

# Documentum TBO, SBO, Aspect and REST Extensibility, Part III

Version 1

created by William Zhou on Jul 19, 2017 5:28 AM, last modified by William Zhou on Jul 20, 2017 6:28 AM

rest-version 7.3+

In this series, we will walk through Documentum full-stack extensibility from Business Object Framework (BOF) to REST API development. The tutorials are divided into 3 parts:

- [Part I](#): Design user story and install BOF
- [Part II](#): Consume Type-based Business Object (TBO) / Aspect by default REST API
- [Part III](#): Design fine-grained REST API for TBO / Service-based Business Object (SBO) / Aspect

This article is the last part of the series, talking about how to design a fine-grained REST API to consume custom types.

## Preliminary

Please make sure you have installed all the artifacts following [Part I: Design user story and install BOF](#). It's also good to walk through [Part II](#) to get well understanding of the context.

The REST extension project code can be downloaded from GitHub: [documentum-rest-extensibility-samples/rest-bof-sample/acme-rest at master · Enterprise-Content-Management/documentum-res...](#)

The Postman script can be downloaded from GitHub: [documentum-rest-extensibility-samples/rest-bof-sample/postman-test at master · Enterprise-Content-Management/documentum-...](#)

## Design custom REST API

As said, we'll create a custom REST API to have fine-grained design for the **name\_card** object type. Here is the worksheet:

Resource Name	Resource URI	HTTP Methods	Description
Name Card Collection <i>acme#name-cards</i>	/repositories/{repositoryName}/name-cards	1. GET 2. POST	1. Get the collection of all name cards <ul style="list-style-type: none"><li>• Support customized entry summary</li></ul> 2. Create a new name card <ul style="list-style-type: none"><li>• Support to set aspect properties</li><li>• Support to create with photo</li><li>• Support to create with biography</li><li>• Support name validation</li></ul>
Name Card Instance <i>acme#name-card</i>	/repositories/{repositoryName}/name-cards/{name}	1. GET 2. POST 3. DELETE	1. Get a name card <ul style="list-style-type: none"><li>• Support direct links for photo and biography</li><li>• Support links to manager</li><li>• Support links to direct reports</li></ul> 2. Update a name card <ul style="list-style-type: none"><li>• Each update creates a new version</li></ul> 3. Delete a name card

## Create REST extension project

Please download the **acme-rest** project from GitHub. To build it, you need to install Documentum Maven artifacts first following [Documentum REST Extensibility Tutorial \(1\): Install Documentum REST Artifacts](#).

The code structure for **acme-rest** is same as a regular REST extension project, [Documentum REST Extensibility Tutorial \(3\): Explore The Sample Project](#).

## acme-rest-model

The only class in this module is **NameCard**. It is the REST resource model for a name card. When we put NameCard as a request body parameter in the controller, the REST server will be able to accept a multipart request with multiple contents (bio and photo) for the new name card.

