Bynar Documentation

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Date | Reason for Change | Version |
| Michelle Zhong | 10/8/2019 | Outline the Document | 0.1 |
| Michelle Zhong | 10/9/2019 | Outline the Document Modules, fill in the API section, Config File section, start filling out the Backend Section | 0.2 |
| Michelle Zhong | 10/10/2019 | Reorganize Headers in API section, Fill out the Backend, add Database Schema, add Error Module, Host Information, Helper Library | 0.3 |
| Michelle Zhong | 10/11/2019 | Update Database Schema, Add Client, Jira Modules, Database Logging Section | 0.4 |
| Michelle Zhong | 10/14/2019 | Start Updating the Disk Testing Section | 0.5 |
| Michelle Zhong | 10/16/2019 | Updated the Disk Testing Section, Add Hardware Testing Section, add in the main Bynar program section | 0.6 |

# Table of Contents

[Revision History 2](#_Toc22131549)

[Table of Contents 3](#_Toc22131550)

[API 6](#_Toc22131551)

[Introduction 6](#_Toc22131552)

[Messages 6](#_Toc22131553)

[Enums 6](#_Toc22131554)

[Structs 7](#_Toc22131555)

[Configuration Files 9](#_Toc22131556)

[Introduction 9](#_Toc22131557)

[List of Config Files 9](#_Toc22131558)

[Bynar JSON 9](#_Toc22131559)

[Ceph JSON 10](#_Toc22131560)

[Disk-Manager JSON 10](#_Toc22131561)

[Backend 10](#_Toc22131562)

[Introduction 10](#_Toc22131563)

[Backend Module 10](#_Toc22131564)

[Enums 10](#_Toc22131565)

[Interface 10](#_Toc22131566)

[Ceph 11](#_Toc22131567)

[Structs 11](#_Toc22131568)

[Helper Functions 15](#_Toc22131569)

[Database Schema 19](#_Toc22131570)

[Introduction 19](#_Toc22131571)

[Postgres 19](#_Toc22131572)

[Schema 20](#_Toc22131573)

[Database Logging 20](#_Toc22131574)

[Introduction 20](#_Toc22131575)

[Logging 20](#_Toc22131576)

[Enums 20](#_Toc22131577)

[Structs 21](#_Toc22131578)

[Interface and Helper Functions 24](#_Toc22131579)

[Helper Functions 29](#_Toc22131580)

[Introduction 29](#_Toc22131581)

[Error Module 29](#_Toc22131582)

[Type 29](#_Toc22131583)

[Enums 29](#_Toc22131584)

[Structs 30](#_Toc22131585)

[Host Information 31](#_Toc22131586)

[Enums 31](#_Toc22131587)

[Structs 31](#_Toc22131588)

[Helper Functions 32](#_Toc22131589)

[Helper Module 32](#_Toc22131590)

[Structs 33](#_Toc22131591)

[Helper Functions 33](#_Toc22131592)

[Client 35](#_Toc22131593)

[Introduction 35](#_Toc22131594)

[Client Interface 35](#_Toc22131595)

[Support Tickets 37](#_Toc22131596)

[Introduction 37](#_Toc22131597)

[JIRA Support 37](#_Toc22131598)

[Disk Manager 37](#_Toc22131599)

[Introduction 37](#_Toc22131600)

[Disk Manager 37](#_Toc22131601)

[Structs 37](#_Toc22131602)

[Functions 38](#_Toc22131603)

[Disk Testing 40](#_Toc22131604)

[Introduction 40](#_Toc22131605)

[State Machine 40](#_Toc22131606)

[Type 41](#_Toc22131607)

[Trait 41](#_Toc22131608)

[Enums 41](#_Toc22131609)

[Structs 42](#_Toc22131610)

[Functions 48](#_Toc22131611)

[Hardware Testing 51](#_Toc22131612)

[Introduction 51](#_Toc22131613)

[Hardware Tests 51](#_Toc22131614)

[Struct 51](#_Toc22131615)

[Functions 52](#_Toc22131616)

[Bynar 53](#_Toc22131617)

[Introduction 53](#_Toc22131618)

[Main Process Functions 53](#_Toc22131619)

# API

## Introduction

This package uses Protobuf version 2 to create Messages that can be sent over the network. Protobuf is a fast and small protocol for serializing structs (or structured data). Serialized messages can be sent between Sockets, unpackaged, and read easily and quickly. The protobuf package automatically generates the rust code needed to create, modify, and destroy Messages as well as their attributes.

## Messages

### Enums

#### DiskType

The type of disk or device

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| LOOPBACK | Special loopback device |
| LVM | Logical Volume Device |
| MDRAID | Linux software RAID |
| NVME | Non-Volatile Memory Express, a logical device interface |
| RAM | Ramdisk |
| ROTATIONAL | Regular rotational device |
| SOLID\_STATE | SSD |
| VIRTUAL | Virtual Disk |
| UNKNOWN | Unknown disk |

#### ResultType

A result value

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| OK | ok |
| ERR | There is an error |

#### Op

An operation on a disk

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Add | Generic Add Disk command, returns an OpResult |
| AddPartition | Add a Partition Command, returns an OpResult |
| List | List the Disks, returns a list of Disks |
| Remove | Remove a Disk, returns an OpResult |
| SafeToRemove | Checks if a Disk is safe to remove, returns a bool |
| GetCreatedTickets | list created tickets, returns a list of created tickets |

#### DatacenterOp

Datacenter API’s, these all require server\_id as a parameter for the operation

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| GetDc | Get ? Returns an OpStringResult |
| GetRack | Get the rack of a server, returns an OpStringResult |
| GetRow | Get the row of a server, returns an OpStringResult |
| GetElevation | Get the elevation of a server, returns an OpStringResult |

### Structs

#### Osd

A Ceph OSD object descriptor

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| fsid | Option<String> | OSD File System ID, if one exists |
| id | u64 | OSD ID number |
| block\_device | String | Block Device of the OSD |
| journal | Option<String> | Name of the Journal if the OSD has one set |
| active | bool | Whether or not an OSD is active or a spare |
| used\_space | u64 | How much space in the OSD is currently used |
| total\_space | u64 | Total space in the OSD |

#### Partition

A single partition descriptor

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| uuid | String | The id of the partition |
| first\_lba | u64 | The first logical block address of the partition |
| last\_lba | u64 | The last logical block address of the partition |
| flags | u64 | Flags associated with the partition |
| name | String | The name of the partition |

#### PartitionInfo

A list of Partitions

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| partition | Vec<Partition> | List of partitions |

#### Disk

A disk object descriptor

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| type | DiskType | The type of disk |
| dev\_path | String | ?? Device path? |
| partitions | PartitionInfo | Disk partitions |
| serial\_number | Option<String> | Disk serial number |

#### OpResult

A result of an Op message

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| result | ResultType | Whether the result is ok or Error |
| error\_msg | Option<String> | Error message if there is an error |

#### OpBoolResult

A boolean result of an Op message

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| result | ResultType | Whether Ok or Error |
| value | Option<bool> | A value is set if OK |
| error\_msg | Option<String> | Error message is set if there is an Error |

#### OpStringResult

A String result of an Op message

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| result | ResultType | Whether Ok or Error |
| value | Option<String> | A value is set if OK |
| error\_msg | Option<String> | Error message is set if there is an Error |

#### JiraInfo

A Jira Ticket information descriptor

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| ticket\_id | String | Ticket number |
| server\_name | String | Name of the JIRA server |

#### OpJiraTicketsResult

A Jira ticket result

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| result | ResultType | Whether Ok or Error |
| value | Option<String> | A value is set if OK |
| error\_msg | Option<String> | Error message is set if there is an Error |

#### DatacenterOperation

A Datacenter operation message

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Op\_type | DatacenterOp | The type of operation to be performed |
| server\_id | String | The ID of the server to be operated on |

#### Operation

A service operation that can be performed

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Op\_type | Op | The operation type |
| disk | Option<String> | The disk name, used for an Add or Remove |
| simulate | Option<bool> | Whether the operation is a simulation, used for Add, Remove, and SafeToRemove |
| partition\_start | Option<u64> | Optional field for AddPartition, start of a partition |
| partition\_end | Option<u64> | Optional field for AddPartition, end of a partition |
| partition\_name | Option<String> | Optional field for AddPartition, partition name |
| osd\_id | Option<u64> | Optional Ceph related field, the id of an OSD |
| replica\_set | Vector<String> | Host:/dev/disk strings list for gluster replica sets |

# Configuration Files

## Introduction

Bynar uses a set of configuration files to configure different settings. Bynar uses JSON as the format for its configuration files, as JSON files are easily parsed, serialized, and deserialized using the Rust serde and serde-json crates.

## List of Config Files

### Bynar JSON

This config file, bynar.json, is used to configure several different settings, including a Slack webhook, JIRA support, Redfish access, Vault password access, and Postgres database access

|  |  |  |
| --- | --- | --- |
| Name | Description | Example Value |
| proxy | Proxy web server? | “https://my.proxy” |
| manager\_host | The host ip of the bynar disk manager | “localhost” |
| manager\_port | The port of the Bynar disk manager | 5555 |
| slack\_webhook | Slack webhook to access Slack API | "<https://hooks.slack.com/services/ID>" |
| slack\_channel | Slack channel to post messages to | “#my-channel" |
| slack\_botname | Name of the Bot to post messages under | "my-bot" |
| jira\_user | JIRA username to create tickets under | “test\_user” |
| jira\_password | JIRA password | “user\_pass” |
| jira\_host | JIRA host to create tickets under | “https://tickets.jira.com” |
| jira\_issue\_type | JIRA issue type name to create tickets of | “3” |
| jira\_priority | JIRA priority value of tickets created | “4” |
| jira\_project\_id | JIRA project id to create tickets under | “MyProject” |
| jira\_ticket\_assignee | User created JIRA tickets are assigned to | “assignee\_username” |
| redfish\_ip | IP address of a Redfish instance | “localhost” |
| redfish\_username | Username to access Redfish instance | “redfish\_user” |
| redfish\_password | Password to access Redfish | “redfish\_pass” |
| redfish\_port | Port of the Redfish instance | 4443 |
| vault\_endpoint | Hashicorp vault endpoint | “https://my\_vault.com” |
| vault\_token | Hashicorp vault token to access the vault | “token\_234464562” |
| database | List of Database parameters |  |
| database:username | Username to access database with | “postgres” |
| database:password | Password to access database with | “” |
| database:port | Port of the database | 5432 |
| database:dbname | Name of the database | “bynar” |
| database:endpoint | Database endpoint | “some.endpoint” |

### Ceph JSON

This config file, ceph.json, is used to tell Bynar where the ceph.conf file is, what user to use when running Ceph commands, and what journal devices are known?

|  |  |  |
| --- | --- | --- |
| Name | Description | Example Value |
| config\_file | The path to the ceph.conf file | “/etc/ceph/ceph.conf” |
| user\_id | User to use when running Ceph commands | “admin” |
| journal\_devices | Journal device list |  |
| journal\_devices:device | Path of the device | “/dev/sda” |
| journal\_devices:partition\_id | Partition ID number | 1 |

### Disk-Manager JSON

This config file, disk-manager.json is used to tell Bynar what the backend storage system is

|  |  |  |
| --- | --- | --- |
| Name | Description | Example Value |
| backend | The backend type of the storage system | "ceph” |

# Backend

## Introduction

Different distributed storage clusters have different ways of adding and removing disks, the backend module seeks to create an interface to the different backends

## Backend Module

A Generic Module for interfacing with different storage backends

### Enums

#### BackendType

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Ceph | Ceph is the backend type |
| Gluster | GlusterFS is the backend type |

##### Trait Implementations

###### FromStr

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| from\_str | s: &str | Converts a string to a BackendType. Return Ok(BackendType) if successful or an Error if the string is not a BackendType | BynarResult<BackendType> |

###### Clone, Debug, Deserialize

### Interface

#### Backend

##### Trait Function Definition

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| add\_disk | device: &Path  id: Option<u64>  simulate: bool | Add a disk at path *device*, *id* an optional OSD id for Ceph clusters to ensure the OSD is set to that id, if *simulate* is passed no action is taken. Returns Ok(()) if successful or an Error if one occurs | BynarResult<()> |
| remove\_disk | device: &Path  simulate: bool | Remove a disk at path *device* from a cluster. If *simulate* is passed no action is taken. Returns Ok(()) if successful or an Error if one occurs | BynarResult<()> |
| safe\_to\_remove | device: &Path  simulate: bool | Check if safe to remove a disk from a cluster at path *device*. If *simulate* passed then return true. Returns Ok(true) if successful and safe, Ok(false) if successful and not safe to remove, or an Error if one occurs | BynarResult<bool> |

##### Public Functions

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| load\_backend | backend\_type: &BackendType  config\_dir: Option<&Path> | Given a BackendType, *backend\_type,* and a config file directory from *config\_dir*, return Ok(Backend) if successful or Error if one occurs. | BynarResult<Box<dyn Backend>> |

## Ceph

The Ceph backend implementation

### Structs

#### CephBackend

This is a public struct object defining a Ceph cluster

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| cluster\_handle | Rados | A handle to the ceph librados |
| config | CephConfig | Handle for the Ceph Configuration File |
| version | CephVersion | The Ceph Version |

