World Reserve System (WRS)

The World's most stable currency

API

Draft Version: 0.01.01 (very early stage)

Author: Ramin Assisi, Computer Scientist

Contents

Inhaltsverzeichnis

Introduction

This is the official Javascript library for the WRS Core. It implements both the official API, API wrapper functions as well as newly proposed functionality (such as signing, bundles, utilities and conversion).

It should be noted that the Javascript Library as it stands right now is an **early alpha release**. As such, there might be some unexpected results. Please join the community (see links below) and post [issues on here](https://github.com/rassisi/worldreservesystem/issues), to ensure that the developers of the library can improve it.

> **Join the Discussion**
>
> If you want to get involved in the community, need help with getting setup, have any issues related with the library or just want to discuss Blockchain, Distributed Ledgers and IoT with other people, feel free to join our Slack. Slack You can also ask questions on our dedicated forum at: WRS Forum.

Installation

```
npm install wrs.lib.js
```

```
bower install wrs.lib.js
```

The Javascript library can be easily installed either via `npm` or via `bower`. You can also directly use the browserify'ed version, which can be found in the `dist` folder or you can compile it yourself by cloning the repo and running `gulp`.You can either use `wrs.js` or the minified version `wrs.min.js` in the browser.

Getting Started

```
// Create WRS instance with host and port as provider
var wrs = new WRS({
    'host': 'http://localhost',
    'port': 14265
});

// Create WRS instance directly with provider
var wrs = new WRS({
    'provider': 'http://localhost:14265'
});

// now you can start using all of the functions
```

```
wrs.api.getNodeInfo();
```

After you’ve successfully installed the library, it is fairly easy to get started by simply launching a new instance of the WRS object with an optional settings object. When instantiating the object you have the option to decide the API provider that is used to send the requests to and you can also connect directly to the Sandbox environment.

The optional settings object can have the following values:

1. `host`: `String` Host you want to connect to. Can be DNS, IPv4 or IPv6. Defaults to `localhost`
2. `port`: `Int` port of the host you want to connect to. Defaults to 14265.
3. `provider`: `String` If you don’t provide host and port, you can supply the full provider value to connect to
4. `sandbox`: `Bool` Optional value to determine if your provider is the WRS Sandbox or not.

You can either supply the remote node directly via the `provider` option, or individually with `host` and `port`, as can be seen in the example on the right.

Overall, there are currently four subclasses that are accessible from the WRS object:

- `api`: Core API functionality for interacting with the WRS core.
- `utils`: Utility related functions for conversions, validation and so on
- `multisig`: Functions for creating and signing multi-signature addresses and transactions.
- `validate`: Validator functions that can help with determining whether the inputs or results that you get are valid.

In the future new WRS Core modules (such as Flash, MAM) and all IXI related functionality will be available.

How to use the Library

```
wrs.api.getNodeInfo(function(error, success) {
    if (error) {
        console.error(error);
    } else {
        console.log(success);
    }
})
```

It should be noted that most API calls are done asynchronously. What this means is that you have to utilize callbacks in order to catch the response successfully. We will add support for sync API calls, as well as event listeners in future versions.

On the right is a simple example of how to access the `getNodeInfo` function.

api

Standard API

This Javascript library has implemented all of the core API calls that are made available by the current WRS Reference Implementation. For the full documentation of all the Standard API calls, please refer to the official documentation: official API.

You can simply use any of the available options from the `api` object then. For example, if you want to use the `getTips` function, you would simply do it as such:

```
wrs.api.getTips(function(error, success) {
    // do stuff here
})
```

---

getTransactionsObjects

Wrapper function for `getTrytes` and the Utility function `transactionObjects`. This function basically returns the entire transaction objects for a list of transaction hashes.

Input

```
wrs.api.getTransactionsObjects(hashes, callback)
```

1. `hashes`: `Array` List of transaction hashes
2. `callback`: `Function` callback.

Return Value

1. `Array` - list of all the transaction objects from the corresponding hashes.

---

findTransactionObjects

Wrapper function for `getTrytes` and the Utility function `transactionObjects`. This function basically returns the entire transaction objects for a list of transaction hashes.

