



**ITI**

INSTITUTO TECNOLÓGICO  
DE INFORMÁTICA

Entregable E4.3

# Lenguajes de Programación (English)

*SaaS*DK



31/12/17

PROYECTO COFINANCIADO POR:



UNIÓN EUROPEA  
Fondo Europeo de  
Desarrollo Regional

*Una manera de hacer Europa*



GENERALITAT  
VALENCIANA

**IVACE**

INSTITUTO VALENCIANO DE  
COMPETITIVIDAD EMPRESARIAL

## Información del documento

Título: **Lenguajes de Programación (English)**

Title: *Programming Languages*

Cod. documento: Entregable E4.3

Proyecto: SaaS SDK

Fecha publicación: 31/12/17

Palabras clave: Cloud, tools, sdk



ITI - Instituto Tecnológico de Informática  
Camino de Vera, s/n. Edif. 8G. Acc. B – 4ª planta  
46022 Valencia - España / Spain  
[www.iti.es](http://www.iti.es)

Personas de  
contacto:

**Sr. José María Bernabé Gisbert, Gestor de Equipos**  
Departamento de I+D - SiDi  
TLF: +34 963 877 069; Email: [jbgisber@iti.es](mailto:jbgisber@iti.es)

**Agradecimientos:** Las actividades descritas en este documento se encuadran en el proyecto "SaaS SDK: Herramientas y servicios para el desarrollo y gestión de software como servicio sobre un PaaS", que está cofinanciado por el Instituto Valenciano de Competitividad Empresarial (IVACE) y por la Unión Europea a través del Fondo Europeo de Desarrollo Regional (FEDER), a través de la convocatoria de ayudas dirigidas a centros tecnológicos de la Comunidad Valenciana para proyectos de I+D en cooperación con empresas 2017 con nº expediente IMDEEA/2017/141

## Nota legal



Este documento está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivar 4.0 Internacional. Se permite libremente copiar, distribuir y comunicar públicamente esta obra siempre y cuando se reconozca la autoría y no se use para fines comerciales. No se puede alterar, transformar o generar una obra derivada a partir de esta obra. Los derechos de autor de todas las marcas, nombres comerciales, marcas registradas, logos e imágenes pertenecen a sus respectivos propietarios.



**ITI**  
INSTITUTO TECNOLÓGICO  
DE INFORMÁTICA



# Contents

Resumen .....	1
Abstract .....	1
1. Introduction .....	2
2. ECloud TypeScript Components Manual.....	2
2.1. Introduction .....	2
2.2. The Component Class Signature.....	2
2.3. The Runtime Object .....	5
2.4. The Channels API.....	6
3. ECloud Scala Components Manual.....	9
3.1. Introduction .....	9
3.2. The Component Class Signature.....	10
3.3. The SLAP Object .....	13
3.4. The Channels API .....	14



This document describes an under development software. TBD (To Be Done) is used to mark pending or unfinished functionality.

## Resumen

El presente documento forma parte del proyecto SaaS SDK cuyo objetivo es el desarrollo de un conjunto de herramientas, aplicaciones y mecanismos para facilitar el desarrollo y gestión de servicios elásticos automantenidos. Este proyecto ha sido subencionado por el Institut Valencià de Competitivitat Empresarial (IVACE) y por el Fondo Europeo de Desarrollo Regional (FEDER).

Este proyecto forma parte de una de las líneas estratégicas del Instituto Tecnológico de Informática (ITI) cuyo objetivo es potenciar el desarrollo de software elástico en la nube. Todo ello se enmarca dentro del ámbito de las Tecnologías de la Información y las Comunicaciones (TIC).

El documento es un Manual que describe cómo desarrollar componentes ECloud con los lenguajes [TypeScript](#) y [Scala](#).

## Abstract

This document is part of SaaS SDK project. This project has been granted by the by the Institut Valencià de Competitivitat Empresarial (IVACE) and the European Regional Development Fund (ERDF). The goal of SaaS SDK is the design and implementation of a set of tools, applications and mechanisms to easy the development and management of auto-escaled elastic services.

This project is aligned with one of the Instituto Tecnológico de Informática (ITI) research line focused on elastic software development for the cloud.

