MVC

Model-View-Controller Architecture

MVC

- Software **architecture pattern**

  - A way to organize the code of an entire project

  - As opposed to **design patterns** which solve a specific problems within a project

- Separate code into a Model, View, and Controller

  - Decouple the project into 3 pieces

- All three parts work independently and communicate with each other through APIs

API

- In CSE115 you've seen **web** APIs which have various endpoints

- An API is a set of functions/methods that can be called

- In the state pattern, define an API for the state

    - These are the methods that can be called and are deferred to the current state for functionality

    - Other classes only look at the API and call those methods

        - List of ways of interacting with the object

- The methods of any class/object define an API

API

- Can change functionality if the API remains the same

- Example:

  - We can add dynamic collisions to the physics engine

  - If we don't change the way updateWorld is called, all games using the engine now have dynamic collisions

- Changing the API will break all code using that API

- Example: Changing a return type from Double to Int will cause type mismatch errors in any code calling that API method

MVC

- Model (Data and Logic)

  - Controls the app and its data

  - The core of the app

- View (Display)

  - Visualizes the app

  - No logic

- Control (User Inputs)

  - Handles user inputs

  - Calls model API methods based on inputs

MVC
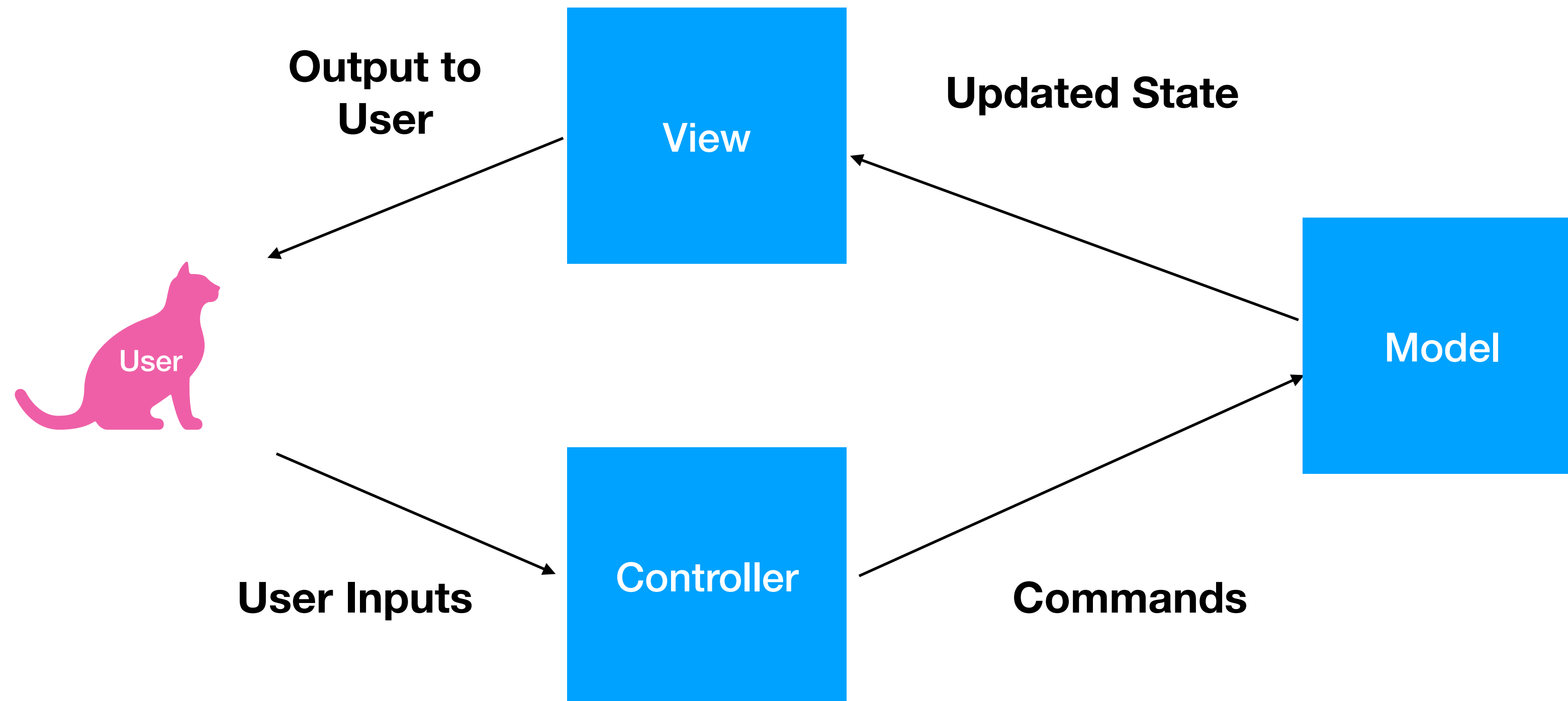
MVC - Model

- The core of the app

