

# Documentum REST Extensibility Tutorial (10): REST Error Handling

Version 1

created by William Zhou on Aug 12, 2016 4:36 AM, last modified by William Zhou on Aug 12, 2016 6:09 AM



**Previous:** [Documentum REST Extensibility Tutorial \(9\): Create Persistence Managers](#)

## Preliminary

Before diving into this tutorial, you must at least complete the tutorial [Documentum REST Extensibility Tutorial \(2\): Create A Sample Project](#).

## Overview

In this tutorial, we will learn how to handle exceptions in Documentum REST Services.

Here we will design a very simple resource **Who Am I**.

Resource	URI	HTTP Method	Mime Type
WhoAmI	/repositories/{repositoryName}/whoami	GET	application/xml application/json

The resource accepts a query parameter **name**.

- When the name is missing or empty, the server returns an error
- When the name length is exceeding 16, the server returns an error
- When the name is not current login user, the server returns an error
- When the name is equaling to login user name, the returns the login user information

We are going to explore different error handling technologies to for the 3 errors. But before the error handling, let's implement the normal resource function.

## Basic Implementation

Here is the sheet of files we are going to modify in the project created by [Documentum REST Extensibility Tutorial \(2\): Create A Sample Project](#). The project location is at `<SDK_ROOT>/maven-kit/generated/acme-rest/`. The file name with asterisk (\*) means the file is newly added.

- `acme-rest-resource/src/main/java/com/acme/rest/controller/WhoAmIController.java*`
- `acme-rest-resource/src/main/java/com/acme/rest/view/impl/CustomUserView.java*`
- `acme-rest-web/src/main/resources/com/emc/documentum/rest/script/rest-api-custom-resource-registry.yaml`

### WhoAmIController

```
01. @Controller("acme#whoami")
02. @RequestMapping("/repositories/{repositoryName}/whoami")
03. @ResourceViewBinding(CustomUserView.class)
04. public class WhoAmIController extends AbstractController {
```

```

05.
06.
07.     @RequestMapping(
08.         method = RequestMethod.GET,
09.         produces = {
10.             MediaType.APPLICATION_JSON_VALUE,
11.             MediaType.APPLICATION_XML_VALUE
12.         }
13.     )
14.     @ResponseBody
15.     @ResponseStatus(HttpStatus.OK)
16.     public UserObject whoami(
17.         @PathVariable("repositoryName") final String repositoryName,
18.         @RequestParam(value = "name", required = false) final String who,
19.         @RequestUri final UriInfo uriInfo)
20.         throws Exception {
21.         validate(who);
22.         UserObject user = currentUser();
23.         return getRenderedObject(repositoryName, user, true, uriInfo, null);
24.     }
25.
26.
27.     private UserObject currentUser() {
28.         UserObject user = new UserObject();
29.         user.addAttribute(new Attribute<Object>("user_login_name", RepositoryContextHolder.getLoginName());
30.         user.addAttribute(new Attribute<Object>("user_name", RepositoryContextHolder.getUserName());
31.         user.addAttribute(new Attribute<Object>("auth_type", RepositoryContextHolder.getAuthType());
32.         user.addAttribute(new Attribute<Object>("user_privileges", RepositoryContextHolder.getUserPrivileges());
33.         return user;
34.     }
35. }

```

### CustomUserView

```

01.     @DataViewBinding(modelType = UserObject.class)
02.     public class CustomUserView extends LinkableView<UserObject> {
03.         public CustomUserView (UserObject serializableData, UriInfo uriInfo, String repositoryName, Boolean returnLinks, others);
04.         super(serializableData, uriInfo, repositoryName, returnLinks, others);
05.     }
06.     @Override
07.     public void customize() {
08.     }
09.     @Override
10.     public String canonicalResourceUri(boolean validate) {
11.         return getUriFactory(validate).buildUriByTemplateName("X_WHOAMI_URI_TEMPLATE", null);
12.     }
13.     @Override
14.     protected Map<String, Object> resolveUriTemplateVariables(Map<String, String> valueMapping) {
15.         return Collections.emptyMap();
16.     }
17. }

