

Java™ magazine

By and for the Java community 

JANUARY/FEBRUARY 2016

UIの裏側 WEB アプリケーション 開発

- 05 **SPRING BOOT:**
早く簡単な
WEBアプリケーション
- 13 **JAX-RS 2.0:**
徹底活用のすすめ



表紙画像: I-HUA CHEN

02
編集長より
2015年の言語の盛衰

35
Java 9へのパス
**プレリリース版で学ぶ
Java 9モジュール**
Ben Evans
Javaの次期メジャー・リリースの中
核となるモジュールの導入に備える

41
コネクタ
**複数のDockerコンテナの
使用**
Arun Gupta
Dockerコンテナのクラスタを組み
合わせ、ハウスキーピングの手間が
かからないJava EEアプリケーショ
ンを実行する

13
**JAX-RS 2.0:
良いものはすべて利用する**
新規クライアントAPI、フィルタ、
インターセプタ、その他の有用なREST機能

23
**WEBSOCKETを使用した双方向の
データ・プッシュ**
Danny Coward
WebSocketの長期持続型コネクションを
使ったシンプルなチャット・アプリの構築

49
マイクロサービス
**KumuluzEE: Java EEでの
マイクロサービスの構築**
Tilen FaganelおよびMatjaz B. Juric
オープンソースのKumuluzEEフレー
ムワークを使用して、標準Java EE
APIで自己完結型マイクロサービス
を開発する

58
Fix This
Simon Roberts
最新のコードの問題に挑戦
22
注目のJDK強化提案
JEP 254: Compact Strings



//from the editor /

Andrew Binstock: Java Magazine編集長。以前はDr. Dobbの編集者であり、さまざまな技術文書の出版も担当。また、オープンソースPDFライブラリ「iText」の商業部門として、iText Softwareを共同で創業。さまざまなオープンソース・プロジェクトにも貢献している。著書に、『Practical Algorithms』(Addison-Wesley、1995)を含む3冊のプログラミング関連書籍がある。



2015年の言語の盛衰

去年はJava言語、そしてJVM言語にとって、いつになく良い年でした。

私 はここ数年、新年最初の編集作業で、その前年のプログラミング言語の状況を見てきました。言語の人気に対する一般的認識はゆっくりと移り変わるものですが、ときにプログラミング言語は急激に低迷したり (Objective-C)、逆に思いもよらず浮上することもあります (Java。後ほど説明します)。その人気をはかる上で広く使われている手法がTIOBE Indexです。TIOBE Indexでは、言語が検索された回数をカウントして、その結果を検索総数に対する割合として標準化します。Web検索がはたして人気をはかる正確な代替指標であるのかどうかは議論の余地がありますが、TIOBE Indexには大きな長所が1つあります。それは、15年分を遡って指標のデータを取得できることでの数年にわたる傾向を簡単に確認できます。

当然ながら、言語の人気について確実に分析するには他のソースも必要です。私は[GitHub](#)から抽出したデータや、[Open Hub](#)、[Google Trends](#)も使っています。GitHubは、オープンソース・プロジェクトや非公開プロジェクトのホストとして一般的です。また、Open Hubはすべてのアクティブなオープンソース・プロジェクトを対象として調査しています。それぞれの基準によって定量化されるデータは異なります

が、データの背景も見て、その後そのデータからどのような有益な情報が得られるかを判断することが大切です。

ほとんどの基準で、Javaにとっては最高の1年でした。先日、TIOBEはJavaに2015年の「Language of the Year」を贈りました。どの言語よりも検索の割合が急激に伸びたという理由からです。TIOBEは、その躍進の原因をAndroidでのJavaの利用だと見ています。私もそうと思いますが、それは物語の一面に過ぎません。Java 8が急速に採用されたことも一因でしょう。「Javaはもう終わりなのか」を問う、思わずクリックしたくなる記事が流行りましたが、Javaの躍進はその流行を終わらせました。結局、そのような記事ではいつも長々とした説明の後に「Javaは終わっていない」と締めくくられていたのですが。

GitHubは、若いプログラマーの間の人気をはかる良い手段です。ここでもJavaは活況です。現在の人気順でJavaの上にあるのはJavaScriptだけで、Javaは驚くほど順位を上げました。GitHubは最初の頃、Rubyコミュニティで人気を集めたため、2008年のGitHub人気言語ランキングでJavaは7位。1位はRubyでした。それから2015年までの間にJavaが5つ順位を上げたのは、他に類を見ません。その同じ期間で、

CREATE
THE FUTURE

oracle.com/java



写真: BOB ADLER/GETTY IMAGES



02

//from the editor /

3つ以上順位を上げたプログラミング言語はないのですから。私はこの人気は今後も続くと思っています。Javaのクラウドでの採用率が高いこと(メジャーなクラウド・プロバイダのすべてがJavaをサポートしています)、そしてInternet of Thingsで中心的な役割を果たしていることがその理由です。

さて、サード・パーティのJVM言語を見ると、TIOBEのトップ50またはOpen Hubのランク内に入った新参の言語はGroovy (17位)とScala (30位)だけです。TIOBEでは、Groovyが記録的な1年になりました。その原因を探るのは難しいのですが、Groovyに対する一番の不満であったパフォーマンス面が改善されたことが影響したことは間違いありません。オープンソース・プロジェクトでは、Scalaが優勢です。ここから推測すると、Groovyはビジネス環境での人気が高いのではないのでしょうか。このようなビジネスへの移行は、Scalaが末端から抜け出すために今後数年で歩まなければならない道のりであると言えます。Scalaがこのキャズムを超えられるか非常に楽しみです。何年前か、Scalaに対して次のような不満が出ていました。新しいリリースのバイナリ互換性がないこと。コンパイルに時間がかかること。そして言語が複雑であること。これらのうち後の2つの懸念点は今も大きな障害として残っています。その間にも、Kotlinなどの直接の競合とみなされている言語がScalaにプレッシャーをかけています。一方、JavaもScalaにとって間接的なプレッシャーです。Scalaの自慢は、開発者がオブジェクト指向(OO)と関数型パラダイムを併

用できること。しかし、Scalaよりもまだまだ控えめですが、Java 8で関数型プログラミングの要素が取り入れられました。このことが、関数型とOOのハイブリッド品質を求めてScalaの導入を検討していた企業を踏みとどまらせるかもしれません。

TIOBEは、構文がLispに似ていて現在はJVM言語の3番手に落ちているClojureが、まもなくトップ50の言語に名を連ねるだろうと予測しています。関数型パラダイムがお好きな開発者にとって嬉しいニュースでしょう。その他の関数型言語、たとえばHaskellやErlangも、ともにトップ40入りを果たしています。

JVM言語以外を取り上げると、複数の指標に実に面白い傾向が見られます。どうやらJavaScriptの人気がピークに達したようです。2、3年ほど前は、フロントエンドでどこでも(Webでもモバイルでも)使えること、そしてNode.jsが出現したことで、JavaScriptが新しい万能型プログラミング言語になるのではと言われていました(「Atwoodの法則」を参照してください)。しかし、この言語にはさまざまな制約があり、大規模プロジェクトでの利用を難しくしています。結果として、JavaScriptの発展が、Dart、CoffeeScript、TypeScriptなどの代替言語へと派生しています。その中でも個人的に高く評価しているのがTypeScriptです。TypeScriptは、開発者の間でもその勢いを増しているようです。ECMAScript 6標準(ECMAScript 2015とも呼ばれます)が昨年の6月に承認されたことでJavaScriptの人気はさらに上がるのか、興味が尽きません。その結果については来年に確認しましょう。

言語の機能や、今後どの言語が採用されるのかに関する予想は、プログラミングに関する議題としてもっとも楽しめるものです。異なる意見をお持ちの方は、ぜひお知らせください。

編集長Andrew Binstock
javamag_us@oracle.com
[@platypusguy](https://twitter.com/platypusguy)

CREATE THE FUTURE
oracle.com/java

20 YEARS
1995-2015

Java™

ORACLE®



[illegible]

最後に、実績十分のJavaServer Faces (JSF) を使う開発者のために、今月のJava MagazineではOmniFaces (※US) を詳しく解説し、チュートリアルを取り上げています。OmniFacesは2015年のDuke's Choice Awardに輝いた、多岐にわたるJSFユーティリティ・ライブラリです。



JOSH LONG

Spring Bootを使ってみる

長く待ち望まれてきた簡単なWebアプリケーション開発を実現する新しいSpringフレームワーク

Spring Bootは、独特の手法によってSpringやJava EEを含むさまざまな技術を組み合わせて統合し、本番環境向けのWebアプリケーションを素早く構築できるようにした仕組みです。シンプルなUIからさまざまな技術にアクセスできるため、適切なコンポーネントを連動させて短時間で部品を組み合わせることが可能で、そこにアプリケーション固有のナレッジを追加できます。このアプローチに対する人気が高まっているのは、定型コードの量やハウスキeping作業を大幅に減らすことができるためです。本記事では、やや複雑なUIと永続性レイヤーを含むシンプルなアプリケーションを設定する手順について、順を追って説明します。また、現在のソフトウェア開発手法に従ったテストについても触れます。

ます。JVMのバージョンは、できる限り1.8を選ぶようにします。その他の2つのバージョンはすでに公式サポート期限が過ぎているためです。

本記事では、さまざまな関連テクノロジーについても簡単に紹介します。紹介するテクノロジーを組み合わせることで便利なアプリケーションを作るのがどれほど簡単であるかを理解していただくためです。今回の例では、ウィザードから「Web」、「JPA」(Java Persistence API)、「REST Repositories」、「Vaadin」(UIライブラリ)、「Actuator」(Spring BootのWebサービス・コンポーネント)、「Remote Shell」、「H2」(データベース)、「Thymeleaf」(テンプレート・エンジン)を選択します(図1)。「Generate Project」をクリックすると、お好みのIDEで開くことができる

使用の開始

Spring Bootでは、さまざまな方法で開発を始めることができます。筆者はSpring Initializrをよく使用します。Spring Initializrは、直接、またはお好みのIDE (NetBeans、IntelliJ IDEA 14 Ultimate、またはEclipse ディストリビューションであるSpring Tool Suite) のウィザードから使用するWebサービスです。いずれの方法でも、Spring Initializrで表示される、複数のオプションを選択できるメニューから、ワークロードの種類や使用するテクノロジーを選択します。パッケージの種類は、JARファイルとWARファイルから選択できます。また、言語はJavaとGroovyから選択でき、使用するJVMのバージョンも1.6から1.8の間で指定でき

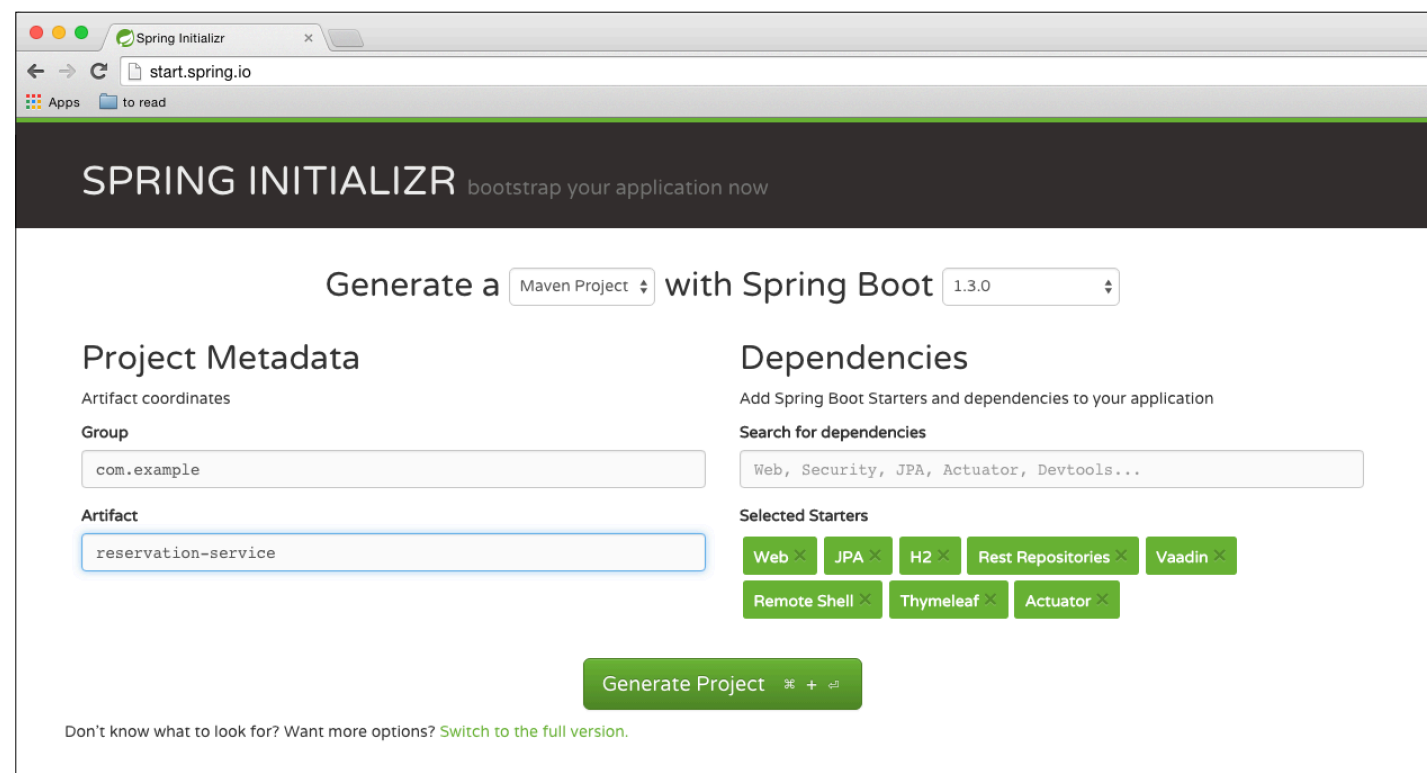


図1.プロジェクト・オプションの選択



zipファイルをダウンロードできます。

プロジェクトを開くと、中に単純なpublic static void mainメソッドと空の単体テストがあることがわかるでしょう。さらに、src/main/resourcesディレクトリの中には、staticとtemplatesという2つの空のディレクトリがあります。サーバー・サイドで処理されるテンプレート(FreeMarker、Velocity、Thymeleafなど)はtemplatesディレクトリに、クライアントに直接送信される資産(JavaScript、画像、CSSなど)はstaticディレクトリに格納されます。図2にディレクトリ構成を示します。この図には、サンプルのThymeleafテンプレート(reservations.html)と、何の処理も行わずにHTTPクライアントに送信する単純なhi.htmlファイルが含まれています。

まず、Mavenビルド・ファイルpom.xmlを開きます。Spring Initializrで選択した内容に応じた依存性が含まれていることがわかります。さまざまなstarter-*という依存性がありますが、独特の依存性であり、コードは含まれていません。別の依存性を取り込むだけです。たとえば、org.springframework.boot:spring-boot-starter-data-jpaは、JPAを使用するために必要となる可能性があるすべてのものを取り込みます。具体的には、もっとも新しく高機能なJPA 2仕様の型、最新のHibernateの実装、SpringがサポートするJPA、ORM、JDBCなどが取り込まれます。

