SEED Cities API Endpoints

This document lists the API Endpoints as they are documented in the SEED code (dated Aug 17, 2014). Note that not all of these endpoints are not part of the the specified API features list. These endpoints are marked with **.

api.py

Function name	Use	Request	Payload	Returns

get_api_schema	Returns a hash of all API endpoints and their descriptions.			{'/example/url/here': {
				'name': endpoint name,
				'description': endpoint description
				}...
				}

accounts.py

Function name	Use	Request	Payload	Returns
update_role	Sets a user's role within an organization.		{ 'organization_id': organization's id, 'user_id': user's id, 'role': one of 'owner', 'member', 'viewer' }	{'status': 'success or error', 'message': 'error message, if any'}

save_org_settings	Saves an organzation's settings: name, query threshold, shared fields		{ 'organization_id: 2,
			'organization': {
			'query_threshold': 2,
			'name': 'demo org',
			'fields': [# All shared fields
			{ 'sort_column': database/search field name, e.g. 'pm_property_id', }
			],
			}
			}
				{ 'status': 'success or error',
				'message': 'error message, if any' }

get_query_threshold	Returns the "query_threshold" for an org. Searches from members of sibling orgs must return at least this many buildings from orgs they do not belong to, or else buildings from orgs they don't belong to will be removed from the results.	:GET: Expects organization_id in the query string.		{ 'status': 'success', 'query_threshold': The minimum number of buildings that must be returned from a search to avoid squelching non-member-org results. }

get_shared_fields	Retrieves all fields marked as shared for this org tree.	:GET: Expects organization_id in the query string.		{ 'status': 'success',
				'shared_fields': [
				{
				"title": Display name of field,
				"sort_column": database/search name of field,
				"class": css used for field,
				"title_class": css used for title,
				"type": data type of field,
				(One of: 'date', 'floor_area', 'link', 'string', 'number')
				"field_type": classification of field. One of:
				'contact_information', 'building_information',
				'assessor', 'pm',
				"sortable": True if buildings can be sorted on this field,
				}
				...
				]
				}

create_sub_org	Creates a child org of a parent org.		{ 'parent_org_id': ID of the parent org,
			'sub_org': {
			'name': Name of new sub org,
			'email': Email address of owner of sub org, which
			must already exist
			}
			}
				{ 'status': 'success' or 'error',
				'message': Error message, if any,
				'organization_id': ID of newly-created org
				}

get_user_profile	Retrieves the request's user's first_name, last_name, email and api key if exists.			{ 'status': 'success',
				'user': {
				'first_name': user's first name,
				'last_name': user's last name,
				'email': user's email,
				'api_key': user's API key
				}
				}

update_user	Updates the request's user's first name, last name, and email		{ 'user':
			{ 'first_name': :first_name,
			'last_name': :last_name,
			'email': :email
			}
			}
				{ 'status': 'success',
				'user': {
				'first_name': user's first name,
				'last_name': user's last name,
				'email': user's email,
				'api_key': user's API key
				}
				}

main.py

Function name	Use	Request	Payload	Returns

export_buildings	Begins a building export process.		{
			"export_name": "My Export",
			"export_type": "csv",
			"selected_building": [1234,], (optional list of building ids)
			"selected_fields": optional list of fields to export
			"select_all_checkbox": True // optional, defaults to False
			}
				{
				"success": True,
				"status": "success",
				"export_id": export_id; see export_buildings_download,
				"total_buildings": count of buildings,
				}

export_buildings_progress	Returns current progress on building export process.		{"export_id": export_id from export_buildings }	{'success': True, 'status': 'success or error', 'message': 'error message, if any', 'buildings_processed': number of buildings exported }
get_total_number_of_buildings_for_user	Gets a count of all buildings in the user's organizations.			{'status': 'success', 'buildings_count': buildings_count}
get_building	Retrieves a building. If user doesn't belong to the building's org, fields will be masked to only those shared within the parent org's structure.	:GET: Expects building_id and organization_id in query string.		{ 'status': 'success or error', 'message': 'error message, if any', 'building': {'id': the building's id, other fields this user has access to... }, 'imported_buildings': [A list of buildings imported to create this building's record, i the same format as 'building'],

