I'm not a robot

The software catalog backend has a JSON based REST API, which can be leveraged by external systems. This page describes its shape and features. The OpenAPI spec for this API can be found here. A UI visualizing the OpenAPI endpoints including the ability to try them out in the browser can be found here. Overview The API surface consists of a few distinct groups of functionality. Each has a dedicated section below. Note: This page only describes some of the most commonly used parts of the API, and is a work in progress. All of the URL paths in this article are assumed to be on top of some base URL pointing at your catalog installation. For example, if the path given in a section below is /entities, and the catalog is located at during local development, the full URL would be . The actual URL may vary from one organization to the other, especially in production, but is commonly your backend.baseUrl in your app config, plus /api/catalog at the end. Some or all of the endpoints may accept or require an Authorization header with a Bearer token, which should then be the Backstage token returned by the identity API. Entities These are the endpoints that deal with reading of entities directly. What it exposes are final entities - i.e. the output of all processing and the stitching process, not the raw originally ingested entity data. See The Life of an Entity for more details about this process and distinction. GET /entities/by-query Query entities. Supports the following query parameters, described in the section below: filter, for selecting only a subset of all entities fields, for selecting only parts of the full data structure of each entity limit for limiting the number of entities returned (20 is the default) orderField, for deciding the order of the entities fullTextFilter, for filtering the entities by text cursor, for retrieving the next or previous batch of entities The return type is JSON, on the following form { "items": [{ "kind": "Component", "metadata": { "name": "foo" } }], "totalItems": 4, "pageInfo": { "nextCursor": "a-cursor", "prevCursor": "another-cursor" }} Filtering You can pass in one or more filter sets that get matched against each entity. Each filter set is a number of conditions that all have to match for the condition to be true (conditions effectively have an AND between them). At least one filter set has to be true for the entity to be part of the result set (filter sets effectively have an OR between them). Example: /entities/by-query?filter=kind=user,metadata.namespace=default&filter=kind=group,spec.type Return entities that match Filter set 1: Condition 1: kind = user AND Condition 2: metadata.namespace = default OR Filter set 2: Condition 1: kind = group AND Condition 2: spec.type exists Each condition is either on the form , or on the form =. The first form asserts on the existence of a certain key (with any value), and the second asserts that the key exists and has a certain value. All checks are always case insensitive. In all cases, the key is a simplified JSON path in a given piece of entity data. Each part of the path is a key of an object, and the traversal also descends through arrays. There are two special forms: Array items that are simple value types (such as strings) match on a key-value pair where the key is the item as a string, and the value is the string true Relations can be matched on a relations.= form Let's look at a simplified example to illustrate the concept: { "a": { "b": ["c", { "d": 1 }], "e": 7 }} This would match any one of the following conditions: a a.b a.b.c a.b.c=true a.b.d a.b.d=1 a.e a.e=7 Some more real world usable examples: Return all orphaned entities: /entities/by-query?filter=metadata.annotations.backstage.io/orphan=true Return all users and groups: /entities/by-query?filter=kind=user&filter=kind=group Return all service components: /entities/by-query?filter=kind=component,spec.type=service Return all entities with the java tag: /entities/by-query?filter=metadata.tags.java Return all users who are members of the ops group (note that the full reference of the group is used): /entities/by-query?filter=kind=user,relations.memberof=group:default/ops Full text filtering TODO Field selection By default the full entities are returned, but you can pass in a fields query parameter which selects what parts of the entity data to retain. This makes the response smaller and faster to transfer, and may allow the catalog to perform more efficient queries. The query parameter value is a comma separated list of simplified JSON paths like above. Each path corresponds to the key of either a value, or of a subtree root that you want to keep in the output. The rest is pruned away. For example, specifying ?fields=metadata.name,metadata.annotations,spec retains only the name and annotations fields of the metadata of each entity (it'll be an object with at most two keys), keeps the entire spec unchanged, and cuts out all other roots