➡

Writing Typescript Functions

1) Clone the Web3 Function Hardhat Template

```
Copy git clone https://github.com/gelatodigital/web3-functions-hardhat-template.git cd web3-functions-hardhat-template
```

or use thetemplate on Github's UI.

2) Install dependencies

```
Copy yarn install
```

3) Copy the example .env file

```
Copy cp .env.example .env
```

4) Fill in your Alchemy & private key for local testing

Note:PRIVATE_KEY is optional, and only needed if you wish to deploy contracts or create a task from the CLI instead of the UI

```
Copy ALCHEMY_ID= // <= Your Alchemy key PRIVATE_KEY= // <= Your Private key
```

Typescript Function Example

This typescript function updates an oracle smart contract with data returned by Coingecko's price API at an interval. Check out more exampleshere .

```
Copy import{ Web3Function,Web3FunctionContext }from"@gelatonetwork/web3-functions-sdk"; import{ Contract,ethers }from"ethers"; importkyfrom"ky";// we recommend using ky as axios doesn't support fetch by default

constORACLE_ABI=[ "function lastUpdated() external view returns(uint256)", "function updatePrice(uint256)", ];

Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{userArgs,gelatoArgs,multiChainProvider}=context; constprovider=multiChainProvider.default();

// Retrieve Last oracle update time constoracleAddress="0x71b9b0f6c999cbbb0fef9c92b80d54e4973214da"; constoracle=newContract(oracleAddress,ORACLE_ABI,provider); constlastUpdated=parseInt(awaitoracle.lastUpdated()); console.log(Last oracle update:{lastUpdated});

// Check if it's ready for a new update constnextUpdateTime=lastUpdated+300;// 5 min consttimestamp=(awaitprovider.getBlock("latest")).timestamp; console.log(Next oracle update:{nextUpdateTime}); if(timestamp<nextUpdateTime) { return{ canExec:false,message:Time not elapsed}; }

// Get current price on coingecko constcurrency="ethereum"; constpriceData:any=awaitky .get( https://api.coingecko.com/api/v3/simple/price?ids={currency}&vs_currencies=usd, { timeout:5_000,retry:0} ) .json(); price=Math.floor(priceData[currency].usd); console.log(Updating price:{price});

// Return execution call data return{ canExec:true, callData:[{ to:oracleAddress, data:oracle.interface.encodeFunctionData("updatePrice",[price]) }], }; });
```

```

Create your functionschema.json to specify your runtime configuration.

```

Copy { "web3FunctionVersion": "2.0.0", "runtime": "js-1.0", "memory": 128, "timeout": 30, "userArgs": {} }

```

Note: For now the configuration is fixed and cannot be changed.

Typescript Function Context

When writing the Web3 Function, it is very helpful to understand thecontext Gelato injects into the execution, providing additional features to widen the Web3 Functions applicability.

```

Copy Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{userArgs,storage,secrets,multiChainProvider,gelatoArgs}=context; constprovider=multiChainProvider.default(); ... }

```

User Arguments

1. Declare your expecteduserArgs
2. in your schema, accepted types arestring
3. ,string[]
4. ,number
5. ,number[]
6. ,boolean
7. ,boolean[]
8. :
9.

```

Copy { "web3FunctionVersion":"2.0.0", "runtime":"js-1.0", "memory":128, "timeout":30, "userArgs":{ "currency":"string", "oracle":"string" } }

```

1. Access youruserArgs
2. from the Web3Function context:
3.

```

Copy Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{userArgs,gelatoArgs,secrets}=context;

// User args: console.log('Currency:',userArgs.currency) console.log('Oracle:',userArgs.oracle) ... });

```

1. In the same directory as your web3 function, create a fileuserArgs.json
2. and fill in youruserArgs
3. to test your web3 function:
4.

```

Copy { "currency":"ethereum", "oracle":"0x71B9B0F6C999CBbB0FeF9c92B80D54e4973214da" }

```

Test out the Coingecko oracle web3 function:

```

Copy npxhardhatw3f-runoracle--logs

```
```

State / Storage

Web3Functions are stateless scripts, that will run in a new & empty memory context on every execution. If you need to manage some state variable, we provide a simple key/value store that you can access from your web3 functioncontext .

