Understanding Explicit Function Signatures in Solidity & Ethers.js

The Problem: Function Overloading

In our enhanced DAO contract, we have two `vote` functions with different signatures:

```
// Function 1: Legacy vote (always votes in favor)
function vote(uint256 _id) external onlyInvestor { ... }

// Function 2: Enhanced vote (specify direction)
function vote(uint256 _id, bool _inFavor) external onlyInvestor { ... }
```

This is called **function overloading** - multiple functions with the same name but different parameters.

The Challenge

When using ethers.js to call these functions, the library doesn't know which one you want:

```
// ❌ This is ambiguous - which vote function?
dao.connect(signer).vote(1)
```

Ethers.js throws an error: `dao.connect(...).vote is not a function`

The Solution: Explicit Function Signatures

We use bracket notation with the full function signature to specify exactly which function to call:

```
// ✅ Calls: vote(uint256 _id)
dao.connect(signer)["vote(uint256)"](1)

// ✅ Calls: vote(uint256 _id, bool _inFavor)
dao.connect(signer)["vote(uint256,bool)"](1, true)
```

How Function Signatures Work

Signature Format

```
functionName(parameterType1,parameterType2,...)
```

Examples from Our Contract

Function Call	Signature	Parameters
`vote(1)`	`"vote(uint256)"`	`proposalId`

vote(1, true)	"vote(uint256,bool)"	proposalId, inFavor
createProposal(...)	"createProposal(string,string,uint256,address)"	name, description, amount, recipient

Real Examples from Our Tests

Legacy Vote (In Favor)

```
// Test code
transaction = await dao.connect(investor1)["vote(uint256)"](1)

// This calls the Solidity function:
// function vote(uint256 _id) external onlyInvestor
```

Enhanced Vote (Specify Direction)

```
// Vote in favor
transaction = await dao.connect(investor1)["vote(uint256,bool)"](1, true)

// Vote against
transaction = await dao.connect(investor1)["vote(uint256,bool)"](1, false)

// This calls the Solidity function:
// function vote(uint256 _id, bool _inFavor) external onlyInvestor
```

Why This Happens

1. Solidity Allows Function Overloading

```
contract Example {
    function transfer(address to, uint256 amount) { ... }
    function transfer(address to, uint256 amount, bytes data) { ... }
}
```

2. JavaScript/Ethers.js Needs Clarity

JavaScript doesn't have native function overloading, so ethers.js needs explicit instructions.

3. ABI (Application Binary Interface) Contains All Signatures

The contract's ABI includes both functions:

```
[
  {
    "name": "vote",
    "inputs": [{"type": "uint256", "name": "_id"}]
  },
  {
    "name": "vote",
    "inputs": [
      {"type": "uint256", "name": "_id"},
      {"type": "bool", "name": "_inFavor"}
    ]
  }
]
```

Alternative Approaches

1. Different Function Names (Avoided Overloading)

```
function voteInFavor(uint256 _id) external { ... }
function voteAgainst(uint256 _id) external { ... }
```

2. Single Function with Required Parameter

```
function vote(uint256 _id, bool _inFavor) external { ... }
// No overloading, always require both parameters
```

3. Using Contract Interface

```
// Create interface with specific function
const voteInterface = new ethers.utils.Interface([
  "function vote(uint256 _id, bool _inFavor)"
]);
```

Best Practices

1. Be Explicit in Tests

```
// ✅ Clear and unambiguous
dao.connect(investor1)["vote(uint256,bool)"](proposalId, false)

// ❌ Ambiguous
dao.connect(investor1).vote(proposalId, false)
```

2. Document Function Signatures

```
// Legacy vote function: vote(uint256)
const legacyVote = dao.connect(signer)["vote(uint256)"](proposalId);

// Enhanced vote function: vote(uint256,bool)
const enhancedVote = dao.connect(signer)["vote(uint256,bool)"](proposalId, t
```

3. Consistent Usage

Always use explicit signatures when function overloading exists, even if only one version is currently used.

Summary

Explicit function signatures solve the ambiguity problem when:

- Multiple functions have the same name (overloading)
- Ethers.js needs to know which specific function to call
- You want to be explicit about which version you're using

The syntax `["functionName(type1,type2)"]` tells ethers.js exactly which function signature to use from the contract's ABI.

