Scala MVC Unit Tests

I need to give out an exercise to my students on writing unit test in scala. Can you geneate me a mvc pattern code and ask them to generate 10 unit tests . Can you also include the answers

Memory updated

Sure, I'll generate a simple MVC pattern code in Scala, and then I'll provide some examples of unit tests and their answers.

MVC Pattern Code

Model

```scala
case class User(id: Int, name: String, email: String)

class UserModel {
  private var users = List[User]()

  def addUser(user: User): Unit = {
    users = user :: users
  }

  def getUser(id: Int): Option[User] = {
    users.find(_.id == id)
  }

  def removeUser(id: Int): Boolean = {
    val initialSize = users.size
    users = users.filterNot(_.id == id)
    initialSize > users.size
  }

  def getAllUsers: List[User] = users
}
```

View

```scala
class UserView {
  def displayUser(user: User): Unit = {
    println(s"User: ${user.name}, Email: ${user.email}")
  }

  def displayAllUsers(users: List[User]): Unit = {
    users.foreach(displayUser)
```

```
  }
}
```

Controller

```scala
class UserController(model: UserModel, view: UserView) {
  def createUser(id: Int, name: String, email: String): Unit = {
    model.addUser(User(id, name, email))
  }

  def showUser(id: Int): Unit = {
    model.getUser(id).foreach(view.displayUser)
  }

  def deleteUser(id: Int): Unit = {
    if (model.removeUser(id)) {
      println(s"User with ID $id deleted.")
    } else {
      println(s"User with ID $id not found.")
    }
  }

  def showAllUsers(): Unit = {
    view.displayAllUsers(model.getAllUsers)
  }
}
```

Exercise

Ask the students to generate 10 unit tests for the MVC code above using ScalaTest. Here are the tests:

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class UserModelSpec extends AnyFlatSpec with Matchers {
  "UserModel" should "add a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    model.getUser(1) shouldBe Some(user)
  }

  it should "retrieve a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    model.getUser(1) shouldBe Some(user)
  }

  it should "return None for a non-existent user" in {
    val model = new UserModel
    model.getUser(1) shouldBe None
  }

  it should "remove a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    model.removeUser(1) shouldBe true
    model.getUser(1) shouldBe None
```

```
  }

  it should "not remove a non-existent user" in {
    val model = new UserModel
    model.removeUser(1) shouldBe false
  }

  it should "retrieve all users correctly" in {
    val model = new UserModel
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(2, "Jane Doe", "jane@example.com")
    model.addUser(user1)
    model.addUser(user2)
    model.getAllUsers should contain allOf(user1, user2)
  }
}

class UserControllerSpec extends AnyFlatSpec with Matchers {
  "UserController" should "create a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    controller.createUser(1, "John Doe", "john@example.com")
    model.getUser(1) shouldBe Some(User(1, "John Doe", "john@example.com"))
  }

  it should "show a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    controller.showUser(1)
  }

  it should "delete a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    controller.deleteUser(1)
    model.getUser(1) shouldBe None
  }

  it should "show all users" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(2, "Jane Doe", "jane@example.com")
    model.addUser(user1)
    model.addUser(user2)
    controller.showAllUsers()
  }
}
```

Answers

1. **Add User Test**

```
scala
```

```
  "UserModel" should "add a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
```

```
    model.getUser(1) shouldBe Some(user)
  }
```

2. **Retrieve User Test**

```scala
  "UserModel" should "retrieve a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    model.getUser(1) shouldBe Some(user)
  }
```

3. **Non-Existent User Test**

```scala
  "UserModel" should "return None for a non-existent user" in {
    val model = new UserModel
    model.getUser(1) shouldBe None
  }
```

4. **Remove User Test**

```scala
  "UserModel" should "remove a user correctly" in {
    val model = new UserModel
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    model.removeUser(1) shouldBe true
    model.getUser(1) shouldBe None
  }
```

5. **Remove Non-Existent User Test**

```scala
  "UserModel" should "not remove a non-existent user" in {
    val model = new UserModel
    model.removeUser(1) shouldBe false
  }
```

6. **Retrieve All Users Test**

```scala
  "UserModel" should "retrieve all users correctly" in {
    val model = new UserModel
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(2, "Jane Doe", "jane@example.com")
    model.addUser(user1)
    model.addUser(user2)
```

```
    model.getAllUsers should contain allOf(user1, user2)
  }
```

7. **Create User Test**

