Streamlining Jakarta EE 11 Development with Payara and start.jakarta.ee: A Hands-On Guide

Jakarta EE 11, the latest evolution of enterprise Java, brings a wealth of new features and improvements for building robust, scalable applications. Combined with Payara Server, a developer-friendly Jakarta EE runtime, and start.jakarta.ee, a handy project generator, the development process becomes smoother than ever. In this guide, we'll walk you through building a sample Jakarta EE 11 application on Payara, highlighting some of the features along the way.

Setting Up Your Development Environment

Before we dive into coding, let's ensure your environment is ready:

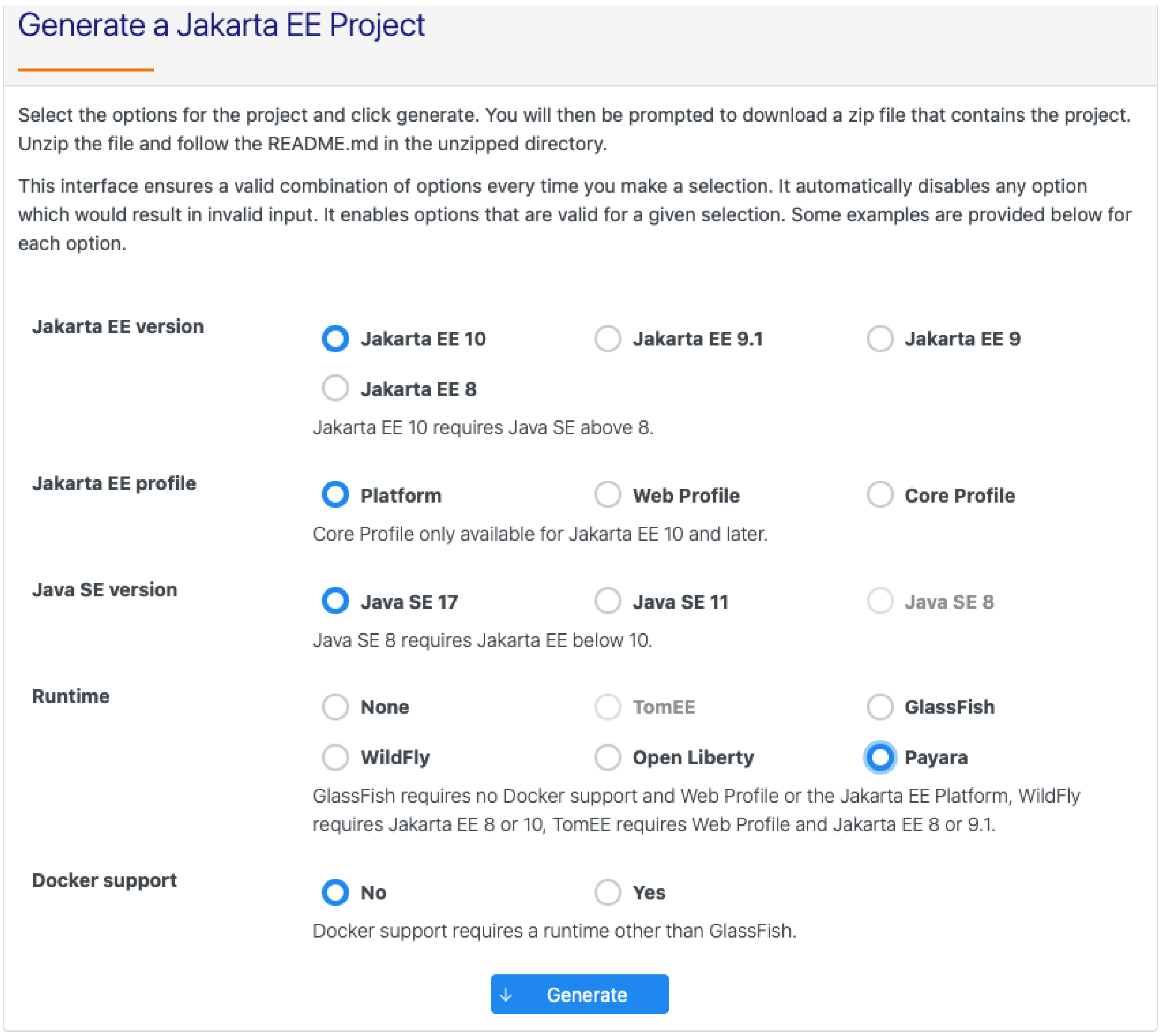
1. **JDK Installation:** Make sure you have JDK 21 installed. You can download it from the official Oracle website or use a package manager like SDKMAN!
2. **Maven:** We will use Maven for dependency management, so install it using SDKMAN or the Maven website.
3. **IDE:** Install your preferred IDE. We'll not use anything IDE-specific here, but I use IntelliJ IDEA, a popular choice among Java developers.

