

# Spec Proposal Guide (Walk-thru)



## UNDER CONSTRUCTION

This guide is a work-in-progress! Please forgive errors, inconsistencies, or incompleteness.

## Overview 🔍

This guide is intended to assist someone with the process of writing and designing a spec proposal in a step-by-step way. The process of creating or maintaining a spec can seem overwhelming initially, but is actually a straightforward process once you understand the fundamentals. Follow along below to get through the entire process of making a specification.

## Step 1: Create the Proposal/JSON

Create a specification `JSON` file

Creating a specification involves multiple steps and can be an involved process. One of the best exercises you can do to prepare yourself is to visit existing specs and inspect them. Looking at existing specs can give a future spec creator a feel for the standard structure and layout of a spec.

There are three ways to start in the creation of a spec

1. 📖 **Cookbook** - All of the specs which are currently live on Lava can be found here - in the [Lava cookbook](#). Use one of them to get started, changing fields as needed.
2. 📄 **Template** - A blank template is available, right [here](#) from the docs. Simply fill in the blanks as you progress.
3. ✏️ **From Scratch** - Writing a spec completely from scratch gives the opportunity to avoid unnecessary messiness and superfluous fields.



## Define Proposal

The recommended approach is to start from scratch and go field by field. Let's start with a blank JSON and create our top-level field. This key is called `proposal` and it describes the specifications we're about to propose:

```
{
  "proposal": {
    "title": "Add Specs: X",
    "description": "Adding new specification support for relaying X data on
Lava"
  }
}
```

The format of it is pretty much standard, so you can copy the example above and replace `X` with the name of the API being created.

🚩 REFERENCE: `Proposal`

## Define Specs

Each proposal introduces new `specs` to Lava. As mentioned elsewhere, specs are the minimum requirements that an API provider must meet in order to serve the API. Lava uses several fields to establish the initial parameters for a spec including an `index`, a `name`, the `data_reliability_enabled` marker, and the `minimum_stake_provider` must make.

These fields combined with others make the first section of a spec:

```
{
  "proposal": {
    "title": "Add Specs: X",
    "description": "Adding new specification support for relaying X data on
Lava",
    "specs": [
      {
        "index": "X",
        "name": "X mainnet",
        "enabled": true,
        "reliability_threshold": 268435455,
        "data_reliability_enabled": true,
```



```
"block_distance_for_finalized_data": 1,
"blocks_in_finalization_proof": 1,
"average_block_time": 5000,
"allowed_block_lag_for_qos_sync": 2,
"shares": 1,
"min_stake_provider": {
  "denom": "ulava",
  "amount": "50000000000"
}
}
]
}
}
```

🔍 Let's investigate these fields one-by-one:

#### ▼ `index`

The index is the universal identifier for a spec. The index must not be shared by any other spec. It is what will be referenced anywhere else the spec is referenced or imported. The naming convention method for an index is to use all caps, no spaces. Shorter/more abbreviated indexes are preferred. Optionally, to truncate long indexes, many specs employ the strategy of dropping vowels or shortening words (e.g. `OPTM` for Optimism, `STRK` for Starknet, or `AGR` for Agoric) .

#### ▼ `name`

The name is the longform descriptive identifier for a spec. The name should indicate what exactly the spec is/distinguish from specs which serve similar data. If it is a testnet or mainnet of a specific blockchain it should say so.

#### ▼ `enabled`

The enabled field describes whether the spec is active. There are times when a spec is defined but not used - or when a spec is to be deactivated temporarily. In our case, this should default to `true`.



### ▼ `reliability\_threshold`

---

Reliability threshold sets the frequency of reliability related messages. By default, we set this to `268435455` which is the minimum and efficient choice. If you'd like to set it higher - there are more details that can be learned here. [/spec-reference#terms]

### ▼ `data\_reliability\_enabled`

---

Data reliability should be enabled unless there is a compelling reason to disable it! The default value here is `true`. This means Lava protocol will work to ensure data is accurate by doing data reliability checks. Note that if you are creating a spec for something other than a blockchain - you will want to set this to `false`.

