JAKARTA EE

Get started with concurrency in Jakarta EE

One of the cornerstones of any well-written application is good performance, and that often means being able to run two or more tasks at the same time in parallel.

by Josh Juneau

June 22, 2020

When you are developing applications and services for the Jakarta EE platform, the Jakarta Concurrency API is an important tool. That API enables a number of strategies that can be used to build a seamless experience for your user base. Moreover, the API closely resembles Java SE's Concurrency API (`java.util.concurrent`), making it an easy transition between the two. In this article, I will touch on a number of the strategies you can use to develop robust, well-performing applications on the Jakarta EE platform.

First, let's quickly review threading, which is the core concept of concurrency. A thread encapsulates the execution of a process for an application. The JVM allows for multiple threads to be processed concurrently. Threads have priorities, allowing higher priority threads the ability to be executed prior to those designated with lower priority.

This type of threading works very well for development of Java SE applications but, unfortunately, uncontrolled spawning of multiple threads in a server environment is not an acceptable solution because a single thread could consume all the memory provided for the environment. Moreover, containers can run multiple applications at a time, and multiple running threads would increase the possibility for starvation of other processes within the container. The Jakarta EE server environment, provides a number of services that manage the execution of tasks, so tasks can be handed off to these services in a similar manner as spawning multiple threads in a Java SE environment.

By the way, all the code in this article is available on GitHub.

Getting started with concurrency

The full Jakarta EE platform contains support for the Jakarta Concurrency API, so it is easy to get started. Any application server or container that is fully Jakarta EE–compliant will contain the API; if you are using the Jakarta EE Web Profile or only portions of the Jakarta EE platform, it is easy enough to pull in the required dependencies by adding the following to the POM file of a Maven project:

```
<dependency>
    <groupId>jakarta.enterprise.concurrent</g
    <artifactId>jakarta.enterprise.concurrent
    <version>2.0.0-RC1</version>
</dependency>
```

The container plays an important role in the use of the Jakarta Concurrency API because it contains services that are used for running tasks in the background, just as you would spawn off long-running tasks to a background thread for a Java SE desktop solution. That being said, a compliant Jakarta EE container will have default services available for use with the API. However, it's possible to customize these services or create multiple services for use with different applications, if you wish to do so.

Note: If you are using Jakarta Concurrency 1.1, the package naming convention is `javax.enterprise.concurrent.*`. However, if you are using release 2.0.0 or later, the packages have been renamed to `jakarta.concurrent.*`. In this article, I will use Jakarta EE 8, which contains Jakarta Concurrency 1.1.

Configuring the container

Compliant containers include the following services for use with Jakarta concurrency:

- `ManagedExecutorService`: This service extends the ExecutorService, which is part of the Java SE Concurrency API. It is used for submitting tasks to the server for processing in an asynchronous manner. Once this service completes the task, it returns a `Future` object to the caller.
- `ManagedScheduledExecutorService`: This service extends the `ScheduledExecutorService`, which is part of the Java SE Concurrency API. It is used for submitting delayed or periodic tasks for processing. Once this service completes the task, it returns a `Future` object to the caller.
- `ManagedThreadFactory`: This service extends the ThreadFactory, which is part of the Java SE Concurrency API. Its main purpose is to produce managed threads for an application so tasks can be handed off to the server for

processing, functioning in a very similar manner as Java SE thread management.

- `ContextService`: This service provides methods for generating contextual dynamic proxy objects. It is not discussed in this article.

When tasks are submitted to each of these services, they run within the same application context as the caller. Therefore, the tasks have access to the same resources as the calling application, making it easy to manage state, share data, and so on.

Although there are preconfigured resources available within all compliant Jakarta EE containers, it is also possible to create custom resources. For instance, it might make sense to generate a custom `ManagedExecutorService` for each of the applications or services that require it. Most likely, in a microservices environment, only a single service will be deployed to a container. In this case, use of the default resources should be sufficient.

Because all compliant application server containers must provide a means to create custom managed resources, many of them provide multiple ways of doing so. Using the Payara server admin console as an example, it is possible to create one of these resources via the administrative console or the CLI. To create a `ManagedExecutorService` via the CLI, use the `asadmin` utility and issue the following command:

```
bin/asadmin create-managed-executor-service c
```

Use cases for Jakarta concurrency

There are a number of different use cases for concurrency, from creating a server-side task that can perform multiple actions asynchronously, to providing a seamless front-end experience for retrieving costly resources such as large datasets from a remote server.

