tekom EUROPE

Anne Tarnoruder

Standards and Guidelines for API Documentation

For Technical Writers, Software Developers, Information and Software Architects

Practical Guides

1 Introduction

The purpose of standards and guidelines (S&G) for API documentation is to help achieve consistency of content and style across various types and areas of the API documentation.

API Documentation Deliverables

API documentation deliverables fall into two complementary categories: API reference documentation and developer guides.

Documentation Deliverable	Description
API reference	Contains detailed reference information about all elements of APIs. Created and maintained by developers in software source code, reviewed by technical writers. Written in structured mode. Auto-generated from source code and integrated with the relevant developer guide or delivered separately. If auto-generation option is not available or not applicable, written manually by technical writers in the documentation system as part of developer guides.
Developer guide	Explains how to use the APIs. Contains concepts, diagrams, setup information, tutorials, tasks, code samples, and more. Created and maintained by technical writers in cooperation with developers in the documentation system. Written in freestyle mode. Delivered as part of the product documentation.

Scope and Target Audience

Standards and guidelines defined in this document:

– Apply mostly to API reference documentation and define the writing style and formatting rules for this documentation.
– Encompass major development languages and technologies, such as Java, JavaScript, Microsoft.NET, C/C++, REST and OData.
– Are targeted at technical writers and software developers who co-author API reference documentation either in source code or in the documentation system.

The following topics are out of scope of this document:

– API design principles and guidelines.
– Actual auto-generation and production of API documentation.

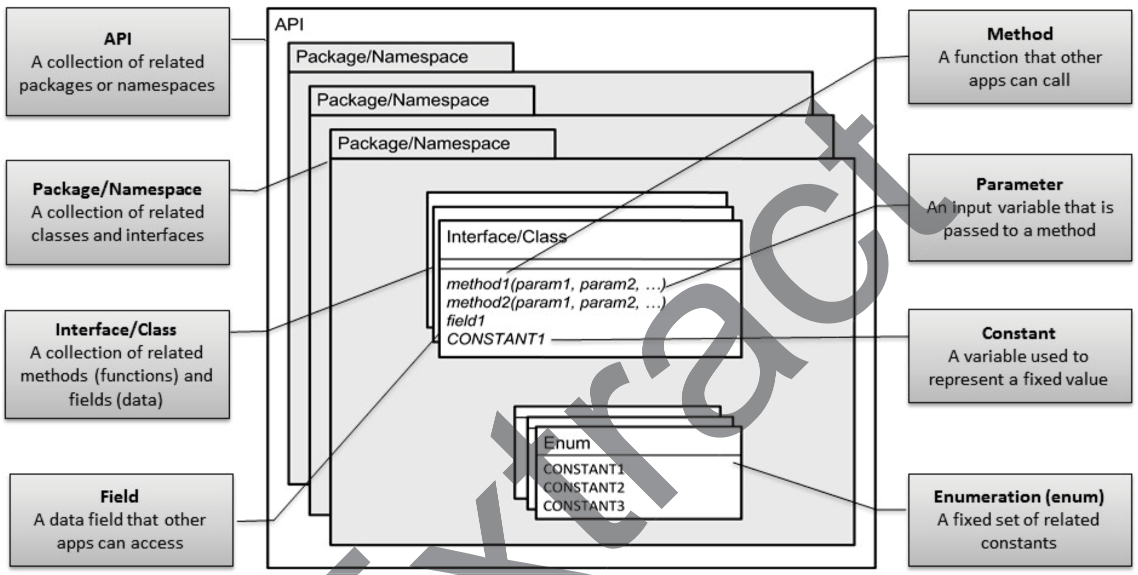
2 Terms and Concepts

Introduces the main concepts and terminology of APIs and API documentation.

