Saraki Documentation

Release 0.1.0a0

José María Domínguez Moreno

Dec 12, 2020

Contents:

Welcome to this documentation. Saraki is a framework for multi-tenant application.

CHAPTER 1

Quickstart

1.1 A Minimal Application

Since Saraki is just Flask, a basic app looks exactly the same way with the difference that we must use the Saraki class:

```
from saraki import Saraki
app = Saraki(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "postgresql://user:pass@hostname/db"
```

We haven't done any special yet. we just created an app instance and set up our database URI, but if we run the application we are going to get out of the box an API with the next features:

- User signup.
- Multiple organization accounts (tenant) per user.
- Organization members (memberships).
- Role management per organization.
- Authentication and authorization.

Now, let create a `Todo` class which will store to-do lists for each organization account.

```
from sqlalchemy import Column, ForeignKey, Integer, String

class Todo(Model):
    id = Column(Integer, primary_key=True)
    task = Column(String)
    org_id = Column(Integer, ForeignKey("org.id"))
```

This is just another SQLAlchemy declarative base class. The only important thing here is the column **org_id**. This column will tell to Saraki that this entity is going to store multi-tenant data.

Now let create a tenant endpoint to access a to-do list per organization account.

```
from saraki.auth import require_auth
from saraki.endpoints import collection

@app.route('/orgs/<aud:orgname>/todos')
@require_auth()
@collection()
def list():
    return Todo
```

Let's talk about what we did in the above code:

1. First, we added a route rule with a special converter **aud**. This converter will define the tenant accessed in the current request. So, a request to `/orgs/acme/todos` means that we are asking for data from the Acme organization.
2. Then we use the `require_auth()` decorator, which will validate HTTP requests looking for a valid access token. This decorator is mandatory for all tenant endpoints since it checks that an access token corresponds to the organization account accessed.
3. We use the *`collection()`* decorator. This will handle operations such as filtering and sorting, but more importantly, it will ensure that a query to the database is properly segregated by filtering the results by the column **org_id**.
4. And finally, we just return the model class to let the collection decorator handle it.

We have not talked about how to insert, update and delete data until now. Each of these operations can be implemented normally as you would in any other application based on Flask and SQLAlchemy, for example, an endpoint to add new records would look like this:

```
from saraki.auth import current_org
from saraki.model import database

@app.route('/orgs/<aud:orgname>/todos', methods=["POST"])
@require_auth()
def add_todo():
    todo = Todo()
    todo.task = "Stop being lazy"
    todo.org_id = current_org.id

    database.session.add(todo)
    database.session.commit()

    return "", 201
```

When a request is send to a tenant endpoint, the local proxy `current_org` is available and points to the current organization being accessed.

1.2 Protecting endpoints

Every application will have one or more endpoints that should not be open to the public. The way we protect an endpoint from unauthorized access is by requiring a token on each HTTP request.

Use the `require_auth()` decorator to protect an endpoint.

```
@app.route('/chat')
@require_auth()
```

(continues on next page)

(continued from previous page)

```
def hello_world():
    return "Messages of this chat"
```

The above snippet is the most basic way of protecting an endpoint. At the minimum, it will require someone to sigup first and then get an access token previous authentication. It doesn't specify any authorization constraint so it won't check the scope of the access token in the current request.

To learn how to add authorization constraints read the *Authorization* documentation.

CHAPTER 2

Authorization

Saraki uses an **ability based authorization** mechanism to determine if a given user can access to an endpoint. This mechanism is composed of **resources**, **actions**, **abilities**, and **roles**. On each HTTP request, a client must provide an access token with enough privileges (abilities) to perform a given action on a given resource.

Before we start with examples and usage information, let define some concepts and terms:

- **Resource**: It is any unit or group of data accessible through an API. To all the resources we want to be protected we assign a single name to them.
- **Action**: An action is any type of operation that can be performed on a resource. We must give a name to the action or task that an API endpoint performs. Most of the time it will be one of the classic CRUD operations; create, read, update and delete, but it can be any name, for instance, follow or listen, for a service that propagates information using WebSockets.
- **Ability:** The ability to perform an action on a resource. For instance; read products, create products, etc. It is basically just a resource/action pair. But you can add a name and description to it too.
- **Role**: A set of one or more abilities. For example, a role Cashier could have the abilities "read payment", "create payment" or a role Seller can have the abilities "read product", "read order", "update order", "delete order". A user can have various roles assigned to him.

