

Gobierno de APIs. (API Owner)

RAML

Índice del capítulo

- Raml de un vistazo.
- Recursos.
- Métodos.
- Parámetros.
- Cuerpo.
- Respuestas.
- El modelo de entidades.
- Seguridad.
- Modularización.

Raml de un vistazo

- ▶ Introducción.
- ▶ ¿Qué es RAML?
- ▶ Sintaxis RAML.
- ▶ Sintaxis YAML.
- ▶ Trabajando con un editor.
- ▶ La especificación.

Introducción

- ▶ Hablando del **diseño de API's** existe un principio que nos dice que todo software que se comunica con otro está acoplado de alguna manera, este acoplamiento puede ser débil o fuerte.
- ▶ El **principio de Acoplamiento Débil** del Servicio (**Service Loose Coupling**) descrito por **Thomas Erl** en su libro **SOA Principles of Service Design** nos dice que existen dos tipos de acoplamientos: **positivos** y acoplamientos **negativos**.
- ▶ Un **tipo de acoplamiento positivo** es aquel en el que el **diseño del contrato del servicio** se realiza **antes** de implementar la lógica interna al servicio (approach conocido como **API First**).
- ▶ Esto significa que como diseñador de APIs voy a **poner atención** principalmente en la **definición del contrato** sin tener en cuenta el Backend y sin pensar cómo lo vamos a implementar. Esos detalles se describirán **en etapas posteriores** al diseño del contrato.

Introducción

- ▶ Entre otras ventajas, este enfoque también nos brinda el **beneficio de permitirnos** reemplazar la **lógica del Backend de manera transparente** y con un impacto prácticamente nulo de cara a aquellos que consumen el servicio.
- ▶ Existen **varios lenguajes** en la industria que nos permiten definir el contrato de **nuestra API REST**.
- ▶ **El lenguaje RAML** es uno de ellos, nos permite enfocarnos en el diseño de nuestra API antes de implementarla.

¿Qué es RAML?

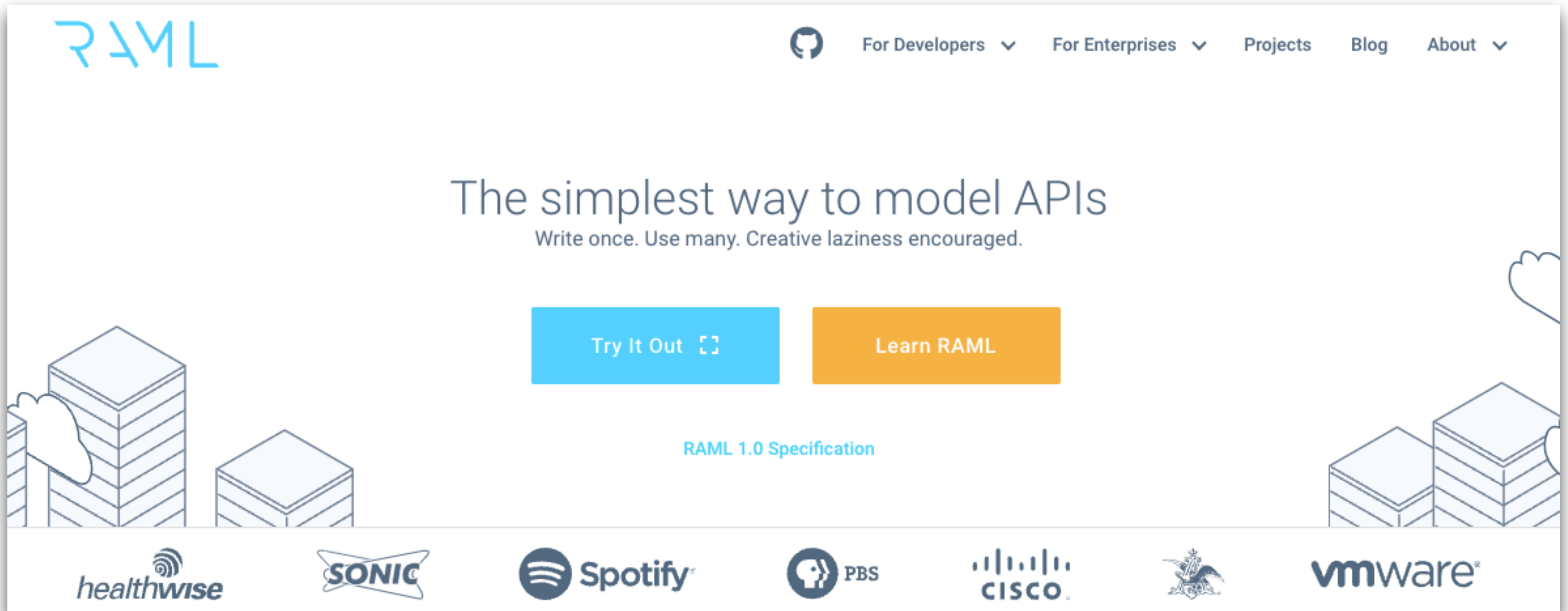
- ▶ RAML es un **lenguaje de modelado para APIs RESTful**.
- ▶ Nos permite escribir el **contrato de la API** y todos sus aspectos como definir sus **recursos, métodos, parámetros, respuestas**, tipos de medios y otros componentes HTTP básicos.
- ▶ Finalmente, puede usarse también para **generar documentación más amigable** de cara a los consumidores del API.
- ▶ Pero **RAML no es el único lenguaje** usado en la industria para describir API's RESTful, también existen otros productos como **Open API Swagger** ambos con mucha popularidad.

¿Qué es RAML?

- ▶ Sin embargo existen **algunas diferencias** entre ambos:
 - ▶ La gran característica de **Swagger** es que está **diseñado** como una especificación con approach **bottom-up** (pone atención principalmente en la implementación y de ahí parte para exponer el contrato), lo **contrario a RAML** que es una especificación del tipo top-down.
 - ▶ **Swagger** cuenta con **una gran comunidad** y algunas herramientas disponibles cosa que con RAML aún no es tanto, existen pocas herramientas para RAML por lo que algunas empresas que lo usan han optado por desarrollar sus propias herramientas personalizadas a las necesidades propias.

¿Qué es RAML?

- <https://raml.org/>



Sintaxis RAML

- ▶ **RAML** se basa en **otro lenguaje llamado YAML**.
- ▶ Define un formato de datos legible por humanos (human friendly) que se alinea bien con los **objetivos de diseño de la especificación RAML**.
- ▶ Al igual que en YAML, todos los nodos son **claves, valores y etiquetas** lo cual ayuda a comprender mejor la lectura de un fichero RAML.
- ▶ Pasamos a describir los elementos básicos de su sintaxis.

Sintaxis YAML

- ▶ Todos los **archivos YAML** pueden comenzar **opcionalmente** con `---` y finalizar con `...`.
- ▶ Esto forma parte del formato YAML e indica el inicio y el final de un documento.
- ▶ Ejemplo:

```
---  
# A list of tasty fruits  
fruits:  
  - Apple  
  - Orange  
  - Strawberry  
  - Mango  
...
```

Sintaxis YAML

- YAML trabaja con dos elementos fundamentales:
 - Diccionarios.
 - Arrays/lista de elementos.

