Lab 1. Introduction

1.1 Our first actor

Let us create our first app to get the feel of working with actors.

Working package: **com.luxoft.akkalabs.day1.firstactor**

1.2 Create UntypedActor

- Create **MyFirstActor** by extending the **UntypedActor**.
- Create **onReceive** method
- In **onReceive** write a code that expects a **String** message "**ping**".
- Write a code that reacts on "**ping**". For example, writes something to system console.

1.3 Create a main class

- **FirstActorApp.java** contains a template of a main class.
- Write a code that:
  - Creates an actor system.
  - Tells actor something that it expects.
  - Put current thread to sleep for a short time*.
  - Shut down the system.
- Run the app and see how actor reacts on your message.

*It is necessary to give an actor some time to start up and react on your message. The simplest way is to put the main thread to sleep for a short time before shutting down the system. If you avoid the system shutdown, your app will not exit upon completing the main method because Akka spawned a few of its own threads.

```
ActorSystem system = ActorSystem.create();

ActorRef actor = system.actorOf(Props.create(MyFirstActor.class));
actor.tell("ping", null);

Thread.sleep(1000);
system.shutdown();
```

Lab 2. Futures

2.1 Tweet languages

Let us count which languages people use to tweet about Apple and Google with help of Twitter Client provided with lab materials.

Working package: **com.luxoft.akkalabs.day1.futures**

Our app will work in the following way:

- First, we collect a few tweets for each topic. **TwitterClient** will help us with this.
- Then we analyze collected tweets and count how many times different languages were used in the sample.
- Finally, we present results for each topic.

For efficiency, our app will collect samples for each topic in parallel by using Futures.

2.2 Main class

Open **AppleVsGoogle** class. In the **main** method start the Actor System:

```
ActorSystem system = ActorSystem.create("AppleVsGoogle");
```

2.3 Result class

Result class will serve as a container for tweets collected for some topic.

Result
+keyword: String
+tweets: Collection<String>

2.4 Collector class

Now it is time to write a code that will collect tweets. Since we decided to parallelize tasks, let us wrap this logic into a **Callable**.

Create **CollectTweets** class as implementation of **Callable<Result>**. In the **call()** method fetch 10 tweets for a given keyword, then construct and return the **Result** object.

- Launch a **QueueTwitterClient** by calling **TwitterClients.start(system, keyword)**
- Fetch 10 **TweetObject** objects by calling **next()** on the client.
- Pack all tweets into the **Result** object and return it.
- Do not forgot to shut down the client by calling **stop()** or wrapping it into **try(client){}** section.

2.5 First run: slow one

Now we are ready to collect some tweets.

- First try calling the **CollectTweets** class from the **main** method. First collect results for “**Apple**” keyword, then for “**Google**” keyword.
- Each tweet will be packed into a **TweetObject**. Obtain language by calling **getLanguage()**.

- Calculate languages statistics for both keywords.
- Display results for each keyword.

2.6 Second run: make it fast

It is time to switch to collecting tweets in parallel. This will make our app work up to two times faster.

- **Futures.future()** method will place our **CollectTweets** tasks for parallel execution:

```
Future<Result> futureAppleResult =
    Futures.future(appleTask, system.dispatcher());

Future<Result> futureGoogleResult =
    Futures.future(googleTask, system.dispatcher());
```

- Now we may collect our results by waiting on them:

```
Result appleResult = Await.result(futureAppleResult,
    Duration.create(60, TimeUnit.SECONDS));

Result googleResult = Await.result(futureGoogleResult,
    Duration.create(60, TimeUnit.SECONDS));
```

- Process collected results same as before.

Notice that if our collectors fail to collect tweets within 60 seconds timespan, an exception will be thrown. Not very convenient, isn't it?

2.7 Get the final result instead of intermediate ones

There is a way to arrange our code so we do not wait for multiple intermediate results. We are going to wait for our final stats instead.
First, define the **FinalResult** container for our language stats:

FinalResult
+keyword: String +languages: Map<String,Integer>

Now create **FinalResultCalculator** class that takes **Result** as input and calculates resulting stats into **FinalResult** object.

Do it by extending **Mapper<Result, FinalResult>**:

```
private static class FinalResultCalculator
    extends Mapper<Result, FinalResult> {
        @Override
        public FinalResult apply(Result r) {
…
        }
}
```

Last step is to map the expected result:

```
Future<FinalResult> futureAppleLanguages =
    Futures.future(appleTask, system.dispatcher()).
       map(new FinalResultCalculator(), system.dispatcher());

Future<FinalResult> futureGoogleLanguages =
    Futures.future(googleTask, system.dispatcher()).
       map(new FinalResultCalculator(), system.dispatcher());
```

2.8 Finally, stop waiting on multiple things!

Why wait on many things? Combine them into one **Future**:

```
List<Future<FinalResult>> allResults =
    Arrays.asList(futureAppleLanguages, futureGoogleLanguages);

Future<Iterable<FinalResult>> result =
    Futures.sequence(allResults, system.dispatcher());
```

Now you have to wait only on one future instead of multiple.

2.9 Finally, stop waiting!

If it is possible to consume the result asynchronously, we can get rid of blocking operations completely.

```
result.onSuccess(
  new OnSuccess<Iterable<FinalResult>> () {
     @Override
     public void onSuccess(Iterable<FinalResult> success) {

         Iterator<FinalResult> i = success.iterator();
         …

     }
  }, system.dispatcher());
```

Lab 3. First steps with Actors

3.1 Tweet languages-2

As a warmup let us make an actor that collects all the stats from our twitter client. This simple app will show us how to make concurrent applications without locks or synchronized blocks.

