MCV Actors
An Akka Experiment

Alexandre Zua Caldeira
zuacaldeira@gmail.com

September 17, 2016

Contents

List of Figures

List of Tables

List of Equations

Part I

Overview

Chapter 1

Introduction

Chapter 2

Java, The Tool Not The Target

Chapter 3

Vaadin, Alone With Java

Chapter 4

*Akka*ros, MVC Actors

Chapter 5

Akkaria, A World To Live In

Part II

The Akka Experiment

Chapter 6

Methodology

6.1 Version Control

6.1.1 Branches

Our repository contains the branches described in Table.

Branch	Description	Grants
`master`	Master branch of the akka-web repository	`admins`
`master/development`	Development branch	`developers`
`master/release`	Release branch	`ci`
`master/production`	Production branch	`users`

Table 6.1: Github repositories in the *akka-web* project.

Branch Naming Convention

Let f be a named feature, and t a named task. Every feature and task is maintained in *github* branches of the *akka-web* repository. Branching follows a naming convention:

- A new feature f will be maintained in branch `development/feature/f`
- A new task t will be maintained in `development/feature/f/t`

6.2 DDV

6.2.1 Preparation

During Preparation phase we make the project converge into the new task or feature. This includes some synchronization at team level: save, commit, pull rebase, conflict resolution, commit, share, creation of a new feature or task branch.

6.2.2 Specification

6.2.3 Test

6.2.4 Implementation

6.2.5 Evolution

Chapter 7

Development Log

7.1 Feature: Vaadin-Akka Integration

The architecture on Section 5 describes a system with *Akka* actors integrated in a *Vaadin* web application. Today's story shows a possible implementation of such integration.

Feature 7.1. Vaadin-Akka Integration consists in the integration of a *Vaadin* web application with an *Akka* actor system to provide a highly scalable and performant web application. The integration architecture is shown in Figure 7.1.

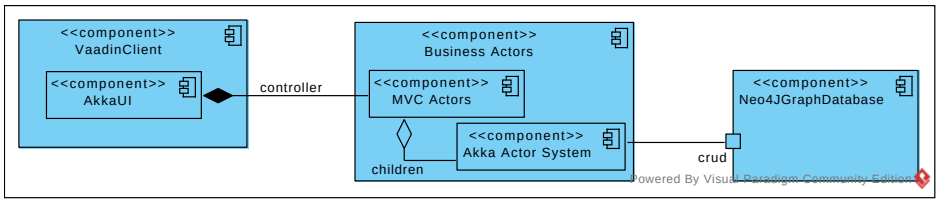

Figure 7.1: *Vaadin-Akka* integration. *Vaadin* subsystem provides a view for clients and communicates with the backend asynchronously via the *Akka* actor system

Task 7.1.1. Create abstract class AkkaUI

Task 7.1.2. Rename MyUI to WelcomeUI

Task 7.1.3. Make WelcomeUI extend AkkaUI

Task 7.1.4. Create class UserUI extends AkkaUI

Task 7.1.5. Merge akkaUI with userUI

Task 7.1.6. Determine the communication protocol in form of a session type specification

Task 7.1.7. Determine the client-side projection of the communication protocol

Task 7.1.8. Determine the server-side projection of the communication protocol

Task 7.1.9. Create tests that asserts about the behaviour expected by the specification, both on client and server sides. This tests should verify that:

- All expected messages are received

  Feature 7.2. feature/task/test/todo

  All messages are processed in the order predefined by the session type

  If termination is mandatory, assert about termination status

Task 7.2.1. Create WelcomeMVCActor, a subclass of MVCActor, as a static[1] inner class of WelcomeUI. This actor will implement the MVC pattern of this architecture:

Task 7.2.2. Implement the communication protocol inside the `onReceive()`, as asynchronously as possible.[2]

Task 7.2.3. Use a *session type based finite state machine* to guide communication dealing with message processing order.

Task 7.2.4. Store incoming messages locally to decide how to proceed and react to them when; messages make the fsm to advance in the session type performing a state transition

Task 7.2.5. Define server-side business actors as BusinessActors

Task 7.2.6. Implement the server-side projection of the asynchronous communication protocol in the `onReceive()` method.

[1] Why `static`

[2] Use `tell` and `forward` actor communication patterns and reserve the `ask` communication pattern for special cases.