

# A brief foray into v6

*ethers.js*

Richard Moore

a random developer

# What is ethers.js?

*A complete, compact and friendly Ethereum library.*

**The defaultProvider**

**TypeScript**

**Very few dependencies**

**Large test suite (26k+)**

**ENS as first-class citizen**

**Extensive Documentation**

**MIT License (incl. deps)**

I made ethers for myself, and I used it. A lot. A lot, a lot. :)  
(so, I'm heavily incentivized to keep making it better)





# Modern ES Features



# ES2020 BigInt

*ES2020 BigInt – Adiós, BigNumber!*

```
// Use literals when and where you want to (notice the `n`)
signer.sendTransaction({
  to: "ricmoo.eth", value: 100000000000000000000n
})

// Equality *just* works
if ((await contract.getBalance()) === oldBalance) { ... }
```

Smaller code, fewer dependencies and removes one of the most confusing classes, plus there already exists plenty of ES2020 BigInt documentation.

# ES6 Proxy

*Run-time "do the right thing", please*

```
// Works in v5 and v6  
contract["ownerOf(bytes32)"]
```

```
// Works in v6; Proxy can lookup non-normalized signatures  
contract["ownerOf ( bytes32 tokenId)"]  
contract["  ownerOf ( bytes32 ) public returns(address) "]
```

No more "duplicate ABI definition errors"; ethers only complains if you try to **use** something that's ambiguous



# Typed Values



# Typing Call Parameters

*Being meaningfully non-ambiguous, with style*

```
// An ABI with ambiguous methods... Eek!
```

```
ABI = [  
    "function foo(uint256, address)",  
    "function foo(uint256, uint256)"  
]
```

```
// Error; no way to *know* what was really meant
```

```
contract.foo(someValue, someAddr)
```

```
// But this is ok; we explicitly cast it to an address
```

```
contract.foo(someValue, Typed.address(someAddr))
```



# Keyword Call Parameters

*Shinigami eyes...*

```
// Parameters *must* be named to use keyword parameters  
ABI = [ "function transferFrom(address from, address to, uint  
value)"]
```

```
// Using positional parameters, as per usual...  
contract.transferFrom(fromAddr, toAddr, someValue)
```

```
// ...or using keyword parameters  
contract.transferFrom(Typed.keywords{  
    from: fromAddr, to: toAddr, value: someValue  
}), overrides)
```





Things Have Class(es)

# Signatures

~~*splitSignature and joinSignature*~~

```
// Signature just knows... EIP-2098, 65-bytes, r, s, v, yParity, etc.  
sig = Signature.from(await signer.signMessage("¡Hola!"))  
sig = Signature.from({ r, s, yParity })  
  
// Everything is computed and consistency checked  
console.log(sig.r, sig.s, sig.v, sig.networkV)  
console.log(sig.compactSerialized, sig.yParityAndS,  
sig.legacyChainId)
```

This opens up some powerful opportunities...



# Signatures - Raw

```
// This is fairly common practice today, which consumes 160 bytes  
// of calldata and requires some non-trivial byte manipulation  
contract TestingSignatureRaw {  
    function verify(bytes32 digest, bytes sig) public returns  
    (address){  
        (bytes32 r, bytes32 s, uint8 v) = someByteManipulationLib(sig);  
        return ecrecover(digest, v, r, s);  
    }  
}
```

ABI = [ "function **verify**(bytes32, **bytes** sig) **returns** (address)" ]

```
const someSig = "0xc2f0488159d4...1232390b801b";  
contract.verify(someDigest, someSig);
```

# Signatures - Decomposed

```
// The Signature object decodes all values as a ABI-friendly object
contract TestingSignatureDecomposed {
    struct Sig { r: bytes32, s: bytes32, v: uint8 }

    function verify(bytes32 digest, Sig sig) public returns (address) {
        return ecrecover(digest, sig.v, sig.r, sig.s);
    }
}

// JavaScript
ABI = [ "function verify(bytes32, (bytes32 r, bytes32 s, uint8 v))
        returns (address)" ]

contract.verify(someDigest, Signature.from(someSig));
```