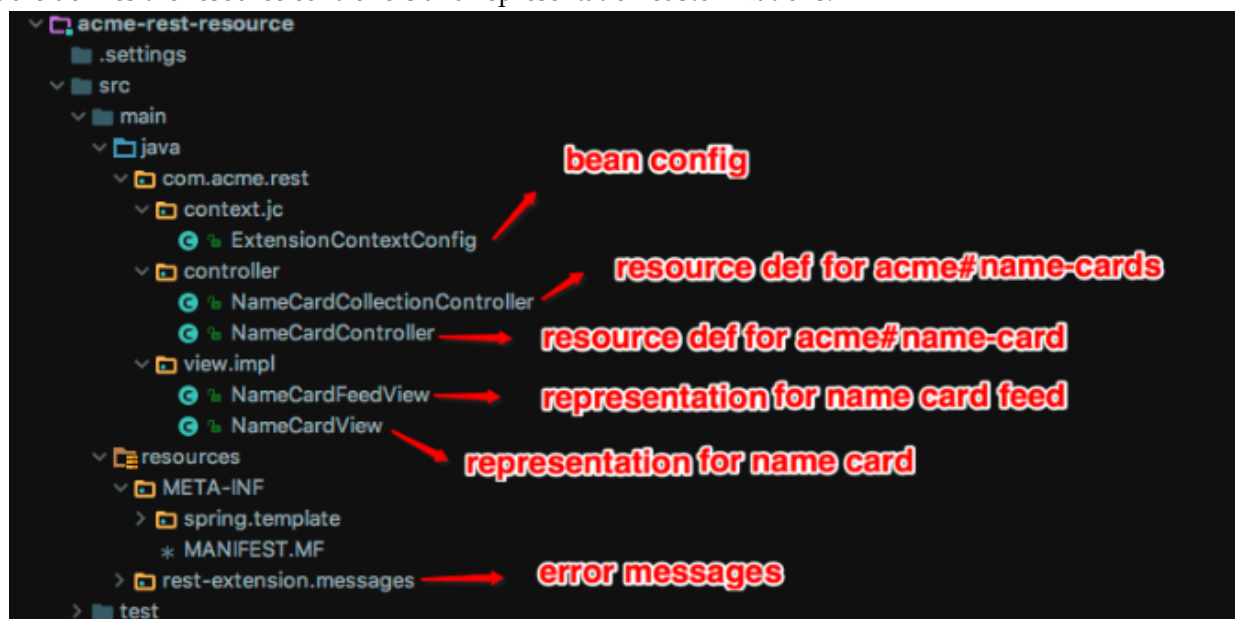
```

01. @SerializableType("name-card")
02. public class NameCard extends ContentfulObject {
03.     public NameCard() {
04.         setType("name_card");
05.     }
06.     ...
07. }

```

## acme-rest-resource

This module defines the resource controllers and representation customizations.



What needs to be emphasized is the class **NameCardView**. By the view class, we customize the **self** link for the name card resource.

```

01. @Override
02. public String canonicalResourceUri(boolean validate) {
03.     return nameUriBuilder("acme#name-card", getDataInternal().getName()).build();
04. }

```

Still in this class, we customize link relations for a name card. Please note 4 link relations are added to a name card resource by conditions.

- biography
- photo
- manager
- direct-reports

```

01. @Override
02. public void customize() {
03.     Integer contentSize = (Integer) getDataInternal().getMandatoryAttribute("r_content_size");
04.     if (contentSize != null && contentSize > 0) {
05.         makeLink("biography",

```

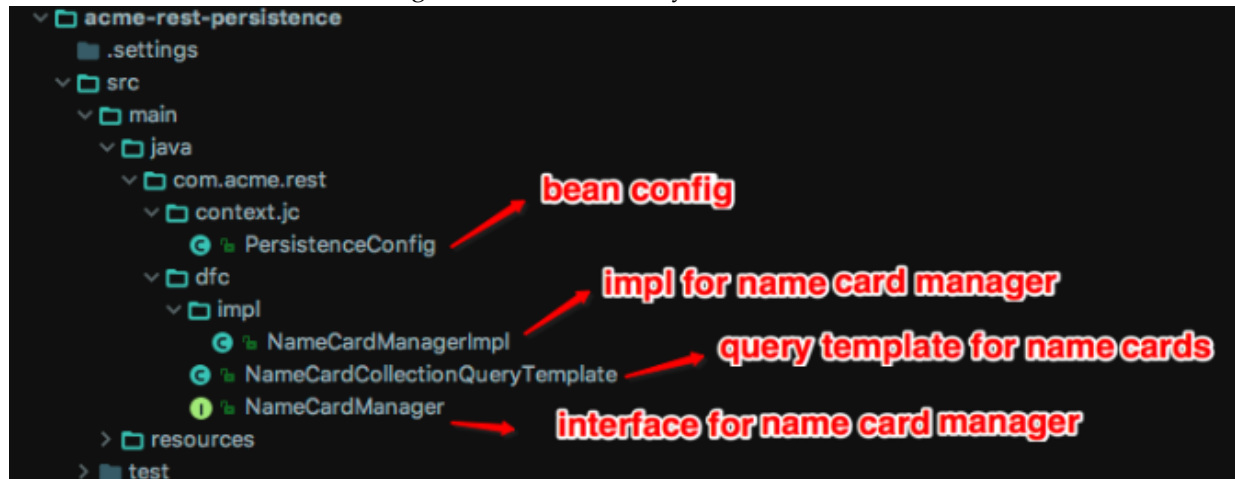
```

06.         idUriBuilder("content-media", getDataInternal().getId())
07.             .queryParams("format", "text")
08.             .queryParams("page", 0)
09.             .queryParams("modifier", "")
10.             .build();
11.     makeLink("photo",
12.         idUriBuilder("content-media", getDataInternal().getId())
13.             .queryParams("format", "png")
14.             .queryParams("page", 0)
15.             .queryParams("modifier", "")
16.             .build());
17.     }
18.     String manager = getDataInternal().getManager();
19.     if (StringUtils.isEmpty(manager)) {
20.         makeLink("manager",
21.             nameUriBuilder("acme#name-card", getDataInternal().getManager())
22.                 .build());
23.     }
24.     List<String> directReports = getDataInternal().getDirectReports();
25.     if (directReports != null && !directReports.isEmpty()) {
26.         makeLink("direct-reports",
27.             uriBuilder("acme#name-cards")
28.                 .queryParams("filter", "organization_aspect.manager='" + getDataInternal().getN
29.                 .build());
30.     }
31. }