Working package: **com.luxoft.akkalabs.day1.languages**

3.2 Reuse your work

It is important to create reusable code because it saves time and money. In this and following labe we are going to reuse as much code as possible. In this lab, we will be reusing **CollectTweets**, **Result**, **FinalResult** and **FinalResultCalculator**.

3.3 Create an actor

Create **LanguagesCounterActor** class from following template:

```
import akka.actor.UntypedActor;
import com.luxoft.akkalabs.day1.futures.FinalResult;
import java.util.*;

public class LanguagesCounterActor extends UntypedActor {

    private final Map<String, Integer> languages = new HashMap<>();

    @Override
    public void onReceive(Object message) throws Exception {
     …
    }
}
```

Actor should react on **FinalResult** objects by adding their contents to values in the map.
If actor receives "**get**" string, it should reply a sender with a copy of its data. It is important not to expose the original map to the sender.

3.4 Run actor and feed it with data

In the main method of **PopularLanguages** class create the actor system and launch the actor:

```
public static void main(String[] args) throws Exception {
   ActorSystem system = ActorSystem.create("PopularLanguages");

   ActorRef actor = system.actorOf(
        Props.create(LanguagesCounterActor.class));
   …
}
```

This will start an actor and give you a handle to it.

Now, spawn multiple **CollectTweets** tasks for some keywords.

```
List<String> keywords =
   Arrays.asList("Google", "Apple", "Android", "iPhone", "Lady Gaga");

for (String keyword : keywords) {
    Futures.future( new CollectTweets(keyword), system.dispatcher()).
        map( new FinalResultCalculator(), system.dispatcher()).
            onSuccess( new SendToActor(actor), system.dispatcher());
}
```

Now we should collect results from the **Future** and pass it to an actor. We can make it by listening to the **onSuccess** event on the **Future**. Create **SendToActor** class and pass its instance to **onSuccess**:

```
Private static class SendToActor extends OnSuccess<FinalResult> {

    private final ActorRef actor;

    public SendToActor(ActorRef actor) {
        this.actor = actor;
    }

    @Override
    public void onSuccess(FinalResult success) throws Throwable {
        actor.tell(success, null);
    }
}
```

Another and more preferable way to achieve the same functionality is to use the **pipe** pattern.

```
akka.pattern.Patterns.pipe(future, system.dispatcher()).to(actor);
```

3.5 Run and query stats

Now write code that periodically queries an actor for stats and displays it to console.

Use **ask** pattern for that: `Patterns.ask(actorRef, message,timeout)`

```
Timeout timeout = Timeout.apply(1, TimeUnit.SECONDS);
for (int i = 0; i < 30; i++) {
   Thread.sleep(1000);
   Future<Object> future = Patterns.ask(actor, "get", timeout);
   //Get the data and display it. Use Await or future.onComplete.
}
```

Your keyword tasks may take longer than 30 seconds to complete. Stop our actor system after that time:

```
system.shutdown();
```

Lab 4. Wiring actors

4.1 Grab Wikipedia links from tweets

Our next app will scan twitter for links to Wikipedia and display information on topics behind that links.

Working package: **com.luxoft.akkalabs.day1.wikipedia**

Technically speaking our app should:

- Fetch tweets for keyword "**wikipedia**".
- Extract links from these tweets.
- Call Wikipedia API to get a page title behind that link.
- Print the page title to console.

4.2 Wikipedia link grabber actor

Create the **WikipediaActor**. It should receive Strings and treat them like URLs. Use **WikipediaClient** class to fetch page info from Wikipedia.

Note: You may receive any URL, so please check if that URL points to one of **Wikipedia.org** subdomains. Also, extract language version information from URL and pass it to a client.

```
If (message instanceof String) {
    URL url = new URL(message.toString());
    if ( url.getHost().toLowerCase().endsWith(".wikipedia.org")
         && url.getPath().length() > 6) {
            String lang = url.getHost().substring(0, 2)
            String term = url.getPath().substring(6);
            WikipediaPage page = WikipediaClient.getPage(lang, term);
            if (page != null)
                System.out.println(page.getTitle());
        }
    }
```

4.3 Tweets processor

Create an actor that collects URLs from tweets and passes them to a consumer actor.

Create **TweetLinksActor** class that receives an **ActorRef** reference to a consumer actor in a constructor and reacts on **TweetObject**. Each tweet may contain a set of URLs. Feed URLs to the consumer actor.

4.4 Launch everything

In **GrabWikipediaLinksFromTweets**'s main method:

- Start the actor system, and then WikipediaActor.
- Start TweetLinksActor. Provide a reference to WikipediaActor via Props.

Use the following call to begin the flow of tweets:

```
TwitterClient.start(system, tweetsActor, "wikipedia");
```

Lab 5. Wiring more actors

5.1 Show Wikipedia links on the web. In real-time.

Now we will create a web app. It will display a real-time feed of Wikipedia topics.

