



Plan 2024

Programación II

Parte II - v.2



Ing. David CECCHI

UTN

UML

Un modelo es una representación simplificada de la realidad, capta los aspectos más relevantes (desde un punto de vista determinado), y simplifica u omite el resto. Permite abstraer la realidad a un nivel comprensible, sin perderse en los detalles.

El modelo de un sistema de software ayuda a los desarrolladores a explorar fácilmente diversas arquitecturas y soluciones de diseño antes de proceder con la codificación.

UML (Lenguaje de Modelado Unificado) es un lenguaje de modelado de propósito general que pueden utilizar todos los modeladores; es extenso y variado, pensado para ser utilizado a diferentes niveles y en múltiples etapas del ciclo de vida de desarrollo. No tiene propietario y está basado en el común acuerdo de gran parte de la comunidad informática.

A través de UML pueden representarse diferentes modelos, entre ellos: diagrama de clases, diagrama de objetos, diagrama de casos de uso, diagrama de transición de estados, etc.

DIAGRAMA DE CLASES

En un diagrama de clases, como su nombre lo indica, se representan las clases y sus relaciones. Se pueden incluir interfaces y tipos de enumeración.

Cada clase se representa por un recuadro constituido por tres segmentos: en el segmento superior se detalla el nombre de la clase, en el segmento medio los atributos y en el último segmento los métodos.

Adicionalmente, sobre cada propiedad y método se podrá indicar el grado de visibilidad y el tipo de dato asociado. A su vez, cada tipo de relación cuenta con una representación predeterminada, permitiendo además indicar la cardinalidad de la misma, si corresponde.

El nivel de detalle o abstracción representado dependerá de la etapa de modelado en que se encuentre el proceso y del objetivo que se pretenda alcanzar con el diagrama en ese momento. Es decir, si se está en una fase inicial del proceso de modelado quizás con solo mencionar los nombres de las clases y sus vinculaciones sin profundizar en los detalles de las mismas se esté cumpliendo el objetivo. Luego, en etapas posteriores, donde se requiera especificidad para avanzar, se profundizará en los detalles pendientes dando lugar a un diagrama completo.

Si es necesario extender la aplicación del lenguaje a través de algún elemento para el cual no existe una representación predeterminada, es posible recurrir a los estereotipos, los mismos se definen encerrando su nombre o denominación entre símbolos dobles de menor y mayor, ej: <<nombre del estereotipo>>.

Visibilidades más relevantes		
Símbolo	Definición	Alcance
-	Privada	El atributo o método puede ser accedido solo dentro de la misma clase
#	Protegida	El atributo o método puede ser accedido dentro de la misma clase y desde las clases que hereden de ella.
+	Publica	El atributo o método puede ser accedido desde cualquier lugar de la aplicación.

Los atributos estáticos (atributos de clase) no cuentan con un modificador de visibilidad, solo se notan subrayados.

Los **atributos** se notan con el siguiente formato:

visibilidad nombre_atributo: tipo = valor_inicial

Aunque el formato detallado anteriormente es el estándar, es común simplificar el mismo especificando el nombre y el tipo o únicamente el nombre.

Los **métodos** se notan con el siguiente formato:

visibilidad nombre_metodo (parametros): tipo_devuelto

Aunque el formato detallado anteriormente es el estándar, es común simplificar el mismo especificando el nombre del método y, en ocasiones, el tipo devuelto.

A los métodos que no retornan un valor, se suele no especificar el tipo de dato “devuelto” (void).

El marcador de visibilidad se puede suprimir. La ausencia de un marcador de visibilidad solo indica que la visibilidad no se muestra, no implica que sea pública o no esté definida.

A continuación, se presentan tres alternativas válidas para la representación de la clase Empleado:





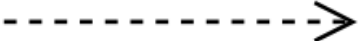


Empleado	Empleado	Empleado
-legajo: int -nombre: string -fechaNac: date -edad: int {readOnly}	-legajo: int -nombre: string -fechaNac: date -edad: int {readOnly}	legajo: int nombre: string fechaNac: date edad: int {readOnly}
+Empleado() +Empleado(int, string, date) +getLegajo() +setLegajo() +getNombre() +setNombre() +getFechaNac() +setFechaNac() +getEdad()	+Empleado() +Empleado(int, string, date)	Empleado() Empleado(leg: int, nom: string, fec: date)

Se debe tener presente que en POO los atributos no suelen definirse como públicos, de hacerlo se estaría brindando acceso directo a los mismos desde el exterior y consecuentemente no se respetaría el concepto de encapsulamiento. Si se requiere que un atributo sea accesible desde el exterior, el mismo se debe mantener como privado e implementar su método de propiedad, separando getter y setter, con visibilidad pública.

En las dos últimas representaciones, con el objetivo de simplificar la legibilidad del diagrama, se asume implícitamente que todas las propiedades cuentan con getter y setter, exceptuando Edad para la cual se ha especificado el modificador readOnly. Dicho modificador indica que para Edad únicamente se deberá definir el getter (solo lectura).

Es posible especificar si un atributo debe aceptar múltiples valores, es decir, representará una colección. Para ello, se deben utilizar corchetes indicando en su interior la cantidad de instancias aceptadas, ej: dirección: String [*].

TIPOS DE RELACIONES ENTRE CLASES

Representación	Tipo de Relación
	Asociación Bidireccional
	Asociación Unidireccional
	Agregación
	Composición
	Dependencia
	Realización/Implementación
	Generalización/Herencia

RELACIÓN DE ASOCIACIÓN

Una relación de asociación describe una vinculación entre objetos, estos objetos pueden pertenecer a clases diferentes o a la misma clase (autoasociación).