The following sections outline some of the most common use cases along with examples of how to make use of the container resources. The examples for this article build a sports team roster application, which which uses a central RDBMS for data storage and delivers reports for displaying roster data. The user interface is developed using Jakarta Server Faces and stateless views, while the back-end logic for performing database calls resides within a series of services, each being a separate Jakarta EE application. It's important to note that the concurrency resources can be configured via XML or via the use of annotations. This article covers the use of annotations.

The ManagedExecutorService

One of the most commonly used services in the Jakarta Concurrency API is the `ManagedExecutorService`. There are two different interfaces a class can implement to use this service: `Runnable` and `Callable`. The `Runnable` interface can be used for sending a task to the `ManagedExecutorService` for processing, and it returns a `Future` that can be used to determine when the job has completed. The `Callable` implementation is used similarly, but it provides the opportunity to return a result.

In this example, a button within the application user interface will be used to invoke a report, which will spawn a background process to execute the report and then return back to the user once complete. In this scenario, I will generate a class that implements `Runnable`, and this class will encapsulate the logic that is used to invoke a web service to query the database and retrieve the results. The code in **Listing 1** is the `ReportRunnable` class, which implements `Runnable` and thereby contains an implementation of the `run()` method, which executes the long-running task.

Listing 1:

```
public class ReportRunnable implements Runnab

    private static Logger log = LogManager.ge
    private WebTarget resource;
    private String reportName;
    private List<Roster> rosterList;

    public ReportRunnable(String reportName)
        this.reportName = reportName;
    }

    /**
     * This method is overridden to execute a
     */
    @Override
    public void run() {
        if ("RosterReport".equals(reportName)
            invokeRosterReport();

        } else if ("DifferentReport".equals(r
            System.out.println("running diffe

        }
    }

    /**
     * Invokes web service to return roster l
     */
    protected void invokeRosterReport() {
        // Web Service Call
        resource = Utilities.obtainClient(Con

        setRosterList(resource.request(javax.
                .get(new GenericType<List<Ros
                }));
        rosterList.stream().forEach(r -> Syst
    }
```

```
    /**
     * @return the rosterList
     */
    public List<Roster> getRosterList() {
        return rosterList;
    }

    /**
     * @param rosterList the rosterList to se
     */
    public void setRosterList(List<Roster> ro
        this.rosterList = rosterList;
    }

}
```

Taking it from the user interface, the Roster Report view includes a button that is used to launch the report. The following excerpt from `rosterReport.xhtml` shows the code for the button that is used to invoke the report:

```
<p:commandButton id="rosterReport"
       actionListener="#{rosterController.inv
        value="Roster Report"/>
```

When the button is pressed, an `actionListener` initiates an action method named `invokeRosterReport()` within the controller class. The controller class is named `RosterController` (see **Listing 2**), and it is `ViewScoped`. This means that every time the view is visited, the context is rebased and the scope of the class is restarted. When the view is left, the scope is lost.

