JavaScript EventSource and Java AsyncContext

A true push service on the web

Our goal:

- Build a publisher of events on a web server
- Build a client for events on the browser
- Allow for multiple simultaneous clients

In the following, only the most relevant parts of code are shown

Relevant, not well known classes used in the code

`JavaScript:`

`EventSource`

`java:`

The following ones should be well known, but we'll provide a short reminder

`Runnable, Thread`

Some Java classes usefuf when dealing with concurrency

`LinkedBlockingQueue`
`ConcurrentHashMap`
`CopyOnWriteArrayList`
`AtomicLong`
`UUID`

UUID: class that represents an immutable universally unique identifier (UUID).
A UUID represents a 128-bit value.

AtomicLong part of the Package **java.util.concurrent.atomic**
A small toolkit of classes that support lock-free thread-safe programming on single variables.

JavaScript EventSource

Interface to server-sent events.
Opens a persistent connection to an HTTP server, which
sends events in text/event-stream format.
The connection remains open until closed by calling EventSource.close().

incoming messages from the server are delivered in the form of **events**.

Unlike WebSockets, server-sent events are unidirectional: you can not use an EventSource channel to send message from browser to server.

When **not used over HTTP/2**, SSE suffers from a limitation to the maximum number of open connections: the limit is *per browser* and set to a 6.
When using **HTTP/2**, the maximum number of simultaneous *HTTP streams* is negotiated between the server and the client (defaults to 100).

see https://developer.mozilla.org/en-US/docs/Web/API/EventSource

JavaScript EventSource

Method onmessage: callback function activated on incoming messages

```
var evtSource = new EventSource('/mysource');
evtSource.onmessage = function(e) {
        document.getElementById('sse').innerHTML = e.data;
}
```

eventisteners:

```
sse = new EventSource('/api/v1/sse');
sse.addEventListener("notice", function(e) { console.log(e.data) })
sse.addEventListener("update", function(e) { console.log(e.data) })
sse.addEventListener("message", function(e) { console.log(e.data) })
```

This will listen only for events of type
event: notice
data: somedata
id: someid

This will listen for events of type
event: update

The event "message" will capture:

- events without an event field
- events of type `event: message`

see https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

Java Runnable and Thread

java.lang.Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread.
method run(): body of the Runnable, is never called explicitly, but is activated when the method start is called on a Thread that encapsulates the Runnable.

java.lang.Thread: java implementation of the thread concept. A thread a line of execution within a program. A Thread must either be instantiated by encapsulating a Runnable, or subclassed redefining the run method.

see
https://www.geeksforgeeks.org/runnable-interface-in-java/
https://www.geeksforgeeks.org/java-lang-thread-class-java/

image from https://www.geeksforgeeks.org

Java LinkedBlockingQueue

in packege **java.util.concurrent** : Utility classes commonly useful in concurrent programming.

