WebSocket Server

The Problem

- In CSE115 you used HTTP request/responses to build web apps
- If you wanted more data from the server after the page loads, you used AJAX
  - Server hosts JSON data at certain end points
  - Client makes an AJAX call to retrieve the most current data
- But the server has to wait for a request before sending a response

The Problem

- What if the server wants to send time-sensitive data without waiting for a request?
- In CSE115
  - Built a chat app using polling
  - Client sent AJAX requests at regular intervals
  - Only get updates when AJAX request is sent
- Can use long-polling
  - Server hangs on poll requests until it has new data to send

Web Sockets

- A newer solution (Standardized in 2011)
- Establishes a lasting connection
  - Enables 2-way communication between server and client
- Server can push updates to clients over the web socket without waiting for the client to make a new request

socket.io

- A library built on top of Web Sockets
- Maintains connections and reconnecting
- Uses message types
  - Similar to actors, except the message type is always a string
- Add listeners to react to different message types
  - Receiving a message is an event
  - Listener code will be called when the event occurs

socket.io Server in Scala

- New library
- Link on the course website
- Dependency included in pom.xml in examples repo

Web Socket Server

- Import from the new library
- Setup and start the server

```
import com.corundumstudio.socketio.listener.{ConnectListener, DataListener, DisconnectListener}
import com.corundumstudio.socketio.{AckRequest, Configuration, SocketIOClient, SocketIOServer}

class Server() {

  val config: Configuration = new Configuration {
    setHostname("localhost")
    setPort(8080)
  }

  val server: SocketIOServer = new SocketIOServer(config)

  server.addConnectListener(new ConnectionListener())
  server.addDisconnectListener(new DisconnectionListener())
  server.addEventListener("chat_message", classOf[String], new MessageListener())

  server.start()
}
```

Web Socket Server

- Create a configuration object for the server
- This server will run on localhost port 8080

```
import com.corundumstudio.socketio.listener.{ConnectListener, DataListener, DisconnectListener}
import com.corundumstudio.socketio.{AckRequest, Configuration, SocketIOClient, SocketIOServer}

class Server() {

  val config: Configuration = new Configuration {
    setHostname("localhost")
    setPort(8080)
  }

  val server: SocketIOServer = new SocketIOServer(config)

  server.addConnectListener(new ConnectionListener())
  server.addDisconnectListener(new DisconnectionListener())
  server.addEventListener("chat_message", classOf[String], new MessageListener())

  server.start()
}
```

Web Socket Server

- Create and start the server
- Use the configuration to tell the library how to setup the server
- Call the start() method to start listening for connections

```
import com.corundumstudio.socketio.listener.{ConnectListener, DataListener, DisconnectListener}
import com.corundumstudio.socketio.{AckRequest, Configuration, SocketIOClient, SocketIOServer}

class Server() {

  val config: Configuration = new Configuration {
    setHostname("localhost")
    setPort(8080)
  }

  val server: SocketIOServer = new SocketIOServer(config)

  server.addConnectListener(new ConnectionListener())
  server.addDisconnectListener(new DisconnectionListener())
  server.addEventListener("chat_message", classOf[String], new MessageListener())

  server.start()
}
```

Web Socket Server

- Add listeners to handle different event types
- Connect and disconnect listeners to react to clients connecting and disconnecting
- Event listeners for each different message type received from clients

```
import com.corundumstudio.socketio.listener.{ConnectListener, DataListener, DisconnectListener}
import com.corundumstudio.socketio.{AckRequest, Configuration, SocketIOClient, SocketIOServer}

class Server() {

  val config: Configuration = new Configuration {
    setHostname("localhost")
    setPort(8080)
  }

  val server: SocketIOServer = new SocketIOServer(config)

  server.addConnectListener(new ConnectionListener())
  server.addDisconnectListener(new DisconnectionListener())
  server.addEventListener("chat_message", classOf[String], new MessageListener())

  server.start()
}
```

