

Diseño de Contratos APIs REST.

Necesidad

- Muchas veces por las necesidades (urgentes) se exponen componentes internos de nuestras aplicaciones en servicios REST públicos.
- La inercia suele llevar a ir creando este tipo de APIs sin un diseño previo.
 - Eso conlleva **problemas** por la **falta** de **planificación**.
 - E **inconsistencia** entre los **objetos** y **métodos**, sin hablar de agujeros de **seguridad**.

Necesidad

- La tendencia cambia (SOA):
 - **Cada vez toma más importancia el diseño previo de APIs** utilizando herramientas que tengan en cuenta:
 - La usabilidad.
 - Las necesidades de los consumidores/aplicaciones que vayan utilizar los servicios,
 - Permitir realizar mocks testeables.
 - Posibilitar el versionado.
 - Y, por supuesto, **crear de forma conjunta al desarrollo de la documentación.**

Ejemplo

```
##RAML 0.8
---
title: Citizens Location Service
baseUri: http://www.dtic.ua.es/v1.0
version: v1.0

/citizenlocations:
  post:
    description: create locations read from RFID smart sensors
    protocols: [HTTPS]
    body:
      application/json:
        example: |
          {
            "idrfid":1,
            "location":[
              "lat":38.384993156837425,
              "lng":-0.5133978999999727
            ],
            "locations":[
              "citizen":["cid": 101010101,"pw": 64,"ts":1443723690],
              "citizen":["cid": 101010101,"pw": 62,"ts":1443723695],
              "citizen":["cid": 203330107,"pw": 62,"ts":1443723695]
            ]
          }
    responses:
      201:
        description: Locations have been successfully created.
        body:
          application/json:
            example: |
              {
                "message": "Locations have been successfully created."
              }
      400:
        description: Locations have not been created.
        body:
          application/json:
            example: |
              {
                "message": "Locations have not been created."
              }
```

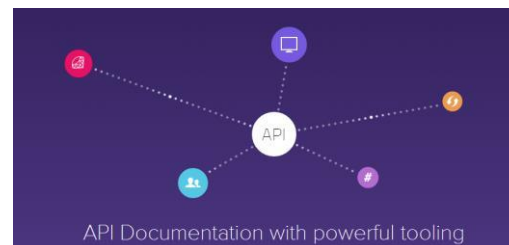
Frameworks Actuales

- **API Blueprint, RAML y Swagger** representan tres excelentes herramientas para diseñar APIs.
 - <https://apibuildprint.org/>
 - <http://swagger.io/>
 - <http://raml.org/>
- Podemos diseñar sobre el papel antes de su implementación la definición de la API en formato JSON o usando markdown para describir la interfaz, estructura y el modelo de datos.



• API Blueprint

- Con [API Blueprint](#) tenemos un amplio ecosistema entorno al desarrollo de APIs.
- Tiene un lenguaje de markdown para escribir la definición y transformarla en JSON.
 - Esto mejora la legibilidad, pensado para humano y no maquinas.
- Podemos usar Node.JS, .NET o Ruby directamente para realizar el binding con nuestra API.
- También posibilita el uso de herramientas con [Apiary.io](#) para crear documentación interactiva, crear API mocks, validaciones, etc... combinado con [Dredd](#) para realizar testing.



- **API Blueprint**

Create your first API

```
$ cat << 'EOF' | snowcrash --format json
# My API
## GET /message
+ Response 200 (text/plain)

    Hello World!
EOF
{
  "name": "My API",
  "description": ""
  ...
}
```

– <https://apiblueprint.org/>

- **swagger**
 - Swagger Editor fué el primer editor creado para diseñar API con la especificación OpenAPI (OAS).
 - El Editor valida su diseño en tiempo real, verifica el cumplimiento de la especificación y proporciona comentarios visuales sobre la marcha.
 - Todo lo que cambie en el código de la API se refleja automáticamente en la documentación.
 - <https://swagger.io/>

MTIS

REST con SOA

Swagger

- **swagger**
 - Para ello, haremos uso de diversas anotaciones
 - Swagger presenta diversas herramientas para:



Design

Design and model APIs according to specification-based standards



Build

Build stable, reusable code for your API in almost any language



Document

Improve developer experience with interactive API documentation



Test

Perform simple functional tests on your APIs without overhead



Standardize

Set and enforce API style guidelines across your API architecture

• Swagger Codegen

- Simplifica el proceso de compilación generando stubs y cliente para cualquier API, definida con la especificación

```
12 Available Clients: [ akka-scala,  
11   android, async-scala, clojure, cpprest, csharp, CsharpDotNet2,  
10   cwiki, dart, dynamic-html, flash, go, groovy, html,  
9   html2, java, javascript, javascript-closure-angular,  
8   jaxrs-cxf-client, jmeter, objc, perl, php, python,  
7   qt5cpp, ruby, scala, swagger, swagger-yaml, swift,  
6   swift3, tizen, typescript-angular, typescript-angular2,  
5   typescript-fetch, typescript-node],  
4  
3 Available Servers: [ aspnet5, aspnetcore,  
2   erlang-server, go-server, haskell, inflector,  
1   jaxrs, jaxrs-cxf, jaxrs-cxf-cdi, jaxrs-resteasy,  
13   "jaxrs-spec", "lumen", "msf4j", "nancyfx", "nodejs-server",  
1   python-flask, rails5, scalatra, silex-PHP, sinatra,  
2   slim, spring, undertow]
```

- **Swagger Editor**

- Diseña, describe y documente APIs.
- Editor de código abierto completamente dedicado a las API basadas en OpenAPI.
- Cumple la especificación OpenAPI, con soporte para Swagger 2.0 y OpenAPI 3.0.

• Swagger Editor

The screenshot displays the Swagger Editor interface. On the left, a code editor shows the Swagger JSON definition for a Petstore API. The definition includes metadata like version (1.0.0), title (Swagger Petstore), and contact information, as well as API endpoints such as /pet and /pet with their respective methods (POST, PUT) and descriptions.

On the right, the rendered API documentation page is shown. It features the title "Swagger Petstore" with a version badge (1.0.0) and a base URL. The page includes a description of the sample server, links to external resources (Terms of service, Contact the developer, Apache 2.0, Find out more about Swagger), and a section for API schemes. The "HTTP" scheme is selected, and an "Authorize" button is present. Below, the API endpoints are listed: a POST endpoint for "/pet" (Add a new pet to the store) and a PUT endpoint for "/pet" (Update an existing pet).

```
1 swagger: "2.0"
2 info:
3   description: "This is a sample server Petstore server. You can find
4     out more about Swagger at [http://swagger.io](http://swagger.io)
5     or on [irc.freenode.net, #swagger](http://swagger.io/irc/). For
6     this sample, you can use the api key 'special-key' to test the
7     authorization filters."
8   version: "1.0.0"
9   title: "Swagger Petstore"
10  termsOfService: "http://swagger.io/terms/"
11  contact:
12    email: "apiteam@swagger.io"
13  license:
14    name: "Apache 2.0"
15    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
16  host: "petstore.swagger.io"
17  basePath: "/v2"
18  tags:
19    - name: "pet"
20      description: "Everything about your Pets"
21    externalDocs:
22      description: "Find out more"
23      url: "http://swagger.io"
24    - name: "store"
25      description: "Access to Petstore orders"
26    - name: "user"
27      description: "Operations about user"
28    externalDocs:
29      description: "Find out more about our store"
30      url: "http://swagger.io"
31  schemes:
32    - "http"
33  paths:
34    /pet:
35      post:
36        tags:
37          - "pet"
38        summary: "Add a new pet to the store"
39        description: ""
```

- **Swagger UI**

- Permite a cualquier persona, visualizar e interactuar con los recursos de la API sin tener implementada ninguna de las lógicas de implementación.
- Se genera automáticamente a partir de su especificación OpenAPI, con la documentación visual que facilita la implementación de back-end y el consumo del cliente.