```

## acme-rest-persistence

This module defines the DFC API call managers, which are used by resource controllers.



The name card creation method calls SBO code IDfDirectoryService to create a name card. This shows you an example to call SBO in REST server code.

### NameCardManagerImpl:

```

01. @Override public NameCard create(NameCard nameCard) throws DfException {
02.     ...
03.     IDirectoryService service = (IDirectoryService) CLIENT.getLocalClient().newService(
04.         IDirectoryService.class.getName(),
05.         closeableSession.getDfSession().getSessionManager());
06.     ...
07.     IDfPersistentObject card = service.create(
08.         RepositoryContextHolder.getRepositoryName(),
09.         getAttributesMap(nameCard),
10.         bio,

```

```

11.         photo);
12.     return new TypedObjectConverter(card, AttributeView.ALL).convert(NameCard.class);
13.     ...
14. }

```

In [Part I](#), we have talked about the necessary to have a SBO, but we did not tell the reason. Now let's look at the SBO code.

### DirectoryService:

```

01. public IDfPersistentObject create(String repositoryName, Map<String, Object> attrs,
02.     File bio, File photo) throws DfException {
03.     IDfSession session = null;
04.     try {
05.         session = getSessionManager().getSession(repositoryName);
06.         ISysObject obj = (ISysObject) session.newObject("name_card");
07.         setAspects(obj);
08.         setAttributes(obj, attrs);
09.         if (bio != null) {
10.             obj.setFileEx(bio.getAbsolutePath(), "text", 0, null);
11.         }
12.         if (photo != null) {
13.             obj.setFileEx(photo.getAbsolutePath(), "png", 0, null);
14.         }
15.         validate(session, obj);
16.         obj.save();
17.         return obj;
18.     } finally {
19.         if (session != null) {
20.             getSessionManager().release(session);
21.         }
22.     }
23. }

```

The **create()** method in the SBO does the follow things:

- Create a new name\_card object -- so that the client doesn't need to specify a type in requests
- Set aspects prior to set attributes -- so that the creation request can set aspect attributes
- Set attributes
- Set contents -- so that set biography and photo can be separated
- Validate object -- make sure no duplicated name cards with the same name

The other method we'll explore is **update()** in **NameCardManagerImpl**.