バージョン範囲の管理や、共通ライブラリへの依存性の列挙を行う必要はありません。Mavenには、別のビルド・アーティファクトのビルド構成をインポートまたは継承を行う機能があります。Springは、この機能を活用して、バージョン範囲や、Servlet APIやHibernateの依存性などの必要な項目を定義している有用な親ビルドを公開しています。この手法には、2つのメリットがあります。1つ目は、事前に定義されている依存性のバージョン宣言を省略できることです。2つ目は、新しいバージョンが登場した際に、使用しているバージョンの親ビルドを単純に上書きすることによって、最新バージョンのSpring Bootにアップデートできることです。関連付けられている依存性はすべて、所定の方法で自動的にアップグレードされます。

データは語る

それでは、単純なJPAエンティティReservation.javaを作成してみましょう。サンプル・アプリケーションでは、このエンティティを操作してゆきます。

```
package com.example;
```

```
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
@Entity  
public class Reservation {
```

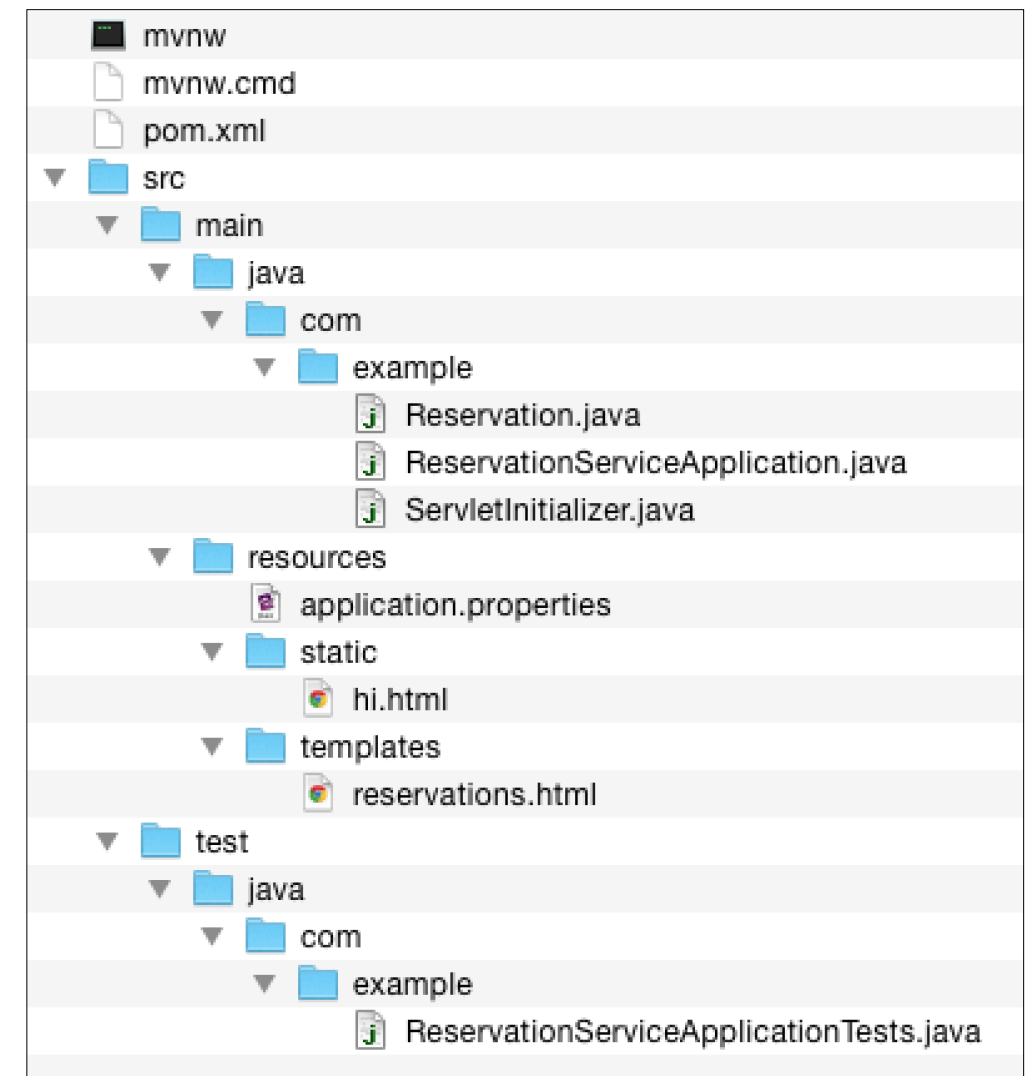


図2.いくつかのファイルを追加したプロジェクト・ディレクトリ



```

@Id
@GeneratedValue
private Long id;

private String reservationName;

public Reservation(String reservationName) {
    this.reservationName = reservationName;
}

Reservation() {
}

@Override
public String toString() {
    return "Reservation{" + "id=" + id +
        ", reservationName='" +
        reservationName + '\'' +
        '}';
}

public Long getId() {
    return id;
}

public String getReservationName() {
    return reservationName;
}
}

```

このReservationエンティティは単純なものです、JPAの持つ力を完全に活用できることを実証できます。ここでは、上位プロジェクトであるSpring DataプロジェクトのSpring Data JPAモジュールを使用します。このモジュールは、Oracle Database、MongoDB、Neo4j、Cassandraなどの他の永続化テクノロジーも同様にサポートしています。このモジュールを使用すると、次のようなインタフェース・メソッドの命名規則に基づいてリポジトリ・オブジェクトを宣言的に定義できます。

```
interface ReservationRepository extends
```

```

JpaRepository<Reservation, Long> {

    Collection<Reservation>
        findByReservationName(String rn);
}

```

このことを考慮した上で、@Beanプロバイダ・メソッドをReservationServiceApplicationクラスに定義します。ReservationServiceApplicationクラスには、@SpringBootApplicationアノテーションが付加されています。そのため、このクラスは@Beanプロバイダ・メソッドを配置できる@Configurationクラスでもあります。@Configurationクラスでは、プロバイダ・メソッドの戻り値として定義済みのBeanを使うことができるため、コンポーネント自身に@Component、@RestControllerなどの定型的なアノテーションを付加する必要がなくなります。返されるオブジェクトはCommandLineRunner型で、Spring Bootアプリケーションの起動時に実行されます。そのため、デモ用のサンプル・データを挿入するのは、この場所が最適です。

```

@Bean
CommandLineRunner runner(ReservationRepository rr) {
    return args -> Stream.of(
        "Julia", "Mia", "Phil", "Dave", "Pieter",
        "Bridget", "Stéphane", "Josh", "Jennifer")
        .forEach(n -> rr.save(new Reservation(n)));
}

```

Web

次に、Spring MVCを使用してREST APIを作成します(Spring Initializrで選択することで、Jersey JAX-RSの実装やRatpackを作成することもできます)。コードは、次のとおりです。

```

@RestController
class ReservationRestController {

    private final ReservationRepository
        reservationRepository;
}

```



08

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title> Reservations </title>
</head>
<body>
<H1> Reservations </H1>
<div th:each="r :${reservations}">
    <div>
        <b th:text="${r.id}">ID</b> -
        <span th:text="${r.reservationName}">
            Reservation Name
        </span>
    </div>
</div>
</body>
</html>
```

データ駆動型のユーザー・インタフェースを短時間で構築しようとしている場合は、高評価のオープンソースの[Vaadin Framework](#)など、UIコンポーネントベースのアプローチを使用することもできます。今回の例では、最初にSpring InitializrでVaadinも選択しました。そのため、Spring BootとVaadinを簡単に連携させるためのサード・パーティによる依存性が追加されています。Vaadinは、[GWT](#)上に構築されており、UIコンポーネントは、高速で応答性がよく、トランスコンパイルされたクライアントサイドのJavaScriptで構成されています。しかし、Vaadinアプリケーションでは、ビジネス・ステータスはサーバー上にあります。次に示すのは、Vaadinコンポーネントを使って[Reservation](#)データを表示する単純なデータ・グリッドです。




```

@SpringUI(path = "ui")
@Theme("valo")
class ReservationUI extends UI {

    private final ReservationRepository
        reservationRepository;

    @Autowired
    public ReservationUI(ReservationRepository
        reservationRepository) {
        this.reservationRepository =
            reservationRepository;
    }

    @Override
    protected void init(VaadinRequest request) {
        Grid table = new Grid();
        BeanItemContainer<Reservation> container =
            new BeanItemContainer<>(
                Reservation.class,
                this.reservationRepository.findAll());
        table.setContainerDataSource(container);
        table.setSizeFull();
        setContent(table);
    }
}

```

これで完了です。http://localhost:8080/uiを開くと、アプリケーションが実行され、図5の画面が表示されます。

テスト

通常は、アプリケーションのすべての部分に対してテストを行う必要があります。Spring MVCテスト・フレームワークには、柔軟なドメイン固有言語（DSL）が組み込まれているため、HTTPエンドポイントを非常に簡単に実行できます。

```

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes =
    ReservationServiceApplication.class)
@WebAppConfiguration
public class ReservationServiceApplicationTests {

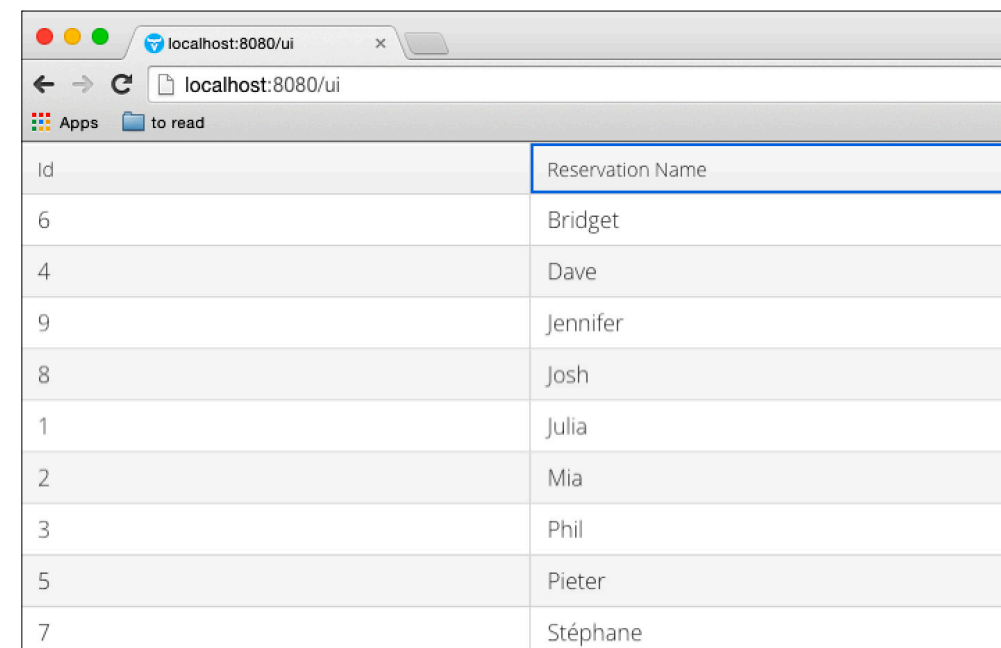
    private MediaType mediaType =
        MediaType.parseMediaType(
            "application/hal+json");

    private MockMvc mockMvc;

    @Autowired
    private WebApplicationContext
        webApplicationContext;

    @Before
    public void before() throws Exception {
        this.mockMvc =
            MockMvcBuilders.webAppContextSetup(
                this.webApplicationContext).build();
    }
}

```



Id	Reservation Name
6	Bridget
4	Dave
9	Jennifer
8	Josh
1	Julia
2	Mia
3	Phil
5	Pieter
7	Stéphane

図5.UIライブラリを使用したアプリケーション



```

@Test
public void contextLoads()
    throws Exception {
    this.mockMvc.perform(
        get("/reservations")
        .accept(this.mediaType))
        .andExpect(content().
            contentType(this.mediaType))
        .andExpect(status().isOk());
    }
}

```

このコードでは、JUnitやTestNGと連携してカスタム・テストを実行するSpring MVCテスト・フレームワークを使用しています。このテスト・フレームワークは、疑似統合テストに理想的です。ここでは、このフレームワークを使って、まったくソケットに接続せずにサンプルのSpring Bootアプリケーションを立ち上げてみます。Spring MVCテスト・フレームワークは、Spring MVCのすべての機能を使用して呼び出しを行い、実際にすべてのWebコンポーネントを実行しますが、実際に着信するHTTPのTCPパケットをリスンするソケットを使用せずにこの処理を行っています。つまり、フレームワークの機能やリクエストの処理を担当するコンポーネントをテストするためだけにコンテナを立ち上げる必要はないということです。

本番環境

だいぶ面白いアプリケーションになってきました。しかし、面白いだけでは不十分です。ソフトウェアは、デプロイしなければなりません。Michael Nygard氏の素晴らしい著書『[Release It!](#)』を読んだことがあるなら、「コードの完成」と「稼働準備完了」の間の最後の一步は、大変長い道のりになる可能性があることをご存知でしょう。たくさんの組織の担当者から話を聞けば、アプリケーションのコードを本番環境に移す前に満たす必要がある非機能要件、それもどこかのWikiページに埋もれているような要件がうんざりするほどあることがわかるはずです。

こういった話は、まったく面白いものではなく、差別化要因でもありません。アプリケーションの環境はどんなものでしょうか。成長指標をどのように取得しますか。アプリケーションは、健全性を示すことができるエンドポイントを提供しているでしょうか。どのバージョンのサービスが稼働

しているかをどのように見分けるでしょうか。トラフィックが多いのは、どのHTTPリソースでしょうか。こういったことをはじめとするさまざまな問いには、Spring Boot Actuatorが自動的に作成するエンドポイントが回答してくれます。Spring Boot ActuatorはSpring Bootのモジュールのひとつで、本番環境のアプリケーションの操作をサポートするために設計されています。Spring Boot Actuatorは、一般的にコードを本番環境に移す際に制約となるさまざまな非機能要件の軽減または削除に役立ちます。この仕組みによって、/health、/beans、/trace、/mappings、/metrics、/envなどのさまざまな管理エンドポイントが追加されます。こういった便利なエンドポイントは、簡単にカスタマイズすることもできます。ここでは、application.propertiesファイルに次のプロパティを指定し、管理エンドポイントの接頭辞を変更してみます。Spring Bootは、設定キーとその値をこのファイルとapplication.ymlから探します。

management.context-path=/admin

このプロパティが設定されていると、ユーザーは/admin/health、/admin/envなどにアクセスする必要があります。CLASSPATHにspring-boot-starter-securityを追加すると、デフォルトのHTTP BASIC認証のプロンプトが表示されるようになります。コンソールには、デフォルトのユーザー名とパスワードが出力されます。適切なSpring Securityオプションを設定する代わりに、HTTP BASIC認証のチャレンジをSpring Security AuthenticationProviderに委譲することもできます。

システムの健全性を計測するhealthエンドポイントも興味深いものです。このエンドポイントは自動的に、稼働中のテクノロジーについて最大限の情報を提供してくれます。たとえば、ファイル・システム、DataSourceコネクション・プール、JavaMail、JMS、Cassandra、Elasticsearch、Solr、MongoDB、RabbitMQ、Redisなどのテクノロジーについての情報が得られます。必要に応じて、このデータをHealthIndicatorセマンティックに追加することもできます。



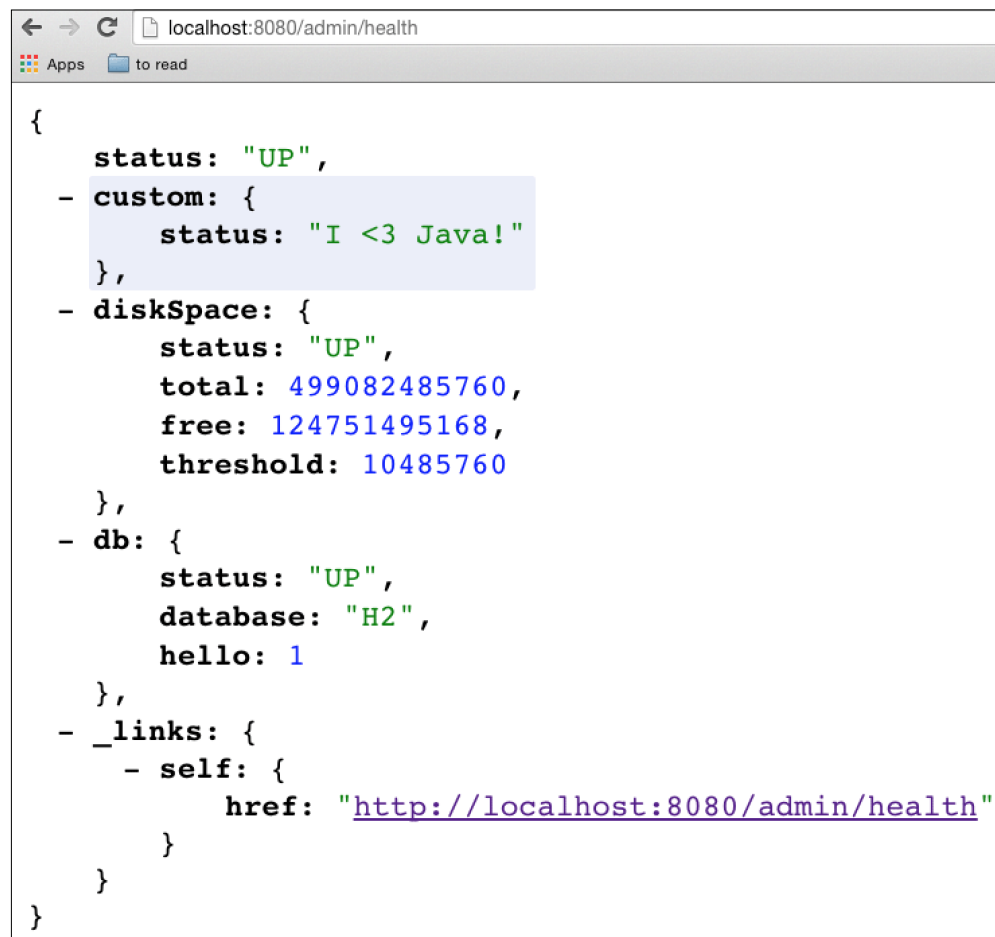

