

Force Login for REST Authentication in 4D

By Abir HSAINI, Technical Services Engineer, 4D Inc.

Technical Note 25-10

Table of Contents

Table of Contents.....	2
Abstract	4
Introduction.....	4
New Projects vs. Existing Projects	4
Force Login Mode for New Projects (v20 R6+)	4
Here is a picture in Qodly that shows the details of roles.json in a simpler way.	5
Migrating Existing Projects to Force Login Mode.....	6
How Force Login Works	7
Step 1: Initial REST Request - Guest Session Creation	7
Descriptive REST Requests:.....	7
Step 2: Creating the authenticate() Function.....	7
Implementation Approaches.....	7
Option A: Open Access (No Authentication Required)	8
Option B: Using 4D User Groups (Legacy Read/Write Access).....	8
Option C: Custom User Management System (Legacy "On REST Authentication"	
Method).....	9
Step 3: Authentication Request	10
REST License Consumption	10
Guest Sessions - No License Required.....	10
License Consumption Trigger	10
License Management Lifecycle.....	11
Roles and Privileges (roles.json).....	11
Permission actions	12
The roles.json File.....	14
Error Handling	16
Configuration Best Practices.....	16
Demo.....	17
Role.JSON Configuration.....	19
Understanding the Configuration	20
Privileges Defined	20
Permissions Structure.....	20
Using Custom DataStore Functions.....	21
To illustrate this example, two methods “ hasPrivilege() and getPrivileges() ” are	
created in the DataStoreImplementation class, and execution permission is	
granted exclusively to users with the Admin privilege.....	21
Checking Privileges with ds.hasPrivilege()	22
Getting All Session Privileges with ds.getPrivileges()	22
Attempting to Call Admin Methods Without Admin Privilege.....	23
Fetching Data After Authentication	24
Example 1: Fetching All Companies.....	24
Example 2: Fetching a Specific Company by Key	25
Example 3: Querying Companies with class function	26

Creating a New Company Record.....	27
Updating an Existing Company Record	28
Deleting a Company Record	30
Conclusion	31

Abstract

The “Force Login” feature in 4D secures access to REST services and Qodly applications by requiring users to sign in before performing protected actions. When a connection is first established, a guest session is created with no privileges and no assigned license until valid login credentials are provided.

This feature not only strengthens security but also improves license management for REST and Qodly access, while enabling your application to use detailed roles and privileges for precise control over user permissions.

This document explains how to migrate from older authentication settings to the new “Force Login” feature, describes how the authenticate method works and how licenses are distributed, and shows how to define access rights in the roles.json file. Clear REST call examples demonstrate how to sign in, view data, and create new records—ensuring secure and well-managed access to REST services in 4D.

Introduction

In modern web-architecture, **REST** services play a critical role in exposing application logic and data to diverse clients (web frontends, mobile apps, external systems). But with this level of accessibility comes a paramount concern: security. In the context of 4D, the “Force Login” mode is a more robust and fine-grained mechanism for protecting REST endpoints, ensuring that only authenticated sessions with appropriate privileges can perform sensitive operations.

Starting in 4D 20 R6 and later versions, Force Login becomes the default REST authentication mode in new projects. Older “legacy” approaches such as relying on the “On REST Authentication” database method are now deprecated.

This tech note presents the process of moving from legacy REST configurations to the Force Login model. It explains how the sign-in workflow operates, how sessions and licenses are managed, and how access rights can be defined using the roles.json file. Practical REST call examples are provided to demonstrate how to view, create, update and protect data, ensuring secure and well-structured access to REST services.

New Projects vs. Existing Projects

Force Login Mode for New Projects (v20 R6+)

Starting from version 20 R6, 4D implements a secure-by-default approach for new projects through the Force Login mode. This mechanism ensures that all protected REST operations require proper authentication before execution.

When creating a new project in 4D v20 R6 or later, the system automatically:

- Generates a roles.json file in the *Sources* folder

- Sets the forceLogin attribute to true
- Applies a "none" privilege by default, which denies access to the entire REST API

This default configuration follows data security best practices by adopting a deny-all approach, requiring developers to explicitly grant permissions as needed rather than operating in an open-access mode.

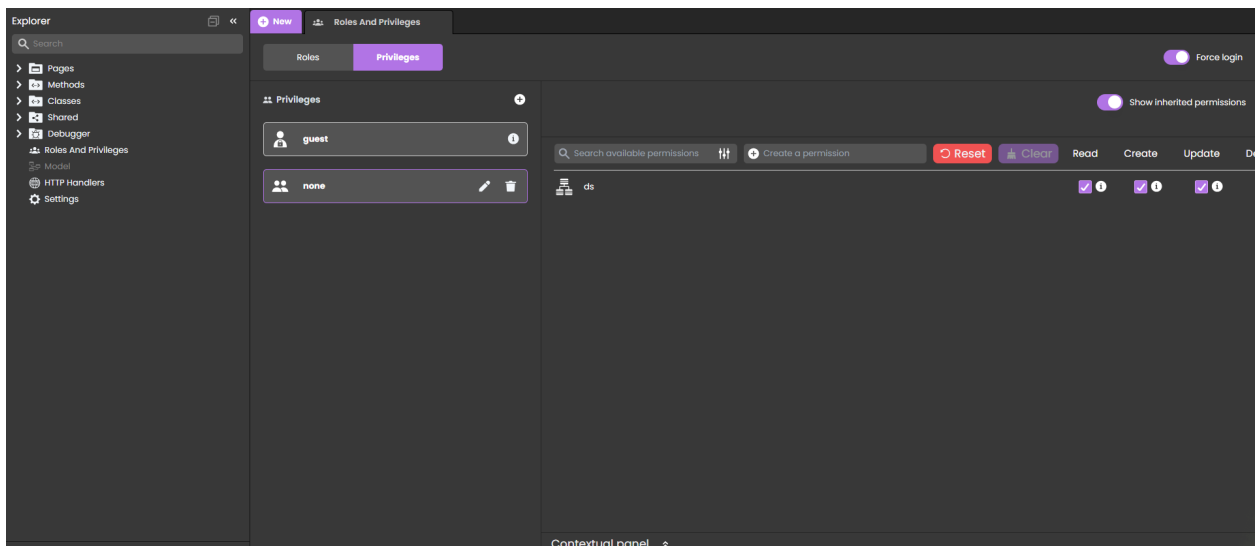
```

roles.json
File Edit View

{
  "privileges": [
    {
      "privilege": "none",
      "includes": []
    }
  ],
  "roles": [],
  "permissions": {
    "allowed": [
      {
        "applyTo": "ds",
        "type": "datastore",
        "read": ["none"],
        "create": ["none"],
        "update": ["none"],
        "drop": ["none"],
        "execute": ["none"],
        "promote": ["none"]
      }
    ]
  },
  "forceLogin": true
}

```

Here is a picture in Qodly that shows the details of roles.json in a simpler way.