Input

```
wrs.api.findTransactionObjects(hashes, callback)
```

1. `hashes`: `Array` List of transaction hashes
2. `callback`: `Function` callback.

Return Value

1. `Array` - list of all the transaction objects from the corresponding hashes.

---

getLatestInclusion

Wrapper function for `getNodeInfo` and `getInclusionStates`. It simply takes the most recent solid milestone as returned by getNodeInfo, and uses it to get the inclusion states of a list of transaction hashes.

Input

```
wrs.api.getLatestInclusion(hashes, callback)
```

1. `hashes`: `Array` List of transaction hashes
2. `callback`: `Function` callback.

Return Value

1. `Array` - list of all the inclusion states of the transaction hashes

---

broadcastAndStore

Wrapper function for `broadcastTransactions` and `storeTransactions`.

Input

```
wrs.api.broadcastAndStore(trytes, callback)
```

1. `trytes`: `Array` List of transaction trytes to be broadcast and stored. Has to be trytes that were returned from `attachToTangle`
2. `callback`: `Function` callback.

Return Value

`Object` - empty object.

---

getNewAddress

Generates a new address from a seed and returns the address. This is either done deterministically, or by providing the index of the new address to be generated.

Input

```
wrs.api.getNewAddress(seed [, options], callback)
```

1. `seed`: `String` tryte-encoded seed. It should be noted that this seed is not transferred
2. `options`: `Object` which is optional:
    - `index`: `Int` If the index is provided, the generation of the address is not deterministic.
    - `checksum`: `Bool` Adds 9-tryte address checksum

    - `total`: `Int` Total number of addresses to generate.
    - `returnAll`: `Bool` If true, it returns all addresses which were deterministically generated (until findTransactions returns null)
3. `callback`: `Function` Optional callback.

Returns

`String | Array` - returns either a string, or an array of strings.

---

getInputs

Gets all possible inputs of a seed and returns them with the total balance. This is either done deterministically (by genearating all addresses until `findTransactions` returns null for a corresponding address), or by providing a key range to use for searching through.

You can also define the minimum `threshold` that is required. This means that if you provide the `threshold` value, you can specify that the inputs should only be returned if their collective balance is above the threshold value.

Input

```
wrs.api.getInputs(seed, [, options], callback)
```

1. `seed`: `String` tryte-encoded seed. It should be noted that this seed is not transferred
2. `options`: `Object` which is optional:
    - `start`: `int` Starting key index
    - `end`: `int` Ending key index
    - `threshold`: `int` Minimum threshold of accumulated balances from the inputs that is requested
3. `callback`: `Function` Optional callback.

Return Value

1. `Object` - an object with the following keys:
    - `inputs` `Array` - list of inputs objects consisting of `address`, `balance` and `keyIndex`
    - `totalBalance` `int` - aggregated balance of all inputs

---

prepareTransfers

Main purpose of this function is to get an array of transfer objects as input, and then prepare the transfer by **generating the correct bundle**, as well as **choosing and signing the inputs** if necessary (if it's a value transfer). The output of this function is an array of the raw transaction data (trytes).

You can provide multiple transfer objects, which means that your prepared bundle will have multiple outputs to the same, or different recipients. As single transfer object takes the values of: `address`, `value`, `message`, `tag`. The message and tag values are required to be tryte-encoded.

For the options, you can provide a list of `inputs`, that will be used for signing the transfer's inputs. It should be noted that these inputs (an array of objects) should have the provided `keyIndex` and `address` values: `var inputs = [{ 'keyIndex': //VALUE, 'address': //VALUE }]`

The library validates these inputs then and ensures that you have sufficient balance. The `address` parameter can be used to define the address to which a remainder balance (if that is the case), will be sent to. So if all your inputs have a combined balance of 2000, and your spending 1800 of them, 200 of your tokens will be sent to that remainder address. If you do not supply the `address`, the library will simply generate a new one from your seed.

