there can be enough
actors for anyone

who

Jiří Zůna

@zuzkins

github.com/zuzkins/chakka

vzdelaninadotek.cz

- solution for bringing iPads into teaching
- support and train teachers
- own sw to test students on iPad

controlling the progress of a test in realtime

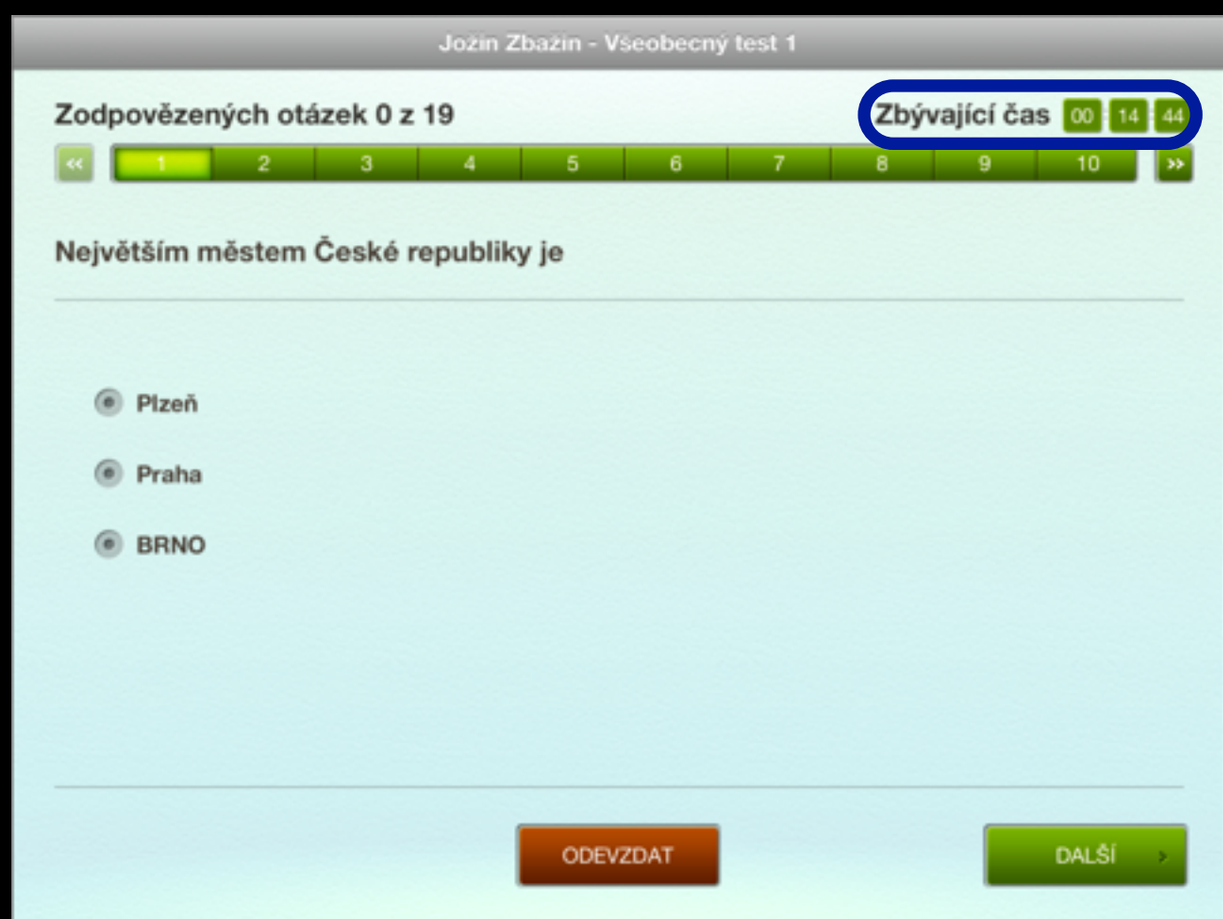
we rolled out our own© solution based on polling the API