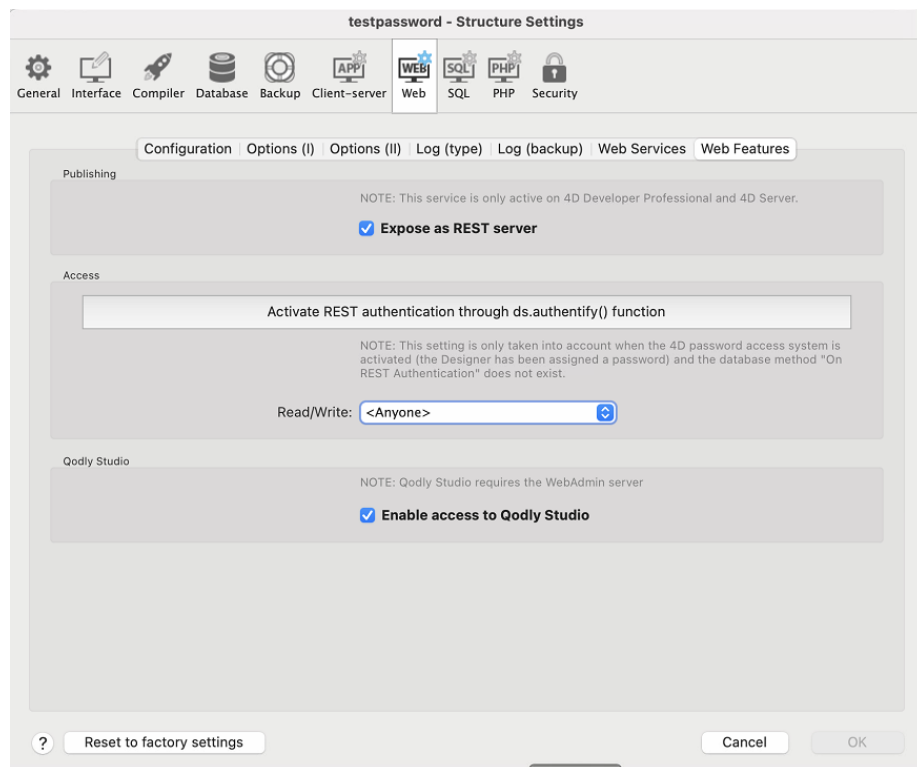


Migrating Existing Projects to Force Login Mode

Projects created before version 20 R6 or those still using legacy REST authentication methods require migration to adopt the new security model. 4D provides a streamlined automatic conversion process.

To migrate an existing project:

1. Open Settings → Web → Web Features
2. Click the button: "Activate REST authentication through ds.authenticate() function"



During conversion, 4D executes the following steps:

- **Removes legacy permissions:** Deletes the user group previously granted REST access in the old REST settings
- **Deactivates legacy authentication:** Moves the **On REST Authentication** database method into the system folder (deactivated state)
- **Creates security configuration:** Generates a **roles.json** file (if missing) in the *Sources* folder, with `forceLogin: true` and a default "none" privilege

Important: Simply editing the **roles.json** file and adding `"forceLogin": true` is **not sufficient** to update the database. You **must** click the **"Activate REST authentication through ds.authenticate() function"** button to complete the migration properly.

How Force Login Works

The Force Login mechanism in 4D follows a three-step authentication workflow that ensures secure REST API access while optimizing license consumption.

Step 1: Initial REST Request - Guest Session Creation

When a remote 4D client makes the **first REST request** to a 4D application configured with Force Login:

- A **"guest" web user session** is automatically created by default
- This session has **no privileges** and **no rights** to execute data requests
- Only **descriptive REST requests** are allowed in guest sessions

Descriptive REST Requests:

Guest sessions can process the following types of descriptive REST requests without requiring authentication or a license:

- `/rest/$catalog` requests (e.g., `/rest/$catalog/$all`) - provides access to available dataclasses
- `/rest/$catalog/authenticate` - the datastore function used to authenticate the user
- `/rest/$getWebForm` - renders a Qodly page (useful for displaying login forms)

These descriptive requests are always processed by the server, even when no licensed web user session is open. They allow remote 4D clients to display information about the number of data classes, number of data classes fields, and authentication endpoints before consuming a license.

Step 2: Creating the `authenticate()` Function

Before authentication can occur, you must create an **`authenticate()` function** as an **exposed datastore class function**. This function :

- Receives and validates user credentials
- Calls `Session.setPrivileges()` with appropriate privileges if credentials are valid
- Controls when a 4D license is consumed

Important: The `authenticate()` function must be explicitly exposed in your datastore class to be accessible via REST.

Implementation Approaches

There are multiple approaches to implementing the `authenticate()` function, depending on the legacy authentication configuration:

Option A: Open Access (No Authentication Required)

If the previous configuration exposed the REST server without defining an access group or filtering requests in the "On REST Authentication" database method, this behavior can be reproduced by granting full access to the entire REST API:

```
Class extends DataStoreImplementation exposed Function authenticate() : Boolean return
    Session.setPrivileges("Administrator")
```

In this example:

- No credentials are required (no parameters)
- The function immediately grants "Administrator" privileges
- This provides unrestricted access to the REST API, similar to the legacy open configuration

Security Warning: This implementation is not recommended because it is not secured. All data and functions are accessible to everyone without any authentication or authorization checks.

Option B: Using 4D User Groups (Legacy Read/Write Access)

If a group of 4D users with Read/Write is defined in the legacy configuration access to the REST API, this behavior can be reproduced in the new mode by checking credentials and granting full access to members of the defined group:

```
Class extends DataStoreImplementation exposed Function authenticate($identifier : Text; $pwd : Text) : Boolean
If ($identifier#"" )
    If (Validate password($identifier; $pwd))
        If (User in group($identifier; "RestAccess"))
            return Session.setPrivileges("Administrator")
        End if
    End if
End if
```

In this example:

- The function validates the user's password using `Validate password()`.
- It checks if the user belongs to the "RestAccess" group (the group previously defined in Structure Settings).
- If both conditions are met, it grants "Administrator" privileges.
- This restriction secures data access and functions against malicious connections.
- Each authorized connection has the same rights.

Option C: Custom User Management System (Legacy "On REST Authentication" Method)

If the "**On REST Authentication**" database method was previously used to check credentials against a custom user management system, this logic can be migrated to the `authenticate()` function.

Legacy "On REST Authentication" method:

```
#DECLARE($identifier : Text; $pwd : Text) : Boolean

If ($identifier# "")
  var $user : cs.UserEntity:=ds.User.query("identifier = :1"; $identifier).first()
  If ($user#Null)
    If (Verify password hash($pwd; $user.pwd))
      return True
    End if
  End if
End if
```

Replacing "`ds.authenticate()`" function:

```
Class extends DataStoreImplementation

exposed Function authenticate($identifier : Text; $pwd : Text) : Boolean
  If ($identifier# "")
    var $user : cs.UserEntity:=ds.User.query("identifier = :1"; $identifier).first()
    If ($user#Null)
      If (Verify password hash($pwd; $user.pwd))
        return Session.setPrivileges("Administrator")
      End if
    End if
  End if
End if
```

For this custom authentication to work, the developer must create a new privilege called "**userAccess**" in the `roles.json` file with the following configuration:

- Grant **read access** to the User table
- Add the **authenticate** function to the **promote** list

Security Warning: Do NOT simply grant read access to the User table for guest sessions in your `roles.json`! This is a common mistake that would expose sensitive user data (including password hashes) to unauthenticated users. Instead, the **promote** mechanism ensures that only the `authenticate` function can access the User table during the authentication process, without exposing it to general guest access.

This is necessary because the guest session needs permission to access the User table to verify credentials and promote itself through the `authenticate` function. Without this configuration, the authentication will fail.

In this example:

- User credentials are stored in a custom User dataclass
- The function queries the User table to find the user by identifier
- It verifies the password hash against the stored hash
- If validation succeeds, it grants "Administrator" privileges
- As with the previous option, each authorized connection has the same rights
- Data and function access restrictions need to be implemented separately in the code

Regardless of your authentication strategy (internal 4D passwords, OAuth2 with Office 365, Google authentication, LDAP, custom user tables, etc.), **all authentication must go through the `ds.authenticate()` function.**

Step 3: Authentication Request

The client sends an authentication request to the server:

```
POST /rest/$catalog/authenticate
```

This request includes the user credentials (username and password). At this stage :

- Only a basic login form is required (no data access needed)
- The request is processed within the existing guest session

REST License Consumption

The Force Login mode in 4D introduces a new approach to license management for REST sessions, where licenses are consumed only when privileges are assigned, not when sessions are created, providing precise control over resource allocation.