##### Implementation

|  |
| --- |
| Function Definition |
| new(config\_dir: Option<&Path>) -> BynarResult<()>  DESCRIPTION: Create a new CephBackend  PARAMETERS: config\_dir – the directory of the ceph.json file or NONE if in the .config directory of the HOME directory  RETURNS: Ok(CephBackend) on success, Error otherwise  IMPLEMENTATION: Get the ceph.json file from the config\_dir parameter. If successful, create the CephConfig object from the ceph.json file. Using the CephConfig object, connect to the specified Ceph instance using the specified user id to get the librados handle. Using the Rados handle, get the Ceph version string and convert it into a CephVersion object. If all steps are successful return a new CephBackend object with the CephConfig, Rados handle, and CephVersion. |
| add\_bluestore\_osd(&self, dev\_path:&Path, id:Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: Add a bluestore OSD to the Ceph Cluster  PARAMETERS: dev\_path – the device path of the OSD  id– the OSD id of the OSD to add  simulate – if passed skip execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Find a journal device that has enough free space? Create a new osd and get its osd\_id (if id is not NONE then the new osd id should match id. Create an osd\_fsid, and use it, the osd id, the device path, and the journal to create an lvm. Create a mount point path for the drive if necessary. Write the osd fsid to a file. Resolve the created lvm name to a true device path and chown it so ceph can use it. Symlink the lvm device name to the mount point’s /block, and if a journal device with enough space was found, symlink the journal to the mount point’s /block.wal and change the permissions so ceph can use it. Write activate monmap out by getting the map, and creating a file activate.monmap. Lookup the ceph user id and change all the permissions on the created files so ceph can use them. Create a ceph authorization entry, get the keyring created and save it. Format the osd with the osd filesystem. Use the ceph bluestore tool, and add the osd to the crush. Enable the osd, and then initialize the osd. If all steps are successful return (), else it error’d out somewhere. |
| add\_filestore\_osd(&self, dev\_path:&Path, id:Option<u64>, simulate:bool) -> BynarResult<()>  DESCRIPTION: Add a new /dev/ path as an osd, with xfs, for Jewel or earlier  PARAMETERS: dev\_path – the device path of the OSD  id– the OSD id of the OSD to add  simulate – if passed skip execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Format the drive with the Xfs filesystem. Prove the drive by getting the device info and checking if it has a filesystem id. Create a new osd and get its id, which should be the same as the input id if one was input. Create the mount point path and mount the drive. Select a journal with enough space (if there is one, can be None). Format the osd with the osd filesystem. Create a ceph authorization entry, get the authorization key and save the keyring. Add the osd to the crush, add the osd to the fstab, then init the osd. If all steps are successful return (), else it error’d out somewhere. |
| change\_permissions(&self, paths: &[&Path], perms: &Passwd) -> BynarResult<()>  DESCRIPTION: change permissions of many files at once  PARAMETERS: paths – the paths of the files to change the permissions of  perms – the group and owner permissions to change the file permissions to  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: loop through the paths and chown each path to the input permission values. If all steps are successful return (), else it error’d out somewhere. |
| create\_lvm(&self, osd\_fsid: &uuid::Uuid, new\_osd\_id: u64, dev\_path: &Path, journal\_device: Option<&JournalDevice>) -> BynarResult<(PathBuf, u64)>  DESCRIPTION: Create the lvm device and return the path and size of it  PARAMETERS: osd\_fsid – the osd filesystem id  new\_osd\_id – the id of the osd  dev\_path – the path to the device of the osd  journal\_device – an optional journal device ? Dunno what it’s used for...  RETURNS: Ok(PathToLvm,Size) on success, Error otherwise  IMPLEMENTATION: probe the device for its udev info. create a volume group name, and logical volume name, and use them to create the logical volume device name. Initialize a new LVM, and scan it. create the volume group on the LVM, then add the device path to the volume group by extending it and writing. create a linear logical volume in the volume group, create its tags. If all steps are successful return the path to the lvm device name and the volume group size, else it error’d out somewhere. |
| create\_lvm\_tags(&self, lv: &LogicalVolume<\_,\_>, lv\_dev\_name: &Path, osd\_fsid: &uuid::Uuid, new\_osd\_id:u64, info:&block\_utils::Device, journal\_device:Option<&JournalDevice)->BynarResult<()>  DESCRIPTION: Add the lvm tags that ceph requires to identify the osd  PARAMETERS: lv – the logical volume  lv\_dev\_name – the path to the logical volume device  osd\_fsid – the osd filesystem id  new\_osd\_id – the id of the osd  info – the device info  journal\_device – an optional journal device ? Dunno what it’s used for...  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create the lvm tags. If there is a journal device input, add a tag for the wal\_device and add the wal\_uuid. Once all tags are created add them to the logical volume. If all steps are successful return (), else it error’d out somewhere. |
| remove\_bluestore\_osd(&self, dev\_path:&Path, simulate:bool) -> BynarResult<()>  DESCRIPTION: Remove a bluestore OSD to the Ceph Cluster  PARAMETERS: dev\_path – the device path of the OSD  simulate – if passed skip execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Initialize an lvm and scan it for volume groups and LVM metadata. Get the volume group that the device is associated with, if it cannot find the volume group, check if it is a filestore and if so fall back. otherwise, open the volume group and list all logical volumes in the volume group. List the tags to get the osd id and osd fsid. Set the osd as out, remove it from the crush, delete the authorization key, stop the osd, and remove it. Then, wipe the disk. remove all the logical volumes associated with the volume group, remove the volume group, and remove the physical volume and erase the physical volume. Then disable the osd. If all steps are successful return (), else it error’d out somewhere. |
| remove\_filestore\_osd(&self, dev\_path: &Path, simulate: bool) -> BynarResult<()>  DESCRIPTION: Remove a bluestore OSD to the Ceph Cluster  PARAMETERS: dev\_path – the device path of the OSD  simulate – if passed skip execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the mountpoint of the dev path and get the osd\_id. Set the osd as out, remove it from the crush, delete the osd auth key, and remove the osd. Then, wipe the disk by erasing the block device. If all steps are successful return (), else it error’d out somewhere. |
| resolve\_lvm\_device(&self, lv\_dev\_name: &Path) -> BynarResult<PathBuf>  DESCRIPTION: Resolve the lvm device name to an absolute path, since the lvm device name is a symlink, so it needs to be resolved to an absolute path to do anything with it.  PARAMETERS: lv\_dev\_name – the device name of the lvm  RETURNS: Ok(Lvm Absolute Path) on success, Error otherwise  IMPLEMENTATION: read the symlink. If it is a relative path, get its parent and the relative path to its parent, and canonicalize it, which returns the canonical, absolute form of a path with all intermediate components normalized and symbolic links resolved. If all steps are successful return the absolute path, else it error’d out somewhere. |
| select\_journal(&self) -> BynarResult<Option<JournalDevice>>  DESCRIPTION: Find a journal device that has enough free space if there is one  PARAMETERS:  RETURNS: Ok(Some(JournalDevice)) or Ok(None) on success, Error otherwise  IMPLEMENTATION: get the journal size from the Rados config. Convert it from MB to bytes. Get the journal devices from the ceph.json and sort them by the number of partitions. Iterate over the journal devices and remove the devices that are too small, and take the first journal with enough space. If all steps are successful, return Ok(Some(JournalWithEnoughSpace)) or Ok(None) if there are no journals with enough space, else it error’d out somewhere. |

##### Trait Implementation

###### Backend

|  |
| --- |
| Trait Function Definition |
| add\_disk(&self, device: &Path, id: Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: Add a disk to the Cluster  PARAMETERS: device – the device path of the disk to add  id – an optional id to give the osd  simulate – if passed, skip the evaluation of this function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: If the Ceph Version is >= Luminous, then run add\_bluestore\_osd. Otherwise, run add\_filestore\_osd. If all steps are successful return (), else it error’d out somewhere. |
| remove\_disk(&self, device:&Path, simulate: bool) -> BynarResult<()>  DESCRIPTION: remove a disk from the Cluster  PARAMETERS: device – the device path of the disk to add  simulate – if passed skip execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: check if the Ceph Version is >= Luminous. If so, run remove\_bluestore\_osd. Otherwise, run remove\_filestore\_osd. If all steps are successful return (), else it error’d out somewhere. |
| safe\_to\_remove(&self, \_device:&Path, \_simulate:bool) -> BynarResult<()>  DESCRIPTION: check if a disk is safe to remove from the cluster  PARAMETERS: device – the unused device path of the disk to remove  simulate – if passed skip execution of the function  RETURNS: Ok(True) or Ok(False)on success, Error otherwise  IMPLEMENTATION: Create a DiagMap and run an exhaustive check. If all steps are successful, then return true if the Status is Safe, return false if the Status is NonSafe or Unknown, otherwise the function error’d out somewhere. |

#### JournalDevice

A Journal Device

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| device | PathBuf | The device name? Device path??? |
| partition\_id | Option<u32> | The id of the partition |
| partition\_uuid | Option<uuid::Uuid> | The user? Unique? id of the partition |
| num\_partitions | Option<usize> | The number of partitions in the Journal Device |

##### Implementation

|  |
| --- |
| Function Definition |
| update\_num\_partitions(&mut self) -> BynarResult<()>  DESCRIPTION: Discover the number of partitions on the device and update the num\_partitions field  PARAMETERS:  RETURNS: Ok(CephBackend) on success, Error otherwise  IMPLEMENTATION: use GPT to get the number of partitions from the partition table, and update the num\_partitions field. If all steps are successful, then return (), else the function error’d out somewhere |

##### Trait Implementation

###### Display

|  |
| --- |
| Trait Function Definition |
| fmt(&self, f: &mut fmt::Formatter) -> fmt::Result  DESCRIPTION: format the journal device for printing as a string/displaying as a string  PARAMETERS: f: some formatter  RETURNS: Ok(()) on success, fmt::Result error type otherwise  IMPLEMENTATION: if there is a partition\_id, display the device and the id, otherwise just display the device. |

###### Clone, Debug, Deserialize, PartialEq

#### CephConfig

The ceph configuration object descriptor

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| config\_file | String | The location of the ceph.conf file |
| user\_id | String | The cephx user to connect to the Ceph service with |
| journal\_devices | Option<Vec<JournalDevice>> | The /dev/xxx devices to use for journal partitions. Bynar will create new partitions on these devices as needed if no journal\_partition\_id is given. |