```

@Bean
HealthIndicator customHealthIndicator() {
    return new HealthIndicator() {
        @Override
        public Health health() {
            return Health.status("I <3 Java!")
                .build();
        }
    };
}

```

このコードで取得できるデータを図6に示します。

アプリケーションは、リクエストされたHTTPリソース数、使用可能なメモリ、ロードされたクラス数など、一般的な指標を自動的に収集します。さらに、アプリケーションのどのコンポーネントでも、CounterServiceをインジェクションしてセマンティック指標を生成できます。[DropWizard Metrics](#)



```

{
  status: "UP",
  custom: {
    status: "I <3 Java!"
  },
  diskSpace: {
    status: "UP",
    total: 499082485760,
    free: 124751495168,
    threshold: 10485760
  },
  db: {
    status: "UP",
    database: "H2",
    hello: 1
  },
  _links: {
    self: {
      href: "http://localhost:8080/admin/health"
    }
  }
}

```

図6.アプリケーションの健全性監視データ

という優れたライブラリのMetricReporterを設定するか、あるいはSpring Bootに組み込まれているいずれかのMetricWriterの実装をそのまま使用すれば、Graphite、StatsD、OpenTSDBなどの単一の統合ダッシュボードで、アプリケーションの指標を収集してグラフ化することができます。

同様に、アプリケーションの実行も簡単です。Spring InitializrのPackagingで「Jar」を選択すると、自己完結型のJARベースのデプロイを使用できます。または、「War」を選択して、Apache Tomcat/TomEE、Oracle WebLogicまたはGlassFish、Payara Micro、IBM WebSphere、Red HatのWildFlyアプリケーション・サーバーなどの任意のサーブレット・コンテナにデプロイすることもできます。CloudFoundry、Heroku、Google App Engine、OpenShiftなどの最新クラウド環境でも、Spring Bootアプリケーションを実行できます。

まとめ

この短いチュートリアルは、表面的な点をわずかに説明したものにすぎません。Spring Bootを使うと、JAX-RS、JMS、Servlet API、JTA、JDBCなどの多くのJava EE技術をはじめとしたさまざまな技術だけでなく、Springエコシステムのあらゆるコンポーネントも統合できます。また、Spring Bootは、マイクロサービスベースの分散システムの作成をサポートするために設計された一連のコンポーネントであるSpring Cloudの基盤にもなっています。Spring Bootは、一般的なタスク向けのデフォルト設定を活用することによって開発を進めることができるという独特な手法です。そのような設定の多くは、それぞれの分野の先頭に立つ業界のエキスパートが定義、提供しています。まずは、[Spring Initializr](#)をチェックしてみてください。

</article>

Josh Long (@starbuxman) :PivotalのSpringデベロッパー・アドボケート。Java Champion。プログラミングに関する5冊の書籍(近日発刊予定の『Cloud Native Java』を含む)と3つのベストセラー・ビデオ・トレーニングの著者で、長年にわたってオープンソースにも貢献。





ABHISHEK GUPTA

JAX-RS 2.0: 良いものはすべて利用する

新規クライアントAPI、フィルタ、インターセプタ、その他の有用なREST機能

JAX-RSフレームワークは、RESTfulの原理を実装するアプリケーションやサービスを構築するためのAPIセットです。Plain Old Java Object (POJO) をアノテーションによって修飾して、どこでも使用できるWebプロトコルであるHTTP経由でサービスやビジネス・ロジックの一部を公開できる、単純でありながらも強力なプログラミング・モデルを提供します。

JAX-RSは2008年、JCP仕様のJSR 311リリース1.0として標準化されました。また、Java EE 6仕様(JSR 316)の一部にもなりました。2013年にJAX-RS 2.0(JSR 339)がリリースされました。JAX-RS 2.0は、Java EE 7仕様(JSR 342)の不可欠な要素です。次のエディションであるJAX-RS 2.1(JSR 370)の作業が現在進行中であり、Java EE 8プラットフォームの一部としてリリースされる予定です。

広く普及しているJAX-RS実装として、Jersey (リファレンス実装)、RESTEasy、Apache CXFなどがあります。これらの実装ではいずれも、基本のJAX-RS仕様に準拠し、仕様をサポートしながら、さらに便利な機能も追加されています。

本記事では、JAX-RS 2.0の最新機能に焦点を当てます。JAX-RS 2.0の最新機能には、新規クライアントAPI、フィルタ、インターセプタ、新しい例外処理方法、非同期プログラミングの拡張機能、その他多くの追加機能があります。

新規クライアントAPI

本格的なクライアントAPIが追加される前は、HTTP指向の (REST) サービスと通信するために、開発者はサード・パーティの実装に頼るか、JDKのURLConnection APIを操作する必要がありました。新規クライアントAPI (javax.ws.rs.client/パッケージ内) はかなりコンパクトで無駄がなく、流れるようなAPIです。その一部のクラスとインタ

フェースについて紹介します。

ClientBuilderは、呼出しプロセスを開始するためのクラスであり、buildメソッドや、オーバーロードされたnewClientメソッドがエントリ・ポイントとなっています。Clientは、オーバーロードされたtargetメソッドによってWebTargetインスタンスを作成するためのインタフェースです。WebTargetは、HTTPリクエスト呼出し用のURIエンドポイントを表したものです。WebTargetを使用して、クエリー、マトリックス、パス・パラメータなどのさまざまな属性を設定できます。WebTargetは、オーバーロードされたrequestメソッドを公開しており、これらのメソッドによってInvocation.Builderのインスタンスを取得します。Invocation.Builderには、さらにHTTPリクエストを構成して、ヘッダー、Cookie、キャッシュ管理などの属性や、メディア・タイプ、言語、エンコーディングなどのコンテンツ・ネゴシエーション関連のパラメータを設定する機能があります。加えて、ClientのbuildXXXメソッドのいずれかを使用して、Invocationオブジェク

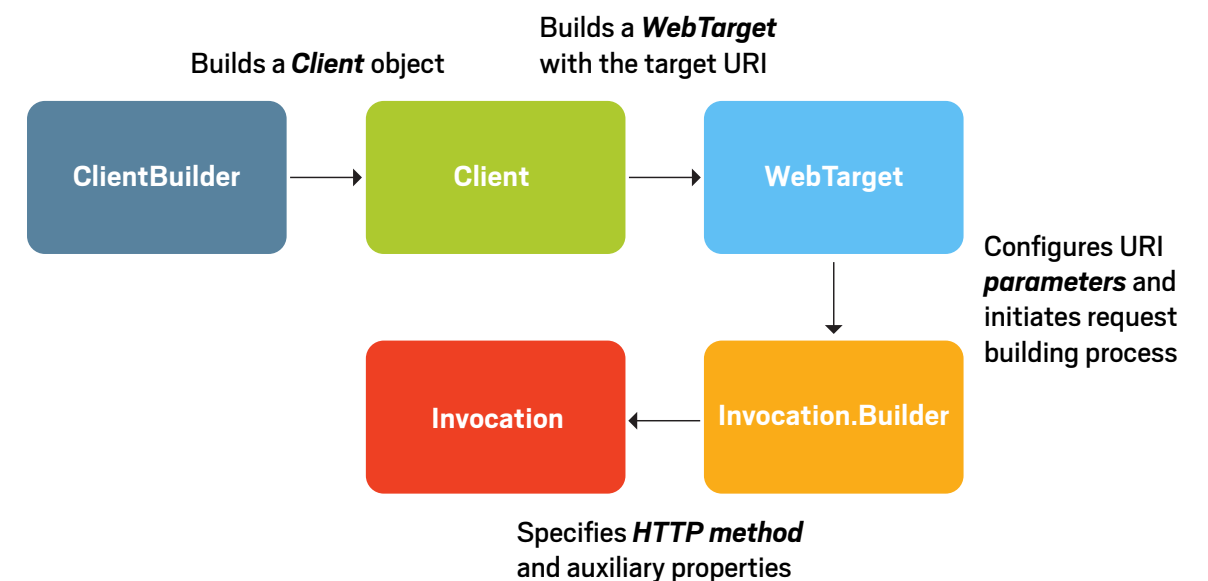


図1: JAX-RS 2.xクライアント



トのインスタンスを取得します。InvocationのインスタンスはHTTPリクエストをカプセル化したものであり、オーバーロードされたinvokeメソッドを使用して同期リクエストを、submitメソッドを使用して非同期リクエストをそれぞれ送信できます。図1に、この一連の流れを示します。

例を見てみましょう。

```
Client client = ClientBuilder.newClient();
WebTarget webTarget =
    client.target("http://service.com/user")
        .queryParams("card", "4275391126915480");
Invocation.Builder builder =
    webTarget.request("text/plain");
Invocation invocation =
    builder.header("testheader", "testvalue")
        .buildGet();
Response response = invocation.invoke();
```

次に、このコードを分析し、処理の流れを詳しく理解しましょう。

まず、ClientBuilderクラス経由で取得した、Clientのインスタンスを使用してターゲットURIを指定しています。その結果、WebTargetのインスタンスが作成されます。さらに、作成されたインスタンスを使用して、期待されるレスポンス/メディア・タイプ(HTTPヘッダーのAcceptに相当するもの)および関連するURIパラメータ(パスとクエリー)を指定しています。以上で、Invocation.Builderのインスタンスが作成されます。このインスタンスを使用して、完全なHTTP GETリクエストを構成します。その後、Invocationインスタンスを使用して、リクエストをサーバーに送信しています。

Configurableインタフェース: Client、ClientBuilder、WebTarget、Invocationの各オブジェクトはjavax.ws.rs.core.Configurableインタフェースを実装しています。そのため、フィルタ、インターセプタ、エンティティ・プロバイダ(メッセージ・リーダー/ライター)などのカスタムのJAX-RSコンポーネントを定義することが可能です。定義するためには、オーバーロードされたregisterメソッドを使用します。定義方法については、後ほど簡単に説明します。なお、このAPIは、サーバー・サイドのJAX-RSコンポーネント(フィルタ、インターセプタなど)にも適用されます。

フィルタ

フィルタと、後述のインターセプタもJAX-RS 2.0の目玉機能です。JAX-RSアプリケーション内でアスペクト指向プログラミング(AOP)のようなものが可能になり、開発者がアプリケーション固有の横断的機能(ビジネス・ロジック全体に分散させるべきでない機能)を実装できるようになります。そのような横断的機能には、認証、認可、リクエスト/レスポンスの検証、ロギングなどが含まれます。このAOPベースのプログラミング・モデルでは、JAX-RSリソース・クラスのメソッドに割り込んで、HTTPリクエスト/レスポンス・ヘッダーの各要素、リクエストURI、呼び出されたHTTPメソッド(GET、POSTなど)を操作(変更)します。

サーバー・サイドのリクエスト・フィルタ: サーバー・サイドのリクエスト・フィルタは、クライアントから受信するHTTPリクエストに対して動作します。リクエスト・フィルタは、JAX-RSリソース・メソッドの呼出しよりも前に呼び出されます。そのため、受信したHTTPリクエストの特定の属性を操作することができます。

サーバー・サイドのリクエスト・フィルタを実装するためには、JAX-RSによって提供される拡張機能であるjavax.ws.rs.container.ContainerRequestFilterインタフェースを実装します。ContainerRequestFilterのfilterメソッドに、javax.ws.rs.container.ContainerRequestContextインタフェースのインスタンスがコンテナによってシームレスにインジェクションされます。ContainerRequestContextは、HTTPリクエスト・コンポーネントへのアクセス用メソッドと変更用メソッドを公開しており、可変オブジェクト(意図的なものです)になっています。

サーバー・サイドのレスポンス・フィルタは、機能(レスポンスのHTTPヘッダーなどを読み取り、変更する)とプログラミング・モデル(たとえば、ユーザーが定義した順序またはデフォルトの順序で連鎖的に実行される)の両面で、対応するリクエスト・フィルタに似ています。