Input

```
wrs.api.prepareTransfers(seed, transfersArray [, options], callback)
```

1. `seed`: `String` tryte-encoded seed. It should be noted that this seed is not transferred
2. `transfersArray`: `Array` of transfer objects:
   - `address`: `String` 81-tryte encoded address of recipient
   - `value`: `Int` value to be transferred.
   - `message`: `String` tryte-encoded message to be included in the bundle.
   - `tag`: `String` Tryte-encoded tag. Maximum value is 27 trytes.
3. `options`: `Object` which is optional:
   - `inputs`: `Array` List of inputs used for funding the transfer
   - `address`: `String` if defined, this address will be used for sending the remainder value (of the inputs) to.
4. `callback`: `Function` Optional callback.

Return Value

`Array` - an array that contains the trytes of the new bundle.

---

sendTrytes

Wrapper function that does `attachToTangle` and finally, it broadcasts and stores the transactions.

Input

```
wrs.api.sendTrytes(trytes, depth, minWeightMagnitude, callback)
```

1. `trytes` `Array` trytes
2. `depth` `Int` depth value that determines how far to go for tip selection
3. `minWeightMagnitude` `Int` minWeightMagnitude

4. `callback`: `Function` Optional callback.

Returns

`Array` - returns an array of the transfer (transaction objects).

---

sendTransfer

Wrapper function that basically does `prepareTransfers`, as well as `attachToTangle` and finally, it broadcasts and stores the transactions locally.

Input

```
wrs.api.sendTransfer(seed, depth, minWeightMagnitude, transfers [, options],
callback)
```

1. `seed` `String` tryte-encoded seed. If provided, will be used for signing and picking inputs.
2. `depth` `Int` depth
3. `minWeightMagnitude` `Int` minWeightMagnitude
4. `transfers`: `Array` of transfer objects:
   - `address`: `String` 81-tryte encoded address of recipient
   - `value`: `Int` value to be transferred.
   - `message`: `String` tryte-encoded message to be included in the bundle.
   - `tag`: `String` 27-tryte encoded tag.
5. `options`: `Object` which is optional:
   - `inputs`: `Array` List of inputs used for funding the transfer
   - `address`: `String` if defined, this address will be used for sending the remainder value (of the inputs) to.
6. `callback`: `Function` Optional callback.

Returns

`Array` - returns an array of the transfer (transaction objects).

---

replayBundle

Takes a tail transaction hash as input, gets the bundle associated with the transaction and then replays the bundle by attaching it to the tangle.

Input

```
wrs.api.replayBundle(transaction [, callback])
```

1. `transaction`: `String` Transaction hash, has to be tail.
2. `depth` `Int` depth
3. `minWeightMagnitude` `Int` minWeightMagnitude

4. `callback`: `Function` Optional callback

---

broadcastBundle

Takes a tail transaction hash as input, gets the bundle associated with the transaction and then rebroadcasts the entire bundle.

Input

```
wrs.api.broadcastBundle(transaction [, callback])
```

1. `transaction`: `String` Transaction hash, has to be tail.
2. `callback`: `Function` Optional callback

---

getBundle

This function returns the bundle which is associated with a transaction. Input has to be a tail transaction (i.e. currentIndex = 0). If there are conflicting bundles (because of a replay for example) it will return multiple bundles. It also does important validation checking (signatures, sum, order) to ensure that the correct bundle is returned.

Input

```
wrs.api.getBundle(transaction, callback)
```

1. `transaction`: `String` Transaction hash of a tail transaction.
2. `callback`: `Function` Optional callback

Returns

`Array` - returns an array of the corresponding bundle of a tail transaction. The bundle itself consists of individual transaction objects.

---

getTransfers

Returns the transfers which are associated with a seed. The transfers are determined by either calculating deterministically which addresses were already used, or by providing a list of indexes to get the addresses and the associated transfers from. The transfers are sorted by their timestamp. It should be noted that, because timestamps are not enforced in WRS, that this may lead to incorrectly sorted bundles (meaning that their chronological ordering in the Tangle is different).

