**Deploy the lottery on the testnet pt.1**

**Deploying on Anvil using Makefile**

One way to deploy everything on a testnet is to use a giant command like forge script DeployRaffle --rpc-url $(SEPOLIA\_RPC\_URL) --private-key $(PRIVATE\_KEY) --broadcast --verify --etherscan-api-key $(ETHERSCAN\_API\_KEY) -vvvv. And run this over and over again. It's a legit way of deploying but it's very prone to error and most of the time you end up typing/pasting the same giant string of arguments over and over again. Not cool!

To avoid that mess we will use a Makefile. We already introduced this concept in the previous sections, but if you forgot, let me refresh your memory:

The answer for all your troubles is a Makefile!

A Makefile is a special file used in conjunction with the make command in Unix-based systems and some other environments. It provides instructions for automating the process of building software projects.

The main advantages of using a Makefile are:

* Automates tasks related to building and deploying your smart contracts.
* Integrates with Foundry commands like forge build, forge test and forge script.
* Can manage dependencies between different smart contract files.
* Streamlines the development workflow by reducing repetitive manual commands.
* Allows you to automatically grab the .env contents.

In the root folder of your project create a new file called Makefile.

After creating the file run make in your terminal.

If you have make installed then you should receive the following message:

make: \*\*\* No targets. Stop

If you don't get this message you need to install make. This is a perfect time to ask your favorite AI to help, but if you still don't manage it please come on the Updraft section of Cyfrin discord and ask the lovely people there.

Open the newly created Makefile.

Let's start with -include .env on the first line. This way we don't have to call source .env every time we want to access something from it.

Next do .PHONY: all test clean deploy fund help install snapshot format anvil. This declares that all, test, clean, deploy, fund, help, install, snapshot, format, and anvil are phony targets. This means that whenever one of these targets is specified in a make command, make will execute the associated commands without checking for a file or directory with the same name.

Next comes the help target:

help:

@echo "Usage:"

@echo " make deploy [ARGS=...]"

To test this one run make help.

The next target is build:

build:; forge build

Keep in mind that if you want to use an oneliner then we use :; after the target instead of just :. Run make build in your console to build the project.

The next target is install:

install:; forge install Cyfrin/foundry-devops@0.1.0 --no-commit && forge install smartcontractkit/chainlink@42c74fcd30969bca26a9aadc07463d1c2f473b8c --no-commit && forge install foundry-rs/forge-std@v1.7.0 --no-commit && forge install transmissions11/solmate@v6 --no-commit

In case someone ever clones your repo, they'll just run make install and every dependency will be installed for them in a super convenient way.

The next target is test:

test :; forge test

Run make test in your console to test the project.

The next target is deploy:

deploy:

@forge script script/DeployRaffle.s.sol:DeployRaffle $(NETWORK\_ARGS)

deploy:

@forge script script/DeployRaffle.s.sol:DeployRaffle $(NETWORK\_ARGS)

The @ indicates that we don't want this line printed out. We do this because if not used it will print out your private key, and we don't want that.

Every time we use an environment variable in the Makefile we use the dollar sign and parenthesis to flag it as an environment variable. The forge script above won't work as is because we don't have any NETWORK\_ARGS defined. Our main goal is to be able to call the following command make deploy ARGS="--network sepolia" and our Makefile to automatically grab the proper Sepolia deployment parameters.

Put the following above your deploy target:

NETWORK\_ARGS := --rpc-url http://localhost:8545 --private-key $(DEFAULT\_ANVIL\_KEY) --broadcast

# if --network sepolia is used, then use sepolia stuff, otherwise anvil stuff

ifeq ($(findstring --network sepolia,$(ARGS)),--network sepolia)

NETWORK\_ARGS := --rpc-url $(SEPOLIA\_RPC\_URL) --private-key $(SEPOLIA\_PRIVATE\_KEY) --broadcast --verify --etherscan-api-key $(ETHERSCAN\_API\_KEY) --legacy -vvvvv

Endif

This is how you write a basic if statement in a Makefile. Looking at the code above we can see that we never defined a DEFAULT\_ANVIL\_KEY in our .evm or inside our Makefile. Let's define it at the top of our Makefile, right after the .PHONY line.

DEFAULT\_ANVIL\_KEY := 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Create another target called anvil:

anvil :; anvil -m 'test test test test test test test test test test test junk' --steps-tracing --block-time 1

Let's test it!

Run make anvil which will create a new Anvil instance, and open a new terminal window by clicking on the + button on the right of your existing terminal window. Then run make deploy.

This should deploy our Raffle contract on the Anvil instance we just created.

Amazing work! Let's continue our Sepolia deployment in the next lesson.

**Deploying on Sepolia using Makefile**

In the previous lesson, we created a Makefile that helped us deploy our contract on Anvil. We also used an if statement to check if our deploy target is followed by Args="--network sepolia". We never tested it. Let's do that now!

Run the following command in your terminal:

make deploy ARGS="--network sepolia"

Everything should go smoothly, and your contract should be verified.

Let's interact with it using etherscan. Find your contract by searching your deployment address at <https://sepolia.etherscan.io/>. Click on the Contract button, which should have a green tick signifying that it's verified. Click on Write Contract. Click on Connect to Web3. Accept the warning, select Metamask, and select your testing account. Click on enterRaffle and put 0.01 ether there. Wait for your transaction to go through. Then, click on Read Contract and then click on getNumberOfPlayers. You should see a 1. Which means we just entered our Raffle contract. GREAT!

Let's take care of the Automation side now. Go to [automation.chain.link](https://automation.chain.link/), log in with your test account using Metamask, then click on Register new Upkeep. Chose Custom logic and paste in your Sepolia Raffle contract address. Give it a nice name like Start Draw, give it a starting balance of 2 LINK, scroll down and click on Register Upkeep, sign the transaction, wait a bit, then sign the message, then wait a bit, then click on View Upkeep.

**Reminder:** Everyone can call performUpkeep and it will work if all the conditions are met. But we don't want that to be the main way that function is called. We want to use the Chainlink Automation service to call it.

On the Start Draw automation page, we will see that Chainlink already ran the performUpKeep function. Go to <https://vrf.chain.link/> and click on your subscription to see your Pending request. After some time, you will see its status update to Success. AMAZING! Let's go back to etherscan to check our raffle contract. Go to Contract > Read Contract click on Connect to Web3 then click on getRecentWinner. You'll see that we indeed have a recent winner, which means our protocol worked flawlessly.

This time we chose to use Etherscan's interface to interact with our contract, but we could have done all this 100% using Foundry. You can use cast call to perform all the operations we did in Read Contract. We could have used cast send to perform everything we did in Write contract.

This will do for now! See you in the next lesson!

**Updates**

**Obtaining testnet funds -> Tenderly virtual testnets**

Obtaining testnet funds can be difficult as faucets often require you to hold mainnet funds or sign in with some other service like GitHub.

So, instead of deploying and interacting on a testnet like Sepolia, you can use a Tenderly virtual testnet. More information on creating your own virtual testnet can be found in [this video](https://updraft.cyfrin.io/courses/blockchain-basics/basics/sending-your-first-transaction-tenderly)!

Note that, as of today, Tenderly does not support ZKsync.