back:

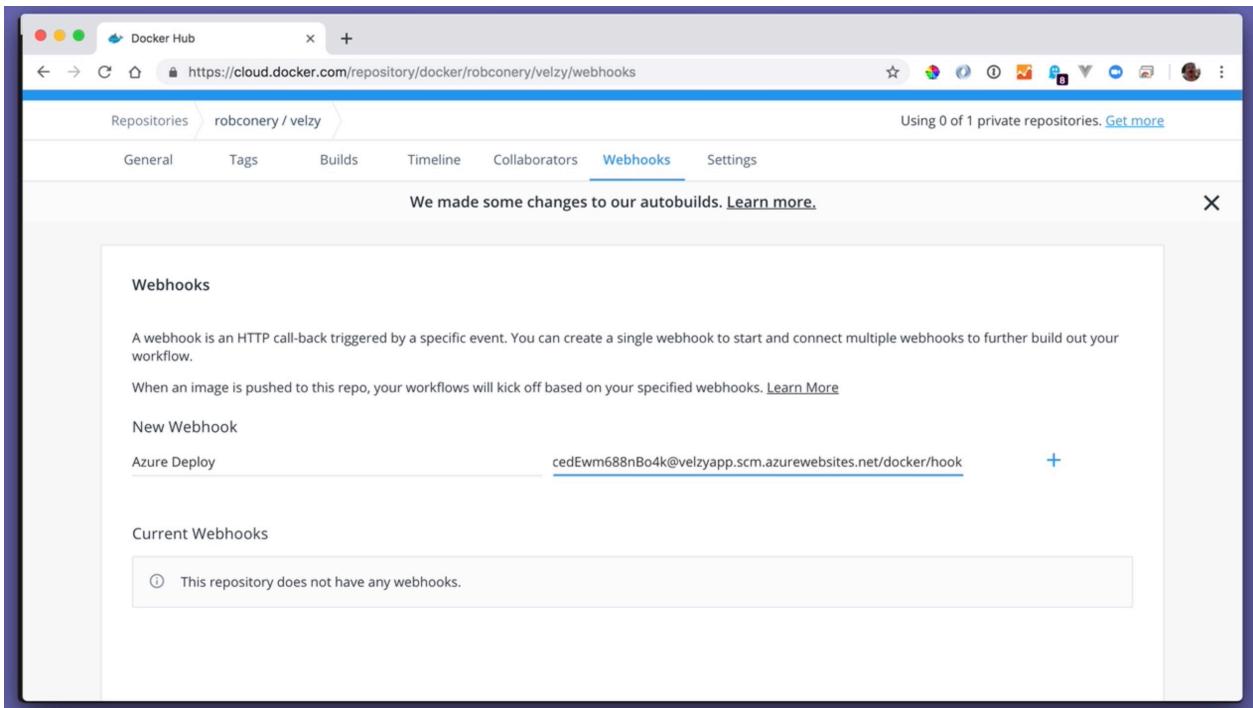
```

→ 003 az webapp deployment container config -n velzyapp -g velzy --enable-cd true
az webapp deployment container config: 'enable-cd' is not a valid value for '--enable-cd'. See 'az webapp deployment container config --help'. x 9:43
→ 003 az webapp deployment container config -n velzyapp -g velzy --enable-cd true
{
  "CI_CD_URL": "https://$velzyapp:mGxFGJoomahhpW2CiTYstfYypcaB3x89vxqDQYg9si4fDMcedEwm688nBo4k@velzyapp.scm.azurewebsites.net/docker/hook",
  "DOCKER_ENABLE_CI": true
} 9:43

```

Great! We're all set and we don't even have to ask for the web hook URL, Azure was nice enough to send it back in the response.

That's the last step - telling DockerHub to ping Azure whenever I push a new image. To do that, I need to go over to DockerHub where my image is hosted and, under "Webhooks", I'll add the URL Azure just gave me:



Great! Now we're ready to update our image.

