Date: ~~January~~April 202~~1~~2

REST for CORBA

V1.0 ~~– beta 1~~

OMG Document Number: ptc/202~~1~~2-0~~1~~4-0~~3~~2

Normative reference: https://www.omg.org/spec/CORBA-REST

This OMG document replaces the submission document (mars/2020-12-12). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by June 14, 2021.

You may view the pending issues for this specification from the OMG revision issues web page https://issues.omg.org/issues/lists.

The FTF Recommendation and Report for this specification will be published in December 2021. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 9C Medway Road, PMB 274, Milford, MA 01757, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

Index of Tables

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel™); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

https://www.omg.org/spec

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
9C Medway Road, PMB 274
Milford, MA 01757
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult https://www.iso.org

1 Scope

The REST for CORBA specification defines a standard and interoperable mechanism to enable CORBA objects to be exposed as REST services.

The REST for CORBA specification defines:

- a set of standard IDL annotations (IDL-RS) to decorate IDL constructs in order to expose the corresponding CORBA objects as REST services that can be consumed by REST client applications.
- Data Representation Formats to externalize objects defined with the IDL Type Representation format to JSON and XML Data Representation formats.

2 Conformance

This specification defines the following conformance points:

Table 2.1: Conformance Points

IDL-RS	Conformant implementations must comply with section 8
JSONDR	Conformant implementations must comply with section 9
XMLDR	Conformant implementations must comply with section 10

This specification defines the following compliance levels:

Table 2.2: Compliance Levels

IDL-RS	All conformant implementations must comply with the IDL-RS conformance point.
IDL-RS + JSONDR	Conformant implementations that provide a JSON representation of CORBA resources must comply with the IDL-RS and JSONDR conformance points.
IDL-RS + XMLDR	Conformant implementations that provide a JSON representation of CORBA resources must comply with the IDL-RS and XMLDR conformance points.
IDL-RS + JSONDR + XMLDR	Conformant implementations that provide a JSON representation of CORBA resources and an XML representation of CORBA resources must comply with the: IDL-RS, JSONDR, and XMLDR conformance points.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

[IDL] Object Management Group, Interface Definition Language, Version 4.2 (https://www.omg.org/spec/IDL/4.2) – IDL is a descriptive language used to define data types and interfaces in a way that is independent of the programming language or operating system/processor platform.

[CORBA] Object Management Group, Common Object Request Broker Architecture, Version 3.3 (https://www.omg.org/spec/CORBA/3.3) - OMG specification: formal/2012-11-12 (part1), formal/2012-11-14 (part2).

[C2WSDL] Object Management Group, CORBA to WSDL/SOAP Interworking Specification, Version 1.2.1 (https://www.omg.org/spec/C2WSDL), formal/08-08-03 – Defines a mapping from IDL to XmlSchema and provides a natural mapping from IDL to WSDL that is also suitable for a reverse mapping, from the mapped subset of WSDL back to IDL.

[CORBABINDING] Object Management Group, CORBA Binding for WSDL, Version 1.0 (https://www.omg.org/spec/CORBABINDING), formal/2010 05 08 – Defines a WSDL extension called a "CORBA type map." The type map specifies CORBA type definitions that are used within CORBA bindings to accurately specify constants, parameter types, return types, and exception types.

[REST] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. dissertation, University of California, Irvine, 2000. See http://roy.gbiv.com/pubs/dissertation/top.htm

[HTTP] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. RFC, IETF, January 1997. See http://www.ietf.org/rfc/rfc2616.txt

[HTML] W3C HTML 4.01 Specification: https://www.w3.org/TR/html4

[URI] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. RFC, IETF, January 2005. See http://www.ietf.org/rfc/rfc3986.txt

[URI-TEMPLATE] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, D. Orchard, RFC 6570: URI Template. RFC, IETF, March 2012. See https://tools.ietf.org/html/rfc6570

[JSON] The JavaScript Object Notation (JSON) Data Interchange Format; https://www.ietf.org/rfc/rfc8259.txt.

[XML] Extensible Markup Language (XML) 1.0 (Fifth Edition) Specification. Available from: https://www.w3.org/TR/2008/REC-xml-20081126

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Camel Case

A naming convention that represents phrases composed of multiple word using a single word where spaces and punctuation are removed, and every word begins with a capital letter. In this specification, the term Camel Case refers to the variation of Camel Case commonly-known as Lower Camel Case, where the first letter is not capitalized. For example, the Camel Case representation of “these are my words” would be “theseAreMyWords”.

Pascal Case

Also known as Upper Camel Case, is a variation of Camel Case where the first letter is capitalized. For example, the Pascal Case representation of the phrase “these are my words” would be “TheseAreMyWords”.

5 Symbols

The acronyms used in this specification are show in Table 5.1.

Table 5.1: Acronyms

Acronym	Meaning
JSONDR	JSON Data Representation format, defined in section 9
XMLDR	XML Data Representation format, defined in section 10

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specification.

6.2 Acknowledgments

The following companies submitted this specification:

- Micro Focus

7 REST for CORBA overview

The goal of the REST for CORBA specification is to define a standard and interoperable mechanism to enable CORBA objects to be exposed as REST [REST] services and to enable pure REST clients to transparently use CORBA services through the exposed REST façade.

In order to achieve the above goals, this submission defines:

- a set of standard IDL annotations (IDL-RS) to decorate IDL constructs in order to expose the corresponding CORBA objects as REST services that can be consumed by REST client applications.
- a set of standard IDL annotations (IDL-RS) to decorate IDL constructs in order to expose the corresponding CORBA objects as REST services that can be consumed by REST client applications.

7.1 IDL-RS annotations overview

The IDL-RS annotations provide a standard mechanism to decorate IDL constructs with IDL annotations to clearly and unambiguously define REST representations of CORBA services. The IDL-RS annotations strive to comply with the Representational State Transfer (REST) architectural style. The design of the IDL-RS annotations assume that HTTP is the underlying network transport layer protocol and aim to provide a clear mapping between HTTP resources and the corresponding CORBA objects.

IDL-RS annotations are described in section 8.

7.2 Data representation formats overview

A Data Representation format defines the way in which objects of the types defined by the IDL Type System may be externalized such that they can be stored in a file or communicated over a network (this is also commonly referred as “data serialization” or “data marshaling”). The JSON and XML Data Representation formats are typically used in REST applications and provide an alternative to the encoding mechanism to the standard Common Data Representation (CDR) format used in CORBA applications.

The Data Representation Formats to externalize objects defined with the IDL Type Representation format to JSON and XML Data Representation formats are described in section 9 and section 10 respectively.

8 IDL-RS annotations

The IDL-RS annotations are a set of standard IDL annotations. They provide a standard mechanism to decorate IDL constructs with IDL annotations to clearly and unambiguously define REST representations of CORBA services.

IDL-RS annotations are designed to comply with the Representational State Transfer [REST] architectural style and assume that HTTP is the underlying network transport layer protocol.

The standard IDL-RS annotations are defined in the IDL module **IDL_RS** as specified below.

8.1 Assigning REST Uniform Resource Identifiers to IDL constructs

CORBA objects are exposed as REST resources [REST] using the **@Path** annotation. The **@Path** annotation assigns a mapping between HTTP resources and the corresponding CORBA objects.

The **@Path** annotation binds the specified URI [URI] or URI templates [URI-TEMPLATE] to the IDL construct that it is is applied to.

8.1.1 @Path annotation

The **@Path** annotation is defined as:

```
module IDL_RS {
   @annotation Path {
      string uri;
      string rir default “”;
   };
};
```

The **uri** parameter specifies the Uniform Resource Identifier to bind to. The accepted values are strings representing a valid URI path or a valid URI path template. The effective URI assigned to the annotated IDL construct is obtained as follows:

- if IDL constructs in the enclosing scopes are not annotated with **@Path** annotations, then the effective URI is defined by the value of the **uri** parameter.
- if IDL constructs in the enclosing scopes are annotated with **@Path** annotations, then the effective URI is defined by the concatenation of the URI assigned to the enclosing scopes with the URI assigned to the **uri** parameter.

The **rir** parameter allows binding the URI specified in the **uri** parameter to the initial reference specified in the **rir** parameter. A typical usage pattern of the **rir** parameter is to bind a persistent CORBA object reference to its URI. The **rir** parameter is optional. When the **rir** parameter is not specified, the default value of empty string is implied and no object reference is explicitly bound to the **uri** parameter.

8.1.1.1 @Path Applicability

The **@Path** annotation is applicable to:

- IDL module
- IDL interface
- IDL operation
- IDL attribute

8.1.1.2 URI path example

```
import IDL_RS;

@Path("/service")
module Mod {

   interface Int {

      typedef sequence<octet> OctetSeq;

      @Path("send-data")
      void op(in OctetSeq bytes);
   };
};
```

The URI to represent the IDL operation: **::Mod::Int::send_bytes()** would be a concatenation of all the **@Path** annotations as we walk the IDL tree from our current enclosing scope all the way to the root scope of IDL tree. Each **@Path** annotations **uri** parameter would be concatenated together, using a forward slash "/" as a separator, if the current URI does not already end in a forward slash. In the example above this would generate the URI: **/service/send-data**.

8.1.2 @PathParam annotation

The **@PathParam** annotation specifies a binding between the matching URI path template parameter and the IDL parameter it is applied to.

The matching URI path template parameter is defined as the URI path template parameter that:

- has the same name as the string specified in the **path_param_id** annotation parameter
- is contained in a URI path template (as specified by the **@Path** annotation) bound to an enclosing IDL naming scope.

The **@PathParam** annotation is defined as:

```
module IDL_RS {
   @annotation PathParam {
      string path_param_id;
   };
};
```

The **path_param_id** parameter specifies the name of the matching URI path template parameter.

The **@PathParam** annotation is applicable to:

- IDL operation parameters of basic types as defined in IDL 4.2, Section 7.4.1.4.4.2, Basic Types.