```
public interface ContainerRequestFilter{
    public void filter(
        ContainerRequestContext reqCtx) throws
        IOException;
}
```

JAX-RSのリクエスト処理パイプラインには、JAX-RSプロバイダによって実装されているマッチング・アルゴリズムに基づいて、HTTPリクエストをリソース・クラス内の適切なJavaメソッドにディスパッチする処理が含まれます。サーバー・サイドのリクエスト・フィルタではこの処理が考慮されており、マッチング前とマッチング後の2つのカテゴリに分けられています。

マッチング前フィルタは、受信したHTTPリクエストが特定のJavaメソッドにマッピング/ディスパッチされる前に実行されます。マッチング前フィルタは、クラスに対してjavax.ws.rs.container.PreMatchingアノテーションを付加することで容易に設定できます。なお、以下のコードでの実装クラスに対するjavax.ws.rs.ext.Providerアノテーションは、JAX-RSランタイムでこのフィルタがJAX-RSフィルタとして認識されるために必要になります。

```
@Provider
@PreMatching
public class PreMatchingAuthFilter{
    public void filter(
        ContainerRequestContext reqCtx)
        throws IOException {
        if(reqCtx.getHeaderString(
            "Authorization") == null){
            reqCtx.abortWith(
                Response.status(403).build());
        }else{
            //check credentials....
        }
    }
}
```

マッチング後フィルタは、メソッドのディスパッチまたはマッチング・プロセスの完了後にのみ、JAX-RSコンテナによって実行されます。マッチング前フィルタとは異なり、マッチング後フィルタでは明示的なアノテーションは必要ありません。そのため、@PreMatchingアノテーションのないフィルタ・クラスは、デフォルトでマッチング後フィルタとみなされます。以下のコードに使用例を示します。

```
@Provider
public class PostMatchingFilterExample{
    public void filter(ContainerRequestContext
        reqCtx)
        throws IOException{
        System.out.println(
            "Referrer: " +
            reqCtx.getHeaderString("referrer"));
        System.out.println(
            "Base URI: " +
            reqCtx.getUriInfo().getBaseUri());
        System.out.println(
            "HTTP Request method: " +
            reqCtx.getMethod());
    }
}
```

1つのJAX-RSアプリケーションに複数のフィルタを設定できます(フィルタは連鎖構造を形成します)。複数のフィルタは、開発者が定義した順序(後述)か、コンテナにより設定されたデフォルトの順序で実行されます。ただし、フィルタの実装ロジックから例外をスローするかabortWithメソッドを呼び出すことで、この処理の連鎖を断つことができます。いずれのケースでも、連鎖内の後続のフィルタは呼び出されません。

サーバー・サイドのレスポンス・フィルタ:サーバー・サイドのレスポンス・フィルタは、JAX-RSのリソース・メソッドによってレスポンスが生成されてから、そのレスポンスが呼出し元/クライアントにディスパッチされるまでの間に呼び出されます。レスポンス・フィルタは、機能(レスポンスのHTTPヘッダーなどを読み取り、変更する)とプログラミング・モデル(たとえば、開発者が定義した順序またはデフォルトの順序で連鎖的に実行される)の両面で、対応するリクエスト・フィルタに似ています。

サーバー・サイドのレスポンス・フィルタは、javax.ws.rs.container.



ContainerResponseFilterインタフェースを実装するだけで簡単に設定できます。サーバー・サイドのリクエスト・フィルタと同様に、ContainerResponseFilterインタフェースのfilterメソッドにContainerResponseContextをインジェクションする処理が、JAX-RSランタイムによって実行されます。また、JAX-RSランタイムで認識されるように、サーバー・サイドのレスポンス・フィルタにもjavax.ws.rs.ext.Providerアノテーションを付加する必要があります。以下に例を示します。

```
@Provider
public class AContainerResponseFilter{
    public void filter(
        ContainerRequestContext reqCtx,
        ContainerResponseContext resCtx)
        throws IOException{
        //adding a custom header to the response
        resCtx
            .getHeaders()
            .add("X-Search-ID",
                "qwer1234-tyuio5678-asdfg9876");
    }
}
```

ご想像のとおり、クライアント・サイドのフィルタにも、リクエスト・フィルタとレスポンス・フィルタがあります。これらのフィルタについて以下にまとめます。

クライアント・サイドのリクエスト・フィルタ: クライアント・サイドのリクエスト・フィルタは、HTTPリクエストが作成されてから、そのリクエストがサーバーにディスパッチされるまでの間に呼び出されます。このタイプのフィルタは、HTTPリクエストのプロパティ（ヘッダー、Cookieなど）を変更する機会を提供するものです。クライアント・サイドのリクエスト・フィルタを実装するためには、javax.ws.rs.client.ClientRequestFilterによって提供される拡張インタフェースを実装します。

クライアント・サイドのレスポンス・フィルタ: クライアント・サイドのレスポンス・フィルタは、HTTPレスポンスをサーバーから受信してから、呼出し元/クライアントでの処理用にディスパッチされるまでの間に呼び出されます。クライアント・サイドのリクエスト・フィルタと同様に、レスポンス・フィルタは、HTTPレスポンスのプロパティを変更する機会を提供します。

クライアント・サイドのレスポンス・フィルタを使用するためには、以下に示すように、javax.ws.rs.client.ClientResponseFilterによって提供される拡張インタフェースを実装します。

```
public class ClientResponseLoggerFilter {
    public void filter(
        ClientRequestContext reqCtx,
        ClientResponseContext resCtx)
        throws IOException{
        System.out.println(
            "Response status: " +
            resCtx.getStatus());
    }
}
```

インターセプタ

インターセプタは、HTTPリクエストやHTTPレスポンスを変更する目的でも使用される点ではフィルタに似ていますが、大きく異なる点として、インターセプタはおもにHTTPメッセージのペイロード部を操作するために使用されることが挙げられます。インターセプタは、javax.ws.rs.ext.ReaderInterceptorとjavax.ws.rs.ext.WriterInterceptorの2つのカテゴリに分けられます。前者はHTTPリクエストを処理するために、後者はHTTPレスポンスを処理するために使用されます。なお、フィルタとは異なり、サーバー・サイドでもクライアント・サイドでも同じインターセプタのセットが適用できます。

JAX-RSランタイムで検出されるように、サーバー・サイドのインターセプタには@Providerアノテーションを付加する必要があります。また、クライアント・サイドのインターセプタは、ClientBuilder、Client、WebTargetのクラスまたはインタフェースによって登録する必要があります（registerメソッドを使用します）。



ReaderInterceptor:このタイプのインターセプタは、JAX-RS APIによって提供される契約(拡張インタフェース)です。サーバー・サイドのリーダー・インターセプタはクライアントから送信されたHTTPペイロードに対して操作を行います。それに対して、クライアント・サイドのリーダー・インターセプタは、リクエストのペイロードがサーバー・サイドに送信される前に、そのペイロードに対して操作(読取りや変更)を行うことが想定されています。

WriterInterceptor:このタイプのインターセプタは、サーバー・サイドではリソース・メソッドによって生成されたHTTPペイロードに対して操作を行います。それに対して、クライアント・サイドのライター・インターセプタは、サーバーによって送信されたペイロードが呼出し元にディスパッチされる前に、そのペイロードに対して操作(読取りや変更)を行うことが想定されています。以下のコードに、サーバー・サイドのインターセプタの例を示します。

```
public interface WriterInterceptor{
    public void aroundWriteFrom(
        WriterInterceptorContext writerCtx)
        throws IOException,
        WebApplicationException;
}
```

JAX-RSのインターセプタは、フィルタと同様に連鎖的に呼び出されます。また、HTTPメッセージとJavaオブジェクト表現の間で相互に変換するためにエンティティ・プロバイダ(javax.ws.rs.MessageBodyReaderおよびjavax.ws.rs.MessageBodyWriter)が必要になったときに初めてトリガーされます。ReaderInterceptorはMessageBodyReaderを、WriterInterceptorはMessageBodyWriterをそれぞれラップします。したがって、インターセプタとエンティティ・プロバイダは同じコール・スタック内で実行されます。

JAX-RS 2.0には、サーバー・サイドおよびそれに対応するクライアント・サイドの非同期処理を行うための新しいAPIが含まれています。そもそも非同期とは、リクエストを開始したスレッドとは異なるスレッドでリクエストが処理されることを意味します。

フィルタおよびインターセプタのバインディング戦略

JAX-RSでは、フィルタおよびインターセプタをそれぞれのターゲット・コンポーネントにバインディングするための複数の方法を定義しています。サーバー・サイドのフィルタの場合、グローバルなデフォルトの振る舞いに加え、名前付きバインディングおよび動的バインディングも存在します。それぞれの方法について手短かに確認しましょう。

グローバルなデフォルトの振る舞い:デフォルトでは、JAX-RSのフィルタおよびインターセプタは、アプリケーション内のリソース・クラスのすべてのメソッドにバインディングされます。そのため、クライアントからのHTTPリクエストへの応答としてリソースのメソッドが呼び出されたときは常に、リクエスト・フィルタ(マッチング前およびマッチング後)とレスポンス・フィルタの両方が呼び出されます。この規約は、名前付きバインディングまたは動的バインディングを使用することでオーバーライドすることが可能です。

名前付きバインディング:フィルタおよびインターセプタのスコープを詳細に(リソース・クラスごと、あるいはメソッドごとに)設定するために、@NameBindingアノテーションを利用できます。そのためには、以下の手順を実行します。

- **手順1:**@NameBindingアノテーションを使用してカスタムのアノテーションを定義します。

```
@NameBinding
@Target({ ElementType.TYPE,
           ElementType.METHOD })
@Retention(value =
    RetentionPolicy.RUNTIME)
public @interface Audited { }
```

- **手順2:**定義したカスタムのアノテーションをフィルタまたはインターセプタに適用します。

```
@Provider
@Audited
public class AuditFilter implements
    ContainerRequestFilter {
    //filter implementation...
}
```



- **手順3:** 同じアノテーションを、必要なリソース・クラスまたはリソース・メソッドに適用します。クラスに適用した場合、フィルタまたはインターセプタは、そのクラスのすべてのリソース・メソッドにバインディングされます。

```
@GET
@Path("/{id}")
@Produces("application/json")
@Audited
public Response find(
    @PathParam("id") String custId){
    //search and return customer info
}
```

動的バインディング: JAX-RSは、フィルタおよびインターセプタを実行時に動的にバインディングするためのDynamicFeatureインタフェースを提供しています(動的バインディングは、前項で説明した@NameBindingアノテーションを使用して実行できる、より静的なバインディングと併用することも可能です)。

```
public interface DynamicFeature {
    public void configure(
        ResourceInfo resInfo,
        FeatureContext ctx);
}
```

インジェクションされたResourceInfoインタフェースのインスタンスを使用して各種メソッドを公開し、リソース・メソッドを動的に選択できます。また、FeatureContextインタフェースは、リソース・メソッドの選択後にフィルタまたはインターセプタを登録するために使用できます。以下のコードにその例を示します。

```
@Provider
public class DynamicAuthFilterFeature
    implements DynamicFeature {
    @Override
    public void configure(
        ResourceInfo resInfo, FeatureContext ctx) {
        if (UserResource.class
```

```
.equals(resInfo.getResourceClass()) &&
resInfo.getResourceMethod().getName()
.contains("PUT")) {
    ctx.register(
        AuthenticationFilter.class);
}
}
```

クライアント・サイドのフィルタおよびインターセプタのバインディングと登録

サーバー・サイドのフィルタでは、クラス・レベルの@Providerアノテーションが必要であることはすでに確認したとおりです。クライアント・サイドでは、フィルタおよびインターセプタはClientBuilder、Client、またはWebTargetのいずれかのインタフェースを使用して登録します。これらのインタフェースはすべて、javax.ws.rs.core.Configurableインタフェースを実装します。Configurableインタフェースには、オーバーロードされた複数のregisterメソッドがあります。

フィルタおよびインターセプタの順序の設定

@Priorityアノテーションを使用して、フィルタおよびインターセプタの実行順序を定義できます。このアノテーションには数値を指定できます。この数値の解釈は、フィルタおよびインターセプタがリクエスト用かレスポンス用かによって異なります。

リクエスト・フィルタ(ContainerRequestFilterとClientRequestFilter)およびリクエスト・インターセプタ(ReaderInterceptorとWriterInterceptor)は、優先度の数値の昇順に実行されます。つまり、@Priorityアノテーションが付加されたプロバイダのうち、その値の小さいものから先に実行されます。

レスポンス・フィルタ(ContainerResponseFilterとClientResponseFilter)は、その逆の順序(降順)で実行されます。@Priorityアノテーションがない場合は、デフォルト値が適用されます。デフォルト値は、javax.ws.rs.PrioritiesのUSER定数に定義されています(この定数の値は5,000です)。クライアント・サイドのコンポーネントの優先度は、Configurableを実装するインタフェースのregisterメソッドを使用して設定できます。



非同期処理のサポート

JAX-RS 2.0には、サーバー・サイドおよびそれに対応するクライアント・サイドの非同期処理を行うための新しいAPIが含まれています。そもそも非同期とは、リクエストを開始したスレッドとは異なるスレッドでリクエストが処理されることを意味します。クライアントの視点で言えば、非同期処理を行った場合、リクエスト・スレッドがブロックされなくなります。サーバーからのレスポンス待ちの時間が発生しないからです。同様に、サーバー・サイドの非同期処理では、元のリクエスト・スレッドを一時的に停止し、別のスレッドでリクエストの処理を開始します。そのため、元のサーバー・サイドのスレッドが解放され、別のリクエストを受信できるようになります。非同期処理を適切に利用すれば、スケーラビリティ、応答性、スループットが向上します。

サーバー・サイドの非同期処理:サーバー・サイドの非同期プログラミング・モデルは、おもに@javax.ws.rs.container.Suspendedアノテーションとjavax.ws.rs.container.AsyncResponseインタフェースの2つのAPI抽象表現によって機能します。

JAX-RSのリソース・クラスのメソッド・パラメータに@Suspendedアノテーションを付加することで、そのメソッド・パラメータにAsyncResponseのインスタンスを透過的にインジェクションできます。このインスタンスは、呼出し元/クライアントと通信して、レスポンス送信(リクエスト後の処理の完了)、リクエストのキャンセル、エラーの伝播などの操作を実行するためのコールバック・オブジェクトの役割を担います。