Sintaxis YAML

- ▶ Un **diccionario** es una estructura, un simple **clave: valor** (los dos puntos deben ir seguidos de un espacio).

```
clave: valor
```

- ▶ El **valor**, a su vez, puede ser **otro diccionario** o una lista de **valores** que, a su vez, podrán ser diccionarios o listas.
- ▶ Ejemplo:

```
martin:  
  name: Martin D'vloper  
  job: Developer  
  skill: Elite
```

Sintaxis YAML

- Una lista se define mediante líneas que comienzan en **el mismo nivel de sangría** comenzando por un "-" (un guión y un espacio):

```
---
```

```
fruits:
```

- Apple
- Orange
- Strawberry
- Mango

```
...
```

Sintaxis YAML

- Podemos encontrar estructuras de datos más complejas:

```
- martin:  
  name: Martin D'vloper  
  job: Developer  
  skills:  
    - python  
    - perl  
    - pascal  
- tabitha:  
  name: Tabitha Bitumen  
  job: Developer  
  skills:  
    - lisp  
    - fortran
```

Sintaxis YAML

- Los **diccionarios y listas** también pueden representarse de forma abreviada:

```
---  
martin: {name: Martin D'vloper, job: Developer, skill: Elite}  
fruits: ['Apple', 'Orange', 'Strawberry', 'Mango']
```

Sintaxis YAML

- ▶ Los **valores** pueden **ocupar varias líneas** usando `|` o `>`.
- ▶ Abarcando múltiples líneas usando `|` incluirá las líneas nuevas.
- ▶ Usar `>` ignorará las líneas nuevas; se utiliza para hacer que, de lo contrario, sería una línea muy larga más fácil de leer y editar. En cualquier caso, se ignorará la sangría. Los ejemplos son:

```
include_newlines: |  
    exactly as you see  
    will appear these three  
    lines of poetry
```

```
ignore_newlines: >  
    this is really a  
    single line of text  
    despite appearances
```


Sintaxis YAML

- ▶ Si bien YAML es generalmente amigable, lo siguiente resultará en un error de sintaxis YAML:

```
foo: somebody said I should put a colon here: so I did  
windows_drive: c:
```

- ▶ Pero esto sí funcionará:

```
windows_path: c:\windows
```

- ▶ Cuando **:** aparece **al final o con un espacio en blanco**, debemos incluirlo entre comillas:

```
foo: "somebody said I should put a colon here: so I did"  
windows_drive: "c:"
```

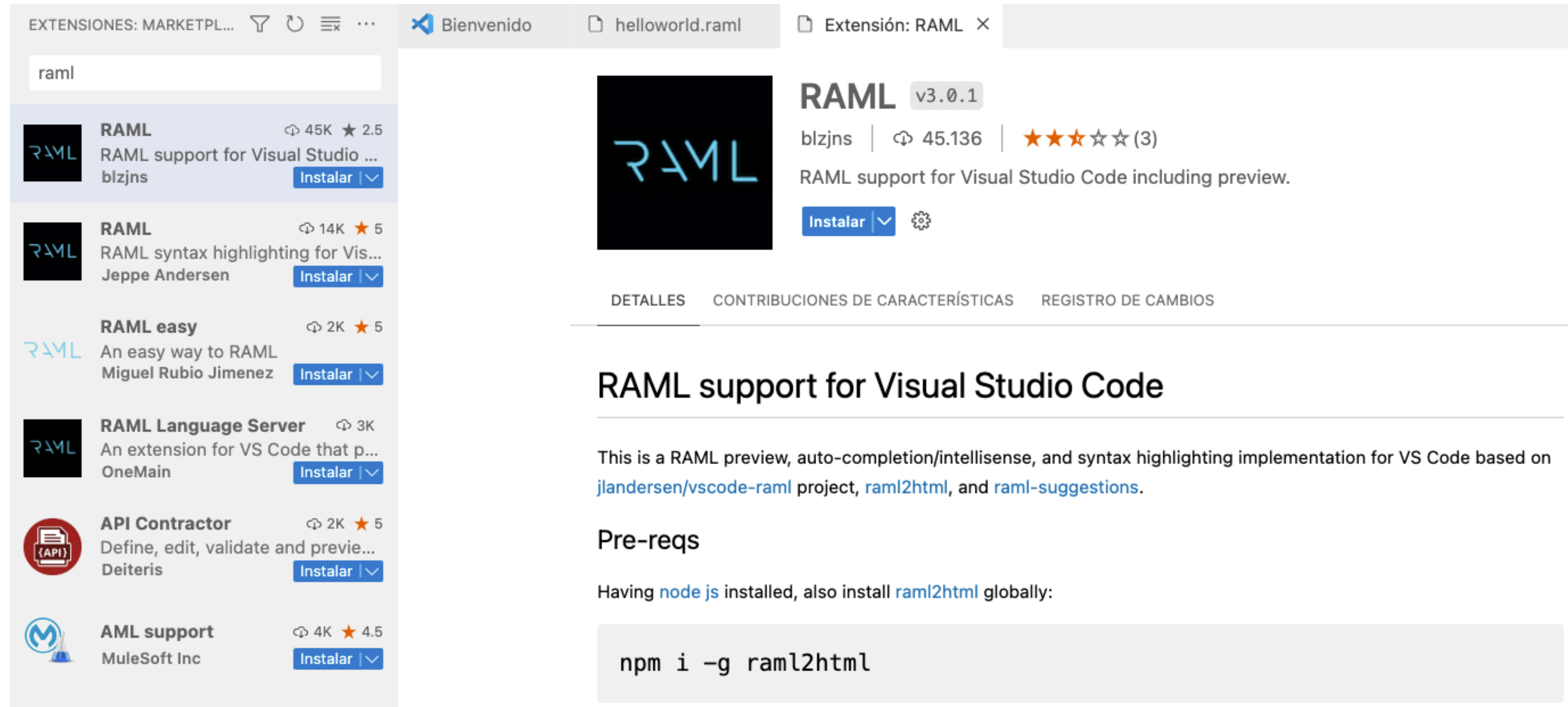
Lenguaje

- **RAML 1.0** es la versión más actual del lenguaje, de tal manera que son documentos compatibles con YAML 1.2 que comienzan con una línea de comentarios YAML requerida que indica la versión RAML, de la siguiente manera:

```
#%RAML 1.0
title: My API
```

Trabajando con un editor

- Podemos trabajar con el editor que más nos guste, por ejemplo, **Visual Studio Code** tiene un **plugin** específico para trabajar en estos entornos:



The screenshot shows the Visual Studio Code interface. On the left, the 'EXTENSIONES: MARKETPL...' sidebar is open, displaying a search for 'raml'. Several extensions are listed, including 'RAML' by blzjns, 'RAML syntax highlighting for Vis...' by Jeppe Andersen, 'RAML easy' by Miguel Rubio Jimenez, 'RAML Language Server' by OneMain, 'API Contractor' by Deiteris, and 'AML support' by MuleSoft Inc. The main editor area shows the 'RAML' extension details for version 3.0.1 by blzjns, with 45,136 downloads and a 3-star rating. The extension description states: 'RAML support for Visual Studio Code including preview.' Below the details, there are tabs for 'DETALLES', 'CONTRIBUCIONES DE CARACTERÍSTICAS', and 'REGISTRO DE CAMBIOS'. The 'DETALLES' tab is active, showing the title 'RAML support for Visual Studio Code' and a description: 'This is a RAML preview, auto-completion/intellisense, and syntax highlighting implementation for VS Code based on [jlandersen/vscode-raml](#) project, [raml2html](#), and [raml-suggestions](#).' Under the 'Pre-reqs' section, it says: 'Having [node js](#) installed, also install [raml2html](#) globally:'. A code block at the bottom shows the command: `npm i -g raml2html`.