8.1.2.1 URI path template example

```
import IDL_RS;

interface Account{};

interface Bank {
```

```
        @Path("/bank/account/{id}")
        Account create_account(@PathParam("id") in long account_id);
};
```

In the above example the URI template **{id}** is replaced by the value of IDL parameter **account_id**, to generate a unique URI that corresponds to our CORBA object. The **@PathParam** annotation is used to bind a URI template to an individual IDL parameter. Possible values for our URI may be: **/bank/account/1**, **/bank/account/2**, and so on.

8.1.3 @QueryParam annotation

The **@QueryParam** annotation specifies a binding between a matching URI query parameter contained in the request target URI field of the HTTP request message and the IDL parameter it is applied to.

URI query parameters are contained within a URI query string. A URI query string (as defined in [URI] and [URI-TEMPLATE]) is a string that is appended to a URI path and that begins with the character "?". It is composed of one or more URI query parameters. A URI query parameter is a string representation of a key-value pair. The query parameter string is composed of the key, followed by the "=" character, followed by the value. Multiple URI query parameters are separated by the "&" character.

A matching URI query parameter is defined as the URI query parameter that has the same key as the string specified in the **query_param_id** annotation parameter.

The **@QueryParam** annotation is defined as:

```
module IDL_RS {
   @annotation QueryParam {
      string query_param_id;
   };
};
```

The **query_param_id** annotation parameter specifies the URI query parameter key. The value extracted from the matching URI query parameter is injected into the IDL operation parameter that the **@QueryParam** annotation is applied to.

The **@QueryParam** annotation is applicable to:

- IDL operation parameters of basic types as defined in IDL 4.2, Basic Types, Section 7.4.1.4.4.2.

8.1.3.1 @QueryParam Example

```
import IDL_RS;

interface Account{

   @Path("withdraw")
   void withdraw_funds(@QueryParam("account-id") in long account_id,
                       @QueryParam("amount") in float funds);
};
```

In the above example, the URI **/withdraw** is bound to the IDL operation **::Account::withdraw_funds()** by the @Path annotation. The **@QueryParam** annotations attached to the IDL operation parameters **account_id** and **funds**, of type **long** and **float** respectively, specify that an HTTP request to the **/withdraw?account_id=500&amount=100.00** target URI will result in the extraction of the *(account_id, 500)* and *(amount, 100.00)* key-value pairs and the injection of the value **500** into the **long account_id** parameter and **100.00** into the **float funds** parameter.

8.1.4 Obtaining string representations of IDL object references

The **{objkey}** URI template parameter has special semantics, different from what is defined in @Path annotation, section 8.1.1, when:

- the URI template containing **{objkey}** is used in an **@Path** annotation applied to an IDL interface; and
- this annotated IDL interface is used as an object reference in the IDL; i.e. being used as an IDL operation parameter type or return type.

In this case, the value of the **{objkey}** URI template parameter will be assigned an arbitrary string that uniquely identifies the specific CORBA Object instance that the object reference refers to. Compliant implementations must provide this unique identifier and inject its value into the **{objkey}** URI template parameter.

Compliant implementations may compute the value of the **{objkey}** URI template parameter by retrieving the object key for the specific CORBA Object instance (defined in IDL as a **sequence<octet>** type) and converting it to a string representation by Base64 encoding it.

8.1.4.1 {objkey} URI path template parameter example

```
import IDL_RS;

@Path("account/{objkey}")
interface Account { /* … */};

interface Bank {
   @PUT
   @Path("/bank/account/{id}")
   Account create_account(@PathParam("id") in long account_id);
};
```

In the above excerpt, the use of **{objkey}** in **@Path** annotation applied to the Account interface meets the conditions defined in section 8.1.4. The *Account* interface is used as an object reference in the return parameter of the *create_account()* operation. The URI anchored to an instance of the *Account* object reference return parameter, as specified by the **@Path** annotation to the *Account* interface, will appear as **/account/4YYASBSDFJ3456JSDF==**, where the **4YYASBSDFJ3456JSDF==** string is the unique string representation of a specific CORBA *Object* instance of type *Account*.

8.2 Assigning Request Method Designators to IDL constructs

The **@GET**, **@PUT**, **@POST**, **@DELETE** annotations designate the HTTP request methods that are associated with the IDL constructs they are applied to.

The **@GET**, **@POST**, **@PUT**, **@DELETE** annotations are applicable to:

- IDL operation

The **@GET**, **@POST**, **@PUT** annotations are applicable to:

- IDL attribute

A resource IDL operation is an IDL operation:

- anchored to a URI or URI template path as assigned by an **@Path** annotation in an enclosing IDL naming scope.
- anchored to an HTTP method designator as assigned by one of **@GET**, **@POST**, **@PUT**, **@DELETE** annotations.

A resource IDL attribute is an IDL attribute:

- anchored to a URI or URI template path as assigned by an **@Path** annotation in an enclosing IDL naming scope.
- anchored to an HTTP method designator as assigned by one of **@GET**, **@POST**, **@PUT**, **@DELETE** annotations.

8.2.1 @GET annotation

The **@GET** annotation specifies that the annotated IDL operation or attribute is capable of receiving a HTTP GET requests from a REST client.

The **@GET** annotation specifies that HTTP request messages can be dispatched to the annotated resource IDL operation or annotated resource IDL attribute if:

- the target URI of the HTTP request message matches the URI anchored to the IDL operation or attribute
- the HTTP method specified in the HTTP request method is **GET**

When **@GET** is applied to an IDL attribute, it binds to the attribute accessor function that retrieves the value of the attribute.

The IDL definition for this annotation is:

```
module IDL_RS {
   @annotation GET {};
};
```

8.2.1.1 @GET example

```
import IDL_RS;

@Path("/account/{objkey}")
interface Account {

   @GET
   float get_balance();
};
```

The **@Path** annotation applied to the *Account* interface binds the specific *Account* object instance to URI path template **/account/{objkey}**, where the **{objkey}** URI template parameter will be injected with a unique identifier of the CORBA object, as defined in section 8.1.4.

The **@GET** annotation applied to IDL operation *get_balance()* binds it to HTTP GET requests targeting URI **/account/{objkey}**. HTTP GET requests targeting a URI such as: **/account/4YYASBSDFJ3456JSDF==** will be dispatched to the IDL operation *get_balance()*.

8.2.2 @POST annotation

The **@POST** annotation specifies that the annotated IDL operation or attribute is capable of receiving a HTTP POST requests from a REST client.

The **@POST** annotation specifies that HTTP request messages can be dispatched to the annotated resource IDL operation or annotated resource IDL attribute if:

- the target URI of the HTTP request message matches the URI anchored to the IDL operation or attribute
- the HTTP method specified in the HTTP request method is **POST**

When **@POST** is applied to an IDL attribute, it binds to the attribute accessor function that sets the value of the attribute.

The IDL definition for this annotation is:

```
module IDL_RS {
   @annotation POST {};
};
```

8.2.2.1 @POST example

```
import IDL_RS;

@Path("/account/{objkey}")
interface Account {

   @POST
   void deposit(@QueryParam("amount") in float funds);
};
```

The **@Path** annotation applied to the *Account* interface binds the specific *Account* object instance to URI path template **/account/{objkey}**, where the **{objkey}** URI template parameter will be injected with a unique identifier of the CORBA object, as defined in section 8.1.4.

The **@POST** annotation applied to IDL operation *deposit()* binds it to HTTP POST requests targeting URI **/account/{objkey}.** HTTP POST requests targeting a URI such as: **/account/4YYASBSDFJ3456JSDF==** will be dispatched to the IDL operation *deposit()*.

The **@QueryParam** annotation applied to the *funds* parameter of operation deposit() specifies that the operation binds to a URI containing a query parameter, such as **/account/4YYASBSDFJ3456JSDF==?amount=1000.45** (as defined in section 8.1.3). The *amount* query parameter is parsed and its value is extracted and injected into the *funds* parameter of the *deposit()* operation.

8.2.3 @PUT annotation

The **@PUT** annotation specifies that the annotated IDL operation or attribute is capable of receiving a HTTP PUT requests from a REST client.

The **@PUT** annotation specifies that HTTP request messages can be dispatched to the annotated resource IDL operation or annotated resource IDL attribute if:

- the target URI of the HTTP request message matches the URI anchored to the IDL operation or attribute
- the HTTP method specified in the HTTP request method is **PUT**

When **@PUT** is applied to an IDL attribute, it binds to the attribute accessor function that sets the value of the attribute.

The IDL definition for this annotation is:

```
module IDL_RS {
   @annotation PUT {};
};
```

8.2.3.1 @PUT example

```
import IDL_RS;
```

```
@Path("/account/{objkey}")
interface Account { /* … */ };

interface Bank {
   @Path("/bank/account/{account-id}")
   @PUT
   Account find_account(@PathParam("account-id") in long account_id);
};
```

The **@Path** annotation applied to the *find_account()* operation binds it to the **/bank/account/{account-id}** URI.

The **@PathParam** annotation applied to the *account_id* parameter of operation *find_account()* specifies that the URI path template parameter **{account-id}** will be parsed and its value extracted and injected into the *account_id* IDL parameter of operation *find_account()* (as defined in section 8.1.2).

The **@PUT** annotation applied to IDL operation *create_account()* binds it to HTTP PUT requests targeting URI **/account/{objkey}**. HTTP PUT requests targeting a URI such as: **/bank/account/1337** will be dispatched to the IDL operation *find_account()* and the value *1337* will be injected into parameter *account_id*.

The return result of operation *find_account()* is an *Account* object instance. The **@Path** annotation applied to the *Account* interface binds the specific *Account* object instance to URI path template **/account/{objkey}**, where the **{objkey}** URI template parameter will be injected with a unique identifier of the CORBA object, as defined in section 8.1.4, such as: **/account/4YYASBSDFJ3456JSDF==**

8.2.4 @DELETE annotation

The **@DELETE** annotation specifies that the annotated IDL operation or attribute is capable of receiving a HTTP DELETE requests from a REST client.