Web Socket Server

- For connect and disconnect
  - Create classes overriding ConnectListener and DisconnectListener
  - Implement the onConnect/onDisconnect methods
- These methods take a reference to the sending socket as a parameter
  - Can use this reference to send messages to the client
  - Usually want to store each reference to send messages later

```
  server.addConnectListener(new ConnectionListener())
  server.addDisconnectListener(new DisconnectionListener())
```

```
class ConnectionListener() extends ConnectListener {
  override def onConnect(socket: SocketIOClient): Unit = {
    println("Connected: " + socket)
  }
}
```

```
class DisconnectionListener() extends DisconnectListener {
  override def onDisconnect(socket: SocketIOClient): Unit = {
    println("Disconnected: " + socket)
  }
}
```

Web Socket Server

- To receive messages, specify the message type and the class of the message
  - Create classes extending DataListener[message_type]
- For the message class we'll use String to receive text data

```
  server.addEventListener("chat_message", classOf[String], new MessageListener())
```

```
class MessageListener() extends DataListener[String] {
  override def onData(socket: SocketIOClient, data: String, ackRequest: AckRequest): Unit = {
    println("received message: " + data + " from " + socket)
    socket.sendEvent("ACK", "I received your message of " + data)
  }
}
```

Web Socket Server

- The DataListeners must implement onData with parameters:
  - A socket reference. Can be used to lookup a user after storing this reference on connection/registration
  - data with type matching the class of the message. This is the content of the message received
  - AckRequest. Not used in this course

```
  server.addEventListener("chat_message", classOf[String], new MessageListener())
```

```
class MessageListener() extends DataListener[String] {
  override def onData(socket: SocketIOClient, data: String, ackRequest: AckRequest): Unit = {
    println("received message: " + data + " from " + socket)
    socket.sendEvent("ACK", "I received your message of " + data)
  }
}
```

Web Socket Server

- Use the reference to the Socket to send messages to the client
- Specify the type of the message as a String
- If the message contains data, use a second String

```
  server.addEventListener("chat_message", classOf[String], new MessageListener())
```

```
class MessageListener() extends DataListener[String] {
  override def onData(socket: SocketIOClient, data: String, ackRequest: AckRequest): Unit = {
    println("received message: " + data + " from " + socket)
    socket.sendEvent("ACK", "I received your message of " + data)
  }
}
```

Web Socket Server

- If responding to message with no data (Similar to case object)
  - Use Nothing as the type

```
server.addEventListener("ping", classOf[Nothing], new StopListener(this))
```

```
class StopListener(server: Server) extends DataListener[Nothing] {
  override def onData(socket: SocketIOClient, data: Nothing, ackRequest: AckRequest): Unit = {
    socket.sendEvent("pong")
  }
}
```

Web Clients

Web Socket Clients

- We've set up a web socket server that will listen for connections and process messages
- Now, let's build a web socket client that will connect to the server

WebSocket Client - Web

- First, setup the HTML
- Layout and style of the page
  - Could add CSS for more style

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web Socket Client Example</title>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
</head>
<body>


<input type="text" id="chat_input"/>
<button id="gold" onclick="sendMessage();">Submit</button>

<div id="display_message"></div>

<script src="WebClient.js"></script>

</body>
</html>
```

WebSocket Client - Web

- Download the socket.io JavaScript client library
- This library contains all the code we'll need to connect to our server

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web Socket Client Example</title>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
</head>
<body>


<input type="text" id="chat_input"/>
<button id="gold" onclick="sendMessage();">Submit</button>

<div id="display_message"></div>

<script src="WebClient.js"></script>

</body>
</html>
```

WebSocket Client - Web

- Add elements for the user to enter and send a message
- In JavaScript, we'll implement the sendMessage() function

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web Socket Client Example</title>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
</head>
<body>


