1．開発環境

1－1．ソフトウェア

ソフトウェア	バージョン	説明
Adoptium OpenJDK	17.0.11+9	
Eclipse IDE for Enterprise Java and Web Developers	2023-12	
WildFly	31.0.1.Final	WildFly Maven Plugin で使用
Apache Maven	3.9.6	

※ Eclipse を日本語化する場合

Pleiades 日本語化プラグインのサイトから Pleiades プラグイン単体をダウンロードします。

ダウンロードした zip を任意のディレクトリに展開し、zip 内の readme/readme_pleiades.txt の内容に従ってインストールした Eclipse を日本語化してください。

1－2．環境変数

環境変数	説明
JAVA_HOME	Adoptium OpenJDK 17.0.11+9 インストールディレクトリのパスを指定
MAVEN_HOME	Apache Maven 3.9.6 インストールディレクトリのパスを指定
Path	%Java_HOME%\bin、%MAVEN_HOME%\bin を追加

※ MAVEN_HOME は Maven 3.5 より不要となっていますが、説明を簡単にするために追加しています。

2．チュートリアル

2－1． RESTful Web Service

この項では Jakarta EE を使用して RESTful Web サービスを作成する方法を説明します。
作成するサービスは `http://127.0.0.1:8080/<プロジェクト名>/restfulservice/hello` で HTTP GET リクエストを受け入れ、次のような JSON ペイロードを応答するものとします。

```
{ "name": "world" }
```

作成するプロジェクトの最終的な構成は以下のようになります。（パッケージは任意）

```
.
│  pom.xml
│
└─src
    ├─main
    │  ├─java
    │  │  └─path
    │  │      └─to
    │  │          └─tutorial
    │  │              └─restfulwebservice
    │  │                      HelloApplication.java
    │  │                      HelloRecord.java
    │  │                      HelloResource.java
    │  │
    │  ├─resources
    │  └─webapp
    └─test
        ├─java
        └─resources
```

アプリケーションクラス

サービス名は `Application` クラスを継承したサブクラスで次のように宣言します。ここでは `restfulservice` という名前を指定しています。
配備するモジュールに `Application` のサブクラスが存在する場合、モジュール内のリソースクラスが検索され、Web リソースとして公開されます。

```
import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

@ApplicationPath("restfulservice")
public class HelloApplication extends Application {
}
```

リソースクラス

Jakarta EE アプリケーションでは、クライアントとの対話のターゲットを識別するリソースを公開するためのリソースクラスを作成します。`HelloResource.java` は以下のようになります。

```
package path.to.sample;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;

@Path("hello")
public class HelloResource {

    @GET
    @Produces({ MediaType.APPLICATION_JSON })
    public HelloRecord hello(@QueryParam("name") String _name) {
        if (null == _name || _name.trim().isEmpty()) {
            _name = "world";
        }

        return new HelloRecord(_name);
    }
}
```

提供するリソースとサービスは、グローバルアドレス空間を提供する URI によって識別されます。

`@Path` アノテーションは、ユーザが指定した URL とリクエストの処理を担当する Java クラスとの間の接続を確立します。

`@GET` アノテーションは、Jakarta REST によって定義された実行時アノテーションの一つであり、同様の名前の HTTP メソッドに対応し、上記のコードではユーザがリソースにアクセスするには `HTTP GET` メソッドが必要であることを示します。Jakarta REST では、一般的な HTTP メソッドである GET、POST、PUT、DELETE、及び HEAD の一連のリクエストメソッド指定子が用意されています。

`@Produces` アノテーションは、HTTP リクエストまたはレスポンスの MIME メディアタイプを指定することができる。上記のコードでは、JSON 形式の応答を返すための `application/json` を指定します。
戻り値の `HelloRecord` オブジェクトを JSON にシリアライズしてレスポンスが生成されます。

`@QueryParam` アノテーションは、リクエスト URI からクエリパラメータを抽出して引数に指定することができます。

レコードクラス

`hello(String)` メソッドは `HelloRecord` を返すように定義されています。`record` は Java 16 で追加された新しいレコードクラスです。

```
public record HelloRecord(String _name) {
}
```