MTIS

REST con SOA



Swagger

- **Swagger UI**

Schemes
HTTP

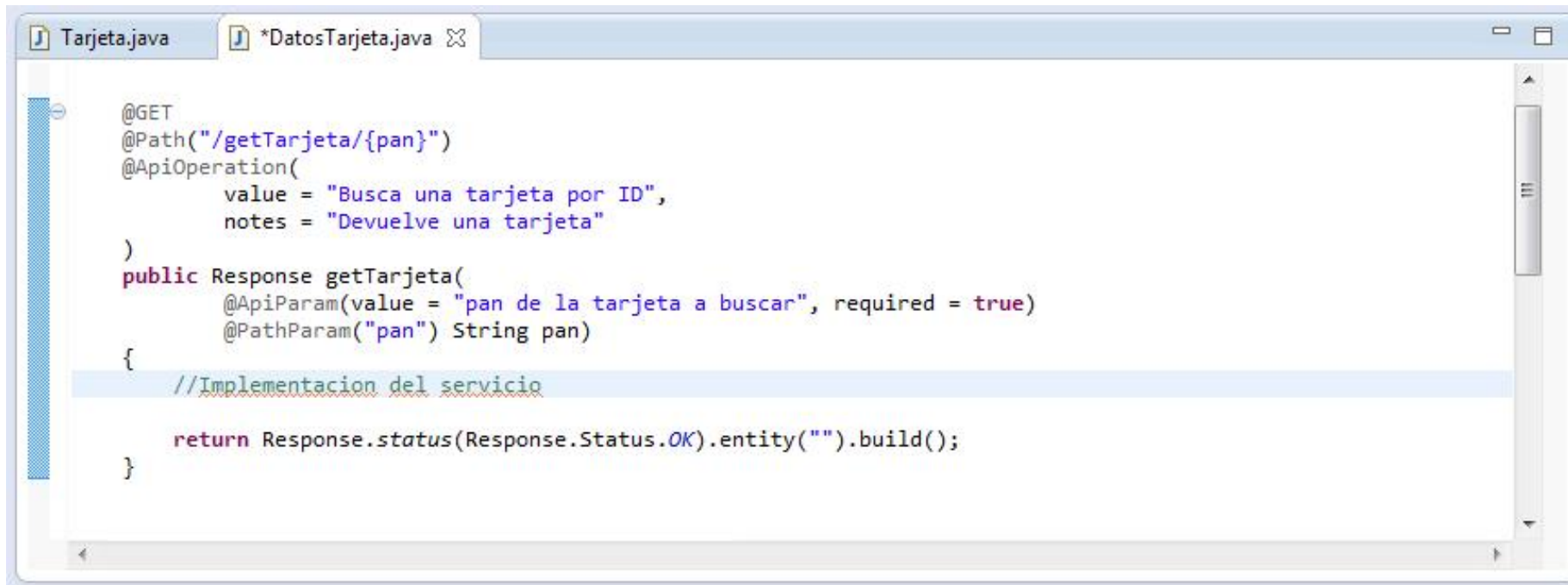
Authorize

pet Everything about your Pets

- POST** /pet Add a new pet to the store
- PUT** /pet Update an existing pet
- GET** /pet/findByStatus Finds Pets by status
- GET** /pet/findByTags Finds Pets by tags
- GET** /pet/{petId} Find pet by ID
- POST** /pet/{petId} Updates a pet in the store with form data
- DELETE** /pet/{petId} Deletes a pet
- POST** /pet/{petId}/uploadImage uploads an image

store Access to Petstore orders

- swagger



```

Tarjeta.java
*DatosTarjeta.java

@GET
@Path("/getTarjeta/{pan}")
@ApiOperation(
    value = "Busca una tarjeta por ID",
    notes = "Devuelve una tarjeta"
)
public Response getTarjeta(
    @ApiParam(value = "pan de la tarjeta a buscar", required = true)
    @PathParam("pan") String pan)
{
    //Implementacion del servicio

    return Response.status(Response.Status.OK).entity("").build();
}

```

- swagger

TarjetaService

[Show/Hide](#) | [List Operations](#) | [Expand Operations](#) | [Raw](#)

POST /TarjetaService/postTarjeta/ Crea una tarjeta

PUT /TarjetaService/putTarjeta/ Actualiza una tarjeta

DELETE /TarjetaService/deleteTarjeta/ Borra una tarjeta

GET /TarjetaService/getTarjeta/{pan} Busca una tarjeta por ID

Implementation Notes

Devuelve una tarjeta

Parameters

Parameter	Value	Description	Parameter Type	Data Type
pan	<input type="text" value="(required)"/>	pan de la tarjeta a buscar	path	string

Try it out!