This manual describes ECloud components can be developed using [TypeScript](#) or [Scala](#).

---

# 1. Introduction

This manual describes the main interfaces, classes and libraries needed to develop components using either [TypeScript](#) or [Scala](#)

The rest of the document is divided in two main sections

- [ECloud TypeScript Components Manual](#) describes the classes, interfaces and libraries needed to develop ECloud components using TypeScript.
- [ECloud Scala Components Manual](#) describes the classes, interfaces and libraries needed to develop ECloud components using Scala.

## 2. ECloud TypeScript Components Manual

### 2.1. Introduction

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript ([TypeScript webpage](#)). So, all classes, interfaces and libraries used to develop ECloud Components are typed, either because they have been entirely recoded in TypeScript or because declaration files have been generated for the JavaScript versions.

In the following sections we describe those clases, interfaces and libraries in depth and explain how they can be used to develop a fully functional ECloud component using TypeScript.

### 2.2. The Component Class Signature

In ECloud Service Application Model (see ECloud or SDK manual), the business logic is contained in components. In TypeScript, components should extend the `BaseComponent` class:

```
export class BaseComponent implements Component {
  logger: Logger
  _pid?: NodeJS.Timer

  constructor
    (public runtime: Runtime      // TODO: type of RUNTIME instead
    , public role: string        // ID of the role as a string.
    , public iid: string
    , public incnum: number      // An integer, also
    , public localData: string   // TODO: what is this? a Path?
    , public resources: Object   // A dictionary of key/value pairs.
  TODO: revise this definition
    , public parameters: Object // Again, an object used as a
  dictionary
    , public dependencies: ChannelHash
    , public offerings: ChannelHash
    ) {
    runtime.setLogger([BaseComponent])
  }

  run (): void {
    this._pid = setInterval(() => this.runtime.ping(), PING_INTERVAL)
  }

  shutdown (): void {
    try {
      if (this._pid !== undefined) {
        clearInterval(this._pid)
        this._pid = undefined
      }
    } catch (error) {
      this.logger.warn(COMPONENT_SHUTDOWN_ERROR, this.iid,
error.message)
    }
  }

  reconfig (resources: ResourceHash, parameters: ConfigurationHash):
boolean {
    resources = resources
    parameters = parameters

    return true
  }
}
```

This class constructor expects a set of parameters provided by the underlying ECloud platform when an instance of the component is created. The parameters are:

- **runtime**: a pointer to the ECloud platform representative. This object is mostly used to interact with the platform (excluding the interaction through channels). It is an instance of the

---

**Runtime** interface, which is explained in section [The Runtime Object](#).

- **role**: a strong with role of this instance component in the service according to the service manifest.
- **iid**: the instance id assigned by the ECloud platform.
- **incnum**: the number of times this instance have been reincarnated.
- **localData**: a string with the path to a write enabled folder. The content of this folder is volatile and might be lost when the instance is reincarnated.
- **resources**: a map of resorces. The key is the resource name and the path is the resource value, which depends on the resource type:
  - Client certificate: the certificate as an object.
  - Server certificate: the certificate as an object.
  - Virtual host: the host as an object with a domain and port.
  - Volatile volume: the path in the file system where the volume is mounted.
  - Persistent volume: the path in the file system where the volume is mounted.
- **parameters**: a map of parameters. The key is the parameter name and value depends on the parameter type: boolean, integer, string, number, json, list.
- **dependencies**: map of required channels as declared in the component's manifest. The key is the channel name and the value is the channel object. The concrete interface of **Channel** objects are described in section [The Channels API](#).
- **offerings**: map of provided channels as declared in the component's manifest. The key is the channel name and the value is the channel object. The concrete interface of **Channel** objects are described in section [The Channels API](#).

By default, the constructor sets the runtime logger as the instance logger.

The **BaseComponent** class also implements the following **Component** interface.

```
export interface Component {

    // Starts the execution
    run (): void

    // Saves the state and stops the execution.
    shutdown (): void

    // Changes the component instance parameters
    //
    // Parameters:
    // * `resources`: the new resources.
    // * `parameters`: the new parameters.
    //
    // Returns: `true` if the reconfig can be taken and `false`
    // otherwise.
    reconfig (resources: ResourceHash, parameters: ConfigurationHash):
    boolean
}
```

The method **run** is called by the platform to start the execution of the component instance. Once this method is called the instance can assume that the platform is ready to attend this instance messages.