Introduction to APIs

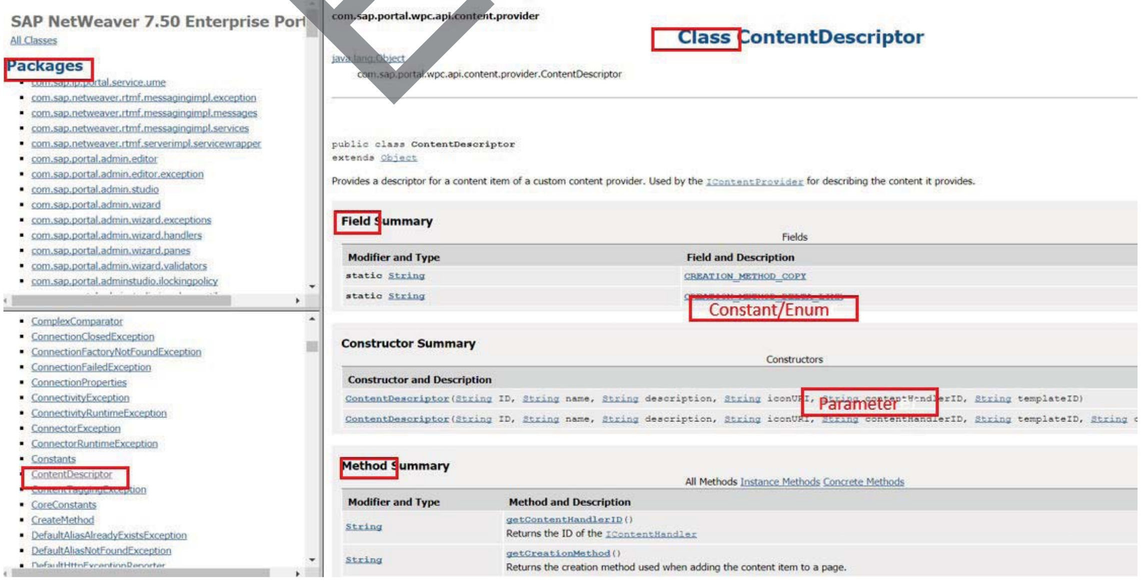
API (Application Programming Interface) is an interface provided by an application for interacting with other applications. Essentially, an API comprises functions that other applications can call and data they can access.

The following figure illustrates the basic structure of an API in object-oriented languages such as Java, JavaScript, .NET, or C/C++, which are discussed in this document, and introduces the main API terms.



API
A collection of related packages or namespaces
Package/Namespace
A collection of related classes and interfaces
Interface/Class
A collection of related methods (functions) and fields (data)
Field
A data field that other apps can access
API
Package/Namespace
Package/Namespace
Package/Namespace
Interface/Class
method1(param1, param2, …)
method2(param1, param2, …)
field1
CONSTANT1
Enum
CONSTANT1
CONSTANT2
CONSTANT3
Method
A function that other apps can call
Parameter
An input variable that is passed to a method
Constant
A variable used to represent a fixed value
Enumeration (enum)
A fixed set of related constants

The following figure shows a sample Java API structure (SAP NetWeaver 7.50 Enterprise Portal):



SAP NetWeaver 7.50 Enterprise Port

All Classes

Packages

- com.sap.ip.portal.service.ume
- com.sap.netweaver.rtmf.messagingimpl.exception
- com.sap.netweaver.rtmf.messagingimpl.messages
- com.sap.netweaver.rtmf.messagingimpl.services
- com.sap.netweaver.rtmf.serverimpl.servicewrapper
- com.sap.portal.admin.editor
- com.sap.portal.admin.editor.exception
- com.sap.portal.admin.studio
- com.sap.portal.admin.wizard
- com.sap.portal.admin.wizard.exceptions
- com.sap.portal.admin.wizard.handlers
- com.sap.portal.admin.wizard.panes
- com.sap.portal.admin.wizard.validators
- com.sap.portal.adminstudio.ilockingpolicy

- ComplexComparator
- ConnectionClosedException
- ConnectionFactoryNotFoundException
- ConnectionFailedException
- ConnectionProperties
- ConnectivityException
- ConnectivityRuntimeException
- ConnectorException
- ConnectorRuntimeException
- Constants
- ContentDescriptor
- ContentTaggingException
- CoreConstants
- CreateMethod
- DefaultAliasAlreadyExistsException
- DefaultAliasNotFoundException
- DefaultHttpExceptionReporter

com.sap.portal.wpc.api.content.provider

Class ContentDescriptor

java.lang.Object
    com.sap.portal.wpc.api.content.provider.ContentDescriptor

```
public class ContentDescriptor
extends Object
```

Provides a descriptor for a content item of a custom content provider. Used by the IContentProvider for describing the content it provides.

Field Summary

Fields

Modifier and Type	Field and Description
static String	CREATION_METHOD_COPY
static String	[illegible]

Constructor Summary

Constructors

Constructor and Description
ContentDescriptor(String ID, String name, String description, String iconURI, String contentHandlerID, String templateID)
ContentDescriptor(String ID, String name, String description, String iconURI, String contentHandlerID, String templateID, String c

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
String	getContentHandlerID() Returns the ID of the IContentHandler
String	getCreationMethod() Returns the creation method used when adding the content item to a page.