The **@DELETE** annotation specifies that HTTP request messages can be dispatched to the annotated resource IDL operation or annotated resource IDL attribute if:

- the target URI of the HTTP request message matches the URI anchored to the IDL operation or attribute
- the HTTP method specified in the HTTP request method is **DELETE**

The IDL definition for this annotation is:

```
module IDL_RS {
   @annotation DELETE {};
};
```

8.2.4.1 @DELETE example

```
import IDL_RS;

@Path("/account/{objkey}")
interface Account {

   @DELETE
   void delete_account();
};
```

The **@Path** annotation applied to the *Account* interface binds the specific *Account* object instance to URI path template **/account/{objkey}**, where the **{objkey}** URI template parameter will be injected with a unique identifier of the CORBA object, as defined in section 8.1.4.

The **@DELETE** annotation applied to IDL operation *delete_account()* binds it to HTTP DELETE requests targeting URI **/account/{objkey}**. HTTP DELETE requests targeting a URI such as: **/account/4YYASBSDFJ3456JSDF==** will be dispatched to the IDL operation *delete_account()*.

8.3 Controlling Data Representation format encoding

The **@Consumes** and **@Produces** annotations specify the Data Representation format that the annotated resource IDL constructs use to produce data representations. This specification defines two standard Data Representation formats:

- JSON Data Representation format, defined in section 9, page 19
- XML Data Representation format, defined in section 10, page 32

The Data Representation formats used are mapped to HTTP request and response media types [HTTP] and are specified in HTTP request and response messages using the standard HTTP **Content-Type** and **Accept** headers.

8.3.1 @Consumes Annotation

The **@Consumes** annotation defines that the annotated IDL construct can consume messages containing data in the Data Representation format specified by **media_type** annotation parameter. The **media_type** annotation parameter shall contain a string representing an HTTP media type or a comma-separated list of HTTP media types.

The **@Consumes** annotation is defined as:

```
module IDL_RS {
   @annotation Consumes {
     string media_type;
   };
};
```

8.3.2 @Produces Annotation

The **@Produces** annotation defines that the annotated IDL construct can produce messages containing data in the Data Representation format specified by **media_type** annotation parameter. The **media_type** annotation parameter shall contain a string representing an HTTP media type or a comma-separated list of HTTP media types.

The **@Produces** annotation is defined as:

```
module IDL_RS {
   @annotation Produces {
      string media_type;
   };
};
```

8.3.3 @Produces and @Consumes Example

Both the above annotations take a string argument that describes the data format as a media type string. For JSON this is: **application/json**, and for XML this is: **application/xml**.

The following IDL excerpt shows an example of **@Consumes** and **@Produces** usage:

```
import IDL_RS;

@Path("/name")
@Consumes("application/xml")
@Produces("application/json")
void greet_me(in string name, out string greeting);
```

The IDL *greet_me()* operation is a resource IDL operation anchored to the */name* URI. The **@Consumes** annotation declares that HTTP messages directed to the */name* target URI and containing data in XML Data Representation format can be dispatched to the *greet_me()* IDL operation. The **@Produces** annotation declares that data returned by the *greet_me()* IDL operation shall be represented in JSON Data Representation format.

IDL types that are applicable to the other IDL-RS annotations. For a list see the applicability section: "@Path Applicability".

8.3.4 @Consumes and @Produces Applicability

The **@Consumes** and **@Produces** annotations are applicable to:

- IDL module
- IDL interface
- IDL operation
- IDL attribute

@Produces and **@Consumes** annotations are inherited from enclosing IDL scopes.

8.4 Assigning HTTP status code representations to IDL exceptions

In CORBA, an IDL operation can raise an exception to indicate that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information, as defined in the IDL exception declaration.

In REST, HTTP responses start with a status line, containing the following information:

- The protocol version, such as *HTTP/1.1*.
- A status code, such as *200* or *404*.
- A status text – a brief, informational, textual description of the status code.

A typical status line might look like: HTTP/1.1 404 Not Found.

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes: informational responses (100–199), successful responses (200–299), redirects (300–399), client errors (400–499), and server errors (500–599).

8.4.1 @HTTPStatus Annotation

The **@HTTPStatus** annotation defines a mapping between a CORBA IDL exception it is applied to and an HTTP status code and status text description. The **code** annotation parameter specifies the HTTP response status code. The optional **description** parameter specifies the HTTP response status text.

The **@HTTPStatus** annotation is defined as:

```
module IDL_RS {
   @annotation HTTPStatus {
      long code;
      string description default "";
   };
};
```

8.4.1.1 @HTTPStatus applicability

The **@HTTPStatus** annotation is applicable to:

- IDL exception

8.4.1.2 @HTTPStatus annotation example

```
import IDL_RS;

@Path("/account/{objkey}")
interface Account {

   @HTTPStatus(code = 409, description = "Insufficient Funds Available")
   exception InsufficentFunds {
      string reason;
   };

   @POST
   @Path("withdraw")
   void withdraw(in float funds) raises InsufficentFunds;
};
```

The **@HTTPStatus** annotation applied to the IDL exception **InsufficientFunds** maps it an HTTP status code of *409* and status text of *Insufficient Funds Available*.

The **@Path** annotation applied to the *Account* interface binds the specific *Account* object instance to URI path template **/account/{objkey}**, where the **{objkey}** URI template parameter will be injected with a unique identifier of the CORBA object, as defined in section 8.1.4.

The **@POST** annotation applied to IDL operation *withdraw()* binds it to HTTP POST requests targeting URI **/account/{objkey}/withdraw.** HTTP POST requests targeting a URI such as: **/account/4YYASBSDFJ3456JSDF==/withdraw** will be dispatched to the IDL operation *withdraw()*. When an IDL exception **InsufficientFunds** is raised, the HTTP response message status line will be set to *HTTP/1.1 409 Insufficient Funds available***.**

8.4.2 Mapping CORBA System Exceptions to HTTP Status Codes (Non-normative)

CORBA Standard System Exception can be raised even though they are not declared in a raises clause in the IDL.

This section outlines a mapping between CORBA Standard System Exceptions and HTTP status codes that a compliant implementation may follow.

Table 8.1: Mapping System Exception to HTTP Status Codes

CORBA System Exception	HTTP Response Code
COMM_FAILURE TIMEOUT	TIMEOUT (408)
OBJECT_NOT_EXIST INV_OBJREF	GONE (410)
TRANSIENT	NOT_FOUND (404)

NO_PERMISSION	FORBIDDEN (403)
BAD_OPERATION BAD_PARAM	METHOD_NOT_ALLOWED (405)
MARSHAL	BAD_REQUEST (400)
INTERNAL INITIALIZE	INTERNAL_SERVER_ERROR (500)
NO_IMPLEMENT	NOT_IMPLEMENTED (501)
IMP_LIMIT NO_MEMORY NO_RESOURCES	SERVICE_UNAVAILABLE (503)
All other System Exceptions	CONFLICT (409)

9 JSON Data Representation format

This section specifies a mapping between IDL types and JSON [JSON] Data Representation (JSONDR) syntax. For the IDL types defined below, a mapping is defined to convert the data from the CORBA Common Data Representation format to JSON Data Representation format and vice-versa (i.e. from CDR to JSONDR and JSONDR to CDR).

9.1 IDL Core Data Types

9.1.1 Basic types

9.1.1.1 Integer types

The value of an instance of an IDL integer data types: **short**, **unsigned short**, **long**, **unsigned long**, **long long**, and **unsigned long long** are represented as a JSON integer number type, as defined in IDL 4.2, section 7.2.6.1, Integer Literals.

For example, an instance of an IDL long type containing the value 123:

long return_code;

shall be represented in JSON as:

123

9.1.1.2 Floating point types

The value of an instance of an IDL floating-point types: **float**, **double**, and **long double** are represented as a JSON number type, as defined in IDL 4.2, section 7.2.6.2, Floating-point Literals.

For example, and instance of an IDL float type containing the value -1.1225E8:

float float_val;

shall be represented in JSON as:

-1.1225E8

9.1.1.3 Character types

The value of an instance of an IDL character data types: **char** and **wchar** are represented as JSON string type containing a single character, and the character shall be encoded as defined in IDL 4.2, section 7.2.6.2 Character Literals.

For example an instance of an IDL char type containing the value ‘x’:

char char_val;

shall be represented in JSON as:

“x”

9.1.1.4 Boolean type

The value of an instance of the IDL boolean data type shall be represented as either JSON **true** value or JSON **false** value.

For example, an instance of the following IDL boolean containing the value FALSE:

boolean bool_val;

shall be represented in XML as:

false

9.1.1.5 Octet type

The value of an instance of the IDL octet data type shall be represented as a JSON integer number value, and following the bounds rules for an IDL octet data type as defined in IDL 4.2, Section 7.4.1.4.3, Constants.

For example, an instance of the following IDL octet containing the value 254:

octet octet_val;

shall be represented in JSON as:

254

9.1.2 Template types

9.1.2.1 Sequence types

The value of an instance of the IDL sequence data type shall be represented as a JSON array that, for each element in the IDL sequence, contains a JSON array element set to the JSON representation of the value of the IDL sequence element, as specified in section 9.

For example, an instance of the following IDL sequence containing octet value 2, 3, 5:

typedef sequence<octet> octetSeq;

shall be represented in JSON as:

[2, 3, 5]

9.1.2.2 String types

The value of an instance of the IDL **string** and **wstring** data types shall be represented as a JSON string set to the value of the IDL string or wstring instance, as defined in IDL 4.2, section 7.2.6.3 String Literals.

For example, an instance of the following IDL string containing “my example string”:

typedef string my_string;

shall be represented in JSON as:

my example string

9.1.2.3 Fixed point types

The value of an instance of the IDL **fixed** data type shall be represented as a JSON fraction number set to the representation of the value as defined in IDL 4.2, section 7.2.6.5 Fixed-Point Literals (the “d” or “D” character may be omitted).

For example, an instance of the following IDL fixed type containing the value 123.45d with digits 5 and scale 2:

typedef fixed<5, 2> my_fixed;

shall be represented in JSON as:

123.45

9.1.3 Constructed types

9.1.3.1 Struct types

The value of an IDL struct data type instance shall be represented as a JSON object. Each IDL struct member shall be represented as a JSON object member, where the JSON object member name is set to the IDL struct member identifier, and the JSON object member value is set to the representation of the IDL struct member value as defined for its type in section 9.