- Most of the code you've written so far in CSE115/116 is part of a model

  - Physics Engine handling how objects move

  - Calculator logic when buttons are pressed

- Controls the logic and functionality of the app

- Maintains the data

  - Controls any data structure, databases, and files related to how the program behaves

- **Has no knowledge of the user of the app**

- Functionality accessed through an API

MVC - View

- Displays the state of the app to the user

- **Output only**

- No logic

  - The view cannot change the state of the app

- Since the view is output only and does not alter the app, it can change and be replaced without affecting the app itself

- Can test the logic of an app without using the view

- Can have the same app with a CLI (command line interface) and a GUI (graphical user interface)

- Can have the same app with a web front-end and a desktop front-end!

MVC - Controller

- Handles user inputs

- In ScalaFX, defined by EventHandlers

- Processes user inputs and converts them into calls of the model API

- Can validate and block invalid inputs

- Acts as a barrier between the GUI and the model

  - If the GUI changes, replace view and controller and model remains unchanged

MVC - Advantages

- Focus on 1 part of a project at a time

  - Reduce spaghetti code

  - Divide work among team members

    - Just agree on the APIs

- Views can be easily replaced

- Keeps code organized

- Easier to add new features

  - Model can add features as long as API remains unchanged

MVC on the Web

- Model runs on the server

- View runs in the browser (HTML/CSS)