Glossary

The following table introduces the API terms and terms related to API documentation, which are used in this document.

Term	Definition
API element	A generic name for any element of an API, such as a package, namespace, interface, class, method, property, constant, or enumeration.
API documentation comment	A description written in the source code for each API element according to certain rules and syntax. These comments are processed by documentation generators. When documentation generation is not available or applicable for certain scenarios or types of APIs they are documented manually. For more information, see 4.1 Documentation Comments [page 13].
API documentation generators	Tools that process source code to extract documentation comments and generate structured API reference documentation, for example, the widely-used industry tools, such as Javadoc, JSDoc, Doxygen, and Swagger.
documentation tag	A tag in a documentation comment that instructs the generator how to format this part of the comment. Each generator recognizes its own set of tags, however, there is a subset of commonly used tags supported by most generators. For more information, see 4.1.2 Tags [page 16] and 4.2 Documentation Tags [page 17]. Note that this document describes a subset of the most commonly used tags. For a complete reference of tags for a certain language or technology, refer to 7 External Resources [page 66].
deprecated	The state of an API element that will no longer be supported in future releases, and therefore is not recommended for use. Deprecated elements cannot be immediately removed from APIs, because this can break existing client code. Deprecation is a tool for ensuring smooth transition between API releases. Deprecated elements must be documented according to the guidelines described in 4.2.1.1 @deprecated Tag [page 17].
exception	An event that is generated when an error occurs during the execution of a method. Exceptions must be caught and handled by a calling application, therefore they need to be documented along with the method. For documentation guidelines, see 4.2.1.9 @throws Tag [page 26] and 4.2.4.2 <exception> Tag [page 32].
public vs. private APIs	An API is public if it is available in the public domain. An API is private or internal, if access to it is limited to the vendor company and/or to its partners or selected customers. Once an API becomes available, both publicly or internally, it is a contract between the vendor and its clients, so it should not be changed in a way that can break existing client code.
return type, value	A value of a certain data type that is returned by a method. For documentation guidelines, see 4.2.1.6 @return Tag [page 23] and 4.2.4.6 <returns> Tag [page 35].
REST (Representational State Transfer) APIs	REST APIs also known as RESTful Web services are cross-platform APIs used to perform CRUD (Create, Read, Update, Delete) operations on data resources over HTTP. For background information and documentation guidelines, see 5 REST and OData API Reference Documentation [page 47].
SPI (Service Provider Interface)	An interface defined by a vendor platform to be implemented or extended by a third-party application to provide a service integrated with the platform. For documentation guidelines, see 4.3.3 Interface and Class Template [page 40].

3 API Documentation Processes

Outlines recommended processes and workflows for creating API reference documentation.

Roles and Responsibilities

The following table describes the roles involved in the authoring of API reference documentation and their respective responsibilities.

Deliverable/ Responsibility	API Naming	Auto-Generated API Documentation	Manually Written API Documentation
Location	Source code	Doc comments in source code (for example, Java APIs).	Topics in the documentation system.
Developer	Creates initially, implements review.	Creates initially and maintains in the code, implements review. Responsible for production.	Provides initial information or specification, reviews when written.
Technical writer	Performs a thorough review in cooperation with developers.	Performs a thorough review in cooperation with developers.	Creates and maintains, implements review. Responsible for production.

API Reviews

API Naming Review

Review of API naming by a technical writer is essential to ensure that the names of API elements are meaningful, clear, consistent, and self-explanatory. It is important to review the API naming early on in the development cycle to minimize changes later on, especially if the APIs will be used internally by other development groups in the organization much earlier than they are released to customers.

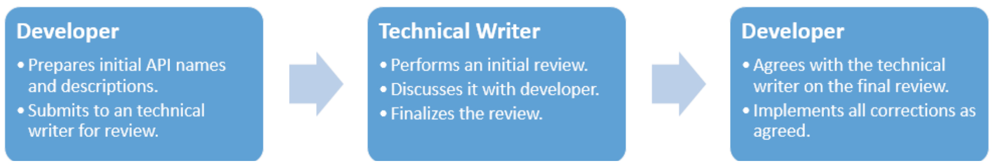
Review of Documentation Comments

If the API documentation is generated automatically, the developer writes and maintains documentation comments in the source code. In this case, the technical writer should review these comments to ensure their quality.

If the API documentation cannot be generated automatically, it is written manually by a technical writer. In this case, a developer should provide a document with the API specifications. The technical writer will review and use these specifications as a source for documentation topics, editing them as required.