For example, an instance of the following IDL struct type, with string_val member set to “Joe Bloggs”, char_val member set to “c”, octet_val set to 200, short_val set to 10000, long_val set to -2323424, ulonglong_val set to 3424234243:

```
struct StructType {
   string string_val;
   char char_val;
   octet octet_val;
   short short_val;
   long long_val;
   unsigned long long ulonglong_val;
};
```

shall be represented in JSON as:

```
{
   “string_val”: “Joe Bloggs”,
   “char_val”: “c”,
   “octet_val”: 200,
   “short_val”: 10000,
```

```
   “long_val”: -2323424,
   “ulonglong_val”: 3424234243
}
```

9.1.3.2 Enum types

The value of an instance of the IDL **enum** data type shall be represented as a JSON string value set to the enums current enumerator identifier value.

For example, an instance of the following IDL enum with value set to RED:

```
enum Color {RED, GREEN, BLUE};
```

shall be represented in JSON as:

“RED”

9.1.3.3 Union types

The value of an instance of an IDL **union** data type shall be represented as a JSON object containing:

- an object member, where the object member name is set to **discriminator** and the JSON object member value is set to either:
  - the JSON representation, as defined in section 9, of the value of the case label of the IDL union discriminator type, if the value of the case label matches one of the cases of the union,
  - **_default**, if the default case of the union is selected.
- an object member, where the object member name is set to **value** and the JSON object member value is set to the JSON representation, as defined in section 9, of the value of the IDL union case type selected by the discriminator.

For example, an instance of the following IDL **union** with discriminator set to LEFT and value set to 10.5:

```
enum Direction {UP, DOWN, LEFT, RIGHT, NONE, UNKNOWN};

union Movement switch(Direction) {
   case UP:
   case DOWN:
   case LEFT:
   case RIGHT:
      float distance;
   case NONE:
      long time_still;
   default:
      short error_code;
};
```

shall be represented in JSON as:

```
{
   “discriminator”: “LEFT”,
   “value”: 10.5
}
```

An instance of the previous Movement IDL union with discriminator set to UNKNOWN and value set to 255 shall be represented in JSON as:

```
{
   “discriminator”: “_default”,
   “value”: 255
}
```

9.1.3.4 Array types

The value of an instance of an IDL array data type shall be represented in JSON according to the same rules as specified for IDL sequence types, as defined in section 9.1.2.1.

9.1.3.5 Native types

A native type may be used only to define IDL operation parameters and results. Native type parameters are permitted only in operations of *local interfaces* or *valuetypes*. Native types cannot be used as operation parameters or results in remote invocations. As such, no JSON data representation mapping of native types is specified.

9.2 IDL Any type

An IDL **any** type instance shall be represented as a JSON object containing:

- an object member, where the object member name is set to **typecode** and the JSON object member value is set to the JSON representation of the **TypeCode**, as defined in section 9.2.1
- an object member, where the object member name is set to **typecode** and the JSON object member value is set to the JSON representation of the **TypeCode**, as defined in section 9

9.2.1 TypeCode

In CORBA Common Data Representation (CDR) transfer syntax (the format in which the GIOP represents OMG IDL data types in an octet stream), **TypeCodes** are encoded as the **TCKind** enum value, followed by zero or more **TypeCode** parameter values. The encodings of the parameter lists fall into three general categories:

- TypeCodes with an empty parameter list
- TypeCodes with simple parameter lists
- TypeCodes with complex parameter lists

Table 9.2 in section 9.3.5.1.2, TypeCode Parameter Notation, of the CORBA Specification, 3.3, part 2, defines the type (i.e. empty, simple, complex parameter list) for each TypeCode TCKind enum value in the **Type** column and defines the ordered list of TypeCode parameters in the **Parameters** column.

The JSON representation of an instance of a **TypeCode** is correspondingly dependent on the TypeCode parameter list encoding rules defined by the CORBA specification, as defined in the following subsections.

9.2.1.1 Empty parameter list TypeCode

An empty **TypeCode** instance shall be represented as a JSON object containing an object member, where the object member name is set to **kind** and the JSON object member value is set to the JSON representation of the **TypeCode**’s **TCKind** enum value as defined in section 9.1.3.2

9.2.1.1.1 Empty parameter list TypeCode example

A TypeCode with a **TCKind** of **tk_long** describing an IDL **long** type such as:

typedef long my_long;

shall be represented in JSON as:

```
{
   “kind”: tk_long”
}
```

9.2.1.2 Simple parameter list TypeCode

A simple parameter list **TypeCode** is a TypeCode with a **TCKind** enum value set to either **tk_string**, **tk_wstring**, or **tk_fixed**.

9.2.1.2.1 TypeCodes with TCKind set to tk_string or tk_wstring

A simple parameter list **TypeCode** instance with **TCKind** equal to either **tk_string** or **tk_wstring** shall be represented as a JSON object containing:

- an object member, where the object member name is set to **kind** and the JSON object member value is set to the JSON representation of the **TypeCode**’s **TCKind** enum value as defined in section 9.1.3.2
- an object member, where the object member name is set to **bound** and the JSON object member value is set to the JSON representation of the IDL unsigned long parameter defining the bound of the IDL string or wstring, as defined in section 9.1.1.1.

Simple parameter list TypeCode with TCKind set to tk_string or tk_wstring examples

A TypeCode with a **TCKind** of **tk_string** describing a bounded IDL string type such as:

typedef string<80> bounded_string;

shall be represented in JSON as:

```
{
   “kind”: “tk_string”,
   “bound”: 80
}
```

9.2.1.2.2 TypeCodes with TCKind set to tk_fixed

A simple parameter list **TypeCode** instance with **TCKind** equal to **tk_fixed** shall be represented as a JSON object containing:

- an object member, where the object member name is set to **kind** and the JSON object member value is set to the JSON representation of the **TypeCode**’s **TCKind** enum value as defined in section 9.1.1.1.
- an object member, where the object member name is set to **digits** and the JSON object member value is set to the JSON representation of the IDL unsigned short parameter defining the digits of the IDL fixed type, as defined in section 9.1.1.1.
- an object member, where the object member name is set to **scale** and the JSON object member value is set to the JSON representation of the IDL short parameter defining the scale of the IDL fixed type, as defined in section 9.1.1.1.

Simple parameter list TypeCode with TCKind set to tk_fixed examples

A TypeCode with **TCKind** of **tk_fixed** describing an IDL **fixed** type such as:

typedef fixed<5, 2> my_fixed;

shall be represented in JSON as:

```
{
   “kind”: “tk_fixed”,
   “digits”: 5,
   “scale”: 2
}
```

9.2.1.3 Complex parameter list TypeCode

9.2.1.3.1 TypeCodes with TCKind tk_sequence or tk_array

An complex parameter list **TypeCode** instance with **TCKind** equal to **tk_sequence** or **tk_array** shall be represented in JSON as an object containing:

- an object member, where the object member name is set to **kind** and the JSON object member value is set to the JSON representation of the **TypeCode**’s **TCKind** enum value as defined in section 9.1.3.2
- an object member, where the object member name is set to **element_typecode** and the JSON object member value is set to the JSON representation of the **TypeCode** describing the type of the elements contained within the IDL sequence or array, as defined in section 9.2.1.
- an object member, where the object member name is set to **length** and the JSON object member value is set to the JSON representation, as defined in section 9.1.1.1, of the IDL unsigned long parameter defining the length the IDL array or the maximum length of the IDL sequence. The value of the **length** object member element shall be set to 0 for TypeCodes representing an unbounded IDL sequence.

Complex parameter list TypeCode with TCKind tk_sequence or tk_array examples

A TypeCode with a **TCKind** of **tk_sequence** describing an unbounded IDL **sequence** type such as:

typedef sequence<long> LongSeq;

shall be represented in JSON as:

```
{
   “kind”: “tk_sequence”,
   “element_typecode”: {
      “kind”: “tk_long”
    },
   “bound”: 0
}
```

9.2.1.3.2 TypeCodes with TCKind other than tk_sequence or tk_array

A complex parameter list **TypeCode** instance with **TCKind** other than **tk_sequence** or **tk_array** shall be represented as a JSON object containing:

- an object member, where the object member name is set to **kind** and the JSON object member value is set to the JSON representation of the **TypeCode**'s **TCKind** enum value as defined in section 9.1.3.2
- an object member, where the object member name is set to **id** and the JSON object member value is set to the JSON representation of the **CORBA::RepositoryId** value returned by the TypeCode::id() operation, for TypeCode instances that allow the id() operation to be invoked, as defined in the CORBA specification 3.3, part 1, section 8.11.1.
- an object member, where the object member name is set to **name** and the JSON object member value is set to the JSON representation of the **CORBA::Identifier** value returned by the TypeCode::name() operation, for TypeCode instances that allow the name() operation to be invoked, as defined in the CORBA specification 3.3, part 1, section 8.11.1.
    -

Complex parameter list TypeCode with TCKind other than tk_sequence or tk_array examples

A TypeCode with a **TCKind** of **tk_struct** describing an IDL struct type such as the following IDL struct named **Example**:

```
struct Example {
   short member1;
   short member2;
   long member3;
};
```

shall be represented in JSON as:

```
{
   "kind": "tk_struct",
   "id": "IDL:Example:1.0",
   "name": "Example"
}
```

9.2.2 Any examples

9.2.2.1 Empty parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL **long** type with a value of 10 inserted into it:

```
typedef long my_long;
typedef any my_any;
```

shall be represented in JSON as:

```
{
   "typecode": {
      "kind": "tk_long"
   },
   "value": 10
```

}

9.2.2.2 Simple parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL bounded **string** type with a *bound* of 80 and a value of “example string” inserted into it:

typedef string<80> bounded_string;
typedef any my_any;

shall be represented in JSON as:

```
{
   “typecode”: {
      “kind”: “tk_string”,
      “bound”: 80
   },
   “value”: “example_string”
}
```

An instance of an IDL **any**, which has had an instance of an IDL **fixed** type, with parameters *digits* equal to 5 and *scale* equal to 2 and a value of 123.45, inserted into it:

typedef fixed<5, 2> my_fixed;
typedef any my_any;

shall be represented in JSON as:

```
{
   “typecode”: {
      “kind”: “tk_fixed”,
      “digits”: 5,
      “scale”: 2
   },
   “value”: 123.45
}
```

9.2.2.3 Complex parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL unbounded **sequence** type inserted into it:

typedef sequence<long> LongSeq;
typedef my_any;

shall be represented in JSON as:

```
{
   “typecode”: {
      “kind”: “tk_sequence”,
      “element_typecode”: {
         “kind”: “tk_long”
```

```
        },
        “bound”: 0
    },
    “value”: [1, 1, 2, 3, 5, 8]
}
```

An instance of an IDL **any**, which has had an instance of an IDL struct type such as the following IDL struct named **Example** inserted into it:

```
struct Example {
    short member1;
    short member2;
    long member3;
};
```

shall be represented in JSON as:

```
{
    “typecode”: {
        “kind”: “tk_struct”,
        “id”: “IDL:Example:1.0”,
        “name”: “Example”
    },
    “value”: {
        “member1”: 100,
        “member2”: 50,
        “member3”: 10000
    }
}
```

9.3 IDL Interfaces

In CORBA, an interface is a description of a set of possible operations that a client may request [CORBA 3.3, Part 1, Section 5.2.2, Requests] of an object [CORBA 3.3, Part 1, Section 5.2.1, Objects], through that interface [CORBA 3.3, Part 1, Section 5.2.5, Interface].