Once these tools are ready, we'll create the project structure from **`start.jakarta.ee`**.

Embracing start.jakarta.ee

`start.jakarta.ee` is your gateway to a streamlined project setup. Here's how to use it:

1. Navigate to https://start.jakarta.ee/. This service will set up all the essential dependencies for an application. The current version of the Starter only supports Maven. In the future, we may be able to choose between Gradle and Maven.
2. Select the desired version of Jakarta EE from the available options (Figure 1). Currently, the options include Jakarta EE 8, Jakarta EE 9.1, and Jakarta EE 10. In addition, you may choose the Jakarta EE Platform or one of the Jakarta EE profiles (Web, Core). For this project, we have chosen the Jakarta EE 10 Platform, Java SE 17, and Payara as a Runtime. Although we intend to use Jakarta EE 11, this starter does not yet support 11, but no worries; as we proceed, we will show you how to upgrade to Jakarta EE 11 and JDK 21.
3. Once you have selected your desired options, click the generate button. This will give you the project structure and sample code you can build and run.



Generate a Jakarta EE Project

Select the options for the project and click generate. You will then be prompted to download a zip file that contains the project. Unzip the file and follow the README.md in the unzipped directory.

This interface ensures a valid combination of options every time you make a selection. It automatically disables any option which would result in invalid input. It enables options that are valid for a given selection. Some examples are provided below for each option.

Jakarta EE version: [✓ Jakarta EE 10] Jakarta EE 9.1 Jakarta EE 9 Jakarta EE 8

Jakarta EE 10 requires Java SE above 8.

Jakarta EE profile: [✓ Platform] Web Profile Core Profile

Core Profile only available for Jakarta EE 10 and later.

Java SE version: [✓ Java SE 17] Java SE 11 Java SE 8

Java SE 8 requires Jakarta EE below 10.

Runtime: None TomEE GlassFish WildFly Open Liberty [✓ Payara]

GlassFish requires no Docker support and Web Profile or the Jakarta EE Platform, WildFly requires Jakarta EE 8 or 10, TomEE requires Web Profile and Jakarta EE 8 or 9.1.

Docker support: [✓ No] Yes

Docker support requires a runtime other than GlassFish.

Figure 1: Jakarta EE starter project generation tool

Let's explore the code structure.

When we unpack the generated code, we will have the structure of an application. We can open it in our favourite IDE and run it from the command line.

```
.
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── eclipse
        │           └── jakarta
        │               └── hello
        │                   ├── Hello.java
        │                   ├── HelloApplication.java
        │                   └── HelloWorldResource.java
        └── webapp
            ├── WEB-INF
            │   └── web.xml
            ├── images
            │   └── jakartaee_logo.jpg
            └── index.html
```

This generated source code comes with an embedded Maven wrapper. So, if you want to use it, make sure you run the following command first for Unix environments (Linux or Mac):

```
$ chmod +x mvnw
```

Since we are using Payara as a runtime, the following command will download the runtime and run the application.

```
./mvnw clean package cargo:run
```

We are using the cargo plugin, which allows you to manipulate various types of Jakarta EE runtimes in a standard way. It will download the Payara runtime and run the Payara application server along with our application so that we don't have to maintain a separate server instance while developing. This helps us speed up the development process.