Guest Sessions - No License Required

When a client initiates a new REST connection:

- A guest web user session is created automatically
- No 4D license is consumed at this stage
- This guest web user session has no privilege at all, it cannot access any data nor execute any code
- This guest web user session can process descriptive REST requests without requiring a license
- Multiple guest sessions can exist simultaneously without any license impact

License Consumption Trigger

The only way to assign privilege to this guest web user session is to execute `ds.authenticate()` and inside it, execute `Session.setPrivileges()`.

A 4D Client license is consumed **as soon as (and only when)** the `Session.setPrivileges()` function is executed. This occurs during the authentication process when:

1. The client calls the `/rest/$catalog/authenticate` endpoint
2. The `authenticate()` function identifies and recognizes the attempted access
3. The `authenticate()` function calls `Session.setPrivileges()` with appropriate privileges
4. At this exact moment, a **4D Client license is consumed** on the server

License Management Lifecycle

- **Dynamic Allocation:** Licenses are dynamically allocated during the first HTTP/REST request from a user (specifically when `Session.setPrivileges()` is called during authentication). Whether the developer connects with a heavy client (Client/Server desktop) or a web browser client, the same licenses (4D Client) are consumed.
- **License Attachment:** Once assigned, the license remains attached to the user session for its entire duration. The license is bound to that specific session and cannot be transferred or shared with other users.
- **License Release:** A license is automatically released under the following conditions:
 - ✓ Session timeout: If no REST request is received during the session timeout period (default minimum of 1 hour), the license is automatically freed
 - ✓ Server restart: The license is released immediately when 4D Server is restarted
 - ✓ User logout: A session license is also released if a 'logout' action is triggered by the end user through a Qodly Page interface (using Qodly Renderer).
- **No License Sharing :** A REST license cannot be used simultaneously by multiple users. Each licensed session requires its own dedicated license from the available pool.

Roles and Privileges (roles.json)

The ORDA security architecture is based upon the concepts of privileges, permission actions (read, create, etc.), and resources. When web users or REST users get logged, their session is automatically loaded with associated privilege(s). Privileges are assigned to the session using the **`session.setPrivileges()`** function.

Every user request sent within the session is evaluated against privileges defined in the project's **roles.json** file. If a user attempts to execute an action and does not have the appropriate access rights, a privilege error is generated or, in the case of missing Read permission on attributes, they are not sent.

Specific permission actions can be assigned to the following resources in a project:

- the datastore
- a dataclass
- an attribute (including computed and alias)
- a data model class function
- a singleton function

Each time a resource is accessed within a session (whatever the way it is accessed), 4D checks that the session has the appropriate permissions, and rejects the access if it is not authorized.

Permission actions

Available actions are related to the target resource:

Actions	datastore	Dataclass	attribute	data model function or singleton function
create	Create entity in any dataclass	Create entity in this dataclass	Create an entity with a value different from default value allowed for this attribute (ignored for alias attributes).	n/a
read	Read attributes in any dataclass	Read attributes in this dataclass	Read this attribute content	n/a
update	Update attributes in any dataclass.	Update attributes in this dataclass.	Update this attribute content (ignored for alias attributes).	n/a

Actions	datastore	Dataclass	attribute	data model function or singleton function
drop	Delete data in any dataclass.	Delete data in this dataclass.	Delete a not null value for this attribute (except for alias and computed attribute).	n/a
execute	Execute any function on the project (datastore, dataclass, entity selection, entity, singleton)	Execute any function on the dataclass. Dataclass functions, entity functions, and entity selection functions are handled as dataclass functions	n/a	Execute this function
promote	n/a	n/a	n/a	Associates a given privilege during the execution of the function. The privilege is temporary added and removed at the end of the function execution. By security, only the process executing the function is added the privilege, not the whole session.

Important: In REST force login mode, the `authenticate()` function is always executable by guest users, whatever the permissions configuration.

The roles.json File

When creating a project, a default roles.json file is created at the following location:

<project folder>/Project/Sources/


For the highest level of security, the "none" privilege is assigned to all permissions in the datastore, thus data access on the whole ds object is disabled by default.

Important: It is recommended not to modify or use this locking privilege, but to add specific permissions to each resource that should be made available from web or REST requests.

The default file has the following contents:

```
{
  "privileges": [
    {
      "privilege": "none",
      "includes": []
    }
  ],
  "roles": [],
  "permissions": {
    "allowed": [
      {
        "applyTo": "ds",
        "type": "datastore",
        "read": ["none"],
        "create": ["none"],
        "update": ["none"],
        "drop": ["none"],
        "execute": ["none"],
        "promote": ["none"]
      }
    ]
  },
  "forceLogin": true
}
```

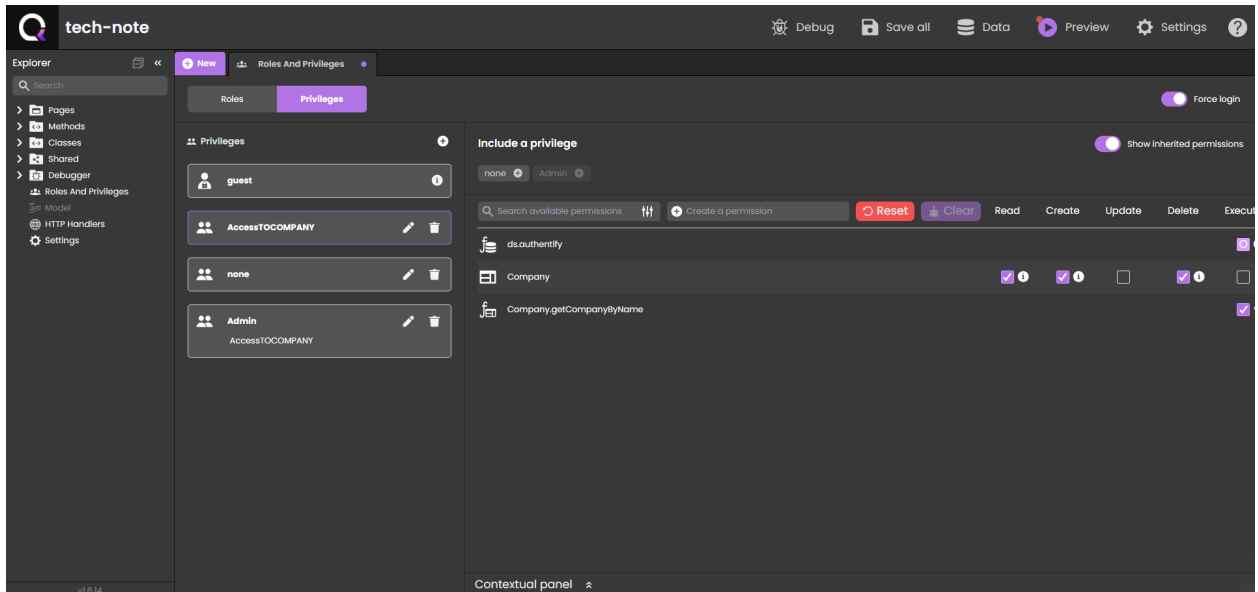
The roles.json file syntax are the following:

Property	Type	Mandatory	Description
privileges	Collection of privilege objects		List of defined privileges
[].privilege	Text		Privilege name
[].includes	Collection of strings		List of included privilege names

roles	Collection of role objects		List of defined roles
[]role	Text		Role name
[]privileges	Collection of strings		List of included privilege names
permissions	Object	✓	List of allowed actions
allowed	Collection of permission objects		List of allowed permissions
[]applyTo	Text	✓	Targeted resource name
[]type	Text	✓	Resource type: "datastore", "dataclass", "attribute", "method", "singletonMethod", "singleton"
[]read, [].create, [].update, [].drop, [].execute, [].promote	Collection of strings		List of privilege names
forceLogin	Boolean		Enable <i>force login</i> mode

Important Notes: The "WebAdmin" privilege name is reserved to the application. It is not recommended to use this name for custom privileges.