Note
Even though it is possible to review comments later on in a development cycle, it is more practical to review names and comments together to ensure their correctness and consistency. A mismatch between an API member name and its description indicates that one of them is incorrect.

The following figure depicts an interactive API review process, which applies to both auto-generated and manually written API documentation.


Developer
• Prepares initial API names and descriptions.
• Submits to an technical writer for review.
Technical Writer
• Performs an initial review.
• Discusses it with developer.
• Finalizes the review.
Developer
• Agrees with the technical writer on the final review.
• Implements all corrections as agreed.

An API review can be performed in any format that is convenient for both technical writer and developer. You can use a code review tool such as Gerrit or work on copies of source code files or generated output.

Process Guidelines for Development Teams

1. Plan API reviews in the same development cycle with the API implementation.
2. Include API reviews in the relevant backlog items.
3. Prepare the API specifications.
4. Submit the APIs for review as soon as possible; late submissions put the review at risk.
5. Implement the review feedback in full, otherwise you lose part of its benefits.
6. Implement the review before APIs are used by any clients (internal and external).
7. For auto-generated API documentation, perform a quality check, as described in 3.3 API Reference Quality Checklist [page 12].

Translation Considerations

In most cases, API reference documentation is delivered in English only, even if the product needs to be localized. However, sometimes it can be translated into selected languages. In this case, technical writers should take into account translation considerations for the target languages.

3.1 API Naming Guidelines

Meaningful, clear, and self-explanatory naming is a key factor in API's usability. Using consistent naming conventions across all platform's APIs contributes to easier adoption of these platforms by customers and partners.

Even though API names are often defined by developers, it is important for technical writers to be involved to ensure that these names are:

- Written in professional and correct English.
- Using correct terminology.
- Consistent, meaningful, and unambiguous.
- Compliant with the industry-wide naming conventions for the relevant language or technology.

Word Combination Conventions

In many cases, a single word is not enough to convey the meaning of an API element, so a name will be a combination of two or more words. The common word combination conventions for names in different languages are as follows:

- Case-separated words: `PascalCase` or `camelCase`.
- Words in lower case delimited by the underscore: `snake_case`.
- Words in lower case delimited by the hyphen: `kebab-case`.

None of these conventions is a preferred industry standard, especially regarding the parameter and property names. The choice of convention largely depends on the original language in which the APIs are written. The following guidelines apply to the languages and technologies such as Java, JavaScript, C/C++, .NET and REST APIs.

4 Java, JavaScript and MS.NET API Reference Documentation

Standards and guidelines discussed in this chapter apply to API reference documentation that is auto-generated from Java, JavaScript and Microsoft.NET source code.

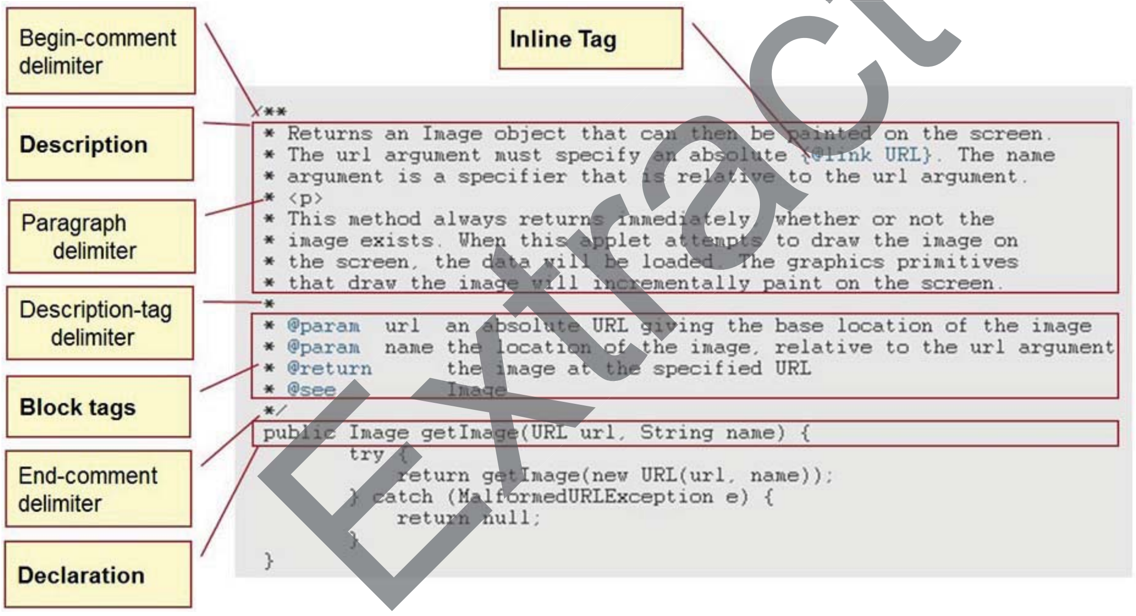
4.1 Documentation Comments

API reference documentation is generated from the documentation comments that are written in the API source code according to certain rules.

