Windchill REST Services User's Guide

1.0

Contents

1

Windchill REST Services Overview

This section explains the basics of REST, OData, and Windchill REST Services.

REST

Representational state transfer or REST is an architectural pattern for web services. In this architecture, business objects on the server are represented as web resources, which are acted upon by clients using HTTP verbs such as, GET, POST, PATCH, and DELETE.

For example, consider a RESTful web service for parts that exposes a web resource `/Parts`. To get a list of parts, the clients of this web service send an HTTP GET request to `/Parts`. To create a part, the clients send a POST request to `/Parts` and specify a payload of the attribute values needed to create the part.

OData

OData (Open Data Protocol) is an ISO/IEC approved, OASIS standard for building and consuming RESTful web services. OData enables exchange of data across web clients using HTTP messages.

The OData protocol mandates that a compliant web service must:

- declare an Entity Data Model (EDM) at a well-known URL.
- provide a uniform way to form URLs for entities and entity sets defined in the EDM.
- enable clients to send HTTP requests POST, GET, PATCH, and DELETE on entity and entity set URLs for creating, reading, updating and deleting entities.
- support request headers and query parameters for client interaction as defined in the standard.

Please refer to the following resources for more information on OData, version 4.0:

- OData.org—Documentation on basics of OData.
- OData Protocol—Documentation of the protocol.
- Common Schema Definition Language (CSDL)—Documentation of the format used to document the EDM of a service.
- URL Conventions—Documentation of how to form URLs for entities and entity sets of an OData service.

Windchill REST Services

Windchill REST Services is a Windchill module that enables developers to configure OData services on Windchill data. This module is installed on top of the supported Windchill versions. The module comprises of a framework and a set of PTC domains. The domains are configured using the functionality available in the framework.

Framework

The framework is designed to read a set of configuration files which is used to set up domains. Domains are OData services and entities. After the framework is set up, it uses the configuration files to build the EDM of the domain. The configuration files are used to show the entities and entity sets available in a domain. While setting up the entity configurations, you can map entities to Windchill persistables.

The framework processes the HTTP requests POST, GET, PATCH and DELETE made to entities. It creates, reads, updates and deletes Windchill persistables mapped to entities.

The framework allows you to configure your own domains. You can also extend the domains provided by PTC.

PTC Domains

Windchill REST Services include a set of domain configurations for specific functional areas of Windchill. The following domain configurations are available:

- **PTC Product Management Domain**
- **PTC Document Management Domain**
- **PTC Data Administration Domain**
- **PTC Principal Management Domain**
- **PTC Common Domain**

2

Installing Windchill REST Services

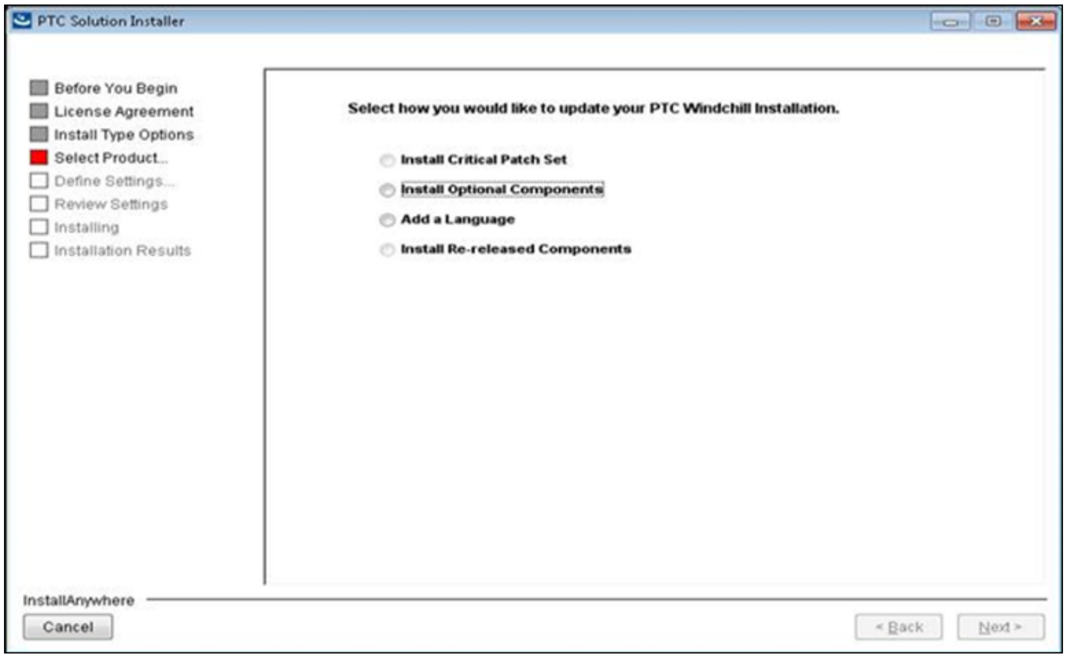
Installation Prerequisites

Windchill REST Services require the supported version of Windchill to be installed on your system. Windchill REST Services 1.0 is supported on Windchill 11.0 M020 and Windchill 11.0 M030.

Installation Process

From Windchill 11.1 F000 onward, Windchill REST Services is a mandatory component of PTC Solution Installer (PSI).

Windchill REST Services 1.0 will also be bundled with future CPS of Windchill 11.0 M020 and Windchill 11.0 M030. Windchill REST Services will be automatically installed when you install the CPS.



Select how you would like to update your PTC Windchill Installation.

- Install Critical Patch Set
- Install Optional Components
- Add a Language
- Install Re-released Components

When installing the CPS, select **Install Critical Patch Set** option to include Windchill REST Services in the installation.

If Windchill REST Services 1.0 is already installed, and you want to update it to the next version, perform the following steps:

1. Copy the new Windchill REST Services CD to your staging area.
2. Launch the CPS installer.
3. Select **Install Re-released Components** option.

Windchill REST Services are updated to the new version.

3

Windchill REST Services Framework Capabilities

Overview

Windchill REST Services framework enables configuration of RESTful services based on the OData protocol. The services allow clients to create, read, update and delete entities that are Windchill persistables. The framework provides default processing logic for HTTP requests made to OData URLs of the domains, entities, entity sets, functions and actions.

The framework provides hooks in the default processing logic for customizers to introduce their own custom code. Using these hooks, customizers can override or enhance the default processing available in the framework.

In this release, the complete OData protocol has not been implemented by the framework. The OData protocol supported by the framework are explained in this section.

The framework allows customizers to configure domains, which are OData services. The EDM of a domain defines entities, relationships, entity sets, functions and actions.

The framework supports Windchill capabilities that apply to a class of persistable entities. For example, persistables that implement Workable can all be checked out. Customizers can inherit Windchill behaviors for entities that are being configured.

Support for OData

The framework is designed to support OData Protocol V4. All aspects of OData standard are not currently supported by Windchill REST Services 1.0. PTC intends to support minimum OData V4 compliance in subsequent releases of Windchill REST Services.

The key features of OData that are supported in the framework are discussed in the subsequent sections.

OData Services as Domains

The framework enables you to set up domains which contain entities that are mapped to Windchill types. Domains are equivalent to OData services.

Similar to an OData web service, a domain can be accessed at the Domain Root URL, which is of the form `https://<Windchill server>/<Windchill App Context>/servlet/odata/[<Domain Version>}/<Domain Identifier>/`

where,

- `<Windchill App Context>` is Windchill in a standard installation
- `<Domain Version>` is the version of the domain API and can be `v1`, `v2`, and so on. `<Domain Version>` is optional in the URL and must be specified only if the client needs a specific version of the domain.
  - If `<Domain Version>` is not specified in the URL, the framework checks if the domain version is specified in the `Accept` header.
  - If `<Domain Version>` is not specified in the URL or in the `Accept` header, then the default configured version is used by the framework.
- `<Domain Identifier>` is the identifier for the domain. For example, the identifier for Product Management domain is `ProdMgmt`.

The URL is called the `Domain Root`, which is equivalent to OData service root URL. A GET request to this URL returns the list of entity sets that are available in a domain.

For example:

- The Domain Root URL of the Product Management domain for version 1 is:

  ```
  https://windchill.ptc.com/Windchill/servlet/odata/v1/ProdMgmt/
  ```
- The Domain Root URL of the Product Management domain for the default version is:

  ```
  https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/
  ```

Entity Data Model of a Domain

An Entity Data Model (EDM) is the specification of entities that are available for a domain. Each entity is further defined by its structural and navigation properties. Structural properties have values. Navigation properties are references to other entities in the domain.

The EDM of a domain is defined in Common Schema Definition Language (CSDL). CSDL defines the entity model as an XML representation. The EDM of a domain can be accessed by adding `$metadata` at the end of the Domain Root URL. For example, the URL for EDM of the Product Management domain is:

```
https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/$metadata
```

Clients can send an HTTP GET request to this URL to get the EDM of the Product Management domain. The EDM enables clients to get more information about the entities, relationships, functions, and actions provided by the domain.

OData Primitives

OData primitives are the data types supported by the OData standard.

The following table lists the OData primitives that are supported by the framework. It also shows the recommended mapping of OData primitives to Windchill and Java types.

Framework Type	OData Type	Windchill/Java Type
SByte	Edm.SByte	byte, java.lang.Byte
String	Edm.String	char, java.lang.Character
Int16	Edm.Int16	short, java.lang.Short
Int32	Edm.Int32	int, java.lang.Integer
Int64	Edm.Int64	long, java.lang.Long
Single	Edm.Single	float, java.lang.Float
Double	Edm.Double	double, java.lang.Double
Boolean	Edm.Boolean	Boolean, java.lang. Boolean
String	Edm.String	String
DateTimeOffset	Edm.DateTimeOffset	Timestamp

OData Query Parameters

Windchill REST Services supports the following query options from the OData standard:

- `$filter`—Query criteria to filter the results. OData calls support filter expressions for a broad range of primitives. The framework supports the following expressions:
  - Expressions that use String, Int16, Int32, Int64, Single and Double types
  - Expressions that use comparison operators EQ, NE, GT, LT, GE, LE
  - Expressions that use logical operators AND, OR
  - Expressions that use unary operator NOT
  - Expressions that use the methods `startswith`, `endswith` and `contains`

Note

- Support for types such as `DateTimeOffset` and additional operators will be provided in a future release of Windchill REST Services.
- The `$filter` expressions are only supported for entity level collections. The expressions are not supported for filtering navigations and expansions.

- `$select`—Comma separated list of entity properties that must be returned as a part of the response. For example, in the URL you can list the Document attributes such as, `Name` and `CheckoutState`, to display the name of the document and its checkout status in the response.
- `$top`—Returns a set of entities from the top, that is, the first N entities, in a collection. When `$top` is not specified, by default, a maximum of 25 entities in a collection are returned in the first set of a response. When `$top` is specified, the specified number of entities are returned. The maximum limit for `$top` is 200. If the number of entities returned for a response is more than the `$top` value, then the response includes the URL to the next set.
- `$skip`—Skips a set of entities from the top in a collection, and displays the next set of entities, N+1 entity onward.