##### Trait Implementation

###### Deserialize, Debug

### Helper Functions

|  |
| --- |
| Helper Function Definition |
| choose\_ceph\_config(config\_dir: Option<&Path>) -> BynarResult<PathBuf>  DESCRIPTION: get the path of the ceph.json file.  PARAMETERS: config\_dir – an optional path to the configuration directory  RETURNS: Ok(ceph.json path) on success, Error otherwise  IMPLEMENTATION: check if a config\_dir was provided. If so, check the directory for a ceph.json file. If a config\_dir is not provided, check in the Home directory under the .config directory for the ceph.json file. If the function was successful return Ok(ceph.json path) else the function error’d somewhere. |
| get\_osd\_id\_from\_path(path: &Path) -> BynarResult<u64>  DESCRIPTION: A fallback function to get the osd id from the mount path. Note, is not 100% accurate but will work for most cases unless the disk is mounted in the wrong location or is missing the osd id in the path name  PARAMETERS: path – the mount path  RETURNS: Ok(osd id) on success, Error otherwise  IMPLEMENTATION: get the last part of the path (file or directory name). If successful, split the name by ‘-’, and the osd-id SHOULD be the second item in the list created by the split. If the function was successful return Ok(osd\_id) else the function error’d somewhere. |
| get\_osd\_id(path: &Path, simulate: bool) -> BynarResult<u64>  DESCRIPTION: Get the osd id from the whoami file in the osd mount directory  PARAMETERS: path – the osd mount directory  RETURNS: Ok(osd id) on success, Error otherwise  IMPLEMENTATION: make the path to the whoami file, and read the whoami file. Contained in the whoami file should be the osd\_id, so convert that into a u64 and return it. if the function is successful return Ok(osd\_id), else the function error’d somewhere |
| save\_keyring(osd\_id: u64, key: &str, uid: Option<u32>, gid: Option<u32>, simulate: bool) -> BynarResult<()>  DESCRIPTION: save a Ceph authentication key to a keyring file (Note: as of now it also overwrites the keyring file every time....)  PARAMETERS: osd\_id – the osd id  key – the authentication key  uid – the uid of the user who will own the keyring file  gid – the gid of the group that will own the keyring file  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: convert the uid and guid into Uid and Gid types. Get the path to the base directory and check if it exists. If so, create the keyring file and write the key to the created file, and chown it to the uid and gid. If successful, return Ok(()), otherwise the function error’d out. |
| add\_osd\_to\_fstab(device\_info: &block\_utils::Device, osd\_id: u64, simulate: bool) -> BynarResult<()>  DESCRIPTION: add the osd to the file systems table (fstab)  PARAMETERS: device\_info: device information gathered from udev  osd\_id – the osd id  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the default value of the fstab (default path is /etc/fstab). Create an entry for the fstab, filling in the attributes: the device id for the fs\_spec, the mount point of the osd, the filesystem type, and the mount options, the dump, and fsck\_order. Add the entry to the fstab. If the function is successful, return Ok(()), else the function error’d somewhere |
| partition\_in\_use(partition\_uuid: &uuid::Uuid) -> BynarResult<bool>  DESCRIPTION: Look through all the /var/lib/ceph/osd/ directories and check if there is a matching partition id to the input id.  PARAMETERS: partition\_uuid – the uid of the partition to check  RETURNS: Ok(partition in use or not) on success, Error otherwise  IMPLEMENTATION: for each osd in the system, get the journal symlink and do a sanity check on the journal symlink. Get the metadata of the symlink and do another sanity check. resolve the symlink path to get the device and probe it. Get the partition uid from the device and compare to the input path. If the same, then return Ok(true), if not the same return Ok(false), otherwise it error’d |
| systemctl\_disable(osd\_id: u64, osd\_uuid: &uuid::Uuid, simulate: bool) -> BynarResult<()>  DESCRIPTION: run the systemctl disable command on an osd  PARAMETERS: osd\_id – the id of the osd  osd\_uuid – the user id? Of the osd  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the command arguments, and create a new Command to run the systemctl command. If the command is successful, return Ok(()), else it error’d |
| systemctl\_enable(osd\_id: u64, osd\_uuid: &uuid::Uuid, simulate: bool) -> BynarResult<()>  DESCRIPTION: run the systemctl enable command on an osd  PARAMETERS: osd\_id – the id of the osd  osd\_uuid – the user id? Of the osd  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the command arguments, and create a new Command to run the systemctl command. If the command is successful, return Ok(()), else it error’d |
| systemctl\_stop(osd\_id: u64, simulate: bool) -> BynarResult<()>  DESCRIPTION: run the systemctl disable command on an osd  PARAMETERS: osd\_id – the id of the osd  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the command arguments, and create a new Command to run the systemctl command. If the command is successful, return Ok(()), else it error’d |
| setup\_osd\_init(osd\_id: u64, osd\_uuid: &uuid::Uuid, simulate: bool) -> BynarResult<()>  DESCRIPTION: initialize (start) the osd after having prepared the osd (it should be down and in) and be up and in once the function is run successfully  PARAMETERS: osd\_id – the id of the osd  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: check which daemon is running on the system to use the correct command. If the daemon is Systemd, use the systemctl command to start the osd and check the output. If the daemon is Upstart, then use the start command to start the osd and check the output. If the function is successful Ok(()) is returned, otherwise it error’d out somewhere. |
| settle\_udev() -> BynarResult<()>  DESCRIPTION: wait for udevd to create device nodes for all detected devices  PARAMETERS: NONE  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: run the udevadm command with the argument “settle”. If successful, return Ok(()), else error. |
| ceph\_mkfs(osd\_id: u64, journal: Option<&JournalDevice>, bluestore: bool, monmap: Option<&Path>, osd\_data: Option<&Path>, osd\_uuid: Option<&uuid::Uuid>, user\_id: Option<&str>, group\_id: Option<&str>, simulate: bool) -> BynarResult<()>  DESCRIPTION: Run ceph-osd –mkfs and return the osd UUID  PARAMETERS: osd\_id – the id of the osd  journal – a JournalDevice if it is used by the OSD  bluestore – whether the OSD is a bluestore or filestore  monmap – optional path to the monmap  osd\_data – optional path to the osd data directory  osd\_uuid – optional user id of the osd?  user\_id – the optional user id permissions of the OSD  group\_id - the optional group id permissions of the OSD  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create the arguments to the ceph-osd –mkfs command. Add more arguments depending on the contents of the input, and run the ceph-osd command. If successful, return Ok(()), else it error’d |
| ceph\_bluestore\_tool(device: &Path, mount\_path: &Path, simulate: bool) -> BynarResult<()>  DESCRIPTION: Prime a bluestore osd, generating the content for an osd data directory that can start up a bluestore osd  PARAMETERS: device – the path to the osd device  mount\_path – the mount path of the osd  simulate – if passed, skip the execution of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create the arguments of the ceph-bluestore-tool command. Run the command. If the command is successful, return Ok(()), else it error’d |
| create\_journal(name: &str, size: u64, path: &Path) -> BynarResult<(u32, uuid::Uuid)>  DESCRIPTION: create a new ceph journal on a given device with the name and size in bytes  PARAMETERS: name – the name pf the ceph journal  size – the size of the journal in bytes  path – the path of the journal  RETURNS: Ok((partition id, partition group id)) on success, Error otherwise  IMPLEMENTATION: open the GPT (GUID partition table) in writable mode and inspect the path in the GPT. Add a new partition to the GPT of type CEPH JOURNAL and write it to the disk. update the partition cache and read it back into the GPT, then check if the partition was added to the GPT. If everything runs successfully return Ok(partition id, partition guid), otherwise it error’d |
| enough\_free\_space(device: &Path, size: u64) -> BynarResult<bool>  DESCRIPTION: Check if there is enough free space on the disk to fit a partition size request  PARAMETERS: device – the path to the osd device  size – the size of the partition request  RETURNS: Ok(is there enough space?) on success, Error otherwise  IMPLEMENTATION: open the GPT and check the device path. Find the free sectors on the dish, and for each pair of free sectors, check if there is enough space (if the length of the free sector > the input size). If the function is successful, return Ok(true) if there is a sector with enough space, Ok(False) if there is no sector with enough space, otherwise there was an error |
| evaluate\_journal(journal: &JournalDevice, journal\_size: u64) -> BynarResult<JournalDevice>  DESCRIPTION: Attempt to discover if there is a device in the journal, create journal partition if needed, and return a path to use for the journal  PARAMETERS: journal – the journal to evaluate  journal\_size – the size of the journal partition  RETURNS: Ok(path to journal) on success, Error otherwise  IMPLEMENTATION: If the journal has a partition id, and a device, check if the partition exists and whether its in use by another osd. We can check using the GPT table, looping over the partitions to find the requested partition id, and check all the other osd’s for this partition id. If it is in use or there is no journal partition, create a new partition for the journal and update the number of partitions. If successful, return Ok(JournalDevice) with the updated partition values, otherwise it error’d somwhere. |
| remove\_unused\_journals(journals: &[JournalDevice]) -> BynarResult<()>  DESCRIPTION: Checks all osd drives on the system against the journals and delets all unused partitions. Note: unused  PARAMETERS: journals – the list of journals  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: For each journal in the list, open the GPT and check the disk at the journal device. get all of the partitions on the disk, and check if each partition is in use. If not, mark it as unused and save and update the partitions, and write all changes to the disk. If successful, return Ok(()), otherwise it error’d somewhere |
| is\_filestore(dev\_path: &Path) -> BynarResult<bool>  DESCRIPTION: Checks if the specified OSD is a filestore  PARAMETERS: dev\_path – the device path of the osd  RETURNS: Ok(is a filestore?) on success, Error otherwise  IMPLEMENTATION: Get the mount point from the device path. If there isn’t a mountpoint, create a temporary osd mount point and mount the device. Add type to the path and check if the path exists. If so, check if the contents of the file contain “filestore”. If the function is successful and “filestore” type is found, return Ok(true), if successful and “filestore” is NOT found, return Ok(false), else it error’d |
| update\_partition\_cache(device: &Path) -> BynarResult<()>  DESCRIPTION: Linux specific ioctl to update the partition table cache  PARAMETERS: device – the device path  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Open the device and run blkrrpart. If successful return Ok(()), else it error’d |

# Database Schema

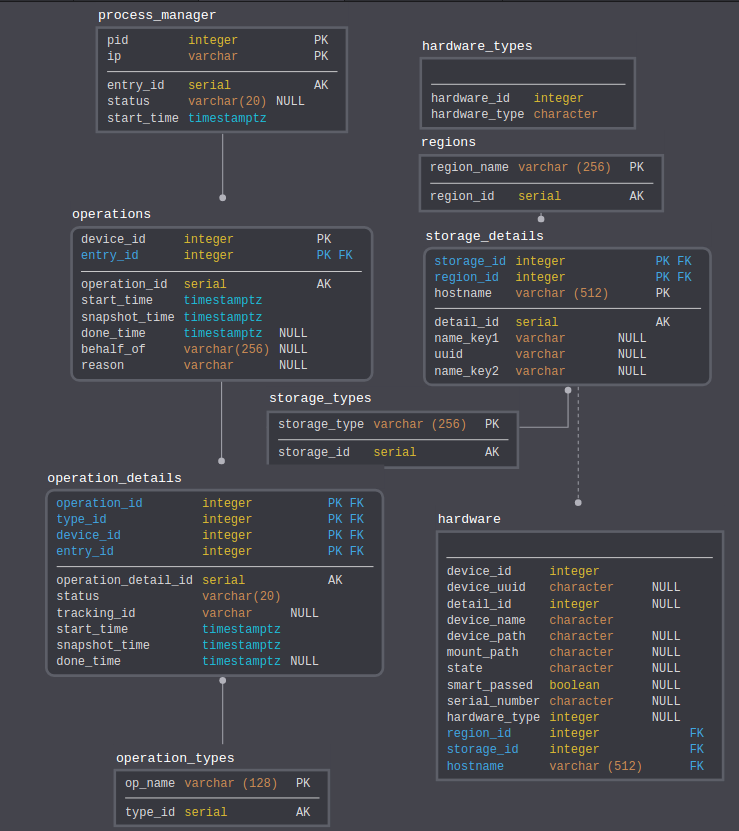
## Introduction

Bynar should have a database to log changes, errors, and other noteworthy messages. Currently Bynar only supports Postgres

## Postgres

In the dbschema folder, there is a bynar\_stats.sql file. You will need to import this into your Postgres Bynar Database. To import, you can run \i <path to file> from inside the psql prompt, or copy paste.

### Schema



# Database Logging

## Introduction

Most database logging functions are in the in\_progress.rs file. This file holds functions that log changes and other important messages to a database. Currently it only handles Postgres database integration.

## Logging

### Enums

#### OperationType

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| DiskAdd | Add a disk |
| DiskReplace | Replace a disk |
| DiskRemove | Remove a Disk |
| WaitingForReplacement | Waiting for a Replacement Disk |
| Evaluation | ???? Evaluate a disk? |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut Formatter | Converts an OperationType to a String for printing | fmt::Result |

###### Debug

#### OperationStatus

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Pending | Operation waiting to start |
| InProgress | Operation is running |
| Complete | Operation has finished |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut Formatter | Converts an OperationStatus to a String for printing | fmt::Result |

###### Debug

### Structs

#### DiskRepairTicket

A Disk Repair Ticket, a table entry?

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| ticket\_id | String | Id number of the ticket |
| device\_name | String | Name of the device to repair |
| device\_path | String | Path to the device to repair |

##### Trait Implementation

###### Debug

#### DiskPendingTicket

Table entry???

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| ticket\_id | String | Id number of the ticket |
| device\_name | String | Name of the device ???? Pending? |
| device\_path | String | Path to the device ??? Pending? |
| device\_id | i32 | ID number of the device? Pending? |

##### Implementation

|  |
| --- |
| Function Definition |
| new(ticket\_id: String, device\_name: String, device\_path: String, device\_id: i32) -> DiskPendingTicket  DESCRIPTION: create a new DiskPendingTicket  PARAMETERS: ticket\_id – the id number of the ticket  device\_name – the name of the pending? device  device\_path – the path of the pending? Device  device\_id – the id of the pending? device  RETURNS: DiskPendingTicket  IMPLEMENTATION: create a new DiskPendingTicket with the input parameters |

##### Trait Implementation

###### Debug

#### HostDetailsMapping

Table entry?

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| entry\_id | u32 | Entry number? |
| region\_id | u32 | Region number |
| storage\_detail\_id | u32 | Storage detail relation number??? |

##### Implementation

|  |
| --- |
| Function Definition |
| new(entry\_id: u32, region\_id: u32, storage\_detail\_id: u32) -> HostDetailsMapping  DESCRIPTION: Create a new HostDetailsMapping table entry  PARAMETERS: entry\_id – the table entry number  region\_id – the region id number  storage\_detail\_id – the reference to the storage\_detail entry ID  RETURNS: HostDetailsMapping  IMPLEMENTATION: create a new HostDetailsMapping with the input parameters |

##### Trait Implementation

###### Debug

#### OperationInfo

An entry for the Operations Table

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| operation\_id | Option<u32> | The operation id |
| entry\_id | u32 | The table entry id |
| device\_id | u32 | The device id |
| behalf\_of | Option<String> | On behalf of what user |
| reason | Option<String> | The reason for the operation |
| start\_time | DateTime<Utc> | The start time of the operation |
| snapshot\_time | DateTime<Utc> | The time when taking a snapshot of the operation |
| done\_time | Option<DateTime<Utc>> | When the operation was finished |

##### Implementation

|  |
| --- |
| Function Implementation |
| new(entry\_id: u32, device\_id: u32) -> OperationInfo  DESCRIPTION: Create a new OperationInfo with an entry\_id and device\_id  PARAMETERS: entry\_id – the table entry id  device\_id – the id number of the device being operated on  RETURNS: OperationInfo  IMPLEMENTATION: Create a new OperationInfo filled with the input parameters with all optional fields set to None and the start and snapshot times defaulted to the current timestamp. |
| set\_operation\_id(&mut self, op\_id: u32)  DESCRIPTION: set the operation id number  PARAMETERS: op\_id – the operation id number  RETURNS: the OperationInfo with its operation id set to the input id number IMPLEMENTATION: set the value of the oepration\_id to the input id |
| set\_done\_time(&mut self, done\_time: DateTime<Utc>)  DESCRIPTION: set the completion time  PARAMETERS: done\_time - the timestamp of when the operation finished  RETURNS: the OperationInfo with its done\_time set to the input completion time  IMPLEMENTATION: set the value of done\_time to the input done\_time |
| set\_snapshot\_time(&mut self, snapshot\_time: DateTime<Utc>)  DESCRIPTION: set the snapshot time  PARAMETERS: snapshot\_time – the time of the latest snapshot of the operation  RETURNS: the OperationInfo with its snapshot\_time set to the latest snapshot time IMPLEMENTATION: set the value of snapshot\_time to the input snapshot time |

