Base Acronis Cyber Platform API operations with Python

How to use examples

To successfully run samples Python version >= 3.6 is required.

To use those samples in your environment you should have pre-requirements installed.

```
requests>=2.22.0
```

To install those requirements, you can use provided requirements.txt

```
pip install -r requirements.txt
```

To use samples, move to the directory which contains the sample you want to run. Than run appropriate python script. They don't need parameters or request them interactively.

Most of scripts, just collect parameters and make corresponding API call directly with appropriate HTTP method, API endpoint URI and an authentication token received by Client during initialization based on data in `cyber.platform.cfg.json` file. Please, check Code Directory for scripts descriptions and The Acronis Cyber Platform API general workflow for the flow details. As well, this documents itself provides introduction to base operations with Acronis Cyber Platform API.

Code Directory

Folder name	File name	File description
common	base_operations.py	Contains code basis functions to call the API. As well as other utility functions, base sanity checks need to be performed before the API calls.

Folder name	File name	File description
common	`__init__.py`	Tells `Python` that it's a module directory.
authorization	`create_api_client.py`	Creates an API Client (`client_id`, `client_secret`) to generate a JWT token and access the API. The Basic Authentication is used. For Acronis Cyber Protect (Acronis Cyber Cloud 9.0) the Management Console can be used to create an API Client. The result of the script is stored in clear text `api_client.json` file. It's raw answer from the API call. For your solutions, please, implement secured storage for `client_id`, `client_secret` as they are credentials to access the API. The scrip asks for login and password to create an API Client.
authorization	`issue_access_token.py`	Issue a JWT token to access the API. The token is expired in 2 hours. During the sanity checks in `base_operations.py` an expiration time for the current token is checked and a token is reissued if needed. The result of the script is stored in clear text `api_token.json` file. It's raw answer from the API call. For your solutions, please, implement secured storage for a JWT token info as they are credentials to access the API.
authorization	`issue_customer_scoped_token_for_bc.py`	Exchange a JWT token to a customer-scoped token to access protection API functionality. The token is expired as soon as the base token expired. Required to have `customer.json` to exchange a token. Not renewed automatically. Requires to be issued for all operations with plans and resources as they expected to be executed in a customer context.
tenants	`create_partner_tenant_and_assign_specific_edition.py`	Creates a partner with name Python Partner RANDOMSTRING v3.0 and enables all available offering items for them for an edition, specified in json configuration files `cyber.platform.cfg.json` and `cyber.platform.cfg.defaults.json`.

Folder name	File name	File description
`tenants`	`create_customer_tenant_and _assign_specific_edition.py`	Creates a customer for Python Partner RANDOMSTRING v3.0 with name Python Customer RANDOMSTRING v3.0 and enables all available offering items for them for an edition, specified in json configuration files `cyber.platform.cfg.json` and `cyber.platform.cfg.defaults.json`.
`tenants`	`create_partner_tenant_and _assign_all_editions_available.py`	Creates a partner with name Python Partner All Editions RANDOMSTRING v3.0 and enables all available offering items for all available editions.
`tenants`	`create_partner_tenant_and _lock.py`	Creates a partner with name Python Locked Partner RANDOMSTRING v3.0 lock/unlock/disable/delete with prompt before operations to check in the Management Console.
`tenants`	`disable_all_applications _for_customer.py`	Disable all applications for previously created Python Customer RANDOMSTRING v3.0 customer.
`tenants`	`get_trial_end_date_ for_customer_legacy.py`	Use version of a customer `pricing` info to identify edn-of-trial date for a customer in the trial state.
`tenants`	`get_trial_end_date_ for_customer.py`	Use `production_start_date` of a customer `pricing` info to identify edn-of-trial date for a customer in the trial state.
`tenants`	`navigate_through_tenants.py`	Simple implementation of recursive tenant hierarchy browser.
`tenants`	`search_for_tenant_or_user.py`	Search for a tenant or user.
`tenants`	`switch_edition_for _customer_legacy.py`	Switch editions for a customer Python Switch Edition Customer RANDOMSTRING v3.0 using the legacy approach.
`tenants`	`switch_edition_for _customer.py`	Switch editions for a customer Python Switch Edition Customer RANDOMSTRING v3.0 using the current approach with a dry-run call.
`tenants`	`update_your_partners _to_all_available_editions.py`	Update all partners in underneath hierarchy to all currently available editions on your root tenant.