PTC Annotations

The OData protocol supports marking elements in EDM with custom annotations that provide additional information to the clients. The Windchill REST Services framework uses two custom annotations to specify operations supported on an entity set and to specify properties of an entity that are read-only. The annotations are:

- PTC.Operations
- PTC.ReadOnly

PTC.Operations

This annotation is used to mark entities with a list of supported operations. In the example below, the annotation is applied to the Part entity in the Product Management domain:

```
<EntityType Name=”Part>
<Key>
    <PropertyRef Name=”ID”/>
</Key>
<Property Name=”ID” Type=”EDM.String”>
…
  <Annotation Term="PTC.Operations">
   <String>READ, CREATE, UPDATE, DELETE</String>
  </Annotation>
</EntityType>
```

The annotation indicates that the framework supports reading, creating, updating, and deleting parts.

PTC.ReadOnly

This annotation is used to mark entity properties that are read-only. In the example below, an annotation from the EDM of the Part Management domain is shown:

```
<EntityType Name=”Part”>
<Key>
    <PropertyRef Name=”ID”/>
</Key>
<Property Name=”ID” Type=”EDM.String”>
    …
<Property Name=”State” Type=”PTC.EnumType”>
        <Annotation Term="PTC.ReadOnly"/>
</Property>
…
</EntityType>
```

The annotation indicates that the property `State` on the `Part` entity is read-only.

Domain Configuration

This section describes how to configure a domain in Windchill REST Services. The folder structure and files that are required to configure a domain are explained in detail.

Configuration Paths and Files

Windchill REST Services reads domain and entity configurations from two locations. It consolidates the configuration files from both these locations to generate a single set of domains and its entities that can be used by clients. The two locations are:

- PTC configuration path—`<Windchill>/codebase/rest/ptc/domain/`, where `<Windchill>` is the Windchill installation directory.
    - This path is reserved for domain configurations that are provided by PTC. The domains and entities are installed at this path.
    - Do not modify the files located at this path.
    - You must not create any new configuration files at this location as future updates from PTC will delete and recreate files in this path.
- Custom configuration path—`<Windchill>/codebase/rest/custom/domain/`, where `<Windchill>` is the Windchill installation directory.
    - This path is provided for custom configuration files.

  - By creating custom configuration files at this location, customizers can extend PTC provided domains, or create new custom domains.

Configuring a Domain

To configure a domain create the following folder structure along with the required JSON files at the custom configuration path:

`<Windchill>/codebase/rest/custom/domain`

- `<Domain Folder>`
  - `<Version Folder>`
    - `complexType`
    - `entity`
    - `import.json`
    - `import.js`
- `<Domain JSON File>`

While configuring a domain, create the `<Domain Folder>`. The name of folder is the name of the domain identifier. The domain identifier of the domain is the name specified by customizers in the `id` property in the `<Domain JSON File>`.

The `<Version Folder>` is the name for the version of the domain API. There can be multiple version folders under the domain folders, each representing the domain configuration for that version. These folders are named `v1`, `v2`, `v3` and so on. The `<Version Folder>` contains subfolders that contain the configuration files for entities, complex types, and configuration for domain imports.

The folder `complexType` contains configuration files for OData complex types. If your domain contains complex types, this folder will contain a `.json` file for each complex type defined in the domain. In OData, complex types are structures of primitive types, and are used to combine related properties. For example, the PTC domain defines a complex type called `EnumType` that combines two string properties, `Value` and `Display`. The `EnumType` complex type is used to represent a Windchill enumeration type. `Value` represents the property value persisted in the database. `Display` represents the localized property value used for display.

The `entity` folder contains two configuration files, a `.json` and a `.js` file, for each entity in the domain. The `.json` file specifies the properties of the entity being configured and the `.js` file contains its hook implementations.

The `import.json` file specifies the other domains that are imported into the domain being configured. Importing other domains in a domain is an OData capability that allows EDM of the imported domains to be used in the referencing domain. For example, PTC Common domain, which is a domain provided by PTC in Windchill REST Services, is imported by all other domains. It contains common constructs, such as, common complex types `EnumType`.

The `import.js` file contains implementations for unbound functions and actions. This file is needed only if unbound actions and functions are defined in the domain.

The `<Domain JSON File>` is a JSON file with the same name as the domain identifier of the domain, and has .json specified as extension in its file name. For example, for the Product Management domain, this file is called `ProdMgmt.json`. The `<Domain JSON File>` contains the metadata configuration for the domain. This file contains properties such as, domain name, domain identifier, and so on.

Domain JSON File

The `<Domain JSON File>` is a JSON file with the same name as the domain identifier, and has `.json` specified as extension in its file name. For example, for domain identifier ProdMgmt, the `<Domain JSON File>` file name is ProdMgmt.json. The file contains configuration metadata for the domain. The configuration metadata is specified in a JSON object with the following properties:

- *name*—Name of the domain. For example, Product Management.
- *id*—An unique identifier of the domain in camel case. For example, ProdMgmt for Product Management domain.
- *description*—Description of the domain.
- *namespace*—An OData identifier that appears in the domain EDM as a namespace qualifier for a domain. For example, PTC.ProdMgmt.
- *containerName*—An OData identifier that appears in the domain EDM as a container for the entity sets of the domain.
- *defaultVersion*—Default version of the domain API that is returned to the clients if they do not request a specific version of the domain API. The values for this property are specified as 1, 2, 3 and so on in the JSON file. The framework interprets the value of 1 as `v1`. It searches for the `<Domain Folder>/v1` folder for entity configurations that must be used for processing requests. Similarly, a value of 2 is interpreted as `v2`, and so on.

For example, the `ProdMgmt.json` file from the Product Management domain is as shown below:

```
{
  "name":"Product Management Domain",
  "id":"ProdMgmt",
  "description":"PTC Product Management Domain",
  "namespace":"PTC.ProdMgmt",
  "containerName":"Windchill",
  "defaultVersion":"1",
}
```

Importing JSON File

The `import.json` file is used to specify the domains that are imported by the domain being configured. The imported domains are specified in the `imports` property, which is a collection of JSON objects. Each object is of the form:

```
{name="<domain name>", version="<domain version>"}
```

where,

- `<domain name>` is the name of the domain being imported
- `<domain version>` is the version of the domain being imported

An example of `import.json` for the Product Management domain. The file shows that the domains PTC, DataAdmin, DocMgmt and PrincipalMgmt with version 1 are being imported into the Product Management domain:

```
{
 "imports":[
   {name="PTC", version="1"},
   {name="DataAdmin", version="1"},
   {name="DocMgmt", version="1"},
   {name="PrincipalMgmt", version="1"},
  ],
  "functions":[],
  "actions":[],
}
```

Versioning of the Domain API

Windchill REST Services supports versioning of the APIs provided by a domain. The domain configurations are defined in version specific folders, such as, `v1`, `v2`, `v3`, and so on.

Clients can request a specific version of a domain resource in the URL. For example, the URL to request version 1 of the Product Management domain is:

```
https://windchill.ptc.com/Windchill/servlet/odata/v1/ProdMgmt/
```

Alternately, the version can be specified in the `Accept` header of the HTTP request. For example, to request version 1 of the Product Management domain use the URL:

```
https://windchill.ptc.com/Windchill/servlet/odata/ ProdMgmt/
```

Specify the version in the `Accept` header as:

```
application/vnd.ptc.api+json;version=3
```

You must specify the version only if clients need a specific version for backward compatibility. If not, it is recommended that version must not be specified in the URL or `Accept` header. The server must send the default version of the API.

Configuring Unbound Functions

The framework supports configuring unbound functions in a domain. In OData protocol an unbound function is considered to be an operation that does not change the state of a service. The framework treats an unbound function as a read-only operation available in a domain. After the unbound function is configured, it is invoked by a GET request to the URL:

```
<Domain Root>/<Unbound Function Name>(<param1>=<value1>, <param2>=<value2>)
```

To configure an unbound function, perform the following steps:

1. Specify the properties of the function: name, input parameters, and return type.
2. Define the implementation logic for the function.

The properties of an unbound function are specified in the `import.json` file of the domain. In the file, under `functions`, specify the following properties:

- *name*—Name of the unbound function. The function is invoked from the URL with this name.
- *importName*—Name of the import operation.
- *description*—Description of the function.
- *includeInServiceDocument*—This is applicable for unbound functions. Defines if the function can be requested as a service in the container. The default value is set to false.
- *parameters*—A collection of parameters to be passed to the function. You can specify multiple parameters separated by commas. Specify the following parameters for a function:
    - *name*—Name of the parameter.
    - *type*—Type of the parameter. The parameter can be a primitive or an entity type. If the parameter is an entity type, the value is specified in the URL to the entity.
    - *isNullable*—Specifies if a property can be set as null. The default value is set to false.
    - *isCollection*—Specifies if the property represents a collection. The default value is set to false.
- *returnType*—Information about what the function returns. Specify the following properties for the return types:
    - *type*—Type of object that is returned. The return type can be a primitive or an entity type.

- *isNullable*—Specifies if a property can be set as null. The default value is set to false.
- *isCollection*—Specifies if the property represents a collection of objects or entities. The default value is set to false.

For example, consider a configuration file `import.json` for an unbound function `GetEndItems`. The function returns a collection of Part entities, and takes no input parameters:

```
{
  "imports:[
     {name="PTC", version="1"},
     {name="DataAdmin", version="1"},
  ],
  "functions":[{
     "name": "GetEndItems",
     "description": "Gets a list of end items parts",
     "parameters": [],
     "returnType": {
        "type": "Part",
        "isCollection": true,
     }
  }]
"actions":[]
}
```

An unbound function is implemented in the `import.js` file. The function is implemented as below:

- The function name starts with `function_`, followed by the name of the function that is defined in the `import.json` file.
- The function takes two input parameters:
  - *data*—The parameter is of type `FunctionProcessorData`, which is a data structure, that contains information about the processing logic of the framework. This information can be used while implementing the function.
  - *params*—The parameter is of type `Map<String, Parameter>` and it contains the hashmap of input parameter names and values passed by the client to the function. The name of the input parameter is the key in the map and the parameter value passed by the client is the value for the key.

Configuring Unbound Actions

The framework supports configuring unbound actions in a domain similar to unbound functions. An unbound action is considered as an operation that can change the state of a service. Due to this, the framework supports calling an action with a POST request. After the unbound action is configured, it is invoked by a POST request to the URL `<Domain Root>/<Domain Namespace>.<Unbound Action Name>`. The body of the POST request contains the parameters that will be passed to the action.