```scala
  "UserController" should "create a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    controller.createUser(1, "John Doe", "john@example.com")
    model.getUser(1) shouldBe Some(User(1, "John Doe", "john@example.com"))
  }
```

8. **Show User Test**

```scala
  "UserController" should "show a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    controller.showUser(1)
  }
```

9. **Delete User Test**

```scala
  "UserController" should "delete a user" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
    controller.deleteUser(1)
    model.getUser(1) shouldBe None
  }
```

10. **Show All Users Test**

```scala
  "UserController" should "show all users" in {
    val model = new UserModel
    val view = new UserView
    val controller = new UserController(model, view)
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(2, "Jane Doe", "jane@example.com")
    model.addUser(user1)
    model.addUser(user2)
    controller.showAllUsers()
  }
```

Can you generate one question on capturing exceptions and one on mocking ?

Sure, here are the two additional questions and their corresponding answers:

Question on Capturing Exceptions

Exercise: Write a unit test to ensure that attempting to add a user with a duplicate ID in the `**UserModel**` throws an `**IllegalArgumentException**` with an appropriate error message.
Answer:

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class UserModelExceptionSpec extends AnyFlatSpec with Matchers {
  "UserModel" should "throw an exception when adding a user with a duplicate ID" in {
    val model = new UserModel
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(1, "Jane Doe", "jane@example.com")
    model.addUser(user1)

    val exception = intercept[IllegalArgumentException] {
      model.addUser(user2)
    }

    exception.getMessage should include ("User with this ID already exists")
  }
}
```

Question on Mocking

Exercise: Using Mockito, write a unit test to verify that the `**UserController**` calls the `**displayUser**` method of the `**UserView**` when `**showUser**` is called.
Answer:

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers
import org.mockito.Mockito._
import org.scalatestplus.mockito.MockitoSugar

class UserControllerMockSpec extends AnyFlatSpec with Matchers with MockitoSugar {
  "UserController" should "call displayUser on the view when showUser is called" in {
    val model = new UserModel
    val view = mock[UserView]
    val controller = new UserController(model, view)

    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)
```

```
      controller.showUser(1)

      verify(view).displayUser(user)
    }
  }
```

These questions focus on handling exceptions and using mocking frameworks like Mockito, which are essential skills for writing robust and maintainable unit tests.

the add user function is'nt throwing any exceptions. Can you make it throw an exception so the unit test works perfectly?

Sure, let's modify the `UserModel` to throw an `IllegalArgumentException` when attempting to add a user with a duplicate ID. Here is the updated `UserModel` and the complete unit test for it.

Updated Model

scala

```
  case class User(id: Int, name: String, email: String)

  class UserModel {
    private var users = List[User]()

    def addUser(user: User): Unit = {
      if (users.exists(_.id == user.id)) {
        throw new IllegalArgumentException("User with this ID already exists")
      }
      users = user :: users
    }

    def getUser(id: Int): Option[User] = {
      users.find(_.id == id)
    }

    def removeUser(id: Int): Boolean = {
      val initialSize = users.size
      users = users.filterNot(_.id == id)
      initialSize > users.size
    }

    def getAllUsers: List[User] = users
  }
```

Unit Test for Exception Handling

Exercise: Write a unit test to ensure that attempting to add a user with a duplicate ID in the `UserModel` throws an `IllegalArgumentException` with an appropriate error message.
Answer:

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class UserModelExceptionSpec extends AnyFlatSpec with Matchers {
  "UserModel" should "throw an exception when adding a user with a duplicate ID" in {
    val model = new UserModel
    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(1, "Jane Doe", "jane@example.com")
    model.addUser(user1)

    val exception = intercept[IllegalArgumentException] {
      model.addUser(user2)
    }

    exception.getMessage should include ("User with this ID already exists")
  }
}
```

Unit Test for Mocking

Exercise: Using Mockito, write a unit test to verify that the `UserController` calls the `displayUser` method of the `UserView` when `showUser` is called.
Answer:

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers
import org.mockito.Mockito._
import org.scalatestplus.mockito.MockitoSugar

class UserControllerMockSpec extends AnyFlatSpec with Matchers with MockitoSugar {
  "UserController" should "call displayUser on the view when showUser is called" in {
    val model = new UserModel
    val view = mock[UserView]
    val controller = new UserController(model, view)

    val user = User(1, "John Doe", "john@example.com")
    model.addUser(user)

    controller.showUser(1)

    verify(view).displayUser(user)
  }
}
```

These modifications ensure that the `addUser` method throws an exception for duplicate IDs, allowing the corresponding unit test to work correctly.

this mocking example seems too simple , Can you make it complicated?

Sure, let's make the mocking example more complex by adding more interactions between the `**UserController**`, `**UserModel**`, and `**UserView**`. We'll include scenarios such as fetching all users, deleting users, and handling exceptions.

Updated MVC Code

For the `**UserController**`, let's add exception handling when deleting users and provide more comprehensive interactions with the view.

Model

(No changes needed)

View

scala

