AKKA – Distributed Systems Using HTTP

Guessing Game Example

- Simple game: Guess a number between 1 and 100
- Game responds with:
  - Correct
  - Too High
  - Too Low
  - Invalid Request

Game Server

```
static void startGameServer (Route route, ActorSystem<?> system) {
    CompletionStage <ServerBinding> futureBinding =
        Http.get (system).newServerAt ("localhost", 8080).bind (route);

    futureBinding.whenComplete ((binding, exception) -> {
        if (binding != null) {
            System.out.println ("Server online at http://localhost:8080");
        } else if (exception != null) {
            System.err.println ("Error starting server: " + exception.getMessage ());
        }
    });
}
```

New Constructs

- CompletionStage
  - Similar to a Future, but with the possibility of separate stages which could allow for intermediate results
- Http.get (ActorSystem).newServerAt (address, port).bind (Route)
  - Creates a new HTTP server “actor” with the AKKA actor system
  - Bound to the specified address and port
  - Using the specified Route (more on routes later)
- futureBinding.whenComplete ((binding, exception) -> *lambda*);
  - Method to be executed when the server has completed binding to the specified port

Server Creation

```
public static void main (String[] args) {
    Behavior <NotUsed> baseBehavior = Behaviors.setup (context -> {
        GuessRoutes routes = new GuessRoutes ();
        startGameServer (routes.guessRoutes(), context.getSystem ());

        return Behaviors.empty ();
    });

    ActorSystem.create (baseBehavior, "GuessingGameServer");
}
```

Behavior to create the GameServer

Initialize the system

Guess Behavior - Stateless

```
static String guessNumber (int guess, int answer) {
    if (guess == answer) return WIN;

    else if (guess < answer) return LOW;

    else if (guess > answer) return HIGH;

    else return ERROR;
}
```

Game Logic

Routing

```
public class GuessRoutes {

  public Route guessRoutes () {

    return pathSingleSlash (() ->

      post (() ->

        parameter ("guess", guess ->

          complete (Guess.guessNumber (

                        Integer.parseInt (guess), 20))

    )));

  }

}
```

Create a simple Route with only one possible path

Specify the result of the path

Lambda’s can quickly lead to “parens hell”, be careful

Routes

- A Route is used to specify how to parse the URL/data provided as part of any HTTP message
- There are many directives that can be used to break the data up into manageable pieces
- The ones used in the example are:
  - `pathSingleSlash(action)` – matches a URL that starts at the root level (*127.0.0.1/*)
  - `post(action)` – matches only a POST HTTP message
  - `parameter(value_name, action)` – checks the message for a specific data item and then performs the action on the value of that item
- Example Message – `127.0.0.1:8080/?guess=50`

Directives

- There a many directives, review the AKKA documentation for a complete list.

Predefined Directives (alphabetically) • Akka HTTP

- You can also make your own, though that is beyond what we will do in this course.

Code Walkthrough

Run the example application and ask questions

Message Used to Send a Guess

```
POST http://127.0.0.1:8080/?guess=50 HTTP/1.1
```

Refactoring

- While the current game works, it is using the AKKA actor system in name only
- What do you feel is missing?

Refactoring

- While the current game works, it is using the AKKA actor system in name only
- What do you feel is missing?

- *There are no messages*
- *Actors aren't really used*

- Refactor the system to use messages between the actors

GameServer Refactor

```
public static void main (String[] args) {
  Behavior <NotUsed> baseBehavior = Behaviors.setup (context -> {
    ActorRef<Guess.Command> guessActor =
                            context.spawn (Guess.create(), "Guess");
    GuessRoutes routes = new GuessRoutes (context.getSystem(), guessActor);
    startGameServer (routes.guessRoutes(), context.getSystem ());

      return Behaviors.empty();
  });

  ActorSystem.create (baseBehavior, "GuessingGameServer");
}
```

Create an Actor to manage to manage the guess

Guess Actor – State and Construction

```
public class Guess extends AbstractBehavior <Guess.Command> {

    sealed interface Command {}

    public final static record GuessResult (String result) {}

    public final static record GuessNumber (int guess, ActorRef<GuessResult> replyTo) implements Command {};

    private static final int MAX_ROUNDS = 6;
    private static final String LOSE = "Out of turns";
    private static final String WIN = "You guessed the number!";
    private static final String HIGH = "Your guess was too high";
    private static final String LOW = "Your guess was too low";
    private static final String ERROR = "Invalid Guess";
    private final int MIN = 1;
    private final int MAX = 100;
    private final int rounds;
    private final int answer;


    private Guess (ActorContext<Command> context) {
        super(context);
        rounds = 0;
        answer = new Random ().nextInt (MIN, MAX);
    }
```

Create an interface – why?