<input type="text" id="chat_input"/>
<button id="gold" onclick="sendMessage();">Submit</button>

<div id="display_message"></div>

<script src="WebClient.js"></script>

</body>
</html>
```

WebSocket Client - Web

- Download our JavaScript file
- This script runs code to connect to the server as soon as it's downloaded
  - Include this at the end of the body so the page loads before connecting to the server

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web Socket Client Example</title>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
</head>
<body>


<input type="text" id="chat_input"/>
<button id="gold" onclick="sendMessage();">Submit</button>

<div id="display_message"></div>

<script src="WebClient.js"></script>

</body>
</html>
```

WebSocket Client - Web

- In WebClient.js
- Call io.connect (from the library) to connect to the server
  - Returns a reference to the created socket

```
const socket = io.connect("http://localhost:8080", {transports: ['websocket']});

socket.on('ACK', function (event) {
    document.getElementById("display_message").innerHTML = event;
});

function sendMessage() {
    let message = document.getElementById("chat_input").value;
    document.getElementById("chat_input").value = "";
    socket.emit("chat_message", message);
}
```

WebSocket Client - Web

- Define how the socket will react to different message types with the "on" method
- The "on" method takes the message type and a function as arguments
  - Call the function whenever a message of that type is received from the server

```
const socket = io.connect("http://localhost:8080", {transports: ['websocket']});

socket.on('ACK', function (event) {
    document.getElementById("display_message").innerHTML = event;
});

function sendMessage() {
    let message = document.getElementById("chat_input").value;
    document.getElementById("chat_input").value = "";
    socket.emit("chat_message", message);
}
```

WebSocket Client - Web

- The function should take a parameter which will contain the data of the message if there is any
- We receive an ACK message containing a string which we display on the page

```
const socket = io.connect("http://localhost:8080", {transports: ['websocket']});

socket.on('ACK', function (event) {
    document.getElementById("display_message").innerHTML = event;
});

function sendMessage() {
    let message = document.getElementById("chat_input").value;
    document.getElementById("chat_input").value = "";
    socket.emit("chat_message", message);
}
```

WebSocket Client - Web

- To send a message, call emit
- Takes the message type and the content of the message, if any
- Can call emit with only message type to send a message with no content (Similar to case object)

```
const socket = io.connect("http://localhost:8080", {transports: ['websocket']});

socket.on('ACK', function (event) {
    document.getElementById("display_message").innerHTML = event;
});

function sendMessage() {
    let message = document.getElementById("chat_input").value;
    document.getElementById("chat_input").value = "";
    socket.emit("chat_message", message);
}
```

Desktop Clients

WebSocket Client - Scala

- Another new library!
- We'll use the Scala/Java version of the socket.io client Library
  - Follows the same structure as the web client
- Add to pom.xml and use maven to download
  - Included in examples repo

Web Socket Client - Scala

- Import relavent code from the socket.io library
- Use IO.socket to create a socket
  - Returns a reference to the created socket
- Call connect() to connect to the server

```scala
import io.socket.client.{IO, Socket}
import io.socket.emitter.Emitter

class ProcessMessageFromServer() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    val message = objects.apply(0).toString
    println(message)
  }
}

object SimpleClient{
  def main(args: Array[String]): Unit = {
    val socket: Socket = IO.socket("http://localhost:8080/")
    socket.on("ACK", new ProcessMessageFromServer())

    socket.connect()
    socket.emit("chat_message", "hello")
    socket.close()
  }
}
```

Web Socket Client - Scala

- Call the "on" method to define the behavior for each message type received from the server
  - Takes a message type and an object that extends Emitter.Listener
  - Implement call(Object*)

```
import io.socket.client.{IO, Socket}
import io.socket.emitter.Emitter

class ProcessMessageFromServer() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    val message = objects.apply(0).toString
    println(message)
  }
}

object SimpleClient{
  def main(args: Array[String]): Unit = {
    val socket: Socket = IO.socket("http://localhost:8080/")
    socket.on("ACK", new ProcessMessageFromServer())

    socket.connect()
    socket.emit("chat_message", "hello")
    socket.close()
  }
}
```