Instead of manually modifying the roles.json file, it is recommended to use the **Roles and Privileges** page in **Qodly**. This interface offers a more intuitive and secure way to define roles, assign privileges, and maintain consistent access control across your application.



Error Handling

The roles.json file is parsed by 4D at startup. The application must be restarted for modifications in this file to be considered.

In case of errors when parsing the roles.json file, 4D loads the project but disables the global access protection, this allows the developer to access the files and fix the errors. An error file named Roles_Errors.json is generated by default in the Logs folder of the project and describes the error lines. This file is automatically deleted when the roles.json file no longer contains errors.

It is recommended to check at startup if a Roles_Errors.json file exists in the Logs folder, which means that a parsing error was generated and that accesses will not be limited.

Configuration Best Practices

The good practice is to keep all data access locked by default thanks to the "none" privilege and to configure the roles.json file to only open controlled parts to authorized sessions.

```
{
  "privileges": [
    {
      "privilege": "none",
      "includes": []
    }
  ],
  "roles": [],
  "permissions": {
    "allowed": [
      {
```

```

    "applyTo": "ds",
    "type": "datastore",
    "read": ["none"],
    "create": ["none"],
    "update": ["none"],
    "drop": ["none"],
    "execute": ["none"],
    "promote": ["none"]
  },
  {
    "applyTo": "ds.loginAs",
    "type": "method",
    "execute": ["guest"]
  },
  {
    "applyTo": "ds.hasPrivilege",
    "type": "method",
    "execute": ["guest"]
  }
]
},
"forceLogin": true
}

```

This configuration ensures that the datastore is locked by default, but specific authentication-related methods are accessible to guest sessions.

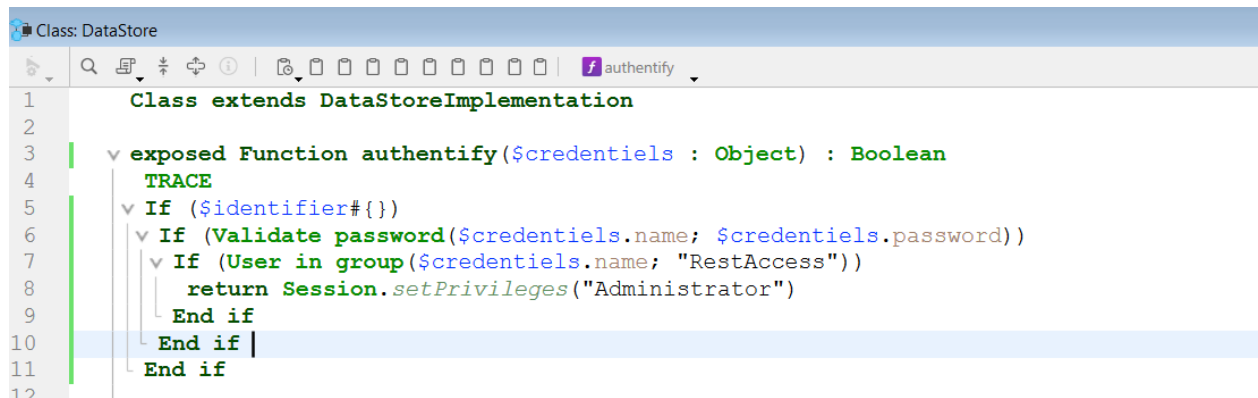
Demo

This section provides practical examples demonstrating how to interact with a 4D REST API configured with Force Login mode, from initial authentication through data manipulation operations.

4D User Group Authentication

The first step in working with a Force Login-enabled REST API is to authenticate the user by calling the `authenticate()` function through the catalog endpoint.

The `authenticate()` function:



```

Class: DataStore
authenticate
1      Class extends DataStoreImplementation
2
3      exposed Function authenticate($credentials : Object) : Boolean
4      TRACE
5      If ($identifier#{})
6      If (Validate password($credentials.name; $credentials.password))
7      If (User in group($credentials.name; "RestAccess"))
8          return Session.setPrivileges("Administrator")
9      End if
10     End if |
11     End if
12

```

In this example, postman is used for sending requests, the request needs to be configured as follows:

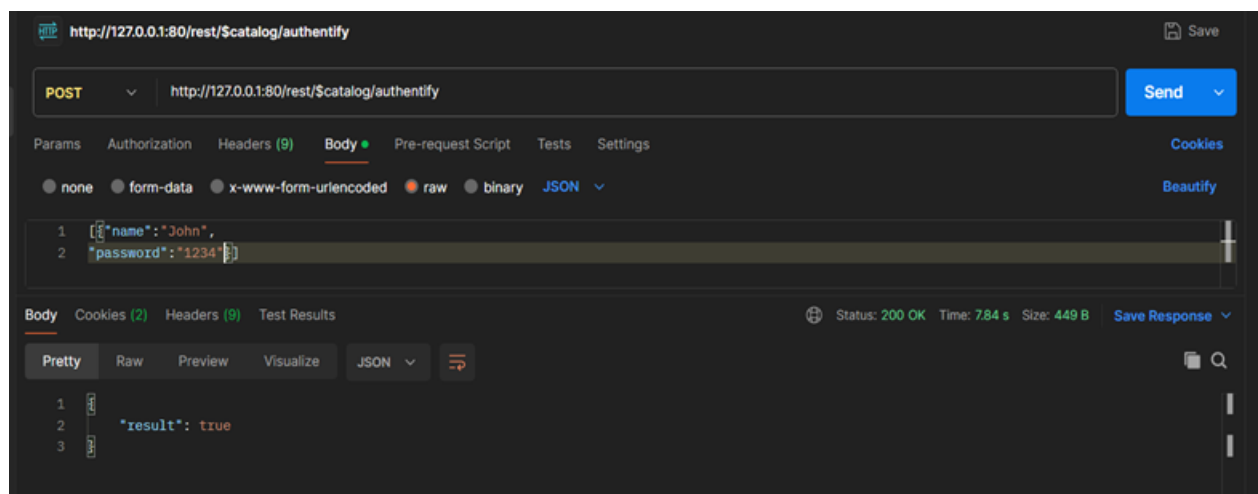
Method: POST

URL: http://localhost/rest/\$catalog/authenticate

Body:

- Select **raw** format
- Choose **JSON** from the dropdown
- Enter the following JSON:

```
{
  "name": "john ",
  "password": "1234"
}
```



What happens:

1. The server receives the credentials
2. The authenticate() function validates the identifier and password
3. If valid, Session.setPrivileges() is called to assign privileges
4. A 4D license is consumed
5. A session cookie is returned in the response headers
6. **Postman automatically captures and stores this session cookie**
7. The session is now authenticated and can access protected resources

Role.JSON Configuration

For these examples, the following roles.json configuration is used:

```
{
  "forceLogin": true,
  "permissions": {
    "allowed": [
      {
        "applyTo": "Company",
        "type": "dataclass",
        "read": [
          "AccessTOCOMPANY"
        ],
        "create": [
          "AccessTOCOMPANY"
        ],
        "drop": [
          "AccessTOCOMPANY"
        ]
      },
      {
        "applyTo": "Company.getCompanyName",
        "type": "method",
        "execute": [
          "AccessTOCOMPANY"
        ],
        "promote": [
          "AccessTOCOMPANY"
        ]
      },
      {
        "applyTo": "ds",
        "type": "datastore",
        "read": [
          "none"
        ],
        "create": [
          "none"
        ],
        "update": [
          "none"
        ],
        "drop": [
          "none"
        ],
        "execute": [
          "none"
        ]
      },
      {
        "applyTo": "ds.hasPrivilege",
        "type": "method",
```

```

        "execute": [
            "Admin"
        ]
    },
    {
        "applyTo": "ds.getPrivileges",
        "type": "method",
        "execute": [
            "Admin"
        ]
    }
]
},
"privileges": [
    {
        "privilege": "AccessTOCOMPANY",
        "includes": []
    },
    {
        "privilege": "none",
        "includes": []
    },
    {
        "privilege": "Admin",
        "includes": [
            "AccessTOCOMPANY"
        ]
    }
]
},
"roles": []
}