ECloud calls the method **shutdown** when the instance is going to be destroyed, either because is going to disappear or because it will be relaunched by some other reason (like due to an instance relocation). Before effectively shutting down the instance, the platform gives some grace time to the instance to let it run an ordered close.

If the instance configuration changes, the ECloud platform will call the instance **reconfig** method with all new parameters and resources.

## 2.3. The Runtime Object

The **runtime** parameter is the object provided to the component instances to interact with the platform. This object implements the following interface:

```
export interface Runtime {
    ping (): void
    timeout (channel: Channel, ...args: any[]): void
    createReplyChannel (requestHandler: Server): Reply
    createDuplexChannel (): Duplex
    log (loglevel: LogLevel, message: string): void
    setLogger (items: any[]): void
}
```

The **Runtime** interface methods are used as follows:



- **ping**: to inform that the instance is still alive. This information is used to restart the instance if it is considered down.
- **timeout**: to inform about timeouts in a provided channel. That can be used to detect communication problems among component instances.
- **createReplyChannel**: returns a brand new **DynamicReply** channel instance with the given handler set (see section **Dynamic channels**).
- **createDuplexChannel**: returns a brand new **DynamicDuplex** channel (see section **Dynamic channels**).
- **log**: used to log message to the platform logger.
- **setLogger**: used to set the platform logger in the instance.

Currently, only **ping**, **createReplyChannel** and **setLogger** are supported by ECloud platform. The rest are still **TBD** and will be available in the future.

## 2.4. The Channels API

Channels are objects provided by an ECloud stamp to intercommunicate component instances. A channel is a sort of socket on steroids with specific semantics, which depend on the channel type:

- **Send**: used to send messages, optionally related to a topic.
- **Receive**: used to receive message sent by others. It can be also used to subscribe only to messages with a given topic.
- **Request**: used to perform requests to other instance. In a request the requester is supposed to get a response asynchronously (using promises).
- **Reply**: used to attend requests. When a request is delivered from a reply channel, a handler is invoked. This handler is expected to reply asynchronously using promises.
- **Duplex**: used to enable bidirectional communication between component instances.

Channels are used to send and receive messages and a message is composed by:

- A sequence of segments, being each segment a **Buffer**.
- Optionally, a sequence of dynamic channels (see section **Dynamic channels**).

```
export type Segment = Buffer
export type Data = Segment[]

export interface ChannelHash {
  [index: string]: Channel
}

export interface Message {
  msg: Data
  dyn?: ChannelHash
}
```

All channels implement the **Channel** interface:

```
export interface Channel extends EventEmitter {  
  type: ChannelType  
}
```

This interface only declares a channel as an EventEmmitter with a **ChannelType**:

```
export type ChannelType =  
  'Request'  
  | 'Reply'  
  | 'Duplex'  
  | 'Receive'  
  | 'Send'
```

Each channel type extends the **Channel** interface setting the type and adding some new methods.

The **Send** interface adds a **send** method to send messages, optionally paired with a **topic**:

```
export interface Send extends Channel {  
  type: 'Send'  
  
  send (msg: Message, topic?: string): void  
}
```

The **Receive** interface adds methods for subscribing and unsubscribing to messages.

```
export interface Receive extends Channel {  
  type: 'Receive'  
  
  readonly subscribed: string[]  
  readonly handlers: {  
    message: MessageHandler  
  }  
  
  subscribe (topic: string): void  
  unsubscribe (topic: string): void  
}
```

When a message is received, a **Receive** channel will publish it by emitting a **message** event.

```
export interface MessageHandler {  
  (msg: Message): void  
}
```

The **Request** interface adds a method to send a request. This method returns a promise resolved with the response message.

```
export interface Request extends Channel {  
  type: 'Request'  
  
  sendRequest (msg: Message): Promise<Message> // Promise fails if  
undeliverable  
}
```

In the other side, the **Reply** channel needs a request handler, which will be called once a request arrives.

```
export interface Reply extends Channel {  
  type: 'Reply'  
  
  handleRequest: Server  
  
  readonly handlers: {  
    destinationUnavailable: MessageHandler; // will receive  
"destinationUnavailable" events  
  }  
}
```

The request handler implements the **Server** interface with a single **handleRequest** method. This method expects a message with the incoming request and returns a promise which have to be resolved with the response.

```
export interface Server {  
  (msg: Message): Promise<Message>  
}
```

The **handleRequest** server should be provided by the component instance.