If you hit the browser with the following URL, you will see the result.

```
http://localhost:8080/jakartaee-hello-world/
```

This starter application comes with a basic setup of the following Rest Endpoint.

```
curl -X GET
"http://localhost:8080/jakartaee-hello-world/rest/hello?name=Balzur"
```

If we curl it, it will return the following output:

```
{
  "hello": "Balzur"
}
```

Upgrading to Jakarta EE 11 and JDK 21

Now that the basic setup is ready, we will make progress on upgrading Jakarta EE 11 and JDK 21. We will probably not be able to touch every feature, but we will try to use the most important one, which is the ability to use virtual threads and a few others.

Let's begin upgrading the project to Jakarta EE 11 and JDK 21.

For that, open the **`pom.xml`** and change the following properties.

```
<maven.compiler.release>21</maven.compiler.release>
<jakartaee-api.version>11.0.0-M3</jakartaee-api.version>
<payara.version>7.2024.1.Alpha1</payara.version>
```

That's all we need. We are using 7.2024.1.Alpha1, as this supports Jakarta Concurrency 3.1, which supports virtual threads.

Run it like we did earlier and see if everything is running perfectly.

Project Idea

For this project, we need a simple app idea. The one I have come up with is as follows: The app will receive a website's URL through an endpoint, fetch the metadata from the website, and store it in the database. The idea is as simple as that.

This functionality could be valuable for various real-world applications. For instance, consider a marketing firm that needs to analyze the metadata of competitor websites to gain insights into their SEO strategies, content keywords, and social media engagement. This application could automate data-gathering, saving the firm valuable time and resources. The metadata could then generate reports, inform marketing strategies, and improve the firm's online presence.

We will implement a scheduler to automate the process of fetching metadata from the URLs stored in the database. The scheduler will periodically check for new tasks (URLs) in the database, fetch the metadata from these URLs, and update the task status accordingly. This will ensure our application continuously processes new URLs and keeps the metadata up-to-date without manual intervention.

Creating an Endpoint to Receive the URL

The next step is to create an endpoint to receive the URL. Let's call it **`TaskResource`**.

```java
package org.eclipse.jakarta.hello;

import jakarta.validation.Valid;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

import java.util.logging.Logger;

@Path("/tasks")
public class TaskResource {
   private final Logger logger =
Logger.getLogger(TaskResource.class.getName());

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   @Produces(MediaType.APPLICATION_JSON)
   public Response createTask(@Valid TaskDTO taskDTO) {
       logger.info("TaskResource.createTask() called with taskDTO: " +
taskDTO);

       return Response.accepted().build();
   }
}
```

The `jakarta.ws.rs.Path` annotation establishes a connection between the URL given by the user and the Java class responsible for handling the request. The `jakarta.ws.rs.GET` annotation tells us we must use the `HTTP GET` method to access our endpoint. The `jakarta.ws.rs.Produces` annotation and allows you to specify the format of the response. In our case, it will produce a JSON[1] response, but the above endpoint will not return anything but will return the 201 status code.

[1] JSON - It stands for JavaScript Object Notation. JSON is a text format for storing and transporting data

```
curl -X POST -H "Content-Type: application/json" -d '{"url":
"https://bazlur.ca"}'
http://localhost:8080/jakartaee-hello-world/rest/tasks -i
```

The output of the above curl command would be:

```
HTTP/1.1 201 Created
Server: Payara Server 7.2024.11 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 7.2024.11 #badassfish
Java/Oracle Corporation/21)
Location: http://localhost:8080/tasks/3
Content-Length: 0
X-Frame-Options: SAMEORIGIN
```

At this point, this endpoint doesn't do anything but only takes a task and logs it. As you can see, we have used the `@Valid` annotation before **`TaskDTO`**, which means this will automatically perform validation, which is part of the Jakarta Validation specification.