レコードクラスを従来の POJO にする場合は以下のようになります。

```
public final class HelloRecord {
    private final String name;

    public HelloRecord(String _name_) {
        this.name = _name_;
    }

    public String name() {
        return this.name;
    }
}
```

プロジェクトの設定

Maven を使用して CLI からプロジェクトを実行する方法を説明します。
このチュートリアルでは `WildFly` を使用しますが、Jakarta EE と互換性のある他のランタイムはこちらのサイトで確認できます。
実行のためには、`pom.xml` ファイルに依存関係とプラグインを追加する必要があります。

```
...
    <packaging>war</packaging>
...
    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupId>
            <artifactId>jakarta.jakartaee-web-api</artifactId>
            <version>10.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
...
    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.4.0</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>4.2.2.Final</version>
                <configuration>
                    <version>31.0.1.Final</version>
                    <server-config>standalone-full.xml</server-config>
                </configuration>
            </plugin>
        </plugins>
    </build>
...
```

wildfly-maven-plugin は、Jakarta EE アプリケーションのデプロイ、再デプロイ、アンデプロイ、または実行に使用されます。

プロジェクトの実行

WildFly のローカルインスタンスを実行する方法はいくつかありますが、通常の実行は wildfly-maven-plugin の Run Examplesを、開発時の実行は Dev Examples を参照してください。
このチュートリアルでは下記の Maven ゴールを使用します。

```
mvn clean package wildfly:run
```

上記の Maven ゴールはアプリをビルドし、Wildfly に配置します。Wildfly がインストールされていない場合は、`version` で指定したバージョンの Wildfly を自動的にダウンロードして実行し、war ファイルがデプロイされます。`version` を省略した場合は最新の安定板がダウンロードされます。

動作確認

WildFly が正常に起動した場合、サービスが実行されているので下記の URL にアクセスするとレスポンスが返されます。

```
http://127.0.0.1:8080/restful-web-service/restfulservice/hello
または
http://127.0.0.1:8080/restful-web-service/restfulservice/hello?name=XYZ
```

コマンドラインから以下のように確認することもできます。

```
curl -v http://127.0.0.1:8080/restful-web-service/restfulservice/hello
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
> GET /restful-web-service/restfulservice/hello HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 16
< Date: Thu, 02 May 2024 06:53:53 GMT
<
{"name":"world"}* Connection #0 to host 127.0.0.1 left intact
```

URL の構造は以下のようになっています。

```
http://<hostname>:<port>/<context-root>/<REST-config>/<resource-config>
```

URL の各パターンは下記の通りです。

URL パターン	説明
hostname	WildFly が実行されているサーバのホスト名または IP アドレス
port	WildFly の HTTP 受信ポート。デフォルトは 8080
context-root	アプリケーションに割り当てられるコンテキストルート。デフォルトは WAR ファイル名（拡張子除く）
REST-config	`@ApplicationPath` アノテーションに指定した値。未指定の場合は単に省略される
resource-config	`@Path` アノテーションに指定した値

２－２． Servlet Application

この項では、Jakarta Servlet を使用して単純な Servlet アプリケーションを構築する方法を追って説明します。

サーブレットの例

Jakarta EE でサーブレットを作成する手順について説明します。
最初に `HttpServlet` を拡張する新しいクラスを作成します。次に、サーブレットで処理する HTTP メソッド（GET リクエストを処理する場合は `doGet`、POST リクエストを処理する場合は `doPost`）をオーバーライドします。
例えば、GET リクエストで単純なテキストを取得するサーブレットを作成するには、次のように `HttpServlet` を拡張して `doGet` メソッドをオーバーライドします。

```
import java.io.IOException;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/hello")
public class HelloWorldServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest _req, HttpServletResponse _res) throws
ServletException, IOException {
        _res.getWriter().println("Hello, World!");
    }
}
```

`@WebServlet` アノテーションは、サーブレットクラスと URL マッピングを定義するために使用されます。クラス定義に `@WebServlet("hello")` がある場合、このサーブレットがアプリケーションのコンテキストルート内の `/hello` パスを対象とする HTTP リクエストに応答することを示します。