Working package: **com.luxoft.akkalabs.day1. wikipedia2**

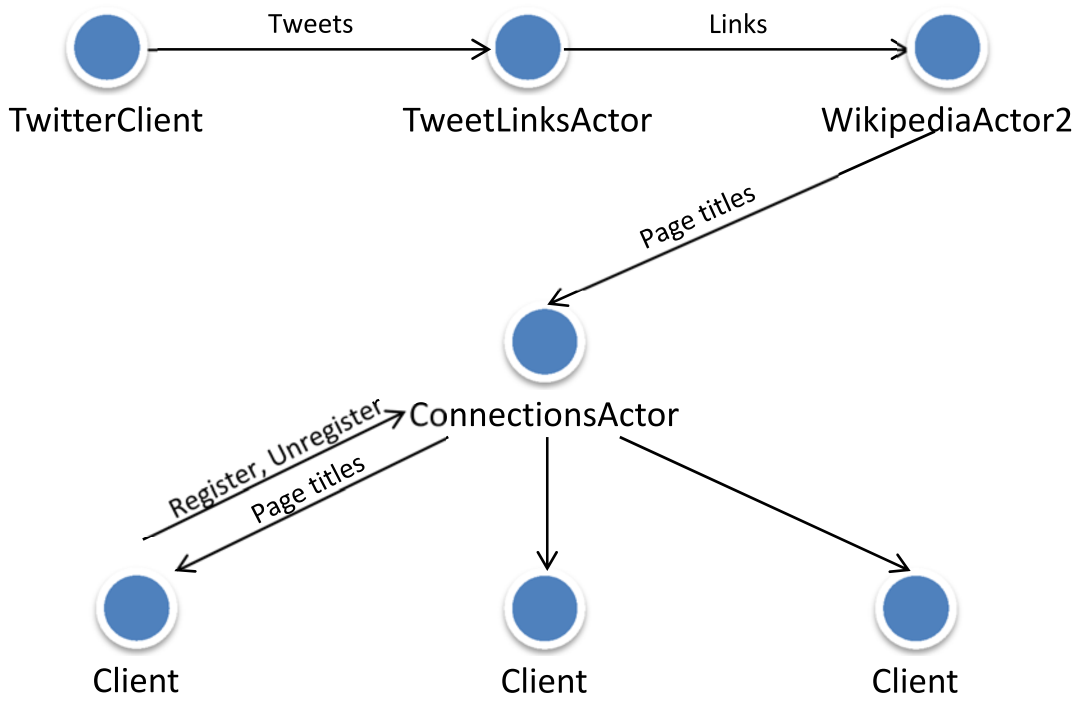
Our app should:

- Fetch tweets for keyword “**wikipedia**”.
- Extract links from tweets.
- Call Wikipedia API to get a page title behind that link.
- Deliver these titles to all currently connected web clients.

We need a way to deliver updates to clients in real time in easy and efficient way. An **EventSource** technology from HTML5 stack will help us with this.

5.2 Plan our app

First, we start planning our messages flow.



TwitterClient and **TweetLinksActor** will be reused from our previous work. **WikipediaActor** from previous lab will be updated to send results to **ConnectionsActor**.

ConnectionsActor will represent a hub. Web clients will use it to subscribe for updates and deliver them back to web browsers. This requires a server infrastructure, which we are going to implement now.

5.3 Create a web app listener

com.luxoft.akkalabs.day1. wikipedia2.web.Init class is an implementation of **ServletContextListener**. This class will be called when our web app starts and stops, so we can initialize and destroy our actor systems.

Implement **contextInitialized** method:

- Create an actor system "AkkaLabs":
  ```
  ActorSystem system = ActorSystem.create("AkkaLabs");
  ```

- Save the ActorSystem instance as a servlet context attribute:
  ```
  sce.getServletContext().setAttribute("actorSystem", system);
  ```

In **contextDestroyed** method:

- Obtain an instance of **ActorSystem** from the context attribute.
- Shut down the actor system.

5.4 Create a servlet for a long-lasting connection

Client's **EventSource** JavaScript object will connect to the server. **com.luxoft.akkalabs.day1. wikipedia2.web.wikitopics.WikipediaStream** is a servlet that will serve that connection.

> EventSource object and servlet both will act as described in http://www.w3.org/TR/eventsource/ recommendation. The whole protocol is very simple. The main idea is that the server keeps the connection open for as long as possible and occasionally sends events to the client. Upon receiving the event data, EventSource notifies JavaScript event handlers. The connection is initiated by the client but essentially is opposite-way: only server sends the data to the client.

The servlet should receive requests to the "**/day1/wikitopics**" and support asynchronous mode:

```
@WebServlet(asyncSupported = true, urlPatterns = {"/day1/wikitopics"})
```

In **doGet** method:

- Set up the response:
  ```
      resp.setContentType("text/event-stream");
      resp.setCharacterEncoding("UTF-8");
  ```
- Switch to a long-living request lifecycle and detach from current thread.
  ```
      AsyncContext context = req.startAsync();
      context.setTimeout(240000);
  ```

From now when **doGet** method completes, the connection be kept running. You will be able to send and receive the data from it asynchronously by passing the **AsyncContext** object to different threads.

Next step is to implement an event handler that will listen on the lifecycle of our connection.

Implement **javax.servlet.AsyncListener** either as internal class or as top-level class. Leave all methods empty for now.

5.5 Build the Connections actor

The next step is to create the **ConnectionsActor**. It will manage client connections and deliver messages:

- Each new connection to WikipediaStream will have to notify this actor about itself.
- After being closed, connection will have to unregister.
- The actor delivers Wikipedia titles to every active connection.