An unbound action is configured in the `import.json` file in the same way as an unbound function except the following differences:

- An unbound action is specified in the `actions` collection property in the `import.json` file.
- While specifying the action in the `import.json` file, the property `includeInServiceDocument` is not applicable to actions. This is because actions cannot be included in the service document available at the domain root.
- The action names defined in the `import.json` file start with `action_`.

Configuring Entities in a Domain

Entities available in domains are configured by creating the following two files in the `entity` folder:

- `<Entity JSON>`
- `<Entity JS>`

The name of the `<Entity JSON>` file is the plural of the entity name, and has `.json` specified as extension in its file name. For example, `Parts.json` contains the configuration of the entity `Part`. Similarly, `<Entity JS>` file is the plural of the entity name, and has `.js` specified as extension in its file name.

The `.json` file specifies the structural properties, navigation properties, inheritance of Windchill functionality, bound functions, and bound actions of an entity. The `.js` file contains JavaScript implementation of the bound actions and functions of an entity, and also contains implementation of hooks provided by customizers to override or enhance framework processing logic for the entity.

Basic Information for Configuring Entities

To configure an entity, the framework requires information on the following entity properties:

- *name*—Name of the entity. For example, Part.
- *collectionName*—Name of the entity collection. For example, Parts.
- *type*—Type of entity. Set the value as `wcType` for entities that are backed by Windchill types. For other entities specify the value as `basic`.
- *wcType*—This property must be set if *type* property is specified as `wcType`. The entity type is backed by Windchill types. For example, `wt.part.WTPart`.

- *description*—Description of the entity type.
- *operations*—List of CRUD operations that are permitted on the entity. For example, `CREATE`, `READ`, `UPDATE`, and `DELETE` are permitted by default.

Configuring Structural Properties

Structural properties in OData are the attributes that define a business object or entity. Windchill REST Services reads the properties of attributes, which is a JSON array, from the `<Entity JSON>` file. Each entry in the attribute defines one structural property for the entity being configured. The structural property comprises of the following parameters:

- *name*—Name of the structural property.
- *internalName*—Internal name of the Windchill property that corresponds to the structural property being configured.
- *type*—Framework data type for the structural property being configured. Framework data type is same as the OData primitive type without the Edm prefix. For example, the framework data type corresponding to OData `Edm.Double` is `Double`.

Configuring Navigation Properties

Navigation properties in OData are the reference attributes of an entity that points to another entity. By default, navigation properties are not available on an entity representation when it is accessed by an OData client. The OData client must expand these properties explicitly if they want the associated entities to be available when the entity is being accessed. Windchill REST Services reads the navigation property, which is a JSON array, from the `<Entity JSON>` file. Each entry in the `navigations` section defines one navigation property for the entity being configured. The navigation property consists of the following parameters:

- *name*—Name of the navigation property.
- *target*—OData entity set which is the target of this navigation. The entity being configured is the source entity. The entity set to which the navigation is being configured is the target entity.
- *type*—OData type of the target entity set.
- *isCollection*—Boolean which checks if the navigation to the target entity results in an entity set.
- *containsTarget*—Boolean which checks if the navigation property is a containment navigation property. A containment navigation property in OData enables the read URL of a navigation property to be implicitly treated as an entity set.

Configuring Bound Functions

The framework supports configuring functions that are bound to entities. A function bound to an entity is invoked on an instance of the entity. In OData protocol, an unbound function is considered to be an operation that does not change the state of an entity instance on which it is invoked. After the bound function is configured, it is invoked by a GET request to the URL:

```
<Domain Root>/<Entity Set>(<key>)/<Bound Function Name>(<param1>=<value1>, <param2>=<value2>)
```

A bound function is configured in the same way as an unbound function except the following differences:

- A bound function is specified in the `functions` collection property in the `<Entity JSON>` file.
- While specifying the function in the `<Entity JSON>` file, the property `includeInServiceDocument` is not applicable to bound functions. This is because bound functions cannot be included in the service document available at the domain root.
- The first parameter in the function specification is called the binding parameter. The parameter must be of the same type as the entity that is bound to the function.
- The function names defined in the `import.js` file start with `function_`.

Configuring Bound Actions

The framework supports configuring actions that are bound to entities. An action bound to an entity is invoked on an instance of the entity. OData protocol considers a bound action to be an operation that changes the state of an entity instance on which it is invoked. After the bound action is configured, it is invoked by a POST request to the URL:

```
<Domain Root>/<Entity Set>(<key>)/<Domain Namespace>.<Bound Action Name>
```

The body of the POST request contains the parameters that must be passed to the action.

A bound action is configured in the same way as an unbound action except the following differences:

- A bound action is specified in the `actions` collection property in the `<Entity JSON>` file.
- While specifying the action in the `<Entity JSON>` file, the property `includeInServiceDocument` is not applicable to bound actions. This is because bound actions cannot be included in the service document available at the domain root.

- The first parameter in the action specification is called the binding parameter. The parameter must be of the same type as the entity that is bound to the action.
- The action names defined in the `import.js` start with `action_`.

Inheriting Windchill Capabilities

Windchill provides a capability called Workable for its business objects. Windchill persistables which implement this capability have certain attributes such as `CheckoutState`. Further, persistables which implement Workable can be checked out and checked in.

The framework supports the Workable capability of Windchill and allows entities being configured to inherit this capability. An entity that inherits Workable automatically inherits the structural property `CheckoutState` without having to explicitly configure it in the `<Entity JSON>` file. Also, the entity inheriting Workable automatically gets bound actions such as, CheckIn, CheckOut, and UndoCheckOut without having to explicitly define them.

Workable is one of the capabilities supported by the framework. The complete list of Windchill capabilities supported by the framework is shown in the following table:

Windchill Capability	Inheriting Entity Behavior
`versioned`	Entity properties are automatically enabled: • VersionID—Version identifier of the entity • Revision—Revision of the entity • Version—Version of the entity • Latest—Checks if the entity is the latest version Entity navigation properties are automatically enabled: • Versions—Collection of all entity versions • Revisions—Collection of latest iteration of each entity revision Bound actions are automatically enabled: • Revise—Revises the entity when called
`contextManaged`	Entity navigation properties are automatically enabled:

Windchill Capability	Inheriting Entity Behavior
	• Context—Supports navigation to a Container
`lifecycleManaged`	Entity properties are automatically enabled: • LifeCycleTemplateName • State
`viewManageable`	Entity properties are automatically enabled: • View
`workable`	Bound actions are automatically enabled: • CheckOut—Checks out the entity • CheckIn—Checks in the entity • UndoCheckOut—Undo an entity checkout • IsCheckoutAllowed—Checks if checkout is allowed on an entity
`representable`	Entity navigation properties are automatically enabled: • Representations—Supports navigation to a viewable representation from the entity

Windchill Capability	Inheriting Entity Behavior
`organizationOwned`	Entity navigation properties are automatically enabled: • Organization—Supports navigation to the organization principal of the entity
`foldered`	Entity properties are automatically enabled: • FolderName—Folder where the entity is located • CabinetName—The cabinet where the folder lives • FolderLocation—The path to the folder Entity navigation properties are automatically enabled: • Folder—Supports navigation to the folder of the entity

For an entity to inherit one of these capabilities, edit the `<Entity JSON>` file and add the following entry in the collection for `inherits` property:

```
{“name”: “<Windchill Capability>”}
```

An example showing an entity inheriting `versioned` and `workable` capabilities is shown below:

```
{
…
“inherits”: [
     {
        “name”: “versioned”
      }, {
        “name”: “workable”
      }
]
…
}
```

Excluding Subtypes of Enabled Windchill Types

When an entity is configured with a Windchill type, any subtypes of the Windchill type are also included in the output of the entity queries. The framework enables you to exclude Windchill subtypes from being mapped to entities. To do this, add the following entry in the `<Entity JSON>` file:

```
"wcExcludedTypes":[“<Windchill Subtype 1>”, “<Windchill Subtype 2>”, …]
```

For example, consider a `WTDocument` with subtypes `Agenda` and `Plan`. If you want to exclude these subtypes when the Document entity is created, add the following entry in the JSON file:

```
“wcExcludedTypes”: [“org.rnd.Agenda”, “org.rnd.Plan”]
```

Disabling Entity Set for an Entity in the Service Document

When an entity is configured, by default the entity set of that entity is available in the service document of the domain. Customizers can choose to remove the entity set from the service document. To do this, set the property `includeInServiceDocument` to `false` in the `<Entity JSON>` file. When you remove an entity set from the service document, the entity set is hidden from the clients that use the service document.

Processing HTTP Requests for OData URLs

The framework provides the default processing for HTTP requests made by clients to OData URLs.

OData URLs for EDM such as, `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/$metadata`, are processed by Domain Provider, Entity Provider, and Entity Delegate classes. The Domain Provider and Entity Provider classes read and process the configuration files. The Entity Delegate classes create the metadata response for entities in the domain.

OData URLs for entities and entity sets are processed by entity processor classes. The framework provides two types of processor classes, `BasicEntityProcessor` and `PersistableEntityProcessor`.

The `BasicEntityProcessor` class is used to process requests for entities and entity sets that are not mapped to Windchill persistables.

The `PersistableEntityProcessor` class is used to process requests made to entities and entity sets that are mapped to Windchill persistables.

A GET request to the URL for an entity set is processed by the `PersistableEntityProcessor` class. For example, consider a GET request to the URL `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts`, which is processed by the `PersistableEntityProcessor`. While processing a GET request, the default processing logic of the framework reads the persistable objects of the mapped type, in this case `WTParts` from Windchill. Each object is converted into relevant OData entity. The framework then returns the entity set in the format requested by the client.

A GET request to the URL for an entity is also processed by the `PersistableEntityProcessor` class. For example, consider a GET request to the URL `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts('OR:wt.part.WTPart:87676')`, which is processed by the `PersistableEntityProcessor`. In this case, the default processing in the framework reads the specific persistable object identified by the object reference in the URL, converts it to an entity, and then returns the entity representation in the requested format.

While processing a POST request on an entity set URL, the framework takes the entity representation provided in the POST body in the specified format, converts it into a Windchill persistable, and saves it to Windchill.

PATCH request to an entity URL reads the persistable, changes it, and then sends the updated representation in the response based on the requested format. DELETE request works similar to the PATCH request.

The framework enables customizers to override or enhance the default processing. When processing entity requests the framework searches for JavaScript implementation of hooks in the `<Entity JS>` file of the entity being processed. If hooks are found, then the framework executes the code they contain. Depending on the value returned from the hooks, the framework either continues its default processing or abandons it.