La especificación

- Se encuentra en la siguiente url:
- <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/>

RAML Version 1.0: RESTful API Modeling Language

Abstract

RAML is a language for the definition of HTTP-based APIs that embody most or all of the principles of Representational State Transfer (REST). This document constitutes the RAML specification, an application of the [YAML 1.2 specification](#). The RAML specification provides mechanisms for defining practically-RESTful APIs, creating client/server source code, and comprehensively documenting the APIs for users.

Elementos raíz

- ▶ Introducción.
- ▶ Documentación del usuario.
- ▶ URL y parámetros base.
- ▶ Protocolos.
- ▶ Más información.

Introducción

- ▶ Primero, debes introducir **información básica** en un editor de texto. Puede guardar la definición RAML de su API como un archivo de texto con una extensión recomendada .raml:

```
#%RAML 1.0
---
title: e-BookMobile API
baseUri: http://api.e-bookmobile.com/{version}
version: v1
```

- ▶ Todo lo que se inserta en la **raíz (o en la parte superior) de la especificación** se aplica al resto de su API.
- ▶ Esto será **muy útil** más adelante a medida que descubra patrones en la forma en que construye su API.
- ▶ La clave **baseURI** que elija se usará con **cada llamada realizada**, así que asegúrese de que sea lo más limpio y conciso posible.

Documentación del usuario

User Documentation

The OPTIONAL **documentation** node includes a variety of documents that serve as user guides and reference documentation for the API. Such documents can clarify how the API works or provide technical and business context.

The value of the documentation node **MUST** be a sequence of one or more documents. Each document is a map that **MUST** have exactly two key-value pairs described in the following table:

| Name | Description |
|---------|---|
| title | Title of the document. Its value MUST be a non-empty string. |
| content | Content of the document. Its value MUST be a non-empty string and MAY be formatted using markdown . |

Documentación del usuario

► Ejemplo:

```
#%RAML 1.0
title: ZEncoder API
baseUri: https://app.zencoder.com/api
documentation:
  - title: Home
    content: |
      Welcome to the _Zencoder API_ Documentation. The _Zencoder API_
      allows you to connect your application to our encoding service
      and encode videos without going through the web interface. You
      may also benefit from one of our
      [integration libraries](https://app.zencoder.com/docs/faq/basics/libraries)
      for different languages.
  - title: Legal
    content: !include docs/legal.markdown
```


URL y parámetros base

- ▶ Ejemplo trabajando con **baseUri**:

```
#%RAML 1.0
title: Salesforce Chatter REST API
version: v28.0
baseUri: https://na1.salesforce.com/services/data/{version}/chatter
```

- ▶ Ejemplo trabajando con **parámetros base**:

```
#%RAML 1.0
title: Amazon S3 REST API
version: 1
baseUri: https://{bucketName}.s3.amazonaws.com
baseUriParameters:
  bucketName:
    description: The name of the bucket
```

Protocolos

- ▶ El nodo protocols es OPCIONAL y especifica los protocolos que admite una API.
- ▶ Si protocols no se especifica explícitamente, SE DEBERÁN utilizar uno o más protocolos incluidos en el nodo baseUrl. Si el nodo protocols se especifica explícitamente, dicha especificación de nodo DEBERÁ anular cualquier protocolo incluido en el nodo baseUrl.
- ▶ El nodo protocols DEBE ser una **matriz no vacía de cadenas**, de valores HTTP y/o HTTPS, y no distingue entre mayúsculas y minúsculas.
- ▶ El siguiente es un ejemplo acepta solicitudes HTTP y HTTPS.

```
#%RAML 1.0
title: Salesforce Chatter REST API
version: v28.0
protocols: [ HTTP, HTTPS ]
baseUrl: https://na1.salesforce.com/services/data/{version}/chatter
```

Más información

- ▶ <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/#the-root-of-the-document>

raml-spec / versions / raml-10 / raml-10.md
↑ Top

Preview

Code

Blame

3793 lines (3047 loc) · 173 KB

Raw

The Root of the Document [↗](#)

The root section of the RAML document describes the basic information about an API, such as its title and version. The root section also defines assets used elsewhere in the RAML document, such as types and traits.

Nodes in a RAML-documented API definition MAY appear in any order. Processors MUST preserve the order of nodes of the same kind within the same node of the definition tree. Examples of such nodes are resources that appear at the same level of the resource tree, methods for a given resource, parameters for a given method, and nodes at the same level in a given type. Processors

Recursos

- Introducción.
- Definición de recurso.
- Recursos anidados.
- Más información.

Introducción

- ▶ Un **recurso** se identifica por su **URI relativa**, que DEBE comenzar con (/).
- ▶ Cada nodo cuya clave **comienza con /** y está en la raíz de la definición de API o es el nodo **secundario de un nodo recurso**, es dicho nodo recurso.
- ▶ Un **recurso definido** como un nodo de nivel **raíz** se denomina **recurso de nivel superior**. La clave del nodo de nivel raíz es el URI del recurso relativo al baseUri, si lo hay.
- ▶ Un recurso definido como **nodo secundario** de otro recurso se denomina **recurso anidado**.
- ▶ La clave del nodo secundario es el URI del recurso anidado en relación con el URI del recurso principal.

Introducción

- ▶ Este ejemplo muestra una definición de API con un recurso de nivel superior /gists, y un recurso anidado, /public.

```
#%RAML 1.0
title: GitHub API
version: v3
baseUri: https://api.github.com
/gists:
  displayName: Gists
  /public:
    displayName: Public Gists
```

Definición de recurso

- ▶ Los **recursos** se definen tal que así:

```
/users:  
  /authors:  
    /books:
```

- ▶ Tenga en cuenta que **todos estos recursos** comienzan con una barra (/).
- ▶ En RAML, así es como **se define un recurso**. Todos los métodos y parámetros anidados bajo estos recursos de nivel superior pertenecen y actúan sobre ese recurso.
- ▶ Ahora, dado que cada uno de estos recursos es una **colección de objetos individuales** (autores, libros y usuarios específicos), necesitaremos definir algunos **subrecursos** para completar la colección.