```

### YAML Configuration

```

01. ---
02. uri-template-registry:
03.   - name: X_ALIAS_SETS_URI_TEMPLATE
04.     href: '{repositoryUri}/alias-sets{ext}'
05.   - name: X_ALIAS_SET_URI_TEMPLATE
06.     href: '{repositoryUri}/alias-sets/{aliasSetId}{ext}'
07. # added a URI template for resource who-am-i
08.   - name: X_WHOAMI_URI_TEMPLATE
09.     href: '{repositoryUri}/whoami{ext}'

```

After rebuilding and deploying the WAR, you can test the new **Who Am I** resource like this:

### Request

```
01. GET http://localhost:8080/acme-rest/repositories/REPO/whoami?name=dmadmin HTTP/1.1
02. Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
03. Content-Type: */*
```

### Response

```
01. HTTP/1.1 200 OK
02. Content-Type: application/json; charset=UTF-8
03.
04.
05. {
06.   "name": "user",
07.   "properties": { "user_login_name": "dmadmin", "user_name": "dmadmin", "auth_type": "PASSWORD", "user_privile
08.   "links": [ { "rel": "self", "href": "http://localhost:8080/acme-rest/repositories/REPO/whoami?name=dmadmin
09. }
```

This is the normal request and response. Now let's experiment the error handling when the parameter **name** is invalid.

## Error Handling Basic

Fundamentally, we should honor [HTTP/1.1: Status Code Definitions](#) to return the error response. Status code **4xx** means to client side errors and could be resolved when the client resubmit a new (different) request. Status code **5xx** means to server side errors and is usually not recoverable.

In your REST server implementation, if you throw a Java exception, Documentum REST server runtime will check the exception type. If it is a managed type, the server will return corresponding HTTP status and error details. For instance, **java.lang.IllegalArgumentException** is mapped to HTTP Status **400**. If the exception type is unknown, the server returns a generic status code **500**.

**500** is not friendly for most REST APIs since it means the client can not recover the request. In the following sections, we will look at different means to handle errors in Documentum REST so that we can manage the error code and error representation for our new resource **Who Am I**.

## Error Handling 1 - Throw and Catch Exceptions

The first example is to handle the error of missing parameter or the parameter is empty. Here we create a new Java exception type for this validation.

### Add to WhoAmIController

```
01. private void validate(String who) {
02.     if (Strings.isNullOrEmpty(who)) {
03.         throw new EmptyUserException("The parameter 'name' can not be null or empty");
04.     }
05. }
06.
07.
08. private static class EmptyUserException extends RuntimeException {
09.     public EmptyUserException(String message) {
```

```

10.         super(message);
11.     }
12. }

```

But then how will the server handle the **EmptyUserException**. **Exception Handling in Spring MVC** tells how to handle exceptions in controllers. We shall follow the same way to catch the exception. If you do not handle this **EmptyUserException**, Documentum REST uses its own error handler to return HTTP Status **500** for Java exception type **RuntimeException**.

Here is the code to handle this **EmptyUserException**. In the code, we return HTTP Status **400**, and a REST error representation aligning with Core REST.

#### Add to WhoAmIController

```

01. @ResponseBody
02. @ResponseStatus(HttpStatus.BAD_REQUEST)
03. @ExceptionHandler({EmptyUserException.class})
04. public RestError onEmptyUserException(EmptyUserException e) {
05.     return createRestError(HttpStatus.BAD_REQUEST, "E_EMPTY_USER_NAME", e);
06. }
07.
08. private RestError createRestError(HttpStatus status, String code, Exception e, String... details) {
09.     RestError error = new RestError();
10.     error.setStatus(status.value());
11.     error.setCode(code);
12.     error.setMessage(e.getMessage());
13.     if (details != null) {
14.         for (String detail : details) {
15.             error.addDetail(detail);
16.         }
17.     }
18.     return error;
19. }

```

Please note the class **come.emc.documentum.rest.model.RestError** is a Documentum Core REST error model for all error representations. You can return it as your error body.