Addendum: When You DON'T Need Explicit Signatures

Modern Approach (Recommended)

After learning more in DAPP University, we discovered that explicit function signatures are **only needed when there's ambiguity**. If you're calling the enhanced function with both parameters, ethers.js can figure it out:

```
// ✅ This works without explicit signatures
dao.connect(user).vote(1, true)   // Calls vote(uint256, bool)
dao.connect(user).vote(1, false)  // Calls vote(uint256, bool)

// ❌ This is ambiguous and needs explicit signature
dao.connect(user).vote(1)         // Which vote function?
```

Updated Test Pattern

```
// ✅ Clean and readable
await expect(dao.connect(user).vote(1, true)).to.be.revertedWith('must be to

// ❌ Unnecessarily verbose when parameters make it clear
await expect(dao.connect(user)["vote(uint256,bool)"](1, true)).to.be.reverte
```

When to Use Each Approach

Scenario	Approach	Example
Ambiguous call	Explicit signature	`dao["vote(uint256)"](1)`

Clear parameters	Direct call	`dao.vote(1, true)`
Better error testing	Direct call + revertedWith	`expect(dao.vote(1, true)).to.be.revertedWith('error')`

Key Takeaway

Use explicit function signatures only when necessary for disambiguation. When the parameters make the function call unambiguous, use the cleaner direct approach for better readability and more specific error testing.

Benefits of .revertedWith() vs .reverted

Specific Error Testing

```
// ✅ Better - Tests exact error message
await expect(dao.connect(user).vote(1, true))
  .to.be.revertedWith('must be token holder')

// ❌ Less specific - Just tests that it reverts
await expect(dao.connect(user).vote(1, true))
  .to.be.reverted
```

Why .revertedWith() is Superior

1. **Precise Testing**: Verifies the exact error condition
2. **Better Debugging**: When tests fail, you know which specific error wasn't triggered
3. **Catches Regressions**: If error messages change, tests will fail
4. **Documentation**: Error messages serve as inline documentation
5. **Contract Validation**: Ensures your contract's error handling works as expected

Common Error Messages in Our DAO

```
// Access control errors
.to.be.revertedWith('must be token holder')

// State validation errors
.to.be.revertedWith('already voted')
.to.be.revertedWith('proposal already finalized')
.to.be.revertedWith('proposal already cancelled')
.to.be.revertedWith('proposal was cancelled')

// Logic errors
.to.be.revertedWith('must reach quorum to finalize proposal')
.to.be.revertedWith('against votes must reach quorum to cancel proposal')
```

Final Addendum: When Explicit Signatures Are Actually Required

The Reality of Function Overloading

After implementing our enhanced DAO with function overloading, we discovered that explicit function signatures are **not verbose** - they're **essential** when you have overloaded functions.

Our Contract Has True Function Overloading

```
// Two different functions with the same name
function vote(uint256 _id) external onlyInvestor { ... }           // Legacy
function vote(uint256 _id, bool _inFavor) external onlyInvestor { ... }  //
```

The Misconception About "Clean" Calls

We initially thought this would work:

```
// ❌ This DOESN'T work with function overloading
dao.connect(user).vote(1, true)  // "dao.connect(...).vote is not a function
```

Why it fails: Ethers.js cannot determine which overloaded function to call, even with different parameter counts.

The Correct Approach

```
// ✅ This WORKS - explicitly specifies which function
dao.connect(user)["vote(uint256,bool)"](1, true)

// ✅ This also works for the legacy function
dao.connect(user)["vote(uint256)"](1)
```

Key Insight: Not Verbose, But Necessary

Scenario	Approach	Reality
No overloading	`dao.vote(1, true)`	Works fine
With overloading	`dao["vote(uint256,bool)"](1, true)`	**Required, not verbose**
Ambiguous call	`dao.vote(1)`	**Always needs explicit signature**

Practical Example from Our Tests

```
// This is the ONLY way that works with our overloaded functions
transaction = await dao.connect(investor1)["vote(uint256,bool)"](1, true)

// This would fail:
// transaction = await dao.connect(investor1).vote(1, true) // TypeError
```

When You Actually Need Explicit Signatures

1. **Always with function overloading** (like our DAO)
2. **When the ABI contains multiple functions with the same name**

3. **When ethers.js cannot resolve the function automatically**

The Bottom Line

Explicit function signatures aren't a "verbose alternative" - they're the **only solution** when you have function overloading. The syntax `["functionName(types)"]` is the precise, technical way to specify exactly which function you want to call from the contract's ABI.

In our DAO: Every vote call MUST use explicit signatures because we have two vote functions. This is a requirement of the architecture, not a stylistic choice.