Listing 2:

```
@Named
@ViewScoped
public class RosterController implements java

    @Resource
    private ManagedExecutorService mes;

    @Resource
    private ManagedThreadFactory mtf;

    Thread rosterThread = null;

    private static Logger log = LogManager.ge
    private WebTarget resource;

    private List<Roster> rosterList;

    private Roster current;

    private boolean managePlayer = false;

    public RosterController() {

    }
```

```
@PostConstruct
public void init() {
    populateRosterList();
}

/**
 * Example of ManagedExecutorService. Thi
 * Refresh List button is pressed within
 * the call for the population of the ros
 * ManagedExecutorService for processing.
 *
 */
public void refreshRosterList() {

}

/**
 * Populate the List<Roster>.
 */
public void populateRosterList() {
    resource = Utilities.obtainClient(Con
    System.out.println(resource.getUri())
    setRosterList(resource.request(javax.
            .get(new GenericType<List<Ros
            }));
}

/**
 * Given an ID, return the corresponding
 *
 * @param id
 */
public void findById(int id) {
    resource = Utilities.obtainClient(Con
    resource = resource.path(java.text.Me
    current = resource.request(javax.ws.r
            // .cookie(HttpHeaders.AUTHOR
            .get(
                    new GenericType<Roste
            });
}

public String addPlayer() {
    String returnPage = null;
    resource = Utilities.obtainClient(Con
    Form form = new Form();
    form.param("firstName", current.getFi
    form.param("lastName", current.getLas
    form.param("position", current.getPos
    Invocation.Builder invocationBuilder
    //  .cookie(HttpHeaders.AUTHORIZATION
    Response response = invocationBuilder
    if (response.getStatus() == Status.CR
        log.info("Successful roster Entry
        Utilities.addSuccessMessage("Play
        rosterList = null;
        populateRosterList();
        returnPage = "index";
    } else {
        log.error("Player entry error");
        Utilities.addErrorMessage("Error
    }
    return returnPage;
}
```

```
public void remove(Roster player) {
    resource = Utilities.obtainClient(Con
    resource = resource.path(java.text.Me
    try {
        resource.request().delete();

        Utilities.addSuccessMessage("Remo
        rosterList = null;
        populateRosterList();
    } catch (Exception e) {
        Utilities.addErrorMessage("Error
    }
}

public void updatePlayer() {
    resource = Utilities.obtainClient(Con
    resource = resource.path(java.text.Me

    Response response
            = resource.request().put(Enti
    if (response.getStatus() == Status.OK
        Utilities.addSuccessMessage("Upda
        rosterList = null;
        managePlayer = false;
        populateRosterList();
    } else {
        Utilities.addErrorMessage("Error
    }
}

public void manage(Roster player) {
    current = player;
    managePlayer = true;
}

public void cancelAction() {
    managePlayer = false;
    current = null;
    rosterList = null;
    populateRosterList();
}

public void clear(AjaxBehaviorEvent event
    cancelAction();
}

public void invokeRosterReport() {
    ReportRunnable rosterReport = new Rep
    /*
         * Typically, the Future object s
         * polled periodically to retriev
     */
    Future reportFuture = mes.submit(rost
    while (!reportFuture.isDone()) {
        System.out.println("Running...");
    }
    if (reportFuture.isDone()) {
        System.out.println("Report Comple

    }
}

public void invokeThreaddedRosterReport()
    RosterRunnable rosterReport = new Ros
    /*
         * Typically, the Future object s
```

```
             * polled periodically to retriev
         */
        rosterThread =mtf.newThread(rosterRep
        rosterThread.start();
    }

    /**
     * @return the rosterList
     */
    public List<Roster> getRosterList() {
        return rosterList;
    }

    /**
     * @param rosterList the rosterList to se
     */
    public void setRosterList(List<Roster> ro
        this.rosterList = rosterList;
    }

    /**
     * @return the current
     */
    public Roster getCurrent() {
        if (current == null) {
            current = new Roster();
        }
        return current;
    }

    /**
     * @param current the current to set
     */
    public void setCurrent(Roster current) {
        this.current = current;
    }

    /**
     * @return the managePlayer
     */
    public boolean isManagePlayer() {
        return managePlayer;
    }

    /**
     * @param managePlayer the managePlayer t
     */
    public void setManagePlayer(boolean manag
        this.managePlayer = managePlayer;
    }

}
```

The `ManagedExecutorService` resource is injected into the `RosterController` via the `@Resource` annotation. Once injected, it can work with the Jakarta Concurrency API to invoke classes that implement the required interfaces. The `invokeRosterReport()` method is where the handoff occurs, because in this method, an instance of the `ReportRunnable` class is created, and it is then passed to the injected `ManagedExecutorService` resource.

The following excerpt shows the handoff to the service, as well as the returned `Future` object. The `Future` is then polled to determine when the spawned task has completed execution.

```
ReportRunnable rosterReport = new ReportRunna
Future reportFuture = mes.submit(rosterReport
while (!reportFuture.isDone()) {
    System.out.println("Running...");
}
if (reportFuture.isDone()) {
    System.out.println("Report Complete");
}
```

Transactions. One concern around spawning background tasks is the use of transactions so that if a portion of the task fails, other pieces that might have already completed can be rolled back to ensure data consistency. Use `jakarta.transaction.UserTransaction` to create and manage a transaction. To use this interface, a `UserTransaction` must be injected into a class via `@Resource`. The `UserTransaction.begin()` method can then be called to create a new transaction, and the `commit()` method can be used to complete the transaction. If needed, the `rollback()` method can be used to roll back. **Listing 3** shows a class that used transactions while performing RESTful service calls and awaiting responses.