The hooks available in the framework are explained below:

`Object readEntityData(EntityProcessorData)`

The framework calls this hook while reading the backing persistent object for the given id of an entity. The id for the entity can be obtained from `EntityProcessorData`. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Collection readEntitySetData(EntityProcessorData)`

The framework calls this hook while reading the backing persistent objects for the given type of persistent objects. It returns a collection of persistent objects. The given type can be obtained from `EntityProcessorData`. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Map<Object, Collection> getRelatedEntityCollection(NavigationProcessorData)`

The framework calls this hook while navigating from source to target entities. It returns a map, the keys of which are the source objects, and the value of each key is a collection of target persistables obtained by navigating from the source. This hook must be implemented for any navigation specified in the JSON entity configuration file. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing

`Entity createEntityData(Entity, EntityProcessorData)`

The framework calls this hook while creating a persistent object for the given entity that is passed in the POST request. This hook is the main hook that calls other hooks such as, `operationPreProcess`, `storeNewObject`, and `operationPostProcess` at various stages of the create process. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Entity updateEntityData(Entity, EntityProcessorData)`

The framework calls this hook while updating a persistent object for the given entity that is passed in the PATCH request. This hook is the main hook that calls other hooks such as, `operationPreProcess`, `storeNewObject`, and `operationPostProcess` at various stages of the create process. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`void deleteEntityData(Entity, EntityProcessorData)`

The framework calls this hook when deleting the persistent object specified by the entity on the DELETE request. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Object readMediaEntity(Entity)`

The framework calls this hook while reading the given media entity. The hook must be implemented to return an object that is the binary media for the given entity. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Entity createMediaEntity(Entity, EntityProcessorData, ContentType)`

The framework calls this hook while creating a media entity for the entity and `contentType` is passed to the POST request. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Entity updateMediaEntity(Entity, EntityProcessorData, ContentType)`

The framework calls this hook while updating a media entity for the entity and `contentType` is passed to the PUT request. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Map<Entity, Object> toObjects(EntityCollection, EntityProcessorData)`

The framework calls this hook when it is converting entities to objects. For example, during create, update, and delete requests. This hook must be implemented to return a map. The keys of the map are entities, and the value for each key is the persistable object that corresponds to the key entity. You can use this hook to process additional entity attributes that need special processing. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Map<Object, Entity> toEntities(Collection, EntityProcessorData)`

The framework calls this hook when it is converting persistable objects to entities. For example, during read requests. This hook must be implemented to return a map. The keys of the map are persistable objects, and the value for each key is the entity that corresponds to the key object. You can use this hook to process additional object attributes that need special processing, and are not converted by the framework. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`boolean isValidEntityKey(String, EntityProcessorData)`

The framework calls this hook to check if the Windchill object reference string, which is used as a primary key for entities in the ID attribute is valid. Customizers should implement this hook if they are changing the entity key or need to do additional validations on the Windchill object reference. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`boolean isValidNavigation(String name, Object sourceObj, String id, EntityProcessorData processorData)`

The framework calls this hook when navigating from source to target entities. The implementation of this hook should check if the navigation from a source object to a target is a valid. The implementation must return true if the navigation is valid, return false if the navigation is invalid, and return null if the navigation is not defined in the JSON file. This hook must be implemented for any navigation specified in the JSON entity configuration file. If the hook implementation sets continue processing the flag in `EntityProcessorData`, the framework continues with the default processing. Otherwise, the framework abandons the default processing.

`Object operationPreProcess(Object object, Entity entity, EntityProcessorData processorData, PersistableEntityProcessor)`

The framework calls this hook from the `createEntityData` and `updateEntityData` hooks. This hook is called before the start of the transaction to create or update the persistable object. The implementation of the hook creates and returns a persistable object from the new or existing entity, which is passed as an argument to this hook. This method can be overridden to introduce special processing of the object before it is created or updated

`Object storeNewObject(Object, Entity, EntityProcessorData)`

The framework calls this hook from the `createEntityData` hook. This hook is called after the start of the transaction to create the persistable object but before the actual store operation for persistence. The implementation of the hook creates and returns a persistable object from the new entity, which is passed as an argument to this hook. This method can be overridden to introduce special processing of the object before it is created.

`Object saveObject(Object entityObject, Entity entity, EntityProcessorData processorData)`

The framework calls this hook from the `updateEntityData` hook. This hook is called after the start of the transaction to update the persistable object but before the actual save operation for persistence. The implementation of the hook creates and returns a persistable object from the existing entity, which is passed as an argument to this hook. This method can be overridden to introduce special processing of the object before it is updated.

`void operationPostProcess(Object object, Entity entity, EntityProcessorData processorData, PersistableEntityProcessor)`

The framework calls this hook from the `createEntityData` and `updateEntityData` hooks. This hook is called after the store or save operations for persistence but before the end of the transaction commit for creating or updating the persistable object. The implementation of the hook can be used to introduce special processing of the object after it has been created or updated but before the transaction is committed. For example, this hook can be overridden to add associations on the persistable.

`Collection<AttributeData> processAdditionalAttributes(Entity entity, Object entityObject, EntityProcessorData processorData, PersistableEntityProcessor)`

The framework calls this hook while processing additional attributes on an entity. These additional attributes are not defined in the JSON file explicitly but can exist on entities as a result of inheritance. The hook must be implemented to determine the additional attributes, create and return a collection of `AttributeData` objects.

Processing Batch Requests

Using batch requests, you can group multiple operations in a single HTTP request. Use the `$batch` attribute to request the data.

For example, run the batch request as below:

```
https://windchill.ptc.com/Windchill/servlet/odata/<domain>/$batch
```

In a batch request, you can specify a series of individual batch requests or create change sets. Batch requests are represented as multipart MIME message. Specify the batch requests and change sets in relevant `Content-Type` header as distinct MIME parts. The requests are processed sequentially.

Individual batch requests support the following types of requests:

- Getting data
- Modifying data
- Invoking an action
- Invoking a function

If any of the individual batch requests from the series fail, the other batch requests are processed.

Change set is an atomic unit inside which you can define a set of requests. In a change set, you define series of individual batch requests. However, if one or more individual batch requests from the series fail, the entire change set fails. In a change set, if batch requests had modified any data before encountering a failed request, then all the data changes are rolled back. A change set has been implemented as a Windchill transaction.

Change set supports the following types of requests:

- Modifying data
- Invoking an action

Change sets do not support the GET operation.

After execution, batch requests return the appropriate HTTP response codes. The HTTP response body lists the response in the same order as the individual requests in the HTTP request body. However, the requests inside a change set may not be executed in the order specified in the change set.

4

Windchill REST Services Domain Capabilities

PTC Domains

This section explains the domains provided by PTC in Windchill REST Services.

When you install Windchill REST Services, some domains defined by PTC are also installed. These domains enable you to work with Windchill types in the REST architecture. You can also create new custom domains, or extend an existing domain to enable more entities.

A domain in Windchill REST Services represents a RESTful web service, which follows the OData standard. A domain describes its Entity Data Model (EDM) by defining the entity sets, relationships, entity types, and operations.

Overview

This section explains the domains provided by PTC in Windchill REST Services.

The following domains are provided as a part of Windchill REST Services:

- *ProdMgmt*—**PTC Product Management Domain** exposes entities representing parts and BOMs –Windchill objects that are most frequently used while developing products. See the section PTC Product Management Domain on page 34, for more information on the domain.
- *DocMgmt*—**PTC Document Management Domain** provides entities that enable users to manage Windchill documents (`WTDocuments`). See the section PTC Document Management Domain on page 36, for more information on the domain.
- *DataAdmin*—**PTC Data Administration Domain** provides entities that enable users to manage data containers such as, organizations, products, libraries and projects in Windchill. See the section PTC Data Administration Domain on page 37, for more information on the domain.
- *PrincipalMgmt*—**PTC Principal Management Domain** provides entities that work with Windchill groups and users. See the section PTC Principal Management Domain on page 38, for more information on the domain.
- *PTC*—**PTC Common Domain** provides some utility entity types that are commonly used. See the section PTC Common Domain on page 39, for more information on the domain.

PTC Product Management Domain

The Product Management domain provides access to the product management capabilities of Windchill. It provides OData entities that represent business objects like Part and BOM. The following table shows the Windchill items that are enabled with OData entities in the Product Management domain. The Product Management domain references the PTC Document Management domain to provide navigations to reference and describe documents.

The following table lists the significant OData entities available in the Product Management domain. To see all the available OData entities in the Product Management domain, please refer to its EDM available at the metadata URL.

Items	OData Entities	Description
Part	`Part`, `ElectricalPart`	The `Part` entity represents a part version. In Windchill, the `WTPart` and `WTPartMaster` classes are used to work with part versions. `ElectricalPart` is derived from `Part` and represents the soft type that is available in Windchill.
Bill of material	`Bom`, `PartUse`, `PartOccurrence`	`BOM` entity represents the part structure expanded to a certain number of levels. `PartUse` is an OData entity that represents the association between parent and child parts. It has attributes such as, quantity, unit, line number, and so on. These attributes of entity models are also available in the `WTPartUsageLink` class. The `PartOccurrence` entity represents the reference designator when a component is used multiple times in a BOM.
Part that resides in a Windchill folder	`PartContent`	This entity is derived from `FolderContent` entity that is available in the `DataAdmin` domain. The entity represents a part residing in a folder.

PTC Document Management Domain

The Document Management domain provides access to the document management capabilities of Windchill. It enables you to create documents. You can also upload and download content from documents.

The following table lists the significant OData entities available in the Document Management domain. To see all the available OData entities in the Document Management domain, please refer to its EDM available at the metadata URL.

Items	OData Entities	Description
Business document	`Document`	The `Document` entity represents a document version. In Windchill, the `WTDocument` and `WTDocumentMaster` classes are used to work with document versions.
Content file associated to a business document	`ContentItem`	The `ContentItem` entity provides a generic view of the content that is associated to a business document. More specialized entities derived from `ContentItem` are `URLData`, `ApplicationData`, `ExternalStoredData`.
A URL that is stored in a business document	`URLData`	The `URLData` entity provides a specialized view of the URL `ContentItem` that is stored in a document

Items	OData Entities	Description
Content stored in an external location	`ExternalStoredData`	The `ExternalStoredData` entity provides a specialized view of an externally stored `ContentItem`, which is stored in a document
Content stored in Windchill application	`ApplicationData`	The `ApplicationData` entity provides a specialized view of the content stored by the Windchill application.

PTC Data Administration Domain

The Data Administration domain provides access to data administration capabilities of Windchill. The domain includes entities that represent Windchill containers such as, site, organization, product, libraries, project containers, and so on. It also includes entities that represent the folder hierarchy in these containers. This domain contains an entity set called `Containers` that enables clients to read the containers available in their Windchill system.

Note

`Containers` entity set is read-only, and does not support update, delete and create operations.

The following table lists the significant OData entities available in the Data Administration domain. To see all the available OData entities in the Data Administration domain, please refer to its EDM available at the metadata URL.