Folder name	File name	File description
`tenants`	`update_your_partners_to_all_offering_items_for_specific_edition.py`	Update all partners in underneath hierarchy to enable all available offering items for the specified edition - enable new offering items scenario.
`agent`	`get_agent_installation_token.py`	Create an Agent installation token for a user personal tenant.
`agent`	`get_all_agents_info.py`	Get list of all Acronis Agents for tenants subtree where the root tenant is a tenant for which an API Client is created.
`agent`	`get_all_agents_info_for_customer.py`	Get list of all Acronis Agents for tenants subtree where the root tenant is a previously created customer.
`monitoring`	`activities_pagination.py`	Get the list of all activities with pagination.
`monitoring`	`alerts_pagination.py`	Get the list of all alerts with pagination.
`monitoring`	`tasks_pagination.py`	Get the list of all tasks with pagination.
`monitoring`	`get_tasks.py`	Get the list of all tasks competed during the last week.
`monitoring`	`get_activities.py`	Get the list of all activities competed during the last week.
`monitoring`	`get_alerts.py`	Get the list of all alerts competed during the last week.
`resources`	`create_dynamic_group.py`	Create a dynamic group for resources.
`resources`	`get_all_resources_info.py`	Get the list of all resources available in authorization scope.
`resources`	`get_all_resources_protection_statuses.py`	Get the list of all resources available in authorization scope with their protection statuses.
`plans`	`apply_plan.py`	Apply the first applicable plan for the first resource.
`plans`	`base_plan.json`	A backup plan template.
`plans`	`create_a_backup_plan.py`	Create a backup plan based on `base_plan.json` template.
`plans`	`get_all_plans_info.py`	the the list of all protection plans available in authorization scope.
`plans`	`revoke_plan.py`	Revoke the first applicable plan from the first resource.

Folder name	File name	File description
`plans`	`start_plan_execution.py`	Start the first plan for the first resource execution.
`reports`	`create_report_retrieve.py`	Create an one time report to save for the root tenant, wait till its creation and download.
`usage`	`get_tenant_usage.py`	Gets usage for the root tenant.
`users`	`create_user_for_customer _activate_assign _backup_user_role.py`	Creates a user for Customer.Bash.Examples.v3 and activate them by sending an e-mail and assign the `backup_user` role. The script asks for a username and an e-mail to create.
`users`	`create_user_for_partner _activate_assign _admin_role.py`	Creates a user for Partner.Bash.Examples.v3 and activate them by sending an e-mail and assign the `partner_admin` role. The script asks for a username and an e-mail to create.
`users`	`search_for_tenant_or_user.py`	Search for a tenant or user.
`users`	`ipmpersonate_user_call_me.py`	Impersonate the user created for the customer and call `/users/me` to check.
`pdf`	`README.pdf`	This guide rendered to PDF format.
	`LICENSE`	The license for the code. It's MIT license.
	`README.md`	This file.
	`cyber.platform.cfg.defaults.json`	Contains default configuration values for the scripts. They are used when the values are not defined in `cyber.platform.cfg.json` file.
	`cyber.platform.cfg.json`	Contains configuration values for the scripts.

The Acronis Cyber Platform API general workflow

#	Operation	When/Period	Prerequisites / Inputs
1	Create an API client under which an integration will be authorized	Initially. Periodically if security policies require your company to regenerate all passwords each X months. Through the API or the Management Portal for ACC 9.0 and greater.	Login and password with a needed level of access in Acronis Cyber Cloud. Usually, it's a service Admin account under your company's Partner tenant in Acronis Cyber Cloud.