Defining TaskDTO

Let's look at the **`TaskDTO`**:

```
public record TaskDTO(
       @NotEmpty(message = "The URL field is mandatory and must not be
empty.")
       @Pattern(regexp = "^(http|https)://.*$", message = "The input
provided is not a valid URL. Please provide a valid URL.")
       String url
) {}
```

As you can see, we've utilized Java records, which simplify the creation and use of Data Transfer Objects (DTOs). A key advantage of records is their seamless integration with Jakarta Validation annotations, allowing for convenient and efficient data validation. We have used two constraints: `NotEmpty` and `Pattern`. So, if we pass a valid URL, the endpoint will return an HTTP 400 status code with a default error response.

However, to make the error message a bit more friendly, we have to add the following Exception Mapper class, which the `@Provider` annotation will automatically register.

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.ConstraintViolationException;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.ExceptionMapper;
import jakarta.ws.rs.ext.Provider;

import java.util.HashMap;
import java.util.Map;

@Provider
public class ConstraintViolationExceptionMapper implements
ExceptionMapper<ConstraintViolationException> {
   @Override
   public Response toResponse(ConstraintViolationException exception) {
       Map<String, String> response = new HashMap<>();
       for (ConstraintViolation<?> violation :
exception.getConstraintViolations()) {
           response.put(violation.getPropertyPath().toString(),
violation.getMessage());
       }
       return
Response.status(Response.Status.BAD_REQUEST).entity(response).build();
   }
}
```

Now, if we post an invalid URL, it will return the validation error as follows:

```
curl -X POST -H "Content-Type: application/json" -d '{"url": "some-url"}'
http://localhost:8080/jakartaee-hello-world/rest/tasks -i

HTTP/1.1 400 Bad Request
Server: Payara Server 7.2024.11 #badassfish
X-Powered-By: Servlet/6.0 JSP/3.1 (Payara Server 7.2024.11 #badassfish
Java/Oracle Corporation/21)
Content-Type: application/json
Connection: close
Content-Length: 92
X-Frame-Options: SAMEORIGIN

{"createTask.arg0.url":"The input provided is not a valid URL. Please
provide a valid URL."}%
```

Storing Tasks in the Database

Now that this endpoint is ready, let's save the task in the database. First, let's create an Entity.

```java
import jakarta.persistence.*;
import java.io.Serializable;
import java.time.Instant;
import java.util.Map;

@Entity
public class Task implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long taskId;

    @Version
    private Long version;
    private String inputUrl;

    @Enumerated(EnumType.STRING)
    private Status status;
    private Instant createdAt;
    private Instant updatedAt;
    private Instant completedAt;

    @ElementCollection
    @MapKeyColumn(name = "meta_key")
    @Column(name = "meta_value")
    @CollectionTable(name = "meta_data", joinColumns = @JoinColumn(name =
"task_id"))
    private Map<String, String> metaData;

    @PrePersist
    public void onPrePersist() {
        this.createdAt = Instant.now();
        this.status = Status.NEW;
    }

    @PreUpdate
    public void onPreUpdate() {
        this.updatedAt = Instant.now();
    }
    // getter/setter
}
```

The **`Task`** entity now uses [**`Instant`**]() to store datetime values, ensuring consistency across different locations by storing event timestamps in UTC. The support of Instant is the new addition Jakarta EE 11.

The **`Status`** enum:

```
public enum Status {
    NEW,
    IN_PROGRESS,
    COMPLETED,
    FAILED
}
```

Configuring Data Source

We are using H2, an in-memory database, for this example, but feel free to use any database.

Open your **`web.xml`** file and add the following configuration:

```
<data-source>
    <name>java:global/myDataSource</name>
    <class-name>org.h2.jdbcx.JdbcDataSource</class-name>
    <url>jdbc:h2:mem:test</url>
</data-source>
```

Configuring Jakarta Persistence

The next step is configuring the database connection by creating a Jakarta Persistence configuration file named **`persistence.xml`**. This file can be placed under the **`resources/META-INF`** folder in our project. If you don't have the resource folder in your project structure, just create one.

The **`persistence.xml`** file allows us to specify the JDBC Connection Settings or the Datasource JNDI name.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
xmlns="[https://jakarta.ee/xml/ns/persistence](https://jakarta.ee/xml/ns/pe
rsistence)"
xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance](http://www.w3.org/20
01/XMLSchema-instance)"

xsi:schemaLocation="[https://jakarta.ee/xml/ns/persistence](https://jakarta
.ee/xml/ns/persistence)
[https://jakarta.ee/xml/ns/persistence/persistence_3_2.xsd](https://jakarta
.ee/xml/ns/persistence/persistence_3_2.xsd)"
            version="3.2">

   <persistence-unit name="tasks">
       <jta-data-source>java:global/myDataSource</jta-data-source>
       <properties>
           <property
name="jakarta.persistence.schema-generation.database.action"
value="drop-and-create"/>
           <property name="eclipselink.logging.level.sql" value="FINE"/>
           <property name="eclipselink.logging.parameters" value="true"/>
           <property name="hibernate.show_sql" value="true"/>
           <property name="eclipselink.ddl-generation"
value="create-tables"/>
           <property name="eclipselink.ddl-generation.output-mode"
value="database"/>
       </properties>
   </persistence-unit>
</persistence>
```

The **`persistence.xml`** file sets up a persistence unit named "**`tasks`**". It specifies the JNDI name of the JDBC datasource to be used for database connection management, which is "**`java:global/myDataSource`**" as already specified in **`web.xml`**.

The **`<properties>`** element contains several properties that configure the behavior of the Jakarta Persistence provider. For example, the "`jakarta.persistence.schema-generation.database.action`" property in the `persistence.xml` file specifies the action to be taken by the Jakarta Persistence provider when generating the database schema. Some options are:

- `none`: The Jakarta Persistence provider won't generate the database schema.
- `create`: The Jakarta Persistence provider will create the database schema.
- `drop`: The Jakarta Persistence provider will drop the database schema.
- `drop-and-create`: The Jakarta Persistence provider will drop the existing database schema and create a new one.

Other properties configure logging for the Jakarta Persistence provider, such as `"eclipselink.logging.level.sql"` and `"eclipselink.logging.parameters"`. Finally, the `"hibernate.show_sql"` property enables SQL query logging for the Hibernate Jakarta Persistence provider.

Setting Up the Jakarta Persistence Repository

Next, create a **`TaskRepository`** class responsible for handling the Task entity's Create, Read, Update, and Delete (CRUD) operations:

```
import jakarta.transaction.Transactional;
import java.util.List;
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@Stateless
public class TaskRepository {

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public Task save(Task task) {
        return em.merge(task);
    }

    public List<Task> findAll() {
        return em.createQuery("SELECT t FROM Task t",
Task.class).getResultList();
    }

    public List<Task> findByStatus(Status status) {
        return em.createQuery("SELECT t FROM Task t WHERE t.status =
:status", Task.class)
                 .setParameter("status", status)
                 .getResultList();
    }
}
```

The class is annotated with `@Stateless`, which makes it a stateless session bean. A Stateless session bean is a type of Enterprise JavaBean (EJB) that is used for implementing business logic in Jakarta EE applications. Stateless session beans are designed for scenarios where the bean does not need to maintain any conversational state with the client between method invocations. In other words, a Stateless session bean doesn't remember any client-specific data between method calls.

The `EntityManager` is the primary interface for managing entities in Jakarta Persistence. It is annotated with `@PersistenceContext`, which automatically injects an instance of the **`EntityManager`** into the class.

Now, let's update our **`TaskResource`** to be able to use this repository.