Finally, the **Duplex** channel can send messages to another instance and receive messages from others.

```
export interface Duplex extends Channel {  
  type: 'Duplex'  
  readonly handlers: {  
    message: CompleteMessageHandler; // Handler of "message" events  
    destinationUnavailable: CompleteMessageHandler; // Handler for the  
    "destinationUnavailable" events  
    changeMembership: ChangeMembershipHandler; // Handler for  
    membership change notifications  
  }  
  
  send (msg: Message, target: Tid): void  
  getMembership (): Tid[]  
}
```

The delivered messages are emitted using the `message` event with the delivered message and sender id.

```
export interface CompleteMessageHandler {  
  (msg: Message, sender: Tid): void  
}
```

### 2.4.1. Dynamic channels

Unlike normal channels, which are declared in the service manifest, dynamic channels are created at runtime by a component instance and sent to other instances (through normal channels) as some sort of capability to allow receivers to communicate directly to the channel owner.

As seen at the beginning of section [The Channels API](#), dynamic channels are sent in messages and must be of one of the existing channel types (see `Message` declaration). However, currently, only dynamic `Reply` channels are allowed. When a `Reply` dynamic is sent (through a channel), it is delivered to the receiver as a `Request` channel.

## 3. ECloud Scala Components Manual

### 3.1. Introduction

Scala is a general-purpose programming language providing support for functional programming and a strong static type system. Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Scala provides language interoperability with Java, so that libraries written in both languages may be referenced directly in Scala or Java code. ([https://en.wikipedia.org/wiki/Scala\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scala_(programming_language)))

In the following sections we describe those classes and interfaces in depth and explain how they can be used to develop a fully functional ECloud component using Scala.

## 3.2. The Component Class Signature

In ECloud Service Application Model (see ECloud or SDK manual), the business logic is contained in components. In Scala, the Component class have the following signature:

```
package kumori

/**
 * All ECloud components must implement this interface
 */
trait Component extends Runnable {

  /**
   * It is called by the platform to init the component with its
   * configuration.
   *
   * @param config the config
   */
  def init(config: ComponentConfig): Unit

  /**
   * It is called when the platform decides to stop this instance.
   * The instance has a grace time in order to release acquired
   * resources.
   */
  def shutdown(): Unit

  /**
   * It is called when the platform receives a change of the component
   * configuration in the deployment.
   *
   * @param resources the resources
   * @param parameters the parameters
   */
  def reconfig(resources: Map[String, Any], parameters: Map[String,
Any]): Unit
}
```

ECloud will instance the component and immediately will init it with an object with the same information that native `nodejs` based components receive in its constructor.

```
package kumori

/**
 * The Interface ComponentConfig.
 */
trait ComponentConfig {
```

```
/**
 * The ECloud object, through which the component can interact with
 the platform.
 *
 * @return the ECloud reference
 */
def getSlap(): Slap

/**
 * The role to which this instance belongs. This is a string with
 the name.
 *
 * @return the role.
 */
def getRole(): String

/**
 * The Instance Id ECloud assigns to this instance.
 * This is a unique value during the lifetime of a service. It is a
 string.
 *
 * @return the instance identifier.
 */
def getIid(): String

/**
 * The incarnation number of the instance.
 * It indicates how many times the instance has been started.
 *
 * @return the incnum
 */
def getIncnum(): Int

/**
 * The path where to store local Data. All components get this path
 even
 * though they do not declare a volatile volume: this is it, the
 default
 * volatile volume for which no effort is made in case of failures.
 Note that
 * component instance restarts may destroy the data in this
 location, thus it
 * is unwise to rely on its persistence accross failures.
 *
 * @return the local data path
 */
def getLocalData(): String

/**
 * The set of resource objects assigned to this instance. This is
 organized
```

```

    * as a dictionary, whose keys are the names of the declared
resources for
    * which data can be queried
    *
    * @return the resources
    */
    def getResources(): Map[String, Any]

    /**
    * The set of configuration parameters, organized as a dictionary
with their
    * names as keys
    *
    * @return the parameters.
    */
    def getParameters(): Map[String, Any]

    /**
    * The set of requires channels organized as a dictionary, where the
keys are
    * the requires channel names, and the values are channel objects.
    *
    * @return the requires channels
    */
    def getRequires(): Map[String, Channel]

    /**
    * The set of provides channels. A similar dictionary to the one
for {@link Kumori.ComponentConfig#getRequires requires} channels.
    *
    * @return the provides channels.
    */
    def getProvides(): Map[String, Channel]
}