See the above example to read & update values from your storage:

```
Copy import{ Web3Function, Web3FunctionContext, }from"@gelatonetwork/web3-functions-sdk";

Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{storage,multiChainProvider}=context; constprovider=multiChainProvider.default();

// Use storage to retrieve previous state (stored values are always string) constlastBlockStr= (awaitstorage.get("lastBlockNumber"))??"0"; constlastBlock=parseInt(lastBlockStr); console.log(Last block:{lastBlock});

constnewBlock=awaitprovider.getBlockNumber(); console.log(New block:{newBlock}); if(newBlock>lastBlock) { // Update storage to persist your current state (values must be cast to string) awaitstorage.set("lastBlockNumber",newBlock.toString()); }

return{ canExec:false, message:Updated block number:{newBlock.toString()}, }; });
```

To populate the storage values in your testing, in the same directory as your web3 function, create a filestorage.json and fill in the storage values.

```
Copy { "lastBlockNumber":"1000" }
```

Test out the storage web3 function:

```
Copy npx hardhat w3f-run storage --logs
```

Secrets

1. In the same directory as your web3 function, create a.env
2. file and fill up your secrets.
3.

```
Copy COINGECKO_API=https://api.coingecko.com/api/v3
```

1. Access your secrets from the Web3Function context:
2.

```
Copy // Get api from secrets constcoingeckoApi=awaitcontext.secrets.get("COINGECKO_API"); if(!coingeckoApi) { return{ canExec:false,message:COINGECKO_API not set in secrets}; }
```

1. Test your web3 function using secrets:
2.

```
Copy npxhardhatw3f-runsecrets--logs
```

1. When deploying a task, you will be able to set your web3 function secrets on our UI or using the SDK, see here
2.

```

Copy importhrefrom"hardhat"; import{ AutomateSDK,Web3Function }from"@gelatonetwork/automate-sdk";

const{ethers,w3f}=hre;

constadBoardW3f=w3f.get("advertising-board");

const[deployer]=awaitethers.getSigners(); constchainId=(awaitethers.provider.getNetwork()).chainId;

constautomate=newAutomateSDK(chainId,deployer); constweb3Function=newWeb3Function(chainId,deployer);

// Deploy Web3Function on IPFS console.log("Deploying Web3Function on IPFS..."); constcid=awaitadBoardW3f.deploy(); console.log(Web3Function IPFS CID:{cid});

// Create task using automate sdk console.log("Creating automate task...");

const{taskId,tx}=awaitautomate.createBatchExecTask({ name:"Web3Function - Ad Board", web3FunctionHash:cid, web3FunctionArgs:{}, });

awaittx.wait(); console.log(Task created, taskId:{taskId}(tx hash:{tx.hash})); console.log( > https://beta.app.gelato.network/task{taskId}?chainId={chainId} );

// Set task specific secrets constsecrets=adBoardW3f.getSecrets(); if(Object.keys(secrets).length>0) { awaitweb3Function.secrets.set(secrets,taskId); console.log(Secrets set); }

```

Multichain Provider

ThemultichainProvider allows us to instantiate RPC providers for every network Gelato is deployed on.

```

Copy import{ Web3Function, Web3FunctionContext, }from"@gelatonetwork/web3-functions-sdk";

Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{multiChainProvider}=context;

// multichainProvider.default() will instantiate // the provider of the chain the W3F is deployed constprovider=multiChainProvider.default();

// passing the chainId as follows, we can instantiate // a rpc provider for that network constpolygonProvider=multiChainProvider.chainId(137) ... }

```

When testing locally, we can provide the different providers by including them in.env at the root folder.

```

Copy // .env file PROVIDER_URLS=RPC1,RPC2

```

Interoperability with Other Libraries

AlthoughmultiChainProvider is designed to work seamlessly within the Gelato Web3 Functions SDK, it is possible to extract the underlying RPC URL and use it with other client libraries. This flexibility is valuable for developers who prefer or require features from other libraries, such asviem .

Here's an example of how to utilize the RPC URL frommultiChainProvider with theviem library, which can be useful if you need to leverage features specific toviem :

```

Copy import{ createPublicClient,http }from"viem"; import{ polygon }from"viem/chains";

Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{multiChainProvider}=context; constprovider=multiChainProvider.default(); consturl=provider.connection.url;
```

// Initialize viem client with the extracted URL constrpc=createPublicClient({ chain:polygon, transport:http(url), });

// Now you can use the viem client for your operations // ... });

```

Gelato Arguments

Gelato injects thechainId , thegasPrice , and thetaskId into the context.

1. chainId
2. : The unique number identifying the blockchain network where the function is running.
3. gasPrice
4. : The cost of executing transactions on the blockchain.
5. taskId
6. : A string that uniquely identifies the task.
7.

```

Copy import{ Web3Function, Web3FunctionContext, }from"@gelatonetwork/web3-functions-sdk";

Web3Function.onRun(async(context:Web3FunctionContext)=>{ const{gelatoArgs}=context;

// chainId: number constchainId=gelatoArgs.chainId;

// gasPrice: BigNumber constgasPrice=gelatoArgs.gasPrice;

// taskId: string consttaskId=gelatoArgs.taskId; ... }

```

Previous Quick Start Next Event Trigger Last updated3 months ago On this page *1) Clone the Web3 Function Hardhat Template * 2) Install dependencies * 3) Copy the example .env file* 4) Fill in your Alchemy & private key for local testing* Typescript Function Example * Typescript Function Context * User Arguments * State / Storage * Secrets * Multichain Provider * Interoperability with Other Libraries * Gelato Arguments
```