Recursos anidados

- ▶ Los **recursos anidados** son **útiles** cuando desea llamar a un **subconjunto particular** de su recurso para restringirlo. Por ejemplo:

```
/authors:  
  /{authorname}:
```

- ▶ Esto permite que el consumidor de la API interactúe con **el recurso clave** y sus **recursos anidados**.
- ▶ Por ejemplo, una solicitud GET a http://api.e-bookmobile.com/authors/Mary_Roach devuelve detalles sobre la escritora científica y humorista **Mary Roach**.

Recursos

- ▶ En el siguiente ejemplo vamos a modelar en REST un **listado de películas** y queremos definir un filtro opcional (**queryParams**) por el año en que salió la película.
- ▶ A continuación esperamos que el servidor nos conteste con **un array de películas** que cumplen con ese criterio de búsqueda.
- ▶ Técnicamente hablando definimos un recurso que se llamará **/films** el cuál podríamos invocar de esta manera:
 - ▶ GET /films?year=

/films:

description: |

Manage Films

get:

description: |

Service for get films.

queryParameters:

year:

description: Filters the film by release year.

type: string

example: «2009»

required: false

responses:

200:

body:

application/json:

type: object

properties:

data:

type: films.films

required: false

Más información

- ▶ <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/#resources-and-nested-resources>

Resources and Nested Resources

A resource is identified by its relative URI, which MUST begin with a slash (`/`). Every node whose key begins with a slash, and is either at the root of the API definition or is the child node of a resource node, is such a resource node.

A resource defined as a root-level node is called a top-level resource. The key of the root-level node is the URI of the resource relative to the baseUri if there is one. A resource defined as a child node of another resource is called a nested resource. The key of the child node is the URI of the nested resource relative to the parent resource URI.

This example shows an API definition with one top-level resource, `/gists` , and one nested resource, `/public` .

Métodos

- ▶ Los **métodos de API RESTful** son operaciones que se realizan en un recurso.
- ▶ Las **propiedades opcionales get, patch, put, post, delete, head, y options** definen sus métodos.
- ▶ Estas propiedades corresponden a los métodos HTTP definidos en la especificación RFC2616 de la versión 1.1 de HTTP y su extensión, RFC5789. El valor de estas propiedades de método DEBE ser un mapa con los siguientes pares clave-valor:

Métodos

- Los **métodos http** cuelgan de cada **recurso, subrecurso o URI Parameter** como se muestra a continuación:

```
/books:
```

```
  get:
```

```
  post:
```

```
  /{id}:
```

```
    get:
```

```
    put:
```

```
    delete:
```

```
  /chapters:
```

```
    get:
```

Métodos

- ▶ Más información:
- ▶ <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/#methods>

Methods

RESTful API methods are operations that are performed on a resource. The OPTIONAL properties **get**, **patch**, **put**, **post**, **delete**, **head**, and **options** of a resource define its methods. These properties correspond to the HTTP methods defined in the HTTP version 1.1 specification [RFC2616](#) and its extension, [RFC5789](#). The value of these method properties MUST be a map with the following key-value pairs:

| Name | Description |
|--------------|---|
| displayName? | An alternate, human-friendly method name in the context of the resource. If the displayName node is not defined for a method, documentation tools SHOULD refer to the resource by its key, which acts as the method name. |

Parámetros

- Introducción.
- Parámetros en la url.
- Parámetros de consulta.

Introducción

- Los recursos que definimos son colecciones de objetos más **pequeños y relevantes**.
- Usted, como diseñador de API, se ha dado cuenta de que los desarrolladores probablemente querrán actuar sobre estos objetos de forma **más granular** y por eso puede introducir el concepto de parámetro.

Parámetros en la url

- ▶ Por ejemplo. Este es un parámetro URI, indicado por llaves alrededor en RAML:

```
/books:  
  /{bookTitle}:
```

- ▶ Por lo tanto, para realizar una solicitud a este recurso anidado, la URI del libro de Mary Roach, Stiff se vería como:
 - ▶ <http://api.e-bookmobile.com/v1/books/Stiff>

Parámetros en la url

► Ejemplo:

```
/books:  
  get:  
  put:  
  post:  
  /{bookTitle}:  
    get:  
    put:  
    delete:  
  /author:  
    get:  
  /publisher:  
    get:
```

Parámetros de consulta

- El **nodo queryParameters** nos permite definir parámetros de consulta.
- Ejemplo: parámetro **page**.

```
/users:  
  get:  
    description: Get a list of users  
    queryParameters:  
      page:  
        description: Specify the page that you want to retrieve  
        type: integer  
        required: true  
        example: 1
```

Parámetros de consulta

► Ejemplo:

...

per_page:

description: Specify the amount of items that will be retrieved per page

type: integer

minimum: 10

maximum: 200

default: 30

example: 50

Cuerpo

- ▶ El **cuerpo de la solicitud HTTP** para un método PUEDE especificarse utilizando el **nodo body** OPCIONAL. Por ejemplo, para crear un **recurso mediante POST o PUT**, el cuerpo de la solicitud normalmente incluiría **los detalles del recurso que se va a crear**.
- ▶ El valor del nodo del cuerpo es una "declaración del cuerpo". Generalmente, la declaración del cuerpo DEBE ser un **mapa cuyos nombres clave sean los tipos de medios válidos** del cuerpo de la solicitud.
- ▶ Cada **nombre de clave** DEBE ser una cadena de tipo media que cumpla con la especificación de tipo de medio en RFC6838.
- ▶ Los valores son la declaración de tipo de datos correspondiente o el nombre del tipo de datos que describe el cuerpo de la solicitud.
- ▶ Alternativamente, si se han declarado tipos media por defecto en la raíz de la API, entonces la declaración del cuerpo PUEDE consistir solo en la declaración del tipo de datos o el nombre del tipo de datos que describe el cuerpo de la solicitud para ese tipo de medio.

Cuerpo

- El siguiente ejemplo ilustra varias combinaciones de tipos de medios predeterminados y no predeterminados, y declaraciones y referencias de tipos de datos.

```
/users:  
  post:  
    body:  
      type: User  
/groups:  
  post:  
    body:  
      application/json:  
        properties:  
          groupName:  
            deptCode:  
              type: number  
      text/xml:  
        type: !include schemas/group.xsd
```

Respuestas

- Introducción.
- Declaración de la respuesta.

Introducción

- ▶ Las secciones de recursos y métodos describen las solicitudes HTTP. Esta sección describe las **respuestas HTTP** a las invocaciones de métodos en recursos.
- ▶ El nodo **responses** de un método en un recurso describe **las posibles respuestas** al invocar ese método en ese recurso.
- ▶ El **valor de las respuestas** DEBE ser un mapa donde cada **nombre de clave** represente un posible **código de estado HTTP** para ese método en ese recurso. Los valores describen las respuestas correspondientes. Cada valor DEBE ser una declaración de respuesta.
- ▶ Las **claves** debe ser **numéricas**, por ejemplo 200 o 204. Los procesadores deben tratar estas claves numéricas como **claves de cadena en todas las situaciones**. Por ejemplo, '200' y 200 DEBEN tratarse como claves duplicadas y, por lo tanto, no se permiten simultáneamente.