#	Operation	When/Period	Prerequisites / Inputs
2	Issue an access token	1. Before the first API Call which is not connected to the authorization flow 2. Each time when your token is near to be expired.	Your API Client credentials
3	Make API calls		An access token issued using your API Client credentials

Prerequisites and basis information

To run the scripts, you need to edit or create the `cyber.platform.cfg.json` file to provide base parameters. At minimum you need to change `base_url` to your data center URL. The `Config` class is initialized from the config file and used for all API requests. All other values can remain unchanged. A `cyber.platform.cfg.json` file example:

```json
{
  "base_url": "https://dev-cloud.acronis.com/",
  "partner_tenant": "partner",
  "customer_tenant": "customer",
  "edition": "pck_per_gigabyte",
  "switch_edition": "pck_per_workload",
  "cyber_protection_application_id": "6e6d758d-8e74-3ae3-ac84-50eb0dff12eb"
}
```

Create an API Client to access the API

A JWT token with a limited time to life approach is used to securely manage access of any API clients, like our scripts, for the Acronis Cyber Cloud. Using a login and password for a specific user is not a secure and manageable way to create a token, but technically it's possible. Thus, we create an API client with a client id and a client secret to use as credentials to issue a JWT token. To create an API Client, we call the `/clients` end-point with POST request specifying in the JSON body of the request a tenant we want to have access to. To authorize this the request, the Basic Authorization with user login and password for Acronis Cyber Cloud is used.

In Acronis Cyber Cloud 9.0 API Client credentials can be generated in the Management Portal.

Creating an API Client is a one-time process. As the API client is used to access the API, treat it as credentials and store securely. Also, do not store the login and password in the scripts itself.

In the following code block a login and a password are requested from a command line and use it for a Basic Authorization for following HTTP requests.

```python
# Read username and password from a command line
username = input("Username: ")
password = getpass.getpass(prompt="Password: ")
```

In those scripts it is expected that the Acronis Developer Sandbox is used. It is available for registered developers at Acronis Developer Network Portal. So the base URL for all requests (https://devcloud.acronis.com/) is used. Please, replace it with correct URL for your production environment if needed. For more details, please, review the Authenticating to the platform via the Python shell tutorial from the Acronis Cyber Platform documentation.

For demo purposes, this script issues an API client for a tenant for a user for whom a login and a password are specified. You should add your logic as to what tenant should be used for the API Client creation.

```python
# Request information about a user which is authenticated by
# username and password - Basic Authentication
response = requests.get(
    f'{cfg.base_url}api/2/users/me',
    auth=(username, password)
)

if response.ok:
    # Read tenant_id from received JSON
    my_tenant_id = response.json()["tenant_id"]

    # Build an object represents an API Client creation request JSON body
    client = {
        "type": "api_client",
        "tenant_id": f"{my_tenant_id}",
        "token_endpoint_auth_method": "client_secret_basic",
        "data": {
            "client_name": "Python.Client"
        }
    }

    ...

else:
    pprint.pprint(response.json())
```

`client_name` value defines the name you will see in the ACC 9.0 Management Console. For real integrations, please, name it carefully to have a way to identify it in a future.

```python
# Create an API Client with Basic Authentication
response = requests.post(
    f'{cfg.base_url}api/2/clients',
    headers={**cfg.header, **{'Content-Type': 'application/json'}},
    auth=(username, password),
    data=json.dumps(client)
)

if response.ok:
    # Save the created API Client info to api_client.json file
    with open(os.path.join(base_path, 'api_client.json'), 'w') as outfile:
        json.dump(response.json(), outfile)
else:
    pprint.pprint(response.json())
```

A generated client inherits access rights from a user used for the generation but it's disconnected from them. You don't need to issue a new client even if the user account is removed from Acronis Cloud.