The following sections define the mapping between:

- an HTTP request targeting a URI resource and containing data representation of that resource in JSON format, and a CORBA request targeting the object, interface, operation, or attribute anchored to that resource.
- an HTTP response produced by a URI resource and containing data representation of that resource in JSON format, and a CORBA reply originating from the object, interface, operation, or attribute anchored to that resource.

9.3.1 CORBA Request and HTTP Request message body mapping

The HTTP request message body shall contain a JSON object, denoted as the *request wrapper object.*

The *request wrapper object* shall contain a JSON object member to represent each of the IDL operation parameters that have directional parameter attributes of type *in* and *inout* as follows:

- the IDL operation parameter name shall map to the JSON object member string, followed a “:” character

- the IDL operation parameter value, represented in JSON according to the rules defined in JSON Data Representation format - section 9, shall map to the JSON object member element

Compliant implementations may optionally include the representation of *in* and *inout* parameters that have been annotated with an **@PathParam** or **@QueryParam** annotation in the *request wrapper object*.

The HTTP request header *Content-Type* shall be set to the value *"application/json"*.

9.3.1.1 HTTP Request message body example

```
@Path("/sample/{objkey}")
Interface SampleInterface { /* … */}

struct SampleStruct {
   string struct_member_string;
   long struct_member_long;
};

@Path("/sample_service")
interface SampleServiceInterface {

   @POST
   @Path("sample_operation")
   SampleInterface sample_operation(in long a_in_param,
                                    inout SampleStruct an_inout_param ,
                                    out string an_out_param);
};
```

The *sample_operation()* operation *in* parameter **a_in_param** and the *inout* parameter **an_inout_param** will be marshalled in the *request wrapper object* in the HTTP request message body.

The excerpt below shows the corresponding HTTP request message:

```
POST /sample_service/sample_operation HTTP/1.1
Content-Type: application/json
/* … HTTP Header not shown … */
{
   "a_in_param" : 1234,
   "an_inout_param": {
                        "struct_member_string": "a struct sample value",
                        "struct_member_long": 54321
                      },
}
```

9.3.2 CORBA Reply and HTTP Response message body mapping

The HTTP response message body shall contain a JSON object, denoted as the *response wrapper object.*

The *response wrapper object* shall contain a JSON object member to represent each of the IDL operation parameters that have directional parameter attributes of type *out* and *inout* as follows:

- the IDL operation parameter name shall map to the JSON object member name, followed by a ":" character
- the IDL operation parameter value, represented in JSON according to the rules defined in JSON Data Representation format - section 9, shall map to the JSON object member element

The *response wrapper object* shall contain a JSON object member to represent the IDL operation *return result* as follows:

- the JSON object member string "_ret", followed by a ":" character
- the IDL return result value, represented in JSON according to the rules defined in JSON Data Representation format - section 9, shall map to the JSON object member element

The HTTP response header *Content-Type* shall be set to the value *"application/json"*.

9.3.2.1 HTTP Response message body example

```
@Path("/sample/{objkey}")
Interface SampleInterface { /* … */}

struct SampleStruct {
   string struct_member_string;
   long struct_member_long;
};

@Path("/sample_service")
interface SampleServiceInterface {

   @POST
   @Path("sample_operation")
   SampleInterface sample_operation(in long a_in_param,
                                    inout SampleStruct an_inout_param ,
                                    out string an_out_param);
};
```

The *sample_operation()* operation *out* parameter **an_out_param**, the *inout* parameter **an_inout_param**, and the return result will be marshalled in the *response wrapper object* in the HTTP request message body.

The **@Path** annotation applied to the *SampleInterface* interface, containing the special **{objkey}** URI template parameter, meets the conditions defined in section 8.1.4. The *::SampleServiceInterface::sample_operation()* operation returns an object reference to an instance of *SampleInterface* object. This results in the JSON representation of the object reference being contained in the response "_ret" JSON object member.

The excerpt below shows the corresponding HTTP response message:

```
HTTP/1.1 200 OK
Content-Type: application/json
/* … HTTP Header not shown … */
{
   "_ret" : "/account/4YYASBSDFJ3456JSDF==",
   "an_inout_param": {
                        "struct_member_string": "a struct sample value",
                        "struct_member_long": 54321
                      },
   "an_out_param": "a sample out param string value"
}
```

9.3.3 CORBA Reply containing an exception and HTTP Response message body mapping

For CORBA replies containing a CORBA exception, the HTTP response message body shall contain a JSON object, denoted as the *exception wrapper object.*

The *exception wrapper object* shall contain a JSON object member to represent the IDL string containing the RepositoryId for the exception, as defined in the CORBA (Part 1) specification, section 14, *The Interface Repository*, as follows:

- the JSON object member string “exceptionRepositoryID”, followed by a “:” character
- the IDL string containing the RepositoryId for the exception value, represented in JSON according to the rules defined in String types, section 9.1.2.2, shall map to the JSON object member element

The *response wrapper object* shall contain a JSON object member to represent the IDL exception members as follows:

- the JSON object member string “exceptionMembers”, followed by a “:” character
- a JSON object containing the JSON representation of the members of the exception, if any. Members of an IDL exception shall be represented in JSON in the same manner as members of an IDL struct, as defined in Struct types, section 9.1.3.1.

The HTTP response header *Content-Type* shall be set to the value *“application/json”*.

9.3.3.1 HTTP Response message body exception example

```
@Path(“/sample_service”)
interface SampleServiceInterface {

   exception SampleException {
      long sample_exception_id;
      string sample_exception_string;
   };

   @GET
   @Path(“sample_operation”)
   void sample_operation (in long a_in_param) raises SampleException;
};
```

When the IDL operation *sample_operation()* raises the IDL exception *SampleException*, the exception will be marshalled in the *exception wrapper object* in the HTTP response message body.

The excerpt below shows the corresponding HTTP response message:

```
HTTP/1.1 200 OK
Content-Type: application/json
/* … HTTP Header not shown … */
{
   “exceptionRepositoryID”: “IDL:SampleServiceInterface/SampleException:1.0”,
   “exceptionMembers”: {
                          “sample_exception_id”: 10202,
                          “sample_exception_string”: “a sample exception string value”
                        }
}
```

10 XML Data Representation format

This section specifies a mapping between IDL types and XML [XML] Data Representation (XMLDR) syntax. For the IDL types defined below, a mapping is defined to convert the data from the CORBA Common Data Representation format to XML Data Representation format and vice-versa (i.e. from CDR to XMLDR and XMLDR to CDR).

10.1 IDL Core Data Types

Instances of IDL Core Data Types are represented as an XML element:

- with the XML element name set to the IDL type identifier
- with the XML element content set to a representation of the value of the instance of the IDL type, according to the rules set out in the following subsections.

10.1.1 Basic Types

10.1.1.1 Integer types

The value of an instance of IDL integer data types: **short**, **unsigned short**, **long**, **unsigned long**, **long long**, and **unsigned long long** shall be represented as an XML element content set to the decimal string representation of the integer, as defined in IDL 4.2, Section 7.2.6.1, Integer Literals.

For example, an instance of an IDL long type defined in IDL:

long return_code;

shall be represented in XML as:

<return_code>50000</return_code>

10.1.1.2 Floating point types

The value of an instance of IDL floating-point data types: **float**, **double**, and **long double** shall be represented as an XML element content set to the floating point representation of the value, as defined in IDL 4.2, Section 7.2.6.4, Floating-point Literals.

For example, an instance of an IDL float type defined in IDL:

float float_val;

shall be represented in XML as:

<float_val>-1.1225E8</float_val>

10.1.1.3 Character types

The value of an instance of IDL character data types: **char**, and **wchar** shall be represented as an XML element content set to the character representation of the value, as defined in IDL 4.2, Section 7.2.6.2, Character Literals.

For example, an instance of an IDL char type defined in IDL:

char char_val;

shall be represented in XML as:

<char_val>x</char_val>

10.1.1.4 Boolean types

The value of an instance of the IDL boolean data type shall be represented as an XML element content set to either the case-insensitive string **true** or **false**.

For example, an instance of the following IDL boolean defined in IDL:

boolean bool_val;

shall be represented in XML as:

<bool_val>false</bool_val>

10.1.1.5 Octet type

The value of an instance of the IDL octet data type shall be represented as an XML element content set to the octet representation of the value, as defined in IDL 4.2, Section 7.4.1.4.3, Constants.

For example, an instance of the following IDL octet defined in IDL:

octet octet_val;

shall be represented in XML as:

<octet_val>254</octet_val>

10.1.2 Template Types

10.1.2.1 Sequence Types

The value of an instance of the IDL **sequence** data type shall be represented as an XML element content that, for each element in the IDL sequence, contains a child XML element named **item**, containing the XML representation of the value of the IDL sequence element, as specified in section 10.