Introducción

- ▶ La definición de las **respuestas** debe venir acompañada del **código http** y del **formato** en el que vendrán los datos, generalmente las respuestas vienen en formato **'application/json'**.
- ▶ La invocación al endPoint puede devolver un error de tipo 404 'Recurso no encontrado'. Ejemplo:

```
...
get:
  description: Get a Book by id
  responses:
    200:
      body:
        application/json:
          type: book.book
    404:
      description: Not found
```

Declaración de la respuesta

- El **valor de la declaración de una respuesta** debe ser un mapa con los siguientes pares clave-valor:

| Name | Description |
|--------------------|---|
| description? | A substantial, human-friendly description of a response. Its value MUST be a string and MAY be formatted using Markdown . |
| (<annotationName>) | Annotations to be applied to this API. An annotation MUST be a map having a key that begins with "(" and ends with ")", where the text enclosed in parentheses is the annotation name and the value is an instance of that annotation. |
| headers? | Detailed information about any response headers returned by this method |
| body? | The body of the response |

Declaración de la respuesta

- ▶ En este caso se define la respuesta cuando todo va bien:

```
/invoices:  
  get:  
    responses:  
    200:  
      body:  
        type: Invoice  
        properties:  
          id: number
```

Declaración de la respuesta

► Otro ejemplo:

```
post:
  body:
    type: Invoice
  responses:
    201:
      headers:
        Location:
          example: /invoices/45612
      body:
        application/json:
          type: !include schemas/invoice.json
        text/xml:
          type: !include schemas/invoice.xsd
```

Declaración de la respuesta

- ▶ Otro ejemplo (cont.):

```
post:  
  ...  
  422:  
    body:  
      properties:  
        error:  
      example:  
        error: Amount cannot be negative
```

El modelo de entidades

- Introducción.
- Definición de modelo de datos.
- Tipos de datos.
- Herencia.

Introducción

- ▶ RAML usa estructuras llamadas **Types** para especificar el **modelo de entidades** a utilizar dentro de nuestras APIs.
- ▶ Es un **modelo sencillo** que permite definir entidades y más fácil de utilizar que otros modelos como por ejemplo **JSON Schema**.

Definición de modelo de datos

- Cada tipo trabaja con dos propiedades: **type** y **properties**.
- Se define de la siguiente forma:

types:

User:

type: object

properties:

firstName: string

lastName: string

age:

type: integer

minimum: 0

maximum: 125

Tipos de datos

- ▶ RAML define diversos **tipos de datos**, parecidos a los que se definen en lenguajes de programación como Java:
 - ▶ **object**.
 - ▶ **array**.
 - ▶ **union**.
 - ▶ tipos escalares:
 - ▶ number, boolean, string, date-only, time-only, datetime-only, datetime, o integer.

Tipos de datos

- Los **'facets'** son configuraciones especiales.
- Algunos de los facets permitidos son:
 - properties, minProperties, maxProperties, additionalProperties, discriminator, y discriminatorValue.
- Solo los objetos pueden declarar el facet **'properties'**.

Tipos de datos

- ▶ **Type** puede tener **atributos sencillos** como parte de su estructura o tener **atributos más complejos** (anidados).
- ▶ El carácter '?' que sigue al nombre de una propiedad declara que la **propiedad opcional**.

```
types:  
  Foo:  
    properties:  
      id: integer  
      name: string  
      ownerName?: string  
  Error:  
    properties:  
      code: integer  
      message: string
```

Tipos de datos

- El siguiente ejemplo muestra el atributo '**status**' que incluye más atributos:

```
types:
  card:
    type: object
    properties:
      alias:
        type: string
        description: |
          Card alias. This attribute allows customers to assign a custom name to their cards.
        required: false
```

Tipos de datos

► (cont.):

status:

type: object

description: |

Card current status.

required: false

properties:

id:

type: string

enum: [INOPERATIVE, BLOCKED, PENDING_EMBOSSING, PENDING_DELIVERY, PRE_ACTIVATED]

description: |

Card status identifier.

reason:

type: string

description: |

Reason of the state of the card.

required: false

Herencia

- ▶ En RAML 1.0 se permite la **Herencia**.
- ▶ Por ejemplo, se puede tener un tipo Persona y otro tipo Profesor que herede algunas de sus atributos:

```
types:  
  person:  
    type: object  
    properties:  
      name: string  
  teacher:  
    type: Person  
    properties:  
      level: string
```

Eliminando la redundancia

- ▶ Introducción.
- ▶ Nuestra API.
- ▶ Identificando patrones.
- ▶ Resource Types.
- ▶ Extracción de Resource Type para colecciones.
- ▶ Extracción de Resource Type para un único elemento.
- ▶ Traits.

Introducción

- ▶ Anteriormente, hemos presentado el lenguaje de modelado API RESTful y creamos una definición de API simple basada en una única entidad.
- ▶ Ahora imagine una API del mundo real en la que tiene varios recursos de tipo entidad, todos con operaciones GET, POST, PUT y DELETE iguales o similares.
- ▶ Puede ver cómo la **documentación** de su API puede volverse **rápidamente tediosa y repetitiva**.
- ▶ Ahora, vamos a mostrar **cómo el uso de tipos de recursos y traits** en RAML puede eliminar **redundancias en las definiciones de recursos** y métodos al extraer y parametrizar secciones comunes, eliminando así **errores de copiar y pegar** y haciendo que las **definiciones de API** sean más **concisas**.

Nuestra API

- Para demostrar los **beneficios de los tipos de recursos y traits**, expandiremos nuestra API original agregando recursos para un segundo tipo de entidad llamado Bar. Estos son los recursos que conformarán nuestra API revisada:

```
GET /api/v1/foos
POST /api/v1/foos
GET /api/v1/foos/{foold}
PUT /api/v1/foos/{foold}
DELETE /api/v1/foos/{foold}
GET /api/v1/foos/name/{name}
GET /api/v1/foos?name={name}&ownerName={ownerName}
GET /api/v1/bars
POST /api/v1/bars
GET /api/v1/bars/{barId}
PUT /api/v1/bars/{barId}
DELETE /api/v1/bars/{barId}
GET /api/v1/bars/foold/{foold}
```

Identificando patrones

- ▶ A medida que leemos la **lista de recursos de nuestra API**, comenzamos a ver que surgen algunos **patrones**.
- ▶ Por ejemplo, existe un **patrón para las URIs** y los **métodos** utilizados para **crear, leer, actualizar y eliminar entidades individuales**, y existe un patrón para las URIs y los métodos utilizados para recuperar **colecciones de entidades**.
- ▶ El **patrón de colección** y de un único **elemento** son dos de los **patrones más comunes** utilizados para extraer tipos de recursos en definiciones RAML.

