Skynet Web API v2.0

Protocol

Request URL: http://api.skynet.unc.edu/#.#/*resource*?*param*=*value*…
#.# -- API version number, currently 2.0.
Below, the prefix http://api.skynet.unc.edu/#.# is omitted for brevity. POST and PUT request arguments are listed in the query string, although they should be sent in the request body using URL encoding.
Response: application/json.
Errors:

1. HTTP status.
2. Content-Type: text/plain.
3. Body: error message.
4. HTTP header web2py_error set to extended error info.

Authentication

1. API access token in request parameters:
   /api/…?access_token=*token*
2. API access token in HTTP headers
   Authentication-Token: *token*
3. Session-based (Skynet login) when redirected from the website; only enabled for /download requests.

Admin API

Users

```
class UserSchema {
        id: int,                              // RO for everyone
        username: string,                     // --//--
        email: string,                        // RW for admins
        owningGroupId: int,                   // --//--
        priority: int,
        level: int,
        canModify: bool,
        groupDeactivated: bool,
        deleted: bool,
        alertsOn: int,                        // RW for user, visible for admins
        reduceImagesByDefault: bool,          // --//--
        obsViewMode: str,
        expViewMode: str,
        prefer16BitImages: bool,
        hideClosedOwnerTimeAccounts: bool,
        hideClosedCollabTimeAccounts: bool,
        hideClosedGroupTimeAccounts: bool,
        firstName: str,                          // RW for user only
        lastName: str,                           // --//--
        phoneEmail: str,
        birthdate: datetime,
        consumerKey: str,
        consumerSecret: str,
        accessToken: str,
        location: str,
        userSelfRegistrationId: int,

        // relationships
        selfRegistration: {id: int, keyCreatedOn: datetime, registrationKey: str, activated: bool, user: {id: int, username: str}},
        timeAccounts: [{id: int}, …],
        groups: [{id: int, name: str}, …],
        owningGroup: {id: int, name: str},
        roles: [RoleSchema, …],
        userCollabRoles: [RoleSchema, …],
        userGroupRoles: [RoleSchema, …],
        userSiteRoles: [RoleSchema, …],
        userTeleRoles: [RoleSchema, …],
        userTeleOwnerRoles: [RoleSchema, …],
        pendingGroupMemberships: {id: int}, …],
        isSkynetAdmin: bool,
}

class RoleSchema {
        id: int,
        name: str,
        description: str,
}
```

GET /users
- return the calling user’s profile (class UserSchema)
- available to any authenticated user

GET /users?*param*=*value*…
- return users matching specific criteria:

    - **group**: group name or ID or comma-separated list of names/IDs users belonging to group(s); multiple **group** arguments are allowed; the user must be a group or an owning collab admin
    - **offset**: only return users starting from the given index (zero-based)
    - **limit**: only return the given number of users at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only UserSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other UserSchema fields are also accepted as query parameters, e.g. GET /users?deleted=1
- Skynet admins must set **group**="" to get all SkyNet users

GET /users/*user_id_or_username*
- return user with the given ID or username
- only SkyNet admins and users themselves can get the full user profiles
- owning group admins can get partial user profiles with sensitive information hidden
- group admins can get IDs and usernames of group members

POST /users?username=*username*&email=*email*[&*param*=*value*…]
- create a new user; optional parameters include:
    - **owningGroup**: name or ID of the user's owning group; new user is automatically included in the group; if unspecified, an "orphan" user is created who can be added later to any group that becomes their owning group
    - any other UserSchema fields
- users with no owning group can be created by SkyNet admins only
- for user created with owning group, the calling user must be group admin with CREATE_MEMBERS role or a SkyNet admin

PUT /users?*param*=*value*…
- update the calling user's profile
- Parameters may include any UserSchema fields
- group admins with MANAGE_MEMBERS role for the user's owning group can modify any non-personal scalar fields
- when updating the calling user's own profile, non-admin user can modify only personal data and preferences

DELETE /users/*user_id_or_username*
- permanently delete user from SkyNet database
- available to SkyNet admins and admins of all user's groups with MANAGE_MEMBERS role only

- currently only a user can be deleted who has no associated time accounts and observations

Groups

```
class GroupSchema {
        id: int,
        name: str,
        priority: int,
        owningCollabId: int,
        owningCollab: {id: int, name: str},
        users: [{id: int, username: str}, …],
        ownedUsers: [{id: int, username: str}, …],
        timeAccounts: [{id: int}, …],
        collabs: [{id: int, name: str}, …],
        pendingGroupMemberships: [{id: int}, …],
        pendingCollabMemberships: [{id: int}, …],
        admins: [{id: int, username: str}, …],
}
```