```
@GET
@Path("/{id}")
public void search(
    @Suspended AsyncResponse asyncResp,
    @PathParam("id") String id){
    //launching search in a new thread
    new Thread(){
        public void run(){
            //execute search op and resume
            UserInfo user = //obtain via search...
            asyncResp.resume(user);
        }
    }.start();
}
```

レスポンスや例外を伝播させるためには、AsyncResponseインタフェースのオーバーロードされたresumeメソッドを使用して、レスポンスや例外をクライアントに返します。

リクエスト処理のタイムアウトを処理するためには、HTTP 503のエラー・レスポンスをクライアントに返すまでのタイムアウトを設定します。タイムアウトのしきい値を設定することで、その設定が可能です。デフォルトでは、タイムアウトによってHTTP 503のエラー・レスポンスが発行されますが、javax.ws.rs.container.TimeoutHandler実装を登録することで、この動作をオーバーライドできます。

javax.ws.rs.container.CompletionCallbackは、コールバック・インタフェースを表します。このインタフェースの実装を登録することで、リクエストの完了後にビジネス・ロジックを実行できます。また、オーバーロードされたcancelメソッドを使用して、リクエストの処理を強制終了することが可能です。強制終了した場合、HTTP 503のエラー・レスポンスがクライアントに送信されます。

クライアント・サイドの非同期処理:JAX-RS APIでは、InvocationインタフェースとAsyncInvokerインタフェースを使用して、非同期のリクエスト呼出しを実行できます。Invocationインタフェースは、オーバーロードされたsubmitメソッドによって非同期の送信を処理します。また、AsyncInvokerインタフェースは、専用のメソッド(get()、post()、put()など)を使用した、標準的なHTTPアクションの非同期呼出しをサポートします。サポートされる標準的なHTTPアクションは、GET、PUT、POST、DELETE、HEAD、TRACE、OPTIONSです。

InvocationCallbackの実装を登録していれば、非同期リクエストが処理された後にその実装が自動的に実行されます。この仕組みによって、例外が発生するシナリオや、正常に実行されたシナリオに対応できます。コールバックが登録されていない状況でFutureオブジェクトが取得される場合は、手動でそのオブジェクトをポーリングしてレスポンスを操作します。isDone、get、cancelなどのメソッドを呼び出すことができます。以下に、コールバックとFutureオブジェクトを使用したレスポンスの処理について示します。

```
Invocation.Builder builder1 = //...
AsyncInvoker invoker = builder1.async();
```



```
//obtain a Future
Future<Response> future1 = invoker.get();

//create another builder
Invocation.Builder builder2 = ...
Invocation invocation = builder2.buildGet();

//provide a callback
Future<Response> future2 =
    invocation.submit(
        new InvocationCallback<Customer>(){
            public void completed(Customer cust){
                System.out.println(
                    "Customer ID:" + cust.getID());
            }
            public void failed(Throwable t){
                System.out.println(
                    "Unable to fetch Cust details: " +
                    t.getMessage());
            }
        }
    );
```

高度な例外処理

JAX-RS 2.0の他の新機能を紹介する前に、堅牢な例外処理のためにJAX-RSフレームワークによって提供されている既存の例外処理機能のいくつかについて確認します。まず、`javax.ws.rs.core.Response`オブジェクトによってHTTPエラー状態をラップして、呼出し元に戻すことができます。また、`javax.ws.rs.WebApplicationException` (非チェック例外) を、ネイティブのビジネス固有/ドメイン固有の例外と、それらの例外に対応するHTTPエラー・レスポンスを橋渡しする目的で使用できます。`javax.ws.rs.ext.ExceptionMapper`は、Javaの例外を`javax.ws.rs.core.Response`オブジェクトにマッピングするプロバイダ用の契約 (インタフェース) を表します。この`ExceptionMapper`は、`javax.ws.rs.WebApplicationException`の拡張版と考えることができ、例外処理でDRY (Don't Repeat Yourself, 「重複の排除」) 原則に従うために使用できます。以下のように、ビジネス・ロジックの例外と適切なHTTPレスポンスの間で柔軟なマッピングを定義できます。

```
public class BookNotFoundExceptionMapper implements
    ExceptionMapper<BookNotFoundException>{
    @Override
    Response toResponse(
        BookNotFoundException bnfe){
        return Response.status(404).build();
    }
}
```

JAX-RS 2.0には、`javax.ws.rs.WebApplicationException`から継承した非チェック例外が追加されました。この設計によって、明示的なHTTPエラー情報を含む`WebApplicationException`を作成してスローする必要がなくなります。新しい例外にはわかりやすい名前が付いており、デフォルトでは、各例外と個別のHTTPエラー・シナリオが1対1でマッピングされています。つまり、新しい例外のいずれかをリソース・クラスからスローすると、マッピングに従って定義済みのHTTPエラー・レスポンスがクライアントに送信されます。たとえば、`NotAuthorizedException`をスローした場合、クライアントはHTTP 403を受け取ることになります (マッピングの詳細については表1を参照)。例外の振る舞いは、`ExceptionMapper`を使用して別の`Response`オブジェクトを返すことによって、実行時に変更するこ

例外	HTTPエラー・コード
<code>BadRequestException</code>	400
<code>ForbiddenException</code>	403
<code>InternalServerErrorException</code>	500
<code>NotAcceptableException</code>	406
<code>NotAllowedException</code>	405
<code>NotAuthorizedException</code>	401
<code>NotFoundException</code>	404
<code>NotSupportedException</code>	415
<code>ServiceUnavailableException</code>	503

表1: 例外クラスとHTTPエラー・コードのマッピング



ともできます。図2に、この新しい例外階層を示します。

これらの例外クラスと、そのそれぞれに対応するHTTPエラー・コードのマッピングについては、表1を参照してください。

その他の注目すべき拡張機能

これまでに紹介した目玉機能の他にも、JAX-RS 2.0には、アプリケーションの要件に合致するならばぜひ使用すべき便利な拡張機能が搭載されています。

@BeanParam: このアノテーションを使用して、カスタムのPOJOまたはBeanを、そのインスタンスにParam系の各種アノテーションを付加できる場合にインジェクションできます。Param系のアノテーションとは、@HeaderParam、@CookieParam、@PathParam、@QueryParamなどです。以下に示すとおり、HTTPリクエストの個別の要素をJAX-RSリソースに

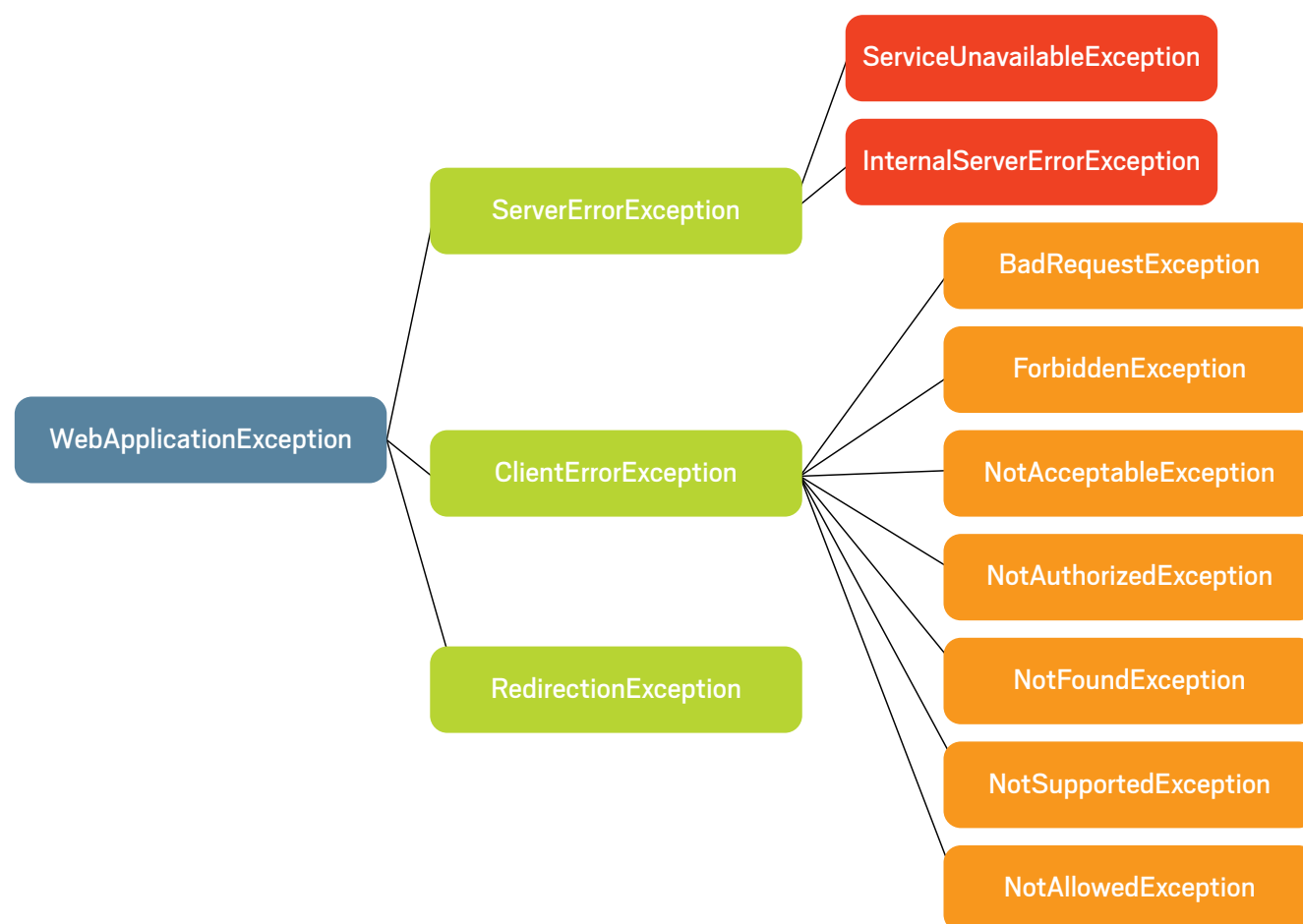


図2: JAX-RS 2.0の例外階層

インジェクションする代わりに、単純なPOJOを利用してHTTP URIのパラメータを取得できるため便利です。

```
public class CustomerSearchRequest{
    @QueryParam("id")
    private String userid;

    @HeaderParam("Accept")
    private String accept;

    @CookieParam("lastAccessed")
    private Date lastAccessed;
    //getters to fetch the values
}
```

JAX-RSランタイムは、`@javax.ws.rs.BeanParam`アノテーションが付加されたメソッド引数(またはインスタンス変数)のインスタンスをインジェクションします。

@ConstrainedTo: JAX-RSには、サーバー・サイドとクライアント・サイドに適用可能なプロバイダ・コンポーネントがいくつかあります。インターセプタはその好例で、ReaderInterceptorとWriterInterceptorの両方を、サーバー・サイドにもクライアント・サイドにも適用できます。この場合、状況によってクライアント・サイドまたはサーバー・サイドのいずれかにのみ使用できるように明示的に制限する目的で、@javax.ws.rs.ConstrainedToアノテーションをプロバイダ・クラスに適用できます。

ParamConverter: ParamConverterインタフェースは、自動インジェクションの前提条件を満たしていないカスタムのオブジェクトまたはPOJOを処理する場合に便利です(この前提条件とは、String引数を受け取るパブリック・コンストラクタがあること、またはString引数を受け取り、同じタイプのインスタンスをPOJOとして返す静的なvalueOfメソッドがあることです)。ParamConverterインタフェースの実装を使用して、Param系のアノテーション(@QueryParam、@PathParamなど)によって取得したString型のパラメータをカスタムのPOJOに変換するためのカスタム・ロジックを指定できます。

まとめ

JAX-RSフレームワークは、需要のある抽象表現をHTTPの上層に配置することで、REST指向アプリケーションを構築するための強固なプラットフォームとして動作します。JAX-RSの2.0リリースでは以前のリリースが全面的に見直されており、APIが多数追加されています。JAX-RSが提供する幅広いオプションをくまなく活用するまでには至っていない場合、手書きのコードが散在している可能性が高く、それらのコードをJAX-RSの機能に置き換えればメリットがあるはずです。

Abhishek Gupta: Oracle Identity Governance (ミドルウェア) 製品スタック専門のJava EE開発者、アーキテクト、コンサルタント。その他、スケーラブル・アーキテクチャ、分散キャッシュ・テクノロジー、デザイン・パターンなどに関心がある。

learn more

最新のJAX-RS 2.0仕様

Jersey

Roy Fielding氏のRESTに関する独創的な論文

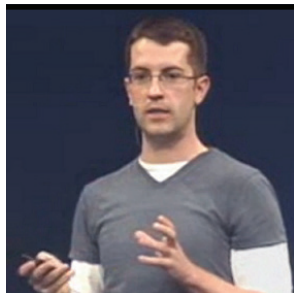
注目のJDK強化提案

JEP 254: Compact Strings

JEP 254では、JVM内部での文字列の内部表現を変えることが提案されています。おそらくご存じのとおり、文字列の格納にはUTF-16が使用されており、1文字あたり2バイトを占めます。このJEPでは、もっとコンパクトな、1文字あたり1バイトの表現を内部的に使用することが提案されています。「多数のさまざまなアプリケーションから収集されたデータから判断すれば、文字列はヒープを利用する主要な要素であり、さらに、大部分のStringオブジェクトにはLatin-1の文字しか含まれていません。Latin-1の各文字には1バイトの記憶域しか必要ありません。そのため、このようなStringオブジェクトの内部char配列の領域は、半分以上が未使用の状態になっています」と、このJEPでは述べられています。

コンパクトな形式に変えても、既存のコードやAPIには影響ありません。純粋にJVM内部の変更となり、プログラマーが認識することはありません。

興味深いことに、このJEPのWebページに、Java 6で文字列圧縮機能のテストが行われたと記載されています。そのテストでは、String.valueを、7ビット文字の配列または通常のJava文字の配列を示すObjectに変換したということです。しかしその後、この機能は削除されています。



DANNY COWARD

WebSocketを使用した 双方向のデータ・プッシュ

WebSocketの長期持続型コネクションを使ったシンプルなチャット・アプリの構築

本記事のパート1では、WebSocketの概要について説明したうえで、基本的なWebSocketプロトコルでは、2つのネイティブ形式(テキストとバイナリ)を使用できることを紹介しました。初歩的なアプリケーションで、単純な情報だけをクライアント/サーバー間でやり取りするなら、この基本機能で十分です。たとえば、パート1の時計アプリケーションでは、WebSocketメッセージのやり取りで交換されるデータは、サーバー・エンドポイントからブロードキャストされる書式設定済みの時刻文字列と、更新を終了するためにクライアントから送信されるstopという文字列だけです。しかし、アプリケーションがWebSocketコネクション経由でもっと複雑な情報を送受信するようになれば、情報を格納するための構造を見つける必要があります。私たちJava開発者は、アプリケーション・データをオブジェクトという形で扱うことに慣れています。オブジェクトには、標準Java APIに含まれるクラスから生成したものと、独自開発したJavaクラスから生成したものがあります。つまり、もっとも低レベルのJava WebSocket APIメッセージング機能を使い続けながら、文字列やバイト配列ではなくオブジェクトを使用してメッセージ・プログラムを開発するなら、オブジェクトから文字列またはバイト配列のいずれかへの変換とその逆を実行するコードを書く必要があります。本記事ではその具体的な方法を確認しましょう。

幸いにも、Java WebSocket APIには、オブジェクトからWebSocketメッセージへのエンコード処理と、WebSocketメッセージからオブジェクトへのデコード処理に役立つ機能があります。

はじめに、Java WebSocket APIは受け取ったメッセージを、リクエストされた任意のJavaプリミティブ型(または対応するクラス)に変換しようとします。つまり、以下の形式でメッセージ処理メソッドを宣言できます。

```
@OnMessage  
public void handleCounter(int newvalue) {...}
```

または

```
@OnMessage  
public void handleBoolean(Boolean b) {...}
```

この場合、Java WebSocket実装は受信メッセージを、宣言されたJavaプリミティブ・パラメータ型に変換しようとします。

同じように、`RemoteEndpoint.Basic`の送信メソッドにも以下の汎用メソッドがあります。