Identificando patrones

- Veamos un par de secciones de nuestra API:

```

/foos:
  get:
    description: |
      List all foos matching query criteria, if provided;
      otherwise list all foos
    queryParameters:
      name?: string
      ownerName?: string
    responses:
      200:
        body:
          application/json:
            type: Foo[]

```

Identificando patrones

- Veamos un par de secciones de nuestra API (cont.):

post:

description: Create a new foo

body:

application/json:

type: Foo

responses:

201:

body:

application/json:

type: Foo

Identificando patrones

- Para bar es exactamente igual:

```
/bars:  
get:  
  description: |  
    List all bars matching query criteria, if provided;  
    otherwise list all bars  
  queryParameters:  
    name?: string  
    ownerName?: string  
  responses:  
    200:  
      body:  
        application/json:  
          type: Bar[]
```

Identificando patrones

- ▶ Veamos un par de secciones de nuestra API (cont.):

```
post:
```

```
  description: Create a new bar
```

```
  body:
```

```
    application/json:
```

```
      type: Bar
```

```
  responses:
```

```
    201:
```

```
      body:
```

```
        application/json:
```

```
          type: Bar
```

Identificando patrones

- ▶ Cuando **comparamos las definiciones RAML** de los recursos /foos y /bars, incluidos los métodos HTTP utilizados, podemos ver varias redundancias entre las diversas propiedades de cada uno, y nuevamente vemos que **comienzan a surgir patrones**.
- ▶ Siempre que haya un patrón en la definición de un recurso o método, existe la oportunidad de utilizar **resource type o trait** RAML.

Resource Types

- Introducción.
- Parámetros reservados.
- Parámetros definidos por el usuario.
- Parameter Functions.

Introducción

- Para implementar los patrones que se encuentran en la API, los tipos de recursos utilizan **parámetros reservados y definidos por el usuario** rodeados por **corchetes angulares dobles** (<< y >>).

Parámetros reservados

- ▶ Se pueden utilizar **dos parámetros reservados** en las definiciones de tipos de recursos:
 - ▶ **<<resourcePath>>** representa la URI completa (después del URI base), y
 - ▶ **<<resourcePathName>>** representa la parte del URI que sigue a la barra inclinada más a la derecha (/), ignorando las llaves { }.
- ▶ Cuando se procesan dentro de una definición de recurso, sus valores se calculan en función del recurso que se define.
- ▶ Dado el recurso /foos, por ejemplo, <<resourcePath>> se evaluaría como "/foos" y <<resourcePathName>> se evaluaría como "foos".
- ▶ Dado el recurso /foos/{foold}, <<resourcePath>> se evaluaría como "/foos/{foold}" y <<resourcePathName>> se evaluaría como "foos".

Parámetros definidos por el usuario

- ▶ Una **definición de tipo de recurso** también puede contener **parámetros definidos por el usuario**.
- ▶ A diferencia de los parámetros reservados, cuyos valores se determinan dinámicamente en función del recurso que se define, a los **parámetros definidos por el usuario** se les deben **asignar valores** dondequiera que se utilice el tipo de recurso que los contiene, y **esos valores no cambian**.
- ▶ Los **parámetros definidos** por el usuario pueden declararse al **comienzo de la definición de un tipo de recurso**, aunque hacerlo **no es obligatorio** y no es una práctica común, ya que el lector normalmente puede deducir su uso previsto dados sus nombres y los contextos en los que se utilizan.

Parameter Functions

- ▶ Hay un **conjunto de funciones de texto útiles** disponibles para su uso siempre que se utilice un parámetro para transformar el valor expandido del parámetro cuando se procesa en una definición de recurso.
- ▶ Estas son las funciones disponibles para la transformación de parámetros:
 - ▶ !singularize y !pluralize
 - ▶ !uppercase y !lowercase
 - ▶ !uppercamelcase y !lowercamelcase
 - ▶ !upperunderscorecase y !lowerunderscorecase
 - ▶ !upperhyphencase y !lowerhyphencase

Parameter Functions

- ▶ Las funciones se aplican a un parámetro mediante la siguiente construcción:

```
<<parameterName | !functionName>>
```

- ▶ Si necesita usar más de una función para lograr la transformación deseada, debe separar cada nombre de función con el símbolo de barra vertical ("|") y anteponer un signo de exclamación (!) antes de cada función utilizada.
- ▶ Por ejemplo, dado el recurso /foos, donde <<resourcePathName>> se evalúa como "foos":

```
<<resourcePathName | !singularize>> ==> "foo"
<<resourcePathName | !uppercase>> ==> "FOOS"
<<resourcePathName | !singularize | !uppercase>> ==> "FOO"
```

- ▶ Y dado el recurso /bars/{barId}, donde <<resourcePathName>> se evalúa como "barras":

```
<<resourcePathName | !uppercase>> ==> "BARS"
<<resourcePathName | !uppercamelcase>> ==> "Bar"
```

Extracción de Resource Type para colecciones

- ▶ Refactoricemos las definiciones de recursos /foos y /bars que se muestran arriba, usando un tipo de recurso para capturar las propiedades comunes.
- ▶ Usaremos el parámetro reservado **<<resourcePathName>>** y el parámetro definido por el usuario **<<typeName>>** para representar el tipo de datos utilizado.

resourceTypes:

collection:

usage: Use this resourceType to represent any collection of items

description: A collection of <<resourcePathName>>

get:

description: Get all <<resourcePathName>>, optionally filtered

responses:

200:

body:

application/json:

type: <<typeName>>[]

post:

description: Create a new <<resourcePathName|!singularize>>

responses:

201:

body:

application/json:

type: <<typeName>>

Extracción de Resource Type para colecciones

- Tenga en cuenta que en nuestra API, debido a que **nuestros tipos de datos** son simplemente versiones **en singular** y en **mayúscula** de los nombres de nuestros recursos base, podríamos haber aplicado **funciones al parámetro <<resourcePathName>>**, en lugar de introducir el parámetro **<<typeName>>** definido por el usuario, para lograr el mismo resultado para esta parte de la API:

```
resourceTypes:  
  collection:  
    ...  
    get:  
      ...  
      type: <<resourcePathName|!singularize|!uppercamelcase>>[]  
    post:  
      ...  
      type: <<resourcePathName|!singularize|!uppercamelcase>>
```


Extracción de Resource Type para colecciones

- Usando la definición anterior que incorpora el parámetro <<typeName>>, así es como aplicaría el tipo de recurso **collection** a los recursos /foos y /bars:

```

/foos:
  type: { collection: { "typeName": "Foo" } }
get:
  queryParameters:
    name?: string
    ownerName?: string
...
/bars:
  type: { collection: { "typeName": "Bar" } }

```

Extracción de Resource Type para un único elemento

- ▶ Centrémonos ahora en la parte de nuestra API que trata con **elementos individuales** de una colección: los recursos `/foos/{foold}` y `/bars/{barId}`.
- ▶ Aquí está el código para `/foos/{foold}`:

```

/foos:
...
/{foold}:
  get:
    description: Get a Foo
    responses:
      200:
        body:
          application/json:
            type: Foo
      404:
        body:
          application/json:
            type: Error
            example: !include examples/Error.json