```

Understanding the Configuration

Privileges Defined

- **AccessTOCOMPANY**: A basic privilege that grants access to the Company dataclass
- **none**: The default locking privilege that denies all access
- **Admin**: An administrative privilege that includes the AccessTOCOMPANY privilege

Permissions Structure

Datastore Level (ds):

- All operations (read, create, update, drop, execute) are locked with the "none" privilege. This means by default, no data access is allowed

Company Dataclass:

- read: Requires "AccessTOCOMPANY" privilege
- create: Requires "AccessTOCOMPANY" privilege
- drop: Requires "AccessTOCOMPANY" privilege
- update: Not explicitly defined, so it inherits from datastore (denied)

Company.getCompanyByName() Method:

- **execute**: Requires "AccessTOCOMPANY" privilege
- **promote**: "AccessTOCOMPANY" privilege is temporarily promoted during execution

Administrative Methods:

- ds.hasPrivilege(): Requires "Admin" privilege to execute
- ds.getPrivileges(): Requires "Admin" privilege to execute

Privilege Hierarchy

- The "Admin" privilege includes the "AccessTOCOMPANY" privilege, meaning:
- A user with "Admin" privilege automatically has access to everything "AccessTOCOMPANY" provides
- A user with only "AccessTOCOMPANY" privilege cannot execute admin-specific methods like ds.hasPrivilege() or ds.getPrivileges()

Using Custom DataStore Functions

To illustrate this example, two methods “**hasPrivilege()** and **getPrivileges()**” are created in the **DataStoreImplementation** class, and execution permission is granted exclusively to users with the **Admin** privilege.

```
Class extends DataStoreImplementation

v exposed Function authenticate($credentials : Object) : Boolean
  v If ($identifier#{})
    v If (Validate password($credentials.name; $credentials.password))
      v If (User in group($credentials.name; "RestAccess"))
        return Session.setPrivileges("Admin")
      End if
    End if
  End if

v exposed Function hasPrivilege($nom : Text) : Boolean
  return Session.hasPrivilege($nom)

v exposed Function getPrivileges() : Collection
  return Session.getPrivileges()
```

Checking Privileges with ds.hasPrivilege()

The request needs to be configured as follows:

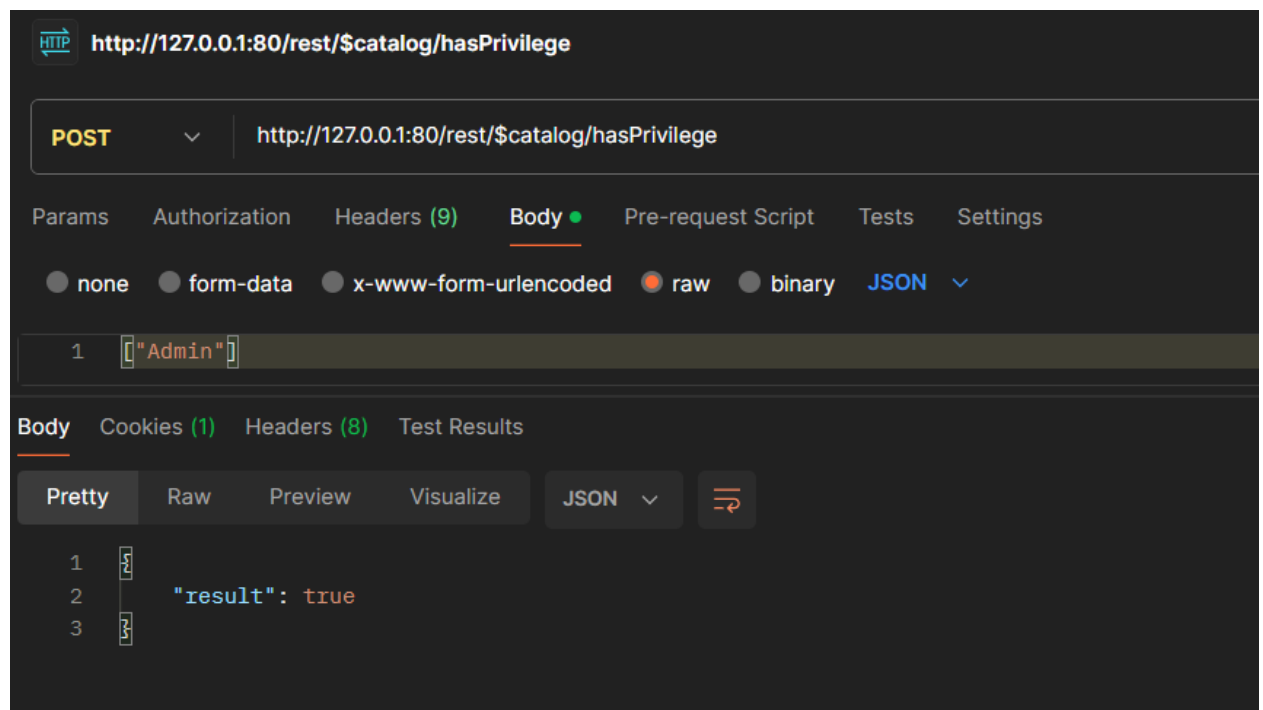
Method: POST

URL: [http://localhost/rest/\\$catalog/hasPrivilege](http://localhost/rest/$catalog/hasPrivilege)

Body: ["Admin"]

In the roles.json file, this function is configured as:

```
{
    "applyTo": "ds.hasPrivilege",
    "type": "method",
    "execute": [
        "Admin"
    ]
},
```



Since the authenticated session has "Admin" privilege, the method returns true.

Getting All Session Privileges with ds.getPrivileges()

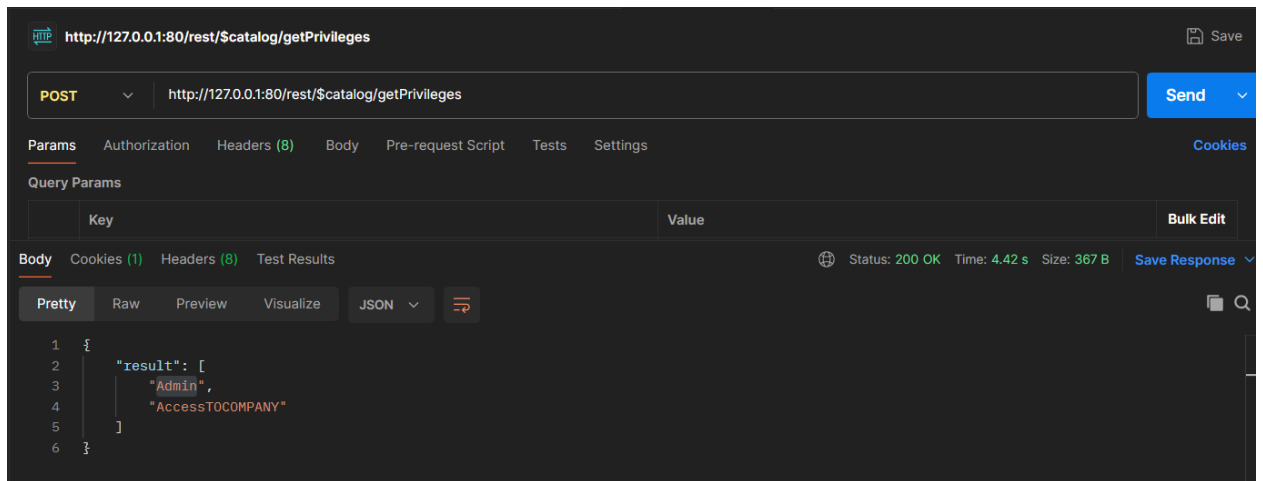
The request needs to be configured as follows:

Method: POST

URL: [http://localhost/rest/\\$catalog/getPrivileges](http://localhost/rest/$catalog/getPrivileges)

In the roles.json file, this function is configured as:

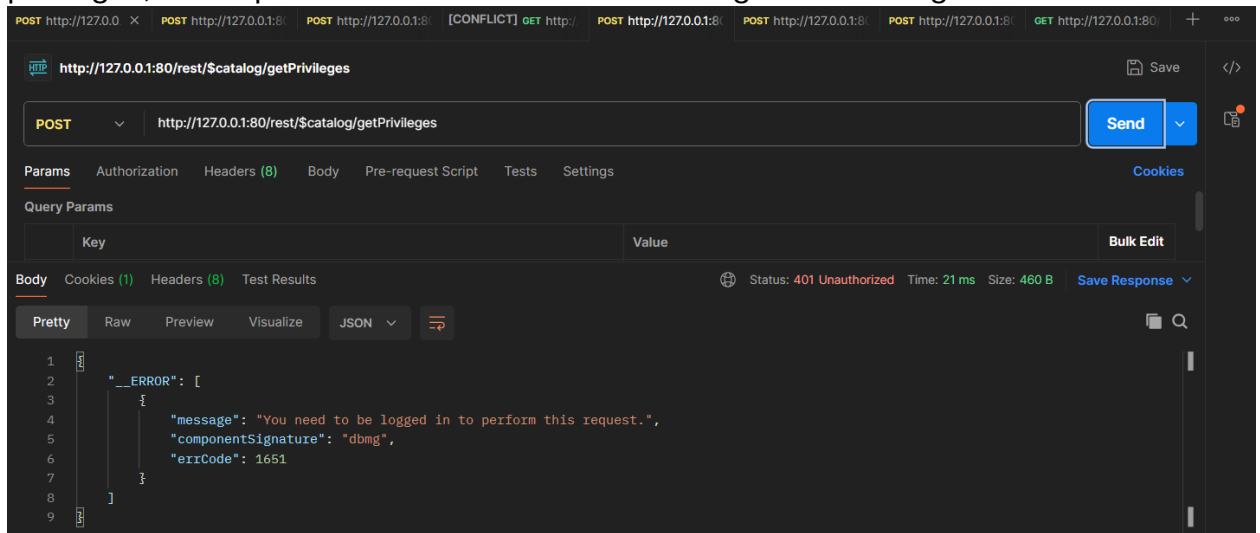
```
{
    "applyTo": "ds.getPrivileges",
    "type": "method",
    "execute": [
        "Admin"
    ]
}
```



Since the authenticated session has "Admin" privilege, The method returns all privileges assigned to the current session, showing both "Admin" and its included privilege "AccessTOCOMPANY"

Attempting to Call Admin Methods Without Admin Privilege

When a guest user attempts to execute these methods without the necessary privileges, the request fails and returns the following error message:



Fetching Data After Authentication

Once authenticated with the "Admin" privilege (which includes "AccessTOCOMPANY"), the session can retrieve data from the Company dataclass.

Example 1: Fetching All Companies

This example demonstrates how to retrieve all records from the **Company** dataclass using a REST request.

Method: GET

URL: http://localhost/rest/Company

```
http://127.0.0.1:80/rest/Company

GET http://127.0.0.1:80/rest/Company Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Body Cookies (1) Headers (8) Test Results Status: 200 OK Time: 13 ms Size: 965 B Save Response

Pretty Raw Preview Visualize JSON

1 2
2  {"__DATACLASS": "Company",
3  "__entityModel": "Company",
4  "__GlobalStamp": 0,
5  "__COUNT": 3,
6  "__FIRST": 0,
7  "__ENTITIES": [
8    {
9      "__KEY": "1",
10     "__TIMESTAMP": "2025-10-10T11:36:16.238Z",
11     "__STAMP": 1,
12     "ID": 1,
13     "companyName": "OpenAI",
14     "email": "contact@openai.com",
15     "city": "San Francisco",
16     "createdAt": "10!10!2025"
17   },
18   {
19     "__KEY": "2",
20     "__TIMESTAMP": "2025-10-13T12:56:57.198Z",
21     "__STAMP": 1,
22     "ID": 2,
23     "companyName": null,
24     "email": "info@globalindustries.com",
```

The request succeeds because the authenticated session has the "AccessTOCOMPANY" privilege, which grants **read** permission on the Company dataclass.

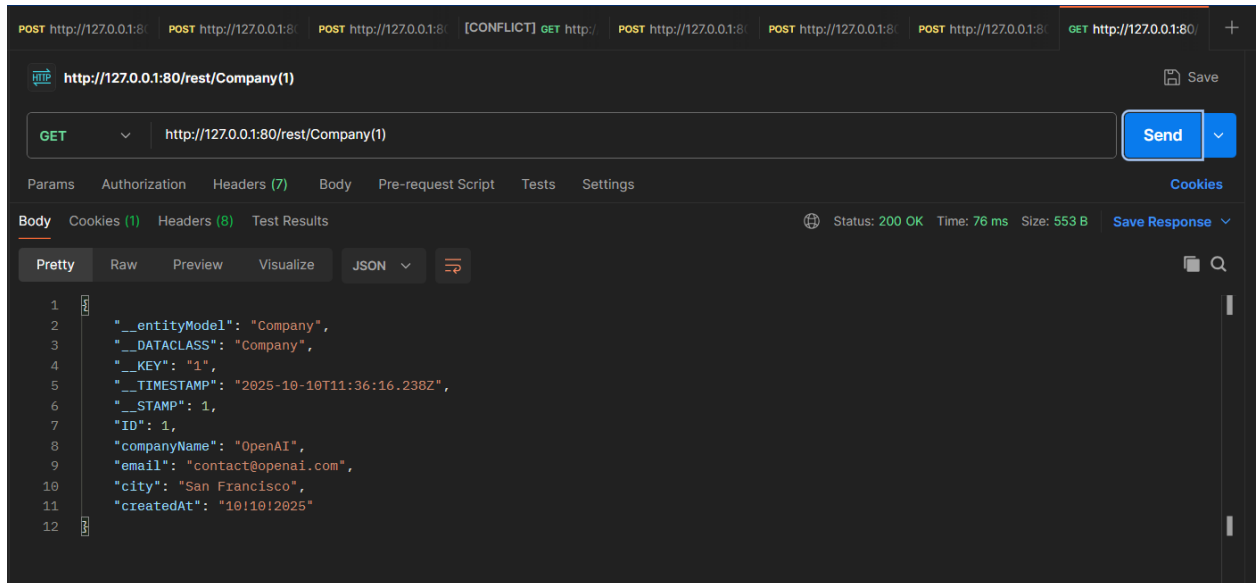
Example 2: Fetching a Specific Company by Key

This example demonstrates how to retrieve a specific **Company** entity by its primary key using a REST request.

The request targets the record whose key (or ID) is **1** in the **Company** dataclass.

Method: GET

URL: http://localhost/rest/Company(1)



The request succeeds because the authenticated session has the **AccessTOCOMPANY** privilege, which grants read permission on individual records of the **Company** dataclass.

Example 3: Querying Companies with class function

In addition to standard REST queries, data can be retrieved by calling exposed dataclass functions. This example demonstrates how to use the `Company.getCompanyByName()` method, which is configured in the `roles.json` with **execute** and **promote** permissions.