Web Socket Client - Scala

- Implement call(Objects*) which is called with the content of the message as an Array (sort of) of Objects
  - The library is written in Java and uses Java's Object class
- Object contains a toString method so we access the first element and convert it to a String to process the content of the message
  - If there is no content to the message this will throw an index out of bounds error

```
import io.socket.client.{IO, Socket}
import io.socket.emitter.Emitter

class ProcessMessageFromServer() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    val message = objects.apply(0).toString
    println(message)
  }
}

object SimpleClient{
  def main(args: Array[String]): Unit = {
    val socket: Socket = IO.socket("http://localhost:8080/")
    socket.on("ACK", new ProcessMessageFromServer())

    socket.connect()
    socket.emit("chat_message", "hello")
    socket.close()
  }
}
```

Web Socket Client - Scala

- Send messages to the server using the emit method
  - Same syntax as the web version of socket.io

```scala
import io.socket.client.{IO, Socket}
import io.socket.emitter.Emitter

class ProcessMessageFromServer() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    val message = objects.apply(0).toString
    println(message)
  }
}

object SimpleClient{
  def main(args: Array[String]): Unit = {
    val socket: Socket = IO.socket("http://localhost:8080/")
    socket.on("ACK", new ProcessMessageFromServer())

    socket.connect()
    socket.emit("chat_message", "hello")
    socket.close()
  }
}
```

Web Socket Client - Scala

- If you need to interact with a ScalaFX GUI when a socket message is received, call Platform.runLater
- Platform.runLater will run your method on the same thread as the GUI
- This allows you to access the GUI elements/variables from your Emitter.Listener

```
class ServerStopped() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    Platform.runLater(() => {
      GUIClient.textOutput.text.value = "The server has stopped"
    })
  }
}


object GUIClient extends JFXApp {

  // ...

  socket.on("server_stopped", new ServerStopped)

  // ...

  val textOutput: Label = new Label

  // ...
}
```