Updating TaskResource

Now, update **`TaskResource`** to use this repository:

```
package org.eclipse.jakarta.hello;

import jakarta.inject.Inject;
import jakarta.validation.Valid;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

import java.net.URI;
import java.util.List;
import java.util.logging.Logger;

@Path("/tasks")
public class TaskResource {
   private final Logger logger =
Logger.getLogger(TaskResource.class.getName());

   private final TaskRepository taskRepository;

   @Inject
   public TaskResource(TaskRepository taskRepository) {
       this.taskRepository = taskRepository;
   }

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   @Produces(MediaType.APPLICATION_JSON)
   public Response createTask(@Valid TaskDTO taskDTO) {
       logger.info("TaskResource.createTask() called with taskDTO: " +
taskDTO);

       Task task = new Task();
       task.setInputUrl(taskDTO.getUrl());
       Task savedTask = taskRepository.save(task);

       return Response.created(URI.create("/tasks/" +
savedTask.getTaskId())).build();
   }

   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public List<Task> getAllTasks(@QueryParam(value = "status") Status
status) {
       if (status == null) {
           return taskRepository.findAll();
       }

       return taskRepository.findByStatus(status);
   }
}
```

Creating a Scheduler

Now that we have a record, we will create a scheduler to periodically pull the tasks and execute them.

```
package org.eclipse.jakarta.hello;

import jakarta.annotation.Resource;
import jakarta.ejb.Lock;
import jakarta.ejb.LockType;
```

```java
import jakarta.ejb.Schedule;
import jakarta.ejb.Singleton;
import jakarta.enterprise.concurrent.ManagedExecutorDefinition;
import jakarta.enterprise.concurrent.ManagedExecutorService;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.io.IOException;
import java.time.Instant;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

@Singleton
@ManagedExecutorDefinition(
    name = "java:app/concurrent/taskScheduler",
    hungTaskThreshold = 120000,
    virtual = true
)
public class TaskScheduler {

    private static final Logger logger =
Logger.getLogger(TaskScheduler.class.getName());

    @Inject
    private TaskRepository taskRepository;

    @Resource(lookup = "java:app/concurrent/taskScheduler")
    private ManagedExecutorService taskScheduler;

    @Schedule(hour = "*", minute = "*", second = "*/10", persistent =
false)
    public void pullTask() {
        List<Task> newTasks = taskRepository.findByStatus(Status.NEW);
        newTasks.forEach(this::execute);
    }
    @Transactional
    private void execute(Task task) {
        task.setStatus(Status.IN_PROGRESS);
        taskRepository.save(task);
        taskScheduler.execute(() -> processTask(task));
    }

    private void processTask(Task task) {
        try {
            String inputUrl = task.getInputUrl();
            logger.info(String.format("Fetching content from URL: %s
executing on thread: %s",
                inputUrl, Thread.currentThread()));

            task.setMetaData(extractAllMetadata(inputUrl));
            task.setStatus(Status.COMPLETED);
            task.setCompletedAt(Instant.now());
            taskRepository.save(task);
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Failed to fetch content from URL: " +
task.getInputUrl(), e);
            updateTaskStatusToFailed(task);
        }
    }

    private Map<String, String> extractAllMetadata(String inputUrl) {
        Map<String, String> metadata = new HashMap<>();
        try {
            Document doc = Jsoup.connect(inputUrl).get();
            extractMetaTags(doc, metadata);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return metadata;
    }

    private void extractMetaTags(Document doc, Map<String, String>
metadata) {
        Elements metaTags = doc.getElementsByTag("meta");
        for (Element metaTag : metaTags) {
            String name = metaTag.attr("name");
            String property = metaTag.attr("property");
            String content = metaTag.attr("content");
```

```
                if (!name.isEmpty()) {
                    metadata.put(name, content);
                } else if (!property.isEmpty()) {
                    metadata.put(property, content);
                }
            }
        }

        @Transactional
        private void updateTaskStatusToFailed(Task task) {
            task.setStatus(Status.FAILED);
            taskRepository.save(task);
        }
    }
```

This class is designated a Singleton EJB to ensure a single instance throughout the application. The magic happens with the `@ManagedExecutorDefinition` annotation:

- `name`: Establishes a JNDI lookup name (`"java:app/concurrent/taskScheduler"`) for the managed executor service.
- `hungTaskThreshold`: Configures a 120-second threshold for detecting and potentially mitigating tasks that run too long.
- `virtual = true`: This is the key to utilizing virtual threads, a new feature in JDK 21 and Jakarta EE 11. Virtual threads are lightweight and designed for high throughput, making them perfect for concurrent task processing needs.