such as relations. Some more real world usable examples: Return only enough data to form the full ref of each entity: /entities/by-query?fields=kind,metadata.namespace,metadata.name Ordering By default the entities are returned ordered by their internal uid. You can customize the orderField query parameters to affect that ordering. For example, to return entities by their name: /entities/by-query?orderField=metadata.name,asc Each parameter can be followed by asc for ascending lexicographical order or desc for descending (reverse) lexicographical order. Pagination You may pass the cursor query parameters to perform cursor based pagination through the set of entities. The value of cursor will be returned in the response, under the pageInfo property: "pageInfo": { "nextCursor": "a-cursor", "prevCursor": "another-cursor" } If nextCursor exists, it can be used to retrieve the next batch of entities. Following the same approach, if prevCursor exists, it can be used to retrieve the previous batch of entities. filter, for selecting only a subset of all entities fields, for selecting only parts of the full data structure of each entity limit for limiting the number of entities returned (20 is the default) orderField, for deciding the order of the entities fullTextFilter NOTE: [filter, orderField, fullTextFilter] and cursor are mutually exclusive. This means that, it isn't possible to change any of [filter, orderField, fullTextFilter] when passing cursor as query parameters, as changing any of these properties will affect pagination. If any of filter, orderField, fullTextFilter is specified together with cursor, only the latter is taken into consideration. GET /entities Lists entities. The endpoint supports the following query parameters, described in sections below: The return type is JSON, as an array of Entity. Filtering You can pass in one or more filter sets that get matched against each entity. Each filter set is a number of conditions that all have to match for the condition to be true (conditions effectively have an AND between them). At least one filter set has to be true for the entity to be part of the result set (filter sets effectively have an OR between them). Example: /entities?filter=kind=user,metadata.namespace=default&filter=kind=group,spec.type Return entities that match Filter set 1: Condition 1: kind = user AND Condition 2: metadata.namespace = default OR Filter set 2: Condition 1: kind = group AND Condition 2: spec.type exists Each condition is either on the form , or on the form =. The first form asserts on the existence of a certain key (with any value), and the second asserts that the key exists and has a certain value. All checks are always case insensitive. In all cases, the key is a simplified JSON path in a given piece of entity data. Each part of the path is a key of an object, and the traversal also descends through arrays. There are two special forms: Array items that are simple value types (such as strings) match on a key-value pair where the key is the item as a string, and the value is the string true Relations can be matched on a relations.= form Let's look at a simplified example to illustrate the concept: { "a": { "b": ["c", { "d": 1 }], "e": 7 }} This would match any one of the following conditions: a a.b a.b.c a.b.c=true a.b.d a.b.d=1 a.e a.e=7 Some more real world usable examples: Return all orphaned entities: /entities?filter=metadata.annotations.backstage.io/orphan=true Return all users and groups: /entities?filter=kind=user&filter=kind=group Return all service components: /entities?filter=kind=component,spec.type=service Return all entities with the java tag: /entities?filter=metadata.tags.java Return all users who are members of the ops group (note that the full reference of the group is used): /entities?filter=kind=user,relations.memberof=group:default/ops Field selection By default the full entities are returned, but you can pass in a fields query parameter which selects what parts of the entity data to retain. This makes the response smaller and faster to transfer, and may allow the catalog to perform more efficient queries. The query parameter value is a comma separated list of simplified JSON paths like above. Each path corresponds to the key of either a value, or of a subtree root that you want to keep in the output. The rest is pruned away. For example, specifying ?fields=metadata.name,metadata.annotations,spec retains only the name and annotations fields of the metadata of each entity (it'll be an object with at most two keys), keeps the entire spec unchanged, and cuts out all other roots such as relations. Some more real world usable examples: Return only enough data to form the full ref of each entity: /entities?fields=kind,metadata.namespace,metadata.name Ordering By default the entities are returned in an undefined, but stable order. You can pass in one or more order query parameters to affect that ordering. Each parameter starts either with