Web Socket Client - Scala

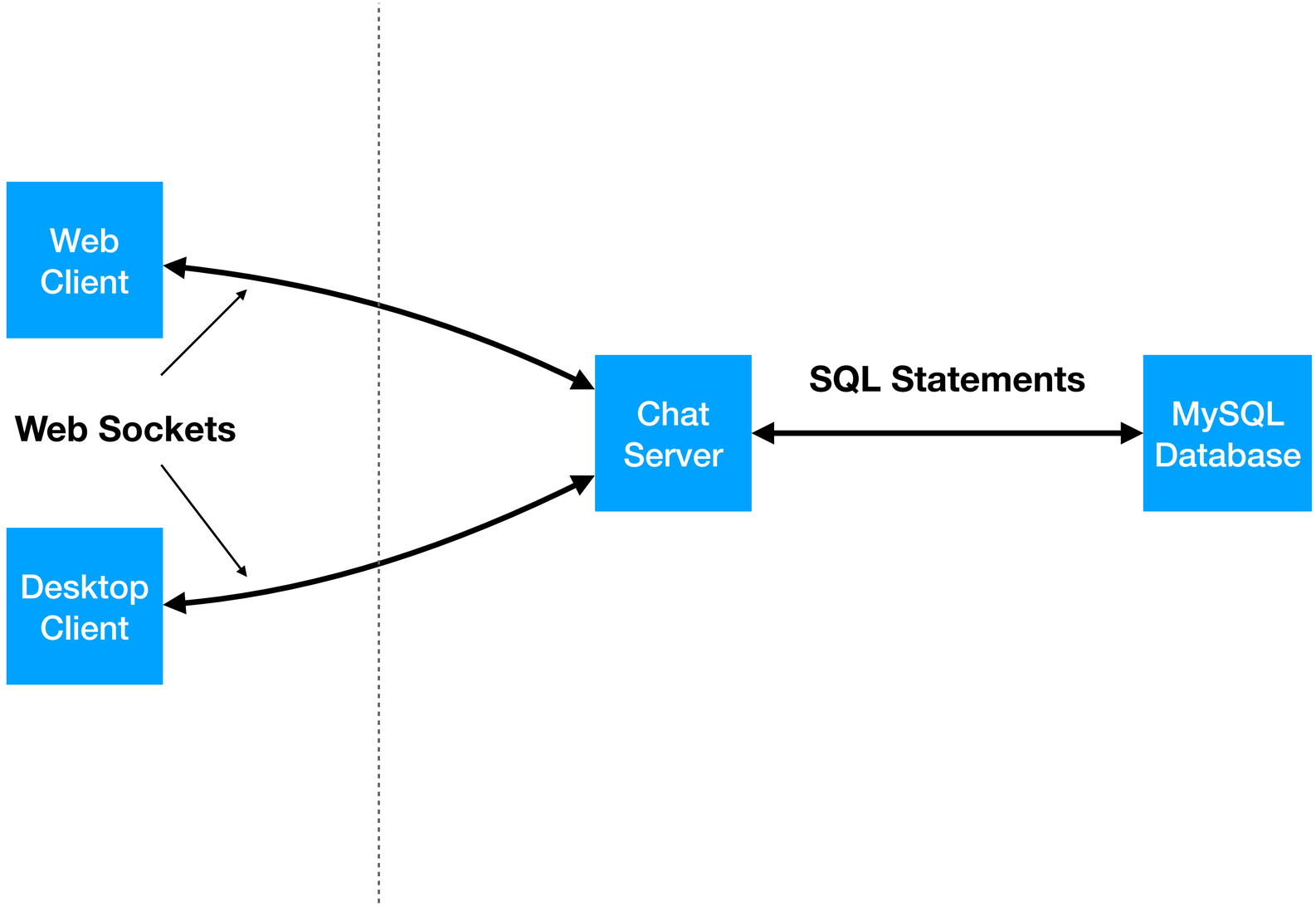
- Takes an object extending Runnable with a method named run with no parameters and return type Unit
- Using Scala syntax to condense this inheritance
  - This syntax can be used when extending a trait with a single method
  - Can create your listeners and event handlers with this syntax if you'd prefer

```
class ServerStopped() extends Emitter.Listener {
  override def call(objects: Object*): Unit = {
    Platform.runLater(() => {
      GUIClient.textOutput.text.value = "The server has stopped"
    })
  }
}


object GUIClient extends JFXApp {

  // ...

  socket.on("server_stopped", new ServerStopped)

  // ...

  val textOutput: Label = new Label

  // ...
}
```

Chat Demo

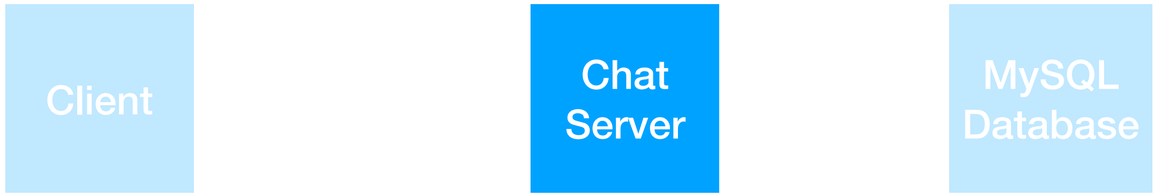
- Let's build a chat app!
  - Code is in the repo
- Users can connect to the chat server
  - Use a web or desktop front end
  - Server doesn't care what type of app a client is using
- All connected users can communicate through text messages