To distinguish different connections assign each connection a unique Id.

5.5.1 Define the protocol

Register
+listener: WikipediaListener

Deliver
+page:WikipediaPage

Unregister
+id: String

To decouple the servlet-specific code from actor code we are going to create an interface **WikipediaListener**. It will serve as an event listener for delivering Wikipedia pages to clients.

<<interface>> **WikipediaListener**
deliver(page: WikipediaPage):void throws NotDeliveredException getStreamId():String close():void

5.5.2 React to messages

Actor must react on three messages:

- On **Register** it saves event listener to its local state.
- On **Deliver** it delivers a page to each saved actor.
- On **Unregister** it forgets about an event listener with given id.

5.5.3 Launch the actor

Open the **Init** listener. Publish the actor into actor system under the "**connections**" name.

```
ActorRef connectionsActor = system.actorOf(
    Props.create(ConnectionsActor.class), "connections");
```

5.6 Obtain Wikipedia pages

Create **WikipediaActor2** that reacts on incoming URLs and deliver results to **ConnectionsActor**. Copy the code from **WikipediaActor** from previous lab.

Since each individual call to the Wikipedia API might take a long time, it may result in a mailbox overflow. To address this problem, launch each task in parallel freeing our actor to process next messages.

```
Futures.future( new Callable<WikipediaPage>() {
    public WikipediaPage call() {
        WikipediaPage page = WikipediaClient.getPage(lang, term);
        if (page != null) {
            system.actorSelection("/user/connections").
               tell(new Deliver(page), null);
        }
        return page;
     }}, system.dispatcher());
```

5.7 Handle Wikipedia pages

Implement the **WikipediaListener**. Make it accept stream id and an **AsyncContext** in constructor.

Implement **deliver** method:

- Obtain (and cache) a response body **Writer** from **AsyncContext**.

  `context.getResponse().getWriter()`

- Obtain JSON representation of a **WikipediaPage** by calling **toJSONString()** on it.
- Write result to a client in the following format:
  `id: some_unique_id_of_the_event\n` – for our app it can be anything
  `data: one line of the data\n` – each data line must begin with that prefix
  `data: next line of the data\n`
  `\n\n` – an empty line will mark the end of the data frame

```
String data = page.toJSONString();
String eventId = Long.toString(System.currentTimeMillis());
String[] lines = data.split("\n");
out.append("id: ").append(eventId).append('\n');
for (String line : lines) {
    out.append("data: ").append(line).append('\n');
}
out.append("\n\n");
context.getResponse().flushBuffer();
```

5.8 Wire it all together

5.8.1 In Init listener

After you create an actor system and launch **ConnectionsActor**:

- Launch **WikipediaActor2**
- Launch **TweetLinksActor** pointing it to the **WikipediaActor2**.
- Launch a **TwitterClient** pointing it to **TweetLinksActor**. Listen to “**wikipedia**” keyword.

```
TwitterClient c =
    TwitterClient.start(system, tweetLinksActor, “wikipedia”);
```

5.8.2 In AsyncListener implementation

- Update class to take stream id and an **ActorSelection** as constructor parameters
- Create a **unregister** method in which send **Unregister** object to **ActorSelection**.
- In **onComplete**, **onError** and **onTimeout** call the **unregister** method. This will notify our hub about closed connection.
- In **onTimeout** also call **complete()** on **AsyncContext** (obtain it from event). This will close the connection properly.

5.8.3 In WikipediaStream servlet

After obtaining an **AsyncResult**:

- Obtain an actor system from the **ServletContext**.
- Make an **ActorSelection** from path “**/user/connections**”.
- Generate an id for current connection. For example, use **UUID** class.

```
String streamId = UUID.randomUUID().toString();
```

- Create a **WikipediaListener** implementation instance. Pass id and **AsyncContext** to constructor.
- Register connection by sending **Register** object to a selection.
- Create an **AsyncListener** implementation instance. Pass id and the **ActorSelection** to constructor.
- Add the listener to **AsyncContext**.

5.9 Run and test the web app

Build, deploy.

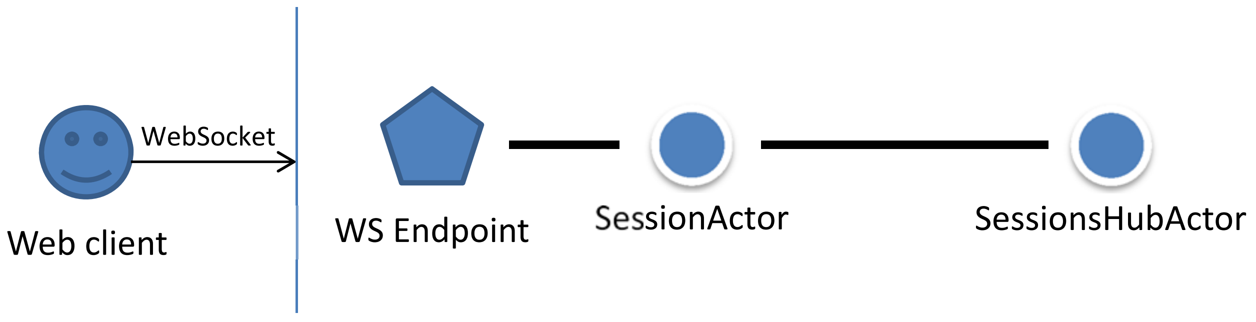
Navigate to **day1/wikipedia-topics.html** and see the results.