				'compliance_projects': [A list of compliance projects this building has been involved in], 'user_role': role of user in this org, 'user_org_id': the org id this user belongs to }
get_datasets_count	Retrieves the number of datasets for an org.	:GET: Expects organization_id in the query string.		{'status': 'success', 'datasets_count': Number of datasets belonging to this org. }

search_buildings	Retrieves a paginated list of CanonicalBuildings matching search params.		{
			'q': a string to search on (optional),
			'show_shared_buildings': True to include buildings from other
			orgs in this user's org tree,
			'order_by': which field to order by (e.g. pm_property_id),
			'import_file_id': ID of an import to limit search to,
			'filter_params': { a hash of Django-like filter parameters to limit
			query. See seed.search.filter_other_params. If 'project__slug'
			is included and set to a project's slug, buildings will include
			associated labels for that project.
			}
			'page': Which page of results to retrieve (default: 1),
			'number_per_page': Number of buildings to retrieve per page
			(default: 10),
			}
				{
				'status': 'success',
				'buildings': [
				{ all fields for buildings the request user has access to;
				e.g.:
				'pm_property_id': ID of building (from Portfolio Manager),
				'address_line_1': First line of building's address,
				'property_name': Building's name, if any
				...
				}...
				]
				'number_matching_search': Total number of buildings matching search,
				'number_returned': Number of buildings returned for this page
				}

search_building_snapshots	Retrieves a paginated list of		{ 'q': a string to search on (optional),	{ 'status': 'success',

	BuildingSnapshots matching search params.		'order_by': which field to order by (e.g. pm_property_id),
			'import_file_id': ID of an import to limit search to,
			'filter_params': { a hash of Django-like filter parameters to limit
			query. See seed.search.filter_other_params.
			}
			'page': Which page of results to retrieve (default: 1),
			'number_per_page': Number of buildings to retrieve per page
			(default: 10),
			}
				'buildings': [
				{
				'pm_property_id': ID of building (fron Portfolio Manager),
				'address_line_1': First line of building's address,
				'property_name': Building's name, if any
				}...
				]
				'number_matching_search': Total number of buildings matching search,
				'number_returned': Number of buildings returned for this page
				}

save_match	Adds or removes a match between two BuildingSnapshots. Creating a match creates a new BuildingSnapshot with merged data.		{ 'source_building_id': ID of first BuildingSnapshot, 'target_building_id': ID of second BuildingSnapshot, 'create_match': True to create match, False to remove it }	{'status': 'success', 'child_id': The ID of the newly-created BuildingSnapshot containing merged data from the two parents. }
get_PM_filter_by_counts	Retrieves the number of matched and unmatched BuildingSnapshots for a given ImportFile record.	:GET: Expects import_file_id corresponding to the ImportFile in question.		{'status': 'success', 'matched': Number of BuildingSnapshot objects that have matches, 'unmatched': Number of BuildingSnapshot objects with no matches. }
get_column_mapping_suggestions	Returns suggested mappings from an uploaded file's		{'import_file_id': The ID of the ImportRecord to examine}	{'status': 'success', 'suggested_column_mappings': {

	headers to known data fields.			column header from file: [(destination_column, score) ...] ... } 'building_columns': [a list of all possible columns], 'building_column_types': [a list of column types corresponding to building_columns],] }
get_raw_column_names	Retrieves a list of all column names from an ImportFile.		{'import_file_id': The ID of the ImportFile}	{'status': 'success', 'raw_columns': [list of strings of the header row of the ImportFile] }
get_first_five_rows	Retrieves the first five rows of an ImportFile.		{'import_file_id': The ID of the ImportFile}	{'status': 'success', 'first_five_rows': [[list of strings of header row], [list of strings of first data row], ... [list of strings of fourth data row]] }
save_column_mappings	Saves the mappings between the raw headers of an ImportFile and the destination fields in the BuildingSnapshot model. Valid source_type values are found		{ "import_file_id": ID of the ImportFile record, "source_type": Type of source data, e.g. 'Assessed Raw' or 'Portfolio' "mappings": [["destination_field": "raw_field"], #direct mapping ["destination_field2": ["raw_field1", "raw_field2"], #concatenated mapping ...	{'status': 'success'}