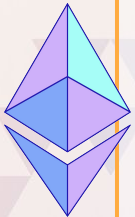
# Signatures - Compact

```
contract TestingSignatureEIP2098 {  
    struct Sig { r: bytes32, yParityAndS: bytes32 }  
  
    function verify(bytes32 digest, Sig sig) public returns (address) {  
        uint8 v = ((uint256(sig.yParityAndS) >> 255) == 0) ? 27: 28;  
        bytes32 s = bytes32((uint256(sig.yParityAndS) << 1) >> 1);  
        return ecrecover(digest, v, sig.r, s);  
    }  
}
```

*// JavaScript*

```
ABI = [ "function verify(bytes32, (bytes32 r, bytes32 yParityAndS))  
        returns (address)" ]
```

```
contract.verify(someDigest, Signature.from(someSig));
```



# Signatures - Comparison

- **Raw – bytes sig**
  - 160 bytes calldata (*1444 tx gas*)
  - semi-expensive and complex byte manipulation
- **Decomposed – struct(bytes32 r, bytes32 s, uint8 v)**
  - 96 bytes calldata (*1148 tx gas*)
  - nothing special; Solidity does all deserialization for you
- **Compact Representation – struct(bytes32 r, bytes32 yParityAndS)**
  - 64 bytes calldata (*1024 tx gas*)
  - cheap and simple math (*~15 gas*)
  - see EIP-2098 (full disclosure, it's by me and therefore somewhat shilling)
  - Notice; in Compact vs Decomposed, there was **no JavaScript change req'd**



# Transactions

~~*parseTransaction and serializeTransaction*~~



```
// Transactions can sort themselves; tx params or raw serialized tx
tx = Transaction.from(await provider.getTransaction(hash))
tx = Transaction.from(rawSerializedTx)

// Everything is computed and consistency checked
console.log(tx.nonce, tx.gasLimit, tx.hash, tx.fromPublicKey, ...)
console.log(tx.serialized, tx.unsignedSerialized)

// Changing properties causes hash, serialized forms, etc. to
“update”
tx.maxFeePerGas = 600n
```



# Bits and Bytes

# Pausing Providers

*Red light, green light...*

```
provider.on(someFilter, (log, eventPayload) => { ... })

// Schedule a provider-based timer
provider._setTimeout(() => { console.log("Woke up!"); }, 10000);

// Pause events and timers when tab is hidden, resume when unhidden
document.addEventListener("visibilitychange", () => {
  provider.paused = document.hidden;
}, false)
```

(this feature also allowed re-subscribing and resuming events; e.g. on WebSocket hang-up)

# And a lot of little things...

- **Network Plugins**

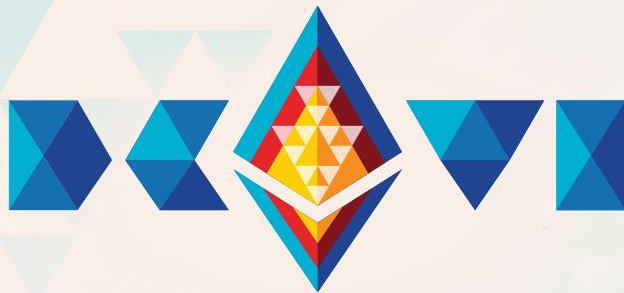
- Some networks are... strange. And that's ok!
- Customize links for backends like Etherscan, INFURA, etc.
- Custom EIP-1559 fee structures
- Extra transaction or block properties or custom hash calculations

- **Package exports (pkg.exports)**

- Build tools (bundlers, compilers, debuggers) just work! No custom magic.
- Much simpler development, better version logging and easier publishing

- **Better and Fewer Dependencies (and fewer dependant authors)**

- Easier to audit, safer code; only 4 well-established authors
- Less maintenance and external-dependency-catch-up for faster turn-around



# Thank You!

Richard Moore

a random developer

me@ricmoo.com



@ricmoo

**Questions?**



(Don't forget T.E.N.C.R.)