Saraki uses *JSON Web Token* and stores the privileges that a user has as a member of given organization in the token payload.

2.1 How it works

Assuming we have an endpoint decorated with `require_auth()`, the way a request is validated against an endpoint happens in this way:

1. First, look for a valid access token in the incoming request.
2. Then check if the variable converters match the claims of the current access token.
3. Finally, check if the scope of the token has the required privileges defined in `require_auth()`.

If any of those steps fail, the application won't execute the view function and will respond with 401 Unauthorized status code.

2.2 Authorization rules

The way we define authorization rules on a view function is passing the name of the **resource** and the **action** required to the `require_auth()` decorator.

The `require_auth()` decorator plays an important role here because it collects all resources and actions used by the application to latter save then in the database.

Take into account the next code:

```
@app.route("/products")
@require_auth("product")
def list_products():
    return []
```

In the above code, we define that a token must contain the `product` resource explicitly and the action `read` implicitly. By implicitly we mean that if an action name is not provided, the actual route rule HTTP method (GET in this case) will be mapped to a predefined action (read in this case). So an access token with the next payload would be able to perform a GET request to the above-defined endpoint.

```
{
    "sub":"coyote",
    "scp": {
        "product": ["read"]
    },
}
```

Here the list of predefined action/method mapping:

Method	Action
GET	read
POST	write
PATCH	write
DELETE	delete

Let's see three more examples to fully understand how this work:

```
@app.route("/products", method=["POST"])
@require_auth("product")
def add_product():
    pass

@app.route("/products/:id", method=["PATCH"])
@require_auth("product", "update")
def update_product():
    pass

@app.route("/products:/id", method=["DELETE"])
@require_auth("product")
def delete_product():
    pass
```

1. The first view function requires an access token with the scope `"product": ["write"]`. The required action is **write** because the method to which the route listen is **POST**.
2. The second view function passes a custom action name `update`, so it will require a scope equal to `"product": ["update"]`. Note that the required action is `update` and not `write` anymore.
3. And the last one requires `"product": ["delete"]` because the HTTP method is **DELETE**.

The next access token scope should be able to perform a request to any of the three defined endpoints above:

```
{
    "sub":"coyote",
    "scp": {
        "product": ["read", "write", "update", "delete"]
    },
}
```

2.3 Variable Converters

Another way of adding authorization constraints are the route rule variable converters. They are very important because they will help the application segregate the data access between tenant in the database. Currently, there are two converts:

converter	value
sub	username. The user account.
aud	orgname. The organization account.

When one of those variable converters appears in a route rule, the authorization mechanism will ensure that the current access token claims match the variable values of the current URL.

Suppose we have a view function with the route rule `/users/<sub:username>/activity`, and an incoming request to `/users/coyote/activity`. For the request to be successful the access token must have the **sub** claim with the value **coyote**.

```
{"sub":"coyote"}
```

If the request is successful, the local proxy `current_user` is available. This object always points to the user performing the current request.

The **aud** converter works in exactly the same way, there is no difference. Let's use both of them in a single route rule:

```
from saraki.auth import current_org, current_user

@app.route("/orgs/<aud:organame>/members/<sub:username>/activity")
@require_auth()
def index(organame, username):
    # your code here
```

In the above code we imported `current_org` which will point to the current organization being accessed.

A request to `/orgs/acme/users/coyote/activity` must have a token with the next payload:

```
{"aud": "acme", "sub":"coyote"}
```

The local proxies `current_org` and `current_user` must be used to ensure that operations to the database are made on the correct organization and user account. So organizations do not end up reading or modifying data from other organizations.

2.4 Access token

Currently, the only supported token format is JSON Web Token. You are going to find a lot of documentation about JWT on the internet, so we are not going to cover the specification here.

There are two types of access token:

1. **User access token:** This token give access to protected endpoints which aren't tenant endpoints. It also gives access to endpoints which handles user-specific data. These type of endpoints usually has the **sub** *converter*.
2. **Org access token:** Gives access to tenant-specific endpoints. Those are endpoints which have the **aud** *converter*.

A JSON Token transport key/value pairs as payload. Here a list of important claims that you should be aware of:

- **sub**: This is the username to which a token belongs. This is always present.
- **aud**: This is the organization to which this token has access. What this means is that a token that belongs to an organization can not access endpoints that belong to other organizations.
- **scp**: This is the scope in which a token can operate. It stores the privileges of a user in a dictionary. The properties are the resources and the values are a list of actions that can be performed on the resource.