Treat API Clients as a specific service account with access to your cloud. All internal security policies applied to your normal account operations should be in place for API Clients. Thus, don't create new API Clients unless really required.

You can receive a `client_secret` only once when `client_secret` is issued. If you lose your `client_secret` further you must reset secret for the client through the Management Console or API Calls. Please, be aware, that all tokens are invalidated on secret reset.

You need to securely store the received credentials. For simplicity of the demo code, a simple JSON format is used for `api_client.json` file. Please remember to implement secure storage for your client credentials.

Issue a token to access the API

A `client_id` and a `client_secret` can be used to access the API using the Basic Authorization but it's not a secure way as we discussed above. It's more secure to have a JWT token with limited life-time and implement a renew/refresh logic for that token.

To issue a token `/idp/token` end-point is called using `POST` request with param `grant_type` equal `client_credentials` and content type `application/x-www-form-urlencoded` with Basic Authorization using a `client_id` as a user name and a `client_secret` as a password.

```python
with open(os.path.join(base_path, 'api_client.json')) as api_client_file:
    api_client_json = json.load(api_client_file)

client_id = api_client_json["client_id"]
client_secret = api_client_json["client_secret"]

response = requests.post(
    f'{cfg.base_url}api/2/idp/token',
    headers={**cfg.header, **{'Content-Type': 'application/x-www-form-urlencoded'}},
    auth=(client_id, client_secret),
    data={'grant_type': 'client_credentials'}
)

if response.ok:
    with open(os.path.join(base_path, 'api_token.json'), 'w') as outfile:
        json.dump(response.json(), outfile)
else:
    pprint.pprint(response.json())
```

You need to securely store the received token. For simplicity of the demo code, the received JSON format is used `api_token.json` file. Please implement secure storage for your tokens.

A token has time-to-live and must be renewed/refreshed before expiration time. The best practice is to check before starting any API calls sequence and renew/refresh if needed.

Currently, the default time-to-live to a token for the API is 2 hours.

Assuming that the token is stored in the JSON response format as above, it can be done using the following functions set.

`expires_on` is a time when the token will expire in Unix time format -- seconds from January 1, 1970. Here we assume that we will renew/refresh a token 15 minutes before the expiration time.

```
# Check if the token valid at least 15 minutes
def __read_token(self):
    if os.path.exists(os.path.join(base_path, 'api_token.json')):
        with open(os.path.join(base_path, 'api_token.json')) as api_token_file:
            self.__api_token = json.load(api_token_file)
            expires_on = self.__api_token["expires_on"]
            if expires_on - time() < 900:
                self.__issue_token()
    elif os.path.exists(os.path.join(base_path, 'api_client.json')):
        self.__issue_token()

 def __issue_token(self):
    if os.path.exists(os.path.join(base_path, 'api_client.json')):
        client_id = self.__api_client["client_id"]
        client_secret = self.__api_client["client_secret"]

        response = requests.post(
            f'{self.base_url}api/2/idp/token',
            headers={'Content-Type': 'application/x-www-form-urlencoded'},
            auth=(client_id, client_secret),
            data={'grant_type': 'client_credentials'}
        )

        if response.ok:
            self.__api_token = response.json()
            self.access_token = self.__api_token["access_token"]
            with open(os.path.join(base_path, 'api_token.json'), 'w') as outfile:
                json.dump(self.__api_token, outfile)
        else:
            pprint.pprint(response.json())
```

Create partner, customer and user tenants and set offering items

So now we can securely access the Acronis Cyber Platform API calls. In this topic we discuss how to create a partner, a customer tenants and enable for them all available offering items, and then create a user for the customer and activate the user by setting a password.

As we discussed above, before making a call to the actual API you need to ensure that an authorization token is valid. Please, use the functions like those described above to do it.

