Gas-Optimized Smart Contracts — Repo Skeleton (Code + Steps)

This document is a ready-to-fork skeleton for a gas-optimized smart-contract library. It contains files, code samples, tests, and CI steps you can paste into a repository and run.

Quick start (commands)

```
# create project
mkdir gas-optimized && cd gas-optimized
npm init -y
npm i -D hardhat @nomiclabs/hardhat-ethers ethers typescript ts-node mocha
chai eth-gas-reporter @nomicfoundation/hardhat-toolbox
npx hardhat # choose "Create an empty hardhat.config.js"
```

File: package.json (minimal)

```
{
  "name": "gas-optimized-smart-contracts",
  "version": "0.1.0",
  "scripts": {
    "test": "hardhat test",
    "coverage": "hardhat coverage",
    "typecheck": "tsc --noEmit"
  },
  "devDependencies": {
    "@nomiclabs/hardhat-ethers": "^2.0.0",
    "@nomicfoundation/hardhat-toolbox": "^3.0.0",
    "ethers": "^5.0.0",
    "hardhat": "^2.12.0",
    "typescript": "^4.0.0",
    "eth-gas-reporter": "^0.2.0"
  }
}
```

File: hardhat.config.js

```
require("@nomicfoundation/hardhat-toolbox");
require("eth-gas-reporter");
```

```
module.exports = {
  solidity: {
    version: "0.8.20",
    settings: {
      optimizer: { enabled: true, runs: 200 },
      metadata: { bytecodeHash: "none" }
    }
  },
  gasReporter: {
    enabled: true,
    currency: 'USD',
    showTimeSpent: true
  }
};
```

File: src/ERC20Packed.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

error InsufficientBalance(uint256 available, uint256 required);
error InsufficientAllowance(uint256 available, uint256 required);

contract ERC20Packed {
    // Events: keep names standard
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256
value);

    // Immutable metadata saves gas
    string public immutable name;
    string public immutable symbol;
    uint8 public immutable decimals;

    // totalSupply as single slot
    uint256 public totalSupply;

    // We use mapping(address => uint256) for balances; it's already optimal.
    mapping(address => uint256) private _balance;

    // allowances as mapping of packed allowances where key = keccak(owner,
spender)
    mapping(bytes32 => uint256) private _allowance;

    constructor(string memory _name, string memory _symbol, uint8 _decimals)
{
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
    }

    function balanceOf(address who) external view returns (uint256) {
        return _balance[who];
    }

    function allowance(address owner, address spender) public view returns
(uint256) {
        return _allowance[keccak256(abi.encodePacked(owner, spender))];
    }

    function approve(address spender, uint256 amount) external returns
(bool) {
        _allowance[keccak256(abi.encodePacked(msg.sender, spender))] =
amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _spendAllowance(address owner, address spender, uint256 amount)
internal {
        bytes32 key = keccak256(abi.encodePacked(owner, spender));
        uint256 current = _allowance[key];
        if (current < amount) revert InsufficientAllowance(current, amount);
        unchecked { _allowance[key] = current - amount; }
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        _transfer(msg.sender, to, amount);
        return true;
    }

    function transferFrom(address from, address to, uint256 amount) external
returns (bool) {
        _spendAllowance(from, msg.sender, amount);
        _transfer(from, to, amount);
        return true;
    }

    function _transfer(address from, address to, uint256 amount) internal {
        uint256 bal = _balance[from];
        if (bal < amount) revert InsufficientBalance(bal, amount);

        unchecked { _balance[from] = bal - amount; }
        _balance[to] += amount;

        emit Transfer(from, to, amount);
    }

    // mint/burn for tests & controlled use
    function _mint(address to, uint256 amount) internal {
        totalSupply += amount;
        _balance[to] += amount;
        emit Transfer(address(0), to, amount);
    }

    function _burn(address from, uint256 amount) internal {
        uint256 bal = _balance[from];
        if (bal < amount) revert InsufficientBalance(bal, amount);
        unchecked { _balance[from] = bal - amount; }
        totalSupply -= amount;
        emit Transfer(from, address(0), amount);
    }
}
```

Notes & optimizations used - Custom errors instead of revert strings. - Cache balances in local variable then `unchecked` for math. - Use `keccak256(owner,spender)` as a single mapping key to reduce nested mapping cost in some cases (tradeoff: slightly more calldata hashing cost; test both approaches in your benchmarks).

File: src/ERC20PackedToken.sol (example token that uses ERC20Packed)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "./ERC20Packed.sol";

contract ERC20PackedToken is ERC20Packed {
    address public owner;

    constructor(string memory name_, string memory symbol_)
ERC20Packed(name_, symbol_, 18) {
        owner = msg.sender;
        _mint(msg.sender, 1_000_000 * 10 ** 18);
    }

    function mint(address to, uint256 amount) external {
        require(msg.sender == owner, "only owner");
        _mint(to, amount);
    }
}
```