Method **take() :** Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

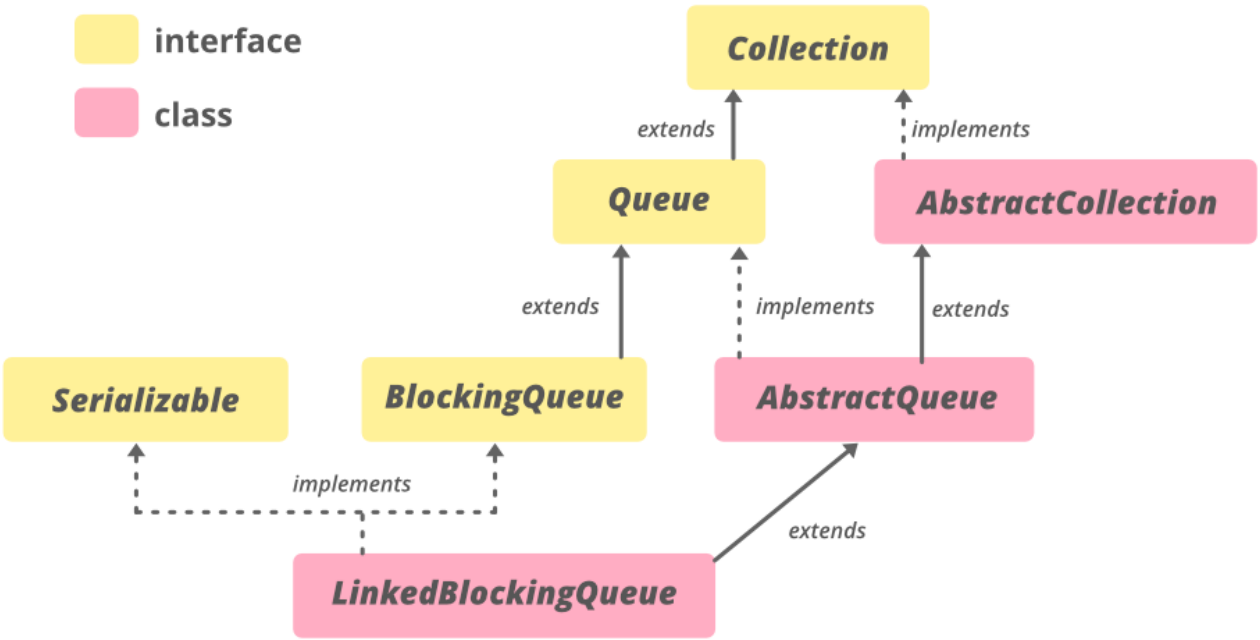


image from https://www.geeksforgeeks.org

Java ConcurrentHashMap

in packege **java.util.concurrent** : Utility classes commonly useful in concurrent programming.

A hash table supporting full concurrency of retrievals and high expected concurrency for updates.

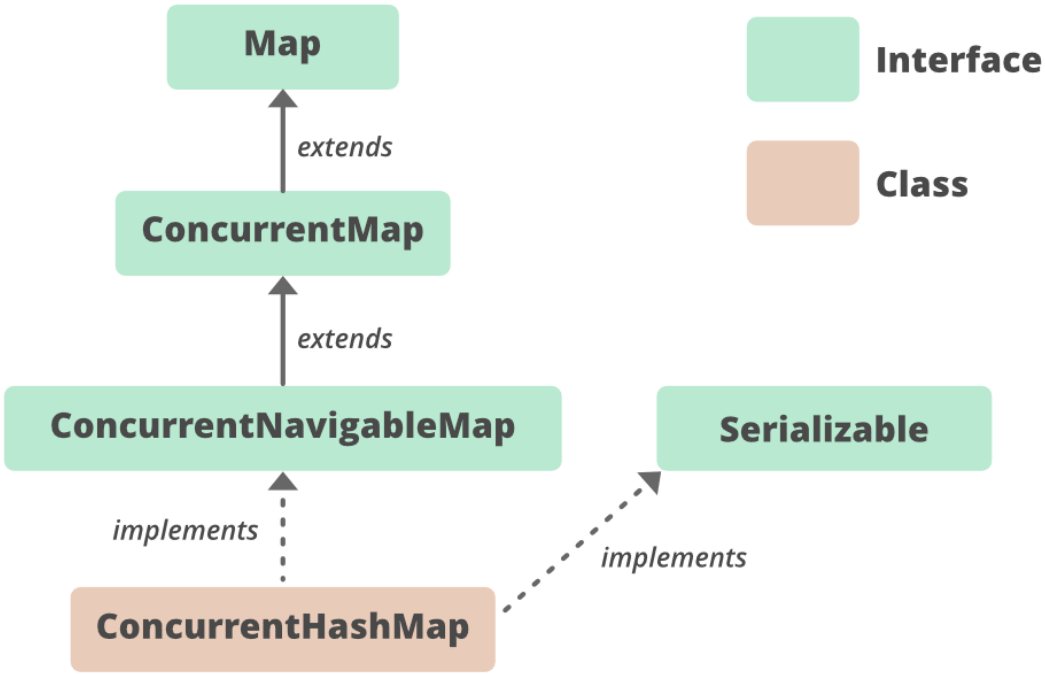


image from https://www.geeksforgeeks.org

Java CopyOnWriteArrayList

in package **java.util.concurrent** : Utility classes commonly useful in concurrent programming.