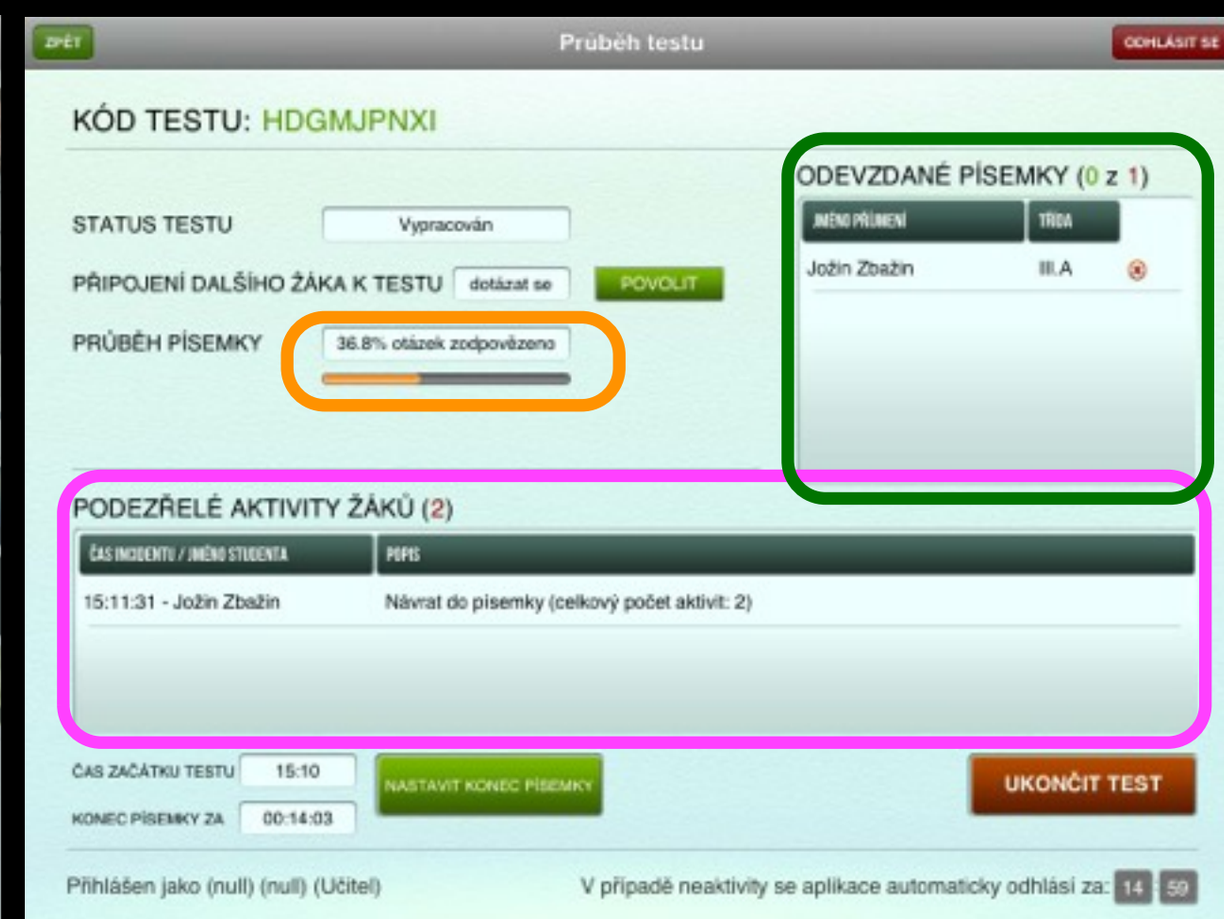
and after some time rewrote it using websockets

controlling the progress of a test in realtime

we rolled out our own© solution based on polling the API

and after some time rewrote it using websockets





implementation

- API: scala with Spring MVC
  - jetty 8.1 supports ws
  - Akka handles msg parsing/sending and clients connected

implementation

- API: scala with Spring MVC
  - jetty 8.1 supports ws
  - Akka handles msg parsing/sending and clients connected
- Objective C
  - https://github.com/square/SocketRocket

jetty

jetty

```
class WsServlet extends WebSocketServlet {
  def doWebSocketConnect(request: HttpServletRequest, protocol: String) = new Ws()
}

class Ws extends OnTextMessage {

  def onMessage(data: String) {
    println("Message got through websocket: " + data)
  }
}
```

jetty

```
class WsServlet extends WebSocketServlet {
  def doWebSocketConnect(request: HttpServletRequest, protocol: String) = new Ws()
}

class Ws extends OnTextMessage {

  def onMessage(data: String) {
    println("Message got through websocket: " + data)
  }
}
```

why actors and Akka

if you do not need to scale?

why actors and Akka

if you do not need to scale?

- lock free solution for shared state

why actors and Akka

if you do not need to scale?

- lock free solution for shared state
- easy to understand model

why actors and Akka

if you do not need to scale?

- lock free solution for shared state
- easy to understand model
- simple to implement

why actors and Akka

if you do not need to scale?

- lock free solution for shared state
- easy to understand model
- simple to implement
- great testability

why actors and Akka

if you do not need to scale?

