MySQL

Lecture Question

Task: Create and test an actor class to track funds in a bank account

- This question is similar to Monday's in that your actor will only track a single value. Through this question you will practice writing a test suite for actors
- In a package named bank create a class named BankAccount that extends Actor
- Create the following case class/objects in the bank package that will be used as messages
  - A case class named Deposit that takes an Int in its constructor
  - A case class named Withdraw that takes an Int in its constructor
  - A case object named CheckBalance
  - A case class named Balance that takes an Int in its constructor
- The BankAccount class must:
  - Initially contain no funds
  - When it receives a Deposit message, increases its funds by the amount in the message
  - When it receives a Withdraw message, decrease its funds by the amount in the message only if the account has enough money to cover the withdrawal. If it doesn't have enough money, no action is needed
  - When it receives a CheckBalance message, sends its current funds back to the sender in a Balance message
- In a package named tests, write a class named TestBankAccount as a test suite that tests the BankAccount functionality

Due: Sunday @11:59pm

Traits and Mixins

- We've seen inheritance by extending abstract classes
- What if we want to extend multiple classes?
  - Example: You want an object class that extends PhysicalObject to have physics applied and extend Actor to run concurrently
  - This is not allowed
- Can avoid this need by using composition
  - Make a class that extends actor and stores a PhysicalObject in an instance variable

Traits and Mixins

- Traits
  - Similar to abstract classes
  - **Cannot have a constructor**
- No limit to the number of traits that can be extended

```scala
trait Database {
  def playerExists(username: String): Boolean
  def createPlayer(username: String): Unit
  def saveGameState(username: String, gameState: String): Unit
  def loadGameState(username: String): String
}
```

Traits and Mixins

- Mixins
    - When extending multiple traits, we use the term mixin (ie. The traits are mixed into the class)
- Extend any one class/abstract class/trait using "extends"
- Add any number of Traits using "with"
- Must implement all abstract methods from all inheritances

```scala
class ActorAndDatabase extends Actor with Database {
  override def receive: Receive = {
    case _ =>
  }
  override def playerExists(username: String): Boolean = {
    false
  }
  override def createPlayer(username: String): Unit = {}
  override def saveGameState(username: String, gameState: String): Unit = {}
  override def loadGameState(username: String): String = {
    ""
  }
}
```

Testing Actors

- We've seen our first actor system where multiple actors can run concurrently
- **But how would we test such a program?**
- Use a FunSuite like we have all semester?
    - FunSuite starts the actor system
    - Creates actors
    - Sends messages
    - Runs some asserts that likely execute before the first message is received by an actor
- Can wait with Thread.sleep to wait for messages, but the FunSuite can't receive messages from the actors
    - Now way of gaining information from the actors

Testing Actors - Library

- Lets pull in a new library to help us out
- The testing library we'll use is the Akka testkit
- Make sure the version number matches your version of the Akka actor library

```xml
<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-actor_2.12</artifactId>
    <version>2.5.25</version>
</dependency>

<dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-testkit_2.12</artifactId>
    <version>2.5.25</version>
</dependency>
```

Testing Actors - Setup

- Using this library, we'll setup a new type of TestSuite using the TestKit class
- This setup can is directly from the test kit documentation and can be reused whenever you setup a test suite for actors

```scala
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- Import classes/traits from the libraries
- The test kit is build on top of scaliest so we'll use both libraries
- Import the duration package from Scala to use 100.millis syntax
    - Important: This must be manually added. IntelliJ will not suggest importing this and you will have an error on your millis

```scala
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- Extend the TestKit class which has a constructor that takes an ActorSystem
- Name and Create a new ActorSystem directly in the constructor

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- Mixin traits for additional functionality
- These traits contain defined methods
    - No methods that need to be implemented

```
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- From the BeforeAndAfterAll train, we inherited the afterAll method which is called after all our tests complete
- By default, afterAll is implement but does nothing
- We override afterAll to properly shutdown the actor system

```scala
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- Finally, we can setup outs tests
- Instead of a test name as we had for unit testing, the test kit uses behavioral tests
    - Name tests by the expected behavior of your code

```scala
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors - Setup

- Methods "must" and "in" are inherited from WordSpecLike and are called inline
  - Uses a string wrapper class (ie. "A value actor" is implicitly converted to the wrapper class which contains the must method)
- All this to achieve more human readable syntax

```scala
import akka.testkit.{ImplicitSender, TestKit}
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

import scala.concurrent.duration._

class TestActors() extends TestKit(ActorSystem("TestValueActor"))
  with ImplicitSender
  with WordSpecLike
  with Matchers
  with BeforeAndAfterAll {

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "A value actor" must {
    "track a value" in {
      // Test actors here
    }
  }