### NameCardManagerImpl:

```

01. @Override public NameCard update(final String name, final NameCard nameCard) throws DfException {
02.     final String id = validate(name, nameCard);
03.     nameCard.setId(id);
04.     return contextSessionManager.executeWithinTheContextTran(
05.         new ContextSessionManager.SessionCallable<NameCard>() {
06.             @Override public NameCard call(IDfSession session) throws DfException {
07.                 versioningManager.checkOut(id);
08.                 if (nameCard.hasContent()) {
09.                     return versioningManager.checkInWithRenditions(
10.                         id, nameCard, "", CheckinPolicy.NEXT_MAJOR, true, 0, false, "");
11.                 } else {
12.                     return versioningManager.checkIn(
13.                         id, nameCard, "", CheckinPolicy.NEXT_MAJOR, true, false, "");
14.                 }
15.             }
16.         }

```

```

17.     );
18.   }

```

By this implementation, the update of a name card will increase the version, and the checkout + checkin is executed within a transaction.

## acme-rest-web

The web module customizes the WAR configuration parameters. The most important ones are these.

### rest\_extension.yaml

```

01. ---
02. resource-link-registry:
03.   - resource:      repository
04.     link-relation: 'name-cards'
05.     target-resource: 'acme#name-cards'
06.     value-mapping: []
07.
08.
09. ---
10. media-type-registry:
11.   - media-type: json

```

The YAML file customizes the extensibility features. Here what it does is:

- Add a link relation **name-cards** on **Repository Resource**
- Make the entire REST API support **JSON** media type only

### rest-api-runtime.properties

```

01. # custom yaml file location for extensibility
02. rest.ext.yaml.package=yaml
03. # custom spring java config location
04. rest.context.config.location=com.acme.rest.context.jc
05. rest.extension.message.packages = rest-extension.messages
06. rest.ext.error.code.mapping.packages = rest-extension.messages

```

The properties file customizes the YAML and project code configurations for bean loading and message packages.

## Build and deploy REST extension project

Prior to build the project, you need to update the Documentum repository properties in the root **pom.xml**. The build process will populate these properties to **dfc.properties** file.

```

01. <test.docbroker.host>127.0.0.1</test.docbroker.host>
02. <test.docbroker.port>1489</test.docbroker.port>
03. <test.repository>ACME01</test.repository>
04. <test.username>dmdadmin</test.username>
05. <test.password>password</test.password>

```

Under the project root of **acme-rest**, please run command **mvn clean install** to build the project. The WAR file is built at **/acme-rest-web/target/acme-rest-web-1.0.0-SNAPSHOT.war**.

To keep using the consistent URLs in Postman test, it's better to rename it to **dctm-rest.war** before deploying it to a web server.

## Run REST requests

Now please follow Postman scripts to test the new REST API:

- c1. Get repository -- can find a link relation name-cards from the response
- c2. Get all name cards -- GET /repositories/{repositoryName}/name-cards
- c3. Create contentless name card -- POST /repositories/{repositoryName}/name-cards

- c4. Create contentful name card -- POST /repositories/{repositoryName}/name-cards with form data
- c5. Get name card -- GET /repositories/{repositoryName}/name-cards/{name}, please find the differences in links
- c6. Update name card -- POST /repositories/{repositoryName}/name-cards/{name}, please notice the updated version label
- c7. Delete name card -- DELETE /repositories/{repositoryName}/name-cards/{name}

Please note the new links collection on a name card resource.

```

01.  "links": [
02.      {
03.          "rel": "self",
04.          "href": "http://localhost:8080/dctm-rest/repositories/ACME01/name-cards/Cindy%2BBaker"
05.      },
06.      {
07.          "rel": "biography",
08.          "href": "http://localhost:8080/dctm-rest/repositories/ACME01/objects/0900000b80037eab/content-i
09.      },
10.      {
11.          "rel": "photo",
12.          "href": "http://localhost:8080/dctm-rest/repositories/ACME01/objects/0900000b80037eab/content-i
13.      },
14.      {
15.          "rel": "manager",
16.          "href": "http://localhost:8080/dctm-rest/repositories/ACME01/name-cards/Beth%2BWiggins"
17.      }
18.  ]

```

Now it's very straightforward to follow its link relations to navigate to other resources. Here is an open question for you:

How to design a REST API to add one user as the direct report of the other one?

## Summary

By the fine-grained REST API design, we found that the REST API calls from the client side would be much easier and more straightforward for the custom type objects. That's the value of the REST extensibility.

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