- lock free solution for shared state
- easy to understand model
- simple to implement
- great testability
- awesome documentation

actor 101

actor 101

- state & behavior

actor 101

- state & behavior
- ref ! Do

actor 101

- state & behavior
- ref ! Do
- val future = ref ? Question

actor 101

- state & behavior
- ref ! Do
- val future = ref ? Question
  - answers will come in the Future[+T]

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms =

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

must extend Actor

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms = Set.empty[ChatRoom]

    case ListRooms                                  => listRooms()
    case msg: JoinRoom                              => joinRoom(msg)
  }
```

has state

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  pty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

defines behavior

actor 102

```
class ChatRoomRegistry extends Actor with ActorLogging {

  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }
```

receive method (PartialFunction[Any, Unit])
is

guaranteed to process only 1 message at a time

one does not simply create an actor

1. you need to create the actor system

```
val actorSystem = ActorSystem("chakka")
```

one does not simply create an actor

1. you need to create the actor system

```
val actorSystem = ActorSystem("chakka")
```

2. then you can ask it to create the actor

```
val ref = actorSystem.actorOf(Props[ChatRoomRegistry], CHAT_REGISTRY_NAME)
```

one does not simply create an actor

1. you need to create the actor system

```
val actorSystem = ActorSystem("chakka")
```

2. then you can ask it to create the actor

```
val ref = actorSystem.actorOf(Props[ChatRoomRegistry], CHAT_REGISTRY_NAME)
```

3. which call gives you back an ActorRef (that is actually really good thing)

ActorRef

ActorRef

- no way to mess with the implementation

ActorRef

- no way to mess with the implementation
- calling code does not know how the message will be handled

ActorRef

- no way to mess with the implementation
- calling code does not know how the message will be handled
- tiny interface: ! (tell) is the most important

ref ! Dolt(For(“me”))

ref ! DoIt(For(“me”))

- fire&forget

ref ! DoIt(For(“me”))

- fire&forget
- puts the message => ref’s mailbox

ref ! DoIt(For(“me”))

- fire&forget
- puts the message => ref’s mailbox
- ActorSystem schedules execution

what does Future[+T] hold

what does Future[+T] hold

- ref can be ? asked to return result

what does Future[+T] hold

- ref can be ? asked to return result
- it actually gives back only a future

what does Future[+T] hold

- ref can be ? asked to return result
- it actually gives back only a future
- doesn’t block and requires a timeout

enter Chakka

enter Chakka

- 1 servlet

enter Chakka

- 1 servlet
- 2 types of actors

enter Chakka

- 1 servlet
- 2 types of actors
  - ChatRoomRegistry joins people into chatrooms

enter Chakka

- 1 servlet
- 2 types of actors
  - ChatRoomRegistry joins people into chatrooms
  - ChatRoomActor handles communication for 1 chat room

ChatRoomRegistry

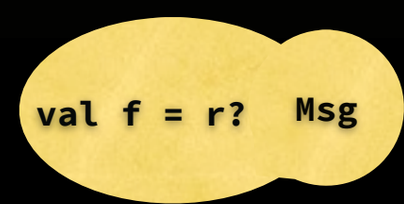
```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```

ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```

ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```



ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```

val f = r?

RoomActor

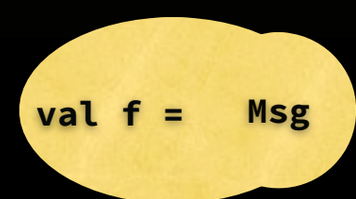
ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```

val f = r?

ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```



ChatRoomRegistry

```
class ChatRoomRegistry extends Actor with ActorLogging {


  var chatRooms = Set.empty[ChatRoom]

  def receive = {
    case ListRooms                          => listRooms()
    case msg: JoinRoom                      => joinRoom(msg)
  }

  def listRooms() {
    sender ! chatRooms.toList
  }

  def joinRoom(msg: JoinRoom) {
    val roomName = msg.roomName
    val room = chatRooms.find(_.name == roomName) match {
      case Some(r)    => r
      case None       =>
        val ref = context.system.actorOf(Props(new ChatRoomActor(msg.roomName)))
        val newRoom = ChatRoom(roomName, ref)
        chatRooms += newRoom
        newRoom
    }

    room.actor forward msg
  }
}
```

val f = r?

RoomActor

ChatRoomActor

```
class ChatRoomActor(val name: String) extends ChatRoomManager
  with ChatController
  with ChatSocketFactory
  with ActorLogging
  with GsonProvider {


  override def receive = super[ChatRoomManager].receive orElse receiveChatMsgs

  def eventTarget = self
}
```

zeta.zunovi.cz:8080

lets look at the code

thank you!