Dirección

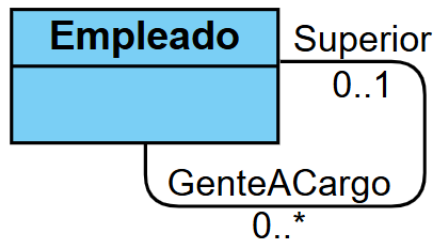
Una asociación puede ser bidireccional o unidireccional. Bidireccional implica que será posible navegar desde instancias de la clase A hacia instancias de la clase B, y desde instancias de la clase B será posible acceder a instancias de la clase A. En una relación unidireccional se incorpora una flecha en uno de los extremos que indicará el único sentido de navegación.

Cardinalidad

En una asociación, es necesario indicar cuántos objetos de cada clase pueden estar involucrados en la relación. En otras palabras, es necesario indicar la cardinalidad (o multiplicidad) de la asociación. Dadas dos clases A y B, la cardinalidad indica con cuántas instancias de A se puede relacionar cada instancia de B (valor mínimo y máximo) y viceversa. Los casos más generales de cardinalidad son los siguientes:

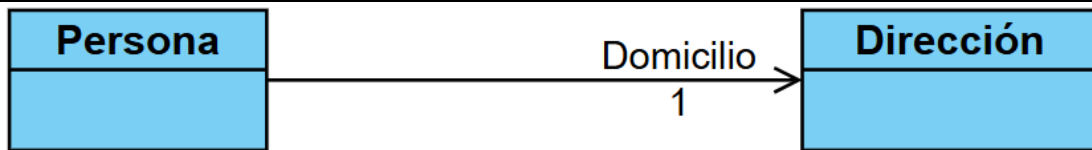
- Cardinalidad “uno a uno”: una instancia de la clase A se relaciona con una única instancia de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad “uno a muchos”: una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con una única instancia de la clase A.
- Cardinalidad “muchos a muchos”: una instancia de la clase A se relaciona con varias instancias de la clase B, y cada instancia de la clase B se relaciona con varias instancias de la clase A.

Cardinalidad	Descripción
1	Solo uno
*	Mínimo ninguno, máximo sin límite
0..*	Mínimo ninguno, máximo sin límite
1..*	Mínimo uno, máximo sin límite
N	Número exacto
N..M	Mínimo N, máximo M



Un Empleado podrá tener a cargo varios Empleados, uno o ninguno, a su vez todos tendrán un Empleado responsable de ellos o ninguno (ninguno: el propietario de la empresa figuraría sin un superior).

Las propiedades que se agregan a Empleado a partir de la relación se denominan GenteACargo y Superior.



Una Persona tendrá asociada una Dirección, mientras que una Dirección no tendrá asociada Personas. Es decir, una Dirección no conoce la Persona o Personas que la tienen asociada.

La propiedad en Persona se denomina Domicilio.

A modo de ejemplo, se podría interpretar que una Dirección contendría propiedades tales como: calle, numeración, piso, depto., etc.



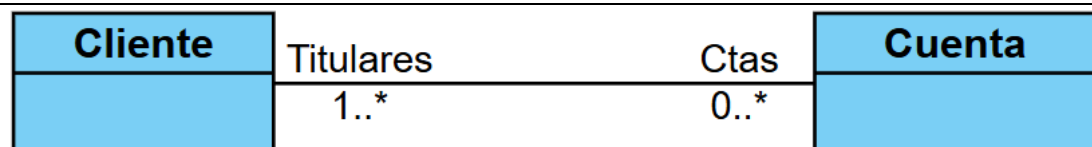
Un Cliente puede o no tener asociada una Cuenta, mientras que una Cuenta no tendrá asociado Clientes. Es decir, una Cuenta no conoce al Cliente o Clientes que la tienen asociada.

La propiedad en Cliente se denomina Cta.



Un Cliente puede o no tener asociada una Cuenta, mientras que una Cuenta siempre tendrá asociado un Cliente.

La propiedad en Cliente se denomina Cta, mientras que la propiedad en Cuenta se denomina Titular.



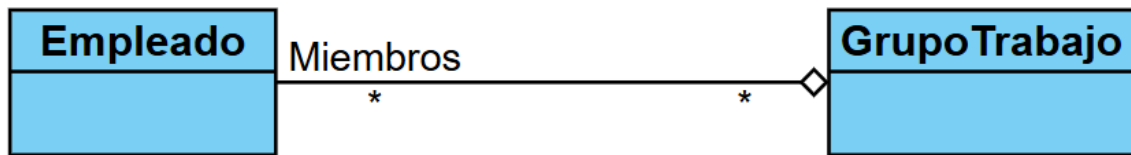
Un Cliente puede tener asociada una, varias o ninguna Cuenta, mientras que una Cuenta siempre tendrá asociado uno o más Clientes.

La propiedad en Cliente se denomina Ctas, mientras que la propiedad en Cuenta se denomina Titulares.

RELACIÓN DE AGREGACIÓN

Es un caso especial de asociación donde se pretende aclarar que una de las clases involucradas representa un TODO y la otra una PARTE de ella, donde la parte puede existir independientemente del todo. La parte se generará fuera del Todo, y es pasada a él a través de algún medio (constructor/método).