- swagger

Implementation Notes

Devuelve una tarjeta

Parameters

Parameter	Value	Description	Parameter Type	Data Type
pan	<input type="text" value="1"/>	pan de la tarjeta a buscar	path	string

[Try it out!](#)[Hide Response](#)

Request URL

http://localhost:8080/SwaggerServer/api/TarjetaService/getTarjeta/1

Response Body

```
{
  "pan": "1",
  "cvv": "1",
  "fechaCaducidad": "01012013",
  "tipo": "debito",
  "saldoActual": "100",
  "disponible": "50",
  "bloqueos": "No",
  "titular": "Juan",
  "cuentaAsociada": "111111111111"
}
```

Response Code

200



- **OPENAPI Initiative**

- La Iniciativa OpenAPI (OAI) fue creada por un consorcio de expertos de la industria con visión de futuro que reconocen el inmenso valor de estandarizar sobre cómo se describen las API REST.
- Fundación Linux.
- OAI se enfoca en crear, desarrollar y promover un formato de descripción neutral para el proveedor.
- SmartBear Software ha donando la Especificación Swagger directamente a la OAI como base de esta Especificación Abierta.
- <https://www.openapis.org/>

Raml

RESTful API Modeling Language

- **RAML**

- RAML es la definición de **RESTful API Modeling Language**, el cual permite describir servicios REST de forma completa.
- Destaca la capacidad de **reutilización** de componentes y patrones para aplicar en las definiciones como buenas prácticas.
- Está construido a partir de estándares como **YAML** y JSON.
- <https://raml.org/>

- **¿Qué es YAML?**

- **YAML** es un formato que permite guardar objetos de datos con estructura de árbol. Sus siglas significan *YAML Ain't Markup Language* (YAML no es otro lenguaje de marcado).
- Este lenguaje es muy legible para las personas, más legible que un **JSON** y sobretodo que **XML**.
- Se utiliza normalmente para:
 - Archivos de configuración
 - Traducciones
 - Representar información

- **¿Por que utilizar un YAML en vez de un JSON/XML?**
 - Un formato mucho más amigable
 - Fácil de entender rápidamente
 - Facilita el mapeo de estructuras de datos complejas.

- **Ejemplo YAML**

- Empleados:
 - Empleado:
 empleado_id: 1
 nombre: Fernando
 apellido: Contreras
 - Empleado:
 empleado_id: 2
 nombre: Luis
 apellido: Castañeda

- **Ejemplo JSON**

```
{  
  "Empleados": [  
    {  
      "Empleado": {  
        "nombre": "Fernando",  
        "empleado_id": 1,  
        "apellido": "Contreras"  
      }  
    },  
    {  
      "Empleado": {  
        "nombre": "Luis",  
        "empleado_id": 2,  
        "apellido": "Castañeda"  
      }  
    }  
  ]  
}
```

MTIS

Raml



Ejemplo Guiado

Ejemplo Guiado

Definiendo Api

- Entidad *Foo* , define las operaciones CRUD básicas y un par de operaciones de consulta. Estos son los recursos que definiremos para nuestra API:
 - GET / api / v1 / foos*
 - POST / api / v1 / foos*
 - GET / api / v1 / foos / {id}*
 - PUT / api / v1 / foos / {id}*
 - DELETE / api / v1 / foos / {id}*
 - GET / api / v1 / foos / name / {nombre}*
 - GET / api / v1 / foos? Name = {nombre} & ownerName = {ownerName}*
- API sin estado,
- Autenticación HTTP básica, encriptada a través de HTTPS.
- JSON para nuestro formato de transporte de datos (XML también es compatible).