- Controller can run on both

  - JavaScript in the browser converts user inputs into AJAX requests

  - Server validates the data and sends the commands to the Model

MVC - Jumper

- Model API

  - Left, right, jump pressed for each player

  - Allows view to access all data

- Controller

  - Convert W, A, D, ←, ↑, → key presses into model API calls

- View

  - Displays all game objects to the player

  - Receives absolute locations of all objects from model

    - Computes vertical scroll and translates objects accordingly

MVC - Calculator

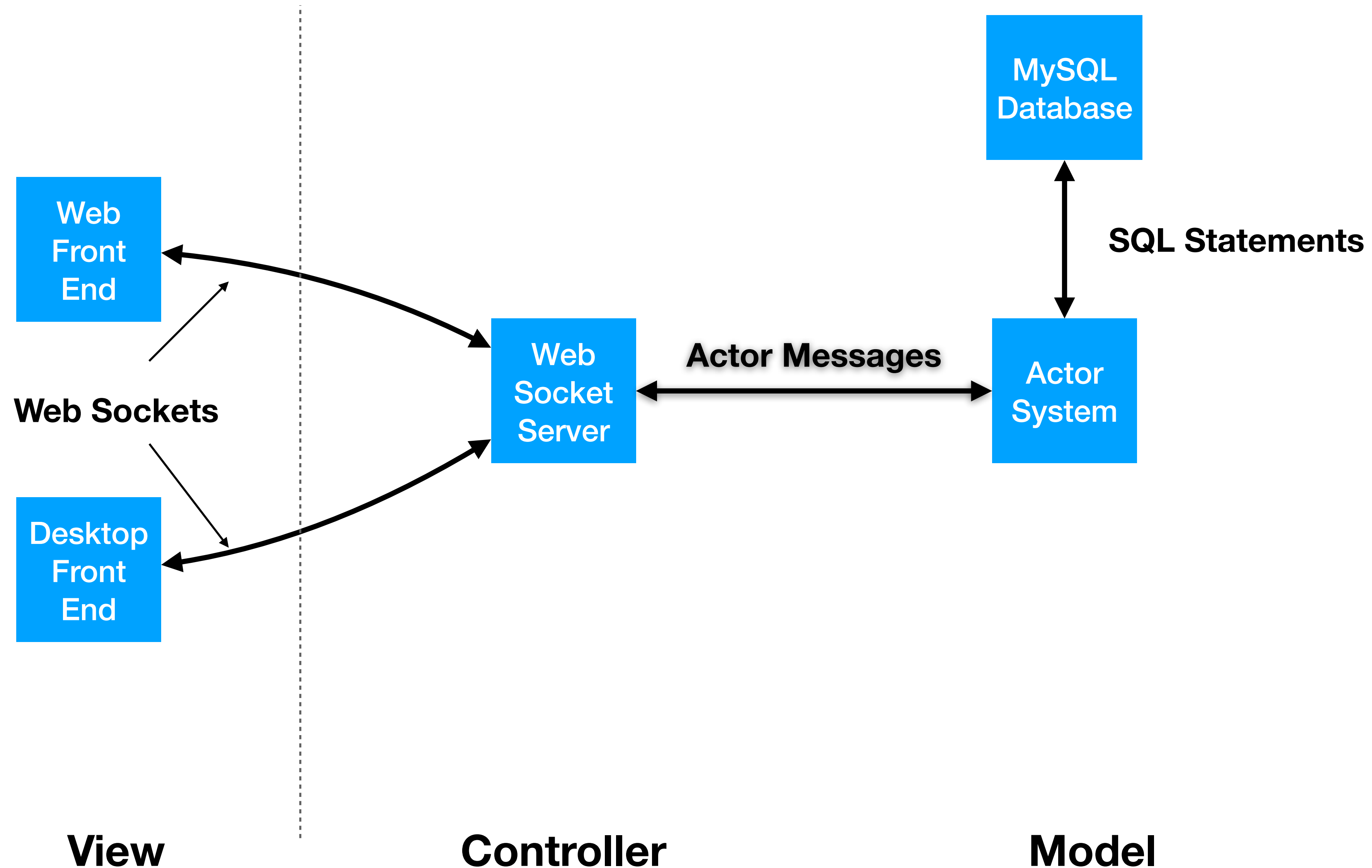- Model API

  - Methods are called by the controller (Correlate with the button presses directly)

  - displayNumber(): Double is called by the view to determine what should be displayed to the user

- Controller

  - Each button on the calculator has an event handler that calls the appropriate model API method