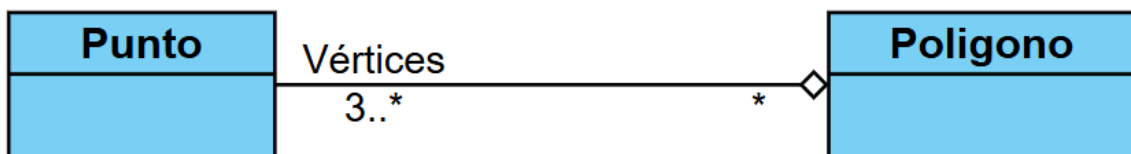
La Parte tiene existencia fuera del Todo, tiene sentido por sí misma. Incluso, un mismo objeto Parte podría ser “parte” de varios objetos Todo.



Un Grupo de Trabajo puede estar constituido por varios Empleados, y un Empleado puede formar parte de varios Grupos de Trabajo.

Eliminar un Grupo de Trabajo, no implica eliminar los Empleados.

La clase GrupoTrabajo incluiría un método que le permitiría agregar un Empleado a su colección.



Un Polígono estará constituido por 3 o más Puntos, y un Punto puede formar parte de varios Polígonos.

Eliminar un polígono, no implica eliminar los Puntos.

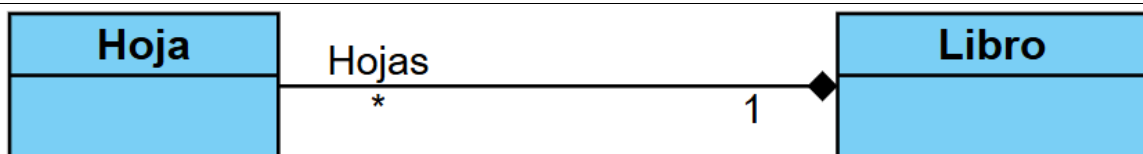
La clase Polígono incluiría un método que le permitiría agregar un Punto a su colección.

RELACIÓN DE COMPOSICIÓN

Es un caso especial de asociación donde se pretende aclarar que una de las clases involucradas representa un TODO y la otra una PARTE de ella, donde la Parte no puede existir independientemente del Todo. La Parte se genera dentro del Todo, no puede ser pasada a él desde el exterior.

La Parte, no tiene de existencia fuera del Todo, no tiene sentido por sí misma. Por lo tanto, nunca un mismo objeto Parte podría ser “parte” de varios objetos Todo.

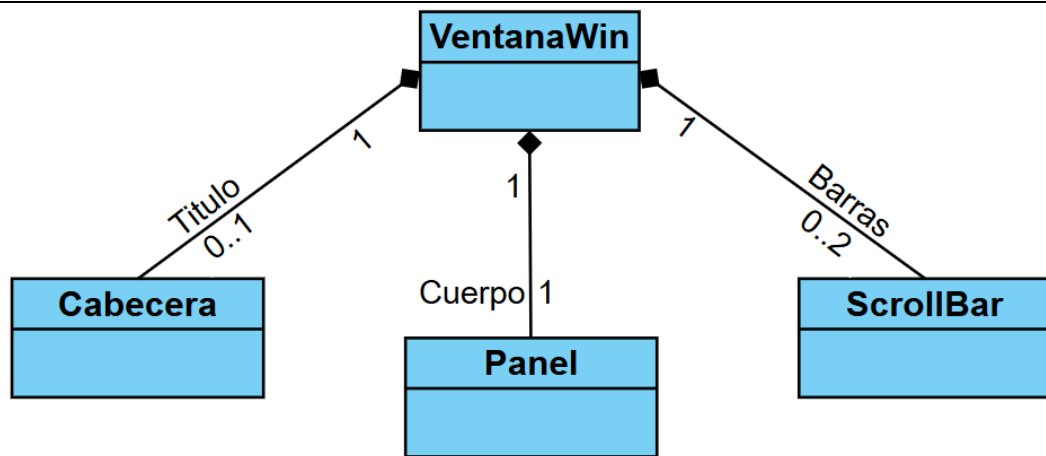
En una relación de composición hay una regla de “no compartir”.



Un Libro estará constituido por varias Hojas, y una Hoja no puede formar parte de otros Libros.

Eliminar un Libro, implica la destrucción de sus Hojas.

La clase Libro incluiría un método que le permitiría generar sus Hojas.



Una Ventana (formulario en aplicación de escritorio) estará constituida por un Panel, hasta dos scrollbars (barras de desplazamiento) y puede o no contener una Cabecera (titulo). Donde su título, cuerpo y barras no pueden formar parte de otras Ventanas.

Eliminar una Ventana, implica la destrucción de los objetos contenidos (Cabecera, Panel y ScrollBars).

La clase VentanaWin incluiría métodos que le permitirían generar su Título, Cuerpo y Barras.

La clase Cabecera podría definir propiedades como: texto contenido, tipo de fuente, tamaño de fuente, etc.

La clase Panel podría definir propiedades como: alto, ancho, color de fondo, etc.

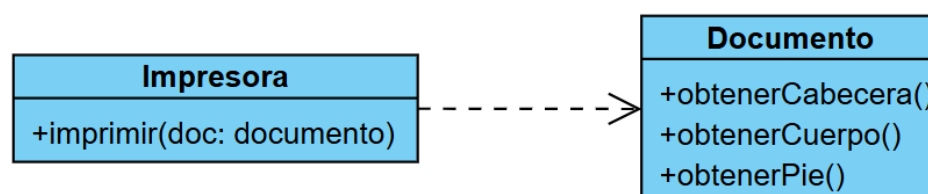
La clase ScrollBar podría definir propiedades como: orientación, posición actual, etc.