Items	OData Entities	Description
Windchill container	`Container`	The `Container` entity represents a Windchill container. This entity exposes only those attributes that are common across all types of containers.
Site container	`Site`	The `Site` entity represents the site container and is derived

Items	OData Entities	Description
		from the `Container` entity.
Organization container	`OrganizationContainer`	The `OrganizationContainer` entity represents the organization container and is derived from the `Container` entity.
Product container	`ProductContainer`	The `ProductContainer` entity represents the product container and is derived from the `Container` entity.
Library container	`LibraryContainer`	The `LibraryContainer` entity represents the library container.
Project container	`ProjectContainer`	The `ProjectContainer` entity represents the project container.
Generic item that resides in the Windchill folder	`FolderContent`	The `FolderContent` entity represents the generic view of an item that resides in a folder. Other domain entities can derive from this entity to create more specific views. For example, in the Product Management domain, the `PartContent` entities derive from `FolderContent`.

PTC Principal Management Domain

The Principal Management domain provides read access to the information related to principals in Windchill. The Windchill principals can be users, groups, or organization principals.

The following table lists the significant OData entities available in the Principal Management domain. To see all the available OData entities in the Principal Management domain, please refer to its EDM available at the metadata URL.

Items	OData Entities	Description
Windchill principal	`Principal`	The `Principal` entity represents the generic view of a Windchill principal.
Windchill user	`User`	The `User` entity represents a principal who is the user. In Windchill, the `WTUser` class is used to work with users.
Windchill group	`Group`	The `Group` entity represents a principal who is the group. In Windchill, the `WTGroup` class is used to work with groups.
Windchill organization principal	`Organization`	The `Organization` entity represents a Windchill group that is an organization principal. In Windchill, the `WTOrganization` class is used to work with organization principals.

PTC Common Domain

PTC Common domain provides access to entities that are common to multiple domains. It is recommended to store common entities in this domain. The domain also provides complex types and functions that are used in other domains.

The following table lists the significant OData entities available in the PTC Common domain. To see all the available OData entities in the PTC Common domain, please refer to its EDM available at the metadata URL.

Items	OData Entities	Description
Windchill representation	`Representation`	The `Representation` entity is a lightweight representation of CAD data that is stored in Windchill, and is associated with parts and documents.

In addition to the entities, this domain also contains the following complex types:

- `QuantityOfMeasureType`—Used to represent Real number with unit data type in Windchill.
- `Hyperlink`—Used to represent a URL data type in Windchill.
- `Icon`—Used to represent an icon in Windchill.
- `EnumType`—Used to represent attributes that are enumerated types in Windchill or attributes that have type constraints defined.

This domain also provides a function `GetEnumTypeConstraint`. This function is used to query the valid values for a property, which are represented as `EnumType`. These values are used for implementing validations on client side.

Accessing Domains

This section describes how clients work with domains. The information in this section applies to all the domains installed with Windchill REST Services.

When Windchill REST Services is installed, the servlet `WcRestServlet` is enabled in the Windchill installation. All requests to the domain resources, which are enabled by Windchill REST Services, pass through this servlet. The root URL to access this servlet is `https://<Windchill server>/Windchill/servlet/odata/`. When an HTTP GET request is sent to this URL, the servlet responds back with a list of domains available on the server.

From the servlet response, clients can select a domain, and send a GET request to the root URL of the domain to get a list of available entity sets. Clients can send GET, POST, PUT, and DELETE requests to the URLs of the entity sets.

The following URLs are used to interact with Windchill REST Services:

- REST Root URL—`https://<Windchill server>/Windchill/servlet/odata/`

  A GET request to this URL lists the domains available on the Windchill server. Clients can use this URL to get the list of services that are available on a Windchill server.

  For example, the output for the following request is as shown below:

Request URL:

```
GET https://windchill.ptc.com/Windchill/servlet/odata
```

Output:

```
[
 {
 "path": "https://windchill.ptc.com/Windchill/servlet/odata/v1/PrincipalMgmt",
 "name": "PTC Principal Management Domain",
 "description": "PTC Principal Management Domain",
 "id": "PrincipalMgmt"
 },
 {
 "path": "https://windchill.ptc.com/Windchill/servlet/odata/v1/DataAdmin",
 "name": "PTC Data Administration Domain",
 "description": "PTC Data Administration domain",
 "id": "DataAdmin"
 },
 {
 "path": "https://windchill.ptc.com/Windchill/servlet/odata/v1/ProdMgmt",
 "name": "PTC Product Management Domain",
 "description": "PTC Product Management Domain",
 "id": "ProdMgmt"
 },
 {
 "path": "https://windchill.ptc.com/Windchill/servlet/odata/v1/DocMgmt",
 "name": "PTC Document Management Domain",
 "description": "PTC Document Management Domain",
 "id": "DocMgmt"
 },
 {
 "path": "https://windchill.ptc.com/Windchill/servlet/odata/v1/PTC",
 "name": "PTC Common Domain",
 "description": "PTC Common Domain",
 "id": "PTC"
 }
]
```

- Domain Root URL—`https://<Windchill server>/Windchill/servlet/odata/<Domain>`

  A GET request to this URL returns the information about the entity sets available in the domain. This request is same as that defined in OData protocol for the Service Root URL. The `<Domain>` in the URL refers to the domain identifier `id`, which is returned by the REST Root URL in the list of domains.

For example, the output for a GET request to the Domain Root URL of the `ProdMgmt` domain is as shown below. The output shows that the `ProdMgmt` domain contains entity sets such as, `Parts`, `Containers` and so on.

Request URL:

```
GET https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/
```

Output:

```
{
    "@odata.context": "$metadata",
    "value": [
        {
            "name": "Parts",
            "url": "Parts"
        },
        {
            "name": "Containers",
            "url": "Containers"
        },
        {
            "name": "Representations",
            "url": "Representations"
        },
        {
            "name": "Documents",
            "url": "Documents"
        }
    ]
}
```

- Domain Metadata URL—`https://<Windchill server>/Windchill/servlet/odata/`*`<Domain>`*`/$metadata`

  A GET request to this URL returns the entity data model for the domain defined in the Common Schema Definition Language (CSDL). In the OData protocol, this is called the Metadata Document URL.

  For example, the output from a GET request to this URL for the `PrincipalMgmt` domain is as shown below. This URL is used to get information about the entity sets, entities, and entity relations provided by the service. For more information on entity sets, entities, and entity relations, please refer to the OData protocol documentation.

  Request URL:

```
GET https://windchill.ptc.com/Windchill/servlet/odata/PrincipalMgmt/$metadata
```

Output:

```
<?xml version='1.0' encoding='UTF-8'?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
 <edmx:Reference Uri="https://windchill.ptc.com/Windchill/servlet/odata/v1/PTC">
        <edmx:Include Namespace="PTC"/>
    </edmx:Reference>
    <edmx:DataServices>
        <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="PTC.PrincipalMgmt">
            <EntityType Name="Group" BaseType="PTC.PrincipalMgmt.Principal">
                <Property Name="Description" Type="Edm.String">
                    <Annotation Term="PTC.ReadOnly"/>
                </Property>
                <Property Name="DomainName" Type="Edm.String">
                    <Annotation Term="PTC.ReadOnly"/>
                </Property>
                <Annotation Term="Core.Description">
                    <String>Groups</String>
                </Annotation>
                <Annotation Term="PTC.Operations">
                    <String>READ</String>
                </Annotation>
            </EntityType>
            <EntityType Name="User" BaseType="PTC.PrincipalMgmt.Principal">
                <Property Name="LastName" Type="Edm.String"/>
                <Property Name="FullName" Type="Edm.String"/>
                <Property Name="EMail" Type="Edm.String"/>
                <Property Name="UserDomain" Type="Edm.String"/>
                <Annotation Term="Core.Description">
                    <String>Users</String>
                </Annotation>
                <Annotation Term="PTC.Operations">
                    <String>READ</String>
                </Annotation>
            </EntityType>
<ComplexType Name="OrgId">
                <Property Name="CodingSystem" Type="Edm.String"/>
                <Property Name="UniqueIdentifier" Type="Edm.String"/>
                <Annotation Term="Core.Description">
                    <String>Organization identifier</String>
                </Annotation>
            </ComplexType>
            <EntityContainer Name="Windchill">
                <EntitySet Name="Representations" EntityType="PTC.Representation"/>
                <EntitySet Name="Groups" EntityType="PTC.PrincipalMgmt.Group"/>
                <EntitySet Name="Users" EntityType="PTC.PrincipalMgmt.User"/>
                <EntitySet Name="Principals" EntityType="PTC.PrincipalMgmt.Principal"/>
            </EntityContainer>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

Entity Set URL—`https://<Windchill server>/Windchill/servlet/odata/`*`<Domain>`*`/`*`<EntitySetURL>`*

An Entity Set URL references an entity set, which is available in the response of a domain, to a GET request by the Domain Root URL.

In the Domain Root URL example above, you can see that there is an entity set named `Parts` that also has a `url` for `Parts`. The Entity Set URL is `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts`. A GET request to this URL returns a set of entities as shown below:

Request URL:

```
GET https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts
```

Output:

```
{
    "@odata.context": "https://windchill.ptc.com/Windchill/servlet/odata/v1/ProdMgmt/
       $metadata#Parts",
    "value": [
        {
            "ID": "OR:wt.part.WTPart:62850",
            "Name": "LOWER_SUPPORT",
            "Number": "GC000019",
            "EndItem": false,
            "TypeIcon": {
                "Path": "https://windchill.ptc.com/Windchill/wtcore/images/part.gif",
                "Tooltip": "Part"
            },
            "Identity": "GC000019, LOWER_SUPPORT, A (Design)",
            "GeneralStatus": null,
            "ShareStatus": null,
            "ChangeStatus": null,
            "Superseded": null,
            "AssemblyMode": {
                "Value": "separable",
                "Display": "Separable"
            },
            "DefaultUnit": "ea",
            "DefaultTraceCode": "0",
            "Source": "make",
            "ConfigurableModule": "standard",
            "GatheringPart": false,
            "PhantomManufacturingPart": false,
            "OwningDesignCenter": null,
            "OwningBusinessUnit": null,
            "view": "Design",
            "CheckoutState": "Checked in",
            "Comments": null,
            "State": {
                "Value": "INWORK",
                "Display": "In Work"
            },
            "LifeCycleTemplateName": "Basic",
            "VersionID": "VR:wt.part.WTPart:62849",
            "Revision": "A",
            "Version": "A.1 (Design)",
            "Latest": true,
            "CreatedOn": "2017-04-08T03:47:23Z",
            "LastModified": "2017-04-08T03:47:23Z"
        }
]
}
```

Entity Set URL is the main endpoint to perform create, read, update, and delete operations using the HTTP requests POST, GET, PATCH and DELETE respectively.

Let us continue using the above example. To create a part the client sends a POST request to the URL `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts`. The body of the request contains a set of property names and values specified in a format that is acceptable to the server. Some of the acceptable formats are JSON, XML, and so on.

To update the part, clients send a PATCH request on the same URL. The body of the PATCH request contains a set of property names and values that will be modified.

To delete a part, clients send a DELETE request to the URL `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts('<key>')`. In this URL, `key` is the unique identifier for the part in the entity set. The object reference string in Windchill is treated as the `key`. To delete a part with object reference ‘`OR:wt.part.WTPart:668899`’, the DELETE request is `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts(‘OR:wt.part.WTPart:668899’).`

Clients usually interact with the REST APIs using the Entity Set URL. All the entities in a domain may not have entity sets. Therefore, some entities in the domain are available using navigations. For example, a specific `PartUse` entity in the ProdMgmt domain is accessed by `https://windchill.ptc.com/Windchill/servlet/odata/ProdMgmt/Parts(<part_key>)/Uses(<uses_key>)`. Here `<part_key>` and `<uses_key>` are object reference strings that uniquely identify a part and a usage link.

Examples for Performing Basic REST Operations

Fetching a NONCE Token from a Service

This example shows you how to fetch a NONCE token from a service. Use the following GET request.

URI