⚠ As of `lava-testnet-2` support for data reliability on diverse APIs is work-in-progress.

### ▼ `block\_distance\_for\_finalized\_data`

---

This field defines the number of blocks that should be considered safe from chain reorganization; it varies from chain to chain. Look to other similarly architected specs for suggested values.

### ▼ `blocks\_in\_finalization\_proof`

---

This field defines the number of blocks that should be found in a finality proof; this will vary from chain to chain. Look to other similarly architected specs for suggested values.

### ▼ `average\_block\_time`

---

The amount of time, on average, that a block passes in milliseconds. This field is used in several algorithms by the protocol to ensure provider quality of service.



### ▼ `allowed\_block\_lag\_for\_qos\_sync`

This is the maximum amount of blocks that can pass before the data a provider serves is considered stale. For faster blockchains/data sources, more blocks will be allowed. For slower blockchains/data sources, less blocks is suitable. Look to other similarly architected specs for values.

#### ▼ `shares`

The default is `1`. This is a bonus multiplier for rewards at the end of each month. There should be no reason to change this unless dictated otherwise.

#### ▼ `min\_stake\_provider`

This field defines the minimum amount that a provider must have staked to serve this API. This value can remain identical with default values supplied by all other specs during our testnet. As of `lava-testnet-2`, the amount is `5000000` in denom `ulava`.

🚀 REFERENCE: `specs`

Once each of these fields has been dealt with, we're ready to move onto setting up inheritance!

## Step 2: Inheritance

| Inherit attributes from an existing spec.

Before defining any APIs, it's wise to pull others that already exist to save work. Most APIs in Lava use common methods and can be created from specs already in use on-chain. In Lava, we call this process of borrowing from earlier specs **inheritance**. Specs use inheritance to eliminate redundancy, minimize size, and save time. An additional benefit - as inherited specs are updated - their descendent specs automatically pull in their updates! Inheritance makes things easy.

## Imports

An import generically brings in all parse directives, verifications, API Collections, and APIs by default. To overwrite specific mandatory behavior - simply define the `parse_directive`, verification,



API Collection, or API by its `name` ( or `function_tag` if editing a parse directive) in the spec which is inheriting.

To inherit, a new spec, use the `imports` field:

```
"specs": [
  {
    "index": "X",
    "name": "X mainnet",
    "enabled": true,
    "imports": [
      "ETH1"
    ],
    "reliability_threshold": 268435455,
    "data_reliability_enabled": true,
    "block_distance_for_finalized_data": 1,
    "blocks_in_finalization_proof": 1,
    "average_block_time": 5000,
    "allowed_block_lag_for_qos_sync": 200,
    "min_stake_provider": {
      "denom": "ulava",
      "amount": "50000000000"
    },
  },
]
```

Common imports will include one of the following:

"COSMOSSDK"	# Cosmos SDK Standard (i.e. Cosmos Chains)
"COSMOSSDK45DEP"	# Cosmos SDK v0.45 deprecated apis
"COSMOSSDKFULL"	# Cosmos SDK + COSMWASM
"ETH1"	# Ethereum-based chains supporting standard EVM RPC calls
"SOLANA"	# Solana-based chains
"OPTM"	# Optimism-based cahins

#### ! INFO

More often than not, a spec will only use one of the aforementioned imports. Specs are completely modular and can import any other specs. This modular design pattern comes in handy, for example, when designing a spec for mainnet and a spec for testnet. Usually, the testnet spec simply inherits the mainnet spec and requires no further configurations.



## Inheritance APIs

If you're picky about the imports you want to do, it is possible to specify individual APIs, using the `inheritance_apis` field under an `api_collection`. If you're confused don't worry - we'll explain more about API Collections next. For now, it is strongly recommended that you use imports instead. Remember, you can always disable unused `apis` and `api_collections`.