A Better Docker Image

In the last video I used the simplest Dockerfile I could make:

```
FROM node

COPY index.js .

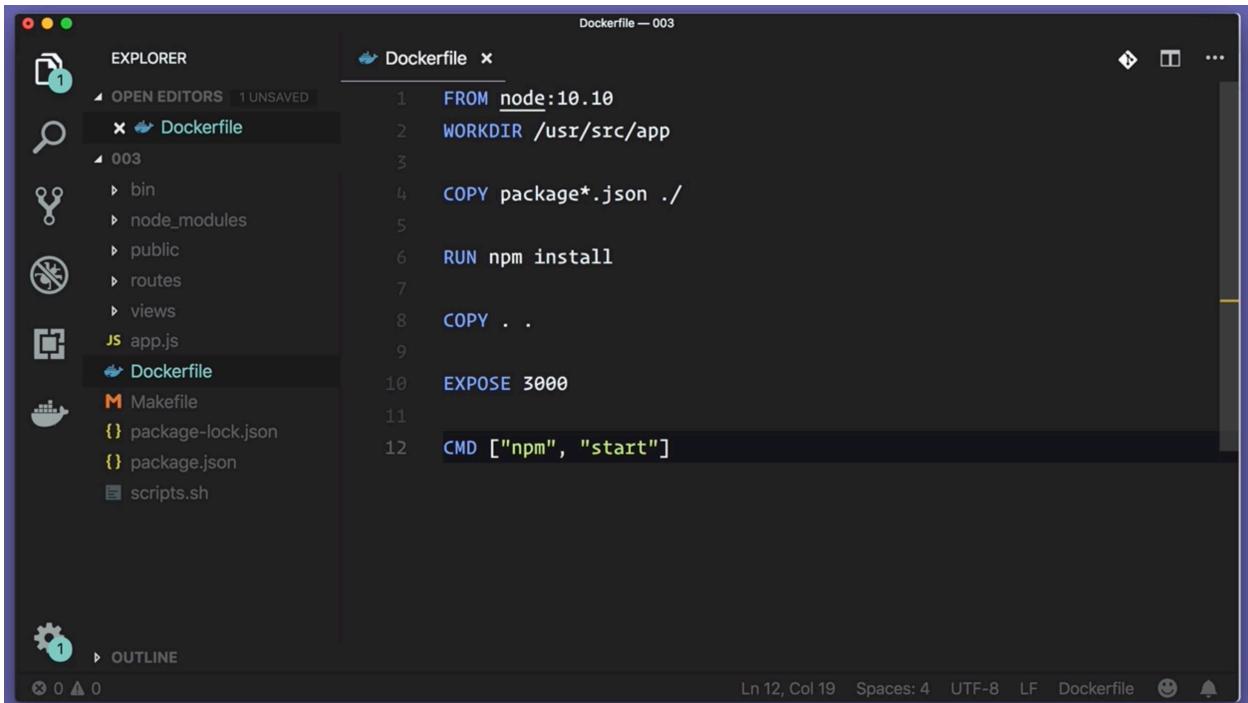
EXPOSE 3000

CMD [ "node", "index.js" ]
```

I did this, once again, because I wanted to make sure that I understood any failures as clearly as I could. Now it's time to upgrade things some.

As I mention, I'm now working with the NodeJS Express web framework and I've generated a simple website. That means I need to rebuild my Docker image so it has all the packages and application code in my project.

To do that, I'll follow the suggestions on the NodeJS web site for how to properly create a Dockerfile:



The screenshot shows the Visual Studio Code interface with a dark theme. The Explorer sidebar on the left lists files and folders: '003' (containing 'bin', 'node_modules', 'public', 'routes', 'views', 'app.js', 'Dockerfile', 'Makefile', 'package-lock.json', 'package.json', 'scripts.sh'), 'Dockerfile — 003' (selected), and 'OUTLINE'. The Dockerfile content is as follows:

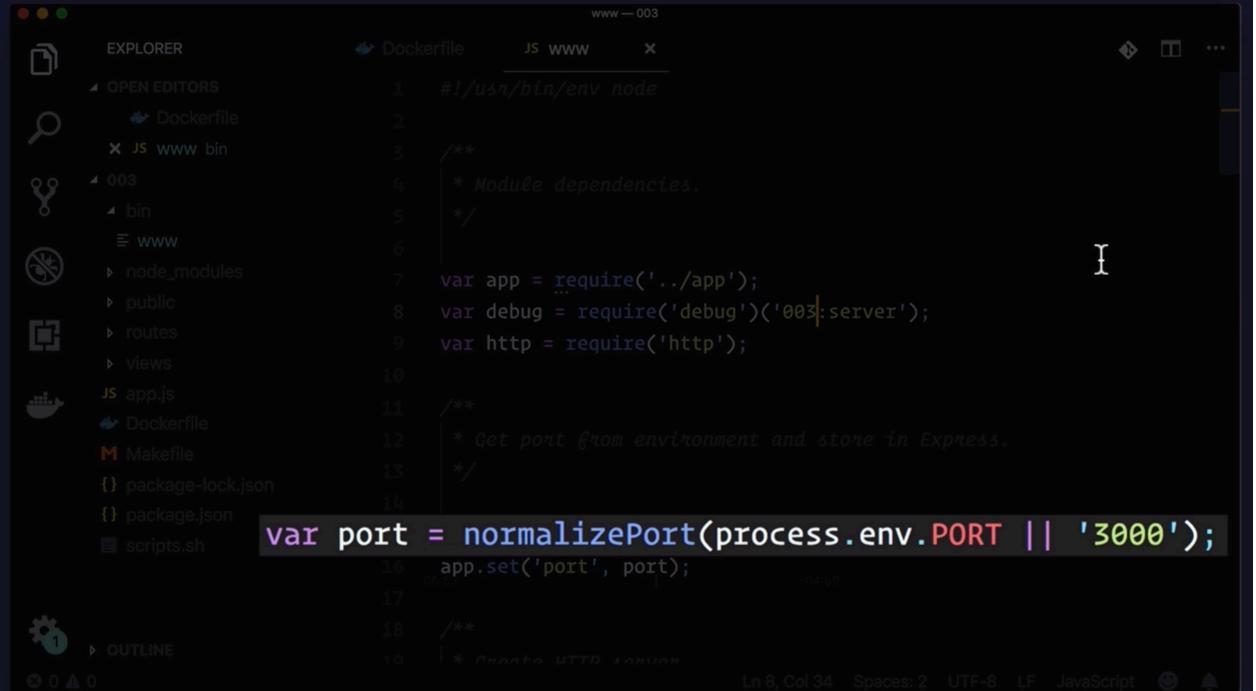
```
FROM node:10.10
WORKDIR /usr/src/app
COPY package*.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

At the bottom, status bar text includes: Ln 12, Col 19 Spaces: 4 UTF-8 LF Dockerfile

I'll set the version of Node to the latest LTS, which is 10.10. I'll set the working directory to the standard `/usr/src/app` and then I'll copy the package files in. I'll run `npm install` to load up the Express packages, copy my application files and then finally set a default command of `npm start`. This is as simple as I can make this Dockerfile, which I like.

I'm also going to learn from my past mistakes by checking the port specification. Having a look in the `/bin/www` file, I can see the port is set as it should be, using the environment variable or a hardcoded value of 3000. Azure will automatically set a

`PORT` environment variable for me, so it's a good practice to use it when starting up your web application.



The screenshot shows the Visual Studio Code interface with a Dockerfile open in the editor. The file content is as follows:

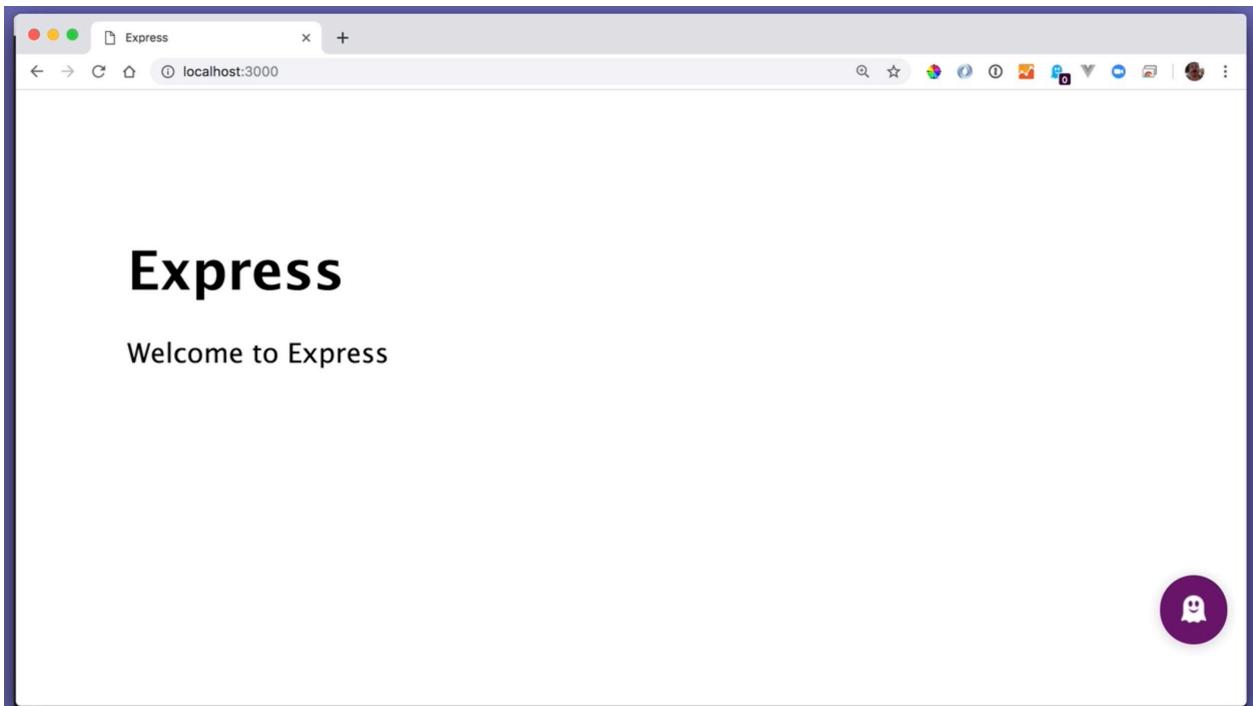
```
#!/usr/bin/env node
/*
 * Module dependencies.
 */
var app = require('../app');
var debug = require('debug')('003:server');
var http = require('http');

/*
 * Get port from environment and store in Express.
 */
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/*
 * Listen on that port.
 */
app.listen(port, function() {
  console.log(`Listening on port ${port}`);
});
```

The code highlights the line `var port = normalizePort(process.env.PORT || '3000');` in blue, indicating it is selected or being edited. The VS Code status bar at the bottom shows the file is 8 lines long, 34 columns wide, uses 2 spaces per indentation, is in UTF-8 encoding, and is a JavaScript file.

I'll rebuild the image and run it to make sure it works... which it does:



Great! Now I can push it to DockerHub and hope for the best:

A screenshot of a terminal window titled "003 — rob@Cherkle — ..r Webapps/003 — -zsh — 106x24". The command "docker push robconery/velzy" is being run, pushing multiple layers of the Docker image. The output shows layer IDs like "a1418229c88e", "96a8c673b93b", etc., followed by "Pushed". Finally, the "latest" tag is pushed with digest "sha256:7becff7c776a3e28aa2bd5aba95ab8bfce042674a38fcda818d5cebd8fd09335" and size "284 10:00".