```

The method `run` is called by the platform to start the execution of the component instance. Once this method is called the instance can assume that the platform is ready to attend this instance messages.

ECloud calls the method `shutdown` when the instance is going to be destroyed, either because is going to disappear or because it will be relaunched by some other reason (like due to an instance relocation). Before effectively shutting down the instance, the platform gives some grace time to the instance to let it run an ordered close.

If the instance configuration changes, the ECloud platform will call the instance `reconfig` method with all new parameters and resources.

### 3.3. The SLAP Object

Class Slap instance received is the way to interact with Ecloud. The Slap can be used by the component to ask for certain resources to the platform (typically, dynamic channels).

This object implements the following interface:

```
package kumori

/**
 * The Interface Slap offers a set of services to interact with
 * ECloud.
 */
trait Slap {

  /**
   * The method ping must be used by the instance to periodically
   * signal its
   * good health to ECloud.
   */
  def ping(): Unit

  /**
   * The timeout method is used by the instance to indicate that a
   * particular
   * channel did not respond in time to some request, or failed to
   * present
   * information when expected. These indications will eventually be
   * used by
   * ECloud to quickly infer instance malfunctions in a service.
   *
   * @param channel the channel
   * @param data the data
   */
  def timeout(channel: Channel, data: Map): Unit

  /**
   * General logging facility
   *
   * @param level the log level
   * @param message the message to be logged
   */
  def log(level: LOG_LEVEL, message: String): Unit

  /**
   * The createReplyChannel method returns a Reply channel ready to be
   * used.
   *
   * The handler parameter will be invoked when messages arrive
   * through the
   * channel.
   */
}
```



```

*
* @param handler the handler for messages that arrive through the
new channel
* @return the reply channel
*/
def createReplyChannel(handler: RequestMessageHandler): Reply

/**
 * The createDuplexChannel method returns a Duplex channel ready to
be used.
 * A Duplex channel is an event emitter, so it must be supplied
handlers for
 * messages and membership events.
 *
 * @param handler the handler for messages
 * @param mhandler the handler for membership change events.
 * @return the duplex channel
 */
def createDuplexChannel(handler: MessageHandler,
                        mhandler: MembershipChangeHandler): Duplex
}

```

Currently, only ping, createReplyChannel and setLogger are supported by ECloud platform. The rest are still TBD and will be available in the future.

## 3.4. The Channels API

Channels are objects provided by an ECloud stamp to intercommunicate component instances. A channel is a sort of socket on steroids with specific semantics, which depend on the channel type:

- Send: used to send messages, optionally related to a topic.
- Receive: used to receive message sent by others. It can be also used to subscribe only to messages with a given topic.
- Request: used to perform requests to other instance. In a request the requester is supposed to get a response asynchronously (using Futures).
- Reply: used to attend requests. When a request is delivered from a reply channel, a handler is invoked. This handler is expected to reply asynchronously using Futures.
- Duplex: used to enable bidirectional communication between component instances.