A documentation comment should precede the declaration statement of a namespace, class, interface, or class or interface element. A comment is made up of two parts: description and block tags, separated by delimiters.

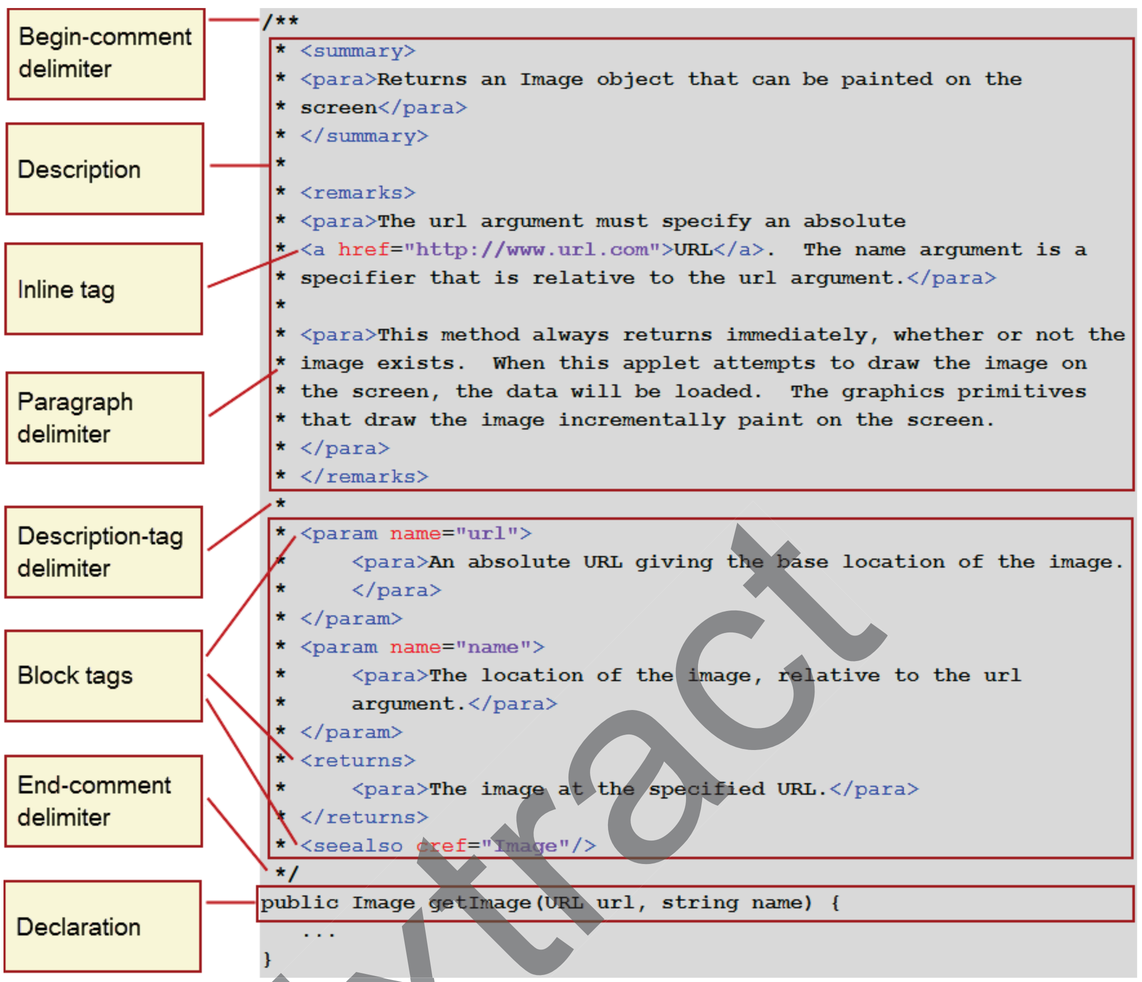
The following figures show the structure and syntax of a documentation comment.