```
→ 003 docker push robconery/velzy
The push refers to repository [docker.io/robconery/velzy]
a1418229c88e: Pushed
96a8c673b93b: Pushed
2fc050694e3a: Pushed
c28fb6fbad9: Pushed
761f74e95991: Mounted from library/node
1ddd2ca07323: Mounted from library/node
2793dc0607dd: Mounted from library/node
74800c25aa8c: Mounted from library/node
ba504a540674: Mounted from library/node
81101ce649d5: Mounted from library/node
daf45b2cad9a: Mounted from library/node
8c466bf4ca6f: Mounted from library/node
latest: digest: sha256:7becff7c776a3e28aa2bd5aba95ab8bfce042674a38fcda818d5cebd8fd09335 size: 284 10:00
→ 003
```

Viewing the Logs

I'll head over to my site on Azure - this is the one I deployed in the last video, which is the canonical "Hello World" app from the Node team. It's still up there and nothing seems to have changed.

Let's log in to Azure and see what's going on. Now that I've enabled logging (and every option too) I should be able to see what's happening with some clarity.

To remind ourselves, let's ask `az webapp log` for help by passing in the `-h` flag. The subcommands come up again and I can see some choices for viewing the logs. The first is to download them as a zip file, which I don't want. The second seems to be what I want: `show`, but the description says it's for the configuration, which I don't need. Finally, `tail` will start a live stream, which is *definitely* what I want.

Let's make sure by running `az webapp log tail -h`. I have a feeling I'll need to pass in the usual application name using `-n` and resource group using `-g`, but I'm glad I had a look first because there are some other arguments I can send in that might prove useful: `--verbose` and `--debug`. I'm not debugging anything right now, but I do want to see as much information as possible, so I'll set the `--verbose` flag.

Let's give it a whirl:

```
az webapp log tail -n velzyapp -g velzy --verbose
```

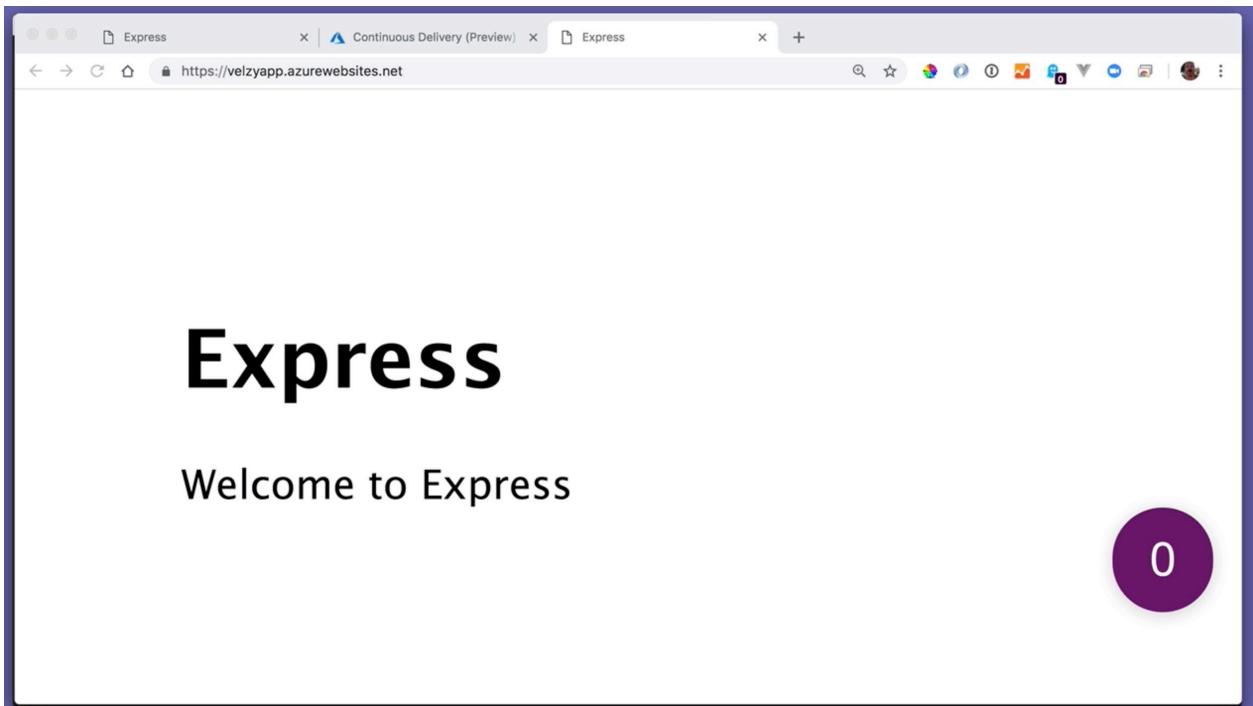
```
003 — az webapp log tail -n velzyapp -g velzy --verbose — az — Python + az webapp log tail -n velzyapp -g velzy --verbose — 106x24
f189db1b88b3: Pull complete
3d06cf2f1b5e: Pull complete
687ebdda822c: Pull complete
99119ca3f34e: Pull complete
e771d6006054: Pull complete
b0cc28d0be2c: Pull complete
1dca192b9b44: Pull complete
4b3efb8dd890: Pull complete
90d65fd82f8f: Pull complete
95d1a7c5dc59: Pull complete
dda0aa9ae411: Pull complete
1c7c0949a424: Pull complete
Digest: sha256:7beccff7c776a3e28aa2bd5aba95ab8bfce042674a38fc818d5cebd8fd09335
Status: Downloaded newer image for robconery/velzy:latest