For example, an instance of the following IDL sequence containing octet value 2, 3, 5:

typedef sequence<octet> octetSeq;

shall be represented in XML as:

```
<octetSeq>
   <item>2</item>
   <item>3</item>
```

```
   <item>5</item>
</octetSeq>
```

10.1.2.2 String Types

The value of an instance of the IDL **string** and **wstring** data types shall be represented as an XML element content set to the representation of the value of the IDL string or wstring type, as defined in IDL 4.2, section 7.2.6.3 String Literals.

For example, an instance of the following IDL string containing “my example string”:

typedef string my_string;

shall be represented in XML as:

```
<my_string>my example string</my_string>
```

10.1.2.3 Fixed Types

The value of an instance of the IDL **fixed** data type shall be represented as an XML element content set to the representation of the value, as defined in IDL 4.2, section 7.2.6.5 Fixed-Point Literals (the “d” or “D” character may be omitted).

For example, an instance of the following IDL fixed type containing the value 123.45d with digits 5 and scale 2:

typedef fixed<5, 2> my_fixed;

shall be represented in XML as:

```
<my_fixed>123.45</my_fixed>
```

10.1.3 Constructed Types

10.1.3.1 Struct Types

The value of an instance of the IDL **struct** data type shall be represented as an XML element content that, for each struct member of the IDL struct, contains a child XML element with the name set to the struct member identifier, containing the XML representation of the value of the IDL struct member value, as specified in section 10.

For example, an instance of the following IDL struct type, with string_val member set to “Joe Bloggs”, char_val member set to “c”, octet_val set to 200, short_val set to 10000, long_val set to -2323424, ulonglong_val set to 3424234243:

```
struct StructType {
   string string_val;
   char char_val;
   octet octet_val;
   short short_val;
   long long_val;
   unsigned long long ulonglong_val;
};
```

shall be represented in XML as:

```
<StructType>
   <string_val>Joe Bloggs</string_val>
   <char_val>c</char_val>
   <octet_val>200</octet_val>
   <short_val>10000</short_val>
   <long_val>-2323424</long_val>
   <ulonglong_val>3424234243</ulonglong_val>
</StructType>
```

10.1.3.2 Enum Types

The value of an instance of the IDL **enum** data type shall be represented as an XML element content set to the string representation of one of the valid values of the IDL enum.

For example, an instance of the following IDL enum with value set to RED:

enum Color {RED, GREEN, BLUE};

shall be represented in XML as:

```
<Color>RED</Color>
```

10.1.3.3 Union Types

The value of an instance of the IDL **union** data type shall be represented as an XML element content that contains:

- a child XML element named **discriminator**, with the XML element content set to either:
  - the XML representation, as defined in section 10, of the value of the case label of the IDL union discriminator type, if the value of the case label matches one of the cases of the union.
  - **_default**, if the default case of the union is selected.
- a child XML element named **value**, containing the XML representation, as defined in section 10, of the value of the IDL union case type selected by the discriminator.

For example, an instance of the following IDL **union** with discriminator set to LEFT and value set to 10.5:

```
enum Direction {UP, DOWN, LEFT, RIGHT, NONE, UNKNOWN};

union Movement switch(Direction) {
   case UP:
   case DOWN:
   case LEFT:
   case RIGHT:
      float distance;
   case NONE:
      long time_still;
   default:
      short error_code;
};
```

shall be represented in XML as:

```
<Movement>
   <discriminator>
      <Direction>LEFT</Direction>
   </discriminator>
   <value>10.5</value>
</Movement>
```

An instance of the previously defined Movement IDL **union** with discriminator set to UNKNOWN and value set to 255 shall be represented in XML as:

```
<Movement>
   <discriminator>_default</discriminator>
   <value>255</value>
</Movement>
```

10.1.3.4 Array Types

The value of an instance of an IDL array data type shall be represented in XML according to the same rules as specified for IDL sequence types, as defined in section 10.1.2.1.

10.1.3.5 Native Types

A native type may be used only to define IDL operation parameters and results. Native type parameters are permitted only in operations of *local interfaces* or *valuetypes*. Native types cannot be used as operation parameters or results in remote invocations. As such, no XML data representation mapping of native types is specified.

10.2 IDL Any type

An IDL **any** type instance shall be represented as an XML element with the same name as the **any** type, containing:

- a child XML element named **typecode**, containing the XML representation of the **TypeCode**, as defined in section 10.2.1
- a child XML element named **typecode**, containing the XML representation of the **TypeCode**, as defined in section 10.2.1

10.2.1 TypeCode

In CORBA Common Data Representation (CDR) transfer syntax (the format in which the GIOP represents OMG IDL data types in an octet stream), **TypeCodes** are encoded as the **TCKind** enum value, followed by zero or more **TypeCode** parameter values. The encodings of the parameter lists fall into three general categories:

- TypeCodes with an empty parameter list
- TypeCodes with simple parameter lists
- TypeCodes with complex parameter lists

Table 9.2 in section 9.3.5.1.2, TypeCode Parameter Notation, of the CORBA Specification, 3.3, part 2, defines the type (i.e. empty, simple, complex parameter list) for each TypeCode TCKind enum value in the **Type** column and defines the ordered list of TypeCode parameters in the **Parameters** column.

The JSON representation of an instance of a **TypeCode** is correspondingly dependent on the TypeCode parameter list encoding rules defined by the CORBA specification, as defined in the following subsections.

10.2.1.1 Empty parameter list TypeCode

An empty parameter list **TypeCode** instance shall be represented as an XML element named **typecode**, containing a child XML element named **kind**, containing the XML representation of the **TypeCode**'s **TCKind** enum value as defined in section 10.1.3.2.

10.2.1.1.1 Empty parameter TypeCode example

A TypeCode with a **TCKind** of **tk_long** describing a IDL long type such as:

typedef long my_long;

shall be represented in XML as:

```
<typecode>
   <kind>
      <TCKind>tk_long</TCKind>
   </kind>
</typecode>
```

10.2.1.2 Simple parameter list TypeCode

A simple parameter list **TypeCode** is a typecode with a **TCKind** enum value set to either **tk_string**, **tk_wstring**, or **tk_fixed**.

10.2.1.2.1 TypeCodes with TCKind tk_string or tk_wstring

A simple parameter list **TypeCode** instance with **TCKind** equal to either **tk_string** or **tk_wstring** shall be represented as an XML element named **typecode** containing:

- a child XML element named **kind**, containing the XML representation of the **TypeCode**'s **TCKind** enum value as defined in section 10.1.3.2.
- a child XML element named **bound** containing the XML representation of the IDL unsigned long parameter defining the bound of the IDL string or wstring type, as defined in section 10.1.1.1. The content of the **bound** XML element shall be set to 0 for unbounded IDL string.

Simple parameter list TypeCode with TCKind set to tk_string or tk_wstring examples

A TypeCode with a **TCKind** of **tk_string** describing a bounded IDL string type such as:

typedef string<80> bounded_string;

shall be represented in XML as:

```
<typecode>
   <kind>
      <TCKind>tk_string</TCKind>
   </kind>
   <bound>80</bound>
</typecode>
```

10.2.1.2.2 TypeCodes with TCKind set to tk_fixed

A simple parameter list **TypeCode** instance with **TCKind** equal to **tk_fixed** shall be represented as an XML element named **typecode** containing:

- a child XML element named **kind**, containing the XML representation of the **TypeCode**'s **TCKind** enum value as defined in section 10.1.3.2.
- a child XML element named **digits** containing the XML representation of the IDL unsigned short parameter defining the digits of the IDL fixed type, as defined in section 10.1.1.1.
- a child XML element named **scale** containing the XML representation of the IDL short parameter defining the scale of the IDL fixed type, as defined in section 10.1.1.1.

Simple parameter list TypeCode with TCKind tk_fixed examples
A TypeCode with a **TCKind** of **tk_fixed** describing an IDL **fixed** type such as:

typedef fixed<5, 2> my_fixed;

shall be represented in XML as:

```
<typecode>
   <kind>
      <TCKind>tk_fixed</TCKind>
   </kind>
   <digits>5</digits>
   <scale>2</scale>
</typecode>
```

10.2.1.3 Complex parameter list TypeCode

10.2.1.3.1 TypeCodes with TCKind tk_sequence or tk_array

A complex parameter list **TypeCode** instance with **TCKind** equal to **tk_sequence** or **tk_array** shall be represented as an XML element named **typecode**, containing:

- a child XML element named **kind**, containing the XML representation of the **TypeCode**'s **TCKind** enum value as defined in section 10.1.3.2.
- a child XML element named **element_typecode**, containing the XML representation, as defined in section 10.2.1, of the **TypeCode** describing the type of the elements contained within the IDL sequence or array.
- a child XML element named **length**, containing the XML representation, as defined in section 10.1.1.1, of the IDL unsigned long parameter defining the length the IDL array or the maximum length of the IDL sequence. The content of the **length** XML element shall be set to 0 for TypeCodes representing an unbounded IDL sequence.

Complex parameter list TypeCode with TCKind tk_sequence or tk_array examples

A TypeCode with a **TCKind** of **tk_sequence** describing an unbounded IDL **sequence** type such as:

typedef sequence<long> LongSeq;

shall be represented in XML as:

```
<typecode>
   <kind>
```

```
      <TCKind>tk_sequence</TCKind>
   </kind>
   <element_typecode>
      <TCKind>tk_long</TCKind>
   </element_typecode>
   <bound>0</bound>
</typecode>
```

10.2.1.3.2 TypeCodes with TCKind other than tk_sequence or tk_array

All other complex parameter list **TypeCode** instances shall be represented as an XML element named **typecode**, containing:

- a child XML element named **kind**, containing the XML representation of the **TypeCode**'s **TCKind** enum value as defined in section 10.1.3.2.
- a child XML element named **id**, containing the XML representation of the **CORBA::RepositoryId** value returned by the TypeCode::id() operation, for TypeCode instances that allow the id() operation to be invoked, as defined in the CORBA specification 3.3, part 1, section 8.11.1.
- a child XML element named **name**, containing the XML representation of the **CORBA::Identifier** value returned by the TypeCode::name() operation, for TypeCode instances that allow the name() operation to be invoked, as defined in the CORBA specification 3.3, part 1, section 8.11.1.