GET /groups[?*param*=*value*…]

- return groups matching specific criteria:
    - **collab**: request groups belonging to the given collab(s); must be a comma-separated list of collab IDs or names; multiple **collab** arguments are allowed; the user must be a SkyNet or collab admin
    - **user**: request groups of the given user(s); must be a comma-separated list of user IDs or usernames; multiple **user** arguments are allowed; each user must belong to a group administered by the calling user
    - **offset**: only return items starting from the given index
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only GroupSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other GroupSchema fields are also accepted as query parameters, e.g. GET /groups?priority=2
- only groups that the user has an admin role for, groups belonging to collab administered by the user, and the user's own groups are returned
- SkyNet admins must set e.g. **user**="" to get all groups

GET /groups/*group_id_or_name*

- return the profile (GroupSchema) of group with the given ID or name
- only SkyNet and group admins can get the full collab profiles; non-admin group members and group's collab admins can get a limited group profile

GET /groups/*group_id_or_name*/users

- return users belonging to the given group
- alias to GET /users?group=*group_id_or_name*
- the user must be a SkyNet or group admin

POST /groups?name=*name*[&*owning_collab*=*collab_id_or_name*]

- create a new empty group
- optional parameters -- see GroupSchema
- the user must be a SkyNet admin or admin of a collab that becomes the new group's owning collab

POST /groups/*group_id_or_name*/users/*user_id_or_username*

- add user to group
- the user must be a SkyNet or group admin

PUT /groups/*group_id_or_name*?*param*=*value*…

- modify group parameters (see GroupSchema)
- the user must be a SkyNet or group admin

DELETE /groups/*group_id_or_name*/users/*user_id_or_username*

- remove user from group
- the user must be a SkyNet or group admin

DELETE /groups/*group_id_or_name*

- permanently delete group from SkyNet database
- the user must be a SkyNet admin or admin of the owning collab

Collaborations

```
class CollaborationSchema {
        id: int,
        name: str,
        groups: [{id: int, name: str}, …],
        timeAccounts: [{id: int}, …],
        ownedGroups: [{id: int, name: str}, …],
        pendingCollabMemberships: [{id: int}, …],
        admins: [{id: int, username: str}, …],
}
```

GET /collabs[?*param*=*value*…]

- return collaborations matching specific criteria:
    - **group**: request collabs of the given group(s); must be a comma-separated list of group IDs or names; multiple **group** arguments are allowed; default: return user's own collabs
    - **offset**: only return items starting from the given index (zero-based)
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only CollaborationSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other CollaborationSchema fields are also potentially accepted as query parameters
- only collabs that the user has an admin role of and the user's own collabs are returned
- SkyNet admins must set **group**="" to get all collabs

GET /collabs/*collab_id_or_name*

- return the profile (CollaborationSchema) of collaboration with the given ID or name
- only SkyNet and collab admins can get the full collab profiles; non-admin users can get a limited profile of a collab they belong to via any of their groups

GET /collabs/*collab_id_or_name*/groups

- return groups belonging to the given collaboration
- alias to GET /groups?collab=*collab_id_or_name*
- the user must be a SkyNet or collab admin

POST /collabs?name=name[&param=value]

- create a new empty collaboration
- optional parameters -- see CollaborationSchema
- the user must be a SkyNet admin

POST /collabs/*collab_id_or_name*/groups/*group_id_or_name*

- add group to collaboration
- the user must be a SkyNet or collab admin

PUT /collabs/*collab_id_or_name*?*param*=*value*…

- modify collaboration parameters (see CollaborationSchema)
- the user must be a SkyNet or collab admin

DELETE /collabs/*collab_id_or_name*/groups/*group_id_or_name*

- remove group from collaboration

- the user must be a SkyNet or collab admin

DELETE /collabs/*collab_id_or_name*

- permanently delete collaboration from SkyNet database
- the user must be a SkyNet admin

