Async layer

refactorng

Main (interconnected) problems

- **“Lack of uniquity”** - Existing async layer “services->managers” has lack of uniquity in style between concetualy different layers when it comes to **execution context**
- **“Floating implicitness”** - the implict execution context is passing over the layers when it should stay away in isolation on one particular layer
- “**Week domain asumtions**” based on inheritence (that sets some defaults that hard to change) that leads to lack of flexability

The design (re-define layers)

- We stay with 3 layers (as it was before)
  - **Controller** that accepts http requests throught tomcat/connnection pool/threads
  - **Manager** that accepts delicated calls from Serive layer and repreents “buisiness logic”
    - It (only) manages app sate/cache
    - Make somes (data) trasformations that meet business requrements
  - **Service -** “downstream layer” (instead of calling it “client” or “facade”) represents a downstream resouce

“Execution plan” /pools and contexts

- Tomcat
  - **Connection pool** shold be configured to have n*1000 active threads (if we don't use tomcat 7). they are reletevly cheap asuming that most of them will be in a waiting state (waiting for long-running operation on business layers.
    (We don't want to stop ourself from accepting new upcomming connections only becase unrelaying busines is under load)

Ex. Plan - Controller(s)

- Controllers
  - Owns **“controller-execution context”** that it uses every time when it wants to fork some work execition inside itself or react on underlying event/future
    - This context is **not** intented to be passed to underlaying layers
    - This conetxt is **not** inteneded to do any blocking work (so it may be non-blocking/core-size) [against “floating implicitness”]
    - It is **not** only one possible “dedicated” context, there maybe more than that (see next) [againts “week domain assumption”].
    - One controller may/should **not** use more than one ex context

Ex. plan context injecting

Conrollers, Managers and Services may have several ex contexts wich is neither inhereted (extended) nor implicitly imported

Example:

```
object Controller {  // set of expclicit ex-contexts
 val ex1 = .
 val ex2 = ...
}

class Controller1 {
  implicit val ex1 = Service.ex1   // do not inhering but just “use”
 def doControllerStuff1() { ..  }    // that approaceh is more “save”,
                                     // since we know that there is no any out-of-nowhere-conxtex-injection happening


def doControllerStuff2() {
    implicit val ex1 = Service.ex2   //
    ...
 }
}
```

Also as recommendation is to use one ex context in one Controller/Manager or Service

Ex. Plan - Manager(s)

- Same as controllers – may have many ex-contexts. No inheretence is used to inject ex-context
- Ex contexts in managers are “blocking” or not “blocking” but they all grouped into Manager object (same as controles does)
- No function(implicit ex:ExecutionContext)
- No ex context passing from Controller to Managers

Ex. Plan - Services(s)

- Same as for Controllers and Managers – it has set/group of ex-contexts – **all are blocking** since all that Service(s) should do is connecting to Downstream seystems or work with I/O
-
- ...