Chat Architecture



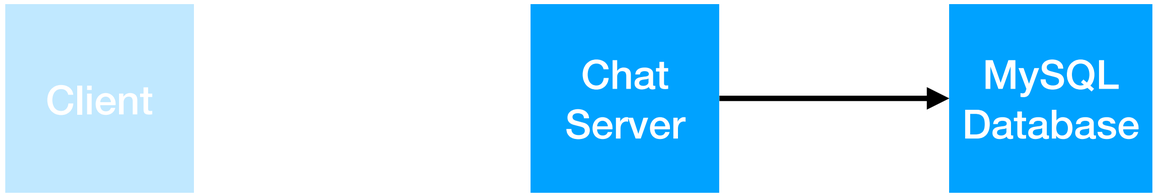
Chat App

- Chat server starts up
- Listens for WebSocket connections on port 8080
- Initialize data structures that will store references to each WebSocket



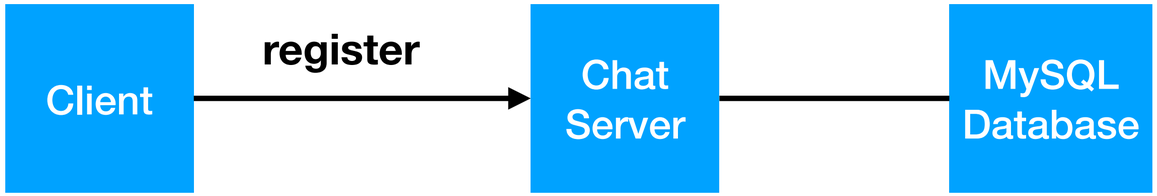
Chat App

- Server connects to a MySQL database to store the chat history
- Communicates via SQL statements
  - MySQL reacts to the event of receiving a statement



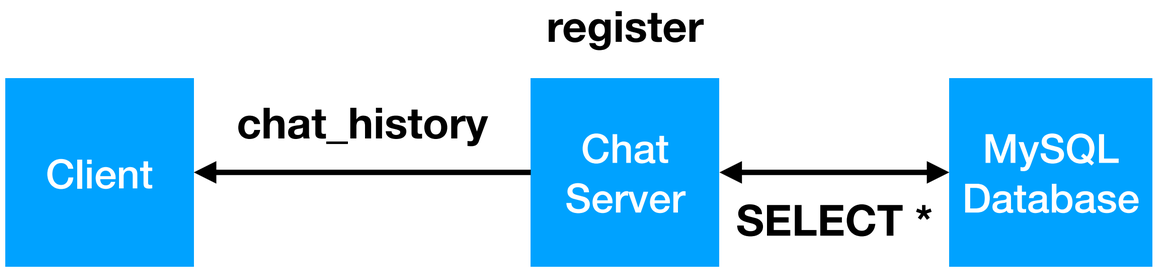
Chat App

- Clients connect to the server using WebSockets
  - Client could be web or desktop
- After the connection is established:
  - Client sends a message of type register containing their username



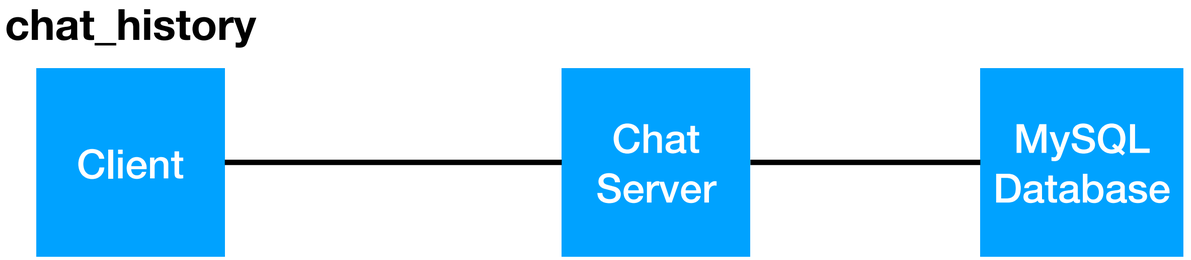
Chat App

- The server receives the register message and reacts to this event
- Adds the new user to the data structures
  - Data structure remembers the username associated with this socket
- Retrieve the chat history from the database and send it to the client