Telescopes

```
class TelescopeSchema {
        id: int,
        name: str,
        site: {id: int, name:str, latDegs: number, lngDegs: number},
        teleType: str = “optical”,
        ipAddress: str,
        siteId: int,
        thumbnail: str,
        webcamUrl: str,
        description: str,
        primaryImage: str,
        isAvailable: bool,
        lastTeleOwnerUsageReset: datetime,
        minEl: number,
        maxSlewRateAxis1: number,
        maxSlewRateAxis2: number,
        localHorizonData: [{azDegs: number, elDegs: number}, …],
        ownerShares: [TeleOwnerShareLimitedSchema, …],
        guestShares: [TeleOwnerGuestShareLimitedSchema, …],
        status: TeleStatus2Schema,
        admins: [{id: int, username: str}, …],
        tcsPort: int,
        maxTrackingDuration: number,
        settleTime: number,
        grbFilterNames: str,
        siteName: str,
        mountType: str,
        meridianSec: int,
        owner: str,
        flatTarget: number,
        flatTolerance: number,
        fileBlockSize: int,
        maxMbs: number,
        flatMaxTime: number,
        flatMinTime: number,
        darkElev: number,
        flatElev: number,
        lightElev: number,
        dmsPort: int,
        pixScale: number,
        extraText: str,
        dmsIP: str,
        aperture: number,
        avgWaitTime: number,
        focalLength: number,
        teleGroupId: int,
        rbiTimeConstant: number,
        rbiAvgBkg: int,
        allowBrightTargets: bool,
        efficiency: number,
        scheduleNextFlatObsOn: datetime,
        scheduleNextDarkObsOn: datetime,
        allowObsLinking: bool,
        initiateDmsConnection: bool,
        enforceMaxExpLength: bool,
        allowCustomBinning: bool,
        defaultBinning: int,
        dmsStatus = DmsStatusSchema,
        filters = [{id: int, name: str}, …],
        grbFilters = [{id: int, name: str}, …}],
        darkCalibrationConfigs = [{darkConfigId: int, teleId: int, numExps: int, expLength:
        number}, …],
        flatCalibrationConfigs [{flatConfigId: int, teleId: int, numExps: int, filterId: int}, …],
        genericFilters [{id: int, name: str}, …],
        flatEfficiency: number,
        fov = [number, number],
        pixelScale: number,
        uniqueId: str,
}

class AntennaSchema {
        id: int,
        name: str,
        site: {id: int, name:str, latDegs: number, lngDegs: number},
        teleType: str = “optical”,
        ipAddress: str,
        siteId: int,
        thumbnail: str,
        webcamUrl: str,
        description: str,
        primaryImage: str,
        isAvailable: bool,
        lastTeleOwnerUsageReset: datetime,
        minEl: number,
        maxSlewRateAxis1: number,
        maxSlewRateAxis2: number,
        localHorizonData: [{azDegs: number, elDegs: number}, …],
        ownerShares: [TeleOwnerShareLimitedSchema, …],
        guestShares: [TeleOwnerGuestShareLimitedSchema, …],
        status: TeleStatus2Schema,
        admins: [{id: int, username: str}, …],

        slewOverhead: number,
        dmsIp: str,
        diameter: number,
        receiverHistory: [{id: int, teleId: int, receiverId: int, startDate: datetime}, …],
        receiver: ReceiverSchema,
}

class TeleOwnerShareLimitedSchema {
        teleId: int,
        teleOwnerId: int,
        totalShare: number,
        ownerShare: number,
        guestShare: number,
        publicShare: number,
        ownerContesting: int,
        guestContesting: int,
        publicContesting: int,
        hasTooPrivilege: bool,
        totalUsage: number,
        totalContestedUsage: number,
        totalTimeWaiting: number,
        ownerUsage: number,
        ownerContestedUsage: number,
        ownerTimeWaiting: number,
        guestUsage: number,
        guestContestedUsage: number,
        guestTimeWaiting: number,
        publicUsage: number,
        publicContestedUsage: number,
        publicTimeWaiting: number,
        pendingTotalUsage: number,
        pendingTotalContestedUsage: number,
        pendingTotalTimeWaiting: number,
        pendingOwnerUsage: number,
        pendingOwnerContestedUsage: number,
        pendingOwnerTimeWaiting: number,
        pendingGuestUsage: number,
        pendingGuestContestedUsage: number,
        pendingGuestTimeWaiting: number,
        pendingPublicUsage: number,
        pendingPublicContestedUsage: number,
        pendingPublicTimeWaiting: number,
        lastUpdate: datetime,
}

class TeleOwnerGuestShareLimitedSchema {
        teleId: int,
        teleOwnerId: int,
        totalShare: number,
        contesting: int,
        totalUsage: number,
        totalContestedUsage: number,
        totalTimeWaiting: number,
        pendingTotalUsage: number,
        pendingTotalContestedUsage: number,
        pendingTotalTimeWaiting: number,
        lastUpdate: datetime,
}

class TeleStatus2Schema {
        teleId: int,
        connected: bool,
        cameraStatus; str,
        domeStatus; str,
        mountStatus: str,
        weatherStatus: str,
        obsId: int,
        radioObsId: int,
        lastUpdate: datetime,
        raHours: number,
        decDegs: number,
        azDegs: number,
        elDegs: number,
        expProgress: number,
        expLength: number,
        expId: int,
        focusStatus: str,
        operatingMode: str,
}

class CameraSchema {
        id: int,
        name: str,
        pixelWidth: int,
        pixelHeight: int,
        pixelSize: number,
        blankLevel: number,   // ADU
        gain: number,         // $e^-$/ADU
        readNoise: number,    // $e^-$
}

class ReceiverSchema {
        bandStartFreq: number,
        bandStopFreq: number,
        hasTunableLo: bool,
        discreteFrequencyOffset: number,
        lowResBandwidth: number,
        lowResUsesFilters: bool,
        lowResDualBand: bool,
        lowResCenterFreqDefault: number,
        lowResSecondaryFreqDefault: number,
        highResBandwidth: number,
        highResUsesFilters: bool,
        highResDualBand: bool,
        highResCenterFreqDefault: number,
        highResSecondaryFreqDefault: number,
        filters: [{id: int, name: str}, …],
}
```