```
GET /Windchill/servlet/odata/ HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: Fetch
```

The NONCE token is returned in the response header `CSRF_NONCE`. The value of `CSRF_NONCE` returned from this request must be passed as request header in all the examples provided in this User’s Guide to create (POST requests), modify (PUT and PATCH requests), or delete (DELETE request) entities.

Creating a Part

This example shows you how to create a part. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/ProdMgmt/Parts HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"Name":"TestWTPart_001",
"DefaultUnit" : "ea",
"AssemblyMode": {
  "Value": "separable",
  "Display": "Separable"
 },

"DefaultTraceCode": "0",
"Source": "make",
"PhantomManufacturingPart" : false,

"Context@odata.bind": "Containers('OR:wt.pdmlink.PDMLinkProduct:48507000')"
}
```

Create a Part Usage Link with Occurrences

This example shows you how to create a part usage link with occurrences. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/ProdMgmt/Parts('VR:wt.part.WTPart:48796525')/Uses HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
 "Quantity" : 2,
 "Unit" : {
"Value": "ea",
    "Display": "Each"
  },
 "FindNumber" : "100",
 "LineNumber" : 100,
 "TraceCode": {
"Value": "0",
    "Display": "Untraced"
  },
 "Uses@odata.bind" : "Parts('OR:wt.part.WTPart:48796415')",
   "Occurrences": [
        {
            "ReferenceDesignator": "R1",
            "Location": {
                "PointX": 0,
                "PointY": 1,
                "PointZ": 1,
                "PointUnit": "m",
                "AngleX": 1.04,
                "AngleY": 1.04,
                "AngleZ": 1.04,
                "AngleUnit": "r"
            }
        },
        {
            "ReferenceDesignator": "R2",
            "Location":  {
                "PointX": 1,
                "PointY": 1,
                "PointZ": 0,
                "PointUnit": "m",
                "AngleX": 3.14,
                "AngleY": 3.14,
                "AngleZ": 3.14,
                "AngleUnit": "r"
            }
        }
    ]

}
```

Deleting a Part Usage Link

This example shows you how to delete a part usage link. Use the following DELETE request.

URI

```
DELETE /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:48796526')/
                 Uses('OR:wt.part.WTPartUsageLink:48796528') HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Reading the Bill of Material (BOM)

This example shows you how to read the bill of material (BOM) for a product structure. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:44148884')/
   PTC.ProdMgmt.GetBOM?$expand=Components($expand=Part($select=Name,Number),
   PartUse,Occurrences;$levels=max) HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"navigationCriteriaId" : "OR:wt.filter.NavigationCriteria:48796407"
}
```

Querying the Part Using a Filter

This example shows you how to query a part using a filter. Use the following GET request.

URI for Filter Based on Soft Attribute

```
GET /Windchill/servlet/odata/v1/ProdMgmt/Parts?$filter=contains(CustomAttribute,'value') HTTP/1.1
```

URI for Filter Based on Part Name

```
GET /Windchill/servlet/odata/v1/ProdMgmt/Parts?$filter=Name eq 'TestWTPart_001' HTTP/1.1
```

Reading a Part by ID with Expanded Navigation

This example shows you how to read a part with its ID with expanded navigation. Use the following GET request.

URI for Part Uses Link with Expand Filter

```
GET /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:48796184')?$expand=Uses HTTP/1.1
```

URI for Part Uses Link and Its Occurrences with Expand Filter

```
GET /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:48796184')
?$expand=Uses($expand=Occurrences) HTTP/1.1
```

Checking Out a Part

This example shows you how to check out a part. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:48796184')/PTC.CheckOut HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"CheckOutNote" : "This is checkout note."
}
```

Checking In a Part

This example shows you how to check in a part. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/ProdMgmt/Parts('OR:wt.part.WTPart:48796184')/PTC.CheckIn HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"CheckInNote" : "This is checkin note."
}
```

Creating a Document

This example shows you how to create a document. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/DocMgmt/Documents HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"Name": "TestDoc1",
"Description": "TestDoc1_Description",
"Title": "TestDoc1_Title",
"Context@odata.bind": "Containers('OR:wt.pdmlink.PDMLinkProduct:48788507')"
}
```

Checking Out a Document

This example shows you how to check out a document. Use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/DocMgmt/Documents HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
"CheckOutNote" : "This is checkout note."
}
```

Updating a Document

This example shows you how to update a document. Use the following PATCH URI with the request body.

URI

```
PATCH /Windchill/servlet/odata/v1/DocMgmt/Documents('VR:wt.doc.WTDocument:48796553') HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
```

```
"Description": "TestDoc1_Description_Update",
"CustomAttribute"  : "This is Test Attribute"
}
```

Uploading Content for a Document

This example shows you how to upload content for a document in the following cases:

- Using a local file
- Using URL data
- Using external data

Use the following POST URI with the request body.

Using a Local File

The content can be uploaded in the following stages:

- Stage1—POST URI

  ```
  POST /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:48796581')/
      PTC.DocMgmt.uploadStage1Action HTTP/1.1
  ```

  Stage 1—Request Headers

  ```
  Content-Type: application/json
  CSRF_NONCE: <Use the value from Fetch NONCE example>
  ```

  Stage1—Request Body

  ```
  {
  "noOfFiles":3
  }
  ```

  Stage1—Sample Output

  ```
  {
   "@odata.context": "$metadata#CacheDescriptor",
   "value": [
    {
     "ID": null,
     "ReplicaUrl": "https://i7752.ptcnet.ptc.com:9090/Windchill/servlet/WindchillGW/
      wt.fv.uploadtocache.DoUploadToCache_Server/doUploadToChache_Master?mk=
      wt.fv.uploadtocache.DoUploadToCache_Server&VaultId=150301&FolderId=150329&
      CheckSum=456186&sT=1507542170&sign=Ca4ouGGOZiopnqbd4mbUVg%3D%3D&site=
      http%3A%2F%2Fi7752.ptcnet.ptc.com%3A9090%2FWindchill%2Fservlet%2F
      WindchillGW&AUTH_CODE=HmacMD5&isProxy=true&delegate=
      wt.fv.uploadtocache.DefaultRestFormGeneratorDelegate",
     "MasterUrl": "https://i7752.ptcnet.ptc.com:9090/Windchill/servlet/WindchillGW",
     "VaultId": 150301,
  ```

```
   "FolderId": 150329,
   "StreamIds": [
                76030,
                76032,
                76031
      ],
   "FileNames": [
                76030,
                76032,
                76031
            ]
        }
]
}
```

- Stage2—The HTTP request for Stage2 must be constructed from `ReplicaUrl` attribute which is retrieved from Stage1.

  Stage2—POST URI

  ```
  https://i7752.ptcnet.ptc.com:9090/Windchill/servlet/WindchillGW/
    wt.fv.uploadtocache.DoUploadToCache_Server/doUploadToChache_Master?
    mk=wt.fv.uploadtocache.DoUploadToCache_Server&VaultId=150301
    &FolderId=150329&CheckSum=456186&sT=1507542170&sign=
    Ca4ouGGOZiopnqbd4mbUVg%3D%3D&site=http%3A%2F%2F
    i7752.ptcnet.ptc.com%3A9090%2FWindchill%2Fservlet%2F
  WindchillGW&AUTH_CODE=HmacMD5&isProxy=true&delegate=
  wt.fv.uploadtocache.DefaultRestFormGeneratorDelegate
  ```

  Stage 2—Request Headers

  ```
  Content-Type: application/json
  CSRF_NONCE: <Use the value from Fetch NONCE example>
  ```

  Stage2—Request Body

  ```
  -----------------------------boundary
  Content-Disposition: form-data; name="Master_URL"
  http://i7752.ptcnet.ptc.com:9090/Windchill/servlet/WindchillGW
  -----------------------------boundary
  Content-Disposition: form-data; name="CacheDescriptor_array"
  76030: 76030: 76030: 3743; 76032: 76032: 76032: 2735; 76031: 76031: 76031:
  2735;.....
  -----------------------------boundary
  Content-Disposition: form-data; name="76030"; filename="TestFile1.txt"
  This is content of test file 1.
  -----------------------------boundary
  ```

```
Content-Disposition: form-data; name="76032"; filename="TestFile2.txt"
This is content of test file 2.
-----------------------------boundary
Content-Disposition: form-data; name="76031"; filename="TestFile3.txt"
This is content of test file 2.
-----------------------------boundary
```

Note

The `CacheDescriptor_array` contains the following information `<streamid>:<filename>:<contentid>:<filesize>` where,

- `streamid`—Specifies the unique content ID from the Stage1 response.
- `filename`—Specifies the name of the file from the Stage1 response.
- `contentid`—Same as `streamid`.
- `filesize`—Specifies size of the file to be uploaded in bytes (Optional).

The response from Stage2 contains information about the `streamId`, size of the file created, and encoded `CachedContentDescriptor`, which is used in Stage3 for uploading content to the document.

Stage2—Sample Output

```
{
"contentInfos": [
{
"streamId": 76035,
"fileSize": 2,
"encodedInfo": "76035%3A2%3A150329%3A76035"
},
{
"streamId": 76034,
"fileSize": 2,
"encodedInfo": "76034%3A2%3A150329%3A76034"
},
{
"streamId": 76033,
"fileSize": 2,
"encodedInfo": "76033%3A2%3A150329%3A76033"
}
]
}
```

- Stage3—POST URI

  ```
  POST /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:48796581')/
       PTC.DocMgmt.uploadStage3Action HTTP/1.1
  ```

  Stage 3—Request Headers

