Berlin JUG 2015

Motivations
. New map-like API, functional + async
. ConcurrentMap? not distributed in mind
  @ show code: Infinispan Cache is ConcurrentMap, show put/replace previous value issues
. Two ways to solve the issue: AdvancedCache.withFlags() or JCache standard API
. But misses point: to better scale remote or persistence calls, we need to make things async!
. JCache has no async call support
. AdvancedCache has async call support but based on pre-Java8 Futures
. Not want to just making async transport -> end result needed, e.g. did it fail?
. AdvancedCache = bazar, so rather than change it, create a new API with async as top prio

New API
. Infinispan 8 -> Java 8: so use better async support + functional constructs to define new API
. One API -> Three map APIs: ReadOnly, WriteOnly, ReadWrite
  @ code: construction (already prepared, so expand it)

Single Key
. WriteOnlyMap & ReadOnlyMap
  . WO operations do not read previous value
  . WO single-key ops return CF<Void> since nothing can be derived from function
  . RO operations do not acquire locks on key, WO operations do
  . Returns of RO ops are flexible, could be anything from value to metadata
  . Read view has a find() method to return an Optional of the value
  . Read view offers a get() as shortcut for when we have certainty (throws NoSuchElementE)
  . Lacks Haskell do notation or Scala fors but CompletableFuture's can be composed
  @ code: WO eval K -> RO eval K get() -> WO eval K remove() -> RO eval K find()
. ReadWriteMap

  . RW operations read previous value and acquire locks
  . RW can be used for ifAbsent operations, replaces, conditional replaces...etc
  . Conditions can be based on value or metadata information, e.g. version
  . Lifespan in example uses Java 8 API but could use concurrent TimeUnit too -> milliseconds
  . Unless the value is known, use Value-bearing eval functions to avoid capturing external vars
  . If not capturing, lambdas can be cached and makes clustering easier (more later)
  @ code: RW eval K, V if absent w/ lifespan -> RO eval K lifespan (w/ sleep in between)

Multi Key
  . Pass in keys (+values) and function to execute on each key(+value), return Traversable
  . Traversable is a lazy & pull-style API
  . Exposes methods for working with returned data from each function
  . WO multi key operations still return CF<Void>
  . Normally Traversable order matches order of input collection (though not guaranteed)
  . Returning view useful for returning value + metadata information
  @ code: WO evalMany KVs, RW evalMany Ks return previous view, Traversable filter + print
  . Also available:
    . RO.keys() for traversing all keys, RO.entries() for traversing keys, values and metadata
    . WO.evalAll() and RW.evalAll() for updating/removing all keys
      . ^ Traversable order not guaranteed
    . push-style API?
    . in push-style, data pushed to callbacks provided by user instead of lazily fetching them
      . this is the approach used by Rx, but we decided against it because:
        . interested in providing Rx-like API but did not want hard dependencies in core
        . didn't want to reinvent an Rx-like API -> not trivial -> backpressure, flow control
      . pull-style API can be asynchronous since user can decide to user Travesable later

        . Traversable providers clear separation between inter & terminating ops
        . ^ which makes it a good abstraction to avoid unnecessary computation
        . strive to keep this separation clear results in no manual iteration methods
        . so, no iterator() nor spliterator() methods
          . associated with manual, user-end iteration, and we want to avoid such thing
          . majority of cases, Infinispan knows best how to exactly iterate over the data

Clustering & Listeners
  . Demonstrate how functional maps work in clustered environments
  . Live code will also use listeners to see effects in remote nodes
  . All listener events are post-event! Events received after event has happened
    . vs Infinispan Cache events, which are pre & post
    . only post because listener events are only there to find out what's happened
    . pre-events can hint at the possibility of being able to alter execution
    . for pre-operation work or to alter flow, use interceptors
  . Listener registration methods return AutoCloseable -> handler to deregister listeners
  . Start w/ write listeners: for write events in WO or RW maps
    . can be added passing a lambda to listeners.on() method, and gets a read view of entry write
    . write events can't differentiate create/modifid events, since no previous value is available
    . they can differentiate between remove and create/modifed since value can be queried
  . WO & RW ops: lambda is replicated to other nodes...
    . So it needs to be somehow serializable!
  @ code: start with WO map with write listener definition (both) -> WO to store a constant value
    . make sure listener is added in both nodes!
  @ code: show failure to serialize and quick and dirty workaround
    . explain downsides of quick and dirty workaround => payloads are huge and slow
    . show helper class with utility externalizer based

functions => quick and tiny payloads!
  @ code: show read-write listeners, RW.evalMany w/ metadata (setValueMetasIfAbsentReturnBoolean)
    . add read-write listener in node1, onCreate + onModify
    . do evalMany in node2 to create new entries
    . do invidiual eval in node1 to modify one entry and see modified events