Now let’s mint our NFTs. Go back to the contract, we gonna make some updates.

Minting is a way of authenticating Data. Creating a new block and storing that Data in the blockchain. The minting method for how blocks are formed and data is added to a block is known as Proof-of-Stake.

```
// SPDX-License-Identifier: MIT
// This is the version of the Solidity compiler we want our contract to use.
// It basically says "when running this, I only want to use version 0.8.0 of the
Solidity compiler,
// nothing lower. nNote, be sure your compiler is set to 0.8.0 in hardhat.config.js.
pragma solidity ^0.8.0;

// NFT contract to inherit from.
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

// Helper functions OpenZeppelin provides.
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

import "hardhat/console.sol";

// Our contract inherits from ERC721, which is the standard NFT contract!
contract MyEpicGame is ERC721 {
// We'll hold our character's attributes in a struct. Feel free to add
// whatever you'd like as an attribute! (ex. defense, crit chance, etc).
 struct CharacterAttributes {
   uint characterIndex;
   string name;
   string imageURI;
   uint hp;
   uint maxHp;
   uint attackDamage;
 }

 // The tokenId is the NFTs unique identifier, it's just a number that goes
 // 0, 1, 2, 3, etc.
 using Counters for Counters.Counter;
 Counters.Counter private _tokenIds;

 // A lil array to help us hold the default data for our characters.
 // This will be helpful when we mint new characters and need to know
 // things like their HP, AD, etc.
 CharacterAttributes[] defaultCharacters;

 // We create a mapping from the nft's tokenId => that NFTs attributes.
 mapping(uint256 => CharacterAttributes) public nftHolderAttributes;

 // A mapping from an address => the NFTs tokenId. Gives me an ez way
 // to store the owner of the NFT and reference it later.
 mapping(address => uint256) public nftHolders;

// Data passed in to the contract when it's first created initializing the characters.
// We're going to actually pass these values in from from run.js.
 constructor(
   string[] memory characterNames,
   string[] memory characterImageURIs,
   uint[] memory characterHp,
   uint[] memory characterAttackDmg
   // Below, you can also see I added some special identifier symbols for our NFT.
   // This is the name and symbol for our token, ex Ethereum and ETH. I just call mine
   // Heroes and HERO. Remember, an NFT is just a token!
 )
   ERC721("Heroes", "HERO")
 {
   //Loop through all the characters, and save their values in our contract so
   //we can use them later when we mint our NFTs.
   for(uint i = 0; i < characterNames.length; i += 1) {
     defaultCharacters.push(CharacterAttributes({
       characterIndex: i,
       name: characterNames[i],
       imageURI: characterImageURIs[i],
       hp: characterHp[i],
       maxHp: characterHp[i],
       attackDamage: characterAttackDmg[i]
     }));

     CharacterAttributes memory c = defaultCharacters[i];

     // Hardhat's use of console.log() allows up to 4 parameters in any order of
following types: uint, string, bool, address
     console.log("Done initializing %s w/ HP %s, img %s", c.name, c.hp, c.imageURI);
   }
```

```
    // I increment tokenIds here so that my first NFT has an ID of 1.
    _tokenIds.increment();
  }

  // Users would be able to hit this function and get their NFT based on the
  // characterId they send in!
  function mintCharacterNFT(uint _characterIndex) external {
    // Get current tokenId (starts at 1 since we incremented in the constructor).
    uint256 newItemId = _tokenIds.current();

    // The magical function! Assigns the tokenId to the caller's wallet address.
    _safeMint(msg.sender, newItemId);

    // We map the tokenId => their character attributes. More on this in
    // the lesson below.
    nftHolderAttributes[newItemId] = CharacterAttributes({
      characterIndex: _characterIndex,
      name: defaultCharacters[_characterIndex].name,
      imageURI: defaultCharacters[_characterIndex].imageURI,
      hp: defaultCharacters[_characterIndex].hp,
      maxHp: defaultCharacters[_characterIndex].maxHp,
      attackDamage: defaultCharacters[_characterIndex].attackDamage
    });

    console.log("Minted NFT w/ tokenId %s and characterIndex %s", newItemId,
_characterIndex);

    // Keep an easy way to see who owns what NFT.
    nftHolders[msg.sender] = newItemId;

    // Increment the tokenId for the next person that uses it.
    _tokenIds.increment();
  }
}
```

WOAH! I know I’m shocked too lmfaoo. However again let’s take it step by step.