}
```

Testing Actors

- With the test suite setup, we're ready to start testing our actors
- We have access to the actor system we created in the constructor in the system variable
  - Use this system to start your actor(s)

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- Send messages to the actor using the ! method

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- **New**: Use the expectNoMessage method (inherited from TestKit) to wait for messages to resolve before testing
- The test suite will wait for the specified amount of time
  - If there's an error on millis, don't forget to import Scala's duration package

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- Send a message to an actor that is expecting a response
- The test suite is part of the actor system and can receive the response for testing

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- **New**: When you expect to receive a message, call expectMsgType with the message type you expect
- The time provided is the maximum amount to time to wait before failing the test
  - If we don't receive a message of type Value within 1 second, this test fails

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- When the message is received, use asserts to make sure it contains the expected values

```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- Caution: When testing message types with undefined variable names **do not access the variables for testing**
  - For today's lecture question, the names of the Int variables are not defined and it is unlikely that we will use the same names
  - If we use different names => error in AutoLab

```scala
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

- Instead of accessing message variables
    - Create a new message of that type and check for equality
    - Recall that case classes have an equals method that compares the values of its variables instead of comparing references

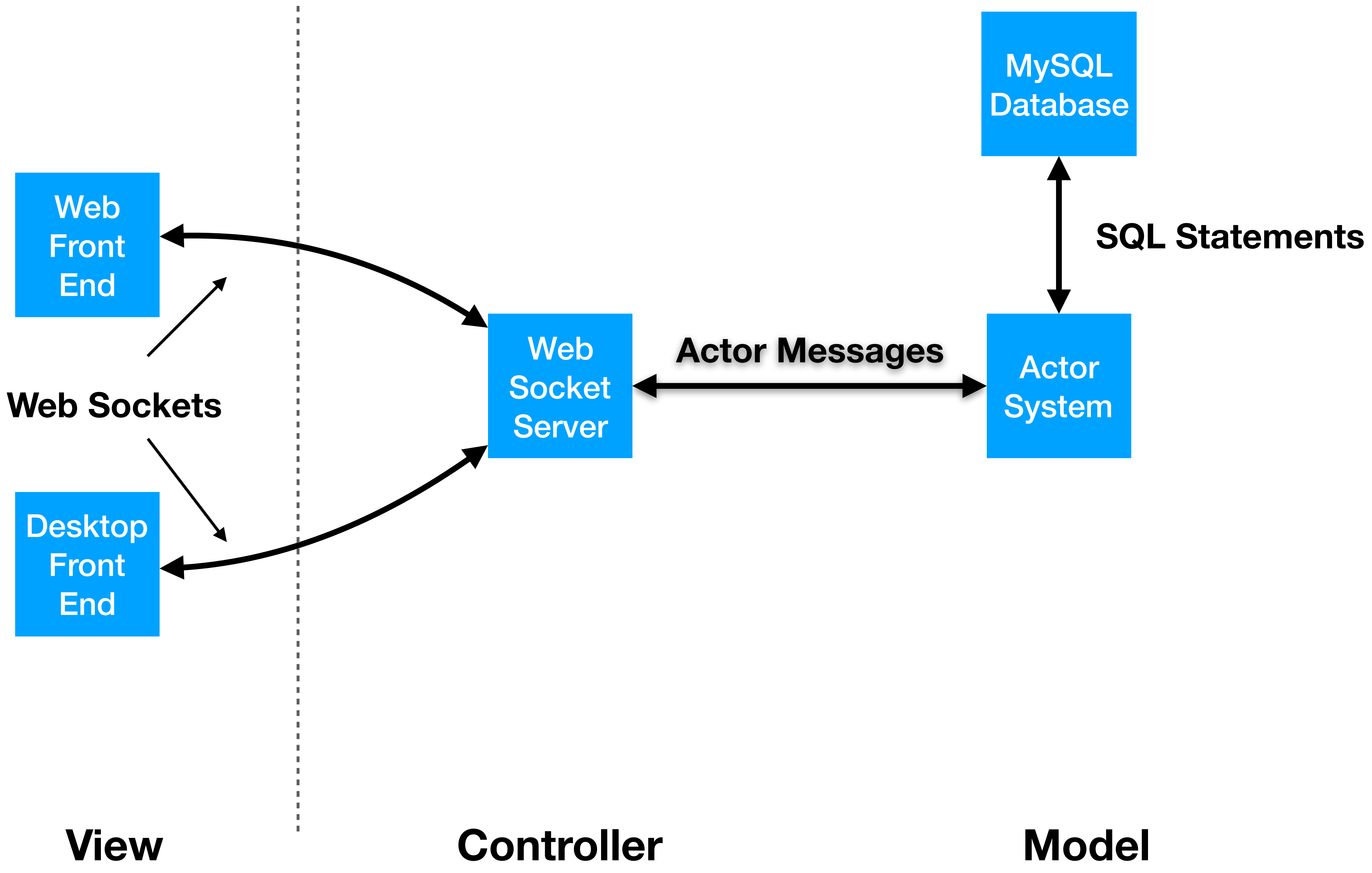
```
"A value actor" must {
  "track a value" in {
    val valueActor = system.actorOf(Props(classOf[ValueActor], 10))