Complex parameter list TypeCode with TCKind other than tk_sequence or tk_array examples

A TypeCode with a **TCKind** of **tk_struct** describing an IDL **struct** type such as the following IDL struct named **Example:**

```
struct Example {
   short member1;
   short member2;
   long member3;
};
```

shall be represented in XML as:

```
<typecode>
   <kind>
      <TCKind>tk_struct</TCKind>
   </kind>
   <id>IDL:Example:1.0</id>
   <name>Example</name>
</typecode>
```

10.2.2 Any examples

10.2.2.1 Empty parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL **long** type with a value of 10 inserted into it:

```
typedef long my_long;
typedef any my_any;
```

shall be represented in XML as:

```
<my_any>
  <typecode>
    <kind>
       <TCKind>tk_long</TCKind>
    </kind>
  </typecode>
  <value>10</value>
</my_any>
```

10.2.2.2 Simple parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL bounded **string** type with a *bound* of 80 and a value of "example string" inserted into it:

```
typedef string<80> bounded_string;
typedef any my_any;
```

shall be represented in XML as:

```
<my_any>
   <typecode>
      <kind>
         <TCKind>tk_string</TCKind>
      </kind>
      <bound>80</bound>
   </typecode>
   <value>example string</value>
</my_any>
```

An instance of an IDL **any**, which has had an instance of an IDL **fixed** type, with parameters *digits* equal to 5 and *scale* equal to 2 and a value of 123.45, inserted into it:

```
typedef fixed<5, 2> my_fixed;
typedef any my_any;
```

shall be represented in XML as:

```
<my_any>
   <typecode>
      <kind>
         <TCKind>tk_fixed</TCKind>
      </kind>
      <digits>5</digits>
      <scale>2</scale>
   </typecode>
   <value>123.45</value>
</my_any>
```

10.2.2.3 Complex parameter list Any examples

An instance of an IDL **any**, which has had an instance of an IDL unbounded **sequence** of IDL **long** elements inserted into it:

```
typedef sequence<long> LongSeq;
typedef my_any;
```

shall be represented in XML as:

```
<my_any>
   <typecode>
      <kind>
         <TCKind>tk_sequence</TCKind>
      </kind>
      <element_typecode>
         <kind>
            <TCKind>tk_long</TCKind>
         </kind>
      </element_typecode>
      <bound>0</bound>
   </typecode>
   <value>
      <item>1</item>
      <item>1</item>
      <item>2</item>
      <item>3</item>
      <item>5</item>
      <item>8</item>
   </value>
</my_any>
```

An instance of an IDL **any**, which has had an instance of an IDL struct type such as the following IDL struct named **Example** inserted into it:

```
struct Example {
   short member1;
   short member2;
   long member3;
};
```

shall be represented in XML as:

```
<my_any>
   <typecode>
      <kind>
         <TCKind>tk_struct</TCKind>
      </kind>
      <id>IDL:Example:1.0</id>
      <name>Example</name>
   </typecode>
   <value>
      <member1>100</member1>
      <member2>50</member2>
```

```
      <member3>10000</member3>
   </value>
</my_any>
```

10.3 IDL Interfaces

In CORBA, an interface is a description of a set of possible operations that a client may request [CORBA 3.3, Part 1, Section 5.2.2, Requests] of an object [CORBA 3.3, Part 1, Section 5.2.1, Objects], through that interface [CORBA 3.3, Part 1, Section 5.2.5, Interface].

The following sections define the mapping between:

- an HTTP request targeting a URI resource and containing data representation of that resource in XML format, and a CORBA request targeting the object, interface, operation, or attribute anchored to that resource.
- an HTTP response produced by a URI resource and containing data representation of that resource in XML format, and a CORBA reply originating from the object, interface, operation, or attribute anchored to that resource.

10.3.1 CORBA Request and HTTP Request message body mapping

The HTTP request message body shall contain an XML document containing in its root an XML element, denoted as the *request wrapper element.*

The name of the *request wrapper element* XML element shall be set as follows:

- The IDL operation name, or IDL attribute name is converted to Pascal case, with the first character replaced with an uppercase character. All underscore characters will be removed, and the character that was immediately following any underscore character is to replaced with an uppercase character.
- The text **Request** is then appended.

The *request wrapper element* shall contain XML elements to represent each of the IDL operation parameters that have directional parameter attributes of type *in* and *inout* as follows:

- the IDL operation parameter name shall map to an XML element of the same name.
- the IDL operation parameter value, represented in XML according to the rules defined in XML Data Representation format - section 10, shall map to the XML element value.

Compliant implementations may optionally include the representation of *in* and *inout* parameters that have been annotated with an **@PathParam** or **@QueryParam** annotation in the *request wrapper element*.

The HTTP request header *Content-Type* shall be set to the value *“application/xml”*.

10.3.1.1 HTTP Request message body example

```
@Path(“/sample/{objkey}”)
Interface SampleInterface { /* … */}

struct SampleStruct {
   string struct_member_string;
   long struct_member_long;
};

@Path(“/sample_service”)
interface SampleServiceInterface {
```

```
   @POST
   @Path("sample_operation")
   SampleInterface sample_operation(in long a_in_param,
                                    inout SampleStruct an_inout_param ,
                                    out string an_out_param);
};
```

The IDL operation *sample_operation()* operation *in* parameter **a_in_param** and the *inout* parameter **an_inout_param** will be marshalled in the *request wrapper element* in the HTTP request message body.

The excerpt below shows the corresponding HTTP request message:

```
POST /sample_service/sample_operation HTTP/1.1
Content-Type: application/xml
/* … HTTP Header not shown … */
<SampleOperationRequest>
   <a_in_param>1234</a_in_param>
   <an_inout_param>
      <SampleStruct>
         <struct_member_string>a struct sample value</struct_member_string>
         <struct_member_long>54321</struct_member_long>
      </SampleStruct>
    </an_inout_param>
</SampleOperationRequest>
```

10.3.2 CORBA Reply and HTTP Response message body mapping

The HTTP response message body shall contain an XML document containing in its root an XML element, denoted as the *response wrapper element.*

The name of the *response wrapper element* XML element shall be set as follows:

- The IDL operation name, or IDL attribute name is converted to Pascal case, with the first character replaced with an uppercase character. All underscore characters will be removed, and the character that was immediately following any underscore character is to replaced with an uppercase character.
- The text **Response** is then appended.

The *response wrapper element* shall contain XML elements to represent each of the IDL operation parameters that have directional parameter attributes of type *out* and *inout* as follows:

- the IDL operation parameter name shall map to an XML element of the same name.
- the IDL operation parameter value, represented in XML according to the rules defined in XML Data Representation format - section 10, shall map to the XML element value.

The *response wrapper object* shall contain an XML elements to represent IDL operation *return result* as follows:

- the IDL return result shall map to the XML element *<_ret>*
- the IDL return result value, represented in XML according to the rules defined in XML Data Representation format - section 10, shall map to the XML element value.

The HTTP response header *Content-Type* shall be set to the value *"application/xml"*.

10.3.2.1 HTTP Response message body example

@Path("/sample/{objkey}")

```
Interface SampleInterface { /* … */}

struct SampleStruct {
   string struct_member_string;
   long struct_member_long;
};

@Path("/sample_service")
interface SampleServiceInterface {

   @POST
   @Path("sample_operation")
   SampleInterface sample_operation(in long a_in_param,
                                    inout SampleStruct an_inout_param ,
                                    out string an_out_param);
};
```

The IDL operation *sample_operation()* *out* parameter **an_out_param**, the *inout* parameter **an_inout_param**, and the return result will be marshalled in the *response wrapper element* in the HTTP request message body.

The **@Path** annotation applied to the *SampleInterface* interface, containing the special **{objkey}** URI template parameter, meets the conditions defined in section 8.1.4. The *::SampleServiceInterface::sample_operation()* operation returns an object reference to an instance of *SampleInterface* object. This results in the JSON representation of the object reference being contained in the response <_ret></_ret> XML element.

The excerpt below shows the corresponding HTTP response message:

```
HTTP/1.1 200 OK
Content-Type: application/xml
/* … HTTP Header not shown … */
<SampleOperationResponse>
   <_ret>/account/4YYASBSDFJ3456JSDF==</_ret>
   <an_inout_param>
      <SampleStruct>
         <struct_member_string>a struct sample value</struct_member_string>
         <struct_member_long>54321</struct_member_long>
      </SampleStruct>
    </an_inout_param>
   <an_out_param>a sample out param string value</an_out_param>
</SampleOperationResponse>
```

10.3.3 CORBA Reply containing an exception and HTTP Response message body mapping

For CORBA replies containing a CORBA exception, the HTTP response message body shall contain an XML document containing in its root an XML element, denoted as the *exception wrapper element.*

The name of the *exception wrapper element* XML element shall be set as follows:

- The IDL operation name, or IDL attribute name is converted to Pascal case, with the first character replaced with an uppercase character. All underscore characters will be removed, and the character that was immediately following any underscore character is to replaced with an uppercase character.
- The text **Exception** is then appended.

The *exception wrapper element* shall contain an XML element to represent the IDL string containing the RepositoryId for the exception, as defined in the *Interface Repository* clause of CORBA (Part 1), as follows:

- the XML element <*exceptionRepositoryID*>, containing:
- the IDL string containing the RepositoryId for the exception value, represented in XML according to the rules defined in XML Data Representation format - section 10, shall map to the XML element value.

The *exception wrapper element* shall contain an XML element to represent the IDL exception members as follows:

- the XML element <*exceptionMembers*>, containing:
- the XML representation of the members of the exception, if any. Members of an IDL exception shall be represented in XML in the same manner as members of an IDL struct, as defined in Struct Types, section 10.1.3.1.

The HTTP response header *Content-Type* shall be set to the value *"application/xml"*.