If you want to have your transfers split into received / sent, you can use the utility function `categorizeTransfers`

Input

`getTransfers(seed [, options], callback)`

1. `seed`: `String` tryte-encoded seed. It should be noted that this seed is not transferred
2. `options`: `Object` which is optional:
   - `start`: `Int` Starting key index for search
   - `end`: `Int` Ending key index for search
   - `inclusionStates`: `Bool` If True, it gets the inclusion states of the transfers.
3. `callback`: `Function` Optional callback.

Returns

`Array` - returns an array of transfers. Each array is a bundle for the entire transfer.

---

getAccountData

Similar to `getTransfers`, just a bit more comprehensive in the sense that it also returns the `balance` and `addresses` that are associated with your account (seed). This function is useful in getting all the relevant information of your account. If you want to have your transfers split into received / sent, you can use the utility function `categorizeTransfers`

Input

`getAccountData(seed [, options], callback)`

1. `seed`: `String` tryte-encoded seed. It should be noted that this seed is not transferred
2. `options`: `Object` which is optional:
   - `start`: `Int` Starting key index for search
   - `end`: `Int` Ending key index for search
3. `callback`: `Function` Optional callback.

Returns

`Object` - returns an object of your account data in the following format: `{ 'addresses': [], 'transfers': [], 'balance': 0 }`

multisig

Multi signature related functions.

> **VERY IMPORTANT NOTICE**
>
> Before using these functions, please make sure that you have thoroughly read our guidelines for multi-signature. It is of utmost importance that you follow these rules, else it can potentially lead to financial losses.

---

getKey

Generates the corresponding private key of a seed.

Input

`wrs.multisig.getKey(seed, index)`

1. `seed`: `String` Tryte encoded seed
2. `index`: ‘Int’ Index of the private key.

Returns

`String` - private key represented in trytes.

---

getDigest

Generates the digest value of a key.

Input

`wrs.multisig.getDigest(seed, index)`

1. `seed`: `String` Tryte encoded seed
2. `index`: 'Int’ Index of the private key.

Returns

`String` - digest represented in trytes.

---

addAddressDigest

This function is used to initiate the creation of a new multisig address. The way that it works is that the first participant of the multi-signature initiates this function with an empty curl state, and then shares the newly generated state with the other participants of the multisig address, who then basically add their key digest. Then finally, once the last co-signer added their digest, `finalizeAddress` can be used to get the actual 81-tryte address value. `validateAddress` can be used to actually validate the multi-signature.

Input

`wrs.multisig.addAddressDigest(digestTrytes, curlStateTrytes)`

1. `digestTrytes`: `String` digest trytes as returned by `getDigest`
2. `curlStateTrytes`: 'String’ curl state trytes to continue modifying (which are returned by this function)

Returns

`String` - curl state trytes

---

finalizeAddress

Finalizes the multisig address generation process and returns the correct 81-tryte address.

Input

```
wrs.multisig.finalizeAddress(curlStateTrytes)
```

1. `curlStateTrytes`: 'String’ curl state trytes to continue modifying (which are returned by this function)

Returns

`String` - curl state trytes

---

validateAddress

Validates a generated multi-sig address by getting the corresponding key digests of each of the co-signers. The order of the digests is of essence in getting correct results.

Input

```
wrs.multisig.validateAddress(multisigAddress, digests)
```

1. `multisigAddress`: `String` digest trytes as returned by `getDigest`
2. `digests`: 'Array’ array of the key digest for each of the cosigners. The digests need to be provided in the correct signing order.

Returns

`Bool` - true / false

---

initiateTransfer

Initiates the creation of a new transfer by generating an empty bundle with the correct number of bundle entries to be later used for the signing process. It should be noted that currently, only a single input (via `inputAddress`) is possible. The `remainderAddress` also has to be provided and should be generated by the co-signers of the multi-signature before initiating the transfer.