GET /scopes[?*param*=*value*…]

- return telescopes matching specific criteria:
    - **site**: return only telescopes located at site(s) with the given IDs/names
    - **teletype**=optical: only return optical telescopes
    - **teletype**=radio: only return radio telescopes
    - **offset**: only return items starting from the given index (zero-based)

        - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
        - **include**: list of fields to return; default: return only *Schema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
        - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
        - any other TelescopeSchema or AntennaSchema fields are also accepted as query parameters, e.g. GET /scopes?isAvailable=1
        - search parameter specific to optical or radio telescopes automatically adds teleType=optical or teleType=radio, respectively
- the user must be a SkyNet or telescope admin or have a time account that provides access to some telescopes

GET /scopes/*telescope_id_or_name*
- return the given telescope profile (serialized TelescopeSchema or AntennaSchema)
- only telescope admins can access the full profiles
- users with time accounts that provide access to the given telescope can view limited profiles

Filters

```
class FilterSchema {
        id: int,
        name: str,
        order: int,
        type: str,                      // "standard" or "generic"
        throughput: number,
        zeroPoint: number,                                                // for standard filters
        flatOrder: int,
        htmlHexColor: str,
        transmissionData: [{filterId: int, nu: number, transmission: number}, …],
        filters: [{id: int, name: str, order: int, type: str="standard", ...}, …],    // for generic filters
}
```

GET /filters
- return all available filters IDs

GET /filters?*param*=*value*…
- return filters matching specific criteria:
    - **include**: list of fields to return; default: return only FilterSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields

    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any FilterSchema fields are accepted as query parameters, e.g. GET /filters?type=standard

GET /filters/*filter_id_or_name*

- return filter description (serialized FilterSchema) with the given ID or name

Optical Observing API

Observations

```
class ObservationSchema {
        id: int,
        name: str,
        raHours: number,
        decDegs: number,
        grbId: int,
        state: str,        // "active", "completed", "canceled"
        minEl: number,
        maxSun: number,
        mode: int,         // 0 = continue on same, 1 = continue on different
        timeIn: datetime,
        type: str,         // "light", "dark", "flat"
        objectName: str,
        objectType: str,           // "sidereal", "planet", "asteroid", "comet", "nonsidereal"
        objectDist: number,
        cancelAfterUtc: datetime,
        timeSubmittedUtc: datetime,
        totalTimeRemaining: number,
        userId: int,
        nextExpStartAfterUtc: datetime,
        collabId: int,
        groupId: int,
        timeAccountId: int,
        parentCollabTimeAccountId: int,
        parentGroupTimeAccountId: int,
        teleOwnerId: int,
        accountType: str,
        ditherEnabled: bool,
        ditherXSize: int,
        ditherYSize: int,
        ditherSpacingArcsecs: number,
        targetTracking: str,
        triggerRepointEnabled: bool,
        triggerRepointArcmins: number,
        pointAheadEnabled: bool,
        pointAheadSecs: number,
        constantRaOffsetArcmins: number,
        constantDecOffsetArcmins: number,
        minMoonSepDegs: number,
        fieldLockUtc: datetime,
        rbiFractionAvgBkgLimit: number,
        tooJustification: str,
        currentTeleId: int,
        priority: int,
        primaryTeleId: int,
        orbitalElementsId: int,
        isToo: bool,
        user: {id: int, username: str},
        group: {id: int, name: str},
        collab: {id: int, name: str},
        teleOwner: {id: int, name: str},
        userPreference: {obsId: int, userId: int, reduce: bool, invert: bool, scalePreset: str},
        currentTelescope: {id: int, name: str, site: {id: int, name: str, latDegs: number, lngDegs: number}},
        primaryTelescope: {id: int, name: str, site: {id: int, name: str, latDegs: number, lngDegs: number}},
        orbitalElements = {id: int, epochJd: number, m: number, peri: number, node: number, incl: number, e: number, a: number},
        obsNotes: [{id: int, obsId: int, note: str, timeCreated: datetime, obs: {id: int, name: str}}, …],
        exps: [ExposureSchema, …],
        telescopes: [{{id: int, name: str, site: {id: int, name: str, latDegs: number, lngDegs: number}}, …],
}
```

