

# Space Overview

ActiveSpace uses the [Space](#) abstraction which is a simple JavaSpaces like abstraction which sits neatly on a high performance message bus.

The Space API is specifically designed for high performance streaming and for lightweight and high throughput [SED](#) style architectures.

The Space API simplifies the JavaSpaces API to create a simple yet powerful API which is capable of reusing all of the features of modern message bus technologies like hierarchical wildcards, headers, routing, rules and so forth along with both queues & publish/subscribe and pluggable Quality of Service features

When using ActiveSpace in a distributed or clustered environment, the creation of a space with or without a predicate will cause the infrastructure to begin buffering up a number of objects in RAM of the client (assuming there are objects available to be delivered) so that take or listen operations are super fast and not require any synchronous networking latency, locking or contention.

To get a better feel for ActiveSpace try the [javadoc](#) or browse the [source code](#)

## Features of a Space

The basic idea is to make a Space more capable of its features and configuration. A space can use a JMS destination, which supports wildcards for creating hierarchies and grouping spaces together. e.g. this allows some consumers to look at a sub-space while other consumers look at larger groups or hierarchies.

A space can work in two primary modes

- queue based (only one consumer receives each object put into the space)
- publish/subscribe based (anyone who's interested in the space will get the object, assuming it doesn't timeout by the time the consumer tries to look in the space)

Also rather than the template mechanism in JavaSpaces, where you use an empty object with public fields as a way of querying a space, we use an SQL 92 query string to filter spaces. So we can create a child space with some predicate applied.

In addition the space can have additional configuration properties like

- durable (all objects will be persisted if there are no consumers available due to failure)
- transient (high performance but eventually objects will be discarded or at least spooled to disk if there are no available consumers for objects)
- message priority to make the objects in some spaces more important so they are seen first before other spaces

Finally we can easily support a variety of transaction modes such as

- JMS message acknowledgement
- JMS transactions
- full XA transactions with other resources like databases etc

## Comparison to JavaSpaces

The core put and take operations are quite similar and things work at the POJO level so things are nice and simple. There are a few differences however.

- no random access query API. ActiveSpace is designed for high performance streaming of objects around a

network; so you create a space for your query, process as many objects as you wish, then you close down the space again. You are encouraged for performance reasons to create a space and reuse it rather than using a random access lookup style approach as is common with JavaSpaces.

- no dependency on Jini or other technologies. ActiveSpace is a really small & easy to use API
- we hide the transaction demarkation to simplify the API. This allows us to use declarative transactions using frameworks like Spring and so forth, whether its
  - message acknowledgement based
  - simple transations (which could map to JMS transactions under the covers)
  - full XA transactions with other resources like databases etc