Lab 6. WebSocket sessions

6.1 Create the WebSocket infrastructure

In this lab, we are going to build an infrastructure for a few next real-time web apps. We will explore basic concepts of building actor systems and learn a few tricks.

Working package: **com.luxoft.akkalabs.day2.sessions**



WebSockets is a technology for creating two-way connections between web clients and web servers. It allows overcoming limitations of HTTP protocol, such as stateless one-way communication model. Simply speaking it allows client and server to send multiple messages to each other at any time through one TCP connection.

When our client connects to Java EE 7 WebSocket endpoint, the session should register with **SessionsHubActor**. Hub actor then spawns a **SessionActor** for each Session. **SessionActor** processes the incoming messages from the client and outgoing messages from other actors.

6.2 Design the Sessions Hub Actor

Let us define a protocol for this actor.

- Use **RegisterSession** to deliver a new **javax.websocket.Session**.
- **UnregisterSession** will tell actor to forget the closed session.
- Wrap messages to specific session into **OutgoingToSession** message.
- Wrap broadcast messages for delivery to every session into **OutgoingBroadcast** message.

An **Outgoing** interface will generalize various types of outgoing messages.

RegisterSession
sessionId: String session: Session

UnregisterSession
sessionId: String

<<interface>> **Outgoing**
+getMessage(): Object

OutgoingToSession implements Outgoing
-sessionId: String -message: Object
+getSessionId(): String +getMessage(): Object

OutgoingBroadcast implements Outgoing
-message: Object
+getMessage(): Object

6.3 Design the Session Actor

The protocol for session actor is:

- **Incoming** message delivers text packets from web clients.
- **Outgoing** delivers messages to actor for processing and eventually for delivering to a client.

Incoming
+message: String

To make **SessionActor** reusable we should define some sort of pluggable strategy.
SessionProcessor is an interface that accepts notifications about the connection lifecycle events. It may react on when connection starts, stops or receives any type of messages. SessionActor will remain free of any domain specific code. Its job would be to call session processor.

<<interface>> **SessionProcessor**

+started(sessionId:String, c:ActorContext, s:Session)
+stopped()
+incoming(message:String)
+outgoing(message:Object)

6.4 Create the Session Actor

Create a **SessionActor** class. Implement its lifecycle so it notifies given **SessionProcessor** about important events.

- Constructor should accept following arguments:
  - Session id,
  - WebSocket's session object,
  - **SessionProcessor** object.

- **preStart** method should register a special listener on a WebSocket session. This listener will receive the incoming data packets from the client.
  - Create a listener as an implementation of **javax.websocket.MessageHandler.Whole<String>** interface.
  - Listener's **onMessage** method will receive a **String** message.
    It is wrong to pass the incoming packet directly to the session processor because the handler is actually executed in a different thread, so the processor might be busy doing something else at the time the message arrives to listener.
  - Instead, **onMessage** method should wrap incoming text packet in the **Incoming** message and send it to the session actor.

```
self().tell(new Incoming(message), self());
```

- **preStart** also should notify session processor that session has started by passing all objects it might require as method parameters.

- **postStop** method should notify session processor that session has stopped.

onReceive method should react on these messages:

- **Incoming** – unwrap and pass incoming text packet to the session processor.
- **Outgoing** – unwrap and pass outgoing object to the session processor.
- **Any other message** – pass it to the session processor as outgoing.

6.5 Create the Sessions Hub Actor

Create a **SessionsHubActor** class. Accept an argument of type Class<? extends SessionProcessor> through the constructor. This parameter will tell the hub which class processes the session.

Write **onReceive** so it reacts on incoming messages:

- **RegisterSession** – launch a **SessionActor** as a child of hub actor.
  Pass all required constructor parameters inside **Props**. Use **Class.newInstance()** to create fresh instance of **SessionProcessor** implementation. Give that actor a name equal to session id.
- **UnregisterSession** – find a child with given id and stop it.

```
getContext().getChild(request.getSessionId())
```

- **OutgoingToSession** – find a child with given id and forward a message to it.

```
child.forward(message, getContext());
```

- **OutgoingBroadcast** – forward the message to all children.

6.6 Test WebSocket Sessions

6.6.1 Create the EchoSessionProcessor

Implement **SessionProcessor** so it returns every message it receives from incoming channel to outgoing channel:

```
@Override
public void incoming(String message) throws IOException {
    session.getBasicRemote().sendText(message);
}
```

6.6.2 Wire everything

In the **com.luxoft.akkalabs.day2.sessions.web.Init** class:

- Launch **ActorSystem** with name “Echo”
- Save it to "**echoActorSystem**" attribute of the **ServletContext**
- Launch **SessionsHubActor**, pass **EchoSessionProcessor**’s class as argument. Name it “**sessions**”.
- Launch **WebSocket** by using **WebSocketLauncher** helper class.
    - First parameter is **ServletContext**.
    - Second is a path to which websocket will listen.
    - Third is the name of a **ServletContext** attribute to which we saves our ActorSystem.

```
ActorSystem system = ActorSystem.create(ACTOR_SYSTEM_NAME);
servletContext.setAttribute(ACTOR_SYSTEM_KEY, system);

system.actorOf(Props.create(SessionsHubActor.class,
     EchoSessionProcessor.class), "sessions");

WebSocketLauncher.launchSessionEndpoint(
     servletContext, "/day2/echo", ACTOR_SYSTEM_KEY);
```

6.6.3 How the WebSocket endpoint works

WebSocket endpoint will receive incoming messages, wrap them into **Incoming** objects and send them to an actor selection for “**/user/sessions**”.