🔗 REFERENCE: `imports`

## Step 3: API Collections

Specify the API collections and interfaces which are mandatory for Providers.

Each spec can contain several categories of API Collections. API Collections are split across different interfaces, although they are constructed similarly. If a spec only contains the APIs of another spec it imports, it may not be necessary to define API collections at all...

Some example API Collections are defined (with differences *highlighted*) below:

**JSONRPC**   **gRPC**   **REST**   **TendermintRPC**

```
"api_collections": [  
  {  
    "enabled": true,  
    "collection_data": {  
      "api_interface": "jsonrpc",  
      "internal_path": "",  
      "type": "POST",  
      "add_on": ""  
    },  
    "apis": [],  
    "headers": [],  
    "inheritance_apis": [],  
    "parse_directives": [],  
    "verifications": [],  
  },  
]
```

Each API collection is composed of various pieces. These pieces collectively give definition to the APIs that a Provider will serve. It's important that we review these pieces in detail so that you're

familiar with what goes where:

📌 REFERENCE: `api_collections`

📌 REFERENCE: `collection_data`

## Collection Data

### ▼ `api\_interface`

```
# pick one of the following:  
"tendermintrpc"  
"grpc"  
"jsonrpc"  
"rest"
```

### ▼ `internal\_path`

This field gives the internal path of the node for this specific ApiCollection. This is **most likely unneeded** unless the API sets vary on internal paths on teh node. The best example is the [AVAX specification](<https://raw.githubusercontent.com/lavanet/lava/main/cookbook/avalanche.json>) which uses internal paths to distinguish between subnets with distinct ApiCollections.

### ▼ `type`

```
# pick one of the following:  
""  
"GET"  
"POST"  
"PUT"  
"PATCH"  
"DELETE"
```





### ▼ `add\_on`

Leaving this field as a blank string("") is the default and expected input. If you add anything to the string, the API Collection will be processed as an addon with the name provided in the string. Under that condition, the collection will be treated as optional to providers. We cover addons in more detail in a [later section](#) of this guide.

## Other Fields

### APIs

This is an array will contain all of the collection's APIs - outlined in such a manner that you can see the compute units. There is a **whole section** dedicated to adding APIs to an API Collection, so we can leave this blank for now, as well.


### Headers

It is possible to specify headers to be used in the API using this array. Leave this blank for now: `[]` unless you want to identify headers that a consumer can send along with their request.

Each Header is composed of a `name` and a `kind`, optionally a `function_tag`.

#### ▶ An Example Header

### Kinds of Headers

Header	Description	Example
pass_ignore	Relies on node-specific information and excludes header from reliability.	Time tag of reply on Aptos (varies per node).
pass_reply	Node returns header to user; user cannot request it. One-way.	Ledger version 0, e.g., x-  aptos-echo from node.

Header	Description	Example
pass_both	Two-way communication: Node and user can both send and receive headers.	Cosmos block.
pass_send	User can send to node, but node cannot send to user. One-way.	Instruction headers.

It is possible to use a `function_tag` to parse the header's response. The `function_tag` *must* correlate to an existing `parse_directive`.

## Parse Directives

Because every API returns data in a different format, Lava protocol establishes a standardized way to deal with data parsing. These standards are called `parse directives`. Parse directives are a critical part of how API responses are handled. Please take a moment to familiarize yourself with Lava parsing functions before continuing:

🚀 REFERENCE: [Parsing](#)