Característica	Agregación	Composición
Dos o más compuestos pueden compartir un mismo componente	Sí	No
Al destruir el compuesto se destruyen los componentes	No	Sí
Cardinalidad del compuesto	Cualquier cardinalidad	1
Representación	Rombo vacío	Rombo relleno

RELACIÓN DE DEPENDENCIA

Una dependencia entre clases denota una relación de uso de una sobre otra. Esta situación se presenta cuando un método de la clase A recibe como parámetro un objeto de la clase B o dentro de algún método de la clase A se instancia un objeto de la clase B para desempeñar una tarea determinada. En estas situaciones, se dice que A depende de B, es decir, hace uso de sus servicios.

Por lo cual, en caso de sufrir modificaciones la clase de la cual se depende, es posible que deban introducirse cambios en la clase dependiente.



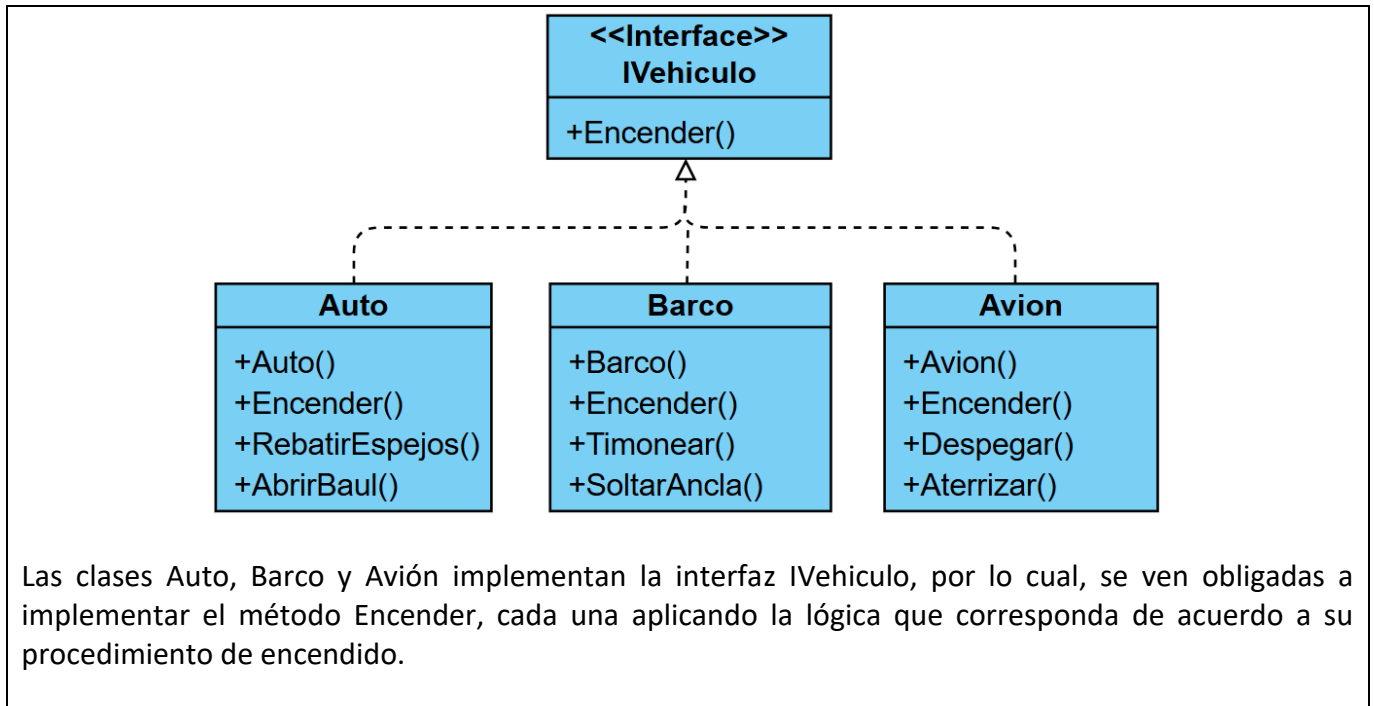
La clase Impresora, para imprimir un documento, invocará a las tres operaciones definidas en la clase Documento. En consecuencia, si los métodos de la clase Documento son eliminados, se modifican sus

parámetros o el tipo de dato retornado, la clase Impresora se verá obligada a modificar la implementación de Imprimir.

RELACIÓN DE REALIZACIÓN/IMPLEMENTACIÓN

Una realización denota una relación entre una interfaz y una clase, donde la clase se encuentra obligada a cumplir con el contrato establecido, es decir, deberá implementar el conjunto de propiedades y métodos declarados en la interfaz.

Una interfaz se identifica con el estereotipo <<Interface>>.



GENERALIZACIÓN/HERENCIA

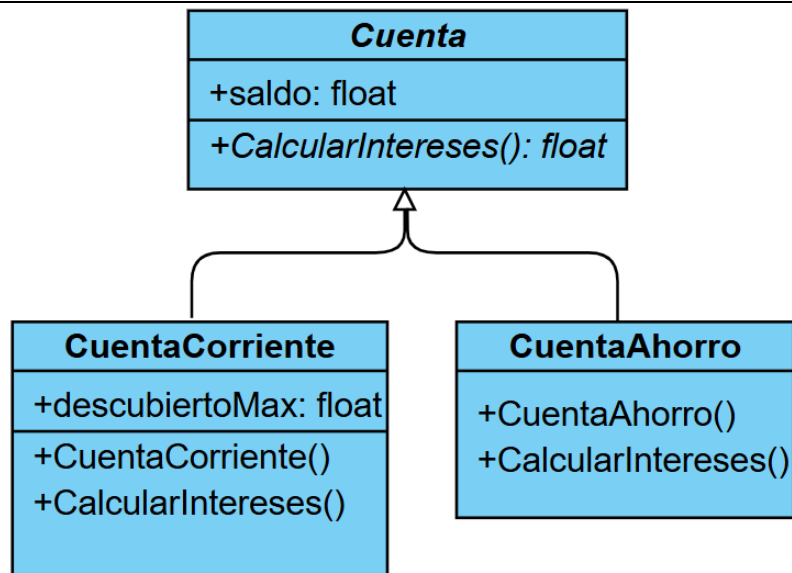
La generalización es una relación entre una clase más general (denominada superclase o clase padre) y una clase más específica (denominada subclase o clase hija).