	in ``seed.models.SEED_DATA_SOURCES``		] }	

create_dataset	Creates a new empty dataset (ImportRecord).		{
			"name": Name of new dataset, e.g. "2013 city compliance dataset"
			"organization_id": ID of the org this dataset belongs to
			}
				{'status': 'success',
				'id': The ID of the newly-created ImportRecord,
				'name': The name of the newly-created ImportRecord
				}

get_datasets	Retrieves all datasets for the user's organization.		{'organization_id': ID of org to retrieve datasets from}	{'status': 'success', 'datasets': [{'name': Name of ImportRecord, 'number_of_buildings': Total number of buildings in all ImportFiles, 'id': ID of ImportRecord, 'updated_at': Timestamp of when ImportRecord was last modified, 'last_modified_by': Email address o user making last change, 'importfiles': [{'name': Name of associated ImportFile, e.g. 'buildings.csv', 'number_of_buildings': Count of buildings in this file, 'number_of_mappings': Number of mapped headers to fields, 'number_of_cleanings': Number of fields cleaned, 'source_type': Type of file (see source_types), 'id': ID of ImportFile (needed for most operations) }], ... },

				...] }
get_dataset	Retrieves ImportFile objects for one ImportRecord.	:GET: Expects dataset_id for an ImportRecord in the query string.		{'status': 'success', 'dataset': {'name': Name of ImportRecord, 'number_of_buildings': Total number of buildings in all ImportFiles for this dataset, 'id': ID of ImportRecord, 'updated_at': Timestamp of when ImportRecord was last modified, 'last_modified_by': Email address o user making last change, 'importfiles': [{'name': Name of associated ImportFile, e.g. 'buildings.csv', 'number_of_buildings': Count of buildings in this file, 'number_of_mappings': Number of mapped headers to fields, 'number_of_cleanings': Number of fields cleaned, 'source_type': Type of file (see source_types), 'id': ID of ImportFile (needed for most operations) }], ... }, ... }
get_import_file	Retrieves details about an ImportFile.	:GET: Expects an ImportFile ID in the query string.		{'status': 'success', 'import_file': { "name": Name of the uploaded file,

				"number_of_buildings": number of buildings in the file,
				"number_of_mappings": number of mapped columns,
				"number_of_cleanings": number of cleaned fields,
				"source_type": type of data in file, e.g. 'Assessed Raw'
				"number_of_matchings": Number of matched buildings in file,
				"id": ImportFile ID,
				'dataset': {
				'name': Name of ImportRecord file belongs to,
				'id': ID of ImportRecord file belong to,
				'importfiles': [# All ImportFiles in this ImportRecord, with
				# requested ImportFile first:
				{'name': Name of file,
				'id': ID of ImportFile
				}
				...
				]
				}
				}
				}

delete_file	Deletes an ImportFile from a dataset.	:GET: Expects 'file_id' corresponding to an ImportFile in the query string		{'status': 'success' or 'error', 'message': 'error message, if any' }
delete_dataset	Deletes all files from a dataset and the dataset itself.	:GET: Expects 'dataset_id' for an ImportRecord in		{'status': 'success' or 'error', 'message': 'error message, if any' }