```
public void sendObject(Object message)  
    throws IOException, EncodeException
```

このメソッドに任意のJavaプリミティブ(または対応するクラス)を渡すことで、Java WebSocket実装によって、その値が対応する文字列に変換されます。



しかし、アプリケーションでもっと高レベルの構造化オブジェクトを使用してメッセージを表したいと考える場合もよくあります。メッセージ処理メソッドでカスタム・オブジェクトを扱うには、エンドポイントと一緒にWebSocketのDecoder実装を提供する必要があります。ランタイムはこの実装を使用して、受信メッセージをカスタム・オブジェクト型のインスタンスに変換します。反対に、送信メソッドでカスタム・オブジェクトを扱うには、WebSocketのEncoder実装を提供する必要があります。ランタイムはこの実装を使用して、カスタム・オブジェクトのインスタンスをネイティブWebSocketメッセージに変換します。この仕組みを要約したものが図1になります。

図1の上部に示したエンドポイントはクライアントとの間で文字列を交換しており、下部のエンドポイントはエンコーダとデコーダを使用して、FooオブジェクトからWebSocketテキスト・メッセージへの変換とその逆を実行しています。

Java WebSocket APIには、`javax.websocket.Decoder`インタフェースおよび`javax.websocket.Encoder`インタフェースのファミリがあり、希望の変換タイプに応じて選択できます。たとえば、テキスト・メッセージをFooというカスタム開発者クラスのインスタンスに変換するDecoderを実装するためには、Fooを総称型として使用して`Decoder.Text<T>`インタフェースを実装します。具体的には、以下のメソッドを実装します。

```
public void sendObject(Object message)
```

```
throws IOException, EncodeException
```

これはデコーダの主要メソッドで、新しいテキスト・メッセージを受け取るたびに呼び出されて、Fooクラスのインスタンスを生成します。その後で、ランタイムが、エンドポイントのメッセージ処理メソッドにこのクラスを渡すことができます。

Decoderにはきょうだいクラスがあり、ブロッキングI/Oストリーム（こちらでもサポート対象）の形で受け取るWebSocketメッセージとバイナリのWebSocketメッセージをデコードします。

カスタム開発者クラスFooのインスタンスをWebSocketテキスト・メッセージに変換するEncoderを実装するためには、Fooを総称型として使用して`Encoder.Text<T>`インタフェースを実装します。具体的には、以下のメソッドを実装します。

```
public String encode(Foo foo)
    throws EncodeException
```

この実装により、Fooインスタンスが文字列に変換されます。RemoteEndpointのsendObject()メソッド（前述）を呼び出して、Fooクラスのインスタンスを渡す場合、Java WebSocketランタイムはこの文字列を必要とします。Decoderと同じように、Encoderにもきょうだいクラスがあり、メッセージを送信するために、カスタム・オブジェクトをバイナリ・メッセージに変換するか、またはカスタム・オブジェクトをブロッキングI/Oストリームに書き込みます。

この方法をエンドポイントで使用するのは簡単です。@ClientEndpointと@ServerEndpointの定義で確認したように、エンドポイントで使用するデコーダおよびエンコーダの実装を、それぞれdecoders()属性とencoders()属性に指定するだけで完了です。

Javaプリミティブ型のエンコーダまたはデコーダを独自に設定した場合、ランタイムでその型のデフォルトになっているエンコーダまたはデコーダは、当然ながらオーバーライドされます。

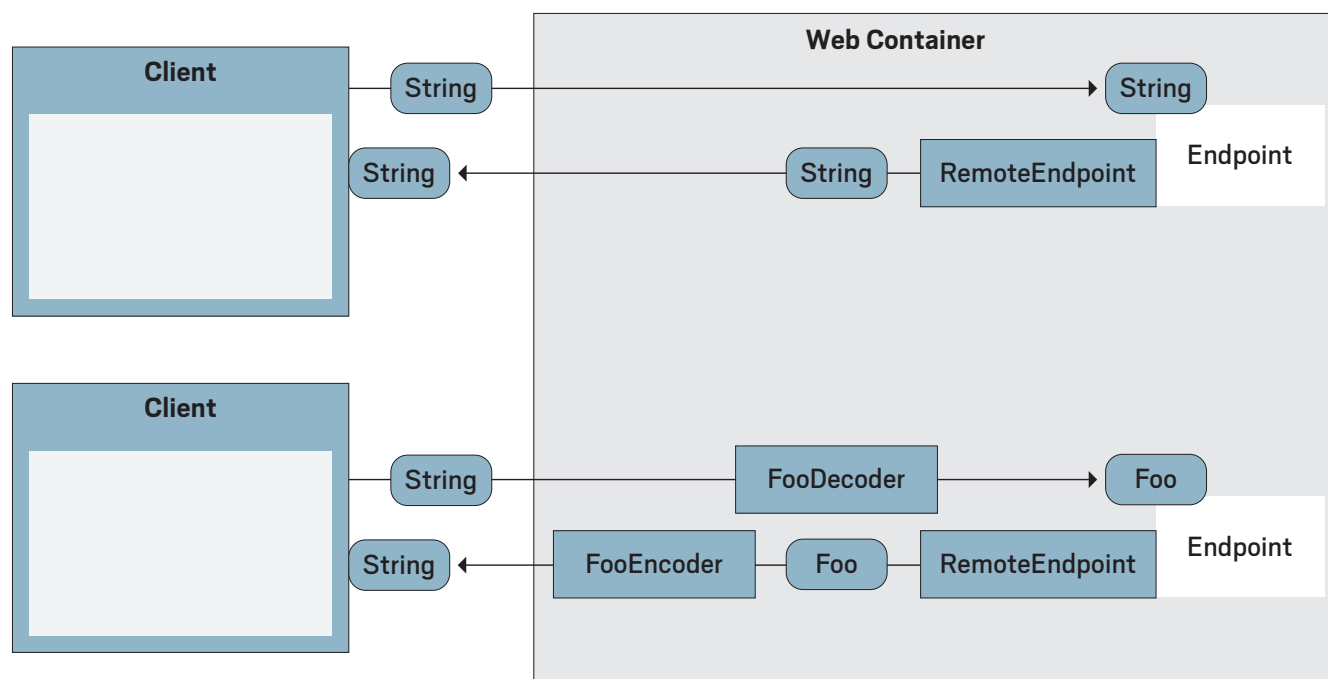


図1:エンコーダとデコーダ



メッセージ処理モード

ここまでで説明したのは、一度に1つのWebSocketメッセージ全体を送信または受信する方法だけです。多くのアプリケーションはアプリケーション・プロトコルで小さいメッセージだけを定義しているため、このシンプルなモデルを使用してメッセージを処理していますが、写真や大容量文書の送信などで大きいWebSocketメッセージを処理するアプリケーションもあります。Java WebSocket APIには、大きいメッセージを適切に効率良く処理するためのモードがいくつかあります。

大きいメッセージの受信: Java WebSocket APIには追加で2つのメッセージ受信モードがあり、メッセージが大きくなることがわかっている場合に適しています。最初のモードでは、エンドポイントがメッセージ消費に使用できるブロッキングI/O APIに対して、エンドポイントを公開します。テキスト・メッセージの場合は[java.io.Reader](#)を、バイナリ・メッセージの場合は[java.io.InputStream](#)を使用します。このモードを使用する場合、メッセージ処理メソッドで[String](#)パラメータまたは[ByteBuffer](#)パラメータのいずれかを使用するのではなく、[Reader](#)または[InputStream](#)を使用します。以下にその例を示します。

```
@OnMessage
public void handleMessageAsStream(
    InputStream messageStream,
    Session session) {
    // read from the messageStream
    // until you have consumed the
    // whole binary message
}
```

2つ目のモードでは、一種の要素分割APIを使用し、WebSocketメッセージを小さいチャンクに分けて[boolean](#)フラグと一緒にメッセージ・ハンドラ・メソッドに渡すことができます。このフラグから、すべてのチャンクが受信済みでメッセージが全部そろっているかどうか分かります。言うまでもなく、メッセージ要素は順序どおりに到着し、合間にその他のメッセージが割り込むことはありません。このモードを使用する場合、メッセージ・ハンドラ・メソッドに[boolean](#)パラメータを追加します。以下にその例を示します。

```
@OnMessage
public void handleMessageInChunks(
    String chunk, boolean isLast) {
    // reconstitute the message
    // from the chunks as they arrive
}
```

このモードでは、メッセージの送信元ピアとJava WebSocketランタイムの構成に関連するいくつかの要素によって、チャンクのサイズが異なります。開発者が知る必要があるのは、多数のチャンクに分割された状態でメッセージを受け取るということだけです。

メッセージ送信モード: 予想されているとおり、WebSocketプロトコルは対称であるため、Java WebSocket APIには、大きいサイズのメッセージ送信に適した同等のモードがあります。前述のようにメッセージすべてを1つのチャンクで送信するだけでなく、ブロッキングI/Oストリームにメッセージを送信することもでき、メッセージがテキストかバイナリかによって、[java.io.Writer](#)または[java.io.OutputStream](#)のいずれかを使用します。以下のメソッドは、当然ですが、[Session](#)オブジェクトから取得する[RemoteEndpoint.Basic](#)インタフェースの追加メソッドです。

```
public Writer getSendWriter() throws IOException
```

および

```
public OutputStream getSendStream()
    throws IOException
```

2つ目のモードはチャンク・モードですが、今回は受信ではなく逆に送信を行います。ここでも、エンドポイントは、以下に示す[RemoteEndpoint.Basic](#)のいずれかのメソッドを呼び出すことで、このモードでメッセージを送信できます。



```
public void sendText(
    String partialTextMessage, boolean isLast)
    throws IOException
```

または

```
public void sendBinary(
    ByteBuffer partialBinaryMessage,
    boolean isLast)
    throws IOException
```

いずれを選択するかは、送信するメッセージの種類によって決まります。

メッセージの非同期送信: WebSocketメッセージの受信は常に非同期です。通常、エンドポイントはいつメッセージが到着するかを認識しておらず、メッセージはピアが選んだタイミングで到着します。ここまでに確認したとおり、[RemoteEndpoint.Basic](#) インタフェースのメッセージ送信メソッド(紹介してきたメソッドの大部分)はすべて同期送信です。つまり、メッセージが送信されるまで、[send\(\)](#) メソッド・コールは常にブロックされます。メッセージが小さい場合、このブロックは問題になりません。しかし、メッセージが大きい場合、メッセージ送信が終わるまで WebSocket を待機させるよりも、別の宛先へのメッセージ送信やユーザー・インタフェースの再描画、受信メッセージの処理へのリソース集中など、WebSocket に別の処理を実行させた方が良いでしょう。このようなエンドポイントでは、[Session](#) オブジェクトから取得できる ([RemoteEndpoint.Basic](#) と同様) [RemoteEndpoint.Async](#) の [send\(\)](#) メソッドを使用することにより、メッセージ全体を(さまざまな形式の)パラメータとして渡すことができます。渡したメッセージが実際に送信される前に、メソッド呼出しは即座に完了します。たとえば、大きいテキスト・メッセージを送信する場合は、以下のメソッドを使用します。

```
public void sendText(
    String textMessage, SendHandler handler)
```

このメソッド呼出しはすぐに完了し、メッセージが実際に送信されたときに、メソッドに渡した [SendHandler](#) がコールバックを受け取ります。こうすることで、メッセージが送信されたことを認識できますが、実際に送信さ

れるまで待つ必要はありません。別の方法としては、非同期メッセージ送信の進捗を定期的に確認する方法もあります。たとえば、以下のメソッドを使用するとします。

```
public Future<Void> sendText(
    String textMessage)
```

この場合、メソッド呼出しは即座に、メッセージが送信される前に完了します。メッセージから返された [Future](#) オブジェクトに送信メッセージのステータスを問い合わせることができ、気が変わった場合は送信を取り消すこともできます。

ご想像のとおり、これらのメソッドにはバイナリ・メッセージ版もあります。

Java WebSocket API を使用したメッセージ送信から次のトピックに移る前に、WebSocket プロトコルには配信保証という考え方がないことを説明する必要があります。言い換えると、メッセージを送信しても、クライアントが受け取ったかどうかを確かめる手段はありません。エラー・ハンドラ・メソッドがエラーを受け取った場合、このエラーは通常、メッセージが正しく配信されなかったことを示す明らかな指標です。しかし、エラーがないからといって、メッセージが正しく配信されたとは限りません。Java WebSocket を使用して独自の相互作用を構築し、重要なメッセージにはピアから受信アクノリッジを返すこともできますが、JMS などの他のメッセージング・プロトコルとは異なり、WebSocket には固有の配信保証機能がありません。



27

実行時のパス情報アクセス: エンドポイントは実行時に、すべてのパス情報に完全にアクセスできます。はじめに、WebSocket実装がエンドポイントを公開したパスはいつでも取得できます。エンドポイントに対して `ServerEndpointConfig.getPath()` を呼び出すことで、この情報を取得できます。このメソッドは、`ServerEndpointConfig` インスタンスを取得できればどこでも簡単に使用できます。詳しくは、リスト2を参照してください。

■ **リスト2:** エンドポイントによるパス・マッピングへのアクセス

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService {
    public void handleConnection(
        Session s, EndpointConfig config) {
        String myPath =
            ((ServerEndpointConfig) config).getPath();
        // myPath is "/travel/hotels/{stars}"
        ...
    }
}
```

このアプローチは、URIが完全マッピングされたエンドポイントでも同様に使用できます。

実行時にエンドポイント内で取得する2つ目の情報は、クライアントがエンドポイントに接続したときに使用したURIです。後述するとおり、この情報はさまざまな形式で取得できますが、以下の主要メソッドにはすべての情報が含まれています。

`Session.getRequestURI()`

このメソッドは、WebSocketが実装されたWebサーバーのルートへの相対的なURIパスを返します。このURIパスには、WebSocketが含まれるWebアプリケーションのコンテキスト・ルートが含まれます。したがって、ホテル予約サンプルが `/customer/services` というコンテキスト・ルートを持つWebアプリケーションにデプロイされており、クライアントが以下のURIを使用して `HotelBookingService` エンドポイントに接続している場合、

`ws://fun.org/customer/services/
travel/hotels/3`

エンドポイントが `getRequestURI()` を呼び出して受け取るリクエストURIは、以下になります。

`/customer/services/travel/hotels/3`

リクエストURIに問合せ文字列が含まれる場合、`Session` オブジェクトに対して追加で2つのメソッドを呼び出すことで、このリクエストURIからさらなる情報を取得できます。ここで、問合せ文字列について確認しましょう。

問合せ文字列とリクエスト・パラメータ: 先ほど確認したように、WebSocketエンドポイントへのURIパスに続くのは、オプションである問合せ文字列です。

```
<ws or wss>://<host:name>:<port:>/
    <web-app-context-path>/<websocket-path>?
    <query-string>
```

URI内の問合せ文字列を最初に一般的にしたのは、Common Gateway Interface (CGI) アプリケーションでした。URIのパス部分がCGIプログラム（通常は `/cgi-bin`）の位置を特定し、URIパスに続く問合せ文字列が、CGIプログラムがリクエストを絞り込むためのパラメータ・リストを提供します。問合せ文字列は、HTML形式を使用してデータを投稿する場合にもよく使用されます。たとえば、Webアプリケーションに以下のHTMLコードが含まれているとします。

```
<form
    name="input"
    action="form-processor" method="get">    Your Username:
<input type="text" name="user">
        <input type="submit" value="Submit">
</form>
```