If a spec is imported, then this is most likely already handled for you and does not require definition. However, in case it is not, there is a need for

```
{
  "function_tag": "GET_BLOCK_BY_NUM",
  "function_template": "
{\\\"jsonrpc\\\":\\\"2.0\\\",\\\"method\\\":\\\"starknet_getBlockWithTxHashes\\\",\\\"params\\\":
[\\\"{\\\"block_number\\\":%d}\\\",\\\"id\\\":1}\\\"],
  \"result_parsing\": {
    \"parser_arg\": [
      \"0\",
      \"block_hash\"
    ],
    \"parser_func\": \"PARSE_CANONICAL\",
    \"encoding\": \"base64\"
  },
  \"api_name\": \"starknet_getBlockWithTxHashes\"
},
{
  \"function_template\": \"
```



```
{\"jsonrpc\": \"2.0\", \"method\": \"starknet_blockNumber\", \"params\":  
[], \"id\": 1},  
  \"function_tag\": \"GET_BLOCKNUM\",  
  \"result_parsing\": {  
    \"parser_arg\": [  
      \"0\"  
    ],  
    \"parser_func\": \"PARSE_BY_ARG\"  
  },  
  \"api_name\": \"starknet_blockNumber\"  
}
```

#### ▼ `function\_tag`

This is the global name and identification of the parse\_directive. Anywhere else that a parse\_directive is referenced it will be referenced by this name.

#### ▼ `function\_template`

This is the (JSON) template from which the response will be parsed. It is used to identify the standard format of responses.

#### ▼ `api\_name`

The `api\_name` refers to the specific API that will be parsed by the parse\_directive. It should correlate to a defined api in the `api\_collections` or one inherited.

### DANGER

Get\_BlockNum and Get\_Block\_by\_Num must be defined for Lava data reliability checks to succeed. If your API does not support block numbers - please ensure that `data_reliability_enabled` is set to `false`.



# Step 4: New APIs

Design APIs which were not inherited from another spec.

```
{
  "name": "blockHashAndNumber",
  "block_parsing": {
    "parser_arg": [
      "latest"
    ],
    "parser_func": "DEFAULT"
  },
  "compute_units": 10,
  "enabled": true,
  "category": {
    "deterministic": true,
    "local": false,
    "subscription": false,
    "stateful": 0
  },
  "extra_compute_units": 0
},
```

## Block Parsing

This area is used to describe how to extract the block number from the API request. Make sure to review the parsing reference and several spec examples to ensure it's defined correctly.

🔗 REFERENCE: [Block Parsing](#), [Parsing](#)

## Compute Units

Describes the number of compute units which each API call expends. This number is a proxy for the compute intensiveness/difficulty and therefore the cost of calling this API. Note: compute units are not just tethered to rewards - they also indirectly inform the protocol of the expected time to response; by default, each compute unit adds ~100 ms to the relay's timeout threshold.

There are a minimum of 10 CU per call - this should be sufficient for most calls.

🚧 Note that `extra_compute_units` is presently not used, but will be useful for varying cost based upon consumer arguments.



🚀 REFERENCE: [Compute Units](#)

## Category

### ▼ `deterministic`

---

**true** if deterministic responses from API (*default*)

**false** disables data reliability for non-deterministic responses.

### ▼ `local`

---

**true** if local information from the node is returned through the API.

**false** if the local information on the node is irrelevant to response. (*default*)

### ▼ 🚧 `subscription`

---

UNDER CONSTRUCTION => mark **false**

subscription indicates when to open up a streaming API with the provider (wss is currently disabled).

### ▼ `stateful`

---

Manages nonce consistency. Use **1** to propagate information to all providers, **0** for no propagation.

🚀 REFERENCE: [Category](#)

## Other Fields

For other fields, please take a look at the reference(s) and observe other specs.