##### Trait Implementation

###### Debug

#### OperationDetail

An entry for the operation\_details table

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| op\_detail\_id | Option<u32> | Operation detail entry id number |
| operation\_id | u32 | Link to the operation id number |
| op\_type | OperationType | The operation type |
| status | OperationStatus | Current status of the operation |
| tracking\_id | Option<String> | The tracking id number of the operation |
| start\_time | DateTime<Utc> | The start time of the operation |
| snapshot\_time | DateTime<Utc> | The last snapshot time of the operation |
| done\_time | Option<DateTime<Utc>> | The time when the operation was completed |

##### Implementation

|  |
| --- |
| Function Definition |
| new(operation\_id: u32, op\_type: OperationType) -> OperationDetail  DESCRIPTION: Create a new OperationDetail with optional fields set to None and start and snapshot time attributes set to the current timestamp  PARAMETERS: operation\_id – the reference to the operation table  op\_type – the operation type  RETURNS: OperationDetail  IMPLEMENTATION: create a new OperationDetail and set all optional values to None, set the operation\_id and op\_type to the input values, and default start and snapshot times to the current timestamp |
| set\_operation\_detail\_id(&mut self, op\_detail\_id: u32)  DESCRIPTION: set the operation detail id PARAMETERS: op\_detail\_id – the entry number RETURNS: OperationDetail with the operation\_detail\_id set to the input  IMPLEMENTATION: set the value of operation\_detail\_id to the input operation detail id |
| set\_tracking\_id(&mut self, tracking\_id: String)  DESCRIPTION: set the tracking id PARAMETERS: tracking\_id – the tracking id RETURNS: OperationDetail with the tracking\_id set to the input value IMPLEMENTATION: set the value of tracking\_id to the input tracking id |
| set\_done\_time(&mut self, done\_time: DateTime<Utc>)  DESCRIPTION: set the done time PARAMETERS: done\_time – the time of the operation completion RETURNS: OperationDetail with the done\_time set to the input completion time IMPLEMENTATION: set the value of done\_time to the input completion time |
| set\_operation\_status(&mut self, status: OperationStatus)  DESCRIPTION: set the operation status PARAMETERS: status – the current status of the operation RETURNS: OperationDetail with the status set to the input status IMPLEMENTATION: set the value of status to the input status value |

##### Trait Implementation

###### Debug

### Interface and Helper Functions

|  |
| --- |
| Helper Function Definition |
| create\_bd\_connection\_pool(db\_config: &DBConfig) -> BynarResult<Pool<ConnectionManager>>  DESCRIPTION: Reads the config file to establish a pool of database connections  PARAMETERS: db\_config – the database configuration  RETURNS: Ok(connectionManager pool) on success, Error otherwise  IMPLEMENTATION: Convert the password in the Config to a str, since that’s what Postgres expects. Set the connection parameters, and create a ConnectionManager with the parameters. Build a pool of 10 threads to the Postgres database. If successful, return Ok(Pool of connections to Postgres) otherwise error out |
| get\_connection\_from\_pool(pool: &Pool<ConnectionManager>) -> BynarResult<PooledConnection<ConnectionManager>>  DESCRIPTION: return one connection from the pool  PARAMETERS: pool – the pool of connections to the database  RETURNS: Ok(A single pooled connection) on success, Error otherwise  IMPLEMENTATION: run pool.get to get a free connection thread. If successful, return Ok(single connection to the database), otherwise error out |
| update\_storage\_info(s\_info: &MyHost, pool: &Pool<ConnectionManager>) -> BynarResult<HostDetailsMapping>  DESCRIPTION: update the storage info in the database, should be called when the Bynar daemon starts and checks if all steps in the function are successful  PARAMETERS: s\_info - the current host information of the program  pool – the pool of connections to the database  RETURNS: Ok(host details mapping) on success, Error otherwise  IMPLEMENTATION: get a single connection to the database. extract the ip address from the host information. start a new Postgres transaction to update the storage information in the database. Register the ip to the process manager, update the region info, and update the storage details. commit the Postgres SQL requests and create a new HostDetailsMapping with the returned values from the transaction calls. Finish the transaction, and if successful, return Ok(host details mapping), otherwise error out. |
| register\_to\_process\_manager(conn: &Transaction<’\_>, ip: &str) -> BynarResult<u32>  DESCRIPTION: stores the pid, ip of the system on which bynar is running to the database  PARAMETERS: conn – the transaction connection to the database  ip – the ip to store  RETURNS: the entry id of the transaction IMPLEMENTATION: get the process id. Create the statement with the pid and ip. Query the database with the statement. If there is a response, get the entry id and update the process\_manager table with the idle status. If there is response, insert into the process\_manager table the pid, ip, and the idle status, getting back the entry id. If successful, return Ok(entry\_id), otherwise error out. |
| deregister\_from\_process\_manager() -> BynarResult<()>  DESCRIPTION: When implemented, should de-register the process from the database when the daemon exists?? Exits???  PARAMETERS: N/A RETURNS: N/A IMPLEMENTATION: N/A |
| update\_region(conn: &Transaction<’\_>, region: &str) -> BynarResult<u32>  DESCRIPTION: checks for the region in the database, inserts if it does not exist and returns the region\_id PARAMETERS: conn – the connection to the database for transactions RETURNS: Ok(region\_id) on success, else Error IMPLEMENTATION: Query the database for the region name. If it exists, return Ok(region\_id), if it doesn’t, insert the region into the database and get the region\_id. If successful, return Ok(region\_id), else error out |
| update\_storage\_details(conn: &Transaction<’\_>, s\_info: &MyHost, region\_id: u32) -> BynarResult<u32>  DESCRIPTION: update the storage details in the database and get the storage\_detail\_id PARAMETERS: conn – the connection to the database for transaction  s\_info – the storage host information  region\_id – the region id number in the database  RETURNS: Ok(storage\_detail\_id) if successful, else Error IMPLEMENTATION: query if the database has the input storagetype. If so, query if the specific details are already in the database. If not, insert the array\_name and pool\_name into the database. If successful, return Ok(storage\_detail\_id), else error out |
| add\_disk\_detail(pool: &Pool<ConnectionManager>, disk\_info: &mut BlockDevice) -> BynarResult<()>  DESCRIPTION: Inserts disk information record into bynar.hardware and adds the device\_database\_id to the struct PARAMETERS: pool – the pool of connections to the database  disk\_info – the BlockDevice info to query about and fill in  RETURNS: Ok(()) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the disk details. If a record of the disk doesn’t exist, insert the disk\_info information into the database and get the device\_database\_id number. If the device exists in the database, check if it matches the input struct and get the device\_database\_id. If successful, return Ok(device\_database\_id), else error out |
| add\_or\_update\_operation(pool: &Pool<ConnectionManager>, op\_info: &mut OperationInfo) -> BynarResult<()>  DESCRIPTION: inserts or updates the operation record. If a successful insert, the provided input op\_info is modified. Errors if insert fails PARAMETERS: pool – the pool of connections to the database  op\_info – the operation info  RETURNS: Ok(()) on success, else Error IMPLEMENTATION: get a single connection to the database. If there is no operation\_id, validate the input record. Insert a new record. If there is an operation id, update the operation record. Update the op\_info with the operation id. If successful return Ok(()), else error out |
| add\_or\_update\_operation\_detail(pool: &Pool<ConnectionManager>, operation\_detail: &mut OperationDetail) -> BynarResult<()>  DESCRIPTION: inserts or updates the operation details record. If a successful insert, the provided input operation\_detail is modified. Errors if insert fails PARAMETERS: pool – the pool of connections to the database  operation\_detail – the operation details info  RETURNS: Ok(()) if success, else Error  IMPLEMENTATION: get a single connection to the database. If there is no operation detail id, insert a new detail record. If there is an operation detail id, update the existing record. Update the operation\_detail with the operation\_detail\_id. If successful return Ok(()), else error out |
| save\_state(pool: &Pool<ConnectionManager>, device\_detail: &BlockDevice, state: State) -> BynarResult<()>  DESCRIPTION: save the state machine information for the device in the database PARAMETERS: pool – the pool of connections to the database  device\_detail – the block device info  state – the state of the state machine  RETURNS: Ok(()) on success, else Error IMPLEMENTATION: get a single connection to the database. Check if the device is in the database (which it should be). Update the state, start a transaction that rolls back if necessary to update the database. If successful, return Ok(()), else error out. |
| save\_smart\_result(pool: &Pool<ConnectionManager>, device\_detail: &BlockDevice, smart\_passed: bool) -> BynarResult<()>  DESCRIPTION: save the result of the smart check of the device in the database PARAMETERS: pool – the pool of connections to the database  device\_detail – the block device info  smart\_passed – whether the smart check passed or not  RETURNS: Ok(()) on success, else Error IMPLEMENTATION: get a single connection to the database. Check if the device is in the database(which it should be). Update smart\_passed. start a transaction that rolls back if necessary to update the database. If successful, return Ok(()), else error out. |
| get\_devices\_from\_db(pool: &Pool<ConnectionManager>, storage\_detail\_id: u32) -> BynarResult<Vec<u32, String, Pathbuf>>  DESCRIPTION: get the currently known disks from the database PARAMETERS: pool – the pool of connections to the database  storage\_detail\_id – the entry number of the storage detail table  RETURNS: Ok(device id, device name, device path) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the device id, name and path. If successful, return Ok(dev\_id, dev\_name, dev\_path), else error out |
| get\_state(pool: &Pool<ConnectionManager>, device\_detail: u32) -> BynarResult<State>  DESCRIPTION: get the state information from the database PARAMETERS: pool – the pool of connections to the database  device\_detail – the entry number of the device in the hardware table  RETURNS: Ok(state) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the state of the device. If successful, return Ok(state), else error out |
| get\_smart\_result(pool: &Pool<ConnectionManager>, device\_detail: u32) -> BynarResult<bool>  DESCRIPTION: get the currently known disks from the database PARAMETERS: pool – the pool of connections to the database  device\_detail – the entry number of the device in the hardware table  RETURNS: Ok(bool) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for whether the device passed the smart checks or not. If successful, return Ok(passed?), else error out |
| row\_to\_ticket(row: &Row<’\_>) -> DiskRepairTicket  DESCRIPTION: convert a row from a query to a DiskRepairTicket PARAMETERS: row – the query result to convert  RETURNS: DiskRepairTicket IMPLEMENTATION: Create a DiskRepairTicket with the values from the row |
| get\_outstanding\_repair\_tickets(pool: &Pool<ConnectionManager>, storage\_detail\_id: u32) -> BynarResult<Vec<DiskRepairTicket>>  DESCRIPTION: get a list of ticket IDs (JIRA/other ids) that belong to “me” that are pending, in progress, or op\_type=WaitForReplacement PARAMETERS: pool – the pool of connections to the database  storage\_detail\_id – the entry number of the storage detail in the tables  RETURNS: Ok(list of disk repair tickets) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for a list of Operations that are InProgress, Pending, WaitingForReplacement, and Good with the specified storage\_detail\_id. Convert the rows returned into DiskRepairTickets and, if sucessful, return Ok(List of disk repair tickets), else error out |
| resolve\_ticket\_in\_db(pool: &Pool<ConnectionManager>, ticket\_id: &str) -> BynarResult<()>  DESCRIPTION: set the status as Complete for the record with the given ticket\_id. Note: this is equivalent to calling the add\_or\_update\_operation\_detaiL() with the appropriate fields set PARAMETERS: pool – the pool of connections to the database  ticket\_id – the ticket id in the support ticket system  RETURNS: Ok(()) on success, else Error IMPLEMENTATION: get a single connection to the database. Update the operation\_details as OperationStatus::Complete where the ticket\_id matches. If successful, return Ok(()), else error out |
| is\_hardware\_waiting\_repair(pool: &Pool<ConnectionManager>, storage\_detail\_id: u32, device\_name: &str, serial\_number: Option<&str>) -> BynarResult<bool>  DESCRIPTION: check if the hardware/device is currently waiting for repair PARAMETERS: pool – the pool of connections to the database  storage\_detail\_id – the entry number of the storage detail  device\_name – the name of the device to check  serial\_number – the serial number of the device to check  RETURNS: Ok(bool) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the device’s Operation/Storage details. check if the OperationType is WaitingForReplacement. If successful, return Ok(true) if the device is waiting for repair, Ok(false) if the device is not waiting for repairs, or error out |
| get\_region\_id(pool: &Pool<ConnectionManager, region\_name: &str) -> BynarResult<Option<u32>>  DESCRIPTION: get the region id based on the region name PARAMETERS: pool – the pool of connections to the database  region\_name – the name of the region to get the database id value of  RETURNS: Ok(id number if exists) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the region name. If successful, return Ok(Some(region\_id)) if the region name is in the database, Ok(None) if it is not in the database, else error out |
| get\_storage\_id(pool: &Pool<ConnectionManager>, storage\_type: &str) -> BynarResult<Option<u32>>  DESCRIPTION: get the storage id based on the storage type PARAMETERS: pool – the pool of connections to the database  storage\_type – the storage type to get the database id value of  RETURNS: Ok(id number if exists) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the storage type. If successful return Ok(Some(storage\_id)) if the storage type is in the database, Ok(None) if it is not in the database, else error out |
| get\_storage\_detail\_id(pool: &Pool<ConnectionManager>, storage\_id: u32, region\_id: u32, host\_name: &str) -> BynarResult<Option<u32>>  DESCRIPTION: get the storage detail id based on the storage id, region id and hostname PARAMETERS: pool – the pool of connections to the database  storage\_id – the id of the storage type information  region\_id – the id of the region name  host\_name - the host name  RETURNS: Ok(storage detail id if exist) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the storage\_detail\_id associated with the input values. If successful, return Ok(Some(storage\_detail\_id)), Ok(None) if it does not exist, or error out |
| get\_all\_pending\_tickets(pool: &Pool<ConnectionManager>) -> BynarResult<Vec<DiskPendingTicket>>  DESCRIPTION: get a list of ticket IDs (JIRA/other) that belong to ALL servers that are in pending state and outstanding tickets PARAMETERS: pool – the pool of connections to the database  RETURNS: Ok(list of pending/outstanding disks) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for ALL tickets with the WaitingForReplacement, Pending, InProgress, and GoodState, convert them to DiskPendingTickets. If successful, return Ok(list of diskpending tickets) else error out |
| get\_host\_name(pool: &Pool<ConnectionManager>, device\_id: i32) -> BynarResult<Option<String>>  DESCRIPTION: get the host name based on the device id PARAMETERS: pool – the pool of connections to the database  device\_id – the id number of the device in the database  RETURNS: Ok(hostname if it exists) on success, else Error IMPLEMENTATION: get a single connection to the database. Query the database for the host name associated with the device id. If successful, return Ok(Some(host\_name)) or Ok(None) if the host name does not exist for the device id. Otherwise, error out |