HTML フォーム

サーブレットにリクエストを送信するための `form` 要素を含む HTML ファイル `coffee_preferences.html` を `WEB-INF` ディレクトリに配置します。

```
<!DOCTYPE html>
<html>
  <head>
     <title>Coffee Preferences</title>
  </head>
  <body>
    <form action="storePreferences" method="post">
       <p>Select your coffee preferences:</p>
       <input type="checkbox" name="coffeeType" value="Black"> Black<br>
       <input type="checkbox" name="coffeeType" value="Latte"> Latte<br>
       <input type="checkbox" name="coffeeType" value="Cold Brew"> Cold Brew<br>
       <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

Jakarta EE における `WEB-INF` ディレクトリは、クライアントが直接アクセスすることができないリソースを配置する安全なディレクトリとして機能します。ファイルを `WEB-INF` ディレクトリに配置すると、クライアントのブラウザからの直接 URL 指定ではアクセスできなくなります。これにより、サーブレットを介してのみアクセスできるリソースに追加のセキュリティ層が提供されます。

設定を保存するサーブレット

`coffee_preferences.html` で選択された内容をセッションに保存するサーブレットを作成します。

```
import java.io.IOException;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;

@WebServlet("/storePreferences")
public class StorePreferencesServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest _req, HttpServletResponse _res) throws
ServletException, IOException {
        _req.getRequestDispatcher("/WEB-
INF/coffee_preferences.html").forward(_req, _res);
    }

    @Override
    protected void doPost(HttpServletRequest _req, HttpServletResponse _res)
throws ServletException, IOException {
        final String[] coffeeTypes = _req.getParameterValues("coffeeType");
        final HttpSession session = _req.getSession();
        session.setAttribute("userCoffeeTypes", coffeeTypes);
        _res.sendRedirect("coffeeDashboard");
    }
}
```

このサーブレットは、`@WegServlet` アノテーションを介して `/storePreferences` URL にマッピングされます。doGet メソッドは、リクエストを `/WEB-INF/coffee_preferences.html` にある HTML ページに転送します。この HTML には `form` 要素が含まれています。doPost メソッドは、送信されたフォームからユーザが選択した設定を取得し、属性名 `userCoffeeTypes` として HTTP セッション内に文字列配列として保存されます。設定が保存されると、ユーザは `coffeeDashboard` サーブレットにリダイレクトされます。

パーソナライズされたコーヒーの推奨事項を生成するサーブレット

次に、保存されたコーヒーの好みを読み取り、推奨されるコーヒーのリストを動的に生成する別のサーブレットを作成します。

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;

@WebServlet("/coffeeDashboard")
public class CoffeeDashboardServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private static final Map<String, String> COFFEE_DESCRIPTIONS = new HashMap<>
();
    static {
        COFFEE_DESCRIPTIONS.put("Black", """
                    <p>Black coffee has a robust flavor, perfect for those who
prefer a coffee with some bite.</p>
                    <p>Try brewing methods like French Press or Aeropress for an
enjoyable black coffee experience.</p>
                """);
        COFFEE_DESCRIPTIONS.put("Latte", """
                    <p>A latte is a creamy delight, suitable for people who enjoy
a smoother and less harsh flavor.</p>
                    <p>Experimenting with various syrups and sweeteners can
elevate your latte experience.</p>
                """);
        COFFEE_DESCRIPTIONS.put("Cold Brew",
                """
                            <p>Cold brew coffee tends to be smoother and less
acidic. It's perfect for those hot summer days.</p>
                            <p>Try brewing a batch in the fridge overnight for a
refreshing morning pick-me-up.</p>
                        """);
    }

    @Override
    protected void doGet(HttpServletRequest _req, HttpServletResponse _res) throws
ServletException, IOException {
        final HttpSession session = _req.getSession();
        final String[] coffeeTypes = (String[])
session.getAttribute("userCoffeeTypes");
```

```
        if (null == coffeeTypes || 0 == coffeeTypes.length) {
            this.handleNoCoffeeTypes(_res);
            return;
        }

        final PrintWriter out = _res.getWriter();
        out.println("""
                    <html>
                    <body>
                    <h1>Your Personalized Coffee Dashboard</h1>
                """);

        for (final String coffeeType : coffeeTypes) {
            final String additionalInfo = COFFEE_DESCRIPTIONS.get(coffeeType);
            out.println("""
                        <h2>Recommended %s</h2>
                        <p>Here are some %s blends you might enjoy.</p>
                        %s
                    """.formatted(coffeeType, coffeeType, additionalInfo));
        }

        out.println("""
                    </body>
                </html>
                """);
    }

    private void handleNoCoffeeTypes(HttpServletResponse _res) throws IOException
{
        final PrintWriter out = _res.getWriter();
        out.println("""
                <html>
                    <body>
                        <h1>No Coffee Types Found</h1>
                        <p>Please select at least one type of coffee.</p>
                    </body>
                </html>
                """);
    }
}
```