		the query string.		
update_dataset	Updates the name of a dataset.		{'dataset': {'id': The ID of the Import Record, 'name': The new name for the ImportRecord } }	{'status': 'success' or 'error', 'message': 'error message, if any' }
save_raw_data	Starts a background task to import raw data from an ImportFile into BuildingSnapshot objects.		{'file_id': The ID of the ImportFile to be saved}	{'status': 'success' or 'error', 'progress_key': ID of background job, for retrieving job progress }
start_mapping	Starts a background task to convert imported raw data into BuildingSnapshots, using user's column mappings.		{'file_id': The ID of the ImportFile to be mapped}	{'status': 'success' or 'error', 'progress_key': ID of background job, for retrieving job progress }
remap_buildings	Re-run the background task to remap buildings as if it hadn't happened at all. Deletes mapped buildings for a given ImportRecord, resets status.		{'file_id': The ID of the ImportFile to be remapped}	{'status': 'success' or 'error', 'progress_key': ID of background job, for retrieving job progress }
start_system_matching	Starts a background task		{'file_id': The ID of the ImportFile to be matched}	{'status': 'success' or 'error',

	to attempt automatic matching between buildings in an ImportFile with other existing buildings within the same org.			'progress_key': ID of background job, for retrieving job progress }
progress	Get the progress (percent complete) for a task.		{'progress_key': The progress key from starting a background task}	{'progress_key': The same progress key, 'progress': Percent completion }

update_building	Manually updates a building's record. Creates a new BuildingSnapshot for the resulting changes.		{'building':
			{
			'pk': The ID of the building,
			'fieldname': 'value'... The rest of the fields in the
			BuildingSnapshot; see get_columns() endpoint for complete
			list.
			}
			}
				{'status': 'success',
				'child_id': The ID of the newly-created BuildingSnapshot
				}

delete_organization_buildings	Starts a background task to delete all BuildingSnapshots in an org.	:GET: Expects 'org_id' for the organization.		{'status': 'success' or 'error', 'progress_key': ID of background job, for retrieving job progress }

projects.py

Function name	Use	Request	Payload	Returns

get_projects	Retrieves all projects for a given organization.	:GET: Expects organization_id in query string.		{'status': 'success',
				'projects': [
				{
				'name': project's name,
				'slug': project's identifier,
				'status': 'active',
				'number_of_buildings': Count of buildings associated with project
				'last_modified': Timestamp when project last changed
				'last_modified_by': {
				'first_name': first name of user tha made last change,
				'last_name': last name,
				'email': email address,
				},
				'is_compliance': True if project is a compliance project,
				'compliance_type': Description of compliance type,
				'deadline_date': Timestamp of wher compliance is due,
				'end_date': Timestamp of end of project
				}
				...
				] }

get_project	Retrieves details about a project.	:GET: Expects the project's identifier (slug) as project_slug in the query string. Expects an organization_id		{ 'name': project's name, 'slug': project's identifier, 'status': 'active', 'number_of_buildings': Count of buildings associated with project 'last_modified': Timestamp when project last changed 'last_modified_by': {

		(to which project belongs) in the query string.		'first_name': first name of user that made last change, 'last_name': last name, 'email': email address, }, 'is_compliance': True if project is a compliance project, 'compliance_type': Description of compliance type, 'deadline_date': Timestamp of when compliance is due, 'end_date': Timestamp of end of project }
delete_project	Deletes a project.		{ 'project_slug': identifier (slug) for the project 'organization_id': ID of the org the project belongs to }	{ 'status': 'success or error', 'message': 'error message, if any' }

create_project	Creates a new project.
	@TODO: What's a compliance_type?
			{ 'organization_id': ID of org to associate new project with,
			'project': {
			'name': name of new project,
			'compliance_type': description of type of compliance,
			'description': description of new project,
			'end_date': Timestamp for when project ends,
			'deadline_date': Timestamp for compliance deadline}
			}
				{ 'status': 'success' or 'error',
				'message': 'error message, if any',
				'project_slug': Identifier of new project, if project successfully created }

add_buildings_to_project	Adds buildings to a project.		{ 'project': { 'project_slug': Identifier of project to add buildings to, 'selected_buildings': A list of building IDs to add to project } }	{ 'status': 'success' or 'error', 'message': 'error message, if any', 'project_loading_cache_key': Identifier for the background job, to determine the job's progress }