# Helper Functions

## Introduction

There are a couple of functions and types that are needed across most of the Bynar program. These include the Error Type, host information, and various connection and requests.

## Error Module

The error module provides the error type for the Bynar program. Various error types are imported and generalized as a BynarResult Error

### Type

#### BynarResult<T>

This is the generic Bynar Errortype, a Result type of type <T, BynarError>

### Enums

#### PwdBError

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| PwdError(PwdError) | An error from the pwd library |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter | Given a PwBError, display the error msg | fmt::Result |

###### Debug

#### BynarError

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| BlkidError(BlkidError) | A blkid command error |
| BlockUtilsError(BlockUtilsError) | A block\_utils library error |
| Error(String) | A generic String error |
| GojiError(GojiError) | A Gojira library error |
| HardwareError(HardwareError) | A hardware error |
| IoError(IOError) | A std::io error |
| LvmError(LvmError) | An lvm error |
| NixError(NixError) | A nix library error |
| ParseIntError(ParseIntError) | A parseint error (integer parser) |
| PostgresError(PostgresError) | A postgres command error |
| ProtobufError(ProtobufError) | A protobuf serializer error |
| PwdError(PwdBError) | A pwd error |
| R2d2Error(R2d2Error) | An R2d2 error |
| RadosError(RadosError) | A Ceph rados error |
| ReqwestError(ReqwestError) | A reqwest library error |
| SerdeJsonError(SerdeJsonError) | A serde json library error |
| SlackError(SlackError) | A Slack error |
| UuidError(UuidError) | A uuid error |
| VaultError(VaultError) | A vault error |
| ZmqError(ZmqError) | A zmq library error |

##### Implementation

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| new | err: String | Create a new BynarError with a String message | BynarError |
| to\_string | self | Convert a BynarError into a String representation | String |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter | Given a Bynar, display the error msg | fmt::Result |

###### From<PwdError>

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| from | err: PwdError | Given a PwdError, create a BynarError | BynarError |

###### From<String>

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| from | err: String | Given a String, create a BynarError | BynarError |

###### From<’a str>

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| from | err: &str | Given a &str, create a BynarError | BynarError |

###### Debug, de::Error

### Structs

#### HardwareError

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| error | String | The error |
| name | String | The name of the error |
| location | Option<String> | The location? Of the error |
| location\_format | Option<String> | Uh, the format?????? |
| serial\_number | Option<String> | Serial number of whatever is having issues? |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter | Given a HardwareError, display the error msg | fmt::Result |

###### Debug

## Host Information

Gather information about the current host

### Enums

#### StorageTypeEnum

The type of distributed storage

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Ceph | Ceph storage type |
| Scaleio | Scaleio storage type |
| Gluster | Gluster storage type |
| Hitachi | Hitachi storage type |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter | Given a StorageTypeEnum, display the storage type | fmt::Result |

###### Debug

### Structs

#### Host

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| hostname | String | The host name |
| ip | IpAddr | The ip address |
| region | String | The region |
| kernel | String | The kernel type |
| server\_type | String | The server type |
| serial\_number | String | The serial number |
| machine\_architecture | String | The machine architecture |
| scsi\_info | Vec<block\_utils::ScsiInfo> | The scsi information |
| storage\_type | StorageTypeEnum | The storage type |
| array\_name | Option<String> | The array name |
| pool\_name | Option<String> | The pool name |

##### Implementation

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| new | N/A | Create a new Host | BynarResult<Host> |

##### Trait Implementations

###### Debug

### Helper Functions

|  |
| --- |
| Helper Function Definition |
| get\_default\_iface() -> BynarResult<String>  DESCRIPTION: get the default interface  PARAMETERS: None  RETURNS: Ok(default interface) on success, Error otherwise  IMPLEMENTATION: open the /proc/net/route file. For each line, try to find the default gateway “00000000” and return the interface. If successfule, return Ok(default interface) else error |
| get\_ip() -> BynarResult<IpAddr>  DESCRIPTION: Find the IP address on the default interface  PARAMETERS: None  RETURNS: Ok(ip address) on success, Error otherwise  IMPLEMENTATION: get all interfaces as well as the default interface. filter all interfaces to get the default. In the interface, loop through the ip addresses until an ipv4 address is found and return it. If successful, return the ipv4 address, else error. |
| get\_region\_from\_hostname(hostname: &str) -> BynarResult<String>  DESCRIPTION: Get the region from the hostname  PARAMETERS: hostname – the hostname  RETURNS: Ok(region) on success, Error otherwise  IMPLEMENTATION: Production hostnames are usually in the format name-regionpart1-regionpart2-\*, so split the hostname by ‘-’, skip the first sub string and combine the region parts. If successful, either return Ok(region) if totally successful, Ok(“test-region”) if the hostname is not regular, or error if it fails. |
| get\_storage\_type() -> BynarResult<StorageTypeEnum>  DESCRIPTION: get the storage type used on this system  PARAMETERS: None  RETURNS: Ok(storage type) on success, Error otherwise  IMPLEMENTATION: for now, it just returns Ceph.... |
| server\_type() -> BynarResult<String>  DESCRIPTION: Find the server type  PARAMETERS: None  RETURNS: Ok(server type) on success, Error otherwise  IMPLEMENTATION: Go to /sys/class/dmi/id/product\_name and read the file. If successful return the file contents as Ok(server type), else error |
| server\_serial() -> BynarResult<String>  DESCRIPTION: get the server serial number  PARAMETERS: None  RETURNS: Ok(server serial number) on success, Error otherwise  IMPLEMENTATION: for now, it just tries the easy way, which is reading the /sys/class/dmi/id/product\_serial file for the number. If successful returns Ok(server serial number), otherwise error |

## Helper Module

Public functions and structures that can be used outside of the library.

### Structs

#### ConfigSettings

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| hostname | String | The host name |
| ip | IpAddr | The ip address |
| region | String | The region |
| kernel | String | The kernel type |
| server\_type | String | The server type |
| serial\_number | String | The serial number |
| machine\_architecture | String | The machine architecture |
| scsi\_info | Vec<block\_utils::ScsiInfo> | The scsi information |
| storage\_type | StorageTypeEnum | The storage type |
| array\_name | Option<String> | The array name |
| pool\_name | Option<String> | The pool name |

##### Trait Implementations

###### Clone, Debug, Deserialize

#### DBConfig

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| username | String | Database username |
| password | Option<String> | Database password |
| port | u16 | Port to connect to database with |
| endpoint | String | Database endpoint |
| dbname | String | Database name |

##### Trait Implementations

###### Clone, Debug, Deserialize

### Helper Functions

|  |
| --- |
| Helper Function Definition |
| load\_config<T>(config\_dir: &Path, name: &str) -> BynarResult<T>  DESCRIPTION: load a config file that is deserializable  PARAMETERS: config\_dir – the directory of the config file  name – name of the file to deserialize  RETURNS: Ok(deserialized structure) on success, Error otherwise  IMPLEMENTATION: create the path to the file, and check if it exists. Read the file and deserialize it into the struct. If successfule, return Ok(deserialized struct) otherwise error out |
| connect(host: &str, port: &str, server\_publickey: &str) -> BynarResult<Socket>  DESCRIPTION: connect to the input host:port ip and securing with the server public key  PARAMETERS: host – the host ip address  port – the port to connect to  server\_publickey – the public key of the server used to secure the socket  RETURNS: Ok(connected socket) on success, Error otherwise  IMPLEMENTATION: create a new zmq REQ socket. create a curveKeyPair to secure the socket. set the keys in the socket and connect using tcp to the host:port ip address. If successful, return Ok(REQ socket), otherwise error out. |
| get\_vault\_token(endpoint: &str, token: &str, hostname: &str) -> BynarResult<String>  DESCRIPTION: get the vault secret from the Hashicorp Vault  PARAMETERS: endpoint – the hashicorp endpoint  token – token to access the vault with  hostname – name of the host to get the secret of  RETURNS: Ok(vault secret??) on success, Error otherwise  IMPLEMENTATION: Connect to the Vault with VaultClient, and get the secret. If successful return Ok(Vault secret) else error |
| add\_disk\_request(s: &mut Socket, path: &Path, id: Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: send a request to add a disk to a cluster  PARAMETERS: s – the socket to send and receive messages from  path – the path of the disk to add to the cluster  id – the osd id of the disk to add  simulate – if passed, skip evaluation  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the Operation message. Convert the message into bytes and send it from the socket and wait for a response. Parse the Operation result for OK or ERROR. If successful, return Ok(()), otherwise something failed. |
| list\_disks\_request(s: &mut Socket) -> BynarResult<Vec<Disk>>  DESCRIPTION: send a request to get a list of disks from a cluster  PARAMETERS: s – the socket to send and receive messages from  RETURNS: Ok(disk list) on success, Error otherwise  IMPLEMENTATION: Create the Operation message. Convert the message into bytes and send it from the socket and wait for a response. Parse the Operation result for the list of disks. If successful, return Ok(disk list), otherwise something failed. |
| safe\_to\_remove\_request(s: &mut Socket, path: &Path) -> BynarResult<bool>  DESCRIPTION: send a request to a cluster to ask if a disk is safe to remove  PARAMETERS: s – the socket to send messages from  path – the path of the disk to check if removable  RETURNS: Ok(is safe to remove?) on success, Error otherwise  IMPLEMENTATION: Create the Operation message. Convert the message into bytes and send it from the socket and wait for a response. Parse the Operation result for whether the disk is safe to remove. If successful, return Ok(true) if safe to remove, Ok(false) if the disk is not safe to remove, otherwise something failed so error out. |
| remove\_disk\_request(s: &mut Socket, path: &Path, id: Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: send a request to remove a disk from a cluster  PARAMETERS: s – the socket to send messages from  path – the path of the disk to add to the cluster  id – the osd id of the disk to add  simulate – if passed, skip evaluation  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the Operation message. Convert the message into bytes and send it from the socket and wait for a response. Parse the Operation result for OK or ERROR. If successful, return Ok(()), otherwise something failed. |
| get\_jira\_tickets(s: &mut Socket) -> BynarResult<()>  DESCRIPTION: send a request to get Jira tickets  PARAMETERS: s – the socket to send messages from  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create the Operation message. Convert the message into bytes and send it from the socket and wait for a response. Parse the Operation result for OK or ERROR. If Ok get the tickets and print the ticket information. If successful, return Ok(()), otherwise something failed. |

# Client

## Introduction

This is a client interface built as a separate binary. It enables a user to make manual calls to the disk\_manager and Bynar.