Toda subclase hereda los atributos y métodos de la superclase, y a su vez puede añadir sus propios atributos y métodos.

Las subclases representan un tipo específico de la superclase, por ello, toda instancia de la subclase puede interpretarse a su vez como instancia de la superclase. Ejemplificando, si se cuenta con la superclase Persona y una subclase de ella es la clase Cliente, implica que una instancia de la clase Cliente (más allá de ser un Cliente) no deja de ser una Persona.

Este razonamiento, permite concluir/entender que toda instancia de la subclase puede emplearse en cualquier espacio donde se requiera una instancia de la superclase.

Para indicar que una clase es abstracta su nombre se nota en cursiva, también está permitido agregar el estereotipo <<abstract>> por encima de su nombre. Los elementos virtuales y abstractos, se notan en cursiva para diferenciarlos de los concretos.



La clase Cuenta es abstracta (se encuentra en cursiva) al igual que su método `CalcularIntereses`, por lo cual, ambas subclases están obligadas a redefinirlo.

JSON

JSON (JavaScript Object Notation) es un formato basado en texto, ligero y de gran popularidad, utilizado para almacenar e intercambiar información con un amplio soporte en diversos lenguajes de programación. Es legible por las personas, y a su vez, tiene la capacidad de ser fácilmente interpretado por las máquinas.

Puntualmente, las características descriptas hacen de Json un formato ideal para intercambiar datos entre el servidor y las aplicaciones web.

Una alternativa a Json es XML, este formato permite representar estructuras de datos más complejas y ricas en detalle (metadatos) pero en contraposición, se torna más complejo para ser interpretado por las personas y pesado al momento de ser procesado/analizado por las máquinas. En consecuencia, Json es ampliamente la opción seleccionada.

Algunas implementaciones de Json:

- Transferencia de datos entre las Aplicaciones Web y el Servidor, la Aplicación captura los datos ingresados por el usuario a través de un formulario, los convierte a formato Json y seguidamente los envía al Servidor para su posterior procesamiento, ej.: almacenamiento en la base de datos.
- Transferencia de datos entre el Servidor y las Aplicaciones Web, donde el Servidor procesa una solicitud y envía a la Aplicación Web los datos en formato Json para que la misma los interprete y presente al usuario en un formato amigable, ej.: formulario, grillas, etc.
- Almacenamiento de datos referidos a la configuración de la aplicación, ej.: appsettings.json.
- Almacenamiento de datos referidos a las credenciales de acceso a la aplicación, ej: JWT (Json Web Token).

SINTAXIS

- **Objeto:** se encierra entre llaves {}, y cada propiedad se expresa por medio de un par Clave/Valor, donde la Clave se encierra entre comillas dobles y representa el nombre de la propiedad, mientras que el Valor representa el estado de la propiedad. El Valor debe respetar el formato asociado al tipo de dato correspondiente. La Clave y el Valor deben estar separados a través de dos puntos, y las propiedades deben estar separadas entre sí a través de una coma.
- **Arreglo:** se encierra entre corchetes [] y los valores contenidos deben estar separados por una coma.
- **Cadena:** se encierra entre comillas dobles "", es posible utilizar la barra invertida como carácter de escape para representar, por ejemplo, un salto de línea dentro de la cadena: \n.
- **Numérico:** admite positivos, negativos y punto decimal, no requiere delimitadores.
- **Booleano:** constantes true y false, no requiere delimitadores.
- **Nulo:** constante null, no requiere delimitadores.

EJEMPLO

El siguiente ejemplo describe, en formato Json y su correspondiente XML, un objeto Materia que contiene una colección de Alumnos.

Json	XML
<pre>{ "nivel": 1, "aula": "H03", "materia": "programacion 2", "docentes": ["Riera, Lujan", "Rojas, Dario"], "alumnos": [{ "legajo": 1002, "apelnom": "Gomez, Luis", "fechaNac": "2000-03-17", "entregoTP": true, "empresaTrabaja": "Siderar SA" }, { "legajo": 1017, "apelnom": "Ruiz, Joaquin", "fechaNac": "2010-04-21", "entregoTP": false, "empresaTrabaja": null }, { "legajo": 1109, "apelnom": "Fernandez, Aldana", "fechaNac": "2008-11-03", "entregoTP": true, "empresaTrabaja": "Electro Integral" }] }</pre>	<pre><?xml version="1.0" encoding="UTF-8" ?> <root> <nivel>1</nivel> <aula>H03</aula> <materia>programacion 2</materia> <docentes>Riera, Lujan</docentes> <docentes>Rojas, Dario</docentes> <alumnos> <legajo>1002</legajo> <apelnom>Gomez, Luis</apelnom> <fechaNac>2000-03-17</fechaNac> <entregoTP>true</entregoTP> <empresaTrabaja>Siderar SA</empresaTrabaja> </alumnos> <alumnos> <legajo>1017</legajo> <apelnom>Ruiz, Joaquin</apelnom> <fechaNac>2010-04-21</fechaNac> <entregoTP>false</entregoTP> <empresaTrabaja></empresaTrabaja> </alumnos> <alumnos> <legajo>1109</legajo> <apelnom>Fernandez, Aldana</apelnom> <fechaNac>2008-11-03</fechaNac> <entregoTP>true</entregoTP> <empresaTrabaja>Electro Integral</empresaTrabaja> </alumnos> </root></pre>