GET /obs

- return all user's observations (list of ObservationSchema objects)
    - **include**: list of fields to return; default: return only ObservationSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**

GET /obs?*param*=*value*…

- return observations matching specific criteria:
    - **user**: for SkyNet admins and group admins only: request observations submitted by specific user(s); must be a comma-separated list of user IDs or names; multiple **user** arguments are allowed

    - **group**: for SkyNet admins and group admins only: request observations submitted by all users of specific group(s); must be a comma-separated list of group IDs or names; multiple **group** arguments are allowed
    - **scope**: only return observations submitted to specific telescope ID(s) or name(s); must be a comma-separated list of telescope IDs or names; multiple **scope** arguments are allowed
    - **after**: request observations submitted after the given date/time
    - **before**: request observations submitted before the given date/time
    - **offset**: only return items starting from the given index (zero-based)
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only ObservationSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other ObservationSchema fields are also accepted as query parameters, e.g. GET /obs?state=active
- Setting either **user** or **group** allows the user to get observations submitted by other users, provided the user has enough privileges to view others' observations (collaboration, group, and telescope admins of the observation's time account and SkyNet admins). If neither of those parameters are set, the user gets only their own observations, possibly restricted by **scope** and other parameters. To get *all* observations for the given telescope, admin needs to set one of those parameters to match all observations, e.g. **scope**=Morehead&**user**=.

GET /obs/*obs_id_or_name*

- return optical observation (serialized ObservationSchema) with the given ID or name
- the calling user must be either the observation submitter, the observation's time account collaboration, group, or telescope admin, or a SkyNet admin

GET /obs/*obs_id_or_name*/exps

- return the given observation's exposures
- alias to GET /exps?obs=*obs_id_or_name*

POST /obs?*param*=*value*…

- submit empty observation with the given parameters (see ObservationSchema; not all fields are settable)
- exposures are added later by POST /exps (see below) or in the same request by specifying a JSON-encoded list of Exposure objects: **exps**=[{ExposureSchema}, {ExposureSchema}, …], e.g. POST /obs?name=M31&primaryTeleId=Prompt5&exps=[{“expLength”: 60, “filterRequested”: “R”}]

PUT /obs/*obs_id_or_name*?*param*=*value*

- modify existing observation; for the list of parameters, see ObservationSchema (not all fields are allowed to change)
- **state**=canceled cancels the whole observation
- **state**=active reactivates observation
- exps updates are not allowed; to add/modify exposures, use the /exps endpoint described below

Exposures

```
class ExposureSchema {
        id: int,
        obsId: int,
        expNum: int,
        expLength: number,
        expLengthUsed: number,
        state: str,                    // “ready”, “completed”, “archived”, “canceled”, “deleted”
        type: str,                     // “light”, “dark”, “flat”
        timeIn: datetime,
        startAfter: datetime,
        endBefore: datetime,
        timeTaken: datetime,
        timeSubmitted: datetime,
        timeArchived: datetime,
        isCompressed: bool,
        commandedRa: number,
        commandedDec: number,
        delay: int,
        timeTakenIsFromHdr: bool,
        compression: str,
        creditsCharged: int,
        teleOwnerIdUsed: int,
        timeAccountId: int,
        parentCollabTimeAccountId: int,
        parentGroupTimeAccountId: int,
        teleId: int,
        filterIdRequested: int,
        filterIdUsed: int,
        wcsId: int,
        wcsState: str,                 // “not_attempted”, “in_progress”, “completed”
        statsState: str,               // “not_attempted”, “in_progress”, “completed”
        numFileErrors: int,
        targetExpId: int,
        binningRequested: int,
        binningUsed: int,
        linkedExps: [Exposure, …],
        filterRequested: {id: int, name: str},
        filterUsed: {id: int, name: str},
        telescope: {id: int, name: str, site: {id: int, name: str, latDegs: number, lngDegs:
        number}},
        teleOwner: {id: int, name: str},
        wcsSolution: {id: int},
        obs: {id: int, name: str},
        centerTime: datetime,
        primaryTeleExpLength: number,
        filterName: str,
        jpgDownloadUrl: str,
        fitsDownloadUrl: str,
        remoteFilename: str,
        camera: CameraSchema,
        blankLevel: number,         // ADU
        gain: number,               // $e^-$/ADU
        readNoise: number,          // $e^-$
        imageId: str,
}
```