Here a JWT payload that illustrates with the three claims above listed.

```
{
    "aud": "acme",
    "sub":"coyote",
    "scp": {
        "catalog": ["read"],
        "sale": ["read", "write", "delete"]
    }
}
```

CHAPTER 3

Configuration

The following configuration values are used internally. Some of them can be configured using environment variables.

SECRET_KEY
: It is used to cryptographically sign each JSON Web Token. Beside that, it is used to securely sign session cookies. This is mandatory for the authorization mechanism to work.

  This can be setup with the `SRK_SECRET_KEY` environment variable.

  Default: `None`

SQLALCHEMY_DATABASE_URI
: The database URI where this app should connect. This can be setup with the `SRK_DATABASE_URI` environment variable. Below an example:

```
postgresql://coyote:12345@localhost/mydatabase
```

  Default: `None`

SERVER_NAME
: This can be setup with the `SRK_SERVER_NAME` environment variable.

  Default: `None`

JWT_ALGORITHM
: The digital signature algorithm used to sign JWTs. Under the hood, PyJWT is used to generate the tokens, so read the documentation to see what cryptographic algorithms are available.

  Default: `'HS256'`

JWT_LEEWAY
: Default: `timedelta(seconds=10)`

JWT_EXPIRATION_DELTA
: Defaul: `timedelta(seconds=300)`

JWT_AUTH_HEADER_PREFIX
: The prefix for the `Authorization` request header. If the value of this header in the current request has a different prefix the toke will be considered invalid.

Default: `'JWT'`

JWT_ISSUER
This value is used to setup the `iss` claim of JSON Web Tokens.

Default to the value of *SERVER_NAME*, otherwise `None`.

JWT_REQUIRED_CLAIMS
A list of required claims in a JWT. If one of them is not present, the token will be considered invalid.

Default: `["exp", "iat", "sub"]`

CHAPTER 4

API

4.1 The Current Account

There are two types of accounts; **user accounts** and **organization accounts**, The user making a request and the tenant being accessed are available throguth *current_user* and *current_org*.

saraki.**current_user**
: A local proxy object that points to the user accessing an endpoint in the current request. The value of this object is an instance of the model class *User* or None if there is not a user.

saraki.**current_org**
: A local proxy object that points to the tenant being accessed in the current request. The value of this object is an instance of the model class *Org* or None if the endpoint is not a tenant endpoint.

Note: *current_user* and *current_org* are available only on endpoints decorated with *require_auth()*.

4.2 Authorization

saraki.**require_auth**(*resource=None*, *action=None*, *parent_resource=None*)
: Decorator to restrict view function access only to requests with enough authorization.

  A valid request must meet the following conditions:

  1. The request header must have the `Authorization` header with a valid JSON Web Token.
  2. The token `sub` claim must contain a username registered in the application. If `aud` claim is present the value must be an orgname also registered in the application.
  3. The token scope must have enough privileges to access the view function being accessed.

  If the parameter **resource** is not provided, the token scope won't be verified.

The **resource** parameter locks an endpoint to access tokens that contain that resource or any other parent resource in their `scp` claim. Let's look to at an example to illustrate how this work:

```
@require_auth("cartoon")
def view_cartoons():
    pass

@require_auth("movie", parent_resource="catalog")
def view_movies():
    pass

@require_auth("comic")
def view_comics():
    pass
```

And a hyipothetical access token `scp` claim:

```
{
    "catalog": ["read"],
    "cartoon": ["read"]
}
```

The above access token would be authorized to access to `view_cartoons` and `view_movies` but not to `view_comics`. In the case of `view_cartoons`, the resource `cartoon` is present in the token scope. The resource `movie` is not present but `catalog` which is a parent of it is present, so that's why `view_movies` can be accessed. `view_comics` is not accessible because neither `comic` nor a parent of it is present.

The **action** parameter locks the endpoint to a specific action, for instance, read, create, update, delete, etc. If this parameter is omitted, the HTTP method of the route endpoint definition will be used:

```
@app.route('/friends')
@require_auth('private', 'follow')
def endpoint_handler():
    pass

@app.route('/friends', methods=['DELETE'])
@require_auth('private')
def endpoint_handler():
    pass
```

The first example above, requires the resource *private* with *follow* action like the example below:

```
{"private": ["follow"]}
```

The second example:

```
{"resource": ["delete"]}
```

The last argument `parent_resource` is optional. It defines the parent resource of the endpoint. That means that if an access token has a resource matching the parent resource, but not the required resource, it still pass the validation. For instance, `@require_auth('resource', 'action', parent='parent')` will pass with the next access token:

```
{"parent": ["action"]}
```

Whenever a request with an unauthorized access token reaches a locked view function an *`AuthorizationError`* exception is raised.