It is a thread-safe version of ArrayList. All modifications (add, set, remove, etc) are implemented by making a fresh copy, hence it is costly and is best used if our frequent operation is read operation.

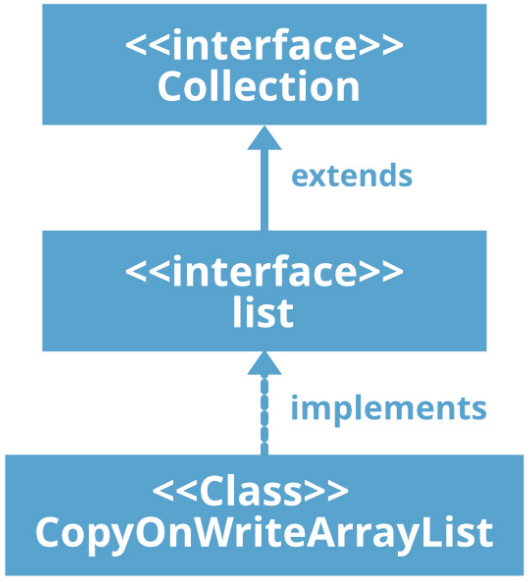


image from https://www.geeksforgeeks.org

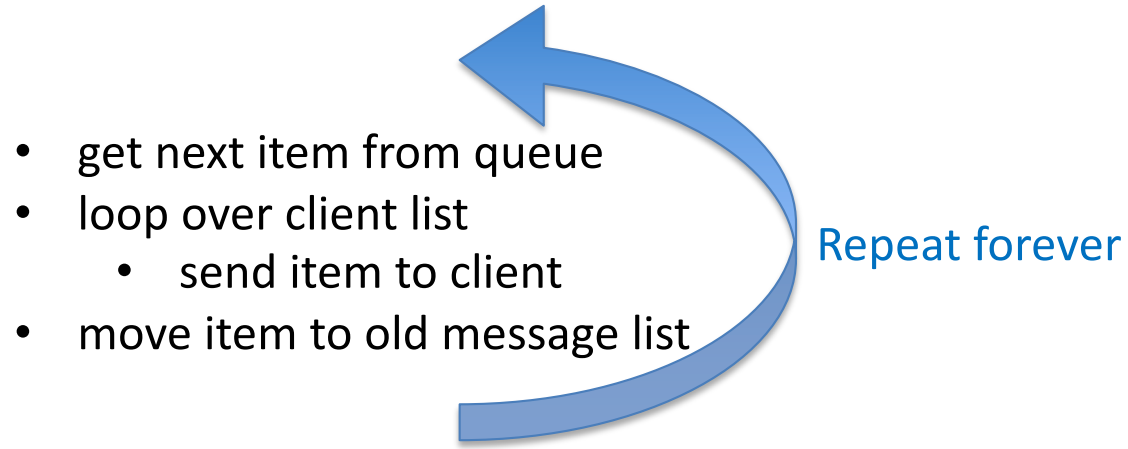
The data structure

- Created at WebApp start

NewsItem:
represents a message
- contains:
  - a serial number
  - a line of text
- knows how:
  - toString(): print itself into a String
  - toJSON() : print itself into a JSON structure