  ```
  Content-Type: application/json
  CSRF_NONCE: <Use the value from Fetch NONCE example>
  ```

  Stage3—Request body

  ```
  {
  "contentInfo" : [
  {
  "StreamId" :76033,
  "EncodedInfo" : "76033%3A2%3A150329%3A76033",
  "FileName" : "DesignSpec.doc",
  "PrimaryContent" : true,
  "MimeType" : "application/vnd.openxmlformats-officedocument.wordprocessingml.
  document",
  "FileSize" : 2
  },
  {
  "StreamId" :76035,
  "EncodedInfo" : "76035%3A2%3A150329%3A76035",
  "FileName" : "ReferenceDoc1.doc",
  "PrimaryContent" : false,
  "MimeType" : "application/vnd.openxmlformats-officedocument.wordprocessingml.
  document",
  "FileSize" : 2
  },
  {
  "StreamId" :76034,
  "EncodedInfo" : "76034%3A2%3A150329%3A76034",
  "FileName" : "ReferenceDoc2.doc",
  "PrimaryContent" : false,
  "MimeType" : "application/vnd.openxmlformats-officedocument.wordprocessingml.
  document",
  "FileSize" : 2
  }
  ]
  }
  ```

Using a URL Data

To create or update primary content from URL data, use the following PUT URI with the request body.

URI

```
PUT /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:2626068')/
    PrimaryContent HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
   "UrlLocation" :"https://www.ptc.com",
   "DisplayName" : "Test_PrimaryContent"
}
```

Using External Storage

To create or update the primary content from external storage, use the following PUT URI with the request body.

URI

```
PUT /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:2626068')/
   PrimaryContent HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
   "ExternalLocation" :"TestExternalLocation",
   "DisplayName" : "TestExternalLocation_DisplayName"
}
```

To create new attachments, use the following POST URI with the request body.

URI

```
POST /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:2626099')/
   Attachments HTTP/1.1
```

Request Headers

```
Content-Type: application/json
CSRF_NONCE: <Use the value from Fetch NONCE example>
```

Request Body

```
{
   "ExternalLocation" :"TestExternalLocation",
   "DisplayName" : "TestExternalLocation"
}
```

To update existing attachments, use the following PUT URI with the request body.

URI

```
PUT /Windchill/servlet/odata/v1/DocMgmt/Documents('OR:wt.doc.WTDocument:2626099')/
     Attachments('OR:wt.content.ExternalStoredData:2626811') HTTP/1.1
```

Request Body

```
{
   "ExternalLocation" :"TestExternalLocation_Update",
   "DisplayName" : "TestExternalLocation_Update"
}
```

Customizing Domains

Windchill REST Services enables you to customize domains provided by PTC. You can also add new domains. For customizing existing domains and creating new domains, new configuration files must be created in the custom configuration path `<Windchill>/codebase/rest/custom/domain`. The configurations from the custom configuration path are merged with the customizations in the PTC configuration path `<Windchill>/codebase/rest/ptc/domain`.

Note

Changes made in the PTC configuration path are not supported. The changes made here will be overwritten in the next update.

Extending Domains

To extend a PTC domain, mirror the domain folder structure from the PTC configuration path to the custom configuration path. Only the folder structure is mirrored. Copies of `.json` or `.js` files are not created. After the folder structure is mirrored, customizers decide what do they want to extend in the PTC domain, and create the required `.json` and `.js` files to extend domain definitions.

Windchill REST Services supports the following types of domain extensions:

- Adding type extensions of Windchill types to PTC Domains
- Adding custom properties to entities in PTC Domains
- Adding custom navigation between entities in PTC Domains
- Adding new functions to PTC Domains
- Adding new actions to PTC Domains

Adding Type Extensions of Windchill Types to PTC Domains

A PTC domain can be extended to add an OData entity that corresponds to a custom soft type created in Windchill. Customizers often create a custom soft type extension in Windchill to add a new behavior to Windchill. For example, consider the case where customizers have created a subclass `WTPart`. A soft type `com.custom.PurchasePart` is created for `WTPart`. Further an additional string attribute called `SupplierName` on `PurchasePart` is also added.

To enable this soft type in Product Management domain, customizers first mirror the ProdMgmt domain folder structure in the custom configuration path. Then, create the `PurchasePart.json` file. Perform the following steps to enable a soft type:

1. In the custom configuration path, create the following folder structure for the Product Management domain at `<Windchill>/codebase/rest/custom/domain/`:
   - `ProdMgmt`
     - `v1`
       - `entity`
2. Create the `PurchaseParts.json` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and add the following content in the file:

```
{
   "name": "PurchasePart",
   “type”: “wcType”,
   “wcType”: “com.custom.PurchasePart”,
   “collectionName”: “PurchaseParts”,
   “includeInServiceDocument”: “false”,
   "parent": {
       “name”: "Part"
   },
   "attributes": [
       {
        "name": "SupplierName",
        "internalName": "SupplierName",
        "type": "String"
       }
   ]
}
```

After the configuration, when you visit the metadata URL for Product Management domain, the new entity `PurchasePart`, which is derived from the part entity is available. The `PurchasePart` entity also has the `SupplierName` property. Since the `PurchasePart` entity is now in the EDM, standard OData URLs can be used to access `PurchasePart`.

Adding Custom Properties to Entities in PTC Domains

A PTC domain can be extended to have custom properties which have been added to Windchill types by customizers. Customizers add new properties for Windchill types such as, `WTParts`, `WTDocuments`, and so on. `WTParts` and `WTDocuments` are available as Part and Document entities in the Product Management and Document Management domains respectively. You can add new attributes as properties for these entities. To add `OwningBusinessUnit` and `DesignCost` attributes to the Part entity from the Product Management domain, the customizers mirror the ProdMgmt domain folder structure in the custom configuration path. Then, create the `PartsExt.json` file to add custom configuration. In the JSON file, under `extends` property, add the PTC domain entity which you want to extend. In the `attributes` property, add the new attributes. Perform the following steps to add custom properties to entities:

1. In the custom configuration path, create the following folder structure for the Product Management domain at `<Windchill>/codebase/rest/custom/domain/`:
   - `ProdMgmt`
     - `v1`
       - `entity`
2. Create the `PartsExt.json` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and add the following content in the file:

```
{
   "extends": "Parts",
   "attributes": [
        {
         "name": "OwningBusinessUnit",
         "internalName": "OwningBusinessUnit",
         "type": "String"
        },
      {
         "name": "DesignCost",
         "internalName": "DesignCost",
         "type": "Double"
        }
   ]
}
```

After the configuration, when you visit the metadata URL for Product Management domain, it shows the new properties `OwningBusinessUnit` and `DesignCost` on the Part entity for ProdMgmt domain. Since the Part entity has additional attributes, they can be used in standard OData URLs.

Adding Custom Navigation Between Entities in PTC Domains

A PTC domain can be extended to have new navigation between entities in a PTC domain. For example, the Product Management domain, does not provide any navigation between Part entities to show parts that are alternates of each other. To provide this navigation, customizers must extend the Product Management domain. To add `Alternates` navigation between Part entities in the Product Management domain, the customizers mirror the ProdMgmt domain folder structure in the custom configuration path. Then create the `PartsExt.json` file to add custom configuration. In the JSON file, under `extends` property, add the PTC domain entity which you want to extend. In the `navigations` property, add the new navigation. Apart from providing the configurations in the `.json` file, customizers must also provide the programming logic to create the target entity set while navigating from the source to target entity. This is done in the `.js` file corresponding to the entity, in this case, `PartsExt.js` file. Perform the following steps to add custom navigation between entities:

1. In the custom configuration path, create the following folder structure for the Product Management domain at `<Windchill>/codebase/rest/custom/domain/`:
   - `ProdMgmt`
     - `v1`
       - `entity`
2. Create the `PartsExt.json` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and add the following content in the file:

```
{
   "extends": "Parts",
   "navigations": [
       {
        "name": "Alternates",
        "target": "Parts",
        "type": "Part",
               "isCollection": true,
               "containsTarget": true,
       }
   ]
}
```

3. Create the `PartsExt.js` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and implement the following hooks:
   - `getRelatedEntityCollection`—The hook returns the following information:
     a. Gets the alternate part entities from the source entities.
     b. Puts the alternate part entities in an entity collection.
     c. Returns the entity collection in a map.
   - `isValidNavigation`—The hook returns the following information:
     a. Checks if the navigation being carried out is Alternates. If not, it returns null so that the framework can continue processing other navigations.
     b. Gets the source part.
     c. Navigates to the target part.
     d. Verifies that the target part is the same as specified in the input.
     e. Returns true or false to indicate the success of the validation.

Many of the hooks have been implemented in PTC provided domains. For code examples of hook implementations, you can see their implementations in any of PTC provided domains.

After the configuration, when you visit the metadata URL for Product Management domain, the `Alternates` navigation is available for the Part entity. You can navigate from a part to get its alternate parts.

Adding New Functions to PTC Domains

A PTC domain can be extended to add both bound and unbound OData functions. OData functions appear in the EDM of a domain. They are invoked with a GET request to the Odata URL of the function. For example, consider a case where you want to add a bound function to the Product Management domain that identifies costly parts within an entity set Parts. Perform the following steps to add a bound function:

1. In the custom configuration path, create the following folder structure for the Product Management domain at `<Windchill>/codebase/rest/custom/domain/`:
   - `ProdMgmt`
     - `v1`

- `entity`

2. Create the `PartsExt.json` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and add the following content in the file:

```
{
   "extends": "Parts",
   "functions": [
       {
          "name": "GetCostlyParts",
           "description": "Return expensive parts",
           "isComposable": false,
             "parameters": [
                 "name": "PartSet",
                 "type": "Part",
                 "isCollection": true,
                 "isNullable": false
             ],
             "returnType": {
                 "type": "Part",
                 "isCollection": true
            }
   ]
}
```

3. Create the `PartsExt.js` file at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity` and implement the function. Ensure that the Part entity has a numeric property `DevelopmentCost`.