File: src/ERC721Lite.sol (gas-savvy NFT)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

error NotTokenOwner();
error TokenDoesNotExist();

contract ERC721Lite {
    event Transfer(address indexed from, address indexed to, uint256 indexed
tokenId);

    string public name;
    string public symbol;

    // ownerOf mapping
    mapping(uint256 => address) private _ownerOf;
    mapping(address => uint256) private _balance;

    constructor(string memory _name, string memory _symbol) {
        name = _name;
        symbol = _symbol;
    }

    function ownerOf(uint256 tokenId) public view returns (address) {
        address owner = _ownerOf[tokenId];
        if (owner == address(0)) revert TokenDoesNotExist();
        return owner;
    }

    function balanceOf(address owner) external view returns (uint256) {
        return _balance[owner];
    }

    function _mint(address to, uint256 tokenId) internal {
        if (to == address(0)) revert();
        _ownerOf[tokenId] = to;
        _balance[to] += 1;
        emit Transfer(address(0), to, tokenId);
    }

    function _transfer(address from, address to, uint256 tokenId) internal {
        address owner = _ownerOf[tokenId];
        if (owner != from) revert NotTokenOwner();
        _ownerOf[tokenId] = to;
        unchecked { _balance[from] -= 1; }
        _balance[to] += 1;
        emit Transfer(from, to, tokenId);
    }
}
```

Optimizations - Minimal approvals/allowance logic omitted for brevity (add as needed). - Keep mappings small and avoid loops.

File: src/AccessControlLite.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

error NotAuthorized();

contract AccessControlLite {
    mapping(bytes32 => mapping(address => bool)) private _roles;
    bytes32 public constant DEFAULT_ADMIN_ROLE = 0x00;

    modifier onlyRole(bytes32 role) {
        if (!_roles[role][msg.sender]) revert NotAuthorized();
        _;
    }

    function _grantRole(bytes32 role, address account) internal {
        _roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return _roles[role][account];
    }
}
```

File: src/MinimalProxyFactory.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract MinimalProxyFactory {
    event Deployed(address proxy);

    function deployMinimal(address implementation, bytes32 salt) external
returns (address proxy) {
        bytes20 impl = bytes20(implementation);
        assembly {
            let ptr := mload(0x40)
            mstore(ptr,
0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
            mstore(add(ptr, 0x14), shl(0x60, impl))
            mstore(add(ptr, 0x28),
0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
            proxy := create2(0, ptr, 0x37, salt)
        }
        require(proxy != address(0), "deploy failed");
        emit Deployed(proxy);
    }
}
```

Tests: test/ERC20Packed.test.ts (TypeScript + Hardhat)

```
import { expect } from "chai";
import { ethers } from "hardhat";

describe("ERC20PackedToken", function () {
  it("basic transfer and gas", async function () {
    const [owner, a, b] = await ethers.getSigners();
    const ERC20 = await ethers.getContractFactory("ERC20PackedToken");
    const token = await ERC20.deploy("GasToken","GST");
    await token.deployed();

    const receipt = await (await token.transfer(a.address,
ethers.utils.parseEther("1"))).wait();
    console.log("transfer gasUsed", receipt.gasUsed.toString());

    expect(await
token.balanceOf(a.address)).to.equal(ethers.utils.parseEther("1"));
  });
});
```

Scripts: scripts/deploy.ts (TypeScript)

```
import { ethers } from "hardhat";

async function main() {
  const ERC20 = await ethers.getContractFactory("ERC20PackedToken");
  const token = await ERC20.deploy("GasToken","GST");
  await token.deployed();
  console.log("ERC20PackedToken deployed to:", token.address);
}
```

```
main().catch((err) => { console.error(err); process.exitCode = 1; });
```

GitHub Actions (ci.yaml) — Minimal

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm run test
      - run: npx hardhat test --network hardhat
      - name: Upload gas report
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: gas-report
          path: gas-report.txt
```

Benchmarks & profiling steps (manual)

1. Implement a baseline in `benchmarks/openzeppelin` using OZ ERC20 standard.
2. Run `npx hardhat test` with gas-reporter enabled.
3. Capture outputs to CSV or JSON and commit to `benchmarks/`.

Example command to log gas numbers:

```
npx hardhat test 2>&1 | tee gas-report.txt
```

Optimization checklist (code changes to try and benchmark)

- Replace nested mappings for allowances with single mapping keyed by keccak where beneficial (measure effect).
- Use `immutable` for addresses/constants.
- Use `unchecked` in math when preconditions make overflow impossible.
- Use custom errors for all reverts.

- Reduce calldata size: avoid passing arrays unless necessary.
- Minimize public getters that read storage heavy data; prefer view functions that compute minimally.
- Avoid `string`/`bytes` heavy usage in hot paths.
- Use assembly only when it measurably reduces gas and maintain tests around it.

Next steps & how I can help

- I can generate additional files (e.g., OpenZeppelin baseline for comparison), or convert tests to Foundry (for faster gas testing), or produce more optimized Yul assembly versions of hot functions.

End of skeleton.