`CoffeeDashboardServlet` クラスは、ユーザのコーヒーの好みに基づいてパーソナライズされたコーヒーダッシュボードを生成します。このコードでは `COFFEE_DESCRIPTIONS` という静的 `Map` を使用して、コーヒーの説明を保存しています。

サーブレットは `doGet` メソッドをオーバーライドして HTTP GET リクエストを処理します。このメソッド内では、まず HTTP セッションに保存されているユーザのコーヒーの好みを取得します。設定が見つからない場合は、`handleNoCoffeeTypes` メソッドを呼び出してデフォルトのメッセージが表示されます。それ以外の場合は、選択したコーヒーの種類を反復処理し、`COFFEE_DESCRIPTIONS` マップから対応する説明を取得します。 最後に、この情報をパーソナライズされたダッシュボードに表示するための HTML コンテンツを生成します。

動作確認

このチュートリアルでは下記の Maven ゴールを使用します。

```
mvn clean package wildfly:run
```

WildFly が正常に起動した場合、サービスが実行されているので下記の URL にアクセスするとレスポンスが返されます。

```
http://127.0.0.1:8080/servlet-application/storePreferences
```

2 – 3． Jakarta Persistence

Jakarta Persistence API は、Java アプリケーションでリレーショナルデータを管理するためのオブジェクトリレーショナルマッピング (ORM) フレームワークを提供する Java プログラミング言語仕様です。　これにより、データベース アクセスが簡素化され、開発者は SQL ステートメントではなくオブジェクトとクラスを操作できるようになります。　この項では、Jakarta Persistence を使用してデータを保存および取得する方法を説明します。

JSON ペイロードの例

このアプリケーションで使用する JSON ペイロードのサンプルです。このサービスは、製品の ID、名前、価格を含む JSON ペイロードを使用します。

```
{
  "id": 1,
  "name": "Coffee-A",
  "price": "2.75"
}
```

persistence.xml

Jakarta Persistence を使用する前に、まず、persistence.xml という Jakarta Persistence 構成ファイルを作成してデータベース接続を構成します。 このファイルは、プロジェクトの resources/META-INF フォルダーの下に配置します。 persistence.xml ファイルでは、JDBC 接続設定またはデータソース JNDI 名を指定できます。ここでは、persistence.xml 内で指定されている WildFly のデフォルトの H2 データベースを使用できます。 下記の内容の persistence.xml ファイルを resources/META-INF フォルダに作成します。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_1.xsd" version="3.0">

    <persistence-unit name="coffees">
        <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
        <properties>
            <property name="jakarta.persistence.schema-generation.database.action"
value="drop-and-create" />
            <property name="jakarta.persistence.sql-load-script-source"
value="META-INF/initial-data.sql" />
            <property name="eclipselink.logging.level.sql" value="FINE" />
            <property name="eclipselink.logging.parameters" value="true" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

この persistence.xml ファイルは、「coffees」という名前の永続性ユニットを設定します。 永続化ユニットにデータベース接続管理に使用する JDBC データソースの JNDI 名を指定します。WildFly の H2 データベースではデフォルトで「java:jboss/datasources/ExampleDS」です。 JNDI 名の構成は通常、%WILDFLY_HOME%/standalone/configuration ディレクトリの standalone.xml ファイルにあります。(WILDFLY_HOME は WildFly のインストールディレクトリです。)