Java, JavaScript



```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

.NET



```
/**
* <summary>
* <para>Returns an Image object that can be painted on the
* screen</para>
* </summary>
*
* <remarks>
* <para>The url argument must specify an absolute
* <a href="http://www.url.com">URL</a>.  The name argument is a
* specifier that is relative to the url argument.</para>
*
* <para>This method always returns immediately, whether or not the
* image exists.  When this applet attempts to draw the image on
* the screen, the data will be loaded.  The graphics primitives
* that draw the image incrementally paint on the screen.
* </para>
* </remarks>
*
* <param name="url">
*     <para>An absolute URL giving the base location of the image.
*     </para>
* </param>
* <param name="name">
*     <para>The location of the image, relative to the url
*     argument.</para>
* </param>
* <returns>
*     <para>The image at the specified URL.</para>
* </returns>
* <seealso cref="Image"/>
*/
public Image getImage(URL url, string name) {
    ...
}
```

For .NET APIs, it is possible to place documentation comments in an external XML file and then use the `<include>` tag to reference that file in the source code.

4.1.1 Description

Description is the first and mandatory part of a documentation comment for a class, interface, or class or interface element.

A description is usually made up of two parts:

- A mandatory summary sentence containing a short and exact description of the declared member.
- An optional detailed description that provides additional information about this element.

Guidelines

- In the summary sentence, omit clauses like "This class" or "This method". For an element that represents an action, start directly with a verb in the third-person form: adds, allocates, constructs, converts, deallocates, destroys, gets, provides, reads, removes, represents, returns, sets, saves and so on. For example:
  - › `Adds a new customer`
  - › `Provides read and write access to employee data`
  - › `Retrieves a Role object`

– For an element that represents an object rather than an action, use a noun phrase. For example:
  › `Base class for navigation`
  › `Alias of a backend system`
– Write the detailed description only to provide additional information that does not repeat the self-explanatory API name or the summary sentence.
– Avoid implementation details and dependencies unless they are important for usage.
– To avoid line wrapping, make sure each line of the description has fewer than 80 characters.
– In the output, the line breaks in a description are ignored, and it appears as a continuous text. To format descriptions, use HTML tags.
– To offset language keywords, API names, and code examples in a description, use the `<code>` tag.

Syntax

Java and JavaScript

Only the summary sentence, terminated by the first period, appears in the summary section of a generated reference. Everything after the first period is cut off, so make sure that the summary sentence can stand on its own.

Java

```
/**
 * This is the summary sentence.
 * <p>
 * This is the detailed description. Note that you can have multiple
 * sentences in the detailed description.
 * </p>
 **/
```

.NET
The summary sentence and detailed description are enclosed by the dedicated tags, `<summary>` and `<remarks>`. To format descriptions, use the `<para>` tag.

Code Syntax
.NET

```
<summary>
   <para>This is the summary sentence.</para>
</summary>
<remarks>
   <para>This is the detailed description.</para>
   <para>Note that you can have multiple sentences in the detailed
description.</para>
</remarks>
```

Related Information

4.2.1 Java and JavaScript Tags [page 17]
4.2.4 .NET Tags [page 31]
4.2.5 HTML Tags [page 36]
7 External Resources [page 66]

4.3.4 Method Template

A template for documenting a method.

The doc comment should provide the following information:

– Background information necessary to understand and use this method.
– Special considerations that apply to this method.

Documentation comments for a method are comprised of two parts: description and block tags. A description has a mandatory first sentence and optional additional sentences. Block tags are listed in a specific order, as shown below.

Syntax

```
/**
 * Constructs/Returns/Sets/Displays/Adds/Removes/Creates/Releases/Other_verb
the ...
 * <p>More information</p>
 *
 * @param param1_name A(n) <code>param1_type</code> object that ...
 * @param param2_name A(n) <code>param2_type</code> that ...
 * @return A(n) <code>method_type</code> object that ...
 * @throws exception_name If ...
 * @see
 * @since
 * @deprecated As of
 * Replaced by {@link anotherMethod_name}
 */ public method_type method_name(param1_type param1_name, param2_type)
throws exception_name;
```

The following table lists standard formulations to use for descriptions of different method types, such as constructors, setters, getters, and so on:

Method Type	Verb to Use
Constructor	Constructs
Boolean	Indicates (whether...)
Getter	Returns/retrieves/gets
Setter	Defines/sets
Other	Adds/Removes/Creates/Releases/Other_verb that applies

The description of a setter method should contain the default value of the property to be set, if any. If the property is set via a constructor, you should mention the default value in the description of the constructor.

Note
You can add snippets of codes to this template using the <pre> HTML tag inside paragraph tags.