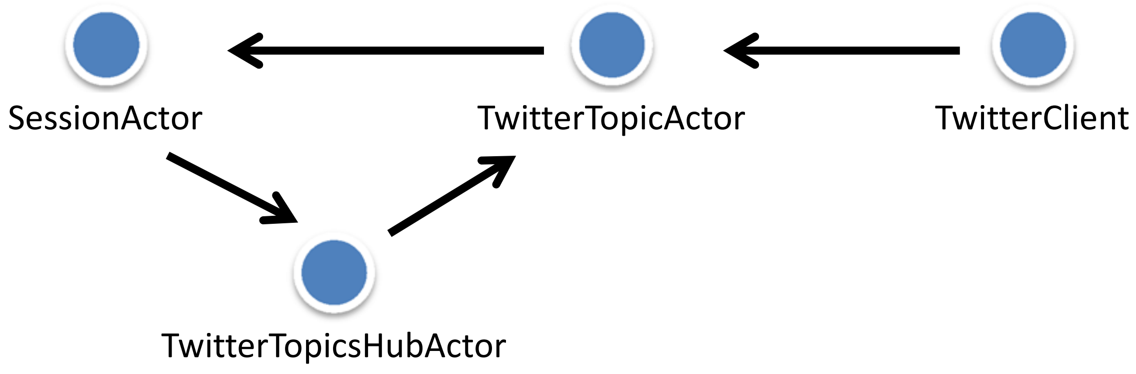
6.6.4 Test in the web

Open **day2/echo.html** page and type something into an input. After submit you should see the same text in the output log.

Lab 7. Twitter Topics

Now having this cool WebSocket infrastructure let us build a simple web app that displays tweets on given topic on the web page in real time.

Working package: **com.luxoft.akkalabs.day2.topics**



This is what we are going to build using our WebSocket sessions and Twitter Client.

- WebSocket session will tell **TwitterTopicsHubActor** to subscribe it to a topic.
- Topics hub will spawn one **TopicActor** per topic and pass registration requests to it.
- **TopicActor** is the actual data source to which session subscribe.
- **TopicActor** itself subscribes for a stream of tweets from TwitterClient and rebroadcast these tweets to every subscribed session.
- When session unsubscribes or disconnects, TopicActor stops delivering tweets to that session.
- If there are no sessions left, then topic actor shut itself down.

7.1 Design the Topics hub

Topics hub follows a pattern that is similar to the one in Sessions actor.

- On **SubscribeToTopic** it will:
  - Check if there is an active topic with this keyword.
    - Spawn a new topic actor if there is no such actor.
  - Forward a message to the topic actor.
- On **UnsubscribeFromTopic**:
  - Check if there is an active topic with this keyword.
  - Forward a message to the topic actor.
- On **TopicIsEmpty:**
  - Check if there is an active topic with this keyword.
  - Notify existing topic that now it may stop by sending **StopTopic** to it.
  - Forget about the topic actor for given keywords.

SubscribeToTopic
keyword: String

UnsubscribeFromTopic
keyword: String

TopicIsEmpty
keyword: String

Q: Why to use such complicated protocol? Why not to allow topics to stop themselves when there are no active sessions left?
A: Every operation takes some time to process. For some time after the actor stops itself the hub will be unaware of this fact and may continue forwarding subscription messages to an invalid ActorRef. Instead, we are going to create a special protocol to ensure that no session request is lost.

7.2 Design the Twitter Topic actor

Twitter topic actor reacts both on subscription messages and on incoming tweets from **TwitterClient**.

- In **preStart** method actor should start **TwitterClient**.
- In **postStop** method actor should stop **TwitterClient**.

In **onReceive** method:

- On **SubscribeToTopic** topic actor will:
  - Add the sender to the set of subscribers
- On **UnsubscribeFromTopic** topic actor will:
  - Remove the sender from the set of subscribers
  - If the set of subscribers is empty, notify parent that topic is empty now.
- On **StopTopic**:
  - Check again if the set of subscribers is empty.
  - If it is not empty, just send **SubscribeToTopic** to the parent from behalf of each subscriber. This will transparently pass all subscribers to the new topic actor.
  - Stop self.
- On **TweetObject**:
  - Broadcast the tweet to each subscriber.

StopTopic

Q: So what is the meaning of such complicated protocol for stopping topic actor?
A: When topic actor realizes that there are no subscribers left, it notifies the hub about it. Hub will forget about the actor. Every new subscription request will result in spawning a new actor. However, before hub received this notification it may already have sent subscription requests to the topic. When topic receives the StopTopic message it then hands these subscribers to the new topic actor through the hub.

7.3 Design Client-Server Communications

Client will:

- Subscribe to topic by sending message:

```
subscribe<SPACE>keyword
```

- Subscribe to topic by sending message:

```
unsubscribe<SPACE>keyword
```

Server will send tweets as message:

```
tweet<SPACE>body of the tweet
```

Treat **<SPACE>** as actual space character.