The agents: Distributor

- An autonomous thread
- **Mission**: distribute events to all clients



Distributor: the loop

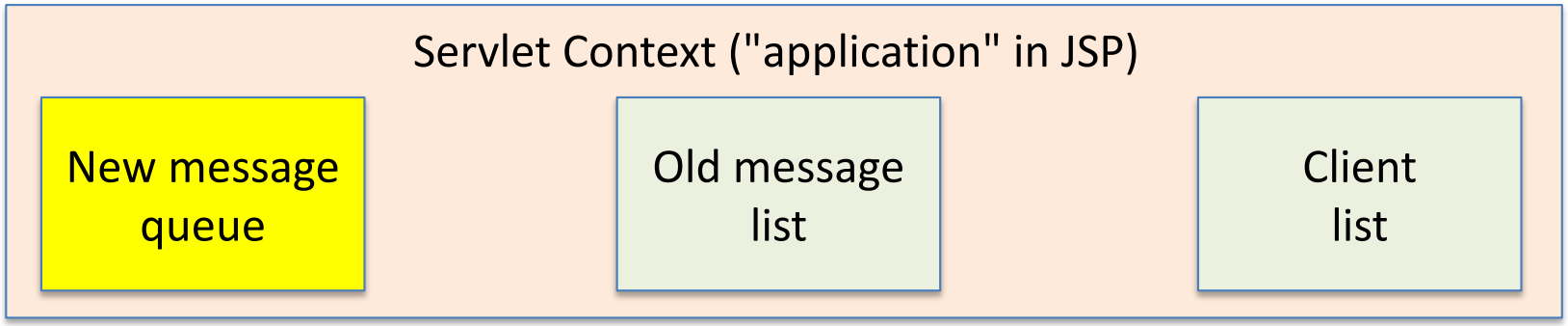
```
private void sendMessage(
   PrintWriter writer, NewsItem item) {
        writer.print("data: ");
        writer.println(item.toJSON());
        writer.println();
        writer.flush(); }
```

```
while (running) {
   try {
     // Waits until a news_item arrives
     NewsItem news_item = newItemsQueue.take();
     // Store into past items, for future clients
     pastItemsList.add(news_item);
     // Sends the item to all the clients
      Iterator<AsyncContext> iter=clientList.values().iterator();
     while (iter.hasNext()) {
       AsyncContext client=iter.next();
       try {
          PrintWriter channel = client.getResponse().getWriter();
          sendMessage(channel, news_item);
       } catch (Exception e) {
          // In case of problems remove context from map
          iter.remove();
       }
     }
   } catch (InterruptedException e) {/* Log exception, etc. */}
 }
```

The views: NewsCreator jsp

- Reads the use input
- sends it to its controller (NewsFeeder servlet)

The agents: NewsFeeder servlet



Mission:

- setup the web app
- start the distributor
- add messages to the New message queue

- At the beginning (init)
  - creates data structure in context
  - starts the Distributor

- When called (from NewsCreator.jsp):
  - adds message to New message queue

iNewsFeederServlet: the init()

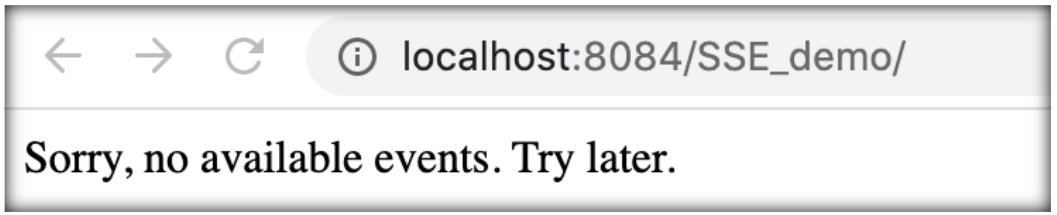
```
public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ctx=getServletContext();
        startEvent();
}
private void startEvent(){
  // create all needed items, add them to context, start the
                                                        distributor
  counter.set(0);
  newItemsQueue=new LinkedBlockingQueue<NewsItem>();
  ctx.setAttribute("queue", newItemsQueue);
  clientList=new ConcurrentHashMap<String, AsyncContext>();
  ctx.setAttribute("clients", clientList);
  pastItemsList=new CopyOnWriteArrayList<NewsItem>();
  ctx.setAttribute("newsList", pastItemsList);
  distributor=new Distributor(ctx);
  ctx.setAttribute("distributor", distributor);
  distributor.start();
}
```