2019-03-27 20:03:16.557 INFO  - Starting container for site
2019-03-27 20:03:16.558 INFO  - docker run -d -p 32464:3000 --name velzyapp_3 -e WEBSITES_ENABLE_APP_SERVICE_STORAGE=false -e WEBSITE_SITE_NAME=velzyapp -e WEBSITE_AUTH_ENABLED=False -e PORT=3000 -e WEBSITE_ROLE_INSTANCE_ID=0 -e WEBSITE_INSTANCE_ID=9a45092f3c96a40cd8a980e6eeadc51886c0c52bb692968eeeca35c2125e43de7 -e HTTP_LOGGING_ENABLED=1 robconery/velzy

2019-03-27 20:03:24.563 INFO  - Container velzyapp_3 for site velzyapp initialized successfully and is ready to serve requests.
```

Alright! This is exactly what I want to see. It took a few minutes for the web hook to reach Azure, and then for Azure to start the pull and reload, but it looks as though everything is ready to go.

Let's refresh the app up on Azure and see:



Victory is mine!

A Few Tips and Tricks

Before I finish up I'd like to share a few tips and tricks for the command-line-inclined out there. One of the things I don't like to do is to type commands repeatedly. To get around that, I use a `.env` file, which is a standard way of sharing environment variables for use in your shell.

I'll create one here in the root of my project using `touch`. Before I go *any* further I'll want to be sure that this file does not get added to source control, so I'll add `.env` to my `.gitignore` file:

The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor pane on the right. The Explorer sidebar shows a project structure with several folders and files, including a .gitignore file. The Editor pane displays a .gitignore file with the following content:

```
66 # dotenv environment variables file
67 .env
68 .env.test
69
70 # parcel-bundler cache (https://parceljs.org/)
71 .cache
72
73 # next.js build output
74 .next
75
76 # nuxt.js build output
77 .nuxt
78
79 # vuepress build output
80 .vuepress/dist
81
82 # Serverless directories
83 .serverless/
84
85 # FuseBox cache
86 .fusebox/
87
88 # DynamoDB Local files
```

Annotations in the form of orange arrows point from two specific lines to a yellow emoji of a person wearing sunglasses. The annotated lines are ".env" and ".env.test".

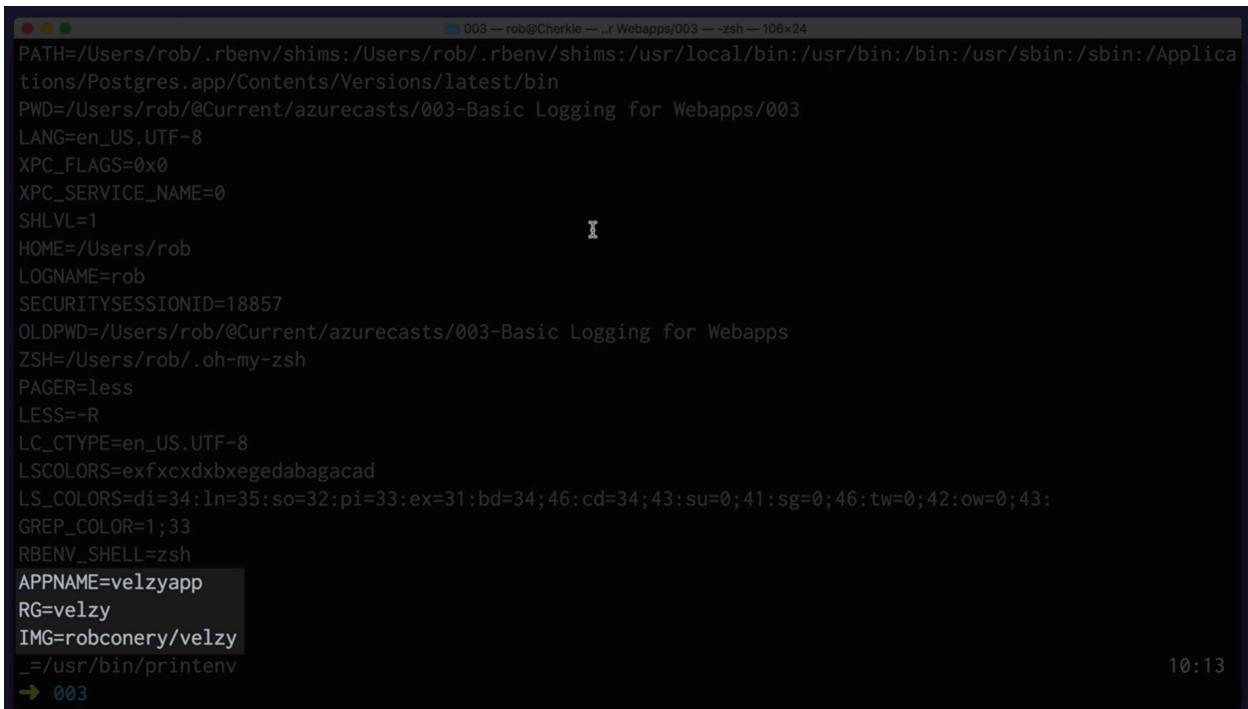
All kinds of settings and scripts can be added to a `.env` file, include API keys, database connection strings, and other sensitive environment information. Checking this into source control is a huge problem! Fortunately for me, I generated this `.gitignore` file using the `gi` command from the good people at [gitignore.io](#), who already had an entry for `.env`. Go have a look at the project if you don't know what it is.

I use zshell for my shell instead of bash and one main reason I do is because of the "Oh My Zsh" project - not quite sure how you say that. This project is great for developers because they take care of so many shortcuts and common tasks that developers use every day. They also have a rich plugin system that pumps up your shell's functionality.

One such plugin is `dotenv`, which you see here. This plugin will load any `.env` file for you when you navigate into a directory, and it will set environment variables automatically. Really useful!

Speaking of, let's finally get around to setting our environment variables. I'll set `APPNAME`, `RG` (for resource group) and `IMG` for DockerHub image. I'll also setup an alias, which is a shortcut recognized by the shell. I'll set one called `logs` and set it equal to my `az webapp log tail` command. Instead of hardcoding the name and resource group of my app, I'll use the shell variables.

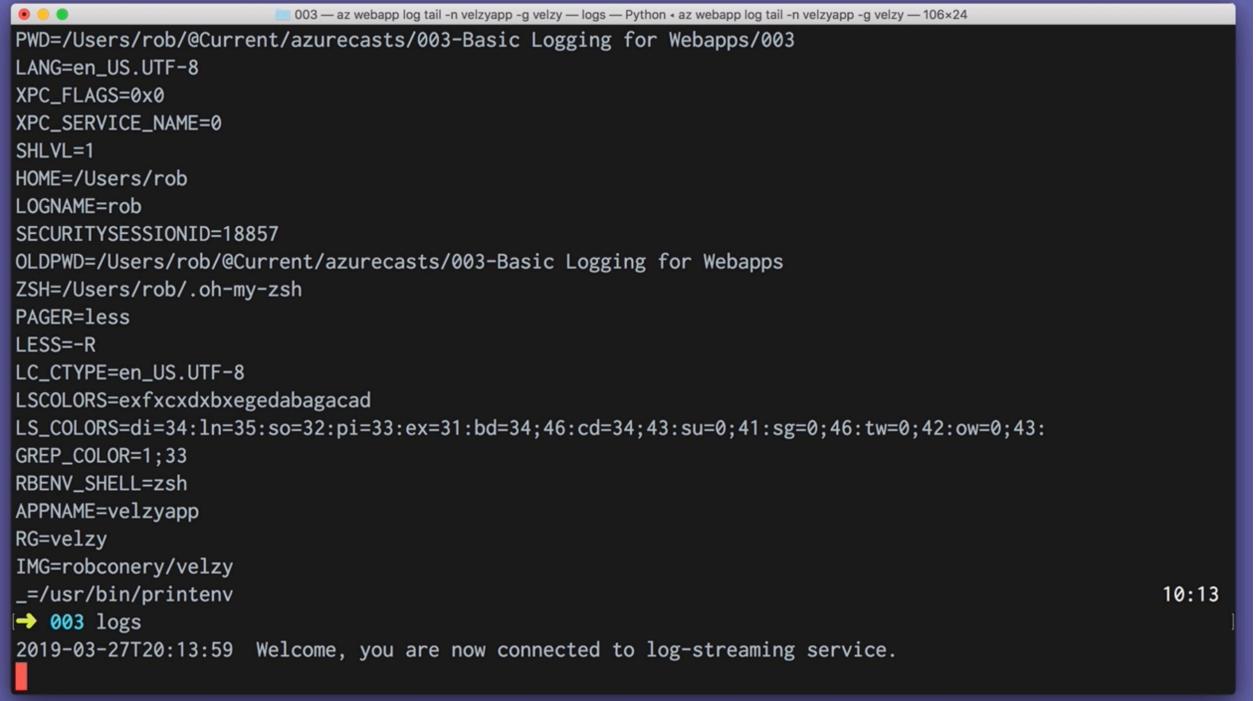
These will come in handy later on. Let's navigate out and back in and use `printenv` to see if things got loaded up - and they did:



A screenshot of a terminal window titled "003 — rob@Cherkie — ..r Webapps/003 — zsh — 106x24". The window displays a list of environment variables. Several variables are highlighted with a gray background: `APPNAME=velzyapp`, `RG=velzy`, and `IMG=robconery/velzy`. The time "10:13" is visible in the bottom right corner of the terminal window.