Now after rebuilding and deploying the WAR, you can test the new **Who Am I** resource with an empty **name**:

#### Request

```

01. GET http://localhost:8080/acme-rest/repositories/REPO/whoami?name= HTTP/1.1
02. Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
03. Content-Type: */*

```

#### Response

```

01. HTTP/1.1 400 Bad Request
02. Content-Type: application/vnd.emc.documentum+json;charset=UTF-8
03.
04.
05. {
06.     "status":400,
07.     "code":"E_EMPTY_USER_NAME",
08.     "message":"The parameter 'name' can not be null or empty"
09. }

```

## Error Handling 2 - Core REST Support

There are 3 small issues with Error handling 1:

- We have to create new Java exceptions
- We have to handle new Java exceptions by our own codes
- Message localization is not implemented

All above issues can be resolved by reusing Core REST exception

type `com.emc.documentum.rest.error.RestClientErrorException`. In your resource controller, you can throw `RestClientErrorException` and Core REST will handle it for you. In the exception constructor, you can specify the HTTP Status, error code, messages and details.

Now let's use this exception type to handle the error of mismatching user name. We want to return HTTP Status **403** for this error.

#### Add if() to validate()

```
01. private void validate(String who) {
02.     if (Strings.isNullOrEmpty(who)) {
03.         throw new EmptyUserException("The parameter 'name' can not be null or empty");
04.     }
05.     if (!who.equals(RepositoryContextHolder.getUserName())) {
06.         throw new RestClientErrorException("E_DISALLOWED_USER_NAME", new Object[]{who}, HttpStatus.FOR
07.     }
08. }
```

For the message localization, you need to create a new properties file to map the error code `E_DISALLOWED_USER_NAME` to a local message.

- Create a file `acme-rest-resource/src/main/resources/com/acme/rest/messages/rest-api-acme-messages.properties`

```
01. E_DISALLOWED_USER_NAME=The specified user {0} is not current user.
```
- Create or update the file `acme-rest-resource/src/main/resources/rest-api-runtime.properties`

```
01. rest.extension.message.packages=com.acme.rest.messages
```

Now after rebuilding and deploying the WAR, you can test the new **Who Am I** resource with an invalid **name**:

#### Request

```
01. GET http://localhost:8080/acme-rest/repositories/REPO/whoami?name=stranger HTTP/1.1
02. Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
03. Content-Type: */*
```

#### Response

```
01. HTTP/1.1 403 Bad Request
02. Content-Type: application/vnd.emc.documentum+json; charset=UTF-8
03.
04. {
05.     "status":403,
06.     "code":"E_DISALLOWED_USER_NAME",
07.     "message":"The specified user stranger is not current user."
08. }
```

## Error Handling 3 - Customizing Error Representation

By Error handling 2, you have well mastered the error handling for Documentum REST Extensibility development. However, you may still feel unsatisfied with Core REST error representation serialized from **RestError**. In this section, we will show you another way to extend the error representation, by adding your data into the error response body. We will use this approach to handle the name too long error.

To return additional data in the error response, we need to extend Core RestError model. We will add a **hint** attribute in the error response.

- Create a new class at *acme-rest-model/src/main/java/com/acme/rest/model/AcmeRestError*

```

01. @SerializableViewType(
02.     inheritValue = true,
03.     fieldVisibility = SerializableType.FieldVisibility.NONE,
04.     fieldOrder = {"status", "code", "message", "details", "hint"},
05.     xmlns = "http://identifiers.emc.com/vocab/documentum",
06.     xmlnsPrefix = "dm"
07. )
08. public class AcmeRestError extends RestError {
09.     @SerializableField
10.     private String hint;
11.     public String getHint() {
12.         return hint;
13.     }
14.     public void setHint(String hint) {
15.         this.hint = hint;
16.     }
17. }

```

Then we need to create a class to convert the custom exception into the new REST error model.