Listing 3:

```
public class ReportRunnableTransaction implem

    private static Logger log = LogManager.ge
    private WebTarget resource;
    private String reportName;
    private List<Roster> rosterList;

    @Resource
    UserTransaction ut;

    public ReportRunnableTransaction(String r
        this.reportName = reportName;
    }

    /**
     * This method is overridden to execute a
     */
    @Override
    public void run() {
        if ("RosterReport".equals(reportName)
            try {
                ut.begin();
                invokeRosterReport();
                // perform some data transact
                ut.commit();
            } catch (NotSupportedException|Ro
SystemException|HeuristicMixedException|
HeuristicRollbackException|SecurityException|
IllegalStateException ex) {
                java.util.logging.Logger.getL
```

```
     ReportRunnableTransaction.class.getName(
    .log(Level.SEVERE, null, ex);
            }

        } else if ("DifferentReport".equals(r
            System.out.println("running diffe

        }
    }

    /**
     * Invokes web service to return roster l
     */
    protected void invokeRosterReport() {
        // Web Service Call
        resource = Utilities.obtainClient(
      Constants.ROSTER_URI, "roster").path("f

        setRosterList(resource.request(
     javax.ws.rs.core.MediaType.APPLICATION_X
                .get(new GenericType<List<Ros
                }));
        rosterList.stream().forEach(r -> Syst
    }

// getters and setters

}
```

Working with two or more asynchronous tasks. There are circumstances when more than one task might need to be asynchronously executed, and in these cases use the `ManageExecutorService.invokeAll()` method. To send more than one task for asynchronous execution, simply populate an `ArrayList` of executable tasks and pass the array to `invokeAll()`. By doing so, a list of `Future` objects is returned for the purposes of checking for task completion and obtaining results. Each of the task classes should adhere to a common format so the `Future`s can be handled in the same manner. This example creates a type class that is implemented by each of the task classes that will be involved in the asynchronous call, and the fields and methods defined within the type can later be used when processing the results.

To begin this example, create the type class that will contain the common fields and methods. The following code shows a type class named `RosterInfo`:

```
public class RosterInfo {
    public String team;
    public List<Roster> players = null;

    public RosterInfo(String team,
                List<Roster> players){
        this.team = team;
        this.players = players;
    }
}
```

Next, create task classes that will implement the type class. In this case, the task class will implement `Callable`, which enables you to return a `Future` result and also lets you use checked exceptions. The task class in this example is named `RosterTask`, and the complete code is in **Listing 4**. This class constructor accepts an `Integer` that will determine which team roster to query.

Listing 4:

```
public class RosterTask implements Callable<R
    // The ID of the request to report on dem
    Integer teamId;
    RosterInfo rosterInfo;
    private WebTarget resource;
    Map<String, String> execProps;

    public RosterTask(Integer id) {
        this.teamId = id;
        execProps = new HashMap<>();

        execProps.put(ManagedTask.IDENTITY_NA
    }

    public RosterInfo call() {
        // Web Service Call

        resource = Utilities.obtainClient(Con
        resource = resource.path(java.text.Me
        Team team = null;
        team = (resource.request(javax.ws.rs.
                .get(new GenericType<Team>()
                }));
        resource = Utilities.obtainClient(Con
        resource = resource.path(java.text.Me
        List<Roster> playerList = null;
        playerList = (resource.request(javax.
                .get(new GenericType<List<Ros
                }));

        return new RosterInfo(team.getName(),
    }

    public String getIdentityName() {
        return "RosterTask: TeamID=" + teamId
    }

    public Map<String, String> getExecutionPr
        return execProps;
    }

    public String getIdentityDescription(Loca
        // Use a resource bundle...
        return "RosterTask asynchronous REST
    }

    @Override
    public ManagedTaskListener getManagedTask
        return new CustomManagedTaskListener(
    }


}
```

Note that this class implements `Callable<RosterInfo>` as well as `ManagedTask`. Implementing `Callable<RosterInfo>` enforces class adherence to the definitions contained within the `RosterInfo` type. `ManagedTask` allows additional functionality, and it is not required. I will discuss this interface in the next section.

```
public class RosterTask
         implements Callable<RosterInfo>, Man
```

When you create a `Callable`, the `call()` method must be implemented, returning the type pertaining to the interface that contains the definitions needed for working with the results. The `call()` method then contains the implementation for the class and returns the result. In this example, `RosterService` is queried via `call()`, obtaining the roster for the team identified by the integer that was passed into the constructor of the task class. A new instance of `RosterInfo` is created and populated with the information pertaining to the roster, and then it is returned.