Configuración Raíz

- Comenzaremos por crear un archivo de texto simple llamado *api.raml* y agregamos la versión de RAML en la línea uno.
- En la cabecera del archivo, definimos la configuración que se aplica a toda la API:

```
#%RAML 1.0  
title: Ejemplo Guiado MTIS API REST  
version: v1  
protocols: [ HTTP ]  
baseUri: http://myapi.mtis.dtic.ua.es/api/{version}  
mediaType: application/json
```

Tipos de datos

- A continuación, definiremos los tipos de datos mediante sintaxis expandida:
 - Tipos de datos mediante sintaxis reducida:

types:

Foo:

type: object

properties:

id:

required: true

type: integer

name:

required: true

type: string

ownerName:

required: false

type: string



types:

Foo:

properties:

id: integer

name: string

ownerName?: string

Error:

properties:

code: integer

message: string

- Ahora, definiremos el recurso de nivel superior (URI) de nuestra API:
`/foos:`

- A continuación, ampliaremos la lista de recursos a partir de nuestro recurso de nivel superior:

`/foos:`

`/{id}:`

`/name/{name}:`

- Las llaves {} alrededor de los nombres de propiedades definen los parámetros de URI.

Métodos

- Ahora se definen los métodos HTTP que se aplican a cada recurso:

```
/foos:
```

```
  get:
```

```
  post:
```

```
    /{id}:
```

```
      get:
```

```
      put:
```

```
      delete:
```

```
        /name/{name}:
```

```
          get:
```

- Ahora definiremos una forma de consultar la colección de *foos* utilizando parámetros de consulta.:

```
/foos:
```

```
  get:
```

```
    description: Muestra todo los Foos siguiendo los criterios si se  
                  proporcionan, en otro caso los muestra todos
```

```
    queryParameters:
```

```
      name?: string
```

```
      ownerName?: string
```

Respuestas

- Los formatos de respuesta generalmente se definen con respecto a los tipos de datos y ejemplos.
- El esquema JSON se puede usar en lugar de los tipos de datos para compatibilidad con versiones anteriores de RAML.

/foos:

...

/ {id}:

get:

description: Get a Foo by id

responses:

200:

body:

application/json:

type: Foo

example: { "id" : 1, "name" : "First Foo" }

- Este ejemplo muestra que al realizar una solicitud GET en el recurso / foos / {id}:
- Recuperamos Foo en formato JSON
- Y código de estado HTTP de 200.

Respuestas

- Aquí es cómo definiríamos la solicitud GET en el recurso / foos :

```
/foos:
get:
  description: Muestra todo los Foos siguiendo los criterios si se
  proporcionan, en otro caso los muestra todos
  queryParameters:
    name?: string
    ownerName?: string
  responses:
    200:
      body:
        application/json:
          type: Foo[]
          example: |
            [
              { "id" : 1, "name" : "First Foo" },
              { "id" : 2, "name" : "Second Foo" }
            ]
```

- El uso de corchetes [] anexados al tipo Foo indica que contiene un array de objetos Foo en JSON.

Cuerpo de solicitud

- A continuación, definiremos los cuerpos de solicitud que corresponden a cada solicitud POST y PUT. Comencemos por crear un nuevo objeto Foo :

```
/foos:
```

```
...
```

```
post:
```

```
  description: Create a new Foo
```

```
  body:
```

```
    application/json:
```

```
      type: Foo
```

```
      example: { "id" : 5, "name" : "Another foo" }
```

```
  responses:
```

```
    201:
```

```
      body:
```

```
        application/json:
```

```
          type: Foo
```

```
          example: { "id" : 5, "name" : "Another foo" }
```

Códigos de estado

- En el ejemplo anterior que al crear un **nuevo** objeto, devolvemos un estado HTTP **201**.
- La operación **PUT** para **actualizar** un objeto devolverá un estado HTTP de **200**, utilizando los mismos cuerpos de solicitud y respuesta que la operación **POST**.
- Además de las respuestas esperadas y los códigos de estado que devolvemos cuando una solicitud es exitosa, podemos definir el tipo de **respuesta** y un código de estado para esperar cuando ocurre un **error**.

- Veamos cómo definiríamos la respuesta esperada para la solicitud GET en el recurso / foos / {id} cuando no se encuentre ningún recurso con la ID dada:

404:

body:

application/json:

type: Error

example: { "message" : "Not found", "code" : 1001 }

- En Raml 0.8 no se podían emplear tipos de datos.
- Se empleaba JSON para definirlos.
- Esto sigue siendo válido, pero es recomendable el empleo de tipos datos.