GET /exps?*param*=*value*…

- return exposures matching specific criteria:
    - **obs**: only return exposures for specific observation ID(s) or name(s); must be a comma-separated list of observation IDs or names; multiple **obs** arguments are allowed
    - **user**: for SkyNet admins and group admins only: request exposures submitted by specific user(s); must be a comma-separated list of user IDs or names; multiple **user** arguments are allowed; if omitted, return calling user's own exposures
    - **group**: for SkyNet admins and group admins only: request exposures submitted by all users of specific group(s); must be a comma-separated list of group IDs or names; multiple **group** arguments are allowed
    - **scope**: only return exposures taken with specific telescope ID(s) or name(s); must be a comma-separated list of telescope IDs/names; multiple **scope** arguments are allowed
    - **after**: request exposures taken after the given date/time
    - **before**: request exposures taken before the given date/time
    - **offset**: only return items starting from the given index (zero-based)
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000

    - **include**: list of fields to return; default: return only ExposureSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other ExposureSchema fields are also accepted as query parameters, e.g. GET /exps?wcsState=completed
- Setting either **obs**, **user**, or **group** allows the caller to get observations submitted by other users, provided the caller has privileges to view others' observations (collaboration, group, and telescope admins of the observation's time account and SkyNet admins). If neither of those parameters are set, the user gets only their own exposures, possibly restricted by **scope**, **state**, and other parameters. To get *all* exposures for the given telescope, admin needs to set one of those parameters to match all exposures, e.g. **scope**=Morehead&**user**=.

GET /exps/*exp_id*
- return exposure (serialized ExposureSchema) with the given ID
- the calling user must be either the observation submitter, the observation's time account collaboration, group, or telescope admin, or a SkyNet admin

POST /exps?obs=*obs_id_or_name*&*param*=*value*…
- create exposure with the given parameters and add it to existing observation (see ExposureSchema; not all fields are settable)
- If exposure is submitted to observation with a non-null primary telescope, or an explicit **teleId**/**telescope** is set, exposure length (**expLength**) is treated as the actual exposure length in seconds for this telescope; otherwise, **expLength** is treated assuming unit telescope efficiency, i.e. for the standard 16-inch telescope. In all cases, the specified exposure length is then scaled by the efficiency of the telescope it is finally dispatched to.

PUT /exps/*exp_id*?*param*=*value*
- modify existing exposure; for the list of parameters, see ExposureSchema (not all fields are allowed to change)
- **state**=canceled cancels exposure
- **state**=ready reactivates exposure

Master Cals

```
class MasterCalibrationSchema {
        id: int,
        date: datetime,
        startDate: datetime,
        stopDate: datetime,
        combineMethod: str,          // “mean”, “median”, “sum”
        rejectMethod: str,           // “none”, “minmax”, “sigclip”, “avgsigclip”, “chauvenet”
        average: number,
        stdev: number,
        expLength: number,           // seconds
        filterName: str,
        imageType: str,              // “bias”, “dark”, “flat”
        scaleMethod: str,            // “none”, “mode”
        rejected: bool,
        rejectedBy: str,             // “mastergen”, “user”, “cascade”
        chauvenetRejectedPercentage: number,
        compression: str,            // “uncompressed”, “gzip”, “fpack”
        onNewRaid: bool,
        teleId: int,
        filterId: int,
        binning: int,
        components: [MasterComponentSchema, ...],
        filter: {id: int, name: str},
        telescope: {id: int, name: str, site: {id: int, name: str, latDegs: number, lngDegs: number}}
}

class MasterComponentSchema {
        expId: int,
        masterId: int,
        included: bool,
        rejectionReason: str,
        rejectionDescription: str,
        biasMasterId: int,
        darkMasterId: int,
        exp: {id: int, obsId: int},
}
```