永続化ユニットには、Jakarta Persistence プロバイダの動作を構成するいくつかのプロパティが含まれています。 `jakarta.persistence.schema-generation.database.action` プロパティは、データベーススキーマを生成するときに Jakarta Persistence プロバイダが実行するアクションを指定します。例えば、以下のオプションが用意されています。

- none：データベーススキーマを生成しません。
- create：データベーススキーマを作成します。
- drop：データベーススキーマを削除します。
- drop-and-create：既存のデータベーススキーマを削除し、新しいスキーマを作成します。

他にも `create-only`、`drop-and-create-script`、`create-script` などの他のオプションも使用できます。

`jakarta.persistence.sql-load-script-source` プロパティは、永続ユニットが初期化されるときに実行される SQL スクリプトの場所を指定します。　ここでは、スクリプトを META-INF/initial-data.sql ファイルとして作成し、以下の挿入クエリをその中に記述します。

```
INSERT INTO coffee (name, price) VALUES ('Coffee-A', 2.75);
INSERT INTO coffee (name, price) VALUES ('Coffee-B', 1.99);
INSERT INTO coffee (name, price) VALUES ('Coffee-C', 3.25);
INSERT INTO coffee (name, price) VALUES ('Coffee-D', 2.99);
```

このスクリプトは、サーバの起動時に、テストやデモンストレーションに使用できるデータをデータベースに挿入します。

`eclipselink.logging.level.sql` や `eclipselink.logging.parameters` などの他のプロパティは、Jakarta Persistence プロバイダのログ機能を構成するために使用されます。　`hibernate.show_sql` プロパティは、Hibernate Jakarta Persistence プロバイダの SQL クエリログ機能を有効にするために使用されます。

エンティティ

最初のエンティティを作成します。 エンティティとは、データベースのテーブルに対応する Java クラスです。この記事では、組み込みの H2 データベースを使用してデータを格納します。 以下は作成する `Coffee` クラスです。

```
import java.io.Serializable;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Coffee implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String price;

    // Getter、Setter は省略
}
```

エンティティクラスには複数の注釈が含まれています。 `@Entity` 注釈は、Java クラスをエンティティクラスとして指定し、クラスフィールドがテーブル列に対応するデータベーステーブルを表します。 `@Id` 注釈は、エンティティクラスのフィールドまたはプロパティを主キーとしてマークします。 さらに、`@GeneratedValue` 注釈を `@Id` と一緒に使用して、主キー値を自動的に生成する必要があることを示しています。

Jakarta Persistence の本質は、Java プログラマがテーブルをオブジェクトとして操作できるようにすることです。プログラマはこれらのエンティティのインスタンスを作成することができ、Jakarta Persistence はそれをデータに変換してデータベースに格納します。 オブジェクトが必要な場合、Jakarta Persistence はデータを取得して、簡単に使用できるようにオブジェクトに変換します。

リポジトリ

ここで、`Coffee` エンティティに対する CRUD 操作を提供する `CafeRepository` クラスを作成します。

```
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.eclipse.jakarta.model.entity.Coffee;

import java.lang.invoke.MethodHandles;
import java.util.List;
import java.util.Optional;
import java.util.logging.Logger;

@Stateless
public class CafeRepository {
    private static final Logger logger =
Logger.getLogger(MethodHandles.lookup().lookupClass().getName());

    @PersistenceContext
    private EntityManager em;

    public Coffee create(Coffee coffee) {
        logger.info("Creating coffee " + coffee.getName());
        em.persist(coffee);

        return coffee;
    }

    public List<Coffee> findAll() {
        logger.info("Getting all coffee");
        return em.createQuery("SELECT c FROM Coffee c",
Coffee.class).getResultList();
    }

    public Optional<Coffee> findById(Long id) {
        logger.info("Getting coffee by id " + id);
        return Optional.ofNullable(em.find(Coffee.class, id));
    }

    public void delete(Long id) {
        logger.info("Deleting coffee by id " + id);
        var coffee = findById(id)
            .orElseThrow(() -> new IllegalArgumentException("Invalid coffee Id:" +
id));
        em.remove(coffee);
    }

    public Coffee update(Coffee coffee) {
        logger.info("Updating coffee " + coffee.getName());
        return em.merge(coffee);
```

```
    }
}
```