Listing 5:

```
@WebServlet(name = "BuilderServlet", urlPatte
public class BuilderServlet extends HttpServl
    // Retrieve the executor instance.

    @Resource(name = "concurrent/BuilderExecu
    ManagedExecutorService mes;
    RosterInfo rosterInfoHome;

    protected void processRequest(HttpServlet
            throws ServletException, IOExcept
        try {
            PrintWriter out = resp.getWriter(
            // Create the task instances
            ArrayList<Callable<RosterInfo>> b
            builderTasks.add(new RosterTask(1
            builderTasks.add(new RosterTask(2

            // Submit the tasks and wait.
            List<Future<RosterInfo>> taskResu
            ArrayList<RosterInfo> results = n
            for (Future<RosterInfo> result :
                out.write("Processing Results
                while (!result.isDone()) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedExcep
                        e.printStackTrace();
                    }
                }
                results.add(result.get());

            }
            out.write("** Results Processed S
            for (RosterInfo result : results)
                if (result != null) {
```

```
                    System.out.println("=====
                    System.out.println("Team:
                    System.out.println("=====
                    for(Roster roster:result.
                        System.out.println(ro
                                roster.getLas
                    }
                }
            }
        } catch (InterruptedException | Execu
            Logger.getLogger(BuilderServlet.c
        }
    }
// HttpServlet Methods
}
```

In this example, a servlet is used to invoke multiple instances of the `RosterTask`, passing a different integer into the constructor each time to return different team results. An `ArrayList` contains each of the task class instances, and the list is passed to `ManagedExecutorService invokeAll()` to execute the tasks. Refer to **Listing 5** to see the entire servlet.

```
ArrayList<Callable<RosterInfo>> builderTasks
    new ArrayList<Callable<RosterInfo>>();
builderTasks.add(new RosterTask(1));
builderTasks.add(new RosterTask(2));
// Submit the tasks and wait.
List<Future<RosterInfo>> taskResults =
   mes.invokeAll(builderTasks);
```

To obtain the results, allow the tasks time to complete and then add the results to a list. In the example, use a `while` loop along with `Thread.sleep()` to perform the check against `Future.isDone()`.

```
List<Future<RosterInfo>> taskResults
     = mes.invokeAll(builderTasks);
ArrayList<RosterInfo> results
     = new ArrayList<RosterInfo>();
for (Future<RosterInfo> result :
         taskResults) {
    while (!result.isDone()) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    results.add(result.get());
}
```

Finally, the list of results is iterated and processed.

```
for (RosterInfo result : results) {
    if (result != null) {
        . . .
```

```
        for(Roster roster:result.players){
            System.out.println(
                 roster.getFirstName()
                   . . . );
        }
    }
}
```

Implementing ManagedTask. The `ManagedTask` interface enables you to obtain identifying information about a task, provide a `ManagedTaskListener` to obtain lifecycle information, or provide additional execution properties. For example, the `RosterTask` class implements `ManagedTask`, which allows the overriding of `getManagedTaskListener()`. This method can return a custom task listener, as in the following code:

```
@Override
    public ManagedTaskListener
      getManagedTaskListener() {
      return
     new CustomManagedTaskListener();
    }
```

The `CustomManagedTaskListener` class (**Listing 6**) implements `ManagedTaskListener`, which enables the overriding of methods such as `taskSubmitted()`, `taskAborted()`, and `taskDone()`. Therefore, when these lifecycle actions occur, custom processing can be invoked.

Listing 6:

```
public class CustomManagedTaskListener implem

    @Override
    public void taskSubmitted(Future<?> futur
        System.out.println("Task Submitted");
    }

    @Override
    public void taskAborted(Future<?> future,
        System.out.println("Task Aborted");
    }

    @Override
    public void taskDone(Future<?> future, Ma
        System.out.println("Task Complete");
    }

    @Override
    public void taskStarting(Future<?> future
        System.out.println("Task Starting");
    }

}
```

The ManagedScheduledExecutorService

Many times, it makes sense for a system to run a lengthy task, such as running a report, on a timed schedule, possibly during nonbusiness hours. The `ManagedScheduledExecutorService` allows such a scheduled task to be executed in a very similar manner to when `ManagedExecutorService` is used.