Parameters

- **resource** – The name of the resource
- **action** – The action that can be performed on the resource.
- **parent_resource** – The parent resource.

4.3 Endpoints

saraki.endpoints.**json**(*func*)

Decorator for view functions to return JSON responses.

When the incoming request is a POST request, it validates the content_type and payload before calling the view function. Next, the returned value of the view function is transformed into a JSON response.

The view function can return the response payload, status code and headers in various forms:

1. A single object. Can be any JSON serializable object, a Flask Response object, or a SQLAlchemy model:

```
return {}

return make_response(...)  # custom Response

return Mode.query.filter_by(prop=prop).first()  # SQLAlchemy model instance

return []

return "string response"
```

2. A tuple in the form **(payload, status, headers)**, or **(payload, headers)**. The payload can be any python built-in type, or a SQLAlchemy based model object.:

```
# payload, status

return {}, 201

return [], 201

return '...', 400

# payload, status, headers

return {}, 201, {'X-Header': 'content'}

# payload, headers

return {}, {'X-Header': 'content'}
```

saraki.endpoints.**collection**(*default_limit=30*, *max_limit=100*)

Decorator to handle collection endpoints. This is an instance of *Collection* so head on to that class to learn more how to use it.

saraki.endpoints.**add_resource**(*app*, *modelcls*, *base_url=None*, *ident=None*, *methods=None*, *secure=True*, *resource_name=None*, *parent_resource=None*)

Registers a resource and generates API endpoints to interact with it.

The first parameter can be a Flask app or a Blueprint instance where routes rules will be registered. The second parameter is a SQLAlchemy model class.

Let start with a code example:

```
class Product(Model):
    __tablename__ = 'product'

    id = Column(Integer, primary_key=True)
    name = Column(String)

add_resource(Product, app)
```

The above code will generate the next route rules.

Route rule	Method	Description
/product	GET	Retrive a collection
/product	POST	Create a new resource item
/product/<int:id>	GET	Retrieve a resource item
/product/<int:id>	PATCH	Update a resource item
/product/<int:id>	DELETE	Delete a resource item

By default, the **name** of the table is used to render the resource list part of the url and the name of the **primary key** column for the resource identifier part. Note that the type of the column is used when possible for the route rule variable type.

If the model class has a composite primary key, the identifier part are rendered with each column name separated by a comma.

For example:

```
class OrderLine(Model):
    __tablename__ = 'order_line'

    order_id = Column(Integer, primary_key=True)
    product_id = Column(Integer, primary_key=True)

add_resource(OrderLine, app)
```

The route rules will be:

```
/order-line
/order-line/<int:order_id>,<int:product_id>
```

Note that the character (_) was sustituted by a dash (-) character in the base url.

To customize the base url (resource list part) use the `base_url` parameter:

```
add_resource(app, Product, 'products')
```

Which renders:

```
/products
/products/<int:id>
```

By default, all endpoints are secured with *require_auth()*. Once again, the table name is used for the resource parameter of *require_auth()*, unless the `resource_name` parameter is provided.

To disable this behavior pass `secure=False`.

Model classes with a property (column) named `org_id` will be considered an organization resource and will generate an organization endpoint. For instance, supposing the model class Product has the property org_id the generated route rules will be:

```
/orgs/<aud:orgname>/products
/orgs/<aud:orgname>/products/<int:id>
```

Notice

If you pass `secure=False` and an organization model class, *current_org* and *current_user* won't be available and the generated view functions will break.

Parameters

- **app** – Flask or Blueprint instance.
- **modelcls** – SQLAlchemy model class.
- **base_url** – The base url for the resource.
- **ident** – Names of the column used to identify a resource item.
- **methods** – Dict object with allowd HTTP methods for item and list resources.
- **secure** – Boolean flag to secure a resource using require_auth.
- **resource_name** – resource name required in token scope to access this resource.

class saraki.endpoints.**Collection**

Creates a callable object to decorate collection endpoints.

View functions decorated with this decorator must return an SQLAlchemy declarative class. This decorator can handle filtering, search, pagination, and sorting using HTTP query strings.