このクラスには `@Stateless` の注釈が付けられており、ステートレスセッション Bean になります。ステートレスセッション Bean は、メソッド呼び出し間でクライアントとの会話状態を維持する必要がないシナリオ向けに設計されています。つまり、ステートレスセッション Bean は、メソッド呼び出し間でクライアント固有のデータを一切記憶しません。

EntityManager

`EntityManager` は、Jakarta Persistence でエンティティを管理するための主要なインターフェースです。`@PersistenceContext` で注釈が付けられ、`EntityManager` のインスタンスがクラスに自動的に挿入されます。

create メソッド

`Coffee` エンティティをパラメータとして受け取り、`EntityManager` を使用してオブジェクトを永続化します。戻り値として永続化された Coffee オブジェクトが返されます。

findAll メソッド

データベースからすべての `Coffee` エンティティを取得します。Jakarta Persistence クエリを作成して実行し、`Coffee` オブジェクトのリストを返します。

findById メソッド

指定された ID を持つ `Coffee` エンティティを検索します。見つかった場合は、エンティティを含む `Optional<Coffee>` を返します。見つからない場合は、空の `Optional` を返します。

delete メソッド

指定された ID を持つ `Coffee` エンティティを検索します。見つかった場合は、`EntityManager` を使用してエンティティを削除します。

update メソッド

`Coffee` オブジェクトをパラメータとして受け取り、`EntityManager` を使用してデータベース内の既存の `Coffee` エンティティを更新します。更新された `Coffee` オブジェクトが返されます。

REST エンドポイントの追加

最後に、リモート REST クライアントからコーヒー サービスにアクセスできるように REST エンドポイントを追加します。

```
import java.lang.invoke.MethodHandles;
import java.util.List;
import java.util.logging.Logger;

import jakarta.inject.Inject;
import jakarta.persistence.PersistenceException;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.DELETE;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.WebApplicationException;
import jakarta.ws.rs.core.Response;

@Path("coffees")
public class CafeResource {
    private final Logger logger =
Logger.getLogger(MethodHandles.lookup().lookupClass().getName());

    @Inject
    private CafeRepository cafeRepository;

    @GET
    @Path("{id}")
    @Produces("application/json")
    public Coffee findCoffee(@PathParam("id") Long id) {
        logger.info("Getting coffee by id " + id);
        return cafeRepository.findById(id).orElseThrow(() -> new
WebApplicationException(Response.Status.NOT_FOUND));
    }

    @GET
    @Produces("application/json")
    public List<Coffee> findAll() {
        logger.info("Getting all coffee");
        return cafeRepository.findAll();
    }

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public Coffee create(Coffee coffee) {
        logger.info("Creating coffee " + coffee.getName());
        try {
```

```
            return cafeRepository.create(coffee);
        } catch (PersistenceException ex) {
            logger.info("Error creating coffee " + coffee.getName());
            throw new WebApplicationException(Response.Status.BAD_REQUEST);
        }
    }

    @DELETE
    @Path("{id}")
    public void delete(@PathParam("id") Long id) {
        logger.info("Deleting coffee by id " + id);
        try {
            cafeRepository.delete(id);
        } catch (IllegalArgumentException e) {
            logger.info("Error deleting coffee by id " + id);
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
    }

    @PUT
    @Consumes("application/json")
    @Produces("application/json")
    public Coffee update(Coffee coffee) {
        logger.info("Updating coffee " + coffee.getName());
        try {
            return cafeRepository.create(coffee);
        } catch (PersistenceException ex) {
            logger.info("Error updating coffee " + coffee.getName());
            throw new WebApplicationException(Response.Status.BAD_REQUEST);
        }
    }
}
```