```

```
/foos:  
...  
  put:  
    description: Update a Foo  
    body:  
      application/json:  
        type: Foo  
    responses:  
      200:  
        body:  
          application/json:  
            type: Foo  
      404:  
        body:  
          application/json:  
            type: Error  
            example: !include examples/Error.json
```

```
/foos:  
...  
delete:  
  description: Delete a Foo  
  responses:  
    204:  
    404:  
      body:  
        application/json:  
          type: Error  
          example: !include examples/Error.json
```

Extracción de Resource Type para un único elemento

- ▶ La definición de recurso **/bars/{barId}** también tiene métodos GET, PUT y DELETE y es idéntica a la definición **/foos/{foold}**, excepto por las apariciones de las cadenas "foo" y "bar" (y sus respectivos plurales y/o en mayúsculas).
- ▶ Extrayendo el **patrón que acabamos de identificar**, así es como definimos un tipo de recurso para elementos individuales de una colección:

resourceTypes:

...

item:

usage: Use this resourceType to represent any single item

description: A single <<typeName>>

get:

description: Get a <<typeName>>

responses:

200:

body:

application/json:

type: <<typeName>>

404:

body:

application/json:

type: Error

example: !include examples/Error.json

```
resourceTypes:
...
item:
  put:
    description: Update a <<typeName>>
    body:
      application/json:
        type: <<typeName>>
    responses:
      200:
        body:
          application/json:
            type: <<typeName>>
      404:
        body:
          application/json:
            type: Error
            example: !include examples/Error.json
```



```
resourceTypes:  
...  
item:  
  delete:  
    description: Delete a <<typeName>>  
    responses:  
      204:  
      404:  
        body:  
          application/json:  
            type: Error  
            example: !include examples/Error.json
```

Extracción de Resource Type para un único elemento

- Y así es como aplicamos el **tipo de recurso item**:

```
/foos:
...
/{foold}:
  type: { item: { "typeName": "Foo" } }
```

```
...
/bars:
...
/{barId}:
  type: { item: { "typeName": "Bar" } }
```

Traits

- ▶ Mientras que **un tipo de recurso** se utiliza para **extraer patrones de definiciones de recursos**, un trait o rasgo se utiliza para **extraer patrones de definiciones de métodos** que son comunes a todos los recursos.
- ▶ Junto con **<<resourcePath>>** y **<<resourcePathName>>**, hay un **parámetro reservado adicional** disponible para su uso en definiciones de trait:
 - ▶ **<<methodName>>**
- ▶ Evalúa el método HTTP (GET, POST, PUT, DELETE, etc.) para **el cuál trait está definido**.
- ▶ Los **parámetros definidos por el usuario** también pueden aparecer dentro de **una definición de rasgo** y, cuando se aplican, adquieren el valor del recurso en el que se aplican.

Traits

- ▶ Observe que **el tipo de recurso item** todavía está lleno de redundancias.
- ▶ Veamos cómo los traits pueden ayudar a eliminarlos.
- ▶ Comenzaremos extrayendo un rasgo para cualquier método que contenga un cuerpo de solicitud:

```
traits:  
  hasRequestItem:  
    body:  
      application/json:  
        type: <<typeName>>
```

Traits

- ▶ Ahora extraigamos traits de métodos cuyas respuestas normales contienen cuerpos:

```
hasResponseItem:  
  responses:  
    200:  
      body:  
        application/json:  
          type: <<typeName>>  
hasResponseCollection:  
  responses:  
    200:  
      body:  
        application/json:  
          type: <<typeName>>[]
```

Traits

- ▶ Finalmente, aquí hay una característica para cualquier método que pueda devolver una respuesta de error 404:

```
hasNotFound:  
  responses:  
    404:  
      body:  
        application/json:  
          type: Error  
          example: !include examples/Error.json
```

- ▶ Luego aplicamos este rasgo a nuestros tipos de recursos:

```
resourceTypes:
  collection:
    usage: Use this resourceType to represent any collection of items
    description: A collection of <<resourcePathName|!uppercamelcase>>
    get:
      description: |
        Get all <<resourcePathName|!uppercamelcase>>,
        optionally filtered
      is: [ hasResponseCollection: { typeName: <<typeName>> } ]
    post:
      description: Create a new <<resourcePathName|!singularize>>
      is: [ hasRequestItem: { typeName: <<typeName>> } ]
  item:
    usage: Use this resourceType to represent any single item
    description: A single <<typeName>>
    get:
      description: Get a <<typeName>>
      is: [ hasResponseItem: { typeName: <<typeName>> }, hasNotFound ]
```

```
resourceTypes:  
  collection:  
    ...  
  put:  
    description: Update a <<typeName>>  
    is: | [ hasRequestItem: { typeName: <<typeName>> }, hasResponseItem: { typeName: <<typeName>> }, hasNotFound ]  
  delete:  
    description: Delete a <<typeName>>  
    is: [ hasNotFound ]  
  responses:  
    204:
```


Traits

- ▶ También podemos aplicar traits a **métodos definidos dentro de los recursos**.
- ▶ Esto es especialmente útil para **escenarios "únicos"** donde una combinación de recurso y método coincide con uno o más rasgos pero no coincide con ningún tipo de recurso definido:

```
/foos:
...
/name/{name}:
  get:
    description: List all foos with a certain name
    is: [ hasResponseCollection: { typeName: Foo } ]
```

Seguridad

- La seguridad también se define en el nivel raíz del archivo .raml. Así que agreguemos nuestra definición de esquema de seguridad básico HTTP.

```
securitySchemes:
  basicAuth:
    description: Each request must contain the headers necessary for
      basic authentication
    type: Basic Authentication
    describedBy:
      headers:
        Authorization:
          description: Used to send the Base64-encoded "username:password"
            credentials
          type: string
      responses:
        401:
          description: |
            Unauthorized. Either the provided username and password
            combination is invalid, or the user is not allowed to access
            the content provided by the requested URL.
```

Modularización

- ▶ Introducción.
- ▶ Includes.
- ▶ Bibliotecas.
- ▶ Overlays y extensiones.

Introducción

- ▶ RAML proporciona varios mecanismos para ayudar a modularizar el ecosistema de una especificación API:
 - ▶ Include.
 - ▶ Bibliotecas.
 - ▶ Overlays.
 - ▶ Extensiones.

Includes

- ▶ Los procesadores RAML deben admitir **la etiqueta OPCIONAL !include**, que especifica la inclusión de archivos externos en la especificación API.
- ▶ Al ser una **etiqueta YAML**, se requiere como prefijo **el signo de exclamación (!)**.
- ▶ Cuando especificamos una API, la **etiqueta !include** se encuentra solo en una **posición de valor de nodo**.
- ▶ La etiqueta !include DEBE ser el valor de un nodo, que asigna el contenido del archivo nombrado por la etiqueta !include al valor del nodo. Por lo tanto, la etiqueta !include no se puede utilizar en ninguna expresión de tipo ni en herencia múltiple.

Includes

- ▶ En el siguiente ejemplo, el conjunto de tipos que se utilizarán en la especificación se recupera de un archivo llamado **myTypes.raml** y se utiliza como valor del nodo types.

```
#%RAML 1.0
title: My API with Types
types: !include myTypes.raml
```

- ▶ La **etiqueta !include** acepta un único argumento, la ubicación del contenido que se incluirá, que DEBE especificarse explícitamente.

