Concurrency and Actors

Lecture Question

Task: Free

- Study actors and concurrency

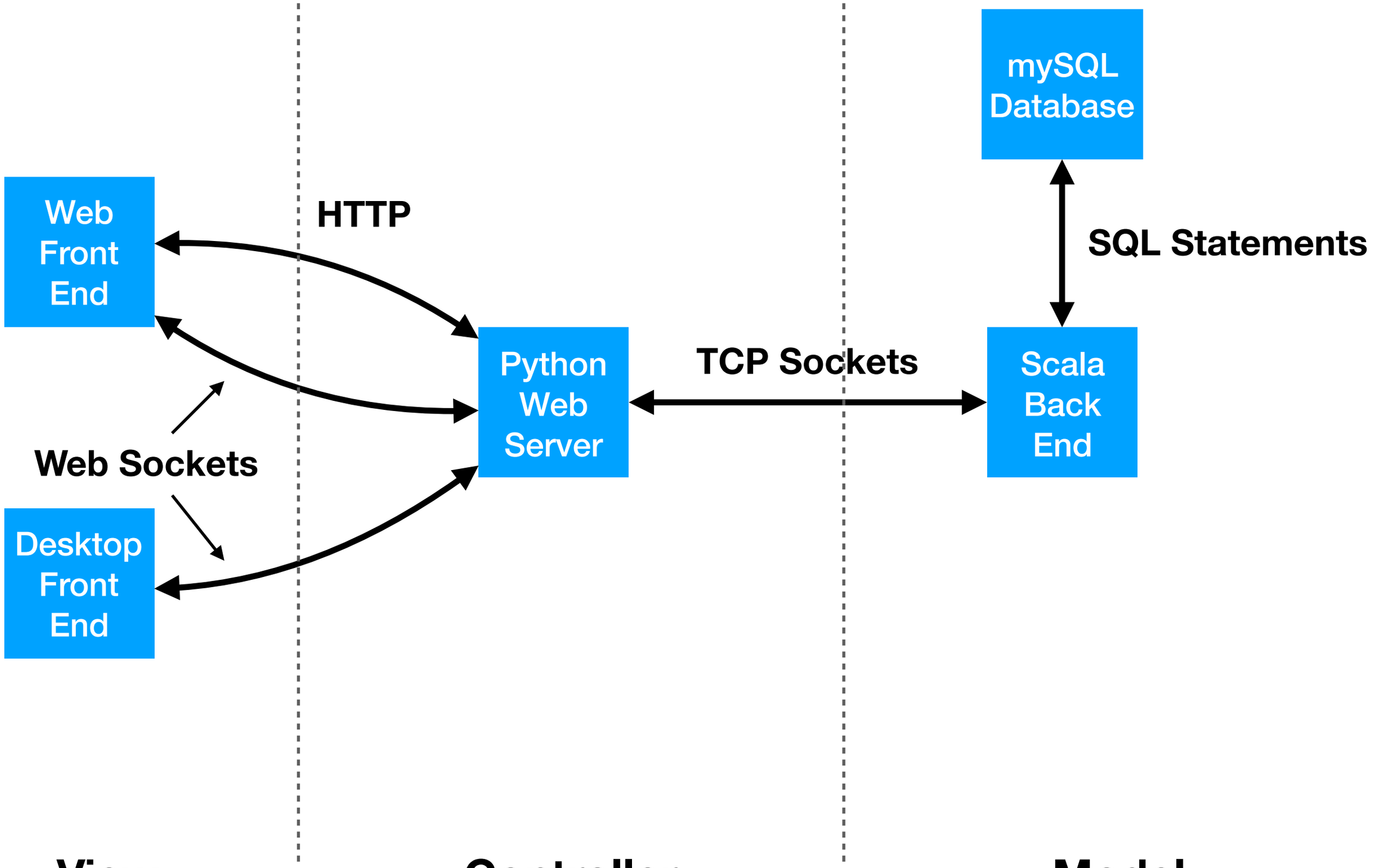
* This question will be open until midnight

CSE116 - End Game

- Multiple pieces of code can run at the same time

- Project part 3 needs 2 front ends
  - But how?