7.4 Build Topic Actor

Create **TwitterTopicActor** as **UntypedActor**. It should receive a keyword in the constructor.

- Start **TwitterClient** inside **preStart** method and save it to instance variable:

```
client = TwitterClient.start(self(), keyword);
```

- In **postStop** method you should stop the **TwitterClient**.

Implement **onReceive** method as described in design. When processing **StopTopic** use following trick:

```
if (!subscribers.isEmpty()) {
    for (ActorRef subscriber : subscribers) {
        context().parent().tell(
            new SubscribeToTopic(keyword),
            subscriber);
    }
}
context().stop(self());
```

It will send the message as if the subscriber has sent it.

7.5 Build the Topics Hub actor

Create **TwitterTopicsHubActor**. It should receive topic's actor class as the constructor parameter.

```
public class TwitterTopicsHubActor extends UntypedActor {
 private final Class<? extends UntypedActor> topicClass;

 public TwitterTopicsHubActor(
             Class<? extends UntypedActor> topicClass) {
       this.topicClass = topicClass;
 }
```

Implement **onReceive** method as described in design.

Use message forwarding when needed:

```
  topicActor.forward(message, context());
```

7.6 Build the Topics Session Processor

Create **TopicsSessionProcessor**. Process incoming messages like this:

```
String[] tokens = message.split(" ", 2);
if (tokens.length == 2) {
  switch (tokens[0]) {
      case "subscribe":
         //...
         break;
      case "unsubscribe":
         //...
         break;
   }
}
```

Process outgoing TweetObjects like this:

```
    session.getBasicRemote().sendText("tweet " + to.getText());
```

7.7 Wire It Up and Test

In **com.luxoft.akkalabs.day2.topics.web.Init**:

- Launch **TwitterTopicsHubActor** as "**topics**". Pass **TwitterTopicActor**'s class as constructor parameter.
- Lanuch **SessionsHubActor** as "**sessions**". Pass **TopicsSessionProcessor**'s class as constructor parameter.
- Launch **WebSocket** on "**/day2/topics**"

Build and deploy web application.

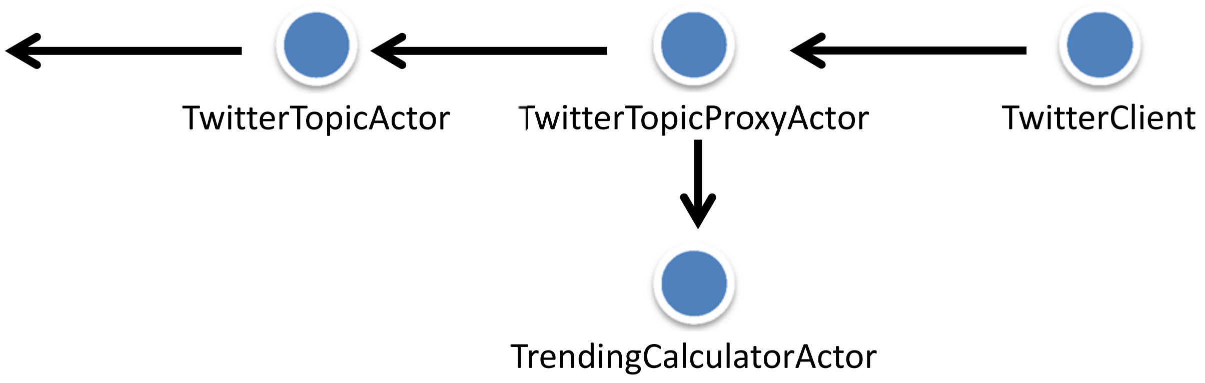
Open **day2/topics.html**. Enter a keyword and subscribe to topic. You should see tweets in the log.

Lab 8. Trending on Twitter Topics

In this lab, we are going to extend our app so it also calculates trending topics in twitter topic streams. Our goal is to do it reusing as much code from previous labs as possible.

Working package: **com.luxoft.akkalabs.day2.trending**

We are going to change the flow of data in this way:



A special proxy actor will be placed within the flow. It will send incoming messages to topics actor and to special actor that will calculate trending topics.

Trending calculator will collect statistics and once a few seconds will broadcast stats to every session.

There also will be an ‘upvote’ function on the client side. It will allow bumping topics up to the top.

8.1 Create the Topic Proxy Actor

Create a **TwitterTopicProxyActor** class.

To make our proxy actor work correctly we need to implement a few tricks.
First, we have to reimplement the subscription mechanics inside the proxy actor. Our proxy actor will process subscription messages by itself, but will pass all other messages freely in both ways.

Proxy actor should:

- Accept the keyword as constructor parameter.

- In **preStart**:
  - Spawn the real **TwitterTopicActor** as its child passing the keyword to its **Props**.
  - Watch over its life.

```
actor = context().actorOf(
          Props.create(TwitterTopicActor.class, keyword));

context().watch(actor);
```

By calling **watch** our actor subscribes for notifications about child actor’s death.

onReceive method is a bit complicated. It should cover all possible interaction cases:

- If we get message NOT from our twitter topic actor:
    - If it is **SubscribeToTopic**:
        - We should remember the sender as our subscriber.
        - If this is our first subscriber, then proxy actor subscribe itself to the twitter topic.
    - If the message is **UnsubscribeFromTopic**
        - We should remove the sender from the list of subscribers.
        - If this is the last subscriber, then proxy actor unsubscribes itself from the topic.
    - On any other type of message, just forward it to topic actor.