Assuming that we create the API client for our root tenant, we start from retrieving the API Client tenant information using GET request to `/clients/{clientId}` end-point. Then, using received `tenant_id` information as a parameter and `kind` equal to `partner`, we build a JSON body for POST request to `/tenants` end-point to create the partner. Next, we are going to enable all applications and offering items for the tenants. Briefly, we take all available offering items for the parent tenant of the partner or the customer using GET request to `/tenants/{tenantId}/offering_items/available_for_child` end-point with needed query parameters specifying `edition` and `kind` of the tenant. Then, we need to enable these offering items for the partner or the customer using PUT request to `/tenants/{tenantId}/offering_items` end-point with all offering items JSON in the request body and appropriate `tenantId`.

The following `kind` values are supported `partner`, `folder`, `customer`, `unit`.

```
# Initialize config and read all required values form JSON config
# an API client and a token files
cfg = Config(full=True)
acronis = Acronis(cfg)

partner = {
    "name": f"Python Partner {id_generator()} v3.0",
    "parent_id": f"{cfg.tenant_id}",
    "kind": f"{cfg.partner_tenant}"
}

response = acronis.post(
            'api/2/tenants',
            data=json.dumps(partner)
)

if response.ok:
    with open(os.path.join(base_path, 'partner.json'), 'w') as outfile:
        json.dump(response.json(), outfile)

    new_partner = Tenant(os.path.join(base_path, 'partner.json'))

    response = acronis.get(
        f'api/2/tenants/{cfg.tenant_id}/offering_items/available_for_child?kind={cfg.par
    )

    if response.ok:

        offering_items = json.loads('{"offering_items":[]}')
        offering_items["offering_items"] = response.json()["items"]

        response = acronis.put(
            f'api/2/tenants/{new_partner.tenant_id}/offering_items',
            data=json.dumps(offering_items)
        )

        if response.ok:
            print(f"Offering items were set for tenant {new_partner.tenant_id}")
        else:
            pprint.pprint(response.json())
    else:
        pprint.pprint(response.json())
```

```
else:
    pprint.pprint(response.json())
```

This is absolutely the same process as for a customer, the only difference is `kind` equal to `customer` in the request body JSON and `/offering_items/available_for_child` parameters.

```
# Initialize config and read all required values form JSON config
# an API client and a token files
cfg = Config(full=True)
acronis = Acronis(cfg)

partner = Tenant(os.path.join(base_path, 'partner.json'))

customer = {
    "name": f"Python Customer {id_generator()} v3.0",
    "parent_id": f"{partner.tenant_id}",
    "kind": f"{cfg.customer_tenant}"
}

response = acronis.post(
     'api/2/tenants',
     data=json.dumps(customer)
)

if response.ok:
    with open(os.path.join(base_path, 'customer.json'), 'w') as outfile:
        json.dump(response.json(), outfile)

    new_customer = Tenant(os.path.join(base_path, 'customer.json'))

    response = acronis.get(
        f'api/2/tenants/{cfg.tenant_id}/offering_items/available_for_child?kind={cfg.cus
    )

    if response.ok:

        offering_items = json.loads('{"offering_items":[]}')
        offering_items_to_filter = response.json()["items"]

        # Only 1 location for a customer as well only 1 storage of 1 type
        # ba2976d0-c13e-4661-ae60-b4593583fce2 - Google DR storage for dev-cloud
        filtered_offering_items = [oi for oi in offering_items_to_filter if (oi["type"]
        filtered_offering_items = [oi for oi in filtered_offering_items if (oi["type"] =

        offering_items["offering_items"] = filtered_offering_items

        with open(os.path.join(base_path, 'customer_offering_items.json'), 'w') as outfi
            json.dump(offering_items, outfile)

        response = acronis.put(
            f'api/2/tenants/{new_customer.tenant_id}/offering_items',
            data=json.dumps(offering_items)
        )

        if response.ok:
            print(f"Offering items were set for tenant {new_customer.tenant_id}")
        else:
            pprint.pprint(response.json())
    else:
        pprint.pprint(response.json())
else:
    pprint.pprint(response.json())
```