Algunas alternativas para generar el Modelo asociado a un Json determinado:

- Desarrollar las clases manualmente.
- En Visual Studio seleccionar la opción “Editar/Pegado Especial/Pegar Json como Clases”.
- Recurrir a los servicios que brindan diversos sitios web, ej.: <https://json2csharp.com/>

Al implementar herramientas automáticas de generación de modelos, las clases obtenidas podrían no respetar los tipos de datos más adecuados o esperados, ejemplificando: atributos que hacen referencia a fechas declarados como string, colecciones declaradas como simples arreglos, etc.

SERIALIZACIÓN

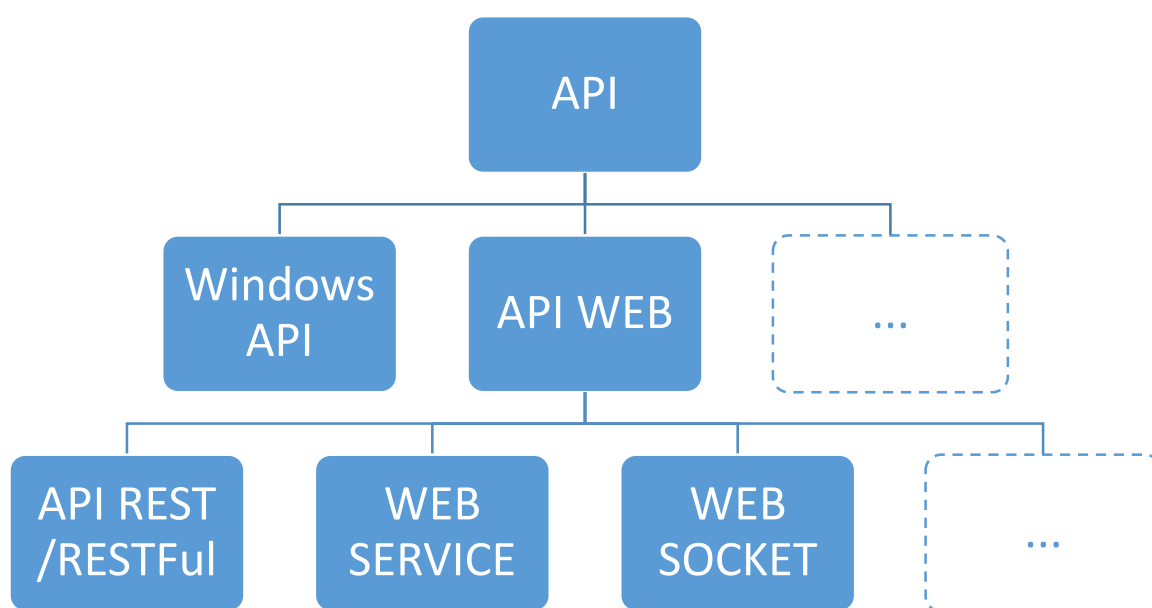
“La *serialización* es el proceso de convertir el estado de un objeto, es decir, los valores de sus propiedades, en un formato que se puede almacenar o transmitir. El formato serializado no incluye información sobre los métodos asociados de un objeto. La *deserialización* reconstruye un objeto a partir del formato serializado.”

Fuente: <https://learn.microsoft.com/es-es/dotnet/standard/serialization/system-text-json/overview>

API

API (Application Programming Interface), representa una Interfaz para la Programación de Aplicaciones. Básicamente, un API es una herramienta que funciona como intermediaria entre dos aplicaciones, permitiendo la comunicación e intercambio de información entre ellas. Al implementar un API, los desarrolladores son capaces de integrar en sus aplicaciones, servicios o funcionalidades externas evitando así la necesidad de invertir tiempo y recursos propios en desarrollarlas desde cero. Incluso, disponiendo del tiempo y recursos necesarios, hay situaciones en las que no es posible acceder por cuenta propia a la información requerida.

La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define la manera en que las aplicaciones involucradas deberán comunicarse entre sí, estableciendo para los desarrolladores la forma en que deben estructurarse las solicitudes y las respuestas sobre los servicios brindados. Las APIs abarcan diferentes formas de interacción, no solo comunicaciones a través de la web.



WINDOWS API

Expone las funcionalidades del kernel de Windows permitiendo a los desarrolladores implementarlas en sus proyectos en situaciones que requieren acceso de bajo nivel sobre el Sistema Operativo (creación de drivers, gestión de memoria, gestión de procesos, construcción de componentes, acceso a red, etc.). El Framework .NET encapsula estas funcionalidades en un nivel más alto de abstracción, facilitando el desarrollo al presentar una experiencia más amigable.

Del mismo modo, Linux también cuenta con su propia API la cual permite acceder a las funcionalidades de su propio kernel.

API WEB

Es un API que ofrece, a través de la Web, un conjunto de funcionalidades proporcionadas por un Servidor para ser utilizadas por aplicaciones Cliente. La comunicación entre los interlocutores se realiza implementando los protocolos HTTP o HTTPS. Es decir, ambos componentes (Servidor y Cliente), pueden implementar lenguajes de programación diferentes siempre que sean capaces de formular solicitudes e interpretar respuestas a través de HTTP.