Channels are used to send and receive messages and a message is composed by:

- A sequence of MessagePart instances, being each part a byte array.
- Optionally, a sequence of dynamic channels (see section [Dynamic channels](#)).

```
package kumori.messages
```

```
/**
 * <p>Represents a message with data that goes from one channel to
 another
 * channel in other component instance. There are different kind of
 data.
 * Depending on the channel type, only some kind of data will be
 relevant.</p>
 *
 * <p>
 * Message instances should be created with the provided factory
 * {@link kumori.messages.MessageFactory}.
 * </p>
 */
trait Message {

  /**
   * Gets an iterator with the parts of the message
   *
   * @return the parts iterator
   */
  def getParts(): Iterator[MessagePart]

  /**
   * Sets an iterator with the parts of the message.
   *
   * @param parts the iterator
   */
  def setParts(parts: Iterator[MessagePart]): Unit

  /**
   * Adds a new part to the message
   *
   * @param part the part
   * @return the message
   */
  def push(part: MessagePart): Message

  /**
   * Gets the first part and removes it from the message.
   *
   * @return the part
   */
  def pop(): MessagePart

  /**
   * Gets the dynamic channels of the message.
   *
   * @return the dynamic channels
   */
  def getDynamicChannels(): List[Channel]
```

```
/**
 * Sets the dynamic channels of the message
 *
 * @param channels the dynamic channels
 */
def setDynamicChannels(channels: List[Channel]): Unit

/**
 * Gets the topic of the message. This kind of data only has sense
in the
 * context of send/receive channels.
 *
 * @return the topic
 */
def getTopic(): String

/**
 * Sets the topic of the message. This kind of data only has sense
in the
 * context of send/receive channels. The message only will arrive to
the channels
 * subscribed to the topic or those who are subscribed to any topic.
 *
 * @param topic the topic
 */
def setTopic(topic: String): Unit

/**
 * Gets the member of the message. This method has sense handling an
incoming
 * message in a duplex channel.
 *
 * @return the member sender of the message.
 */
def getMember(): Member

/**
 * Sets the member target of the message. This method has sense
setting up a
 * message to be sent in a duplex channel.
 *
 * @param member the member target of the message.
 */
def setMember(member: Member): Unit
}
```

MessagePart present this interface:

```
package kumori.messages

/**
 * Represents a part of the data in a message
 * Message instances should be created with the provided factory
 * {@link kumori.messages.MessageFactory}.
 */
trait MessagePart {

  /**
   * Setter for an array of bytes
   *
   * @param bytes the new data
   */
  def setData(bytes: Array[Byte]): Unit

  /**
   * Gets the data as an array of bytes
   *
   * @return the data
   */
  def getData(): Array[Byte]

  /**
   * Gets the data as an input stream.
   *
   * @return the input stream
   */
  def getInputStream(): ByteArrayInputStream

  /**
   * Gets the data as a string.
   *
   * @return the string
   */
  def getString(): String

  /**
   * Sets the data a a string.
   *
   * @param msg the new string
   */
  def setString(msg: String): Unit
}
```

All channels implement the `Channel1` interface:

```
package kumori.channels

/**
 * Basic ECloud channel interface. It only provides the name and the
 * type of the channel.
 * Channel objects offer a set of methods that Component Instances can
 * use to communicate with
 * other Component Instances. Different channels offer different
 * methods.
 */
trait Channel {

  /**
   * Gets the type of the channel. This type is an enumeration of the
   * five
   * possible channel types:
   * <ul>
   * <li>Send</li>Receive</li>Request</li>Reply</li>Duplex
   * </ul>
   *
   * @return the type
   */
  def getType(): ChannelType

  /**
   * Gets the name of the channel.
   *
   * @return the name
   */
  def getName(): String
}
```

Each channel type extends the `Channel` interface setting the type and adding some new methods.

The `Send` interface adds a `send` method to send messages, optionally paired with a `topic`:

```
package kumori.channels
```

```
/**  
 * A send channel implements a send method accepting a message as its  
 * parameter. This message may have a dictionary of dynamic channels  
 * created by the  
 * sender, as well as a string topic.  
 */
```

```
trait Send extends Channel {
```

```
/**  
 * Sends a message. No response to the message is expected.  
 *  
 * @param message the message  
 */
```

```
def send(message: Message): Unit
```

```
}
```