「Submit」ボタンをクリックすることで、以下のURIに対するHTTPリクエストが生成されます。

`/form-processor?user=Jared`

このURIは、HTMLコードを含むページとJaredというテキストを含む入力フィールドの場所に関連しています。`/form-processor`というURIパスにあるWebリソースの性質に応じて、問合せ文字列`user=Jared`を使ってどのようなレスポンスを返すかを決定できます。たとえば、`form-processor`にあるリソースがJavaサーブレットである場合、このJavaサーブレットは、`getQueryString()` APIコールを使用して、`HttpServletRequest`から問合せ文字列を取得できます。

同様に、Java WebSocket APIを使用して作成されたWebSocketエンドポイントに接続するときに、問合せ文字列を含むURIを使用できます。Java WebSocket APIは、最初のハンドシェイク・リクエストのリクエストURIの一部として送信された問合せ文字列を、一致するエンドポイントの特定には使用しません。言い換えると、リクエストURIに問合せ文字列が含まれているかどうかは、サーバー・エンドポイントの公開パスのマッチングには関係ありません。また、エンドポイントの公開で使用するパスに含まれる問合せ文字列は無視されます。

CGIプログラムやその他のWebコンポーネントと同様に、WebSocketエンドポイントは問合せ文字列を使用して、クライアントの接続を追加設定します。WebSocket実装は基本的に、受信リクエストに含まれる問合せ文字列の値を無視するため、問合せ文字列の値を使用するロジックはWebSocketコンポーネント内だけで有効になります。問合せ文字列の値を取得するために使用できる主要なメソッドはすべて、`Session`オブジェクトに含まれています。

```
public String getQueryString()
```

上記メソッドは、問合せ文字列全体（「?」文字に続く部分すべて）を返します。

```
public Map<String,List<String>>
    getRequestParameterMap()
```

上記メソッドは、問合せ文字列から解析されたすべてのリクエスト・パラメータを含むデータ構造を返します。マップの値が文字列リストに

なっているのは、問合せ文字列に、同じ名前を持つパラメータが複数含まれ、値が異なる場合があるからです。たとえば、以下のURIを使用してHotelBookingServiceに接続するとしましょう。

```
ws://fun.org/customer/
    services/travel/hotels/4?
    showpics=thumbnails&
    description=short
```

この場合の問合せ文字列は`showpics=thumbnails&description=short`であり、エンドポイントからリクエスト・パラメータを取得するためには、リスト3のような処理が必要になります。

■ リスト3: リクエスト・パラメータへのアクセス

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService2 {
```

```
    public void handleConnection(
        Session session, EndpointConfig config) {
        String pictureType =
            session.getRequestParameterMap()
                .get("showpics").get(0);
        String textMode =
            session.getRequestParameterMap()
                .get("description").get(0);
        ...
    }
    ...
}
```

上記で、`pictureType`と`textMode`の値は、それぞれ`thumbnails`と`short`になります。

問合せ文字列は、リクエストURIからも取得できます。Java WebSocket APIで`Session.getRequestURI`を呼び出した場合、常にURIパスと問合せ文字列の両方が返されます。



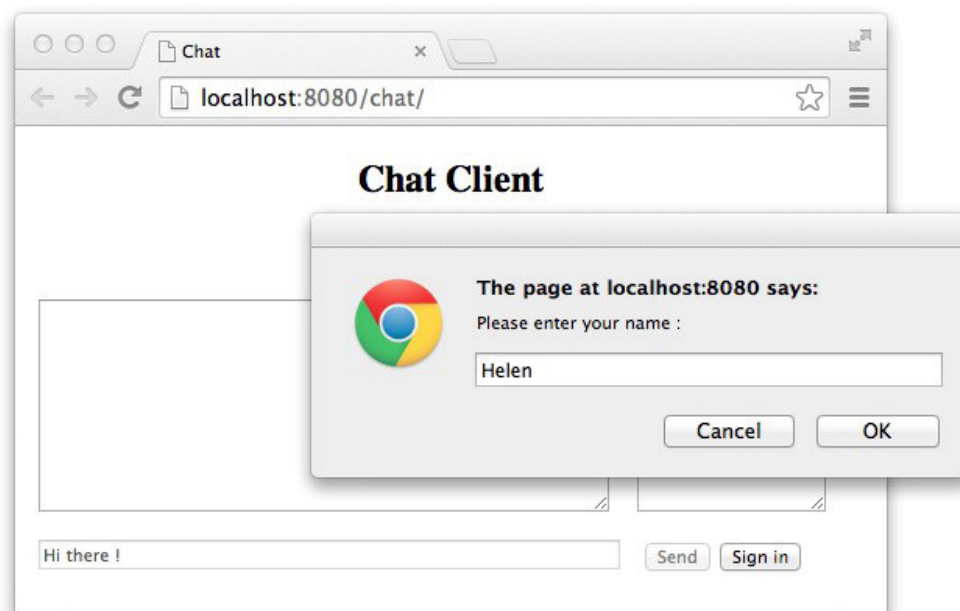


図2:チャットへのログイン

サーバー・エンドポイントのデプロイ:

Java EE WebコンテナへのJava WebSocketエンドポイントのデプロイは、非常に簡単です。[@ServerEndpoint](#)アノテーションが付加されたJavaクラスをWARファイルにパッケージした場合、Java WebSocket実装がこのWARファイルをスキャンして、このアノテーションが付加されたクラスをすべて見つけてデプロイします。つまり、サーバー・エンドポイントのデプロイには、WARファイルへのパッケージ以外に特別な処理は必要ありません。しかし、一連のサーバー・エンドポイントのうち、WARファイルにデプロイするエンドポイントを厳密に制御が必要があるとしてします。この場合、[javax.websocket.ServerApplicationConfig](#)というJavaWebSocket API インタフェースを実装することで、デプロイするエンドポイントをフィルタリングできます。

チャット・アプリケーション

プッシュ・テクノロジーをテストする良い方法として、関係する多数のクライアントに非同期で頻繁に更新を送信するアプリケーションの構築が挙げられます。この目的に適しているのがチャット・アプリケーションです。ここからは、Java WebSocket APIについて学習した内容を活用して、シンプルなチャット・アプリケーションを構築する方法を確認しましょう。

図2は、チャット・アプリケーションのメイン・ウィンドウで、サインインするときにユーザー名を入力します。

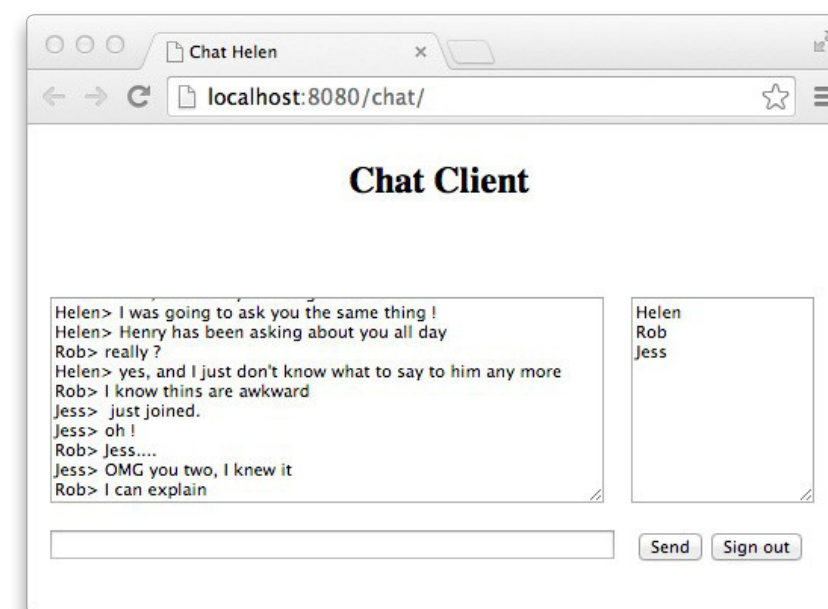


図3:参加中のチャット

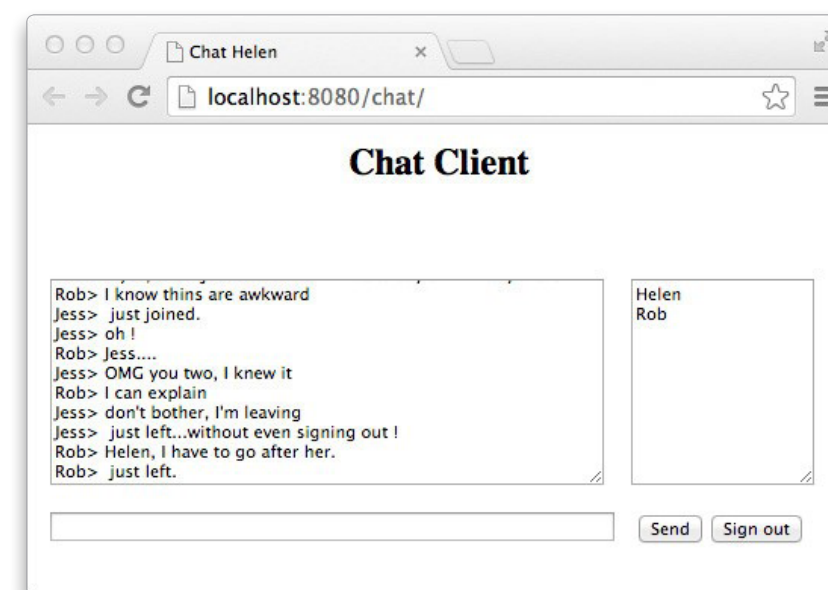


図4:チャット・ルームからの退室

複数のユーザーが同時にチャットでき、ウィンドウ下部のテキスト・フィールドにメッセージを入力して、「Send」ボタンをクリックします。アクティブなチャット参加者が右側に表示され、全員のメッセージを記録したトランスクリプトが共有され、左側に表示されます。図3では、3名の参加者の会話がぎくしゃくしています。

図4では1名の参加者が突然チャットを終了し、もう1名もその後を追ったため、1名だけがチャット・ルームに残されました。



コードを詳しく調べる前に、アプリケーション構築の全体像を把握しましょう。このWebページは、JavaScript WebSocketクライアントを使用して、すべてのチャット・メッセージを送受信しています。Webサーバー上に、[ChatServer](#)という1つのJava WebSocketエンドポイントがあり、誰かがチャット・ルームに入退室するか、グループ宛てにメッセージが送信されるたびに、複数のクライアントからのチャット・メッセージをすべて処理し、アクティブに参加しているクライアントを追跡し、トランスクリプトを維持し、すべての接続クライアントに更新をブロードキャストします。このアプリケーションは、カスタム・オブジェクトに対してWebSocketの[Encoders](#)と[Decoders](#)を使用して、すべてのチャット・メッセージをモデル化しています。

リスト4の[ChatServer](#)エンドポイントに注目してください。[サイズの関係上、リスト4は記事内に記載していません。本記事の[ダウンロード・ページ](#)からダウンロードできます(編集部より)]

このコードには着目すべき点が多くあります。第一に、これは、[/chat-server](#)という相対URIにマッピングされたサーバー・エンドポイントです。このエンドポイントは、それぞれ[ChatEncoder](#)および[ChatDecoder](#)というエンコーダ・クラスとデコーダ・クラスを使用しています。

初めてJava WebSocketエンドポイントを見るときは、ライフサイクル・メソッドからチェックすると良いでしょう。ご存知のとおり、ライフサイクル・メソッドには、[@OnOpen](#)、[@OnMessage](#)、[@OnError](#)、[@OnClose](#)というアノテーションが付加されています。このようにして[ChatServer](#)クラスを見ると、新規クライアントの接続時に[ChatServer](#) WebSocketが最初に処理することは、チャットのトランスクリプト、セッション、[EndpointConfig](#)を参照するインスタンス変数の設定であると分かります。クライアントが接続するたびに、新しいエンドポイント・インスタンスが作成されるため、チャット・ルーム内の参加者ごとに、独自のチャット・サーバー・インスタンスがエンドポイントに関連付けられます。[EndpointConfig](#)の数は、論理的なWebSocketエンドポイントごとに常に1つになるため、各[ChatServer](#)インスタンスの[endpointConfig](#)インスタンス変数が指すのは、[EndpointConfig](#)クラスの単一共有インスタンスになります。このインスタンスはシングルトンであり、任意のアプリケーション・ステートを格納できるユーザー・マップを保持しています。このため、アプリケーションのグローバル・ステートを格納する場所として最適です。クライアント接続ごとに固有のセッション・オブジェクトが作成されるため、それぞれの[ChatServer](#)インスタンスは、独自の[Session](#)インスタンスを参照します。このインスタンスは、[Transcript](#)クラスのコードに従って、関連付けられたクライアントを表します。詳しくは、リスト5を参照してください。

■ リスト5: Transcriptクラス

```
import java.util.ArrayList;
import java.util.List;
import javax.websocket.*;

public class Transcript {
    private List<String> messages =
        new ArrayList<>();
    private List<String> usernames =
        new ArrayList<>();
    private int maxLines;
    private static String
        TRANSCRIPT_ATTRIBUTE_NAME =
            "CHAT_TRANSCRIPT_AN";

    public static Transcript
        getTranscript(EndpointConfig ec) {
        if (!ec.getUserProperties().
            containsKey(TRANSCRIPT_ATTRIBUTE_NAME)) {
            ec.getUserProperties()
                .put(TRANSCRIPT_ATTRIBUTE_NAME,
                    new Transcript(20));
            return (Transcript) c.getUserProperties()
                .get(TRANSCRIPT_ATTRIBUTE_NAME);
        }

        Transcript(int maxLines) {
            this.maxLines = maxLines;
        }

        public String getLastUsername() {
            return usernames.get(usernames.size() - 1);
        }

        public String getLastMessage() {
            return messages.get(messages.size() - 1);
        }
    }
}
```



```
public void addEntry(
    String username, String message) {
    if (usernames.size() > maxLines) {
        usernames.remove(0);
        messages.remove(0);
    }

    usernames.add(username);
    messages.add(message);
}
}
```

コードから、EndpointConfigごとに1つのTranscriptインスタンスがあることが分かります。つまり、1つのTranscriptインスタンスがすべてのChatServerインスタンス間で共有されています。Transcriptがすべてのクライアントに対してグループ・チャット・メッセージを表示する必要があるため、この共有は適切です。

ChatServerでもっとも重要なメソッドは、@OnMessageアノテーションが付加されたメッセージ処理メソッドです。このメソッドは、ChatDecoderを使用して、テキストまたはバイナリのWebSocketメッセージではなく、ChatMessageオブジェクトを扱っていることがシグネチャから分かります。ChatDecoderにより、メッセージはあらかじめ、ChatMessageのサブクラスのいずれかにデコードされています。表記を簡潔にするため、ChatMessageのサブクラスをすべて記載するのではなく、表1にChatMessageの各サブクラスとその目的を示します。

ChatServerのメッセージ処理メソッドであるhandleChatMessage()は、チャット関連の新しいアクションが発生するたびにクライアントから呼び出され、新規ユーザーのサインイン、新規メッセージの投稿、ユーザーの

サインアウトを処理するように設計されていることが容易に分かります。それではここで、ユーザーによる新規チャット・メッセージの投稿がChatServerに通知されたときのコード・パスを調べてみましょう。handleChatMessage()メソッドからprocessChatUpdate()メソッドが呼び出され、さらにaddMessage()が呼び出されて、新規チャット・メッセージが共有Transcriptに追加されます。次に、リスト6に示すbroadcastTranscriptUpdate()が呼び出されます。

■ リスト6: 新規チャット・メッセージのブロードキャスト