ERC 721:

So the ERC 721 is basically the following standard that allows for the implementation of a standard API for NFTs within smart contracts. This standard provides basic functionality to track and transfer NFTs.

The NFT standard is known as `ERC721`. OpenZeppelin essentially implements the NFT standard for us and then lets us write our own logic on top of it to customize it. That means we don't need to write boilerplate code. It's like an SDK.

It would be crazy if you write your own HTTPS server from scratch right? Same thing Here, it would be crazy if we write our NFT contract from scratch.

```
function mintCharacterNFT(uint _characterIndex)
```

This function is where the actual minting is happening.

First, you'll see that we pass in `_characterIndex`. Why?

Well, because players need to be able to tell us which character they want! For example, if I do `mintCharacterNFT(0)` then the character w/ the stats of `defaultCharacters[0]` is minted!

Okay now, wtf did I do?

1. First thing is I created two state variables which are sorta like a permanent global variable on the contract. (State variables are values permanently stored in contract storage. Local variables cannot be accessed from the outside.)

   ```
   mapping(uint256 => CharacterAttributes) public nftHolderAttributes;
   mapping(address => uint256) public nftHolders;
   ```
   nftHolderAttributes will be where we store the state of the player's NFTs. We map the NFT's id to a CharacterAttributes struct.

Remember, every player has their own character NFT. And, every single NFT has their own state like `HP`, `Attack Damage`, etc! So if Player #172 owns a "Pikachu" NFT and their Pikachu NFT loses health in a battle then only Player 172's Pikachu NFT should be changed and everyone else's Pikachu should stay the same! So, we store this player character level data in a map.

2. You will see I "Inheirt" an OpenZepplin contract using the ERC721 when I declare the contract. Inheriting is basically calling other contract's functions for me to use.
3. `_tokenIds.Ids` starts @ 0. It is just a counter `.Increment` does just that.

4. The actual minting happens at the `function mintCharacterNFT(uint _characterIndex)` First we will pass the *`CharacteratIndex`*. Because the user can have the option to choose which player he wants.
5. From there we have a number called `newItemId`. This is the id of the NFT itself. Remember, each NFT is "unique" and the way we do this is we give each token a unique ID. It's just a basic counter. `_tokenIds.current()` starts at 0, but, we did `_tokenIds.increment()` in the `constructor` so it'll be at `1`.
6. Using _tokenIds to keep track of the NFTs Unique Identifier and it's just a number! So, when we first call `mintsCharacterNFT, newitemId` is 1. When I run it again `newitemId` will become 2.
7. `_tokenIds` is a **state variable** which means if we change it, the value is stored on the contract directly like a global variable that stays permanently in memory.
8. Now…. the magic line! When we do _safeMint(msg.sender, newItemId) it's pretty much say "Mint the NFT with newitemId to the user with address msg.sender.

Msg.sender is a variable Solidity itself provides that easily gives us access to the public address of the person calling the contract.

Keep in mind no one can call the contract anonymously, you need to have your wallet credentials connected. This is almost like "signing in" and being authenticated.

Keeping the public address isn't a problem because no one will have your wallet credentials and take all your assets.

Public address is like your home address, anyone can see where you live, but no one has your keys.

Now Keeping Track of Our Player's Data. (Oh god this one is gonna hurt.)