```
PATH=/Users/rob/.rbenv/shims:/Users/rob/.rbenv/shims:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/Postgres.app/Contents/Versions/latest/bin
PWD=/Users/rob/@Current/azurecasts/003-Basic Logging for Webapps/003
LANG=en_US.UTF-8
XPC_FLAGS=0x0
XPC_SERVICE_NAME=0
SHLVL=1
HOME=/Users/rob
LOGNAME=rob
SECURITYSESSIONID=18857
OLDPWD=/Users/rob/@Current/azurecasts/003-Basic Logging for Webapps
ZSH=/Users/rob/.oh-my-zsh
PAGER=less
LESS=-R
LC_CTYPE=en_US.UTF-8
LSCOLORS=exfxcxdxbxgedabagacad
LS_COLORS=di=34:ln=35:so=32:pi=33:ex=31:bd=34;46:cd=34;43:su=0;41:sg=0;46:tw=0;42:ow=0;43:
GREP_COLOR=1;33
RBENV_SHELL=zsh
APPNAME=velzyapp
RG=velzy
IMG=robconery/velzy
_= /usr/bin/printenv
→ 003
```

Great. Now I can run my `logs` alias, which is a lot less typing!

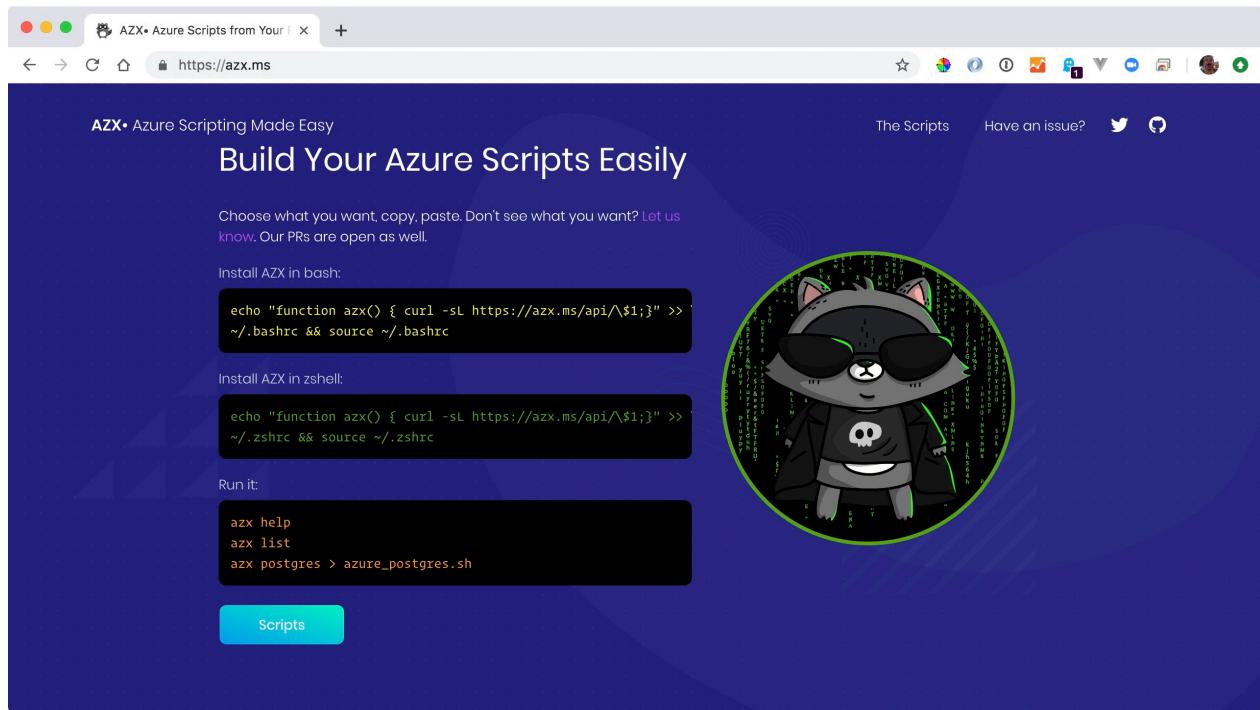


A screenshot of a terminal window titled "003 — az webapp log tail -n velzyapp -g velzy — logs — Python + az webapp log tail -n velzyapp -g velzy — 106x24". The window displays a list of environment variables and a log message. The environment variables include PWD, LANG, XPC_FLAGS, XPC_SERVICE_NAME, SHLVL, HOME, LOGNAME, SECURITYSESSIONID, OLDPWD, ZSH, PAGER, LESS, LC_CTYPE, LSCOLORS, LS_COLORS, GREP_COLOR, RBENV_SHELL, APPNAME, RG, IMG, and _=. The log message at the bottom reads: "2019-03-27T20:13:59 Welcome, you are now connected to log-streaming service." The timestamp on the right side of the window is 10:13.