By default, customers are created in a trial mode. To switch to production mode we need to update customer pricing. To perform this task, we start from requesting current pricing using a GET request to `/tenants/{customerTenantId}/pricing` end-point then change `mode` property to `production` in the received JSON, then, finally, update the pricing using PUT request to `/tenants/{customerTenantId}/pricing` end-point with a new pricing JSON.

⚠ **Please, be aware, that this switch is non-revertible.**

```
response = acronis.get(
    f'api/2/tenants/{new_customer.tenant_id}/pricing'
)

if response.ok:

    pricing = response.json()
    pricing["mode"] = "production"

    response = acronis.put(
        f'api/2/tenants/{new_customer.tenant_id}/pricing',
        data=json.dumps(pricing)
    )

    if response.ok:
        print(f"Customer tenant {new_customer.tenant_id} pricing set to production mode.
    else:
        pprint.pprint(response.json())
else:
    pprint.pprint(response.json())
```

Finally, we create a user for the customer. At first, we check if a login is available using GET request to `/users/check_login` end-point with `username` parameter set to an expected login. Then, we create a JSON body for POST request to `/users` end-point to create a new user.

```
# Initialize config and read all required values form JSON config
# an API client and a token files
cfg = Config(full=True)
acronis = Acronis(cfg)

login = input("New login: ")
email = input("Please enter a valid email, it will be used for account activation: ")

response = acronis.get(
    f"api/2/users/check_login?username={login}"
)
```

```
if response.ok and response.status_code == 204:
    customer = Tenant(os.path.join(base_path, "customer.json"))

    user = {
        "tenant_id": f"{customer.tenant_id}",
        "login": f"{login}",
        "contact": {
            "email": f"{email}",
            "firstname": f"First {login}",
            "lastname": f"Last {login}"
            }
        }

    response = acronis.post(
        'api/2/users',
        data=json.dumps(user)
    )

    if response.ok:
        with open(os.path.join(base_path, 'user.json'), 'w') as outfile:
            json.dump(response.json(), outfile)
    else:
        pprint.pprint(response.json())
else:
    pprint.pprint(response.json())
```

A created user is not active. To activate them we can either send them an activation e-mail or set them a password. The sending of an activation e-mail is the preferable way.

```
new_user = User(os.path.join(base_path, 'user.json'))

response = acronis.post(
    f'api/2/users/{new_user.id}/send-activation-email'
)

if response.ok:
    print(f"User {login} is activated by sending an e-mail.")
else:
    pprint.pprint(response.json())
```

At this point, we've created a partner, a customer, enable offering items for them, create a user and activate them.

The created user has no roles assigned. It means it can't use any service. To enable services/applications you need to assign an appropriate role to a user.

All operations with the user account roles are located under the `/users/{user_id}/access_policies` endpoint.

To build a JSON to assign a role for a user `id` and user `personal_tenant_id` need to be known. All these values can be retrieved from the `user.json` file we've received as result of

the user creation API call.

```
user_role = {
    "items": [
        {
            "id": "00000000-0000-0000-0000-000000000000",
            "issuer_id": "00000000-0000-0000-0000-000000000000",
            "role_id": "backup_user",
            "tenant_id": f"{customer.tenant_id}",
            "trustee_id": f"{new_user.id}",
            "trustee_type": "user",
            "version": 0
        }
    ]
}

response = acronis.put(
          f'api/2/users/{new_user.id}/access_policies',
          data=json.dumps(user_role)
        )
```

Get a tenant usage

A very common task is to check a tenant's usage. It's a simple task. We just need to make a GET request to `/tenants/{tenantId}/usages` end-point, as result we receive a list with current usage information in JSON format.