asc: for ascending lexicographical order or desc: for descending (reverse) lexicographical order, followed by a dot-separated path into an entity's keys. The ordering is case insensitive. If more than one order directive is given, later directives have lower precedence (they are applied only when directives of higher precedence have equal values). Example: /entities?order=asc:kind&order=desc:metadata.name This will order the output first by kind ascending, and then within each kind (if there's more than one of a given kind) by their name descending. When given a field that does NOT exist on all entities in the result set, those entities that do not have the field will always be sorted last in that particular order step, no matter what the desired order was. Pagination You may pass the offset and limit query parameters to do classical pagination through the set of entities. There is also an after query parameter to return the next page of results after the previous one when performing cursor based pagination. Each paginated response that has a next page of data, will have a Link, rel="next" header pointing to the query path to the next page. Example: Getting the first page: GET /entities?limit=2HTTP/1.1 200 OKlink: ; rel="next"[{"metadata":{... Getting the next page, since we detect the presence of the Link header: GET /entities?limit=2&after=eyJsaW1pdCI6Miwib2Zmc2V0IjoyfQ%3D%3DHTTP/1.1 200 OKlink: ; rel="next"[{"metadata":{... GET /entities/by-uid/ Gets an entity by its metadata.uid field value. The return type is JSON, as a single Entity, or a 404 error if there was no entity with that UID. DELETE /entities/by-uid/ Deletes an entity by its metadata.uid field value. Note: This method of deletion is appropriate for orphaned entities, but not for removal of "live" entities that are actively being updated by a location. Please read below. The most common user flow is that you register a location (see below), and then the catalog keeps itself up to date with that location and the subtree of things that may spawn from it. This means that the catalog is a live-updating view of an actual authoritative data source. If there's something keeping the entity "alive" in the catalog, it will just reappear shortly after deletion with the method described in this section. To properly remove entities, you typically want to instead unregister the location that causes the entity to appear. However if you have an orphaned entity, for example after removing the reference to its file from a Location entity, or if a processor has stopped producing your entity, then this deletion method is appropriate. The return type is always an empty 204 response, whether an entity with this UID existed or not. GET /entities/by-name/// Gets an entity by its kind, metadata.namespace, and metadata.name field value. These are special in that they form the entity's unique reference triplet. The return type is JSON, as a single Entity, or a 404 error if there was no entity with that reference triplet. GET /entities/by-name/{kind}/{namespace}/{name}/ancestry Get an entity's ancestry by entity ref. POST /entities/by-refs Gets a batch of entities by their entity refs. This is useful in contexts where you want to fetch a large number of specific entities efficiently, for example in GraphQL resolvers. The request body is JSON, on the form { "entityRefs": ["component:default/foo", "api:default/bar"], "fields": ["kind", "metadata.name"]} where each entityRefs entry is an entity ref that you want to fetch. The fields array is optional and works the same way as the GET /entities fields above, e.g. it's used to fetch only certain slices of each entity. The return type is JSON, on the form { "items": [{ "kind": "Component", "metadata": { "name": "foo" } }, null]} where the items array has the same length and the same order as the input entityRefs array. Each element contains the corresponding entity data, or null if no entity existed in the catalog with that ref. POST /refresh Refresh the entity related to entityRef. Request body is JSON, on the form { "entityRef": ""} POST /validate-entity Validate that a passed in entity has no errors in schema. Request body is JSON, on the form { "location": "", "entity": {}} Locations GET /locations Lists locations. Response type is JSON, on the form [{ "data": { "id": "b9784c38-7118-472f-9e22-5638fc73bab0", "target": " , "type": "url" } }] GET /locations/{id} Gets a location by it's location ID. Response type is JSON, on the form { "id": "b9784c38-7118-472f-9e22-5638fc73bab0", "target": " , "type": "url"} GET /locations/by-entity/{kind}/{namespace}/{name} Gets a location referring to a given entity. Response type is JSON, on the form { "id": "b9784c38-7118-472f-9e22-5638fc73bab0", "target": " , "type": "url"} GET /entity-facets?facet=&facet=&filter=&filter= Get all entity facets that match the given filters. Response type is JSON, on the form { "facets": [{ "value": "", "count": 1 }]} POST /locations