`record` creates a data only class

Guess Actor – Behavior and Receive

```
public static Behavior<Command> create() {
    return Behaviors.setup (Guess::new);
}

@Override
public Receive<Command> createReceive() {
    return newReceiveBuilder()
        .onMessage (GuessNumber.class, this::onGuessNumber)
        .build ();
}
```

Standard Actor setup we all know and love

Guess Actor – Guess Behavior

```
private Behavior<Command> onGuessNumber (GuessNumber guess) {
    GuessResult result;
    if (guess.guess() == answer) {
        result = new GuessResult (WIN);
    }
    else if (guess.guess() < answer) {
        result = new GuessResult (LOW);
    }
    else if (guess.guess() > answer) {
        result = new GuessResult (HIGH);
    }
    else {
        result = new GuessResult (ERROR);
    }

    guess.replyTo ().tell (result);

    return this;
}
```

Same logic, just in the Behavior now

Send the result to the actor that sent the message

GuessRoutes – State and Construction

```
public class GuessRoutes {
  private final ActorRef<Guess.Command> guessActor;
  private final Duration askTimeout;
  private final Scheduler scheduler;

  public GuessRoutes (ActorSystem<?> system,
                     ActorRef <Guess.Command> guessActor) {
    this.guessActor = guessActor;
    askTimeout = Duration.ofSeconds (5);
    scheduler = system.scheduler();
  }
```

Actor that message will be sent to

The `ActorSystem` scheduler can be used to run tasks in a separate thread

GuessRoutes - AskPattern

```
private CompletionStage<Guess.GuessResult> guess (int number) {
    return AskPattern.ask (guessActor, ref ->
        new Guess.GuessNumber(number, ref), askTimeout, scheduler);
}
```

- The `AskPattern` is a standard way to manage synchronous requests, most often with entities that are outside of the actor system.
- The pattern create a new actor (`ref`) that will receive the response from a message (`Guess.GuessNumber`) being sent to a given actor (`guessActor`)
- The new actor is wrapped in a `CompletionStage` (`Future` for actors). If the new actor (`ref`) receives a response in the timeout window, the `CompletionStage` will return the result.

GuessRoutes - Routes

```
public Route guessRoutes () {
  return
    pathSingleSlash (() ->
      post (() ->
        parameter ("guess", guess ->
          onSuccess (guess (Integer.parseInt (guess)),
            guessResult -> {
              return complete (guessResult.result());
    }))));
}
```

Root Path

Only POST Messages

?guess=number

Create a new guess which waits for a response

Return response

Code Walkthrough

Run the example application and ask questions

Message Used to Send a Guess

```
POST http://127.0.0.1:8080/?guess=50 HTTP/1.1
```

Refactoring

- Everything works, but is there anything that feels off or odd?

Refactoring

- Everything works, but is there anything that feels off or odd?

- *Using get and post via the URL feel outdated and potentially insecure*
- *All information is received as strings*
- *Sending multiple pieces of data will be cumbersome*

- Refactor the system to send and receive JSON Messages

Guess - Refactor

```
public final static record AGuess (int guess) {}
```

Add a record that contains all the data in a guess

GuessRoute - Refactor

```
public Route guessRoutes () {
  return
    pathSingleSlash (() ->
      post (() ->
        entity (Jackson.unmarshaller (AGuess.class), guess ->
          onSuccess (guess (guess.guess()), guessResult -> {
            return complete (StatusCodes.OK, guessResult,
Jackson.marshaller());
    }))));
}
```

`Jackson.unmarshaller` turns JSAON data into and instance of `AGuess`

`Jackson.marshaller` turns `guessResult` into a JSON message

Code Walkthrough

Run the example application and ask questions

Message Used to Send a Guess

```
POST http://127.0.0.1:8080/ HTTP/1.1
content-type: application/json


{"guess": 50}
```