Request in Postman

Method: POST

URL: <http://localhost/rest/Company/getCompanyByName>

Body:

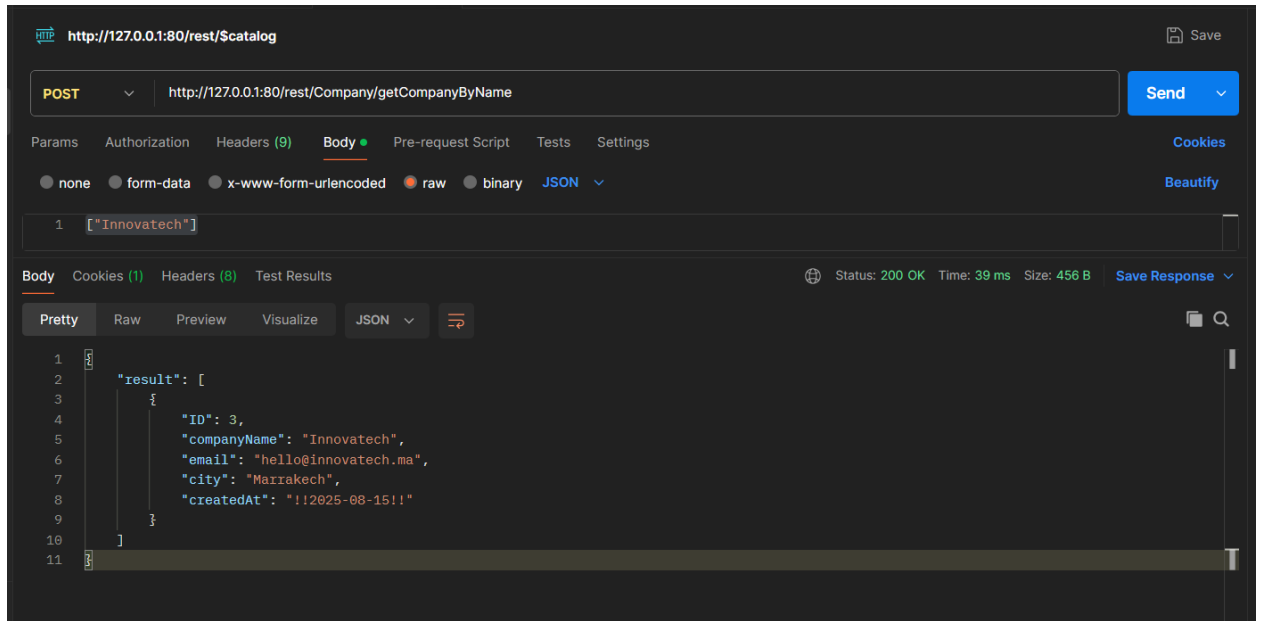
```
["Innovatech"]
```

The Function:

This function searches the `Company` dataclass for records whose `companyName` matches the provided parameter and returns the corresponding collection.

```
exposed Function getCompanyByName($name : Text)
  return This.query("companyName = :1"; $name).toCollection()
```

The result :



In the roles.json file, this function is configured as:

```
{\"applyTo\": \"Company.getCompanyByName\",  
  \"type\": \"method\",  
  \"execute\": [  
    \"AccessTOCOMPANY\"  
  ],  
  \"promote\": [  
    \"AccessTOCOMPANY\"  
  ]  
}
```

- **execute:** Users with the "AccessTOCOMPANY" privilege can call this method
- **promote:** The "AccessTOCOMPANY" privilege is temporarily promoted during the function's execution, allowing it to perform operations that might otherwise be restricted

Creating a New Company Record

After authentication with the "Admin" privilege, new Company entities can be created since the configuration grants **create** permission to users with the "AccessTOCOMPANY" privilege.

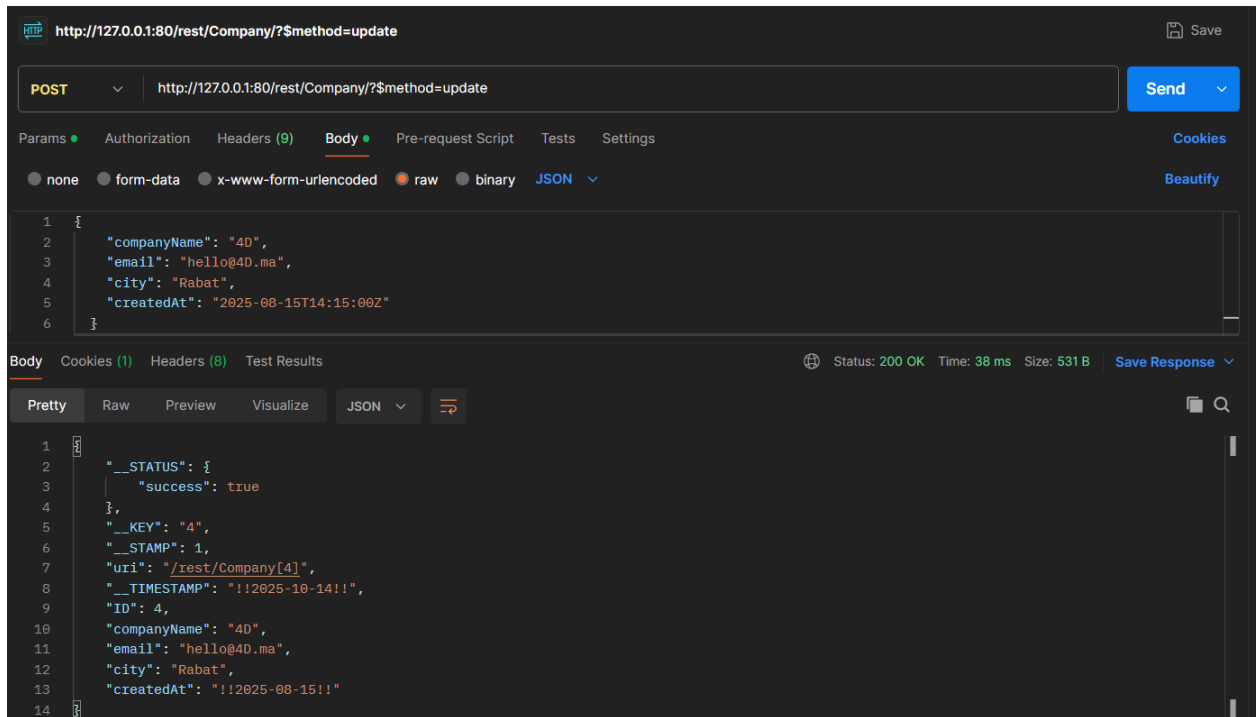
Request

Method: POST

URL: [http://127.0.0.1:80/rest/Company/?\\$method=update](http://127.0.0.1:80/rest/Company/?$method=update)

Body:

```
{
  "companyName": "4D",
  "email": "hello@4D.ma",
  "city": "Rabat",
  "createdAt": "2025-08-15T14:15:00Z"
}
```



What happens:

1. The server validates that the session has the "AccessTOCOMPANY" privilege
2. The **create** permission is granted for the Company dataclass
3. A new entity is created with the provided data
4. The server returns the complete entity with the generated __KEY and __STAMP

Updating an Existing Company Record

Important Note: In the current roles.json configuration, the **update** permission is **NOT explicitly granted** for the Company dataclass. This means update operations will be **denied** even for authenticated users with "AccessTOCOMPANY" privilege.