### Client Interface

##### Implementation

|  |
| --- |
| Function Definition |
| add\_disk(s: &mut Socket, path: &Path, id: Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: Send a message to add a disk to the cluster  PARAMETERS: s – the socket to send and receive messages  path – the path of the disk to add  id – the optional osd id of the disk to add  simulate – if passed, skip evaluation of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: run the Helper library add\_disk\_request function. If successful return Ok(()), else error |
| list\_disks(s: &mut Socket) -> BynarResult<Vec<Disk>>  DESCRIPTION: list the disks in a cluster and print them to the console  PARAMETERS: s – the socket to send and receive messages  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Run the helper library list\_disks\_request and print the disks. If successful return Ok(()), else error |
| remove\_disk(s: &mut Socket, path: &Path, id: Option<u64>, simulate: bool) -> BynarResult<()>  DESCRIPTION: Send a message to remove a disk from the cluster  PARAMETERS: s – the socket to send and receive messages  path – the path of the disk to add  id – the optional osd id of the disk to add  simulate – if passed, skip evaluation of the function  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Run the helper library remove\_disk\_request. If successful return Ok(()), else error |
| handle\_add\_disk(s: &mut Socket, matches: &ArgMatches<’\_>)  DESCRIPTION: Wrapper for adding a disk, parses a command line input to add a disk  PARAMETERS: s – the socket to send and receive messages  matches – the argument inputs parsed from the command line  RETURNS: None  IMPLEMENTATION: get the arguments from the match input, and check their types. Run the add\_disk function on the inputs. If successful print a success message to the terminal, else print the failure message |
| handle\_list\_disks(s: &mut Socket)  DESCRIPTION: Wrapper for listing disks  PARAMETERS: s – the socket to send and receive messages  RETURNS: None  IMPLEMENTATION: list the disks using the list\_disks function and print the list if successful, otherwise print the error message |
| handle\_jira\_tickets(s: &mut Socket) -> BynarResult<()>  DESCRIPTION: Wrapper for getting and printing jira tickets  PARAMETERS: s – the socket to send and receive messages  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: use the helper library get\_jira\_tickets function. If successful, return Ok(()), otherwise error out |
| handle\_remove\_disk(s: &mut Socket, matches: &ArgMatches<’\_>)  DESCRIPTION: Wrapper for removing a disk, parses a command line input to remove a disk  PARAMETERS: s – the socket to send and receive messages  matches – the argument inputs parsed from the command line  RETURNS: None  IMPLEMENTATION: get the arguments from the match input, and check their types. Run the remove\_disk function on the inputs. If successful print a success message to the terminal, else print the failure message |
| get\_cli\_args(default\_server\_key: &str) -> ArgMatches<’\_>  DESCRIPTION: Create the command line arguments and parse them for proper input  PARAMETERS: default\_server\_key – the default value for the server key  RETURNS: An ArgMatches with the matched arguments to the cli inputs  IMPLEMENTATION: Create the App Ceph Disk Manager Client and add the RPC calls. Calls include host, port, server\_key, with subcommands add, list, get\_jira\_tickets, remove, and v for verbosity. Run get\_matches on the App object to get the command line arguments matching the CLI created in App. return the matches. |
| main()  DESCRIPTION: Run the Client  PARAMETERS: None  RETURNS: None  IMPLEMENTATION: create the server key. Get the CLI arguments. match the –v flags to level of verbosity. Get the host and port values for creating sockets. get the server publick key, and use the helper library to connect to the server. depending on the subcommand, either run handle\_add\_disk, handle\_list\_disks, handle\_remove\_disk, or handle\_jira\_tickets. |

# Support Tickets

## Introduction

Bynar won’t always be able to handle a disk problem. So, if for whatever reason Bynar cannot fix a disk or remove it immediately, it needs to be able to create a support ticket. Bynar also needs to be able to scan opened tickets to see if they’ve been resolved, so that Bynar can add the fixed disks back in. For now, the only ticket system supported is JIRA.

### JIRA Support

JIRA is a support ticketing system. We need to be able to create tickets and scan and list them as well.

##### Implementation

|  |
| --- |
| Function Definition |
| create\_support\_ticket(settings: &ConfigSettings, title: &str, description:&str) -> BynarResult<String>  DESCRIPTION: Create a new JIRA support ticket and return the ticket ID associated with it  PARAMETERS: settings – the configuration settings containing the information necessary to log into JIRA and use the API  title – the title of the new ticket  description – the description of the new ticket  RETURNS: Ok(ticket ID) on success, Error otherwise  IMPLEMENTATION: Create an Issue object, filling in the Assignee, component, description, priority, project, and summary attributes. Most of the these are given from the Config Settings. Open the proxy if there is one, and create a reqwest Client with a proxy. Create a Jira object (connect to Jira) and create a new Issue with the description in Jira. If successful, return Ok(created ticket ID), otherwise error out. |
| ticket\_resolved(settings: &ConfigSettings, issue\_id: &str) -> BynarResult<bool>  DESCRIPTION: check to see if a JIRA support ticket is marked as resolved  PARAMETERS: settings – config settings needed to connect to JIRA  issue\_id – the ID of the ticket to check  RETURNS: Ok(bool) on success, Error otherwise  IMPLEMENTATION: Connect to JIRA (with or without a proxy). Open the issue and check if the ticket is resolved. If successful, return Ok(true) if the issue is resolved, Ok(false) if the ticket is not yet resolved, else error out. |

# Disk Manager

## Introduction

This program handles the adding and removing of disks from a server

## Disk Manager

### Structs

#### DiskManagerConfig

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| backend | BackendType | The backend of the server |
| vault\_token | Option<String> | Hashicorp vault token |
| vault\_endpoint | Option<String> | Hashicorp vault endpoint |

##### Trait Implementations

###### Clone, Debug, Deserialize

### Functions

##### Implementation

|  |
| --- |
| Function Definition |
| convert\_media\_to\_disk\_type(m: &MediaType) -> DiskType  DESCRIPTION: convert a MediaType object into a DiskType object  PARAMETERS: m – the object to convert  RETURNS: converted DiskType object  IMPLEMENTATION: convert the MediaType to a DiskType and return it |
| setup\_curve(s: &mut Socket, config\_dir: &Path, vault: bool) -> BynarResult<()>  DESCRIPTION: Set up a curve encryption scheme on a socket  PARAMETERS: s – socket to set the curve encryption on  config\_dir – the config file directory  vault – whether using Hashicorp vault to set the encryption  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: set the socket with a curve server. Create a new CurveKeyPair. Get the hostname and get the key file from the config directory. If using the Hashicorp vault, connect to the vault and set a new secret with the generated keypair and set the socket with the keypair. Otherwise, if not using vault, just set the socket with the secret key and save the key to a file. If successful, return Ok(()), otherwise error out. |
| listen(backend\_type: BackendType, config\_dir: &Path, listen\_address: &str, vault: bool) -> BynarResult<()>  DESCRIPTION: listen for Operation messages from the listen address and run any successfully received messages.  PARAMETERS: backend\_type – the backend type of the server  config\_dir – the config file directory  listen\_address – the address of the client to listen to  vault – whether the program is using the hashicorp vault or not  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create a Responder Socket and set up the curve encryption on the socket. Bind the socket (listen) to the listen\_address using tcp. Loop, and while looping wait to for a message (in bytes). Parse an Operation message from the bytes and check the Op type. If an Add operation, check if it has the necessary fields and run add\_disk. If AddPartition, do nothing (for now). If List, run list\_disks. If Remove, check if the message has the necessary fields and run remove\_disk. If SafeToRemove, check if the message has the necessary fields and run safe\_to\_remove\_disk. If GetCreatedTickets, run get\_jira\_tickets. sleep for 10 milliseconds between each operation. If successful, it should loop continuously until the program is stopped (in which case return Ok(())), otherwise it should error out. |
| respond\_to\_client<T: protobuf::Message>(result: &T, s: &mut Socket) -> BynarResult<()>  DESCRIPTION: send a response back to the client with the result of an operation  PARAMETERS: result – the result of an operation  s – the socket to send and receive messages from  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: convert the message to bytes and send the bytes to the client. If successful, return Ok(()), else error out. |
| add\_disk(s: &mut Socket, d: &str, backend: &BackendType, id: Option<u64>, config\_dir: &Path) -> BynarResult<()>  DESCRIPTION: try to add a disk to the server and send the result back to the requestor  PARAMETERS: s – the socket to send and receive messages from  d– the disk device path to add  backend – the backend type  id – the osd id to use  config\_dir – the configuration file directory  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Load the backend (For now only Ceph). Run backend’s add\_disk function and check the result. Set the OpResult’s attributes depending on the result of the add\_disk, and send the OpResult to the client. If successful, return Ok(()), else error out. |
| get\_disks() -> BynarResult<Vec<Disk>>  DESCRIPTION: try to get a list of Disks from the server  PARAMETERS: None  RETURNS: Ok(list of Disks) on success, Error otherwise  IMPLEMENTATION: Search for all block devices. Gather the udev info of all found block devices. For each device, create a new Disk object, get its partition info (blank disks will fail), translate the block\_utils mediatype to the DiskType (from Protobuf), set the various values in the Disk, and add it to the list of Disks. If successful, return Ok(list of disks), otherwise error out. |
| get\_partition\_info(dev\_path: &Path) -> BynarResult<PartitionInfo>  DESCRIPTION: get partition info of a device/disk  PARAMETERS: dev\_path – the device/disk path  RETURNS: Ok(partition info) on success, Error otherwise  IMPLEMENTATION: create a new Partition Info. Read the header of the disk, then read the partitions using the header. Transform the returned partitions into protobuf PartitionInfo. If successful, return Ok(partition info), else error out. |
| list\_disks(s: &mut Socket) -> BynarResult<()>  DESCRIPTION: get a list of disks on the server and send it to the client  PARAMETERS: s – the socket to send and receive from  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the list of disks with get\_disks. Create the Disks message and set the disks. Write the Disks message to bytes and send to the client. |
| remove\_disk(s: &mut Socket, d: &str, backend: &BackendType, config\_dir: &Path) -> BynarResult<()>  DESCRIPTION: try to remove a disk from the server and send the result back to the client  PARAMETERS: s – the socket to send and receive messages from  d– the disk device path to remove  backend – the backend type  config\_dir – the configuration file directory  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Load the backend (For now only Ceph). Run backend’s remove\_disk function and check the result. Set the OpResult’s attributes depending on the result of the remove\_disk, and send the OpResult to the client. If successful, return Ok(()), else error out. |
| safe\_to\_remove(d: &Path, backend: &BackendType, config\_dir: &Path) -> BynarResult<bool>  DESCRIPTION: check if a disk is safe to remove  PARAMETERS: d– the disk device path to check if safe to remove  backend – the backend type  config\_dir – the configuration file directory  RETURNS: Ok(bool) on success, Error otherwise  IMPLEMENTATION: load the backend, and run the backend safe\_to\_remove function. If successful, return Ok(true) if safe to remove, Ok(false) if not safe to remove, or error out. |
| safe\_to\_remove\_disk(s: &mut Socket, d: &str, backend: &BackendType, config\_dir: &Path) -> BynarResult<()>  DESCRIPTION: Check if a disk is safe to remove and send the result to the client  PARAMETERS: s – the socket to send and receive messages from  d– the disk device path to check if safe to remove  backend – the backend type  config\_dir – the configuration file directory  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create the OpBoolResult message. Run the safe\_to\_remove function. Based on the output fill out the OpBoolResult message and convert it to bytes and send to the client. If successful, return Ok(()), otherwise error out. |
| get\_jira\_tickets(s: &mut Socket, config\_dir: &Path) -> BynarResult<()>  DESCRIPTION: get a list of JIRA tickets and send the list to the client  PARAMETERS: s – the socket to send and receive messages from  config\_dir – the configuration file directory  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create an OpJiraTicketsResult Message. Load the config settings and connect to the database. get all pending tickets from the database, and set the tickets in the OpJiraTicketsMessage, and send the message to the client. If successful, return Ok(()), otherwise error out. |
| main()  DESCRIPTION: run the disk manager  PARAMETERS: None  RETURNS: None  IMPLEMENTATION: Create the Command Line Interface and parse the arguments passed in. Check the verbosity and set the logger, and check other CLI inputs. Then, run listen. |

# Disk Testing

## Introduction

This is the disk testing mechanism of Bynar, which uses a State Machine to check the health of a disk and determine whether it has failed or not, as well as whether it needs replacement or intervention. Disk checks are defined and tested, using the state machine to determine what is and is not possible. The state machine itself can be output as a visual diagram when one of the unit tests is run.

## State Machine

The state machine is set up by adding all the transition states into itself, with each state ordered from the most to least ideal outcome.

The state machine, when run, will attempt to run all transitions until an end state is reached and return. It will start from the current state that the machine is in, and loop through all possible next states (edges). If a transition returns Fail, try the next path until all paths are exhausted.

### Type

#### TransitionFn

A function type fn(State, &mut BlockDevice, &Option<ScsiInfo, Option<ScsiInfo>)> bool) -> State

The Transition function defines a transition between two states, given some information on the current Block Device and information gathered from the scsi commands.

### Trait

#### Transition

Transition trait that defines a transition between two states, given some Event, and uses a database connection to save and resume a state. The input state is the state to transition to if the Event is successful.