```
PWD=/Users/rob/@Current/azurecasts/003-Basic Logging for Webapps/003
LANG=en_US.UTF-8
XPC_FLAGS=0x0
XPC_SERVICE_NAME=0
SHLVL=1
HOME=/Users/rob
LOGNAME=rob
SECURITYSESSIONID=18857
OLDPWD=/Users/rob/@Current/azurecasts/003-Basic Logging for Webapps
ZSH=/Users/rob/.oh-my-zsh
PAGER=less
LESS=-R
LC_CTYPE=en_US.UTF-8
LSCOLORS=exfxcxdxbxegedabagacad
LS_COLORS=di=34:ln=35:so=32:pi=33:ex=31:bd=34;46:cd=34;43:su=0;41:sg=0;46:tw=0;42:ow=0;43:
GREP_COLOR=1;33
RBENV_SHELL=zsh
APPNAME=velzyapp
RG=velzy
IMG=robconery/velzy
_=usr/bin/printenv
[→ 003 logs
2019-03-27T20:13:59 Welcome, you are now connected to log-streaming service.
```

Using a scripts file like this is really helpful. You can pop aliases in there but you can also create your own generator scripts to create Azure resources as you need. That's when these environment variables come in handy.

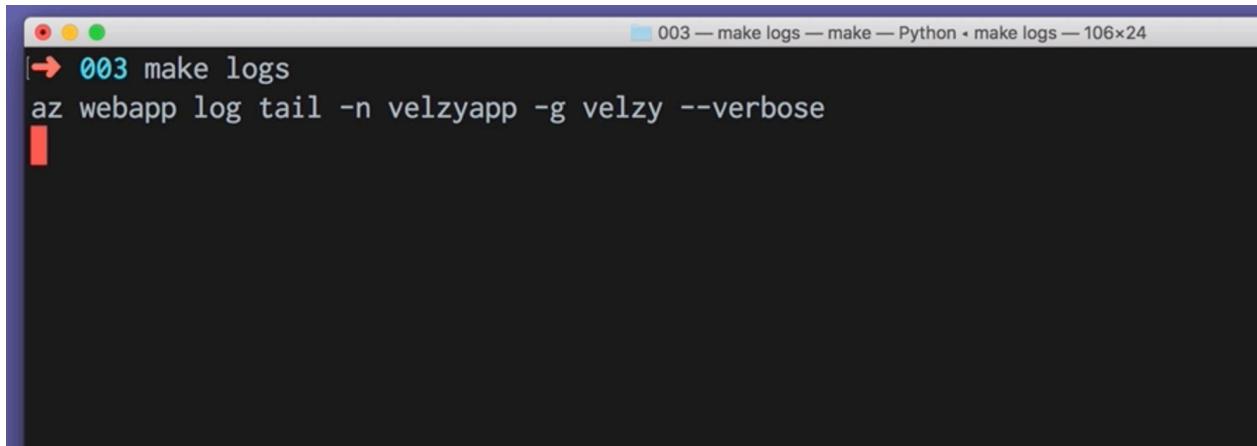
If you want to see what some of these shell scripts look like, head over to azx.ms:



This is a site I put together with my fellow CDAs and is basically a script repository for creating Azure resources. I'll talk about this site in a later episode.

One last tip before you go: let's talk about Make. If you don't know what it is, Make is a build tool that comes bundled with most Unix-based systems. It allows you to orchestrate shell commands, which is precisely what we need. I'm a huge Make fan, so let's take a moment and expand the Makefile we created in episode one.

I'll copy and paste the shell variables from script and drop them right at the top. I'll then add a `web` target (for starting my site) and a `logs` target, using the same `logs` command from my shell script. I have to use slightly different notation here for the variables, however, because each target runs in its own shell process, so I need to use this syntax so Make will expand them before executing the command.



```
003 — make logs — make — Python • make logs — 106x24
└→ 003 make logs
az webapp log tail -n velzyapp -g velzy --verbose
```

Now I can see my logs simply by running `make logs`. In a later episode I'll talk more about how we can use Make to create our resources on Azure, with a rollback feature if anything goes wrong.

That's it for now, however - thanks for watching.