So, certain attributes are supposed to change depending on the character as they play. In order to store all these changes we use the variable **nftHolderAtrributes.** Which maps the tokenID of the NFT to a struct of **CharacterAttributes.** This allows me to easily update values related to the player's NFT.

```
nftHolders[msg.sender] = newItemId;
```

Now we map the User's Public wallet address to the NFT tokenID. Makes it easier to know which player owns what NFT.

```
_tokenIds.increment();
```

After the NFT is minted, we increment `tokenIds` using `_tokenIds.increment()` (which is a function OpenZeppelin gives us). This makes sure that next time an NFT is minted, it'll have a different `tokenIds` identifier. No one can have a `tokenIds` that's already been minted.

Okay running it locally.

```
let txn;

  // We only have three characters.

  // an NFT w/ the character at index 2 of our array.

  txn = await gameContract.mintCharacterNFT(2);

  await txn.wait();



  // Get the value of the NFT's URI.

  let returnedTokenUri = await gameContract.tokenURI(1);

  console.log("Token URI:", returnedTokenUri);
```

When we do mintCharcterNFT(2). HardHat will actually call this function with a default wallet that is set up for us locally. Without hardhat, it would be a pain in the ass to set it up.

Basically, `tokenUri` is a function on every NFT that returns the actual data attached to the NFT. So when I call `gameContract.tokenURI(1)` it's basically saying, *"go get me the data inside the NFT with tokenId 1"*, which would be the first NFT minted. And, it should give me back everything like: my character's name, my character's current hp, etc.

⭐ SettingUp the Token URI:

```
CharacterAttributes memory charAttributes = nftHolderAttributes[_tokenId];
```

This line actually retrieves this specific NFTs data by querying for it using it's _tokenId that was passed in to the function. So, if I did tokenURI(256) it would return the JSON data related the 256th NFT (if it existed!).

Now I formatted our JSON file and then encoded it in Base64. So it turns the JSON file into this super long, encoded string that is readable by our browser when we prepend it with data:application/json;base64,.

We add data:application/json;base64, because our browser needs to know how to read the encoded string we're passing it. In this case we're saying,

"Hey, I'm passing you a Base64 encoded JSON file, please render it properly".

Again, this is considered a standard for a majority of browsers which is perfect because we want our NFTs data to be compatible with as many existing systems as possible.

Why are we doing all this Base64 stuff? Well, basically this is how popular NFT websites like OpenSea, Rarible, and many others prefer when we pass them JSON data from our contract directly :).

Awesome. So, we're at the point we are officially minting NFTs locally and the NFT has actual data attached to it in a way that properly follows standards!

We're ready to deploy our NFT to OpenSea :).

🎉 Getting our NFTs online.

So far we have been running our application locally. The next step is getting out NFT's online.

🦊 Metamask

Next we need an Ethereum wallet. That's where MetaMask comes into play! We need to be able to call functions on our smart contract that live on the blockchain. To do that we need to have a wallet that has our Ethereum address and private key.

But, we need something to connect our website with our wallet so we can securely pass our wallet credentials to our website so our website can use those credentials to call our smart contract. You need to have valid credentials to access functions on smart contracts.

It's almost like authentication. We need something to "login" to the blockchain and then use those login credentials to make API requests from our website.

💳 Transactions

When we want to perform an action that changes the blockchain we call it a *transaction*. For example, sending someone ETH is a transaction because we're changing account balances. Doing something that updates a variable in our contract is also considered a transaction because we're changing data. Minting an NFT is a transaction because we're saving data on the contract.

Deploying a smart contract is also a transaction.

Remember, the blockchain has no owner. It's just a bunch of computers around the world run by miners that have a copy of the blockchain.

When we deploy the contract, we need to tell all of these miners, “Hey new contract coming in, please add it to the blockchain”.

Alchemy helps to do that!

Alchemy **acts as a middleman between blockchain**, the technology made famous by bitcoin, and apps that consumers might use on their phones. Its platform lets developers build applications on top of blockchains such as Ethereum.