##### Trait Function Definition

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| transition | to\_state: State  device: &mut BlockDevice  scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>  simulate: bool | Transition from the current state to an ending state given an Event. | State |

### Enums

#### State

A State in the state machine

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Corrupt | The disk or disk filesystem is corrupted. Repairs are attempted |
| Fail | The Transition failed (for whatever reason) |
| Good | The filesystem is good |
| Mounted | The disk was able to be mounted |
| MountFailed | Mounting the disk failed |
| NotMounted | The disk is not mounted |
| ReadOnly | The device is mounted as read only |
| ReformatFailed | Tried to reformat but failed |
| Reformatted | Reformatting the device succeeded |
| RepairFailed | Tried to repair corruption and failed |
| Repaired | Repair corruption succeeded |
| Replaced | Disk was successfully replaced |
| Scanned | Disk is successfully scanned |
| Unscanned | Disk has not been scanned? Scanning failed? |
| WaitingForReplacement | The disk could not be repaired and needs to be replaced |
| WornOut | The disk spindle is worn out and the drive will need to be replaced soon |
| WriteFailed | Write test failed |

##### Trait Implementations

###### Display

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter | Given a State, display the object as a string | fmt::Result |

###### FromStr

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| from\_str | s: &str | Given a string, return a state | BynarError<State> |

###### Clone, Copy, Debug, Eq, Hash, Ord, PartialEq, PartialOrd

#### Fsck

The result of an fsck Linux command

##### Enum Values

|  |  |
| --- | --- |
| Name | Description |
| Ok | Fsck resulted in okay |
| Corrupt | Filesystem is corrupt somehow |

### Structs

#### BlockDevice

A Block Device object, containing metadata and other information about the device

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| device | Device | Device information |
| dev\_path | PathBuf | The path to the device |
| device\_database\_id | Option<u32> | The id of the device in the database |
| mount\_point | Option<PathBuf> | The mount point of the device |
| partitions | BTreeMap<u32, Partition> | A map of the partitions in the device |
| scsi\_info | ScsiInfo | Scsi Information on the device |
| state | State | Current state of the device |
| storage\_detail\_id | u32 | The storage detail id of the device in the database |
| operation\_id | Option<u32> | The operation id of the device n the database |

##### Implementation

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| set\_device\_database\_id | device\_database\_id: u32 | set the device\_database\_id to the id of the disk in the database | None |

##### Trait Implementations

###### Clone, Debug

#### StateMachine

A State Machine

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| dot\_graph | Vec<(State, State, String)> | A record of transitions to be written as a dot graph for visual debugging |
| graph | GraphMap<State, TransitionFn, Directed> | Mapping of valid From -> To transitions |
| block\_device | BlockDevice | The block device |
| scsi\_info | Option<(ScsiInfo, Option<ScsiInfo>)> | Option info of this device and optional scsi host information used to determine if the device is behind a RAID controller |
| simulate | bool | Whether a simulation or not |

##### Implementation

|  |
| --- |
| Function Definition |
| new(block\_device: BlockDevice, scsi\_info: Option<(ScsiInfo, Option<ScsiInfo>)>, simulate: bool) -> StateMachine  DESCRIPTION: Create a new State Machine  PARAMETERS: block\_device – the block device to create a State Machine of  scsi\_info – the optional information of the device to determine if it is RAID  simulate – whether running the state machine is real or simulated  RETURNS: StateMachine  IMPLEMENTATION: create a new StateMachine and set the Vec and GraphMap as empty, and fill in the other attributes with their matching inputs and return the new StateMachine |
| add\_transition(&mut self, from\_state: State, to\_state: State, callback: TransitionFn, transition\_label: &str)  DESCRIPTION: add a transition to the state machine  PARAMETERS: from\_state – the initial state  to\_state – the state to transition to if the transition function is successful  callback – the transition function to attempt  transition\_label – label used to debug the dot graph creation  RETURNS: StateMachine with transition added  IMPLEMENTATION: push the from state, to\_state, and transition label onto the dot graph, and add an edge to the graph from the from\_state to the to\_state using the callback as the transition function |
| run(&mut self)  DESCRIPTION: run all transitions until no more transitions can be run and return  PARAMETERS: None  RETURNS: None  IMPLEMENTATION: Loop, and loop again for each edge in the current state, check the next state in the transition. If the state is Fail, continue, if the state is WaitingForReplacement break the outer loop, if the state is Good break the outer loop, otherwise continue. before looping, if the transition succeeded, save the state and loop again (does not detect unexpected values). Once the edge loop ends we should have advanced the state machine, if not there is an infinite loop, thus we break the loop, otherwise continue the process |
| print\_graph(& self)  DESCRIPTION: print the state machine as a graph  PARAMETERS: None  RETURNS: None  IMPLEMENTATION: for each entry in the dot graph, walk the graph and create a Dot and add in the edges between the states |
| setup\_state\_machine(&mut self)  DESCRIPTION: add all the transition states here.  PARAMETERS: None the dot graph creation  RETURNS: StateMachine with transition added  IMPLEMENTATION: Add transitions to the state graph, which will be run in they order they are added, for multiple edges the states are ordered from the most to the least ideal outcome. |

##### Trait Implementations

###### Debug

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Inputs | Description | Outputs |
| fmt | f: &mut fmt::Formatter<’\_> | Given a formatter, write a debug statement | fmt::Result |

#### AttemptRepair

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, simulate: bool) -> State  DESCRIPTION: Given a Corrupt state, attempt to repair the filesystem on the disk  PARAMETERS: to\_state – the end state to transition to if event successful  device – the block device information needed to attempt a repair  \_scsi\_info – this parameter is unused  simulate – if passed, skip the evaluation of this function  RETURNS: State after attempting to repair the filesystem  IMPLEMENTATION: if not a simulation, attempt to repair the filesystem. If successful, return the input end state, otherwise the repair failed, so return State::Fail. If a simulation, return the input end state value |

###### Debug

#### CheckForCorruption

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, simulate: bool) -> State  DESCRIPTION: Check if there is corruption on the disk and return an end state of Corrupted if so.  PARAMETERS: to\_state – the end state to transition to if the filesystem is corrupt  device – the block device information needed to attempt a check  \_scsi\_info – this parameter is unused  simulate – if passed, skip the evaluation of this function  RETURNS: State after checking the filesystem  IMPLEMENTATION: if not a simulation, attempt to check if the filesystem is corrupt. If the check returns Ok, then the filesystem might have some other problem, or the filesystem could be read only, so return State::Fail. If it returns Corrupt, then return the end\_state input (State::Corrupt). If it errors, then the filesystem check failed, so return State::Fail. If a simulation, return the input end state value |

###### Debug

#### CheckWearLeveling

This transition currently not working properly. Checking the wear leveling is heavily dependent on the make and model of the drive, so if a smartctl command parser is implemented, it might not be accurate or usable on all drives for checking the wear level as not all drives can even check the wear level. Please note that wear level is an SSD drive

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, \_device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: attempt to check if the wear level on a drive is near fail levels.  PARAMETERS: to\_state – the end state to transition to if the drive is worn out  \_device – this parameter currently unused  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking the wear level  IMPLEMENTATION: Currently just returns the end state.  What it SHOULD do is check the wear level, and if the wear level is worn out return the end state, otherwise return a State::Fail (in the event of the check erroring out or the check returning that the drive passed all of the smart checks and the wear level is still good if the drive can even check the wear level, assuming the drive is SMART aware...) |

###### Debug

#### CheckReadOnly

This transition currently not "implemented”.

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(\_to\_state: State, \_device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: attempt to check if the device? Is read only  PARAMETERS: \_to\_state – this parameter is currently unused  \_device – this parameter currently unused  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking for read-only device  IMPLEMENTATION: Currently just returns the end state.  What it SHOULD do is check for read-only....something, and if the device or filesystem or whatever is readonly return the input end state, otherwise return State::Fail. You could parse the /proc/mounts file for “ro”, or check if the /sys/block/xxx/ro file contents is == 1 |

###### Debug

#### Eval

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: attempt to check if the scanned drive is good  PARAMETERS: to\_state – the end state to return if check passes  device – the device information needed to evaluate the drive  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking if the device is good  IMPLEMENTATION: checks if the disk is blank. If so, assuming a blank disk is good, return the end\_state. If not blank, check the filesystem’s LVM (if it uses an LVM) and if it does not error return the end\_state. Check (if there is no mount point) if mounting the device temporarily works. Then check if the mount is writable. If the mount is writable, clean up the mount used by unmounting the device, and return the end state. If the write to mount fails, return State::WriteFailed. Otherwise error outs should return in returning State::Fails. |

###### Debug

#### MarkForReplacement

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: if a drive is Worn Out, mark the drive for replacement and return the input end state  PARAMETERS: to\_state – the end state to return marking is successful  device – the device information needed to evaluate the drive  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking if the device is good  IMPLEMENTATION: Currently just returns the end state.  What it SHOULD do is mark the drive as needing replacement |

###### Debug

#### Mount

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: try to mount a drive, and return the input end state if successful  PARAMETERS: to\_state – the end state to return if mounting is successful  device – the device information needed to mount the drive  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after trying to mount a device temporarily  IMPLEMENTATION: Returns the input end state if mounting and unmounting is successful, otherwise return State::Fail |

###### Debug

#### NoOp

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, \_device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: Do nothing  PARAMETERS: to\_state – the end state to return after doing nothing  \_device – the device information needed to evaluate the drive  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after doing nothing  IMPLEMENTATION: Currently just returns the end state. |

###### Debug

#### Remount

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, \_device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: attempt to remount a disk if possible  PARAMETERS: to\_state – the end state to return if remounting is successful  \_device – this parameter is currently unused  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking if the device can be remounted  IMPLEMENTATION: Run the remount command (mount with remount flags). If successful, return the end state input to the function. Otherwise, return State::Fail |

###### Debug

#### Replace

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, \_device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: check if the drive has been replaced and the host can see it.  PARAMETERS: to\_state – the end state to return if the disk has been successfully replaced  device – the device information needed to check if the host can see  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after checking if the device is replaced and viewable  IMPLEMENTATION: get the device info (if it works then the host can see the device, so return the end state). Otherwise return State::Fail. |

###### Debug

#### Reformat

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, \_scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: reformat a disk and return an end state  PARAMETERS: to\_state – the end state to return if reformating is successful  device – the device information needed to reformat the drive  \_scsi\_info – this parameter is unused  \_simulate –this parameter is currently unused  RETURNS: State after reformatting the drive  IMPLEMENTATION: ensure the drive is NOT mounted. format the device. And if it works, update the UUID of the block device, by creating a new one, probing for the uuid, and looking up the uuid value, then updated the device id. If this all works, return the end state, otherwise if any of the steps fail return State::Fail. |

###### Debug

#### Scan

##### Trait Implementations

###### Transition

|  |
| --- |
| Trait Function Definition |
| transition(to\_state: State, device: &mut BlockDevice, scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>, \_simulate: bool) -> State  DESCRIPTION: Scan a drive and return a state  PARAMETERS: to\_state – the end state to return if scanning is successful  device – the device information needed to run a scan  scsi\_info – the scsi info needed to runa scan  \_simulate –this parameter is currently unused  RETURNS: State after scanning the drive  IMPLEMENTATION: check if the drive is raid backed. If the .0 raid backed is false, run smart checks on the device. If its okay return the end state else Fail. If raid\_backed.0 is true, and the Vendor is Hp, check the scsi\_info’s state. If the state is us Running, then return the end state, otherwise State::Fail. For any other Vendor, skip the scanning and just return the input end state. |