    valueActor ! Increase(5)
    valueActor ! Increase(5)

    expectNoMessage(100.millis)

    valueActor ! GetValue
    val value: Value = expectMsgType[Value](1000.millis)

    assert(value == Value(20))
  }
}
```

Testing Actors

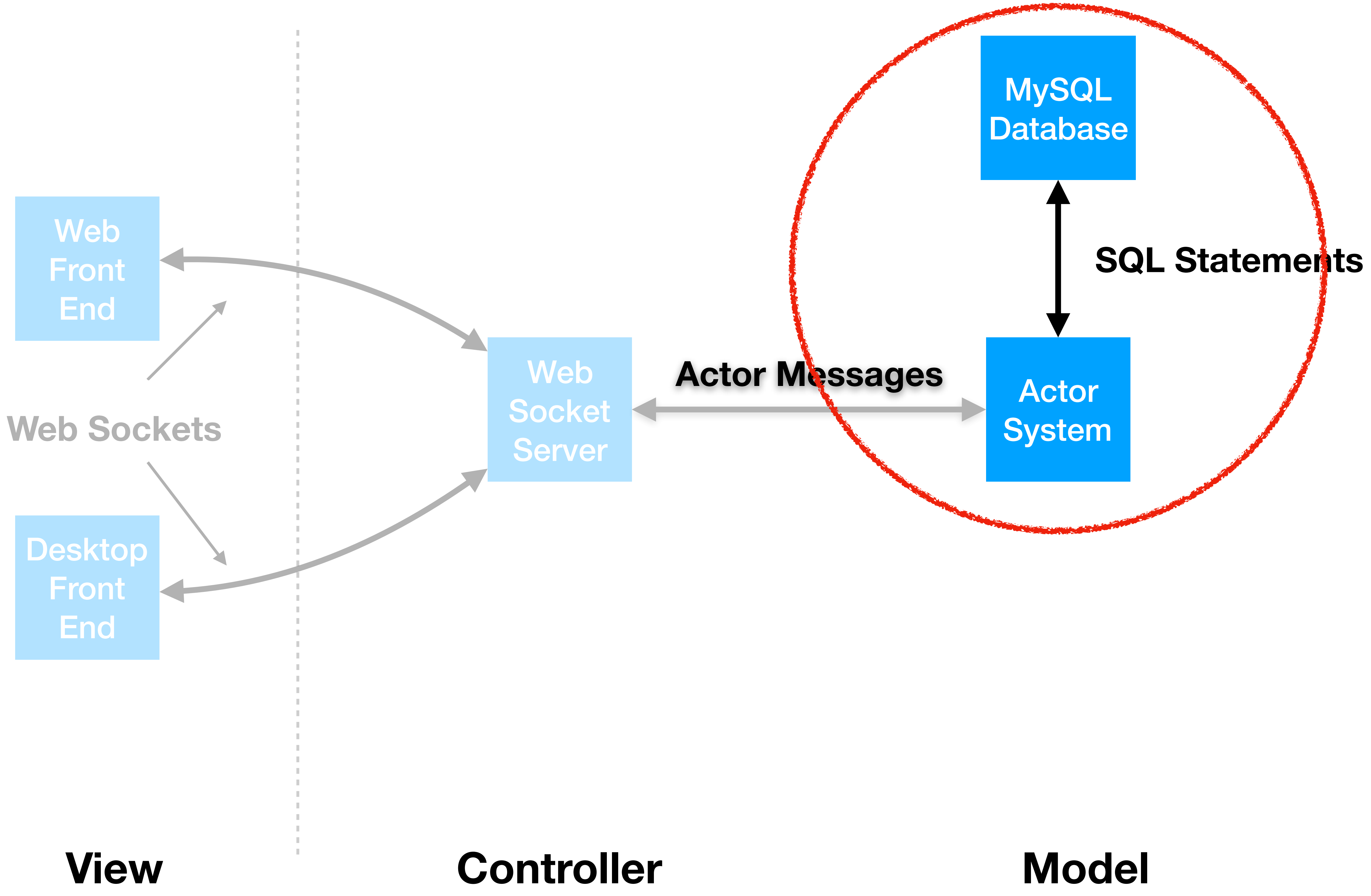
- For Clicker: All the message types and variables are defined and provided in the starter code
  - You can freely access the message variables when writing tests for Clicker

Databases

MMO Architecture



MMO Architecture



MySQL v. SQLite

- MySQL
  - Database server
  - Runs as a separate process that could be running on a different machine
  - Connect to the server and send it SQL statements to execute
- SQLite
  - Removes networking
  - Must run on the same machine as the app
  - Can be used for small apps
    - Common in embedded system - Including Android/iOS apps

MySQL

- A program that must be downloaded, installed, and ran
- Is a server
    - By default, listens on port 3306
- Connect using JDBC (Java DataBase Connectivity)
    - Must download the MySQL Driver for JDBC (Use Maven. Artifact in repo)
    - JDBC abstracts out the networking so we can focus on the SQL statements

MySQL

- After MySQL is running and the JDBC Driver is downloaded..
- Connect to MySQL Server by providing
  - url of database
  - username/password for the database
    - Whatever you chose when setting up the database