The information about a service usage of the tenant, provided by the `/tenants/{tenantId}/usages` endpoint, is updated on average every 0.5 hours and must not be used for billing purposes.

```
# Initialize config and read all required values form JSON config
# an API client and a token files
cfg = Config(full=True)
acronis = Acronis(cfg)

response = acronis.get(
    f'api/2/tenants/{cfg.tenant_id}/usages'
)

if response.ok:
    with open(os.path.join(base_path, f'tenant_usage_{cfg.tenant_id}.json'), 'w') as out
        json.dump(response.json(), outfile)
else:
    pprint.pprint(response.json())
```

It's very useful to store usage information for further processing. In our example we use response JSON format to store it in a file.

Create and download simple report

The reporting capability of the Acronis Cyber Cloud gives you advanced capabilities to understand usage. In the following simple example, we create a one-time report in csv format, and then download it. To check other options, please, navigate to the Acronis Cyber Platform documentation.

To create a report to save, we build a body JSON and make a POST request to `/reports` end-point. Then we look into stored reports with specified `reportId` making a GET request to `/reports/{reportId}/stored` endpoint.

```
# Initialize config and read all required values form JSON config
# an API client and a token files
cfg = Config(full=True)
acronis = Acronis(cfg)

report = {
    "parameters": {
        "kind": "usage_current",
        "tenant_id": f"{cfg.tenant_id}",
        "level": "accounts",
        "formats": [
            "csv_v2_0"
        ]
    },
    "schedule": {
        "type": "once"
    },
    "result_action": "save"
}

response = acronis.post(
    'api/2/reports',
    data=json.dumps(report)
)

if response.ok:
    report_status = "non saved"
    report_id = response.json()["id"]
    stored_report_id = None

    while report_status != "saved":
        response = acronis.get(
            f'api/2/reports/{report_id}/stored'
         )

        if response.ok:
            report_status = response.json()["items"][0]["status"]
        else:
            pprint.pprint(response.json())

        time.sleep(2)

    stored_report_id = response.json()["items"][0]["id"]

    response = acronis.get(
        f'api/2/reports/{report_id}/stored/{stored_report_id}',
        )
    if response.ok:
        with open(os.path.join(base_path, f'report_for_tenant_{cfg.tenant_id}.csv'), 'w'
            outfile.write(response.text)
    else:
        pprint.pprint(response.json())
else:
    pprint.pprint(response.json())
```

Add marks to your API calls for better support

It's technically possibly to identify your API calls as they are connected to your API Client. But still it's required a lot of efforts and hard to find in your Audit log at the Management Portal for your. Thus to better support your development effort it would be a great idea to identify your integrations and API calls somehow. Traditional way to do it in a RESTFul word is using the `User-Agent` header.

There are common recommendations how to build your `User-Agent` header:

```
User-Agent: <product>/<product-version> <comment>
```

For example, for our hands-on lab, you can use:

```
User-Agent: Training/1.0 Acronis #CyberFit Developers Business Automation Training
```

To implement it using our `Python` examples, we need just add the header to each `request` call using API have header:

```
 self.header = {"User-Agent": "ACP 3.0/Acronis Cyber Platform Python Examples"}
```

⚠ **Please, for a real integration, use your real integration name, a specific version and suitable comments to simplify your support.**

Summary

Now you know how to use base operations with the Acronis Cyber Platform API:

1. Create an API Client for the Acronis Cyber Platform API access
2. Issue a token for secure access for the API
3. Establish a simple procedure to renew/refresh the token
4. Create a partner and a customer tenants and enable offering items for them.
5. Create a user for a customer tenant and activate them.
6. Enable services for a user by assigning a role.
7. Receive simple usage information for a tenant.
8. Create and download reports for usage.

Get started today, register on the Acronis Developer Portal and see the code samples available, you can also review solutions available in the Acronis Cyber Cloud Solutions Portal.