```
class UserView {
  def displayUser(user: User): Unit = {
    println(s"User: ${user.name}, Email: ${user.email}")
  }

  def displayAllUsers(users: List[User]): Unit = {
    users.foreach(displayUser)
  }

  def displayError(message: String): Unit = {
    println(s"Error: $message")
  }
}
```

Controller

scala

```
class UserController(model: UserModel, view: UserView) {
  def createUser(id: Int, name: String, email: String): Unit = {
    try {
      model.addUser(User(id, name, email))
    } catch {
      case e: IllegalArgumentException => view.displayError(e.getMessage)
    }
  }

  def showUser(id: Int): Unit = {
    model.getUser(id) match {
      case Some(user) => view.displayUser(user)
      case None => view.displayError(s"User with ID $id not found.")
    }
  }

  def deleteUser(id: Int): Unit = {
    if (model.removeUser(id)) {
      println(s"User with ID $id deleted.")
    } else {
      view.displayError(s"User with ID $id not found.")
    }
  }

  def showAllUsers(): Unit = {
```

```
      view.displayAllUsers(model.getAllUsers)
    }
  }
```

Complex Mocking Example

Exercise: Using Mockito, write a unit test to verify multiple interactions between the `**`UserController`**`, `**`UserModel`**`, and `**`UserView`**`, including exception handling and multiple method calls.
Answer:

scala

```
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers
import org.mockito.Mockito._
import org.mockito.ArgumentMatchers._
import org.scalatestplus.mockito.MockitoSugar

class UserControllerComplexMockSpec extends AnyFlatSpec with Matchers with MockitoSugar {
  "UserController" should "handle multiple interactions correctly" in {
    val model = mock[UserModel]
    val view = mock[UserView]
    val controller = new UserController(model, view)

    val user1 = User(1, "John Doe", "john@example.com")
    val user2 = User(2, "Jane Doe", "jane@example.com")

    // Mocking model interactions
    when(model.getUser(1)).thenReturn(Some(user1))
    when(model.getUser(2)).thenReturn(None)
    when(model.getAllUsers).thenReturn(List(user1, user2))
    when(model.removeUser(2)).thenReturn(false)

    // Test showUser for existing user
    controller.showUser(1)
    verify(view).displayUser(user1)

    // Test showUser for non-existent user
    controller.showUser(2)
    verify(view).displayError("User with ID 2 not found.")

    // Test showAllUsers
    controller.showAllUsers()
    verify(view).displayAllUsers(List(user1, user2))

    // Test deleteUser for non-existent user
    controller.deleteUser(2)
    verify(view).displayError("User with ID 2 not found.")

    // Mocking exception handling in addUser
    doThrow(new IllegalArgumentException("User with this ID already
exists")).when(model).addUser(user1)

    // Test createUser with existing user
    controller.createUser(1, "John Doe", "john@example.com")
    verify(view).displayError("User with this ID already exists")
  }
}
```

Breakdown of the Complex Mocking Test

1. **Mocked `UserModel` and `UserView`:**
    - `**`getUser(1)`**` returns `**`Some(user1)`**`.

- `getUser(2)` returns `None`.
- `getAllUsers` returns a list of `user1` and `user2`.
- `removeUser(2)` returns `false`.

2. **Show User**:
   - `showUser(1)` verifies `view.displayUser(user1)`.
   - `showUser(2)` verifies `view.displayError("User with ID 2 not found.")`.
3. **Show All Users**:
   - `showAllUsers()` verifies `view.displayAllUsers(List(user1, user2))`.
4. **Delete User**:
   - `deleteUser(2)` verifies `view.displayError("User with ID 2 not found.")`.
5. **Create User with Exception Handling**:
   - `addUser(user1)` throws an exception.
   - `createUser(1, "John Doe", "john@example.com")` verifies `view.displayError("User with this ID already exists")`.

This test ensures that the `UserController` interacts correctly with both the `UserModel` and the `UserView`, handling various scenarios including exceptions.