```
val url = "jdbc:mysql://localhost/mysql?serverTimezone=UTC"
val username = "root"
val password = "12345678"

var connection: Connection = DriverManager.getConnection(url, username, password)
```

MySQL - Security

- For real apps that you deploy
  - Do not check your password into version control!
    - A plain text password in public GitHub repo is bad
    - Attacker can replace localhost with the IP for your app and can access all your data
  - Common to save the password in a environment variable to prevent accidentally pushing it to git
  - Do not use the default password for any servers you're running
    - This is what caused the Equifax leak (Not with MySQL)
- Attacker have bots that scan random IPs for such vulnerabilities

```
val url = "jdbc:mysql://localhost/mysql?serverTimezone=UTC"
val username = "root"
val password = "12345678"

var connection: Connection = DriverManager.getConnection(url, username, password)
```

MySQL

- Once connected we can send SQL statements to the server

```
val statement = connection.createStatement()
statement.execute("CREATE TABLE IF NOT EXISTS players (username TEXT, points INT)")
```

- If using inputs from the user, always use prepared statements
    - Indices start at 1 😢

```
val statement = connection.prepareStatement("INSERT INTO players VALUE (?, ?)")

statement.setString(1, "mario")
statement.setInt(2, 10)

statement.execute()
```

MySQL - Security

- Not using prepared statements?
    - Vulnerable to SQL injection attacks
- If you concatenate user inputs directly into your SQL statements
    - Attacker chooses a username of "';DROP TABLE players;"
    - You lose all your data
    - Even worse, they find a way to access the entire database and steal other users' data
    - SQL Injection is the most common successful attack

MySQL

- Use executeQuery when pulling data from the database
- Returns a ResultSet
    - The next() methods queue the next result of the query
    - next returns false if there are no more results to read
- Can read values by index of by column name
    - Use get methods to convert SQL types to Scala types

```
val statement = connection.createStatement()
val result: ResultSet = statement.executeQuery("SELECT * FROM players")

var allScores: Map[String, Int] = Map()

while (result.next()) {
  val username = result.getString("username")
  val score = result.getInt("points")
  allScores = allScores + (username -> score)
}
```

SQL

- SQL is based on tables with rows and column
  - Similar in structure to CSV except the values have types other than string
- How do we store an array or key-value store?
  - With CSV our answer was to move on to JSON
  - SQL answer is to create a separate table and use JOINs (Or move to MongoDB)
  - This is beyond CSE116 so we'll stick to data that fits the row/column structure

MySQL

- No automated testing of MySQL in this course

- Fair warning:
    - You will use MySQL in the next lab activity (After demo 2)
    - Have MySQL installed, running, and tested before that lab so you have enough time to complete the activity

Lecture Question

Task: Create and test an actor class to track funds in a bank account

- This question is similar to Monday's in that your actor will only track a single value. Through this question you will practice writing a test suite for actors
- In a package named bank create a class named BankAccount that extends Actor
- Create the following case class/objects in the bank package that will be used as messages
  - A case class named Deposit that takes an Int in its constructor
  - A case class named Withdraw that takes an Int in its constructor
  - A case object named CheckBalance
  - A case class named Balance that takes an Int in its constructor
- The BankAccount class must:
  - Initially contain no funds
  - When it receives a Deposit message, increases its funds by the amount in the message
  - When it receives a Withdraw message, decrease its funds by the amount in the message only if the account has enough money to cover the withdrawal. If it doesn't have enough money, no action is needed
  - When it receives a CheckBalance message, sends its current funds back to the sender in a Balance message
- In a package named tests, write a class named TestBankAccount as a test suite that tests the BankAccount functionality

Due: Sunday @11:59pm