RAML con esquema JSON

types:

foo:

```
{ "$schema": "http://json-schema.org/schema",  
  "type": "object",  
  "description": "Foo details",  
  "properties": {  
    "id": { "type": "integer" },  
    "name": { "type": "string" },  
    "ownerName": { "type": "string" }  
  },  
  "required": [ "id", "name" ]  
}
```

RAML con esquema JSON

```
/foos:
```

```
...
```

```
/{id}:
```

```
  get:
```

```
    description: Get a Foo by its id
```

```
    responses:
```

```
      200:
```

```
        body:
```

```
          application/json:
```

```
            type: foo
```

```
            ...
```

Refactorización con Include

- La especificación RAML proporciona un mecanismo de inclusión que nos permite externalizar secciones de código repetidas y largas.
- Podemos refactorizar nuestra definición de API usando includes.
- Por ejemplo, podemos poner el tipo de datos para un objeto Foo en los tipos de archivo / Foo.raml y el tipo para un objeto Error en types / Error.raml .

types:

```
Foo: !include types/Foo.raml
```

```
Error: !include types/Error.raml
```


API FINAL

```
#%RAML 1.0
title: Ejemplo Guiado MTIS API REST
version: v1
protocols: [ HTTPS ]
baseUri: http://myapi.mtis.dtic.ua.es/api/{version}
mediaType: application/json
types:
  Foo: !include types/Foo.raml
  Error: !include types/Error.raml
/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[]
            example: !include examples/Foos.json
```

API FINAL

```
post:
  description: Create a new Foo
  body:
    application/json:
      type: Foo
      example: !include examples/Foo.json
  responses:
    201:
      body:
        application/json:
          type: Foo
          example: !include examples/Foo.json
```

API FINAL

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

API FINAL

put:

description: Update a Foo by id

body:

application/json:

type: Foo

example: !include examples/Foo.json

responses:

200:

body:

application/json:

type: Foo

example: !include examples/Foo.json

404:

body:

application/json:

type: Error

example: !include examples/Error.json

API FINAL

delete:

description: Delete a Foo by id

responses:

204:

404:

body:

application/json:

type: Error

example: !include examples/Error.json

/name/{name}:

get:

description: List all Foos with a certain name

responses:

200:

body:

application/json:

type: Foo[]

example: !include examples/Foos.json

Herramientas

- Una de los puntos fuertes de RAML es el soporte de herramientas.
- Hay herramientas para analizar, validar y crear API RAML; herramientas para la generación de código de cliente; herramientas para generar documentación API en formatos HTML y PDF; y herramientas que nos ayudan a probar contra una especificación API RAML.
- Incluso hay una herramienta que convertirá una API Swagger JSON en RAML.

Herramientas

- **API Designer** : una herramienta basada en web orientada a un diseño API rápido y eficiente
- **API Workbench** : un IDE para diseñar, construir, probar y documentar API RESTful que sea compatible con RAML 0.8 y 1.0
- **RAML Cop** : una herramienta para validar archivos RAML
- **RAML para JAX-RS** : un conjunto de herramientas para generar un esqueleto de código de aplicación Java + JAX-RS a partir de una especificación RAML, o para generar una especificación RAML desde una aplicación JAX-RS existente
- **Complemento RAML Sublime** : un complemento de resaltado de sintaxis para el editor de texto Sublime
- **RAML a HTML** : una herramienta para generar documentación HTML desde RAML
- **raml2pdf** : una herramienta para generar documentación PDF desde RAML
- **RAML2Wiki** : una herramienta para generar documentación Wiki (usando el marcado Confluence / JIRA)
- **SoapUI RAML Plugin** - un plugin RAML para el popular conjunto de pruebas funcionales API SoapUI
- **Vigia** : un conjunto de pruebas de integración capaz de generar casos de prueba basados en una definición de RAML.
- **RAML Tools for .NET.**
- www.raml.org

• Bibliografía:

- <https://www.baeldung.com/raml-restful-api-modeling-language-tutorial>
- <http://yaml.org/>
- <https://raml.org/>
- <https://apiblueprint.org/>
- <http://swagger.io/>
- <http://raml.org/>
- <https://www.openapis.org/>