🚀 REFERENCE: [APIs](#)



# Step 5: Verifications

Define tests which confirm that a Provider is serving the proper data

Earlier, we looked at Parse Directives as a means for understanding the type of data that a relay returns. A verification is a `parse_directive` combined with an `expected_value`. It provides a means for the protocol to intelligently check if the provider is serving the correct data. Each `API Collection` has its own set of verifications. Define verifications like below:

```
"verifications": [
  {
    "name": "enabled",
    "parse_directive": {
      "function_template": "
{\\\"jsonrpc\\\":\\\"2.0\\\",\\\"method\\\":\\\"getRawHeader\\\",\\\"params\\\":
[\\\"latest\\\",\\\"id\\\":1}\\\",
      "function_tag": "VERIFICATION",
      "result_parsing": {
        "parser_arg": [
          "0"
        ],
        "parser_func": "PARSE_BY_ARG",
        "encoding": "hex"
      },
      "api_name": "getRawHeader"
    },
    "values": [
      {
        "expected_value": "*"
      }
    ]
  }
]
```

The default behavior of verifications is to restrict a provider from serving the APIs if failed. This can be altered with a field called `severity` but it is not recommended behavior unless specific to your usecase.

🔗 REFERENCE: `Verifications`



# Step 6: Addons/Extensions (Optional APIs)

Define optional API Collections which a Provider may choose to serve for more CU

Specs are both highly modular and composable. Sometimes, the minimum requirements of a provider may not be satisfactory for all consumers on the network. A great example is for archive nodes; not every Provider on a network needs to serve Archive data, but for those who want to opt-in you can define the rules and rewards using extensions. Addons are additional sets of API Collections that are not mandatory- a great example of an addon would be a node which answers debug APIs!

## Creating Addons

Making an Addon is very similar to making any other API Collection. The sole difference is that the `add_on` field must contain a unique name.

```
{
  "enabled": true,
  "collection_data": {
    "api_interface": "jsonrpc",
    "internal_path": "",
    "type": "POST",
    "add_on": "debug"
  },
}
```

🔗 REFERENCE: `Addons`

## Creating Extensions

Making an Extension follows a slightly different process than making an Addon. We define extensions as an array which is a child of an `api_collection` object:

### Archive Example

```
"extensions": [
  {
    "name": "archive",
    "cu_multiplier": 5,
    "rule": {
      "block": 254
    }
  }
]
```



```
}  
]
```

**archive** Providers must return blocks from at least 254 blocks from latest, thus receiving 5x the CU.

### Censorship Example

```
"extensions": [  
  {  
    "name": "censorship",  
    "cu_multiplier": 2,  
    "rule": {  
      "block": 1  
    }  
  }  
]
```

**censorship** Providers may only return blocks 1 block away from the latest, thus receiving 2x the CU rewards.

#### ! INFO

Currently, rules and extensions are hard-coded. As of the time of this guide, "block" is the only rule defined in code and "archive" is the only recognized extension.

🔗 REFERENCE: **Extensions**

## Step 7: Verifications for Optional APIs

Define tests for Providers who serves addons and extension API Collections

### Verifications for Addons

Verifications for addons are simple! They are defined in the **exact same way** as they are for other api\_collections; when defining an api\_collection as an addon, populate **verifications** with your verifications.



## Verifications for Extensions

Verifications for extensions are similarly simple. Within `verifications`, under the child `values`, create another entry with the `extension` name like so:

```
"values": [  
  {  
    "latest_distance": 6646  
  },  
  {  
    "extension": "archive",  
    "expected_value": "0x0"  
  }  
]
```

## Step 8: Test with Local Blockchain

Use the `test_spec_full.sh` script to automatically execute local tests.

### Install Lava

1. Install [Lava Binaries](#) on Your Local Machine
2. Check that the `test_spec_full.sh` exists in the `./scripts` folder of your install

### Run Command

```
./scripts/test_spec_full.sh cookbook X.json <interface> <rpc_url_for_index1>  
<interface> <rpc_url_for_index2>
```

It will scaffold a local block chain and create a test network of several providers running the spec! You can see errors in real-time which will alert you to where you need to debug. Once you have debugged all issues- go on to the next step!



# Step 9: Push to your Repository & Share

Add your `JSON` file to your local `cookbook/spec/` directory.

Share your progress with the [Lava Team & Community!](#)

 [Edit this page](#)