In this example, the Roster application is going to schedule polling the roster for a number of different teams at scheduled intervals throughout the day. The same style of report class is used as in the previous examples, because the class implements `Runnable` and, therefore, overrides a `run()` method to initiate the report logic. Code for the class named `ReportRunnable` is contained within **Listing 7**. The constructor accepts a string-based report name and then executes the resulting report by calling on a service and obtaining the results.

Listing 7:

```
public class RosterRunnable implements Runnab

    private static Logger log = LogManager.ge
    private WebTarget resource;
    private List<Roster> rosterList;

    public RosterRunnable() {
    }

    /**
     * This method is overridden to execute a
     */
    @Override
    public void run() {
       obtainRoster();
    }

    /**
     * Invokes web service to return roster l
     */
    protected void obtainRoster() {
        // Web Service Call
        resource = Utilities.obtainClient(Con

        setRosterList(resource.request(javax.
                .get(new GenericType<List<Ros
                }));
        rosterList.stream().forEach(r -> Syst
    }

    /**
     * @return the rosterList
     */
    public List<Roster> getRosterList() {
        return rosterList;
    }

    /**
```

```
     * @param rosterList the rosterList to se
     */
    public void setRosterList(List<Roster> ro
        this.rosterList = rosterList;
    }

}
```

The important piece of this solution is the code that calls on `ReportRunnable`, since it is invoked via a `ManagedScheduledExecutorService`. In this example, the `ScheduledRosterReportRunner` class is responsible for invoking the report on a scheduled basis. This class is started up once for the application at deployment time and upon startup, and it schedules the report via the code below. See **Listing 8** for the complete code.

```
@PostConstruct
public void rosterScheduler() {
    System.out.println("Scheduling report..."
    ReportRunnable rosterReport
        = new ReportRunnable("RosterReport");
    rosterHandle = mes.scheduleAtFixedRate(
        rosterReport, 5L, 5L, TimeUnit.MINUTE
    System.out.println("Report scheduled...."
}
```

As seen in the example, the `scheduleAtFixedRate()` method is the key to scheduling a concurrent task for execution. This method accepts a `Runnable` as the first parameter, followed by the initial delay (a Long), the period (a Long), and `TimeUnit`. Alternatively, the code could use `scheduleWithFixedDelay()` to schedule along with a given delay between the termination of one execution and commencement of the next.

Listing 8:

```
@Startup
@Singleton
@ApplicationScoped
public class ScheduledRosterReportRunner {

    Future rosterHandle = null;

    @Resource(name = "concurrent/_defaultMana
    ManagedScheduledExecutorService mes;

    public ScheduledRosterReportRunner(){

    }

    @PostConstruct
    public void rosterScheduler() {
        System.out.println("Scheduling report
        ReportRunnable rosterReport = new Rep

        rosterHandle = mes.scheduleAtFixedRat
```

```
        System.out.println("Report scheduled.
    }
}
```

The ManagedThreadFactory

The `ManagedThreadFactory` was introduced to create threads within a Jakarta EE environment. To implement a bare-bones thread within an enterprise application, inject a `ManagedThreadFactory` into a class and then call on the `newThread()` method to create a new thread. The `newThread()` method is inherited from `java.util.concurrent.ThreadFactory`, so the implementation is very familiar for developers who are working with threads in a Java SE environment:

```
. . .
@Resource
    private ManagedThreadFactory mtf;
. . .
public void invokeThreaddedRosterReport(){
        RosterRunnable rosterReport
          = new RosterRunnable();
        rosterThread = mtf.newThread(rosterRe
        rosterThread.start();
  }
```

Conclusion

The Jakarta Concurrency API provides many options for generating concurrent solutions within enterprise applications. It provides the ability to produce threads, managed tasks, and scheduled tasks. To use the API, load the full Jakarta EE 8 profile or include the necessary dependencies for the API in the project. Jakarta EE 8 contains Jakarta Concurrency 1.1, and Jakarta EE 9 will contain Jakarta Concurrency 2.0.

You can learn more here:

- Jakarta EE
- Jakarta Concurrency API
- The source code for this article is available on GitHub.

Josh Juneau

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to

Oracle Technology Network and *Java Magazine* and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the *JavaPubHouse Off Heap* podcast.

Share this Page