GET /master-cals?*param*=*value*…

- return master cals matching specific criteria:
    - **scope**: only return masters for specific telescope(s); must be a comma-separated list of telescope IDs or names; multiple **scope** arguments are allowed
    - **obs**: only return masters for specific observation(s); must be a comma-separated list of observation IDs or names; multiple **obs** arguments are allowed
    - **exp**: only return masters for specific exposure(s); must be a comma-separated list of exposure IDs; multiple **exp** arguments are allowed
    - **delta**: maximum separation in days between master cal and exposure epochs for **obs** and **exp**; default: 10

    - **after**: only return masters with startDate after the given date/time
    - **before**: only return masters with stopDate before the given date/time
    - **offset**: only return items starting from the given index; default: start from 0
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only MasterCalibrationSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other MasterCalibrationSchema fields are also accepted as query parameters, e.g. GET /master-cals?imageType=flat
- users other than SkyNet admins may only get masters for telescopes they are admins of or those they have access to via any of their active time accounts.
- by default, SkyNet admins get masters for the telescopes they have a time account for unless they explicitly set **scope**.

GET /master-cals/*master_id*
- return master cal (serialized MasterCalibrationSchema) with the given ID
- the calling user must have access to the telescope by means of any of their time accounts or be a telescope admin

Data Download API

GET /download/fits?*param*=*value*…
- download data as FITS file(s)
- for optical observations, files are either original or preprocessed, depending on parameters
- if parameters uniquely identify a single data file (e.g. image=# or radio_obs=#), a single uncompressed FITS file is returned (Content-Type: application/force-download; Content-Disposition: attachment; filename="…"; Content-Transfer-Encoding: binary); otherwise, data are returned in a ZIP archive (Content-Type: application/zip; Content-Disposition: attachment; filename="…";), optionally split into several parts; even if a potentially multi-image request (e.g. obs=#) results in a single image matching all given parameters, the image is still returned in a ZIP archive

GET /download/header?*param*=*value*…
- download the original FITS file header(s) as text file(s), one header card per line
- WCS info is appended to the header when present
- if parameters uniquely identify a single data file (e.g. image=# or radio_obs=#), a single FITS header is returned as text (Content-Type: text/plain); otherwise, headers are returned in a ZIP archive, optionally split into several parts

GET /download/jpg?*param*=*value*…

- download data as JPEG file(s): Content-Type: image/jpeg; Content-Disposition: inline; filename="…";
- applies to optical observations only
- images are optionally preprocessed in the same way as /download/fits
- multiple images are returned in a ZIP archive

GET /download/png?*param*=*value*…

- download data as PNG file(s): Content-Type: image/png; Content-Disposition: inline; filename="…";
- applies to optical observations only
- images are optionally preprocessed in the same way as /download/fits
- multiple images are returned in a ZIP archive

GET /download/movie?*param*=*value*…

- download a sequence of images as an MPEG4 AVI file: Content-Type: application/force-download; Content-Disposition: attachment; filename="movie.*fmt*"; Content-Transfer-Encoding: binary
- applies to optical observations only
- request parameters must identify multiple images, e.g. image=#,#,… (multiple exposures listed explicitly), obs=#… (single or multiple observations containing multiple images)

Request parameters for all download formats

- **image**=*t*#[,*t*#…] or **images**=*t*#[,*t*#…]: requested optical image ID(s) separated by commas; each integer ID is preceded by one of the following characters *t*:
    - **r**: raw optical exposure
    - **m**: master calibration image
    - **w**: Afterglow workspace image
    - **t**: Afterglow temporary image
    - **s**: Afterglow sample image
- **obs**=#[,#…] or **observations**=#[,#…]: requested optical observation ID(s)
- **radio_obs**=#[,#…]: requested radio observation ID(s); only /download/fits and /download/header are supported; all three selectors (**image**, **obs**, and **radio_obs**) may be combined in a single request, and at least one of them must be present
- **layer**=#: for multi-HDU observations (e.g. polarimetry, spectral, or radio), return specific layer (0-based); if omitted, return the whole FITS file for /download/fits or the primary image (layer 0) for other formats
- **total_parts=**#: split the ZIP archive returned into multiple parts and return a single part per request; see also **part**; ignored for single-image requests
- **part=**#: if **total_parts** is set and > 1, this is the number of part to return, starting from 1; ignored for single-image requests

Request parameters pertaining to all download formats but specific to optical observations when multiple images are returned

- **wcs**[=1]: only return images with world coordinate system (WCS) in their headers
- **filter**=*filter_id_or_name*: only return images taken with the specified filter
- **explen**=#.#: only return images taken with the specified exposure length (seconds)