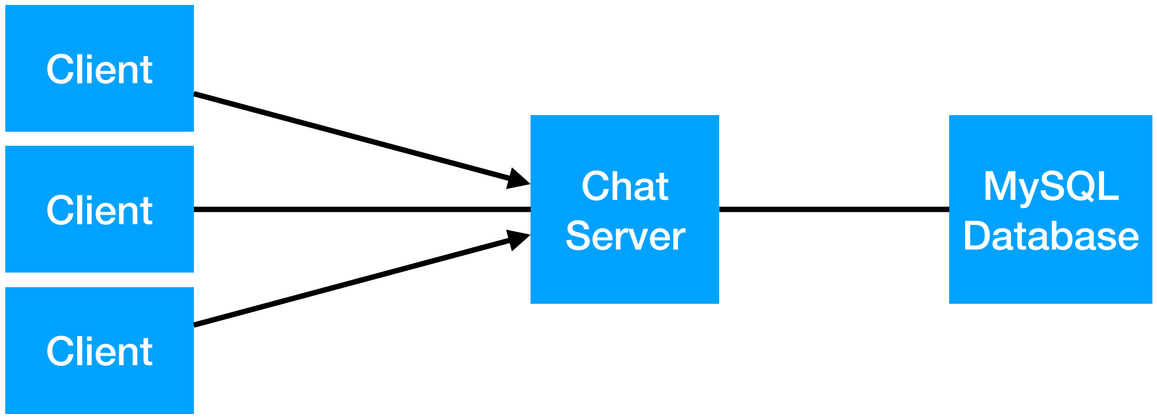
Chat App

- Client reacts to the chat_history message
  - Renders all the content and displays it to the user



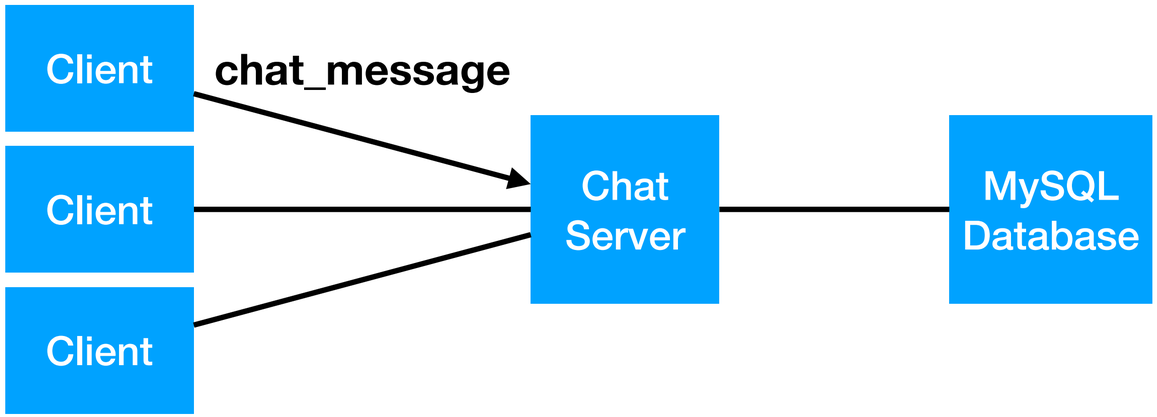
Chat App

- Multiple clients can be connected simultaneously
- Each client sends their username in a register message
- Chat server maps usernames to sockets for all connections