Adds a location to be ingested by the catalog. If successful the response code will be HTTP/1.1 201 Created and a JSON on the form { "entities": [], "location": { "id": "b9784c38-7118-472f-9e22-5638fc73bab0", "target": " , "type": "url" }} If the location already exists the response will be HTTP/1.1 409 Conflict and a JSON on the form { "error": { "message": "Location url: already exists", "name": "ConflictError", "stack": "ConflictError: Location url: already exists..." }, "request": { "method": "POST", "url": "/locations" }, "response": { "statusCode": 409 }} Supports the ?dryRun=true query parameter, which will perform validation and not write anything to the database. In the event of successfully passing validation, the entities field of the response JSON will be populated with entities present in the location. POST /analyze-location Validate a given location. Request body is JSON, on the form { "location": { "type": "", "target": "" }, "catalogFileName": ""} And Response type is JSON, on the form { "generateEntities": [{ "fields": [{ "description": "", "value": "", "state": "needsUserInput", "field": "" }, { "description": "", "value": {}, "state": "analysisSuggestedNoValue", "field": "" }], "entity": {} }], "existingEntityFiles": [{ "entity": "", "isRegistered": "", "location": { "target": "", "type": "" } }]} DELETE /locations/{id} Delete a location by its id. On success response code will be HTTP/1.1 204 No Content. Other TODO Find thousands of job opportunities by signing up to eFinancialCareers today. You can't perform that action at this time. A hands-on walk-through of installing and configuring a Backstage App for our use.What is Backstage?Backstage is an open platform for building developer portals. Powered by a centralized software catalog, Backstage restores order to your microservices and infrastructure and enables your product teams to ship high-quality code quickly — without compromising autonomy.Backstage unifies all your infrastructure tooling, services, and documentation to create a streamlined development environment from end to end.— What is Backstage?Why Backstage?Getting StartedThe Backstage Getting Started document provides us with a Backstage App (project) with sample data. To understand the sample data, it is useful to examine the entity relationship diagram (ERD) and entities for Backstage.There are the three "leaf" entities: Components, APIs, and Resources:A component is a piece of software, for example a mobile feature, web site, backend service or data pipeline (list not exhaustive). A component can be tracked in source control, or use some existing open source or commercial software.APIs form an important (maybe the most important) abstraction that allows large software ecosystems to scale. Thus, APIs are a first class citizen in the Backstage model and the primary way to discover existing functionality in the ecosystem.Resources are the infrastructure a component needs to operate at runtime, like BigTable databases, Pub/Sub topics, S3 buckets or CDNs. Modelling them together with components and systems will better allow us to visualize resource footprint, and create tooling around them.The System and Domain entities are used in organizing the "leaf" entities.With increasing complexity in software, systems form an important abstraction level to help us reason about software ecosystems. Systems are a useful concept in that they allow us to ignore the implementation details of a certain functionality for consumers, while allowing the owning team to make changes as they see fit (leading to low coupling).While systems are the basic level of encapsulation for related entities, it is often useful to group a collection of systems that share terminology, domain models, metrics, KPIs, business purpose, or documentation, i.e. they form a bounded context.— System ModelThe Software Template entity does not have any relationships to other entities, but is rather used in the process to create Components.The Software Templates part of Backstage is a tool that can help you create Components inside Backstage. By default, it has the ability to load skeletons of code, template in some variables, and then publish the template to some locations like GitHub or GitLab.— Backstage Software TemplatesThe User and Group entities are used in building ownership relationships to other entities.An ownership relation where the owner is usually an organizational entity (User or Group), and the other entity can be anything.In Backstage, the owner of an entity is the singular entity (commonly a team) that bears ultimate responsibility for the entity, and has the authority and capability to develop and maintain it. They will be the point of contact if something goes wrong, or if features are to be requested. The main purpose of this relation is for display purposes in Backstage, so that people looking at catalog entities can get an understanding of to whom this entity belongs. It is not to be used by automated processes to for example assign authorization in runtime systems. There may be others that also develop or otherwise touch the entity, but there will always be one ultimate owner.This relation is commonly generated based on spec.owner of the owned entity, where present.