

Tabla de Contenidos



- Consistent
- CRUST y Resiliencia
- No debe ser Ambiguo
- CRUST y Simplicidad
- Pequeña
- Otros artículos relacionados

¿Que es CRUST? . Como siempre cada día aparecen más acrónimos que conocer. Hoy voy a hablar de un acrónimo que se usa en el diseño de APIS , de cualquier tipo de de API y que sirve como una guía a la hora de construirla. CRUST (Consistent, Resilient, Unambiguous, Simple, Tiny) . Son muchas palabras pero nos puedan ayudar a diseñar mejor nuestras APIS. Vamos a empezar

Consistent

Un API se debe diseñar de una forma consistente . ¿Qué quiere decir esto? . Quiere decir que el diseñador del API hará una esfuerzo para que todo sea “consistente” lo cual no es fácil de entender . Pero si lo comparamos con una barra de pan podríamos decir que todo esta unido es uniforme y es muy similar. Veamos un ejemplo del API de Java Collections. Yo puedo tener un ArrayList y desear añadir un item a la lista.

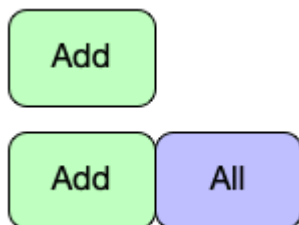
```
List<String> lista= new ArrayList<String>();  
lista.add("hola");
```

Esto es de las operaciones más sencillas .Ahora bien alguien me puede pedir añadir varios elementos de golpe . Para ello tenemos el método addAll()

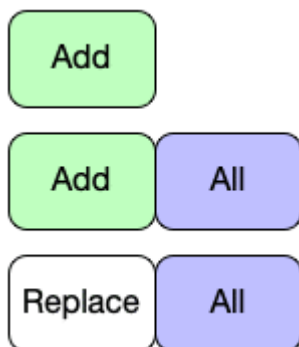
```
lista.addAll(List.of("que", "tal"));
```

Parece que no aporta nada pero la realidad es que es “consistente” no se llama putAll o includeAll , sino addAll. Es decir generamos uniformidad dentro del API ya que antes

teníamos el método `add` y ahora `addAll` los cuales comparten la palabra “add”.



No solo eso sino que si queremos por ejemplo reemplazar todos podemos usar `replaceAll` que contiene la palabra `All` que es compartida con `addAll`. Eso es la consistencia.



CRUST y Resiliencia

La resiliencia es la capacidad de aguante que tenemos ante las adversidades. Por ejemplo nosotros a un `ArrayList` le podemos añadir un elemento , pero también si hace falta podemos añadir una colección . Es decir podemos añadir otra Lista , o otro Conjunto (Set) . No solo eso sino que tambien soporta el añadir elementos a partir de una posición.

```
lista.add(2, "estas");
lista.add(1, "Pedro");
```

Por lo tanto el API aguanta la presión y es resiliente otra de las propiedades que un API debe cumplir.

No debe ser Ambiguo

Por ejemplo cuando añadimos un elemento a un `ArrayList` el `ArrayList` para casi todas las

operaciones devuelve un boolean de si la operación se ha podido realizar o no. Esto nos puede parecer chocante en un primer momento ya que casi nunca lo usamos.

```
List<String> lista= new ArrayList<String>();  
System.out.println(lista.add("hola"));
```

El resultado del programa es true. ¿Qué sentido tiene esto ? . Bueno hay que acordarse que también disponemos de la Colección Set que no admite elementos repetidos de tal forma que necesitamos saber si el elemento se añadió con éxito.

```
Set<String> set= new HashSet<String>();  
set.add("hola");  
System.out.println(set.add("hola"));
```

En este caso el método add devolverá false. Para evitar ambigüedades entre clases y poderlas encajar en la misma Jerarquía de decidió que la clase ArrayList devolviera un booleano cuando añade un elemento . Aunque en ella no se muy necesario . Esto permite encajar el tipo Set de una forma más natural.



Sin ambigüedades



CRUST y Simplicidad

El API debe ser capaz de trabajar de una forma muy sencilla y aquí lo podemos ver añadir un elemento es tan sencillo como pasar un parámetro , no hace falta especificar la posición

el ArrayList o el conjunto se encargarán de colocarlo en la estructura. Lo mismo pasa con el método remove que es capaz de eliminar un elemento de entrada no hace falta pasar la posición.

```
List<String> lista= new ArrayList<String>();  
lista.add("hola");  
lista.remove("hola");  
System.out.println(lista.isEmpty());
```

Pequeña

Muchas veces la gente me pregunta porque el API del interface de Collection es tan sencillo o tan pequeño y hace tan pocas cosas. Esto esta hecho a propósito porque un API debe de ser sencillo de manejar y pequeño. Si necesitamos algo complejo podemos recurrir a la clase [Collections](#) (en plural) y apoyarnos en sus métodos complementarios de esta forma las responsabilidades se dividen y el API pasa a ser mas pequeño y sencillo de manejar.

Otros artículos relacionados

- [Java APIs Diseño y Homogeneidad](#)
- [REST API y su inmutabilidad](#)
- [El concepto de API GateWay Pattern](#)