- View

  - Uses a grid pane for more control over element placement

  - Separate CalculatorButton class to easily change the appearance of all buttons

  - Calls displayNumber to update the display whenever the mouse is clicked on the GUI
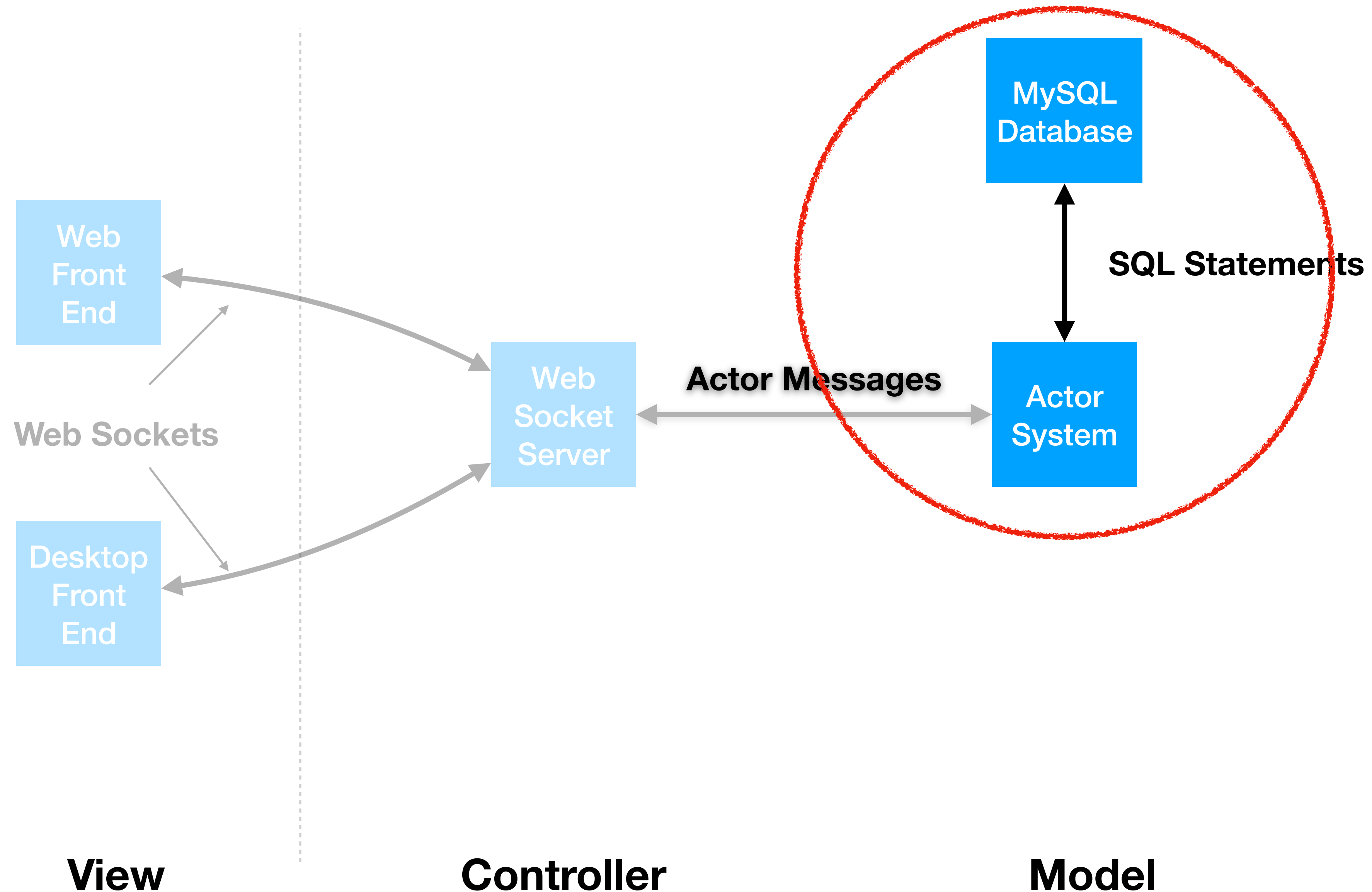
MVC - Calculator

- Model is not aware of ScalaFX

- If we want to build a GUI using a different library

  - No need to change the model at all

  - Build a new view and controller to call the same model API methods

Databases

MMO Architecture

MySQL
Database

SQL Statements

Web
Front
End

Web Sockets

Web
Socket
Server

Actor Messages

Actor
System

Desktop
Front
End

View   **Controller**   **Model**

MMO Architecture

MySQL
Database

SQL Statements

Web
Front
End

Web Sockets

Desktop
Front
End

Web
Socket
Server

Actor Messages

Actor
System

View　　　　　　**Controller**　　　　　　**Model**

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

```scala
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)

  - We can also store JSON strings in MySQL