By injecting the `ManagedExecutorService` and setting it to use virtual threads, the application can concurrently fetch and process URLs without the overhead of traditional threads, significantly enhancing scalability and resource utilization. This is a part of Jakarta Concurrency 3.1 specification.

The core workflow begins with a scheduled task-pulling mechanism triggered every 10 seconds. New tasks, identified by a **`"NEW"`** status, are fetched from the **`TaskRepository`** and passed to the **`execute()`** method. This method marks the task as "**`IN_PROGRESS`**" and saves its state to the database. The task is then submitted to the **`taskScheduler`**, our virtual thread-powered executor service, for processing.

The **`taskScheduler`** intelligently distributes tasks among virtual threads, each operating independently. Each thread fetches content from the task's URL using JSoup. JSoup is a Java library specifically designed to work with HTML. It provides a convenient way to parse HTML documents, extract data, and manipulate the document's structure.

Upon successful metadata extraction, the thread updates the task's status to "**`COMPLETED`**" and records the completion time in the database. Any errors during fetching or processing are handled by marking the task as "**`FAILED`**."

The use of virtual threads brings significant advantages. They are lightweight and consume minimal resources compared to traditional threads. The **`ManagedExecutorService`** efficiently manages a large pool of virtual threads, enabling concurrent processing of numerous tasks without overwhelming the system. Additionally, virtual threads simplify handling asynchronous operations, leading to cleaner and more maintainable code.

Checking the Application

Once you add those, run the application and create a Task using the **`createTask`** endpoint. Then navigate to the following log file:

```
target/cargo/configurations/payara/cargo-domain/logs/server.log
```

You will find something like this:

```
[2024-06-24T21:21:40.059-0400] [Payara 7.2024.1.Alpha1] [INFO] []
[org.eclipse.jakarta.hello.TaskScheduler] [tid: _ThreadID=232 _ThreadName=]
[timeMillis: 1719278500059] [levelValue: 800] [[
 Fetching content from URL: https://bazlur.ca/ executing on thread:
VirtualThread[#232]/runnable@ForkJoinPool-1-worker-1]]
```

This proves that the metadata extraction happens over the virtual threads.

If you want to get the extracted data, you can hit the following endpoint:

```
curl -X GET -H "Content-Type: application/json"
http://localhost:8080/jakartaee-hello-world/rest/tasks
```

You will get something like the following:

```
[
  {
    "completedAt": "2024-06-25T01:21:43.681362Z",
    "createdAt": "2024-06-25T01:21:33.853564Z",
    "inputUrl": "https://bazlur.ca/",
    "metaData": {
      "generator": "WordPress 6.5.4",
      "og:title": "Home-2 - A N M Bazlur Rahman",
      "og:site_name": "A N M Bazlur Rahman",
      "og:url": "https://bazlur.ca/"
    },
    "status": "COMPLETED",
    "taskId": 1,
    "updatedAt": "2024-06-25T01:21:43.682656Z",
    "version": 3
  }
]
```

Conclusion

Following this guide, you have successfully set up a Jakarta EE 11 application with Payara, created endpoints, and added a scheduler using virtual threads. This setup demonstrates the power and flexibility of Jakarta EE 11 and JDK 21, making development more efficient and scalable.

To verify the application, run it and create a task using the **`createTask()`**'s **`(POST /tasks)`** endpoint. Then, check the server logs to see the metadata extraction process happening over virtual threads. You can fetch the extracted data using the **`getAllTasks()`**'s **`(GET /tasks)`** endpoint.

The source code is available on GitHub.

This setup enhances scalability and resource utilization, ensuring smooth operation even under heavy workloads, thanks to virtual threads. Happy coding!