`CafeResource` クラスは、`Coffee` オブジェクトの作成、取得、更新、削除などの CRUD 操作を実行するためのさまざまな HTTP エンドポイントを公開します。

1. `@Path("coffees")` アノテーションは、Web サービスの基本パスを `"/coffees"` に設定します。このクラスのすべての HTTP エンドポイントは、このパスを基準とします。
2. `@Inject` アノテーションは、`Coffee` エンティティの永続化操作を処理する `CafeRepository` クラスのインスタンスを挿入するために使用されます。
3. `@GET`、`@POST`、`@PUT`、`@DELETE` の各アノテーションは、クラス内の対応するメソッドの HTTP メソッドを定義します。
4. `@Path("{id}")` アノテーションは、`@GET` アノテーションおよび `@DELETE` アノテーションと組み合わせて使用され、ID で `Coffee` エンティティを取得または削除するためのパスパラメータ「id」を指定します。
5. `@Produces` および `@Consumes` アノテーションは、メソッドが応答として生成したり、入力として消費したりできるメディアタイプ (`application/json` など) を定義するために使用されます。この場合、メソッドは `Coffee` エンティティの JSON 表現を受け入れて返します。

動作確認

Maven を使用して、次のコマンドを実行してアプリケーションをビルドして実行します。

```
mvn clean package wildfly:run
```

アプリケーションは Web ブラウザまたは REST クライアントからアクセスできます。curl をコマンドライン REST クライアントとして使用して、サービスと対話できます。

新しい Coffee エンティティを作成

```
curl -X POST http://localhost:8080/jakarta-persistence/rest/coffees ^
    -H 'Content-Type: application/json' ^
    -d '{ "id": 1, "name": "Coffee-A", "price": "2.75" }'
```

すべての Coffee エンティティを取得

```
curl -X GET http://localhost:8080/jakarta-persistence/rest/coffees ^
     -H 'Content-Type: application/json'
```

特定の Coffee エンティティを ID で取得

```
curl -X GET http://localhost:8080/jakarta-persistence/rest/coffees/2 ^
     -H 'Content-Type: application/json'
```

既存の Coffee エンティティを更新

```
curl -X PUT http://localhost:8080/jakarta-persistence/rest/coffees ^
     -H 'Content-Type: application/json' ^
     -d `{ "id": 1, "name": "Coffee-A", "price": "2.75" }`
```

ID で Coffee エンティティを削除

```
curl -X DELETE http://localhost:8080/jakartaee-hello-world/rest/coffees/1 ^
     -H `Content-Type: application/json`
```

２－４．Jakarta Faces

Jakarta Faces は、Java Web アプリケーションのユーザインターフェイスの作成を簡素化するフレームワークです。MVC (モデル、ビュー、コントローラ) アーキテクチャに準拠しており、ビューは UI コンポーネントを表し、アプリケーション ロジックはコントローラーとモデルレイヤーに存在します。

Faces の有効化

Faces サーブレットは、JSF リクエストを処理するためのエントリポイントとして機能します。アプリケーション内で使用されるクラスに @FacesConfig アノテーションを追加すると、Jakarta Faces とその CDI 固有の機能がアクティブ化されます。

Faces サーブレット

構成には web.xml 記述子を使用することもできます。web.xml 内の `servlet` 要素を見つけて、次のコードを追加します。

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

この構成は、JSF ビューでよく使用される `.xhtml` 拡張子で終わるリクエストを処理するようにサーブレットに指示します。ビューの命名規則に基づいてパターンを調整できます。