10.3.3.1 HTTP Response message body exception example

```
@Path("/sample_service")
interface SampleServiceInterface {

   exception SampleException {
      long sample_exception_id;
      string sample_exception_string;
   };

   @GET
   @Path("sample_operation")
   void sample_operation (in long a_in_param) raises SampleException;
};
```

When the IDL operation *sample_operation()* raises the IDL exception *SampleException*, this exception will be marshalled in the *response wrapper element* in the HTTP response message body.

The excerpt below shows the corresponding HTTP response message:

```
HTTP/1.1 200 OK
Content-Type: application/xml
/* … HTTP Header not shown … */
<SampleOperationException>
   <exceptionRepositoryID>IDL:SampleServiceInterface/SampleException:1.0</exceptionRepositoryID>
   <exceptionMembers>
      <sample_exception_id>10202</sample_exception_id>
      <sample_exception_string>a sample exception string value</sample_exception_string>
   </exceptionMembers>
</SampleOperationException>
```

This page intentionally left blank.

Appendix A. Exposing a REST API to a sample CORBA application (Non-Normative)

This appendix walks through the steps involved in exposing a REST API to a sample CORBA application using the IDL-RS annotations. The resulting REST API can then be consumed by REST client applications to invoke the services provided by the CORBA application via REST.

This appendix covers the following topics:

- Overview of the sample CORBA Image Processing application
- Defining the desired REST API to the CORBA Image Processing application
- Annotating the CORBA Image Processing application IDL with IDL-RS annotations
- Example output of an interaction with the REST API to the CORBA Image Processing application

A.1. Example CORBA Image Processing Application overview

The example application provides facilities to manipulate graphical images and apply image transformations such as grayscale or sharpen to the images. These manipulation and transformation facilities are implemented on the server side by the following CORBA Objects::

- an *ImageFactory* interface, which can receive/create and store Images; and
- an *Image* interface, which provides the capability to apply image transformations.

A typical CORBA client would use the CORBA operations defined in the above interfaces:

- the *ImageProcessing::ImageFactory::create_image()* IDL operation, to send the image data to the CORBA Server for the creation of an Image Object.
- the *ImageProcessing::ImageFactory::list_images()* IDL operation, used to retrieve a list of CORBA Object references to the corresponding Image Objects located in the server.
- the *ImageProcessing::Image* interface provides several image transformation operations such as: *grayscale()*, *sharpen()*, and *edge_detection()*.
- the *ImageProcessing::Image* interface also provides a *delete_image()* operation, used to destroy instances of Image Objects.

The IDL for this Image Processing application is listed below:

```
module ImageProcessing
{
  exception UnknownImageFormat
  {
    // complete
  };

  interface Image;

  typedef sequence<octet> ImagePayload;
  typedef sequence<Image> ImageSeq;

  interface ImageFactory
  {
    Image create_image(in ImagePayload imgBytes) raises(UnknownImageFormat);
```

```
        ImageSeq list_images();
    };

    interface Image
    {
        readonly attribute ImagePayload img_data;

        void grayscale();
        void sharpen();
        void edge_detection();
        void declassify();
        void delete_image();
    };
};
```

The first step in applying the *REST for CORBA Specification* is to annotate the IDL file above using the IDL-RS annotations, as specified in section 8.

A.2. Defining the REST API to the CORBA server

The following table outlines the desired REST API associated to the CORBA IDL interfaces, operations and attributes.

Desired REST API	Corresponding mapped IDL Operation/Attribute
POST /image-processing	MF_ImageProcessing::ImageFactory::create_image()
GET /image-processing	MF_ImageProcessing::ImageFactory::create_image()
GET /image-processing/images/{objkey}	MF_ImageProcessing::Image::img_data()
POST /image-processing/images/{objkey}/grayscale	MF_ImageProcessing::Image::grayscale()
POST /image-processing/images/{objkey}/sharpen	MF_ImageProcessing::Image::sharpen()
POST /image-processing/images/{objkey}/edge-detection	MF_ImageProcessing::Image::edge_detection()
POST /image-processing/images/{objkey}/declassify	MF_ImageProcessing::Image::declassify()
DELETE /image-processing/images/{objkey}	MF_ImageProcessing::Image::delete_image()

Table A.1: Mapping CORBA API to desired REST API

A.3. Annotating IDL with IDL-RS annotations

The desired REST API can be realized by applying IDL-RS annotation to the original IDL file as follows (IDL-RS annotations are emphasized using a **bold** typeface):

```
module ImageProcessing
{
    exception UnknownImageFormat
    {
        // complete
    };

    interface Image;

    typedef sequence<octet> ImagePayload;
    typedef sequence<Image> ImageSeq;

    @Path(uri = "/image-processing", rir = "corbaloc::example.com:8080/ImageFactory")
    interface ImageFactory
    {
        @POST
        Image create_image(in ImagePayload imgBytes) raises(UnknownImageFormat);

        @GET
        ImageSeq list_images();
    };

    @Path("/images/{objkey}")
    interface Image
    {
        @GET
        readonly attribute ImagePayload img_data;

        @Path("grayscale")
        @POST
        void grayscale();

        @Path("sharpen")
        @POST
        void sharpen();

        @Path("edge-detection")
        @POST
        void edge_detection();

        @Path("declassify")
        @POST
        void declassify();

        @DELETE
        void delete_image();
    };
};
```

A.4. Example output from REST client interaction

The example that follows demonstrates a typical interaction between a simple REST client application and the REST API defined above. The example used the well-known command-line tool *curl*.

The following curl commands invoke a HTTP **GET** request on the URI /**image-processing** which maps to the *ImageFactory* CORBA Object. The HTTP response from this URI returns a collection of URIs which identify unique REST resources representing Image instances in the server. Note the use of the HTTP *Accept* header to control the data representation format in the response.

```
# curl -s http://example.com:8080/image-processing -H 'Accept: application/xml' | xmllint --format -
<?xml version="1.0"?>
<list_imagesResponse xmlns="">
  <_ret>
    <item>/images/AVZCAQIAAAAvACAgAQAAADAgICCcwfdhiDwOAA==</item>
    <item>/images/AVZCAQIAAAAvACAgAQAAADEgICCcwfdhiDwOAA==</item>
    <item>/images/AVZCAQIAAAAvACAgAQAAADIgICCcwfdhiDwOAA==</item>
  </_ret>
</list_imagesResponse>

# curl -s http://example.com:8080/image-processing -H 'Accept: application/json'  | json_pp
{
   "_ret" : [
      "/images/AVZCAQIAAAAvACAgAQAAADAgICCcwfdhiDwOAA==",
      "/images/AVZCAQIAAAAvACAgAQAAADEgICCcwfdhiDwOAA==",
      "/images/AVZCAQIAAAAvACAgAQAAADIgICCcwfdhiDwOAA=="
   ]
}
```

The following curl command invokes a HTTP **GET** request on the URI **/image-processing/images/{objkey}** which identifies a unique *Image* resource. The **{objkey}** URI path template parameter is used to uniquely identify a REST resource and the corresponding CORBA object, as specified in section 8.1.4, Obtaining string representations of IDL object references.

The HTTP response from this API contains the source Image encoded in the chosen media type (the binary data is omitted in the listing below).

```
# curl -s
http://example.com:8080/image-processing/images/AVZCAQIAAAAvACAgAQAAADEgICCcwfdhiDwOAA==
-H ‘Accept: application/json’
{
   "_ret":[
      137,80,78,71,13,10,26,10,0,0,0,13,73,72,68,82,0,0,4,0,0,0,2,179,8,2,
      ...OMITTED…
]}
```

The following curl command invokes a HTTP **POST** request on the URI **/image-processing/images/{objkey}/edge-detection**. This URI extends the URI invoked upon in the previous command by appending **edge-detection** and results in the invocation of a transformation on the image.

```
# curl -s
http://example.com:8080/image-processing/images/AVZCAQIAAAAvACAgAQAAADEgICCcwfdhiDwOAA==/
edge-detection -X POST
```

The following curl command invokes a HTTP **DELETE** request on the URI **/image-processing/images/{objkey}** which identifies a unique image REST resource. The correspoding CORBA *ImageProcessing::Images::delete_image()* IDL operation will be executed on Image object mapped to the URI.

```
# curl -s
http://example.com:8080/image-processing/images/AVZCAQIAAAAvACAgAQAAADEgICCcwfdhiDwOAA==
-X DELETE
```

The final curl command repeats the first example invocation to demonstrate that the available image resources have now been reduced as a result of the previous HTTP **DELETE** curl command invocation.

```
# curl -s http://example.com:8080/image-processing | json_pp
{
   "_ret" : [
      "/images/AVZCAQIAAAAvACAgAQAAADAgICCcwfdhiDwOAA==",
      "/images/AVZCAQIAAAAvACAgAQAAADIgICCcwfdhiDwOAA=="
   ]
}
```

Appendix B. Securing a REST for CORBA application (Non-Normative)

This appendix illustrates how security might be added to the example REST for CORBA application described in Appendix A.

A valid approach would entail the deployment of the REST for CORBA application in a webserver that offers Transport Layer Security capabilities. TLS security certificates can be configured at the REST service endpoint and at REST client endpoints. This would enable authentication, confidentiality and integrity in the communication between the REST clients and the REST service endpoint.

The following example demonstrates how to consume the secured REST for CORBA application described in Appendix A. (some output has been omitted for brevity):

```
# curl -v --cacert <path-to-ca-certificate> --cert  <path-to-client-certificate> --key <path-to-client-private-key>
https://example.com:8080/image-processing | json_pp
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Request CERT (13):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Certificate (11):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS handshake, CERT verify (15):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* Server certificate:
...OMITTED...
> GET /image-processing HTTP/1.1
> Host: example.com:8080
> User-Agent: curl/7.76.1
> Accept: */*
...OMITTED...
{
   "_ret" : [
      "/images/AVZCAQIAAAAvACAgAQAAADAgICCcwfdhiDwOAA==",
      "/images/AVZCAQIAAAAvACAgAQAAADIgICCcwfdhiDwOAA=="
   ]
}
```