Input

```
wrs.multisig.initiateTransfer(inputAddress, remainderAddress, numCosigners,
transfers, callback)
```

1. `inputAddress`: `String` input address which has sufficient balance and is controlled by the co-signers
2. `remainderAddress`: 'String' in case there is a remainder balance, send the funds to this address.
3. `numCosigners`: `Int` the number of co-signers for the multi-sig
4. `transfers`: `Array` Transfers object
5. `callback`: `Function`

Returns

`Array` - bundle

---

addSignature

This function is called by each of the co-signers individually to add their signature to the bundle. Here too, order is important. This function returns the bundle, which should be shared with each of the participants of the multi-signature.

Input

```
wrs.multisig.addSignature(bundleToSign, cosignerIndex, inputAddress, key,
callback)
```

1. `bundleToSign`: `Array` bundle to sign
2. `cosignerIndex`: `Int` total order index of the current signer in the multi-signature. Index starts at 0. e.g. If there are 4 co-signers, and you are the 3rd in order to add your signature, then your index is `2`.
3. `inputAddress`: 'String' input address as provided to `initiateTransfer`.
4. `key`: `String` private key trytes as returned by `getKey`
5. `callback`: `Function`

Returns

`Array` - bundle

---

validateSignatures

This function makes it possible for each of the co-signers in the multi-signature to independently verify that a generated transaction with the corresponding signatures of the co-signers is valid. This function is safe to use and does not require any sharing of digests or key values.

Input

```
wrs.multisig.validateSignatures(signedBundle, inputAddress, numCosigners)
```

1. `signedBundle`: `Array` signed bundle by all of the co-signers

2. `inputAddress`: 'String' input address as provided to `initiateTransfer`.
3. `numCosigners`: `Int` total number of co-signers

Returns

`bool` - true / false

utils

convertUnits

WRS utilizes the Standard system of Units. See below for all available units:

```
'i'   :   1,
'Ki'  :   1000,
'Mi'  :   1000000,
'Gi'  :   1000000000,
'Ti'  :   1000000000000,
'Pi'  :   1000000000000000
```

Input

```
wrs.utils.convertUnits(value, fromUnit, toUnit)
```

1. `value`: `Integer || String` Value to be converted. Can be string, an integer or float.
2. `fromUnit`: `String` Current unit of the value. See above for the available units to utilize for conversion.
3. `toUnit`: `String` Unit to convert the from value into.

Returns

`Integer` - returns the converted unit (fromUnit => toUnit).

---

addChecksum

Takes an 81-trytes address or a list of addresses as input and calculates the 9-trytes checksum of the address(es).

Input

```
wrs.utils.addChecksum(address)
```

1. `address`: `String | List` Either an individual address, or a list of addresses.

Returns

`String | List` - returns the 90-trytes addresses (81-trytes address + 9-trytes checksum) either as a string or list, depending on the input.

---

noChecksum

Takes an 90-trytes address as input and simply removes the checksum.

Input

`wrs.utils.noChecksum(address)`

1. `address`: `String | List` 90-trytes address. Either string or a list

Returns

`String | List` - returns the 81-tryte address(es)

---

isValidChecksum

Takes an 90-trytes checksummed address and returns a true / false if it is valid.

Input

`wrs.utils.isValidChecksum(addressWithChecksum)`

1. `addressWithChecksum`: `String` 90-trytes address

Returns

`Bool` - True / False whether the checksum is valid or not

---

transactionObject

Converts the trytes of a transaction into its transaction object.

Input

`wrs.utils.transactionObject(trytes)`

1. `trytes`: `String` 2673-trytes of a transaction

Returns

`Object` - Transaction object

---

transactionTrytes

Converts a valid transaction object into trytes. Please refer to [TODO] for more information what a valid transaction object looks like.

Input

`wrs.utils.transactionTrytes(transactionObject)`

1. `transactionObject`: `Object` valid transaction object

Returns

`trytes` - converted trytes of

---

categorizeTransfers

Categorizes a list of transfers into `sent` and `received`. It is important to note that zero value transfers (which for example, is being used for storing addresses in the Tangle), are seen as `received` in this function.

Input

`wrs.utils.categorizeTransfers(transfers, addresses)`

1. `transfers`: `List` A list of bundles. Basically is an array, of arrays (bundles), as is returned from getTransfers or getAccountData
2. `addresses`: 'List' List of addresses that belong to you. With these addresses as input, it's determined whether it's a sent or a receive transaction. Therefore make sure that these addresses actually belong to you.