Request parameters specific to radio observations

- **processed**: when present, return specific radio cartographer channel(s)
- **channel**=#: if **processed** is set, return the given radio cartographer channel (**left**, **right**, or **composite**); default: return all three channels

Request parameters pertaining to optical observations and all download formats except “header”

- **reduce**[=1]: bias, dark, and flat correct the requested image(s)
- **reducequiet**[=1]: same as **reduce** except that in the event of a failure during reduction the unreduced image is returned
- **delta**=#.#: maximum separation (in days) allowed between master calibration images and the source image during reduction
- **mbias**=#: override the auto-selection of master bias to be used in calibration by specifying its ID; automatically adds **reduce**=1; **mbias**=0 disables bias correction
- **mdark**=#: override the auto-selection of master dark to be used in calibration by specifying its ID; automatically adds **reduce**=1; **mdark**=0 disables dark correction
- **mflat**=#: override the auto-selection of master flat to be used in calibration by specifying its ID; automatically adds **reduce**=1; **mflat**=0 disables flat correction
- **remove_cosmics**[=1]: remove cosmic rays from the images; defaults to 1 if **reduce**=1 and to 0 otherwise
- **find_cosmics**[=1]: return images containing only cosmic rays present in the original images
- **scale**=#.#: scale images by the given factor
- **width**=#: scale images to the given width
- **height**=#: scale images to the given width

Request parameters pertaining to optical observations and “fits” download format

- **force_int**[=1]: convert FITS images returned to unsigned 16 bit integer

Request parameters pertaining to optical observations and “jpg”, “png”, and “movie” download formats

- **min**=#.#: lower percentile of the histogram used to set the black point of the image
- **max**=#.#: upper percentile of the histogram used to set the white point of the image
- **cmap**=*id*: produce a false-color image or movie; *id* must be one of the colormap names supported by matplotlib; see http://matplotlib.org/users/colormaps.html; default: **cmap**=gray (produce a grayscale image)
- **norm_orient**[=1]: normalize image orientation based on the WCS, flipping it vertically and/or horizontally and swapping the axes to make sure that it is as close as possible to North up, East left
- **horiz_flip**[=1]: flip images horizontally.
- **vert_flip**[=1]: flip images vertically.
- **swap**[=1]: swap image axes.

Request parameters pertaining to optical observations and “jpg” download format

- **quality**=#: quality of JPEG compression

Request parameters pertaining to optical observations and “movie” download format

- **framerate**=#.#: frame rate in frames per second
- **keyframeinterval**=#.#: key frame interval in seconds
- **bitrate**=#.#: bitrate in bits per second
- **format**=*id*: output format (*id* = **avi**, **mov**, or **flv**)

Job API

Observations

```
class JobSchema {
        id: int,
        userId: int,
        apiVersion: str,        // 20
        status: str,            // pending, running, success, failure
        timeCreated: datetime,
        timeCompleted: datetime,
        type: str,              // batch_obs_upload
        args: str,              // JSON
        result: str,            // JSON
```

        user: {id: int, username: str},
}

GET /jobs?*param*=*value*…

- return all user's jobs (list of JobSchema objects) matching certain criteria
    - **offset**: only return items starting from the given index; default: start from 0
    - **limit**: only return the given number of items at maximum; must not exceed 1000; default: 1000
    - **include**: list of fields to return; default: return only ObservationSchema.id if **exclude** is not set, otherwise return all fields except those listed in **exclude**; the sole **include**=* returns all fields
    - **exclude**: list of fields to exclude from serialization; default: return all fields listed in **include**; **exclude** takes precedence if a field is listed both in **include** and **exclude**
    - any other JobSchema fields, e.g. state=running.

GET /jobs/*job_id*

- return the given user's job (JobSchema)

GET /jobs/*job_id*/status

- return job status (JobSchema.status)

GET /jobs/*job_id*/result

- return job result (JobSchema.result) if status=success or error message if status=error

POST /jobs?type=…&args=…

- Submit a job
    - **type**: job type (currently batch_obs_upload)
    - **args**: job-specific arguments as JSON string

Batch-upload observations

POST /jobs?type=batch_obs_upload&args=[{"name": "…", "primaryTelescope": "…", "exps": [{"expLength": …, "filterRequested": "…", …}, …]}, …]   →   job ID

Args are URL-encoded.

GET /jobs/*job_id*/status   →   until status is "success" or "error"

GET /jobs/*job_id*/result   →   JSON containing the list of submitted Observation objects or error message

Python Client

Procedural

Object-oriented (ORM)