

Gobierno de APIs. (API Owner)

RAML

Índice del capítulo

- ▶ Raml de un vistazo.
- ▶ Recursos.
- ▶ Métodos.
- ▶ Respuestas.
- ▶ El modelo de entidades.
- ▶ Seguridad.
- ▶ Anotaciones.
- ▶ 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?

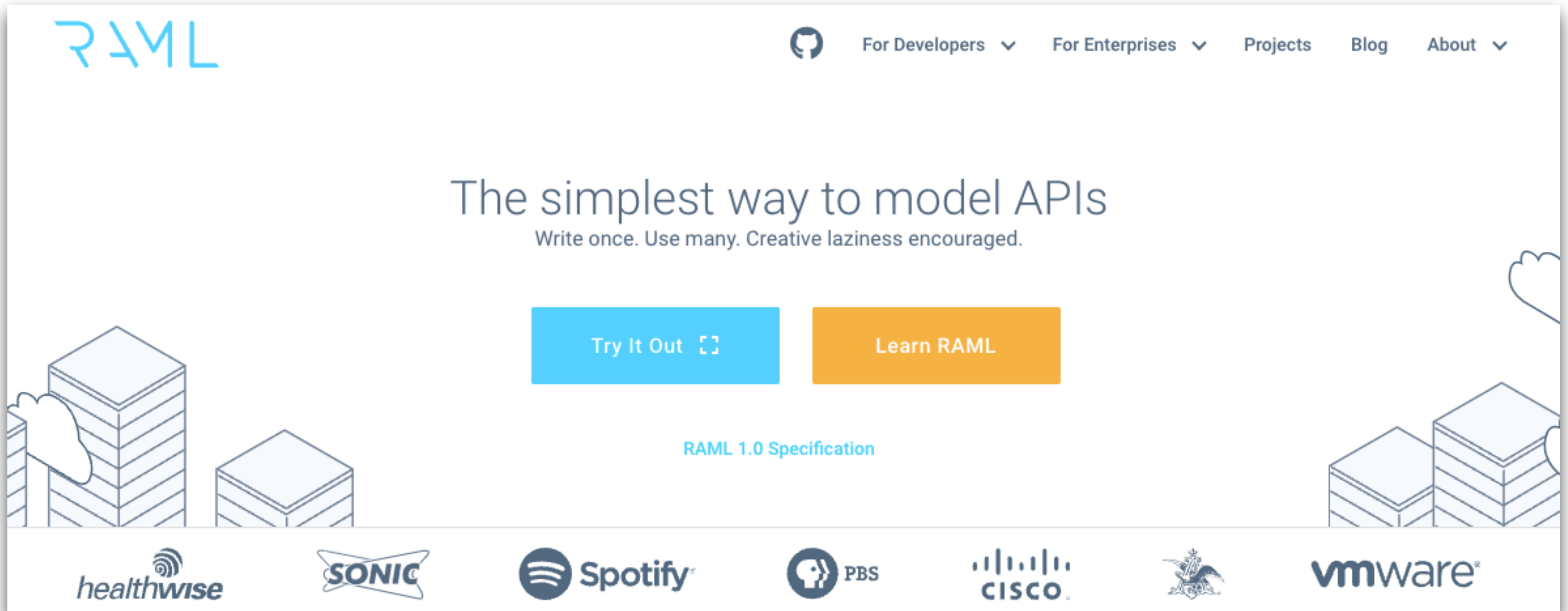
- ▶ RAML es un **lenguaje de modelado para APIs RESTful**.
- ▶ Nos permite escribir el **contrato de la API** y todos sus aspectos como definir sus **recursos, métodos, parámetros, respuestas**, tipos de medios y otros componentes HTTP básicos.
- ▶ Finalmente, puede usarse también para **generar documentación 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:"
```

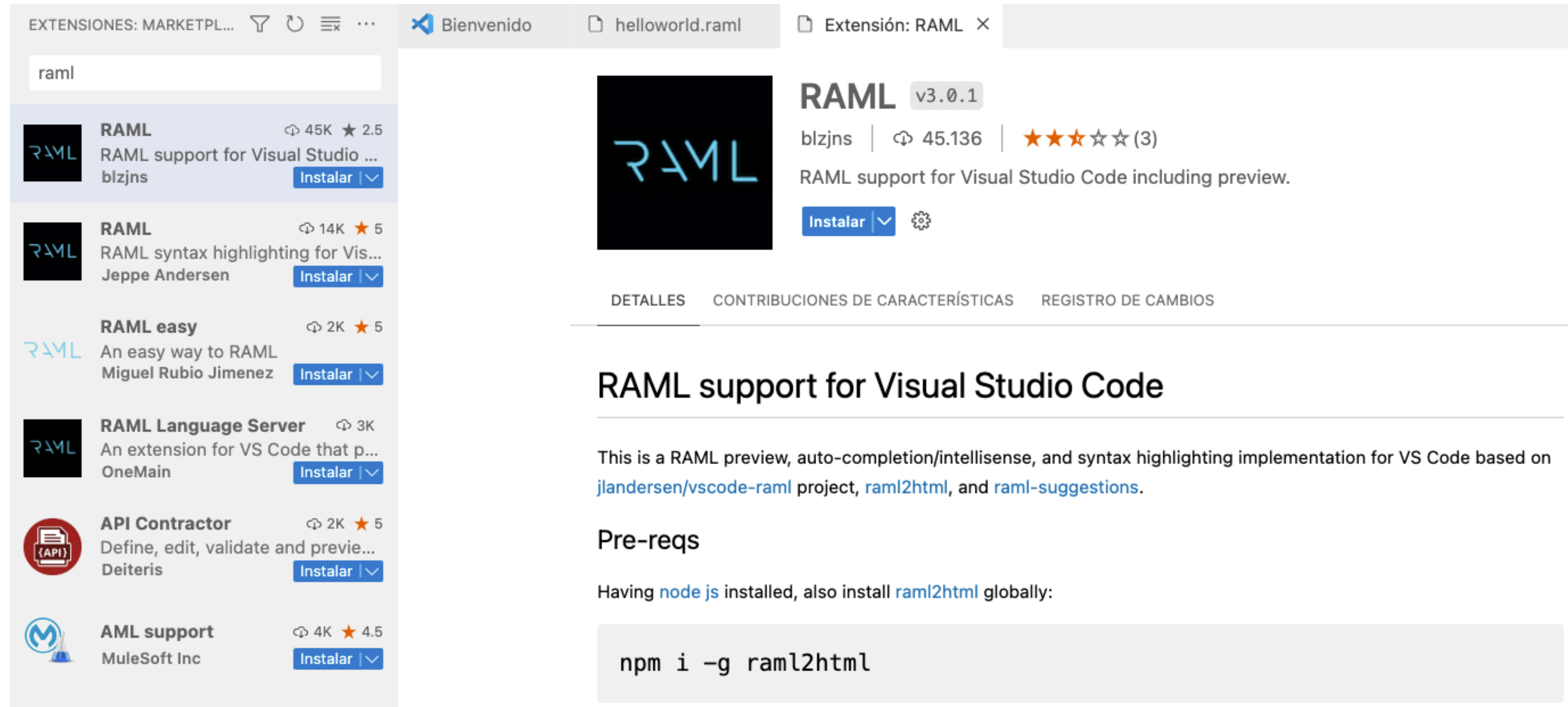
Lenguaje

- **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:' followed by 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

- ▶ 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.

Recursos

- ▶ 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

- ▶ 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 (**queryParam**) 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

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:
```

Parámetros en la url

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

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

- ▶ Por lo tanto, para realizar una solicitud a este recurso anidado, el 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

► Ejemplo:

```
/books:  
  get:  
    queryParameters:  
      author:  
      publicationYear:  
      rating:  
      isbn:  
  put:  
  post:
```

Respuestas

- ▶ La definición de las **respuestas** deben venir acompañados 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'.

```

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

```

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.
```

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 **'properties' facelet**.

El modelo de entidades

- ▶ 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
```

El modelo de entidades

- 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
```

El modelo de entidades

► (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
```


Introducción

- ▶ 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**.