- Create a new class at *acme-rest-resource/src/main/java/com/acme/rest/util/AcmeRestErrorConverter*

```

01. public class AcmeErrorConverter {
02.     private final Exception exception;
03.     private final HttpStatus status;
04.     private final String errorCode;
05.     private final String hint;
06.     private final Object[] args;
07.     public AcmeErrorConverter(Exception e, HttpStatus status, String hint, String code, Object... args) {
08.         this.exception = e;
09.         this.status = status;
10.         this.hint = hint;
11.         this.errorCode = code;
12.         this.args = args;
13.     }
14.     public AcmeRestError convert() {
15.         AcmeRestError error = new AcmeRestError();
16.         error.setHint(hint);
17.         error.setStatus(status.value());
18.         error.setCode(errorCode);
19.         error.setMessage(MessageBundle.INSTANCE.get(errorCode, args));
20.         error.addDetail(exception.getLocalizedMessage());
21.         return error;
22.     }
23. }

```

After that, we add the error handling logic into the controller class. We create a new exception type **NameTooLongException**, and add a method to handle it. For the error response, we will return a hint **THIS IS A CUSTOM REST ERROR!**

### Add to WhoAmIController

```

01. private void validate(String who) {
02.     if (Strings.isNullOrEmpty(who)) {
03.         throw new EmptyUserException("The parameter 'name' can not be null or empty");
04.     }
05.     if (who.length() > 16) {
06.         throw new NameTooLongException(who);
07.     }
08.     if (!who.equals(RepositoryContextHolder.getUserName())) {
09.         throw new RestClientErrorException("E_DISALLOWED_USER_NAME", new Object[]{who}, HttpStatus.FOR
10.     }
11. }
12.
13. private static class NameTooLongException extends RuntimeException {
14.     private String name;
15.     public NameTooLongException(String name) {
16.         this.name = name;
17.     }
18.     public String getName() {
19.         return name;
20.     }
21. }
22.
23. @ResponseBody
24. @ResponseStatus(HttpStatus.CONFLICT)
25. @ExceptionHandler({NameTooLongException.class})
26. public AcmeRestError onNameTooLongException(NameTooLongException e) {
27.     return new AcmeErrorConverter(e, HttpStatus.CONFLICT, "THIS IS A CUSTOM REST ERROR!", "E_NAME_TOO_
28. }

```

The last is to add a localizable message mapping for code **E\_NAME\_TOO\_LONG**.

- Modify the file *acme-rest-resource/src/main/resources/com/acme/rest/messages/rest-api-acme-messages.properties*

```

01. #added for error handling - 2
02. E_DISALLOWED_USER_NAME=The specified user {0} is not current user.
03. #added for error handling - 3
04. E_NAME_TOO_LONG=The specified name {0} has exceeded the length limit 16.

```

Now after rebuilding and deploying the WAR, you can test the new **Who Am I** resource with a long long name:

#### Request

```

01. GET http://localhost:8080/acme-rest/repositories/REPO/whoami?name=0123456789abcdefg HTTP/1.1
02. Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
03. Content-Type: */*
01.

```

#### Response

```

01.
01. HTTP/1.1 409 Bad Request
02. Content-Type: application/vnd.emc.documentum+json;charset=UTF-8
03.
04.
05. {
06.     "status":409,
07.     "code":"E_NAME_TOO_LONG",
08.     "message":"The specified name 0123456789abcdefg has exceeded the length limit 16.",
09.     "hint":"THIS IS A CUSTOM REST ERROR!"
10. }

```

We are done here for this tutorial.

Please stay tuned for more tutorials on this series. If you have already had some experience with Documentum REST extensibility, please put your comments below the page to let us know which topics you are most interested in.

[Learn more about Documentum REST Services >>](#)