###### Debug

### Functions

|  |
| --- |
| Function Definition |
| filter\_disks(devices: &[PathBuf], storage\_detail\_id: u32) -> BynarResult<Vec<BlockDevice>>  DESCRIPTION: filter disks and get their information while skipping devices that should not be changed, removed etc, such as loopback, LVM, CD/DVD rom, RAM, root, and BOOT  PARAMETERS: devices – list of devices  storage\_detail\_id – the id of some storage detail entry in the database  RETURNS: Ok(block devices matching the criteria) on success, Error otherwise  IMPLEMENTATION: get a list of all device info from the list of devices, removing all devices where information cannot be looked up. For each device left, check the mount point and partition information and update them accordingly. Then, filter the devices by removing all loopback devices, LVM devices, CD/DVD rom devices, RAM devices, and skip the root disk |
| add\_previous\_devices(devices: &mut Vec<BlockDevice>, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping) -> BynarResult<()>  DESCRIPTION: Add in any disks that the database new about that linux can no longer find  PARAMETERS: devices – list of devices  pool – the pool of connections to the database  host\_mapping – the details on how hosts are mapped  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get a list of devices from the database. Add back in any missing devices to the vector. Since the host doesn’t know about the device, we check if the device is waiting for repair. If not, then the database things the disk is still good, so set it as waiting for repair so someone goes and replaces it. |
| check\_all\_disks(host\_info: &Host, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping) -> BynarResult<Vec<BynarResult<StateMachine>>>  DESCRIPTION: retrieves a list of disks and sets up a state machine on each of them. Retrieves previous state and runs through the state machine and preserves the final state in the database before returning a vector of StateMachine  PARAMETERS: host\_info – information on the host  pool – the pool of connections to the database  host\_mapping – the details on how hosts are mapped  RETURNS: Ok( a vector of state machines ) on success, Error otherwise  IMPLEMENTATION: get a list of all devices currently attached to the tree (does not show dead and disconnected devices that are still mounted in /etc/mtab and their scsi information. get the information on all currently mounted devices and remove the ones that we already have, which gives us all broken mounted devices. Get all the info on all devices and filter them, then add any previous devices. Add filtered devices to the database and create a state machine per device. run each state machine and save the state to the database then the run finishes. return a list of state machine end states, otherwise error out if not successful. |
| check\_filesystem(filesystem\_type: &FilesystemType, device: &Path) -> BynarResult<Fsck>  DESCRIPTION: check if the filesystem on a device is corrupt, given the device and filesystem type. Note this assumes that the device is unmounted  PARAMETERS: filesystem\_type – the type of filesystem on the device  device – the device to check the filesystem on  RETURNS: Ok(Fsck result ) on success, Error otherwise  IMPLEMENTATION: match the filesystem type and run the correct check function, returning the result. |
| repair\_filesystem(filesystem\_type: &FilesystemType, device: &Path) -> BynarResult<()>  DESCRIPTION: repair a filesystem, given a corrupted device and filesystem type. Note, this assumes the filesystem is corrupt and the device is unmounted  PARAMETERS: filesystem\_type – the type of filesystem on the device  device – the device to repair the filesystem on  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: match the filesystem type and run the correct repair function, returning the result. |
| check\_writable(path: &Path) -> BynarResult<()>  DESCRIPTION: check if the path is writable  PARAMETERS: path – the path to check  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: create a temporary directory in the path, and a temporary file in the path. Try writing to the file. If successful, return Ok(()), else error out. Note that the TempDir is deleted once the function exits |
| check\_lvm(device: &Path) -> BynarResult<Fsck>  DESCRIPTION: check if the lvm on a device is corrupt, given the device. Note this assumes that the device is unmounted  PARAMETERS: device – the device to check the lvm on  RETURNS: Ok(Fsck result ) on success, Error otherwise  IMPLEMENTATION: scan the LVM on the device. get the list of volume groups, and check if the physical volumes in each volume group is accessible |
| check\_xfs(device: &Path) -> BynarResult<Fsck>  DESCRIPTION: check if the xfs filesystem on a device is corrupt.  PARAMETERS: device – the device to check the xfs filesystem on  RETURNS: Ok(Fsck result ) on success, Error otherwise  IMPLEMENTATION: run the xfs\_repair command with the –n flag, which means it will not modify the filesystem but scan it for corruption. If successful, match the error code to Ok, Corrupt, or a fail code. Return the result or error out. |
| repair\_xfs(device: &Path) -> BynarResult<()>  DESCRIPTION: repair an xfs filesystem, given a corrupted device Note, this assumes the filesystem is corrupt and the device is unmounted  PARAMETERS: device – the device to repair the filesystem on  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: run the xfs\_repair command on the device path. If successful return Ok(()), else error out. |
| check\_ext(device: &Path) -> BynarResult<Fsck>  DESCRIPTION: check if the ext filesystem on a device is corrupt.  PARAMETERS: device – the device to check the ext filesystem on  RETURNS: Ok(Fsck result ) on success, Error otherwise  IMPLEMENTATION: run the e2fsk command with the –n flag, which means it will not modify the filesystem but scan it for corruption. If successful, match the error code to Ok, Corrupt, or a fail code. Return the result or error out. |
| repair\_ext(device: &Path) -> BynarResult<()>  DESCRIPTION: repair an ext filesystem, given a corrupted device Note, this assumes the filesystem is corrupt and the device is unmounted  PARAMETERS: device – the device to repair the filesystem on  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: run the e2fsck command with the –p flag and the device path. If successful return Ok(()), else error out. |
| run\_smart\_checks(device: &Path) -> BynarResult<bool>  DESCRIPTION: run smart checks against the disk  PARAMETERS: device – the device to repair the filesystem on  RETURNS: Ok(bool) on success, Error otherwise  IMPLEMENTATION: get the handle for the libatasmart. Get the smart status. If successful, return Ok(true) if the smart status is true, Ok(false) if the smart status is false, otherwise error out. |
| format\_device(device: &Path) -> BynarResult<()>  DESCRIPTION: format the device with the proper filesystem type.  PARAMETERS: device – the device to format  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the filesystem type of the device. If the filesystem type is Xfs then it needs to be forced to overwrite. format the block device. If successful, return Ok(()), else error out |
| is\_device\_mounted(dev\_path: &Path) -> bool  DESCRIPTION: check if the device is mounted  PARAMETERS: dev\_path – the device to check is mounted  RETURNS: true if mounted, false if not  IMPLEMENTATION: first check if the device is mounted, if true then return. otherwise, check if any of the partitions contained in the device are mounted. If so, then return true, otherwise return false |
| is\_disk\_blank(dev: &Path) -> BynarResult<bool>  DESCRIPTION: make a best guess as to whether a disk is blank  PARAMETERS: dev – the device to check if blank  RETURNS: Ok(is blank?) else Error  IMPLEMENTATION: open and scan the LVM. Got each volume group, check if there is a PVM with the same name as the input device. If so, return Ok(false). check if the GPT header is readable. If so, return Ok(false). If the device is currently mounted, return true. Attempt to mount the device to a temporary directory, and if successful return false, otherwise it’s probably blank. If successful return Ok(false) for passing tests, Ok(true) if none of the tests pass, and therefore the disk is probably blank |
| is\_raid\_backed(scsi\_info: &Option<(ScsiInfo, Option<ScsiInfo>)>) -> (bool, Vendor)  DESCRIPTION: check if the disk is RAID backed  PARAMETERS: scsi\_info – the scsi information to check  RETURNS: (is raid backed?, Who is the Vendor?)  IMPLEMENTATION: if both input option values are None (the outer and inner ScsiInfo), then return (false, Vendor::NONE), otherwise, check if the scsi\_type is StorageArray and Enclosure. If so, then check the vendor. If the vendor is Hp, then it is RAID backed, so return (true, Vendor::Hp), otherwise return (false, vendor). |

# Hardware Testing

## Introduction

Hardware testing module, this module uses the libredfish to check the Hardware status.

## Hardware Tests

### Struct

#### HardwareHealthSummary

A summary of all the hardware status information

##### Attributes

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| array\_controllers | Vec<BynarResult<()>> | Status of the array controllers |
| disk\_drives | Vec<BynarResult<()>> | A list of the physical disk drive status |
| manager | Vec<BynarResult<()>> | The iLo status |
| power | Vec<BynarResult<()>> | The Power supply status |
| storage\_enclosures | Vec<BynarResult<()>> | Status of the storage enclosures |
| thermals | Vec<BynarResult<()>> | The fan status |

### Functions

|  |
| --- |
| Function Definition |
| collect\_redfish\_info(config: &ConfigSettings) -> BynarResult<HardwareHealthSummary>  DESCRIPTION: collect the hardware health information from redfish  PARAMETERS: config – the configuration settings needed to connect to redfish  RETURNS: Ok(hardware health summary) on success, Error otherwise  IMPLEMENTATION: Build a Client socket. If the redfish ip address is not specified, skip that check. Otherwise, parse the redfish configuration from the config input. Create a new instance of a Redfish handler, and get the array controllers. For each controller, add it to the list of controllers and get all storage enclosures attached to the controller and grab all disks attached to the controller. On the resulting lists, evaluate the storage and collect the results into vectors. Get the manager from the redfish handler and evaluate it, same with the thermal and power. If everything is successful, return Ok(HardwareHealthSummary) otherwise error out. |
| check\_hardware(config: &ConfigSettings) -> BynarResult<HardwareHealthSummary>  DESCRIPTION: public wrapper function for collect\_redfish\_info  PARAMETERS: config – the configuration settings needed to connect to redfish  RETURNS: Ok(hardware health summary) on success, Error otherwise  IMPLEMENTATION: call collect\_redfish\_info and return the result |
| evaluate\_storage<T>(hardware: T) -> BynarResult<()> where T: Hardware + Status  DESCRIPTION: evaluate an input hardware object of type Hardware + Status  PARAMETERS: hardware – the hardware to evaluate  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Check if the hardware health is NOT OK. If so, return an Error, otherwise return Ok(()) |
| evaluate\_manager(manager: &Manager) -> Vec<BynarResult<()>>  DESCRIPTION: evaluate the ilo status. If there are no issues then the vector should be empty  PARAMETERS: manager – the ilo status manager  RETURNS: Vec<is the status of an ilo okay?>  IMPLEMENTATION: look through all the self test results. If the result status is not OK or Informational, then there is an error, so add an Error to the vector of results. return the vector. |
| evaluate\_power(power: &Power) -> Vec<BynarResult<()>>  DESCRIPTION: evaluate the power status. If there are no issues then the vector should be empty  PARAMETERS: power – the Power status  RETURNS: Vec<is the status of a power supply okay?>  IMPLEMENTATION: look through all the self test results. If the result status is not OK then the power supply failed, so add an Error to the vector of results. return the vector. |
| evaluate\_power(power: &Power) -> Vec<BynarResult<()>>  DESCRIPTION: evaluate the power status. If there are no issues then the vector should be empty  PARAMETERS: power – the Power status  RETURNS: Vec<is the status of a power supply okay?>  IMPLEMENTATION: look through all the power supply status. If the result status is not OK then the power supply failed, so add an Error to the vector of results. return the vector. |
| evaluate\_thermals(thermal: &Thermal) -> Vec<BynarResult<()>>  DESCRIPTION: evaluate the fan status as well as the temperature. If there are no issues then the vector should be empty  PARAMETERS: thermal – the Thermal status  RETURNS: Vec<is the status of a fan okay?>  IMPLEMENTATION: look through all the fans. If the result status is not OK then the fan failed, so add an Error to the vector of results. look through all the temperature readings, if the health is not OK, then the temperature reading is failing, so add an Error to the vector of results. return the vector. |

# Bynar

## Introduction

This program handles detection of failed hard drives, files a ticket for a datacenter technician to replace the drive, waits for the resolution of the ticket and then makes an API call to the disk-manager to add the new disk back into the server.

### Main Process Functions

|  |
| --- |
| Function Definition |
| notify\_slack(config: &ConfigSettings, msg: &str) -> BynarResult<()>  DESCRIPTION: Send a slack notification  PARAMETERS: config – the configuration settings needed to connect to a Slack channel  msg – the message to send  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: Create a Slack handle, and connect to the Slack API system using the information in the config. Create the payload to send to the Slack channel, and send it. If successful, return Ok(()), otherwise error out |
| get\_public\_key(config: &ConfigSettings, host\_info: &Host) -> BynarResult<String>  DESCRIPTION: get the server public key if possible from either Hashicorp vault or some .pem file.  PARAMETERS: config – the configuration settings needed to connect to the Hashicorp Vault  host\_info – the host information  RETURNS: Ok(public key) on success, Error otherwise  IMPLEMENTATION: check if the vault endpoint and token are set. If so, attempt to get the key from the vault using the get\_vault\_token method. otherwise, check for a .pem file in /etc/bynar for the public key. If successful, return the key, otherwise error out. |
| check\_for\_failed\_disks(config: &ConfigSettings, host\_info: &Host, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping, simulate: bool) -> BynarResult<()>  DESCRIPTION: Run a scan to check for failed disks  PARAMETERS: config – the configuration settings needed  host\_info – the host information  pool – the pool of connections to the database  host\_mapping – the mapping of host details  simulate – passed to skip evaluation  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the public key. Create the description needed to open a ticket in case a disk failed. Check all the drives, running check\_all\_disks and for each disk check if the state is Waiting For Replacement. If so, add to the description and check if the database knows the disk is waiting for repair. If so, move on to the next disk, otherwise ask the disk-manager if the disk is safe to remove. If yes, notify slack and remove the disk. If false, notify slack and file a ticket with Jira and update the database. If a disk is stuck in the fail state, error message. Otherwise the state is Good so move on. If successful for all disks, return Ok(()), otherwise error out |
| evaluate(results: Vec<BynarResult<()>>, config: &ConfigSettings, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping) -> BynarResult<()>  DESCRIPTION: check if there are any errors besides hardware errors  PARAMETERS: results – the list of errors after running a hardware scan  config – the configuration settings needed  pool – the pool of connections to the database  host\_mapping – the mapping of host details  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: for each result, check if the error type is a HardwareError type. If so, check if the hardware is waiting for repair. If it isn’t, file a ticket. If the Error Type is NOT a HardwareError type, then Error out. |
| check\_for\_failed\_hardware(config: &ConfigSettings, host\_info: &Host, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping, simulate: bool) -> BynarResult<()>  DESCRIPTION: Run a scan to check for failed hardware  PARAMETERS: config – the configuration settings needed  host\_info – the host information  pool – the pool of connections to the database  host\_mapping – the mapping of host details  simulate – passed to skip evaluation  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the public key. Create the description needed to open a ticket in case there is a hardware failure. Run check\_hardware and evaluate the results disk drives, manager, power, storage enclosures, and thermals for any errors that aren’t Hardware errors. If successful, return Ok(()), otherwise error out. |
| add\_repaired\_disks(config: &ConfigSettings, host\_info: &Host, pool: &Pool<ConnectionManager>, host\_mapping: &HostDetailsMapping, simulate: bool) -> BynarResult<()>  DESCRIPTION: add repaired disks back to the cluster  PARAMETERS: config – the configuration settings needed  host\_info – the host information  pool – the pool of connections to the database  host\_mapping – the mapping of host details  simulate – passed to skip evaluation  RETURNS: Ok(()) on success, Error otherwise  IMPLEMENTATION: get the public key. get the list of outstanding repair tickets and check them for resolved tickets. For each ticket in the list, check if the ticket is resolved. If so, connect to the disk-manager and ask it to add the disk back to the cluster. If successful, resolve the ticket in the database. If the ticket is not resolved, skip it. If everything is successful, return Ok(()), otherwise error out. |
| main()  DESCRIPTION: The main Bynar program, it gathers a list of all disks, checks each disk and decides if it needs replacing and files a ticket. It then records the ticket, checks for resolved tickets and adds the disks back in records everything in the database  PARAMETERS: None  RETURNS: None  IMPLEMENTATION: Create and get the command line arguments. Bynar specifically takes the optional configdir argument, the optional simulate argument, and the optional –v argument. Depending on the number of input –v flags, increase the verbosity of the logs. Check the configdir input and whether the command is a simulation or not. Gather the host information and load the configuration. Connect to the database and update the host information. Check for failed disks, failed hardware, and add in any repaired disks. |