Según la arquitectura que aplique la API Web, se catalogará como: REST, Web Service, Web Socket, etc. Una arquitectura impone condiciones sobre cómo debe funcionar una API.

API REST/RESTful

REST (Representational State Transfer) Transferencia de Estado Representacional.

La comunicación entre el Servidor y sus Clientes se realiza mediante mensajes HTTP, estos mensajes pueden clasificarse en dos tipos: *peticiones* (request) enviados por el Cliente al servidor solicitando alguna acción, y *respuestas* (response) que representan la devolución del servidor para la acción solicitada.

Las API REST, en la mayoría de los casos, tienen por objetivo llevar a cabo operaciones estándar sobre bases de datos. Estas acciones son conocidas como operaciones CRUD: crear, leer, actualizar y eliminar registros.

Por ejemplo, se utilizaría una solicitud GET para recuperar un registro, una POST para crear un nuevo registro, una PUT para actualizar un registro y una DELETE para eliminar un registro.

Algunas características de las API REST:

- **Comunicación Sin Estado:** implica que cada petición HTTP contiene toda la información necesaria para ejecutarla, esta característica permite que ni el cliente ni el servidor deban recordar el estado de solicitudes anteriores para satisfacerla. El servidor no debe ni necesita almacenar ningún dato relacionado con una solicitud, cada interacción constituye un único ciclo.

Sin embargo, algunas aplicaciones HTTP incorporan memoria caché, en estos casos se define un protocolo cliente-caché-servidor sin estado, donde existe la posibilidad de definir algunas respuestas a peticiones HTTP concretas como cacheables, con el objetivo de que el cliente pueda obtener en un futuro la misma respuesta para peticiones idénticas. El objetivo es mejorar el rendimiento en el lado del cliente y, al mismo tiempo, aumentar la escalabilidad en el lado del servidor.

- **Arquitectura cliente-servidor:** implica una separación total entre el cliente y el servidor. La única información que la aplicación cliente debe conocer es el URI del recurso solicitado. No puede ni debe interactuar con la aplicación servidor de ninguna otra forma. Del mismo modo, una aplicación servidor no debe interactuar con la aplicación cliente más allá de enviar la respuesta a través de HTTP.

Un URI (Uniform Resource Identifier) o Identificador Único de Recurso, es una cadena de caracteres que identifica de forma única cada recurso (endpoint) de la API. La estructura básica de un URI es la siguiente:

{protocolo}://{hostname}:{puerto}/{ruta del recurso}

- **Interfaz Uniforme:** los métodos soportados por HTTP (Get, Post, Put, Delete, etc.) junto con la URI, proporcionan una interfaz uniforme que permite la transferencia de datos en el sistema REST, de esta manera, se aplican operaciones concretas sobre un recurso determinado.

Aunque la mayoría de las operaciones que componen una API REST podrían llevarse a cabo con solo recurrir a los métodos GET y POST, se estaría incurriendo en el incumplimiento del protocolo, alejándose del estándar y llevando a la construcción de URIs con nomenclaturas erróneas mediante el uso de verbos.

Una solicitud, estará compuesta por:

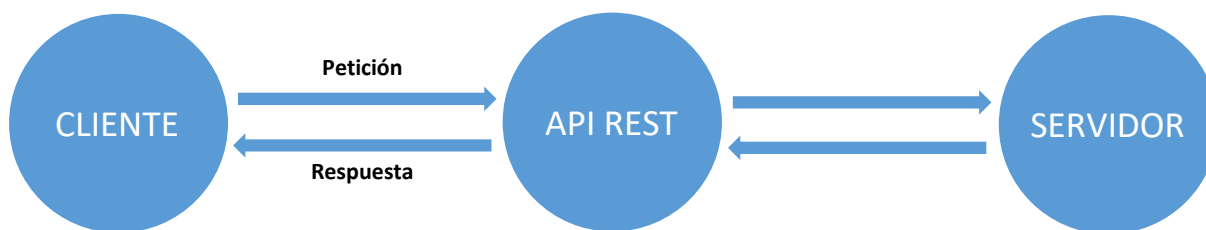
- **Identificador único de recurso (URI):** dirección única del recurso a solicitar.
- **Método HTTP:** hace referencia al tipo de operación a realizar, los más habituales son: Get (consulta), Post (creación), Put (actualización) y Delete (eliminación).
- **Encabezado HTTP:** metadatos que indican, por ej: formato del contenido enviado en la solicitud (ej. JSON/XML), formato del contenido que se espera recibir (ej. JSON/XML), información de autenticación, información (datos y parámetros) asociada al método referenciado, etc.

Una respuesta, estará compuesta por:

- **Código de Estado:** código de 3 dígitos que indica si la solicitud pudo ser procesada correctamente o presentó algún error. Los códigos 2xx indican un procesamiento correcto mientras que los códigos 4xx y 5xx indican errores, los códigos 3xx indican que se requiere una acción adicional para completar la solicitud.

Código	Descripción
200	Ok, la solicitud ha sido procesada correctamente.
201	Create, la solicitud ha sido procesada correctamente, creación exitosa.
202	Accept, la solicitud ha sido recibida pero aún se está procesando.
204	No Content, la solicitud ha sido procesada correctamente, no hay contenido que devolver.
400	Bad Request, la solicitud no ha podido ser procesada, está incompleta o no respeta el formato requerido.
401	Unauthorized, no autorizado. No se han recibido las credenciales de acceso.
403	Forbidden, acceso denegado. El cliente esta autenticado pero no cuenta con permiso suficiente.
404	Not Found, no encontrado. La url no corresponde a un recurso existente.
405	Method Not Allowed, el recurso no admite el método especificado en la solicitud. Ej. indica Post cuando el endpoint es Get.
500	Internal Server Error, error interno del servidor al procesar la solicitud.