— Well-known Relations between Catalog EntitiesThe remaining entity, Locations, does not appear to have any relationship with other entities and the documentation on it is scarce.A Location reference that points to the source code of the entity (typically a Component). Useful when catalog files do not get ingested from the source code repository itself.— Well-known Annotations on Catalog EntitiesGetting Started, configuring BackstageThe Backstage Getting Started, configuring Backstage document walks us through several common configuration tasks.Install and configure PostgreSQLIn the Getting Started document, we used a temporary SQLite database to support the Backstage App; in this document we transition to using a permanent PostgreSQL database.Rather than installing PostgreSQL onto our workstation, it is easier to run it in a Docker container. Here we can create a file, docker-compose.yml, in the Backstage App folder. This file is based off the instructions provided for the official postgres Docker image; except here we also expose the PostgreSQL database to our workstation on port 5432.eWe start PostgresSQL by typing:$ docker-compose up -dWe can continue through the document through updating the app-config.yaml file with the updated database configuration. Because it bad practice to commit secrets into source control, we can create a new file, environment.sh, in the Backstage App folder as follows and add it to the .gitignore file in the same folder.We can set the environment variables and start the Backstage App using the following commands.$ source environment.sh$ yarn devGoing forward, when we start the Backstage App, we will use these commands.Setting up authenticationFollowing the document, we will use GitHub for authentication. Here, however, we use a GitHub Organization account instead of a Personal account.Your team can collaborate on GitHub by using an organization account. Each person that uses GitHub signs into a user account. Multiple user accounts can collaborate on shared projects by joining the same organization account, which owns the repositories. A subset of these user accounts can be given the role of organization owner, which allows those people to granularly manage access to the organization's resources using sophisticated security and administrative features.— About organizationsWhen creating the GitHub OAuth App, we be sure to use localhost instead of 127.0.0.1 in the URLs; the document is inconsistent in this regard.Also instead of adding the Client Id and Secret to the app-config.yaml, we rather add it to environment.sh file as follows.We then update app-config.yaml; updating the root auth key.The last step, a bit unusual, requires us to modify the React code to support GitHub authentication. Once done, we start the Backstage App and find that we have to authenticate now.Observations:We first observe that by using the Settings menu option, we can see who we are logged in with; presumably seeing our name (and maybe avatar image)The list of sample User entities does not list ourselvesIt is easy to miss, but before we enabled authentication we saw that we owned a number of Components as we were associated with the guest User entity. After we enabled authentication, we own no Components; thus we are not associated with the guest User entitySetting up a GitHub IntegrationContinuing to follow the document, we setup an integration to our GitHub organization. As we have a GitHub Organization account, we will use a GitHub App instead of a Personal Access Token.Backstage can be configured to use GitHub Apps for backend authentication. This comes with advantages such as higher rate limits and that Backstage can act as an application instead of a user or bot account.— GitHub Apps for Backend AuthenticationWe follow the steps in Using the CLI (public GitHub only) and Including in Integrations Config in the documentation; GitHub Apps for Backend Authentication.We, additionally, update the GitHub App permissions.App permissions is not managed by Backstage. They're created with some simple default permissions which you are free to change as you need, but you will need to update them in the GitHub web console, not in Backstage right now. The permissions that are defaulted are metadata:read and contents:read.— GitHub Apps for Backend AuthenticationThe full list of required App permissions is provided.Login and Backstage and check profileAfter starting the Backstage App, we can finish up by following the instructions Login to Backstage and check profile.Next StepsIn the next article, Backstage by Example (Part 2), we clean-up the Backstage App for our use. You can't perform that action at this time.