The **Receive** interface adds methods for subscribing and unsubscribing to messages.

```
package kumori.channels

/**
 * A receive channel is an event emitter through its handler property.
The
 * event message carries with it two data: the first is a message, and
the
 * second optionally carries a dictionary of dynamic channels. Besides
the
 * events it emits, receive channels expose subscription methods.
 */
trait Receive extends Channel {

  /**
   * Subscribes the channel to a topic. The topic is a string.
Messages only
   * will arrive to the channel if they have a topic belonging to the
ones
   * subscribed. If a channel have never subscribed to a topic it will
   * receive any message regardless of its topic.
   *
   * @param topic the topic
   */
  def subscribe(topic: String): Unit

  /**
   * Unsubscribes the channel from a topic.
   *
   * @param topic the topic
   */
  def unsubscribe(topic: String): Unit

  /**
   * Sets the handler for arriving messages.
   *
   * @param handler the new handler
   */
  def setHandler(handler: MessageHandler): Unit

  /**
   * Gets the subscribed topics.
   *
   * @return the subscribed topics
   */
  def getSubscribedTopics(): Set[String]
}
```

When a message is received, a Receive channel will publish it by emitting a message event.

```
package kumori.messages

/**
 * This one method interface is the support for implementing the logic
 of
 * message handling in {@link kumori.channels.Receive} and
 * {@link kumori.channels.Duplex} channels.
 */
trait MessageHandler {

  /**
   * Handles an incoming message.
   *
   * @param message the message
   */
  def handleMessage(message: Message): Unit

}
```

The **Request** interface adds a method to send a request. This method returns a promise resolved with the response message.

```
package kumori.channels

/**
 * Request/Reply channels are used to perform direct requests for
 service of
 * other components in the service. Typically a {@link
 kumori.channels.Request}
 * channel is declared as a dependency to a component, whereas a
 * {@link kumori.channels.Reply} channel is declared as an offering.
 * Request/Reply channels can pass along references to dynamic
 * {@link kumori.channels.Reply} or {@link kumori.channels.Duplex}
 channels
 * created using {@link kumori.Slap#createReplyChannel
 createReplyChannel} and
 * {@link kumori.Slap#createDuplexChannel createDuplexChannel}
 methods. On
 * arrival, dynamic {@link kumori.channels.Reply} channels get
 transformed into
 * {@link kumori.channels.Request} channels.
 */
trait Request extends Channel {

  /**
   * Sends a request message. The resolution of the request is
 asynchronous,
   * the platform provides a CompletableFuture of the response message
 for the
```



```

    * request.
    *
    * @param message the request message
    * @return the CompletableFuture of the response
    */
    def sendRequest(message: Message): CompletableFuture[Message]

    /**
     * Sends a request but with a timeout different from the default for
    the
     * channel.
     *
     * @param message the message
     * @param timeout the timeout
     * @return the completable future
     */
    def sendRequest(message: Message, timeout: Long): CompletableFuture
    [Message]

    /**
     * Sets the default timeout in milliseconds for requests using this
    channel.
     *
     * @param timeout the new timeout
     */
    def setTimeout(timeout: Long): Unit

    /**
     * Gets the default timeout in milliseconds for requests using this
    channel.
     *
     * @return the timeout
     */
    def getTimeout(): Long
}

```

In the other side, the **Reply** channel needs a request handler, which will be called once a request arrives.

```

package kumori.channels

/**
 * Request/Reply channels are used to perform direct requests for
    service of
     * other components in the service. Typically a {@link
    kumori.channels.Request}
     * channel is declared as a dependency to a component, whereas a
     * {@link kumori.channels.Reply} channel is declared as an offering.
     * Request/Reply channels can pass along references to dynamic

```

```

* {@link kumori.channels.Reply} or {@link kumori.channels.Duplex}
channels
* created using {@link kumori.Slap#createReplyChannel
createReplyChannel} and
* {@link kumori.Slap#createDuplexChannel createDuplexChannel}
methods. On
* arrival, dynamic {@link kumori.channels.Reply} channels get
transformed into
* {@link kumori.channels.Request} channels.
*/
trait Reply extends Channel {

  /**
   * Sets the handler for arriving requests.
   *
   * @param handler the new handler
   */
  def setHandler(handler: RequestMessageHandler): Unit

  /**
   * If a response message cannot be delivered to the requester, the
   * undelivered message is sent to the DestinationUnavailable handler
   *
   * @param handler the new handler for undelivered messages
   */
  def setDestinationUnavailableHandler(handler: MessageHandler): Unit

  /**
   * Sets the default timeout in milliseconds for resolution of
   requests using
   * this channel.
   *
   * @param timeout the new timeout
   */
  def setTimeout(timeout: Long): Unit

  /**
   * Gets the timeout of this channel.
   *
   * @return the timeout
   */
  def getTimeout(): Long
}