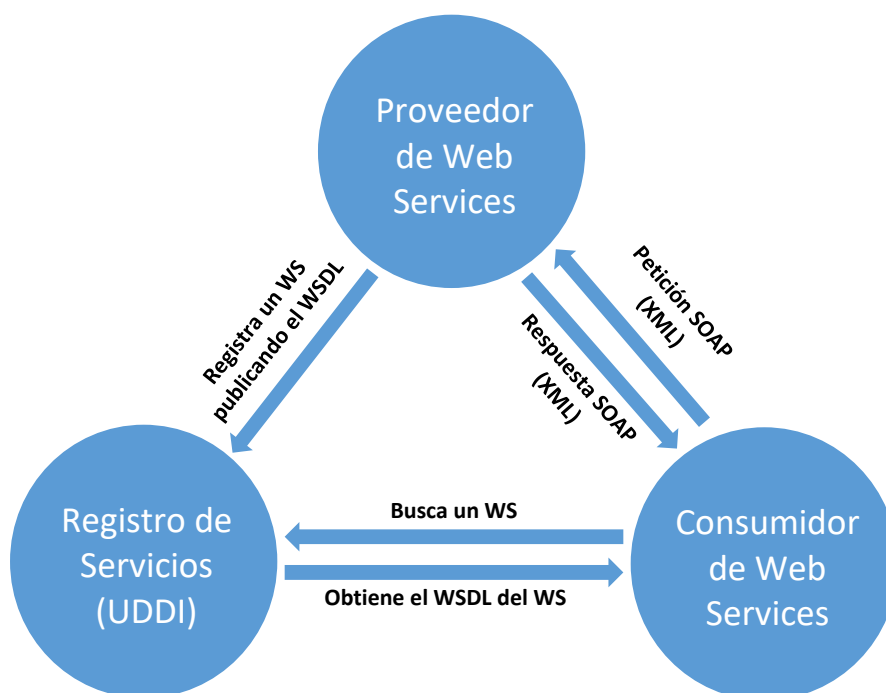
- **Cuerpo de Mensaje:** contiene los datos formateados resultantes del procesamiento de la solicitud. El servidor selecciona el formato de representación de acuerdo a la información recibida a través del encabezado de la solicitud.
- **Encabezado HTTP:** contiene metadatos acerca de la respuesta. Estos brindan más contexto sobre la respuesta e incluyen información relacionada con el servidor, la fecha y hora de la respuesta, el formato del contenido (JSON/XML), etc.



WEB SERVICE

Un webservice, expone un conjunto de operaciones ofrecidas por un servidor para que puedan ser consumidas por aplicaciones cliente. Esta tarea se lleva a cabo implementando los siguientes conceptos:

- Los mensajes enviados y recibidos estarán expresados en formato XML respetando el protocolo SOAP (Simple Object Access Protocol).
- Cada webservice, pone a disposición una descripción de las operaciones que ofrece (nombres de las operaciones, parámetros, tipos de retorno, etc.) expresadas en XML mediante un lenguaje de descripción de servicios a través de un archivo WSDL (Web Service Description Language). Es decir, el archivo WSDL describirá la interface del webservice asociado.
- La publicación de un webservice se realiza a través de la especificación UDDI (Universal Description, Discovery, and Integration) la cual define un modo estándar de publicar y encontrar información sobre servicios web. Existen múltiples registros UDDI, un registro es un repositorio que almacena descripciones de servicios web bajo la especificación UDDI. Las empresas proveedores de webservices se registran (respetando la especificación UDDI) en un repositorio indicando nombre, descripción de la misma, contactos, servicios web ofrecidos, una url al WSDL de cada servicio brindado, etc. Los proveedores podrán publicar sus webservices en uno o más registros de servicios. Luego, los consumidores (clientes) podrán encontrar en los repositorios los servicios web que se ajusten a un determinado criterio de búsqueda.



Los webservices son menos flexibles, altamente estructurados y tienden a transmitir grandes cantidades de datos en sus mensajes en comparación a una API REST, por ello, pueden presentar latencia en los tiempos de respuesta. Están orientados al ambiente empresarial donde se requieren transmisiones más robustas y seguras, es decir, mantener la integridad de los datos es crucial.

Las API REST se han convertido en una opción más popular debido a su simplicidad y escalabilidad.

WEB SOCKET

Una API WebSocket, establece una conexión persistente que permite una comunicación bidireccional en tiempo real. Mientras que en una API Rest las solicitudes y las respuestas son independientes de las anteriores (sin estado), en una API WebSocket se abre un canal de comunicación persistente. Esta característica hace a un WebSocket adecuado para aplicaciones que requieren actualizaciones en tiempo real, ej: juegos online, chats, transmisiones en vivo, etc. donde ninguno de los interlocutores requiere esperar la respuesta de su contraparte para enviar un nuevo mensaje.

El protocolo HTTP es estrictamente unidireccional, por lo cual, el modelo tradicional de solicitud-respuesta de HTTP puede introducir latencia debido a la sobrecarga que supone establecer y finalizar múltiples conexiones entre el cliente y el servidor, el protocolo WebSocket soluciona este inconveniente manteniendo una conexión persistente.

La comunicación comienza cuando el cliente utiliza HTTP para comunicarse con el servidor solicitándole una comunicación por WebSocket, si el servidor acepta, confirmará la solicitud y ambos lados cambiarán al protocolo WebSocket para establecer así una comunicación bidireccional.