NewsFeederServlet: save new item

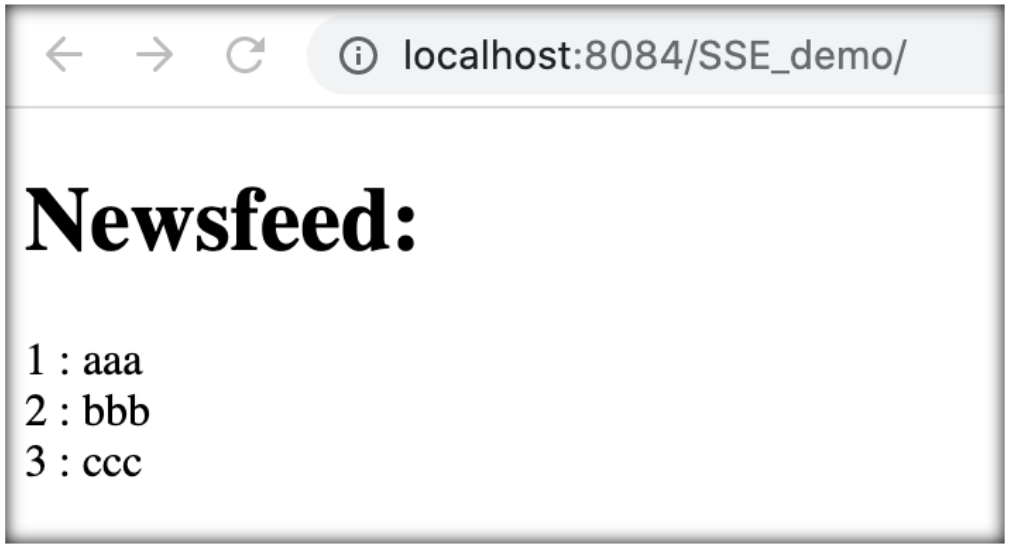
```
String newsLine = request.getParameter("line");
    if ((newsLine != null) && !newsLine.trim().isEmpty()) {
        if (newsLine.compareTo("%END%")==0){
            endEvent();
            request.getRequestDispatcher("/newsCreator.jsp")
                .forward(request, response);
            return;
        }
    try {
        NewsItem news_item = new NewsItem(
                counter.incrementAndGet(),newsLine.trim());
        newItemsQueue.put(news_item);
    } catch (InterruptedException e) { /*manage exception*/}
}
```

The views: index jsp

- Checks if data structure is in place. If not, tells the user that no event stream is there and ends.