```

`RequestMessageHandler` interface only contains a single `handleRequest` method. This method expects a message with the incoming request and returns a `Future` which have to be resolved with the response.

```
package kumori.messages

/**
 * This one method interface is used to define the behavior of
 * {@link kumori.channels.Reply} channels.
 */
trait RequestMessageHandler {

  /**
   * Handle a new request. It must return synchronously a
   * CompletableFuture of
   * the response to the request.
   *
   * @param message the message
   * @return the CompletableFuture of the response.
   */
  def handleRequest(message: Message): CompletableFuture[Message]
}
```

A implementation of this interface must be provided for each Request channel instance.

Finally, the `Duplex` channel can send messages to another instance and receive messages from others.

```
package kumori.channels

/**
 * <p>
 * Duplex channels allow both sending and receiving messages
 * asynchronously,
 * combining the functionality of send and receive channels into one,
 * and
 * augmenting it with the ability to distinguish the destination.
 * </p>
 * <p>
 * Thus, duplex channels issue message events, as do receive channels,
 * with its
 * info (message and channels), augmented with another parameter,
 * sender,
 * containing a string identifying the instance sending the message.
 * </p>
 */
trait Duplex extends Channel {

  /**
   * Sets the handler for arriving messages.
   *
   * @param handler the new handler
   */
}
```

```
*/  
def setHandler(handler: MessageHandler): Unit  
  
/**  
 * Sends a message. No response to the message is expected.  
 *  
 * @param message the message  
 */  
def send(message: Message): Unit  
  
/**  
 * Gets a list of strings with the instance identifiers belonging to  
the  
 * membership of the channel  
 *  
 * @return the membership  
 */  
def getMembership(): List[Member]  
  
/**  
 * Sets the membership change handler.  
 *  
 * @param handler the new membership change handler  
 */  
def setMembershipChangeHandler(handler: MembershipChangeHandler):  
Unit  
  
/**  
 * If a message cannot be delivered to the destination instance, the  
 * undelivered message is sent to the DestinationUnavailable handler  
 *  
 * @param handler the new handler for undelivered messages  
 */  
def setDestinationUnavailableHandler(handler: MessageHandler): Unit  
}
```

When **Ecloud** detects changes in the membership of a duplex channel, a **membershipChange** event is created.

```
package kumori.channels

/**
 * One method interface for managing membership changes in duplex
 * channels.
 */
trait MembershipChangeHandler {

  /**
   * Gets the info of a new membership change.
   *
   * @param current the current channel membership
   */
  def membershipChanged(current: List[Member]): Unit

}
```

A **Member** is just a bean with info of a source or target of messages. A membership is a list of **Member** instances.

```
package kumori

class Member {

  /**
   * Instance identifier of member
   */
  @BeanProperty
  var iid: String = _

  /**
   * Channel name of member
   */
  @BeanProperty
  var endpoint: String = _

  /**
   * Service uri of member
   */
  @BeanProperty
  var service: String = _

}
```

### 3.4.1. Dynamic channels

Unlike normal channels, which are declared in the service manifest, dynamic channels are created at runtime by a component instance and sent to other instances (through normal channels) as

---

some sort of capability to allow receivers to communicate directly to the channel owner.

As seen at the beggining of section [The Channels API](#), dynamic channels are sent in messages and must be of one of the existing channel types (see [Message](#) declaration). However, currently, only dynamic [Reply](#) channels are allowed. When a [Reply](#) dynamic is sent (through a channel), it is delivered to the receiver as a [Request](#) channel.



**ITI**

INSTITUTO TECNOLÓGICO  
DE INFORMÁTICA

INNOVANDO en TIC para las empresas

Camino de Vera, s/n; CPI -UPV  
Ed / Bldg. 8G. Acc. B –Nivel 4/ 4th Floor  
(46022) Valencia - España / Spain