Returns

`object` - the transfers categorized into `sent` and `received`

validate

isAddress

Checks if the provided input is a valid 81-tryte (non-checksum), or 90-tryte (with checksum) address.

Input

`wrs.validate.isAddress(address)`

1. `address`: `String` A single address

---

isTrytes

Determines if the provided input is valid trytes. Valid trytes are: `ABCDEFGHIJKLMNOPQRSTUVWXYZ9`. If you specify the length parameter, you can also validate the input length.

Input

```
wrs.validate.isTrytes(trytes [, length])
```

1. `trytes`: `String`
2. `length`: `int || string` optional

---

isValue

Validates the value input, checks if it's integer.

Input

```
wrs.validate.isValue(value)
```

1. `value`: `Integer`

---

isDecimal`

Checks if it's a decimal value

Input

```
wrs.validate.isDecimal(value)
```

1. `value`: `Integer || String`

---

isHash

Checks if correct hash consisting of 81-trytes.

Input

```
wrs.validate.isHash(hash)
```

1. `hash`: `String`

---

isTransfersArray

Checks if it’s a correct array of transfer objects. A transfer object consists of the following values: `{ 'address': // STRING (trytes encoded, 81 or 90 trytes) 'value': // INT 'mesage': // STRING (trytes encoded) 'tag': // STRING (trytes encoded, maximum 27 trytes) }`

Input

`wrs.validate.isTransfersArray(transfersArray)`

1. `transfersArray`: `array`

---

isArrayOfHashes

Array of valid 81 or 90-trytes hashes.

Input

`wrs.validate.isArrayOfHashes(hashesArray)`

1. `hashesArray`: `Array`

---

isArrayOfTrytes

Checks if it’s an array of correct 2673-trytes. These are trytes either returned by prepareTransfers, attachToTangle or similar call. A single transaction object is encoded 2673 trytes.

Input

`wrs.validate.isArrayOfTrytes(trytesArray)`

1. `trytesArray`: `Array`

---

isArrayOfAttachedTrytes

Similar to `isArrayOfTrytes`, just that in addition this function also validates that the last 243 trytes are non-zero (meaning that they don’t equal 9). The last 243 trytes consist of: `trunkTransaction` + `branchTransaction` + `nonce`. As such, this function determines whether the provided trytes have been attached to the tangle successfully. For example this validator can be used for trytes returned by `attachToTangle`.

Input

`wrs.validate.isArrayOfAttachedTrytes(trytesArray)`

1. `trytesArray`: `Array`

---

isUri

Work in progress. If this is still here while you're reading the documentation, tell either Dominik or someone else from Core to move their asses.

Input

```
wrs.validate.isUri(uris)
```

1. `uris`: `Array`

---

isInputs

Validates if it's an array of correct input objects. These inputs are provided to either `prepareTransfers` or `sendTransfer`. An input objects consists of the following:

```
{
    'keyIndex': // INT
    'address': // STRING
}
```

Input

```
wrs.validate.isInputs(inputsArray)
```

1. `inputsArray`: `Array`

---

isString

Self explanatory.

Input

```
wrs.validate.isString(string)
```

---

isInt

Self explanatory.

Input

```
wrs.validate.isInt(int)
```

---

isArray

Self explanatory.

Input

```
wrs.validate.isArray(array)
```

---

isObject

Self explanatory.

Input

```
wrs.validate.isObject(array)
```

---

isUri

Validates a given string to check if it's a valid IPv6, IPv4 or hostname format. The string must have a `udp://` prefix, and it may or may not have a port. Here are some example inputs:

```
udp://[2001:db8:a0b:12f0::1]:14265
udp://[2001:db8:a0b:12f0::1]
udp://8.8.8.8:14265
udp://domain.com
udp://domain2.com:14265
```

Input

```
wrs.utils.isUri(node)
```

1. `node`: `String` IPv6, IPv4 or Hostname with or without a port.

Returns

`bool` - true/false if valid node format.