- If we get message from the twitter topic actor:
    - If it is a **TweetObject**, then broadcast it to our subscribers AND to selection for "**/user/trending**"
    - If it is NOT a **TweetObject**, just forward it to the parent.

- Receiving **Terminated** message indicates that actor's child has stopped. Proxy actor should stop too.

8.2 Create the Trending Actor

Create **TrendingCalculatorActor** class. In **preStart** create a heartbeat message subscription using the **Scheduler** service from ActorSystem. The following code will sign up this actor to receiving a PING message every second:

```
private final Object PING = new Object();

@Override
public void preStart() throws Exception {
    FiniteDuration oneSecond = FiniteDuration.
                                   create(1, TimeUnit.SECONDS);
    context().system().scheduler().schedule(
        oneSecond, oneSecond,
        self(), PING,
        context().dispatcher(), self());
}
```

Implement **onReceive**:

- On **TweetObject** extract words from the tweet, add one point for each word. Use java.util.Scanner to parse tweets for words. Remember to filter out all links and short words.
- On **UpvoteTrending** add five points to each word inside the object.
- On **PING** put current top words to **CurrentTrending** object, wrap it in **OutgoingBroadcast** object and send it to an **ActorSelection** for "**/user/sessions**".

UpvoteTrending
keyword: String

CurrentTrending
words: List<String>

8.3 Create the Trending Session Processor

Copy the code from **TopicsSessionProcessor** from previous lab.

Update the code for processing incoming messages so it accepts ‘**upvote**’ command:

```
upvote<SPACE>keyword(s)
```

Update outgoing processing so it reacts on CurrentTrending by sending this packet:

```
trend<SPACE>JSON representation of CurrentTrending
```

Get JSON representation of **CurrentTrending** by calling **toJSON** method.

8.4 Wire It Up and Test

In **com.luxoft.akkalabs.day2.trending.web.Init**:

- Launch **TrendingCalculatorActor** as “**trending**”.
- Launch **TwitterTopicsHubActor** as “**topics**”. Pass **TwitterTopicProxyActor**’s class as constructor parameter.
- Lanuch **SessionsHubActor** as “**sessions**”. Pass **TrendingSessionProcessor**’s class as constructor parameter.
- Launch **WebSocket** on “**/day2/ trending**”

Build and deploy web application.

Open **day2/trending.html**. Enter a keyword and subscribe to topic.

You should see tweets in the log. Trending topics should appear shortly.

Lab 9. Instagram pics from Twitter Topics

Now let us try something slightly different. Create an app that will collect Instagram links from tweets and display pictures on the web page. Of course in real time.

Working package: **com.luxoft.akkalabs.day2.instagram**

Infrastructure is the same as in previous apps.

- Clients connect via WebSocket, and subscribe to “**instagram**” twitter topic.
- When session receives tweets:
  - Instead of pushing tweets to client it extracts URLs and sends them to Instagram actor
  - Instagram actor asks the Instagram API for the URL of the picture behind that link and sends the URL back to the Session.
- Session receives the URL of the image and pushes it to the client.
- Client web page displays the picture.

9.1 Build the Instagram Actor

Create a regular actor and call it **InstagramActor**.

- Expect **String** objects with web page URLs.
- Use **`InstagramClient.pageToImageUrl(pageUrl)`** to convert web page URL to direct URL of the image.
- Send that URL back to the sender.

9.2 Build the Instagram Session Processor

- When session starts, subscribe to “**instagram**” topic:

```
context.system().actorSelection("/user/topics").
   tell( new SubscribeToTopic("instagram"), self);
```

- When session stops, unsubscribe from the topic:

```
topics.tell(new UnsubscribeFromTopic("instagram"), self);
```

- When receiving outgoing messages:
  - **TweetObject** – collect tweets and send them to the Instagram actor.

```
for (String url : tweetObject.getUrls()) {
   if ( url.startsWith("http://instagram.com/p/")
        && Math.random() < 0.1 ) {

         instagramActor.tell(url, context.self());

   }
}
```

  - **String** – send the string to the client:

    ```
    session.getBasicRemote().sendText(message);
    ```

9.3 Dealing with possible performance issues

To convert page URL to image URL **InstagramActor** uses **InstagramClient**. Each operation requires a remote call and may take a long time to complete. However, the inbound flow of Instagram tweets is huge, so it is highly possible that single Instagram Client will not be able to process everything in time and mailbox might get flooded. To address this issue we are going to add a load-balancing router in the next step.

9.4 Wire It Up and Test

In **com.luxoft.akkalabs.day2.instagram.web.Init**:

- Launch **TwitterTopicsHubActor** as "**topics**". Pass **TwitterTopicActor**'s class as constructor parameter.
- Lanuch **SessionsHubActor** as "**sessions**". Pass **InstagramProcessor**'s class as constructor parameter.
- Launch **InstagramProcessor** with Round-robin router to prevent overflow of the actor:

  ```
  system.actorOf(
     new RoundRobinPool(8).props(
        Props.create(InstagramActor.class)), "instagram");
  ```

- Launch **WebSocket** on "**/day2/instagram**"

Build and deploy web application.

Open **day2/instagram.html**. Images should appear shortly.