In the roles.json file, the Company dataclass is configured as follows:

```
{
    "applyTo": "Company",
    "type": "dataclass",
    "read": [
        "AccessTOCOMPANY"
    ],
    "create": [
        "AccessTOCOMPANY"
    ],
    "drop": [
        "AccessTOCOMPANY"
    ]
},
```

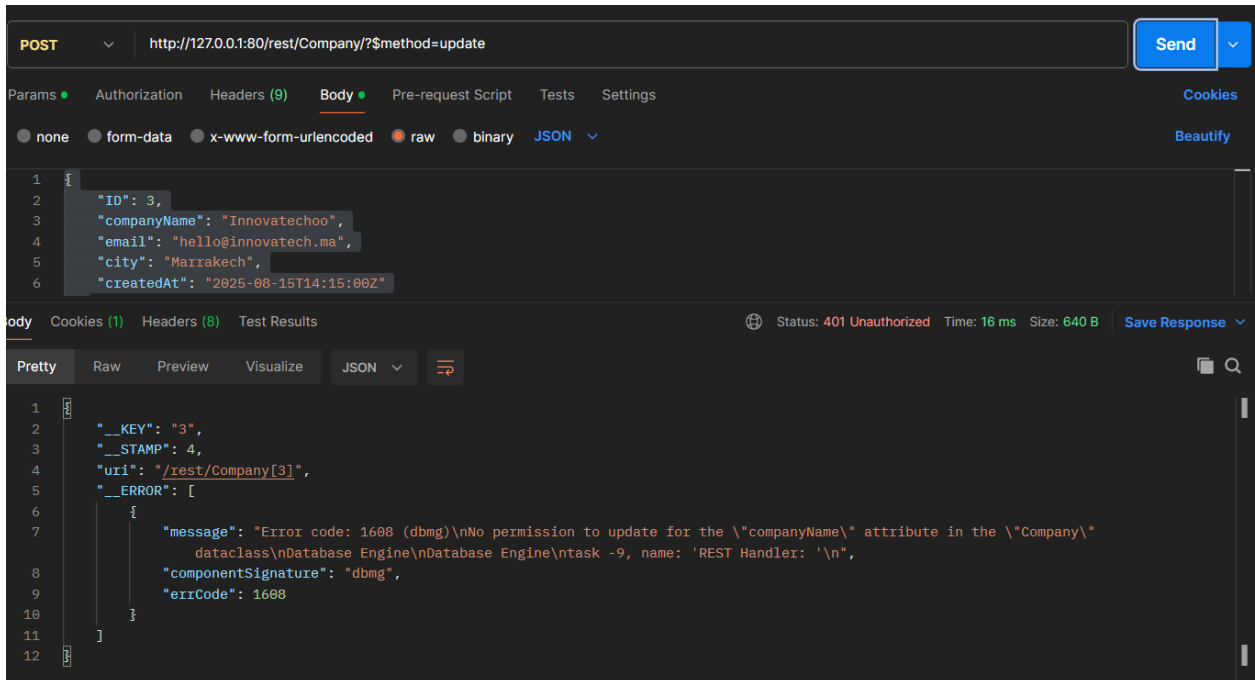
Request in Postman

Method: POST

URL: [http://127.0.0.1:80/rest/Company/?\\$method=update](http://127.0.0.1:80/rest/Company/?$method=update)

Body:

```
{
  "ID": 3,
  "companyName": "Innovatechoo",
  "email": "hello@innovatech.ma",
  "city": "Marrakech",
  "createdAt": "2025-08-15T14:15:00Z"
}
```



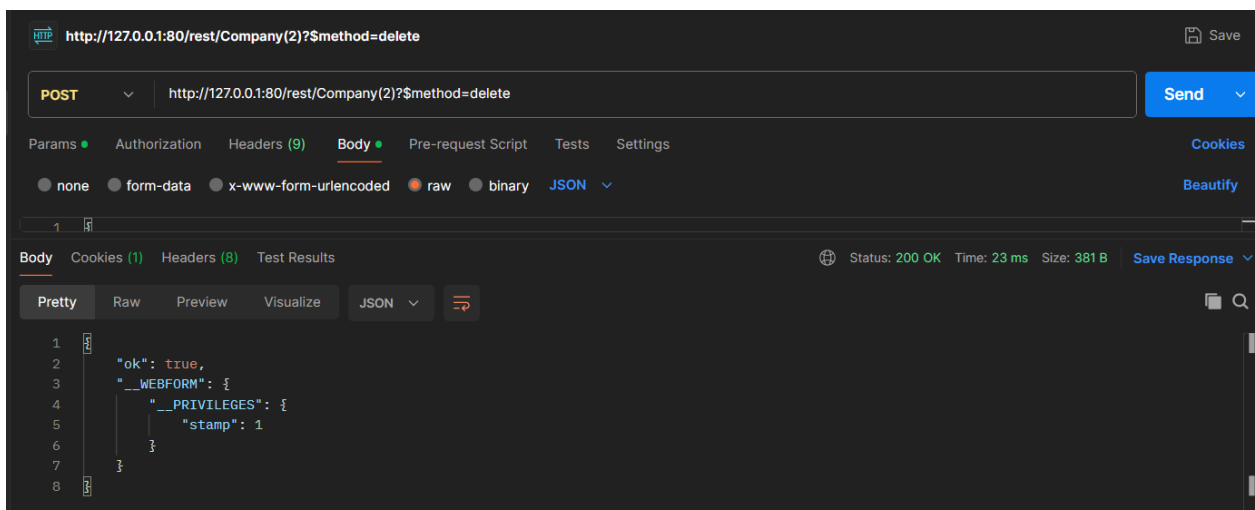
Deleting a Company Record

The user can delete Company entities because the **drop** permission is granted to users with the "AccessTOCOMPANY" privilege.

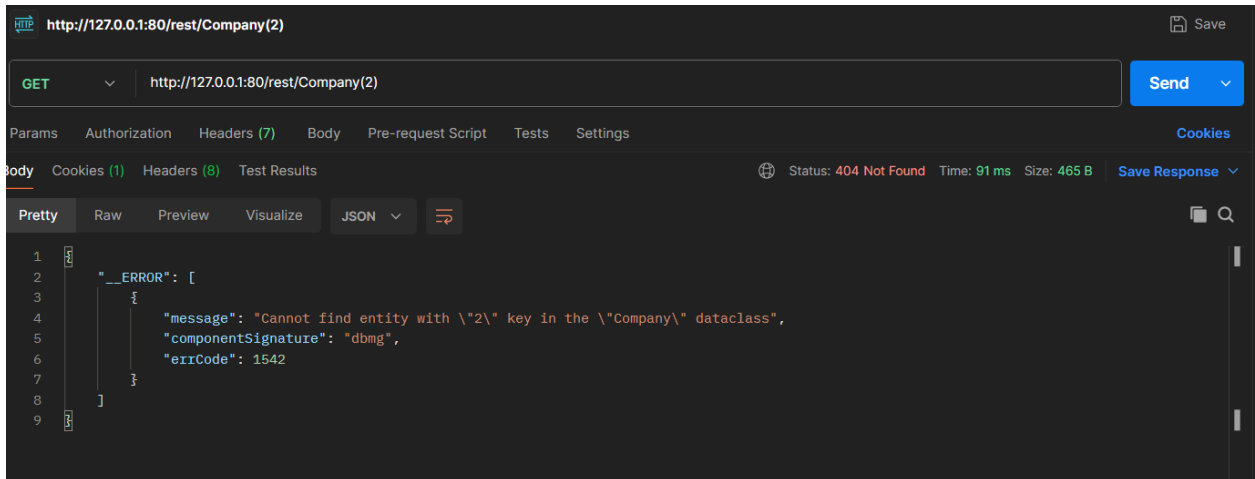
Request

Method: POST

URL: [http://localhost/rest/Company\(2\)?\\$method=delete](http://localhost/rest/Company(2)?$method=delete)



The deletion was successful, and the entity no longer exists in the table.



Conclusion

Force Login mode in 4D brings a modern, secure-by-default foundation for REST API access. It separates authentication from license usage, enforces explicit privilege control, and offers a flexible, role-based permission system. By adopting Force Login, developers ensure that their 4D applications are secure, scalable, and aligned with the best practices of modern web development.