```
      * <p>For example:
      * <pre>
      * …
      * </pre>
      * </p>
```

Constructor Example

```
/**
 * Constructs a new HTTP request.
 *
 * @param logonToken A <code>String</code> used to log on to the session
 * @throws Exception If the object is not correctly initialized
 */
public Request(String logonToken) throws Exception { this.logonToken = logon-
Token;
}
```

Accessor Example

```
/**
 * Returns the fully qualified table name that identifies a table.
 * <p>
 * The returned string is formatted as "qualifier"."owner"."tablename".
 * </p>
 *
 * @param qualifier A <code>String</code> that represents the qualifier of the
table
 * @param owner A <code>String</code> that represents the owner of the table
 * @return A <code>String</code> containing the fully qualified table name
 * @see #splitTableFullName(String)
 * @since 14.1.2
 */
String getTableFullName(String qualifier, String owner, String table);
```

Setter Example

```
/**
 * Sets the column description.
 *
 * @param value A <code>String</code> that represents description of the col-
umn
 * @see #getDescription()
 */
public void setDescription(String value);
```

Boolean Example

```
/**
 * Indicates whether the object is mandatory in the query.
 *
 * @return <code>true</code> if it is mandatory, <code>false</code> otherwise
 * @see #setMandatory(boolean)
 */
boolean isMandatory();
```

5 REST and OData API Reference Documentation

Background information, guidelines and recommendations for the authoring of auto-generated and manual REST and OData API reference documentation.

About REST APIs

REST (Representational State Transfer) APIs, also known as RESTful Web services, are cross-platform APIs used to perform CRUD (Create, Read, Update, Delete) operations on data resources over HTTP. This is done by sending a standard HTTP method request to a specific resource URL and receiving an HTTP response in a structured format.

The following table maps data operations to HTTP methods:

CRUD Operation	HTTP Method
Create	PUT/POST
Read	GET
Update	PUT/PATCH
Delete	DELETE

Unlike APIs in object-oriented languages, REST APIs have a flat structure. An individual REST method is defined by the following:

– Resource URL
– Operation (HTTP method)
– Request format
– Request parameters Response format

A REST service is usually a collection of related methods that perform different data operations on the same resource or related set of resources and/or provide related functionality.

About OData

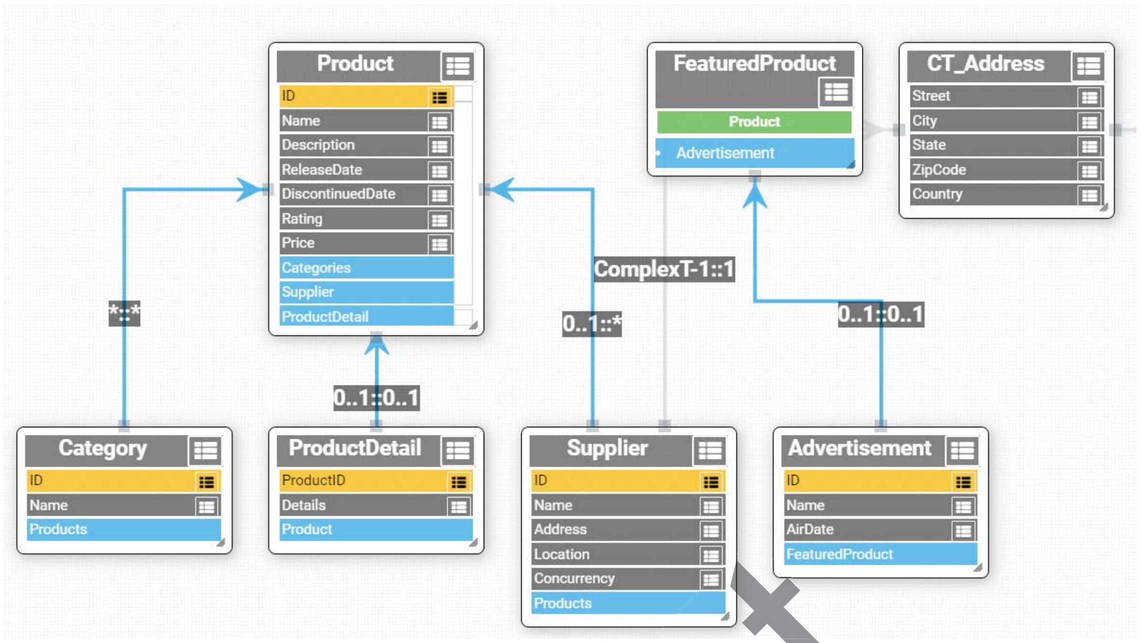
The Open Data Protocol (OData), defined by OASIS, is a standard protocol for interacting with data via RESTful interfaces. The protocol supports the description of data models and the editing and querying of data according to those models.

An OData service is an implementation of the OData protocol that exposes data to external clients. Exposed data is described by an abstract Entity Data Model (EDM). The central concepts in the EDM are entities, entity sets, relationships, and operations.

– Entities are data objects of a certain type, such as Customer or Employee.
– Entity sets are named collections of entities, such as Customers.
– Relationships connect one entity to another.
– Operations, such as Create or Update, are executed on entities.

Client applications can query an OData service to discover its data model and capabilities, and perform CRUD (Create, Read, Update, Delete) operations on entities using REST APIs.

The following figure displays a fragment of a sample data model generated by querying a publicly available OData service (http://pragmatiqa.com/xodata/).



OData Service Types and Documentation Requirements

OData services can be divided into two main types:

– OData producer services.
  Services that expose their data using REST APIs according to the OData protocol.
– OData consumer services.
  Applications that consume OData producer services and publish client APIs in different languages to facilitate data access, for example, client libraries for various development platforms and devices or open data portals.

Audience and documentation requirements differ for APIs published by OData producers and consumers:

Publisher	Audience	Documentation Requirements	Example
OData producer	Developers who directly consume the producer's data services in their applications or create their own consumer services.	Producer OData APIs are REST-based. Documentation should include information about the EDM, service endpoints and permissions, supported authentication protocols, supported OData features and versions of the protocol, relevant implementation specifics and limitations, resources and operations.	OData REST API
OData consumer	Developers who access the data from their client apps using the consumer's APIs.	API documentation is written according to the standard for the platform, technology, or language in which the APIs are created, such as Java or .NET.	XOData: Visualizer and Explorer of OData Services

Related Information

Resource Naming Conventions for REST APIs [page 11]

6 Writing Developer Guides

Guidelines and best practices for writing helpful developer guides.

Developer guides differ by various parameters, such as platform, technology, product, scope, size and more, so there is no one-fits-all standard. Here are some generic guidelines how to make your developer guides clear, concise, helpful, and pleasant to use.

A typical developer guide has the following characteristics:

- It complements API reference documentation by explaining how to use the APIs (and/or services, SDK, development platform).
- It contains information of the following types:
  - Conceptual: the subject domain background, goal, scope and capabilities of the APIs, architectural diagrams that explain the API structure and the usage flow from the user perspective.
  - How to access the APIs: security requirements, initial setup, configuration, etc.
  - How to use the APIs: typical tasks and scenarios, code samples, tutorials, tips and tricks, usage
    considerations.
- It is created and maintained by technical writers in the documentation systems.
- It is written in free style.
- It is delivered as part of product documentation.
- It has an effective navigation and search capabilities.

6.1 Integration of API Reference Documentation

The currently available auto-generation tools don't provide a natural way to integrate a generated API reference with a written developer guide. Unless this integration is supported by a custom solution, direct linking from the developer guide topics to the corresponding API reference pages might be problematic. If this is the case, make sure that your developer guide has a highly visible link to the entry point of the API reference.

If the API reference is written manually, you have more freedom to integrate it with the rest of the guide. You can either make it a separate reference chapter in your guide or distribute topics by usage scenarios. From the task topics, you can link to the relevant reference topics.

6.2 Writing Guidelines

Information Design

- Maintain efficient structure: have separate chapters for concepts, tasks, and reference.
- Apply task-oriented rather than descriptive approach.
- Maintain consistent topic title conventions across the guide.
- Enable easy navigation in the guide.

Choice of Content

- A developer guide should not attempt to cover all APIs in a product. Work with a product owner to determine which use cases need to be included in the guide and plan the topics accordingly.
- Keep the guide topics short and concise.
- Provide only information that is relevant for customers. Avoid describing internal implementation details.

– Good diagrams help a lot to understand concepts. However, make sure they are not too complex or cluttered by redundant details. If you reuse internal architectural diagrams, adapt them for external customers by removing irrelevant parts.

Code Samples

– To explain specific tasks, provide helpful code samples rather than verbose explanations.
– Get code samples from developers, and make sure that they:
  › Compile without errors.
  › Are short, contain code only to illustrate the usage of an API.
  › Are sufficiently commented.
  › Can easily be copied and pasted into a code editor.
– For complex implementation tasks, use tutorial format, breaking it down into smaller chunks or subtopics.