remove_buildings_from_project	Removes buildings from a project.		{ 'project': { 'project_slug': Identifier of project to remove buildings from, 'selected_buildings': A list of building IDs to remove } }	{ 'status': 'success' or 'error', 'message': 'error message, if any', 'project_removing_cache_key': Identifier for the background job, to determine the job's progress }
update_project	Updates an existing project's details and compliance info.		{ 'project': { 'project_slug': Identifier of project to update, 'name': new name for project, 'compliance_type': description of type of compliance, 'description': description of new project, 'end_date': Timestamp for when project ends, 'deadline_date': Timestamp for compliance deadline } }	{ 'status': 'success' or 'error', 'message': 'error message, if any' }
get_adding_buildings_to_project_status_percentage	Returns percentage complete of background task for adding building to project.		{ 'project_loading_cache_key': Job identifier from add_buildings_to_project. }	{ 'status': 'success', 'progress_object': { 'percentage_done': percent job done, 'numerator': number buildings added, 'denominator': total number of building to add } }
get_projects_count	Returns the number of projects within the org tree to which a user belongs. Counts projects in parent orgs and sibling orgs.	:GET: Expects organization_id for the user's org.		{ 'status': 'success', 'projects_count': count of projects }

update_project_building	Updates extra information about the building/project relationship. In particular, whether the building is compliant and who approved it.		{ 'project_slug': identifier of project, 'building_id': ID of building, 'label': { 'id': Identifier of label to apply. } }	{ 'status': 'success', 'approved_date': Timestamp of change (now), 'approver': Email address of user making change. }
move_buildings	Moves buildings from one project to another.		{ "buildings": ["00010811", "00010809"], "copy": true to copy the buildings, false to move, "search_params": { "filter_params": { "project__slug": "proj-1" }, "project_slug": 34, "q": "hotels" }, "select_all_checkbox": false, "source_project_slug": "proj-1", "target_project_slug": "proj-2" }	{'status': 'success'}
get_labels	Gets all labels for any organization the user has access to.			{ 'status': 'success', 'labels': [{ 'name': name of label, 'color': color of label, 'id': label's ID }, ...] }

add_label	Creates a new label.		{ 'label':
			{ "color": "red",
			"name": "non compliant"}
			}
				{
				'status': 'success',
				'label_id': The ID of the new label.
				}

update_label	Updates a label.		{ "label": { "color": Label's new color, "id": ID of label to change,	{'status': 'success'}

			"name": Label's new name, } }	
delete_label	Deletes a label.		{'label': {'id': ID of label to delete} }	{'status': 'success'}
apply_label	Applies a label to buildings (within a project).		{ "buildings": ["00010811", "00010809"], "label": {"id": 1 }, "project_slug": "proj-1", "search_params": { "filter_params": { "project__slug": "proj-1" }, "project_slug": 34, "q": "" }, "select_all_checkbox": false }	{'status': 'success'}
remove_label	Removes labels from buildings (within a project).		{ "buildings": ["IMP75-0004N0027"], "project_slug": "proj-1", "search_params": { "filter_params": { "project__slug": "proj-1" }, "project_slug": 34, "q": "" }, "select_all_checkbox": false }	{'status': 'success'}

meters.py**

Do these functions support GreenButton data?

Function name	Use	Request	Payload	Returns
get_meters	Returns all of the meters for a building.	GET building_id		{ 'status':'success', 'building_id': building_id, 'meters':}

add_meter_to_building	Will add a building to an existing meter.		{
			'organization_id': 435,
			'building_id': 342,
			'meter_name': 'Unit 34.',
			'energy_type': 'Electricity',
			'energy_units': 'kWh'
			}
				{'status': 'success'}

get_timeseries	Return all time series data for a building, grouped by meter.	GET meter_id, offset, num		{'status': 'success', 'meter_id': meter_id, 'timeseries': []}

add_timeseries	Add timeseries data for a meter.		{
			'organization_id': 435,
			'meter_id': 34,
			'timeseries': [
			{
			'begin_time': 2342342232,
			'end_time': 23423433433,
			'cost': 232.23,
			}...
			]
			}
				{'status': 'success'}