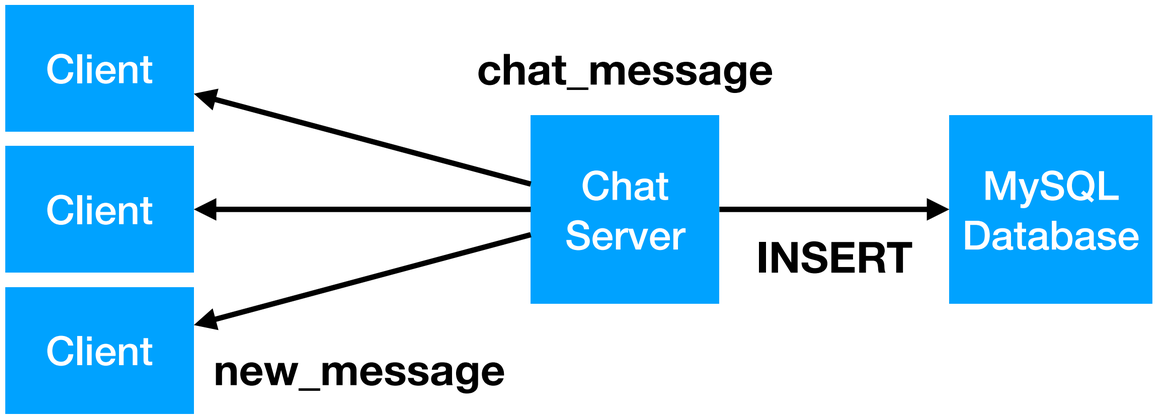
Chat App

- All users can send messages of type chat_message to the server
  - Message is sent when a user sends a message using the GUI
- This message only contains the message (No username)



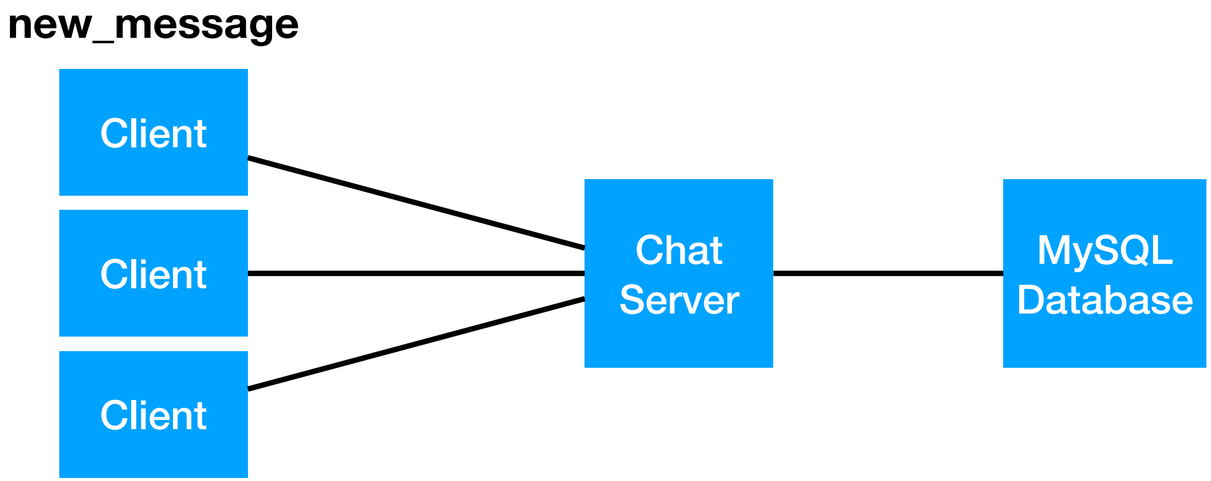
Chat App

- When the server receives a chat_message:
  - Lookup the username for the sending socket
  - Store username/message in the database
  - Send username/message to all connected sockets in a message of type new_message



Chat App

- Clients receive the new_message
- Add it to the GUI for the user to read



Chat App

- When a client disconnects the server reacts to the disconnect event
- Remove the user from data structures