This is implemented as a class to extend or change the format of the query strings. Usually, you will need just one instance of this class in the entire application.

Example:

```
# First create a instance
collection = Collection()

@app.route('/products')
@collection()
def index():
    # return a SQLAlchemy declarative class
    return Product
```

4.4 Model

Saraki implements a set of predefined entities where all the application data is stored, such as users, organizations, roles, etc.

Under the hood, Flask-SQLAlchemy is used to manage sessions and connections to the database. A global object *database* is already created for you to perform operations.

saraki.model.**database**

Global instance of SQLAlchemy

class saraki.model.**Model**(***kwargs*)

Abstract class from which all your model classes should extend.

class saraki.model.**Plan**(***kwargs*)
Available plans in your application.

id
Primary key

name
A name for the plan. For instance, Pro, Business, Personal, etc.

amount_of_members
The amount of members that an organization can have.

price
Price of the plan.

class saraki.model.**User**(***kwargs*)
User accounts.

id
Primary key

email
Email associated with the account. Must be unique.

username
Username associated with the account. Must be unique.

canonical_username
Lowercase version of the username used for authentication.

Don't set this column directly. This column is filled automatically when the *username* column is assigned with a value.

password
The password is hashed under the hood, so set this with the original/unhashed password directly.

active
This property defines if the user account is activated or not. To use when the user verifies its account through an email for instance.

class saraki.model.**Org**(***kwargs*)
Organization accounts.

This table registers all organizations being managed by the application and owned by at least one user account registered in the *Membership* table.

id
Primary Key.

orgname
The organization account name.

name
The name of the organization.

user_id
The primary key of the user that created the organization account. But, this account not necessarily is the owner of the organization account, just the user that registered the organization. See the table `Member` for more information.

plan_id
Plan selected from the *Plan* table.

class saraki.model.**Membership**(***kwargs*)
Users accounts that are members of an Organization.

Application users who belong to an organization are considered members, including the owner of the account. This table is a many to many relationship between the tables *User* and *Org*.

user_id
The ID of a user account in the table *User*.

org_id
The ID of an organization account in the table *Org*.

is_owner
If this is True, this member is the/an owner of this organization. One or more members can be owner at the same time.

enabled
Enable or disable a member from an organization.

class saraki.model.**Action**(***kwargs*)
Actions performed across the application like manage, create, read, update, delete, follow, etc.

This table stores all actions registered using `require_auth()`.

class saraki.model.**Resource**(***kwargs*)
Application resources.

id
Primary Key.

name
The name of the resource.

description
A useful description, please.

parent_id
Parent resource.

class saraki.model.**Ability**(***kwargs*)
An ability represents the capacity to perform an action (create, read, update, delete) on a resource/module/service of an application. In other words is an action/resource pair.

This table is used to define those pairs, give them a name and a useful description.

action_id
Foreign key. References to the column `id` of the table *Action*.

resource_id
Foreign key. References to the column *id* of the table *Resource*.

name
A name for the ability. For instance. Create Products.

description
A long text that describes what this ability does.

class saraki.model.**Role**(***kwargs*)
A Role is a set of **abilities** that can be assigned to organization members, for example, Seller, Cashier, Driver, Manager, etc.

This table holds all roles of all organizations accounts, determining the organization that owns the role by the *Org* identifier in the column *org_id*.

Since the roles of all organizations reside in this table, the column *name* can have repeated values. But a role name must be unique in each organization.

id
Primary Key.

name
A name for the role, Cashier for example.

description
A long text that describes what this role does.

org_id
The *id* of the organization account to which this role belongs.

class saraki.model.**RoleAbility**(***kwargs*)

class saraki.model.**MemberRole**(***kwargs*)
All the roles that a user has in an organization.

This table have two composite foreign keys:

- (*org_id*, *user_id*) references to *Membership* (*org_id*, *user_id*).
- (*org_id*, *role_id*) references to *Role* (*org_id*, user_id).

Those two composite foreign keys ensure that the user to which a role is assigned indeed is a member of the organization.

org_id
Foreign key. Must be present in the tables *Membership* and *Role*.

user_id
Foreign key with *user_id* from the table *Membership*.

role_id
Foreign key. *Role.id* from the table *Role*.

4.5 Utility

saraki.utility.**import_into_sqla_object**(*model_instance*, *data*)
Import a dictionary into a SQLAlchemy model instance. Only those keys in *data* that match a column name in the model instance are imported, everthing else is omitted.

This function does not validate the values coming in *data*.