Includes

- El valor del argumento DEBE ser una ruta o URL como se describe en la siguiente tabla:

| Argument | Description | Examples |
|---------------|---|---|
| absolute path | A path that begins with a single slash (/) and is interpreted relative to the root RAML file location. | /traits/pageable.raml |
| relative path | A path that neither begins with a single slash (/) nor constitutes a URL, and is interpreted relative to the location of the included file. | description.md ../traits/pageable.raml |
| URL | An absolute URL | http://dev.domain.com/api/patterns/traits.raml |

Includes

- ▶ Si un **archivo** comienza con una **línea de identificador de fragmento RAML** y el identificador de fragmento no es una **biblioteca**, overlay o extensión, el contenido del archivo después de eliminar la línea de identificador de fragmento RAML y cualquier nodo de uso DEBE ser **estructuralmente válido** de acuerdo con la especificación RAML.
- ▶ Por ejemplo, un archivo RAML que comienza con **#%RAML 1.0 Trait** debe tener la estructura de una declaración trait RAML como se define en la sección Resource types y trait. Incluir el archivo en una ubicación correcta para una declaración de rasgos da como resultado un archivo RAML válido.

Includes

- El **siguiente ejemplo** muestra un archivo de fragmento RAML que define un **tipo de recurso** y un archivo que incluye este archivo de fragmento escrito.

```
#%RAML 1.0 ResourceType
```

```
#This file is located at resourceTypes/collection.raml
```

```
description: A collection resource
```

```
usage: Use this to describe a resource that lists items
```

```
get:
```

```
  description: Retrieve all items
```

```
post:
```

```
  description: Add an item
```

```
responses:
```

```
  201:
```

```
    headers:
```

```
      Location:
```

Includes

- Aquí se muestra el fichero que usa:

```
#%RAML 1.0
title: Products API
resourceTypes:
  collection: !include resourceTypes/collection.raml
/products:
  type: collection
  description: All products
```

Includes

► Otro ejemplo:

```
#%RAML 1.0
title: Baeldung Foo REST Services API
...
types: !include /types/allDataTypes.raml
resourceTypes: !include allResourceTypes.raml
traits: !include http://foo.com/docs/allTraits.raml
```

Includes

- ▶ En lugar de colocar **todos los tipos, tipos de recursos o características** en sus respectivos archivos de inclusión, también puede usar **tipos especiales de inclusiones** conocidos como **fragmentos tipados** para dividir cada una de estas construcciones en **múltiples archivos de inclusión**, especificando un archivo diferente para cada **type, resource type o trait**.
- ▶ También puede utilizar fragmentos escritos para definir elementos de documentación del usuario, ejemplos con nombre, anotaciones, bibliotecas, superposiciones y extensiones.
- ▶ Aunque **no es obligatorio**, la primera línea de un archivo de inclusión que es un fragmento escrito puede ser un identificador de fragmento RAML con el siguiente formato:

```
#%RAML 1.0 <fragment-type>
```

- ▶ Por ejemplo, la primera línea de un archivo de fragmento escrito para un trait sería:

```
#%RAML 1.0 Trait
```

Includes

- ▶ Si se utiliza un identificador de fragmento, entonces el contenido del archivo DEBE contener sólo RAML válida para el tipo de fragmento que se especifica. Veamos primero una parte de la sección de características de nuestra API:

```
traits:
  - hasRequestItem:
    body:
      application/json:
        type: <<typeName>>
  - hasResponseItem:
    responses:
      200:
        body:
          application/json:
            type: <<typeName>>
            example: !include examples/<<typeName>>.json
```

Includes

- Para modularizar esta sección usando fragmentos escritos, primero reescribimos la sección de rasgos de la siguiente manera:

```
traits:
```

- hasRequestItem: !include traits/hasRequestItem.raml
- hasResponseItem: !include traits/hasResponseItem.raml

Bibliotecas

- ▶ Las **bibliotecas RAML** se utilizan para combinar cualquier **colección de declaraciones** de tipos de datos, declaraciones de tipos de recursos, declaraciones de rasgos y declaraciones de esquemas de seguridad en grupos modulares, externalizados y reutilizables.
- ▶ Si bien las **bibliotecas** están destinadas a **definir declaraciones comunes** en documentos externos, que luego se incluyen cuando sea necesario, las bibliotecas también se pueden definir en línea.
- ▶ Además de los nodos permitidos en la raíz de los fragmentos escritos en RAML, las bibliotecas RAML permiten los siguientes **nodos opcionales**:

Bibliotecas

| Name | Description |
|---|--|
| types? schemas? resourceTypes? traits? securitySchemes? annotationTypes? (<annotationName>)? uses? | The definition of each node is the same as that of the corresponding node at the root of a RAML document. A library supports annotation node like any other RAML document. |
| usage? | Describes the content or purpose of a specific library. The value is a string and MAY be formatted using Markdown . |

Bibliotecas

- El siguiente ejemplo muestra una biblioteca simple como un documento de fragmento RAML reutilizable e independiente.

```
#%RAML 1.0 Library
usage: |
  Use to define some basic file-related constructs.
types:
  File:
    properties:
      name:
      length:
        type: integer
...
resourceTypes:
  file:
    get:
      is: [ drm ]
    put:
      is: [ drm ]
```

Bibliotecas

- ▶ Con la palabra reservada '**Library**' en el encabezado especificamos que estamos creando una librería:

```
#%RAML 1.0 Library
```

```
types:
```

```
  user:
```

```
    type: object
```

```
    properties:
```

```
      firstname: string
```

```
      lastname: string
```

```
      age:    number
```

- ▶ En este caso contiene el tipo llamado **user**. Aunque podemos declarar varios types en una misma librería se suele crear **una librería por cada Type**.

Bibliotecas

- Una vez definida la biblioteca debemos aprender a usarla:

```
#%RAML 1.0
title: Ejemplo
uses:
  mySecuritySchemes: !include libraries/security.raml
  myDataTypes: !include libraries/dataTypes.raml
  myResourceTypes: !include libraries/resourceTypes.raml
  myTraits: !include libraries/traits.raml
```

Overlays y extensiones

- ▶ Las **overlays y extensiones** son **módulos** definidos en archivos externos que se utilizan para **ampliar** una API.
- ▶ Un overlay se utiliza para **ampliar aspectos no funcionales de una API**, como descripciones, instrucciones de uso y elementos de documentación del usuario, mientras que una **extensión** se utiliza para **ampliar o anular aspectos funcionales** de la API.
- ▶ A diferencia de los include, a los que **otros archivos RAML hacen referencia** para ser aplicados como si estuvieran **codificados en línea**, todos los archivos superpuestos y de extensión deben contener una referencia (a través de la propiedad masterRef de nivel superior) a su archivo maestro, que puede ser un archivo válido Definición de API RAML u otro archivo de superposición o extensión, al que se aplicarán.