CSE116 - End Game



Concurrency

- To accomplish our goal we'll have to write programs that can perform multiple tasks at the same time
  - Example: The backend needs to update physics and listen for user inputs
  - We "borrowed" concurrency from ScalaFX earlier in the semester

Concurrency - Actors

- We'll use the Akka library
- Akka uses actors for concurrency
- Actors are based on message passing
    - Multiple actors run in the same program at the same time
    - Pass messages to share information
    - Messages are instances of case classes

Concurrency - Actors

- To define an Actor
  - Extend the Actor class
  - Implement the receive method to define how the Actor responds to different message types

```
import akka.actor._

case class CustomMessageType()
case class AnotherMessageType()

class MyActor extends Actor {

  def receive: Receive = {
    case CustomMessageType => // do something
    case AnotherMessageType => // do something
  }

}
```

Concurrency - Actors

- Messages are instances of case classes
- Use case statements to make decisions based on the type of the message

```
import akka.actor._

case class CustomMessageType()
case class AnotherMessageType()

class MyActor extends Actor {

  def receive: Receive = {
    case CustomMessageType => // do something
    case AnotherMessageType => // do something
  }

}
```

Concurrency - Actors

- Start an actor by creating and object and adding it to an actor system
- Send messages using the ! method

```
object CounterTest extends App {
  val system = ActorSystem("FirstSystem")

  val actor = system.actorOf(Props(classOf[MyActor]))

  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```

Concurrency - Actors

- Cannot create an Actor using the new keyword
- Use Props (part of the Akka library) and pass the class as an argument

```
object CounterTest extends App {
  val system = ActorSystem("FirstSystem")

  val actor = system.actorOf(Props(classOf[MyActor]))

  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```

Concurrency - Actors

- If your Actor class takes constructor parameters pass them in the Props call

```
class MyActor(n: Int) extends Actor {

  def receive: Receive = {
    case CustomMessage => // do something
    case AnotherMessageType => // do something
  }

}
```

```
object CounterTest extends App {
  val system = ActorSystem("FirstSystem")

  val actor = system.actorOf(Props(classOf[MyActor], 10))

  actor ! CustomMessageType
  actor ! AnotherMessageType
}
```

Actors - Counting Example

- Create an Actor that counts down from 20 as fast as it can
- Start message starts the countdown
- Responds to IsDone message to tell another actor if it's done or not

```
case class Start()
case class IsDone()
case class Done()
case class NotDone()
```

```
class Counter(name: String) extends Actor {

  var n = 0

  def countDown(): Unit = {
    if (n >= 0) {
      println(this.name + " - " + n)
      n -= 1
      countDown()
    } else {
      println(this.name + " finished")
    }
  }

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
    case IsDone =>
      if (n <= 0) {
        sender() ! Done
      } else {
        sender() ! NotDone
      }
  }

}
```

Actors - Counting Example

- To use the Actor we'll create 3 objects of this type with different names
- Send each Actor the Start message so they count down

```
class Counter(name: String) extends Actor {

  ...

  def receive: Receive = {
    case Start =>
      this.n = 20
      countDown()
  }
}
```

```
object CounterTest extends App {
  val system = ActorSystem("CountingSystem")

  val one = system.actorOf(Props(classOf[Counter], "1"))
  val two = system.actorOf(Props(classOf[Counter], "2"))
  val three = system.actorOf(Props(classOf[Counter], "3"))

  one ! Start
  two ! Start
  three ! Start
}
```

Actors - Counting Example

- Create another Actor that will communicate with the three counters

```
class Supervisor(counters: List[ActorRef]) extends Actor {

  var total: Int = counters.size
  var done: Int = 0
  var notDone: Int = 0

  def receive: Receive = {
    case Update =>
      this.done = 0
      this.notDone = 0
      counters.foreach((actor: ActorRef) => actor ! IsDone)
    case Done =>
      this.done += 1
      if (this.done == this.total) {
        println("All counters complete")
      }
    case NotDone =>
      this.notDone += 1

  }

}
```

Actors - Counting Example

- Use the ActorRef class to send messages to other actors
  - sender() returns the ActorRef of the sender of a message

```
class Supervisor(counters: List[ActorRef]) extends Actor {

  var total: Int = counters.size
  var done: Int = 0
  var notDone: Int = 0

  def receive: Receive = {
    case Update =>
      this.done = 0
      this.notDone = 0
      counters.foreach((actor: ActorRef) => actor ! IsDone)
    case Done =>
      this.done += 1
      if (this.done == this.total) {
        println("All counters complete")
      }
    case NotDone =>
      this.notDone += 1

  }

}
```

Actors - Counting Example

- Add the supervisor to the system and have it update twice per second
  - This is the basic idea we'll use for our physics loop

```
object CounterTest extends App {
  val system = ActorSystem("CountingSystem")

  import system.dispatcher

  val one = system.actorOf(Props(classOf[Counter], "1"))
  val two = system.actorOf(Props(classOf[Counter], "2"))
  val three = system.actorOf(Props(classOf[Counter], "3"))

  val supervisor = system.actorOf(Props(classOf[Supervisor], List(one, two, three)))

  one ! Start
  two ! Start
  three ! Start

  system.scheduler.schedule(0 milliseconds, 500 milliseconds, supervisor, Update)
}
```

Actors - Live Code

- To IntelliJ to see these examples in action

Lecture Question

Task: Free

- Study actors and concurrency

* This question will be open until midnight