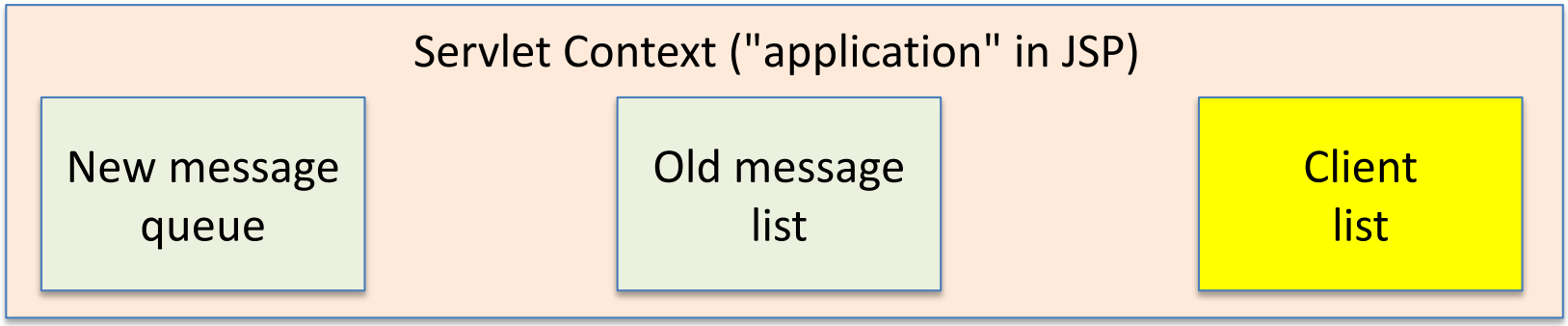
- Prints all old messages
- Starts waiting for new messages:
  - opens an EventSource to the NewsChannelOpener servlet
  - registers a callback for incoming events
  - the callback gets a JSON formatted message and prints it



index.jsp – the JS

```
<script>
    function test() {
        var source = new EventSource(
                        '/SSE_demo/NewsChannelOpenerServlet');
        source.onopen = function(event) {
                console.log("eventsource opened!");
        };
        source.onmessage = function(event) {
            var data = event.data;
            var obj = JSON.parse(data);
            console.log(data);
            document.getElementById('sse').innerHTML +=
                        obj.id+" : "+obj.text + "<br />";
        };
    }
    window.addEventListener("load", test);
</script>
```

The agents: NewsChannelOpener servlet



Mission:

- When new client comes
  - add it to the list
  - setup the response channel

- When called (from index.jsp):
  - initialize the response with the suitable headers
  - create an AsyncContext for the client, and pass to it request and response
  - add it to the Client list

NewsChannelOpenerServlet

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    if (request.getHeader("Accept").equals("text/event-stream")) {
        // setup resonse headers
        response.setContentType("text/event-stream");
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Connection", "keep-alive");
        response.setCharacterEncoding("UTF-8");
        // This a Tomcat specific - makes request asynchronous
        request.setAttribute("org.apache.catalina.ASYNC_SUPPORTED",
                             true);
        clientList = (Map<String, AsyncContext>)
                      ctx.getAttribute("clients");
         addReader(request,response);
     } else {
       response.getWriter().println("Sorry, event stream not
                                     supported");
     }
}
```

NewsChannelOpenerServlet

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    if (request.getHeader("Accept").equals("text/event-stream")) {
        // setup resonse headers
        response.setContentType("text/event-stream");
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Connection", "keep-alive");
        response.setCharacterEncoding("UTF-8");
        // This a Tomcat specific - makes request asynchronous
        request.setAttribute("org.apache.catalina.ASYNC_SUPPORTED",
                             true);
        clientList = (Map<String, AsyncContext>)
                      ctx.getAttribute("clients");
         addReader(request,response);
     } else {
       response.getWriter().println("Sorry, event stream not
                                     supported");
     }
}
```

Summary

1) **On the client side, we need to open an EventSource to read data, and to associate to it callbacks to manage data and events received.**

2) **On the server side, we need to create an AsyncContext for every client. When we want to send a message or an event to clients, we do that through its AsyncContext.**

Notifications API

Allows web pages to control the display of **system notifications** to the end user.

These are outside the top-level browsing context viewport, so therefore can be displayed even when the user has switched tabs or moved to a different app.

The API is designed to be compatible with existing notification systems, across different platforms.

https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API/Using_the_Notifications_API

Push API

Gives web applications the ability to receive messages pushed to them from a server, whether or not the web app is in the foreground, or even currently loaded, on a user agent.
This lets developers deliver asynchronous notifications and updates to users that opt in, resulting in better engagement with timely new content.

This is an experimental technology, in Firefox merged with Notifications

https://developer.mozilla.org/en-US/docs/Web/API/Push_API