拡張子なしのマッピング

JSF 4.0 では拡張子なしのマッピングが導入され、`.xhtml` 拡張子のないビューにアクセスできるようになりました。これを有効にするには、`web.xml` ファイルに次のコンテキストパラメータを追加します。

```
<context-param>
    <param-name>jakarta.faces.AUTOMATIC_EXTENSIONLESS_MAPPING</param-name>
    <param-value>true</param-value>
</context-param>
```

beans.xml ファイルの追加

`beans.xml` ファイルを作成し、そのファイルを `src/main/webapp/WEB-INF` ディレクトリ内に配置します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"
       bean-discovery-mode="annotated"
       version="4.0">
</beans>
```

Facelets

`Facelets` は、Web ページの構造とレイアウトを定義するために Jakarta Faces で使用されるテンプレート言語です。 `Facelets` テンプレートは通常、`.xhtml` 拡張子を持ち、XHTML マークアップと Jakarta Faces コンポーネント タグおよび式の組み合わせで構成されます。これらのテンプレートには、静的コンテンツだけでなく、Jakarta Faces コンポーネントとマネージド Bean によって生成された動的コンテンツも含めることができます。

Facelets 標準タグライブラリ

以下の表は、Faces 4.0 の Faceletsでサポートされている標準ライブラリです。

Prefix	URN	Examples
xmlns:h	jakarta.faces.html	h:head, h:inputText
xmlns:f	jakarta.faces.core	f:facet, f:actionListener
xmlns:faces	jakarta.faces	faces:id, faces:value
xmlns:fn	jakarta.tags.function	fn:toLowerCase, fn:contains
xmlns:ui	jakarta.faces.facelets	ui:component, ui:include
xmlns:c	jakarta.tags.core	c:forEach, c:if
xmlns:pt	jakarta.faces.passthrough	pt:type, pt:placeholder
xmlns:cc	jakarta.faces.composite	

ビューの作成

簡単な Faces ページを作成します。次の内容を含む index.xhtml という名前の新しい XHTML ファイルを src/main/webapp ディレクトリに作成します。

```
<!DOCTYPE html>
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="jakarta.faces.core"
      xmlns:jsf="jakarta.faces"
      xmlns:h="jakarta.faces.html">
    <f:view>
        <h:head>
            <title>Facelets Example</title>
        </h:head>
        <h:body>
            <h1>Welcome to Jakarta Faces!</h1>
            <h:form>
                <h:inputText value="#{userBean.username}" />
                <h:commandButton value="Submit" action="#{userBean.submit}" />
            </h:form>
            <h:outputText value="Welcome, #{userBean.username}" rendered="#
{userBean.submitted}" />
        </h:body>
    </f:view>
</html>
```

単純な XHTML ドキュメント構造を定義します。　フォーム要素を作成するために、Jakarta Faces コンポーネント タグ (例: `h:form`、`h:inputText`、`h:commandButton`) を含めます。　コンポーネントを管理対象 Bean のプロパティとメソッドにバインドするために、EL (式言語) 式 (例: `#{userBean.username}`、`#{userBean.submit}`、`#{userBean.submitted}`) を使用します。　`userBean` は、ユーザー入力の処理とフォーム送信の処理を担当する CDI Bean です。

バッキング Bean の作成

作成したビューに対応する、Jakarta Faces のバッキング Bean (CDI Bean) を作成します。

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Named;

@Named
@RequestScoped
public class UserBean {
    private String username;
    private boolean submitted;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public boolean isSubmitted() {
        return submitted;
    }

    public void setSubmitted(boolean submitted) {
        this.submitted = submitted;
    }

    public String submit() {
        // ユーザ入力を処理したり、必要なアクションを実行します
        this.submitted = true;
        return null; // 同じページへ遷移し画面を更新します
    }
}
```

`@Named` アノテーションは、このクラスを Jakarta Faces フレームワークによって管理される CDI Bean としてマークします。 `@RequestScoped` アノテーションは、クライアントがサーバに対して行う HTTP リクエストごとに、管理対象 Bean の新しいインスタンスが作成されることを指定します。 `username` プロパティは、ユーザー入力フィールドに入力した値を表します。 `submited` プロパティは、フォームが送信されたかどうかを示します。 `submit()` メソッドは、フォームが送信されると呼び出されます。このメソッドに必要なアクションを実装します。

動作確認

Maven を使用して、次のコマンドを実行してアプリケーションをビルドして実行します。

```
mvn clean package wildfly:run
```

WildFly が正常に起動した場合、サービスが実行されているので下記の URL にアクセスするとレスポンスが返されます。

```
http://localhost:8080/jakarta-faces/index.xhtml
```