Parameters

- **model_instance** – A SQLAlchemy model instance.
- **data** – A python dictionary.

saraki.utility.**export_from_sqla_object**(*obj*, *include=()*, *exclude=()*)
Converts SQLAlchemy models into python serializable objects.

This is an instance of *ExportData* so head on to the *__call__()* method to known how this work. This instances globally removes columns named org_id.

saraki.utility.**generate_schema**(*model_class*, *include=()*, *exclude=()*, *exclude_rules=None*)
Inspects a SQLAlchemy model class and returns a validation schema to be used with the Cerberus library. The schema is generated mapping column types and constraints to Cerberus rules:

Cerberus Rule	Based on
type	SQLAlchemy column class used (String, Integer, etc).
readonly	**True** if the column is primary key.
required	**True** if `Column.nullable` is **False** or `Column.default` and `Column.server_default` **None**.
unique	Included only when the `unique` constraint is `True`, otherwise is omitted: `Column(unique=True)`
default	Not included in the output. This is handled by SQLAlchemy or by the database engine.

Parameters

- **model_class** – SQLAlchemy model class.
- **include** – List of columns to include in the output.
- **exclude** – List of column to exclude from the output.
- **exclude_rules** – Rules to be excluded from the output.

class saraki.utility.**ExportData**(*exclude=()*)

Creates a callable object that convert SQLAlchemy model instances to dictionaries.

__call__(*obj*, *include=()*, *exclude=()*)

Converts SQLAlchemy models into python serializable objects. It can take a single model or a list of models.

By default, all columns are included in the output, unless a list of column names are provided to the parameters `include` or `exclude`. The latter has precedence over the former. Finally, the columns that appear in the `excluded` property will be excluded, regardless of the values that the parameters include and exclude have.

If the model is not persisted in the database, the default values of the columns are used if they exist in the class definition. From the example below, the value False will be used for the column active:

```
active = Column(Boolean, default=False)
```

Parameters

- **obj** – A instance or a list of SQLAlchemy model instances.
- **include** – tuple, list or set.
- **exclude** – tuple, list or set.

exclude = None

A global list of column names to exclude. This takes precedence over the parameters `include` and/or `exclude` of this instance call.

4.6 Exceptions

exception saraki.exc.**AuthenticationError**

exception saraki.exc.**AuthorizationError**

exception saraki.exc.**InvalidMemberError**

exception saraki.exc.**InvalidOrgError**

exception saraki.exc.**InvalidPasswordError**

exception saraki.exc.**InvalidUserError**

exception saraki.exc.**JWTError**

exception saraki.exc.**NotFoundCredentialError**

Raised when a token or a username/password pair can not be found in the current HTTP request.

exception saraki.exc.**ProgrammingError**

exception saraki.exc.**TokenNotFoundError**

exception saraki.exc.**ValidationError**(*errors*)

CHAPTER 5

History

5.1 0.1.0a0 (2018-09-23)

5.1.1 Bug Fixes

- Fix tests that break when run individually
- user - Use SQLAlchemy hybrid_property on User's columns
- endpoint - Use the table name for endpoint in add_resource
- auth - Validate only Claim type view_args against token

5.1.2 Features

- Make default auth and database initialization optional
- app - Add add_resource method to Saraki and Blueprint
- endpoint
  - Make collection decorator aware of organization model classes
  - Add automatic API creation for organization resources
  - Add add_resource to automate API endpoints creation
- utility
  - Make @json support returns in the form (payload, headers)
  - Support global column exclusion from response payloads.
  - Use export_data method in export_from_sqla_object
  - Add current_org local proxy object
  - Add custom (Cerberus) validator

  - Add json decorator
  - Add export_from_sqla_object utility function
  - Add validation schema generator
- refactor - Require model_class only with unique rule in Validator
- auth
  - Include member privileges in access token
  - Add persistence for actions and resources
  - Add default scp claim value for organization owners
  - Add resource/action based authorization
  - Add authorization mechanism for org endpoints
  - Add initial authorization mechanism
  - Make iss claim optional by default
  - Add authentication
- role - Add member role management endpoints
- action - Add API to retrieve Action resources
- resource - Add API to retrieve Resource resources
- testing - Add a new module that implements test helpers
- plan - Add basic plans management
- member - Add endpoints to add and list members
- org - Add org account endpoints
- model - Add export_data method to Model class
- signup - Add signup endpoint

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

C

D

E

G

I

J

M

N

O

P

R

S

T

U

V