```
function function_GetCostlyParts(data, params) {
    var ArrayList = Java.type('java.util.ArrayList');
    var EntityCollection = Java.type('org.apache.olingo.commons.api.data.EntityCollection');
    var parts = data.getProcessor().readEntitySetData(data);
    var partEntityMap = data.getProcessor().toEntities(parts, data);
    var partEntities = new ArrayList(partEntityMap.values());

    var entityCollection = new EntityCollection();
    for(var i = 0; i < partEntities.size(); i++) {
        var partEntity = partEntities.get(i);
        var partCostProperty = partEntity.getProperty('DevelopmentCost');
        if(partCostProperty) {
            var partCost = partCostProperty.getValue();
            if(partCost && partCost > 0.10) {
                entityCollection.getEntities().add(partEntity);
            }
        }
    }
```

```
        return entityCollection;
    }
```

After the configuration, when you visit the metadata URL for Product Management domain, the `GetCostlyParts` function is available for the Part entity. You can call the function on the Parts entity set and get a list of the costly parts.

Adding New Actions to PTC Domains

OData actions change the state of the entities and are called with a POST request. These are the basic differences between actions and functions.

In terms of definition, actions are similar to functions. However, there are some differences in definition between actions and functions:

- Actions are defined in the `actions` property of imports and entity JSON files.
- Actions are named with a prefix of `action_`.

Creating New Domains

Windchill REST Services enables you to create new domains. The new domains are created in the custom configuration path.

To create a new domain, perform the following steps:

1. Decide a domain identifier and the domain version. Create the domain folder `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>`
2. Create the `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>.json` file and provide values for domain metadata attributes.
3. Decide which other domains to import and set up the `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>/import.json` file.
4. Decide if the domain must have unbound actions or functions and set up the `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>/import.json` and `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>/import.js` files.
5. If `complexTypes` are required then set up the complex type JSON files at `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>/complexType`.
6. Configure entities and entity relations at `<Windchill>/codebase/rest/custom/domain/<Domain Identifier>/<Domain Version>/entity`.

After these files are setup, the domain is available at the REST root URL and can be accessed by OData URLs.

These are generic instructions to create a domain. You have to create and configure files depending on the entities of the domain. In this User's Guide, we have provided an example, that shows how to create a domain. The example helps you understand which files to create while configuring a domain.

Examples for Customizing Domains

Creating a New Domain

This example shows you how to create a new domain.

Consider an example, where a new domain Reporting must be created for building a reporting application. The Windchill types, `WTChangeIssue` and `Changeable2` must be exposed as `ProblemReport` and `ChangeableItem` entities respectively. Further, the `ReportedAgainst` relationship between `ProblemReport` and `ChangeItem` entities must also be exposed. The version `v1` must also be set up for the Reporting domain. For the reporting purpose, information can only be read from Windchill. The following properties of the two entities of the domain are exposed:

- ProblemReport
  - Number
  - Name
  - Occurrence date
  - Need date
  - Priority
  - Category
  - State

- ChangeableItem
  - Number
  - Name
  - Revision
  - State

To configure a domain for all the criteria mentioned in the example, perform the following steps:

1. Create the folder `<Windchill>/codebase/rest/custom/domain/Reporting`.
2. Create the file `<Windchill>/codebase/rest/custom/domain/Reporting.json` with the following content:

```
{
  "name": "Reporting",
  "id": "Reporting",
  "description": "Reporting Domain",
  "nameSpace": "Custom.Reporting",
  "containerName": "Windchill",
  "defaultVersion": "1"
   }
```

3. Create the folder `<Windchill>/codebase/rest/custom/domain/Reporting/v1`.
4. Create the file `<Windchill>/codebase/rest/custom/domain/Reporting/v1/import.json` with the following content:

```
{
   "imports": [
       {"name": "PTC", "version": "1"}
   ]
    }
```

5. Create the folder `<Windchill>/codebase/rest/custom/domain/Reporting/v1/entity`.
6. Create the file `<Windchill>/codebase/rest/custom/domain/Reporting/v1/entity/ChangeableItems.json` with the following content:

```
{
  "name": "ChangeableItem",
  "collectionName": "ChangeableItems",
  "type": "wcType",
  "wcType": "wt.change2.Changeable2",
  "description": "Changeable Item",
  "operations": "READ",
  "attributes": [
    {"name": "Name", "internalName": "name", "type": "String"},
    {"name": "Number", "internalName": "number", "type": "String"}
  ],
  "inherits": [
    {"name": "lifecycleManaged"},
    {"name": "versioned"}
  ]
}
```

7. Create the file `<Windchill>/codebase/rest/custom/domain/Reporting/v1/entity/ProblemReports.json` with the following content:

```
{
 "name": "ProblemReport",
 "collectionName": "ProblemReports",
 "type": "wcType",
 "wcType": "wt.change2.WTChangeIssue",
 "description": "Problem Report",
 "operations": "READ",
 "attributes": [
    {"name": "Name", "internalName": "name", "type": "String"},
    {"name": "Number", "internalName": "number", "type": "String"},
    {"name": "Priority", "internalName": "theIssuePriority", "type": "String"},
    {"name": "Category", "internalName": "theCategory", "type": "String"},
    {"name": "OccurrenceDate", "internalName": "occurrenceDate", "type": "DateTimeOffset"},
    {"name": "NeedDate", "internalName": "needDate", "type": "DateTimeOffset"}
  ],
 "navigations": [
 {"name": "ReportedAgainst", "target": "ChangeableItems", "type": "ChangeableItem",
    "containsTarget": true, "isCollection": true}
 ],
  "inherits": [
    {"name": "lifecycleManaged"}
  ]
}
```

8. Create the file `<Windchill>/codebase/rest/custom/domain/Reporting/v1/entity/ProblemReports.js` with the following content:

```
function getRelatedEntityCollection(navProcessorData) {
  var HashMap = Java.type('java.util.HashMap');
  var ArrayList = Java.type('java.util.ArrayList');
  var WTArrayList = Java.type('wt.fc.collections.WTArrayList');
  var ChangeHelper2 = Java.type('wt.change2.ChangeHelper2');
  var targetName = navProcessorData.getTargetSetName();
  var map = new HashMap();
  var sourcePRs = new WTArrayList(navProcessorData.getSourceObjects());
  if("ReportedAgainst".equals(targetName)) {
    for(var i = 0; i < sourcePRs.size(); i++) {
      var sourcePR = sourcePRs.getPersistable(i);
      var reportedAgainstItems = ChangeHelper2.service.getChangeables(sourcePR, true);
      var list = new ArrayList();
      while(reportedAgainstItems.hasMoreElements()) {
        list.add(reportedAgainstItems.nextElement());
      }
      map.put(sourcePR, list);
    }
  }
  return map;
}
```

This creates a new domain called Reporting with all the entities and relationships described in the example. To test the domain, use the following URLs:

- To see the EDM for the Reporting domain, use the URL:

  `https://<Windchill server>/Windchill/servlet/odata/Reporting/$metadata`

- To see the list of `ProblemReports`, use the URL:

  `https://<Windchill server>/Windchill/servlet/odata/Reporting/ProblemReports`
- To see the list of `ProblemReport` with `ChangeableItems`, use the URL:

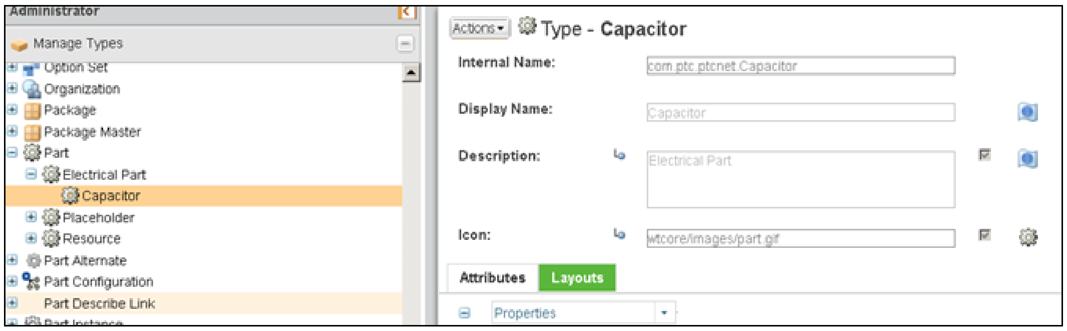
  `https://<Windchill server>/Windchill/servlet/odata/Reporting/ProblemReports?$expand=ReportedAgainst`

Extending Product Management Domain to Add A Soft Type

This example shows you how to extend the Product Management domain to add a soft type of an existing part. Consider a case where you want to create `WTPart` of soft type `Capacitor` which has its parent soft type as `Electrical Part`.

To extend the domain to add the soft type, create a custom configuration file `Capacitors.json` at `<Windchill>/codebase/rest/custom/domain/ProdMgmt/v1/entity`.

`wcType` property must have the same **Internal Name** as defined for the soft type in **Type Management**.



Type - Capacitor

Internal Name: com.ptc.ptcnet.Capacitor

Display Name: Capacitor

Description: Electrical Part

Icon: wtcore/images/part.gif

```
{
    "name": "Capacitor",
    "collectionName": "Capacitors",
    "wcType": "com.ptc.ptcnet.Capacitor",
    "description": "This part extends ElectricalParts entity.",
    "parent": {
        "name": "ElectricalParts"
    },
    "attributes": [
{
     "name":"Capacitance",
     "internalName":"Capacitance",
     "type":"String"
}


    ]
}

Sample request to create WTPart with soft type ‘Capacitor’:
{
"@odata.type": "PTC.ProdMgmt.Capacitor",
"Name":"TestWTPart_002",
"Number":"TestWTPart_002",

"DefaultUnit" : "ea",
"AssemblyMode": {
      "Value": "component",
      "Display": "Component"
     },
    "Source": "buy",
    "PhantomManufacturingPart" : false,
    "Context@odata.bind": "Containers('OR:wt.pdmlink.PDMLinkProduct:48788507')"
}
```

Extending Document Management Domain to Add a Soft Attribute

This example shows you how to extend the Document Management domain to add a soft attribute on a `WTDocument` soft type.

To extend the domain to add the soft attribute, create a custom configuration file `DocumentsExt.json` at `<Windchill>/codebase/rest/custom/domain/DocMgmt/v1/entity`.

```
{
    "extends”: "Document",
    "description”: "This config extends Documents.json.”,
    "attributes”: [
{
         "name":"ODATASTR1",
         "internalName":"ODATASTR1",
         "type":"String"
        },
        {
  "name":"ODATAINT1",
  "internalName":"ODATAINT1",
  "type":"Int16"
        },
        {
         "name":"ODATAFPN1",
         "internalName":"ODATAFPN1",
         "type":"Double"
        },
        {
         "name":"ODATABOOL1",
         "internalName":"ODATABOOL1",
         "type":"Boolean"
        },
        {
         "name":"ODATADATE1",
         "internalName":"ODATADATE1",
         "type":"DateTimeOffset"
        }

    ]

}
```

To create a `WTDocument` with these extended soft attributes use the following request:

```
POST /Windchill/servlet/odata/v1/DocMgmt/Documents HTTP/1.1
{
    "Name": "Test1",
    "Description": "Test1_Desc",
    "Title": "Test1_Title",

    "ODATASTR1": "This is String attribute",
    "ODATAINT1": 1,
    "ODATAFPN1": 1.555,
    "ODATABOOL1": true,
    "ODATADATE1": "2017-10-09T09:42:39Z",
"Context@odata.bind":
"Containers('OR:wt.pdmlink.PDMLinkProduct:48788507')"

}
```

Index