```
private void broadcastTranscriptUpdate() {
    for (Session nextsession :
        session.getOpenSessions()) {
        ChatUpdateMessage cdm =
            new ChatUpdateMessage(
                this.transcript.getLastUsername(),
                this.transcript.getLastMessage());

        try{
            nextsession.getBasicRemote().sendObject(cdm);
        } catch (IOException | EncodeException ex) {
            System.out.println(
                "Error updating a client : " +
                ex.getMessage());
        }
    }
}
```

CHATMESSAGEのサブクラス	目的
ChatUpdateMessage	ユーザー名と、ユーザーが送信したチャット・メッセージを格納したメッセージ
NewUserMessage	新しくサインオンしたユーザーの名前を格納したメッセージ
UserListUpdateMessage	現在アクティブなチャット参加者の名前リストを格納したメッセージ
UserSignoffMessage	サインオフしたユーザーの名前を格納したメッセージ

表1: ChatMessageのサブクラス



このメソッドは、`Session.getOpenSessions()`という非常に便利なAPIコールを使って、1つのエンドポイント・インスタンスから、論理エンドポイントに対するすべてのオープン・コネクションのハンドルを取得しています。この場合、このメソッドは、すべてのオープン・コネクションを含むリストを使用して、新規チャット・メッセージを全クライアントにブロードキャストすることで、クライアントのユーザー・インタフェースが最新チャット・メッセージで更新されるようにします。送信されるメッセージは、`ChatMessage` (実際は`ChatUpdateMessage`)形式である点に注意してください。`ChatUpdateMessage`インスタンスからテキスト・メッセージへのマーシャリングを行うのは`ChatEncoder`であり、このテキスト・メッセージが新規チャット・メッセージの通知とともにクライアントに送り返されます。

受信メッセージを確認したとき、`ChatDecoder`については調べていなかったため、ここでいったん`ChatEncoder`クラスに注目してみましょう。コードをリスト7に示します。

■ リスト7: ChatEncoderクラス

```
import java.util.Iterator;
import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;

public class ChatEncoder implements
    Encoder.Text<ChatMessage> {
    public static final String SEPARATOR = ":";

    @Override
    public void init(EndpointConfig config) {}
    @Override
    public void destroy() {}
```

このクラスで肝心なのは**encode()**メソッドであり、クライアントに送り返せるようにメッセージ・インスタンスを文字列に変換します。

```
@Override
public String encode(ChatMessage cm)
    throws EncodeException {
    if (cm instanceof StructuredMessage) {
        String dataString = "";
        for (Iterator itr =
            ((StructuredMessage) cm)
                .getList().iterator();
            itr.hasNext(); )
        {
            dataString =
                dataString + SEPARATOR +
                itr.next();
        }
        return cm.getType() + dataString;
    } else if (cm instanceof BasicMessage) {
        return cm.getType() +
            ((BasicMessage) cm).getData();
    } else {
        throw new EncodeException(cm,
            "Cannot encode messages of type: " +
            cm.getClass());
    }
}
```

Encoderのライフサイクル・メソッドである`init()`と`destroy()`を実装するためには、`ChatEncoder`クラスが必要です。このエンコーダは、コンテナからのコールバック内で何も実行しませんが、ライフサイクル・メソッド内で高コストなリソースの初期化と破棄を実行するエンコーダもあります。このクラスで肝心なのは**encode()**メソッドであり、クライアントに送り返せるようにメッセージ・インスタンスを文字列に変換します。

`ChatServer`クラスに戻って**handleChatMessage()**メソッドを見ると、このエンドポイントは、サインオフしたクライアントのコネクションを終了する前に、`UserSignoffMessage`を送信してうまく対応しています。また、ブラウザを閉じるか別のページに移動することで一方的にコネクションを切断したクライアントにも、次のように見事に対応しています。`@OnClose`ア



ノテーションを付加した`endChatChannel()`メソッドが、別れの挨拶なしでチャット・ルームから退室したユーザーがいることを通知するメッセージを、すべての接続クライアントにブロードキャストします。チャット画面のスクリーンショットを見返すと、JessとRobではチャット・ルームからの退室の仕方に違いがあることを確認できます。

まとめ

2回に分けてお届けした本記事では、Java WebSocketエンドポイントの作成方法を学習しました。WebSocketプロトコルの基本的な概念と、サーバーからのプッシュを必要とする状況について考察し、Java WebSocket エンドポイントのライフサイクル、Java WebSocket APIの主なクラス、エンコード処理およびデコード処理の手法を確認しました。また、Java WebSocket APIでサポートされる各種のメッセージング・モードに注目しました。サーバー・エンドポイントをWebアプリケーションのURI空間にマップする方法と、その中にあるエンドポイントにクライアント・リクエストをマッチングする方法についても学習しました。最後に、多数のJava WebSocket API機能を使用するチャット・アプリケーションについて考察しました。今回学んだ知識を使うことで、長期持続型コネクションを使用したアプリケーションを簡単に構築できます。</article>

本記事は、書籍『Java EE 7: The Big Picture』の内容を、発行者のOracle Pressの許可を得て改訂したものです。

Danny Coward: Liquid Robotics のプリンシパル・ソフトウェア・エンジニア。以前はオラクル（およびその前のSun Microsystems）でJava開発チームに属し、特にWebSocketに関する業務に従事した。

learn more

[オラクルのJava WebSocketチュートリアル](#)
[Long polling, a WebSocket alternative](#)

A promotional banner for Oracle Java's 20th anniversary. It features a man in a blue hoodie looking out over a city skyline. The text 'CREATE THE FUTURE' is at the top, with 'oracle.com/java' below it. A '20 YEARS 1995-2015' badge is on the right. The Java logo and 'ORACLE' logo are at the bottom.





BEN EVANS

プレリリース版で学ぶJava 9モジュール

Javaの次期メジャー・リリースの中核となるモジュールの導入に備える

現在まで、つまりJava 8まで、Java言語のコードへのアクセス制御モデルはかなり単純でした。

- クラスはパッケージの中に配置。パッケージにはグローバルな可視性があり、自由に拡張可能(ただし、javaまたはjavaxで始まるパッケージは例外)。パッケージはコードを論理的に整理するために作られており、それ以上の意味は持たない。
- コードは、JARファイル単位で提供。通常の(保護されていない)パッケージは、複数のJARファイルにまたがることが可能。
- クラスとメソッドへのアクセスは制限できる。たとえば、同じ型のインスタンスや、同じパッケージのコードのみがアクセスできるような制限をかけることが可能。この制限を表すのが、おなじみのJavaのアクセス制御キーワード。

この設計には、わかりやすく論理的であるというメリットがあります。しかし、アクセス制御に関するいくつかの問題につながることもあります。中でも、一番重大な問題は、JDK以外のライブラリでは、クライアント・コードからライブラリ・パッケージ内に追加クラスを作成することを防止できない点です。そういったクラスを作成すると、パッケージに定義されているすべてのprotectedやパッケージ・アクセスのクラス(やメソッド)にアクセスできてしまいます。

この問題は、「ショットガン・プライバシー」問題と呼ばれることがあります。この名前は、Perlプログラミング言語を設計したLarry Wall氏が自らの言語を分析した際の有名な言葉に由来します。「Perlは、プライバシーを強制することはしません。たとえ相手がショットガンを持っていないくても、招かれていないリビング・ルームには足を踏み入れない方がよいということです」

しかし、Javaでは、ショットガン・プライバシーは重大な問題となります。現在のJavaの言語や環境には、パッケージ全体にアクセス制御を適用する方法はありません。別の言い方をすると、あるJavaライブラリでパブリックAPIを定義したときに、クライアントによってAPIの前提を覆されな

いことや、パッケージの内部に直接アクセスされないことが確実な仕様が求められているということです。

多くのエンタープライズ・アプリケーションでは、厳密なレベルでアクセス制御を行い、決められた既知のAPIを通じてのみアクセスできるようにすることが大いに望まれています。しかし、Java 8以前では、こういったレベルのAPI制御を完璧に行うことは不可能です。これが、Oracle(以前はSun)がこのレベルの安全性を実現できる一連の技術を開発しようとした根本的な理由の1つでした。しかし、この問題への対応によって、他の問題も解決できる可能性が明らかになってきました。

その結果、来たるJDK 9のモジュール・システムの中核的な目標は次のようなものになっています。

- アクセス制御を拡張し、アクセスをデプロイ単位内に「封じ込める」という概念を導入することによって、パブリックAPIを完全に強制する
- 単一パッケージよりも大規模なモジュール式のコードのデプロイを促進する
- JDK自身をモジュール化することによって、JVMの起動時間の高速化、リソース消費量の削減、中核部に位置する必要のないパッケージの削除、内部クラスの削除または隠蔽によるプラットフォームの攻撃対象範囲の削減、といったことをプラットフォーム・レベルで実現する

Oracleは、OpenJDK内のプロジェクトであるProject Jigsawを支援してきました。このプロジェクトの使命は、Java環境のモジュール化を実現することです。JDK 9のリリース日が近づくにつれ、JigsawによるコードがOpenJDKリポジトリのメインラインに移行され始めました。つまり、この記事執筆している時点(2015年末)では、モジュール対応の各種バイナリの完成度レベルはさまざまであるということです。

- Oracleによる、モジュールを含む最初のJava 9 Early Access (EA) ビルドが入手可能になっています。ただし、このバイナリに含まれているモジュール機能には制限があります。
- メインラインのJDK 9ビルドを補完し、いつそう早い段階でモジュール

本記事の以降の部分では、JDK 9メインラインよりも完成度が高いJigsawのバイナリを中心としたサンプルを紹介して説明してゆきます。

Project Jigsawは、完全なモジュール化に向かうロードマップの過程として、いくつかの小目標を設定しました。これらの小目標は、Java Enhancement Process (JEP) で定義されたもので、次のようなテーマが含まれています。

- 最後の目標は、もっとも労力がかかるものの1つでしたが、モジュールを可能な限り独立したものにするためにも必要なことでした。やがて、この目標によって最大限の柔軟性が実現でき、モジュールを個別にロードおよびリンクできるようになります。

別の例として、RMI-IIOPトランスポートを挙げることができます。CORBAがなくても(JMXを使用する)リモート管理モジュールが動作するように、この機能は管理モジュールから切り離されました。

JDKで実際にパブリックAPIからメソッドが削除されたのは数えるほどですが、この変更はそのうちの1回となっています(かなり危険なThread.stopは、Java 1.1以来廃止予定となっているものの、結局のところ、まだ存在し続けています)。メソッドの削除は非常に珍しいことであり、そ

それでは、Java 9 (Jigsaw) EAリリースで何が変わったのかを見てみましょう。

[編集部注:スペースの関係上、重要でないファイル情報の一部はリストから削除しています。]

jimageファイルは、従来のようなzipベースの圧縮ではなく、クラスやリソースを高速に検索できるようにインデックスが付けられています。

jimageのコンテンツ領域には、そのイメージのすべてのクラスとリソースが含まれており、位置情報にひも付けて管理されています。

モジュールのパスをキーとしてインデックスを検索することによって、リソースの位置を特定できます。このパスは、/`<モジュール名>`/`<パッケージ>`/`<ベース名>`.`<拡張子>`という形式です。パスにはバージョンが付けられており、現在のパスは「9.0」です。この番号は現在のリリースのJVMを反映しています。

メインとなるjimageファイルは、lib/modulesにあるbootmodules.jimageというファイルです。しかし、このjimageファイルの他に、拡張子がjmodのファイルも存在します。それらのファイルがJavaモジュールです。Javaモジュール・ファイルは、アクセス速度を向上させるためのjimageには含まれないモジュール機能のテスト・プラットフォームとして使用されています。現在のところ、jmodファイルはまだzipファイルとして実装されていますが、将来的には、ほぼ間違いなく形式が変更されるでしょう。2つのパッケージ形式があることから、実際に移行対象となる現在のJava 9モジュールがどれほどであるがわかります。

java.base.jmodファイルには、一般的なJavaプログラムを実行できる、最低限のクラスの集合体が含まれています。実際に含まれているのは、次のパッケージ(またはその一部)です。

```
java.io
java.lang
java.math
java.net
java.nio
java.security
java.text
java.time
java.util
javax.crypto
javax.net
javax.security
```

その他のモジュールは、ベース・モジュールをサポートする実装クラスで構成されています。当然ながら、Project Jigsawの真の目的は、Java開発者もこのメカニズムを使用して、モジュール式の独自のライブラリやアプリケーションを作成できるようにすることです。そこで、次はシンプルなモ

ジュール式の「Hello World」を題材に、この世界をのぞいてみましょう。

シンプルなJavaモジュール:シンプルなJavaモジュールを作るには、2つのパートを記述する必要があります。最初のパートは、次のようなシンプルなコア・クラスです。

```
package com.jdk9ex;
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Modules!");
    }
}
```

Java 8以前では、他に何もしなくてもこのコードをコンパイルして実行できました。しかし、モジュール式のJavaでこのコードを正しく実行するためには、新しいメタデータを提供する必要があります。そのメタデータが含まれるのが、module-info.javaファイルです。

このファイルは、バージョンニングなどの重要なモジュールの特徴を管理する、かなり高度なメタデータ・プロバイダとなる予定です。しかし、現在のJigsaw EAで提供されているバージョンでは、熱心な開発者がモジュール技術を調査できることを想定した程度の非常に原始的なものとなっています。

次に示すのが、もっともシンプルなモジュールです(先ほどのJDK 9サンプル用のモジュールの名前がcom.jdk9exであると仮定しています)。

```
module com.jdk9ex { }
```



鋭い開発者の方なら、このコードを見てmoduleがJava 9での新しいキーワードであることに気付くことでしょう。このモジュールをコンパイルするためには、ディレクトリ構造を新しいモジュール式ソースのルールに従って配置する必要があります。今回は、com.jdk9exというモジュールであるため、次のような構造になります。

```
|____src
| |____com.jdk9ex
| | |____com
| | | |____jdk9ex
| | | | |____Main.java
| | |____module-info.java
```

モジュール名は、モジュール内にあるJavaパッケージの通常の命名規則とは関係ないことに注意してください。このモジュール・コードをコンパイルするためには、モジュールを認識したビルドが行われるように指定してJavacを実行する必要があります。

```
$ javac -d modules/com.jdk9ex \
    src/com.jdk9ex/module-info.java \
    src/com.jdk9ex/com/jdk9ex/*
```

このコンパイルによって、一連のモジュール式クラス・ファイルが生成されますが、jimageは生成されません。このコードを実行する場合も、モジュールが必要であることを明示的にJVMに伝える必要があります。

```
$ java -modulepath modules -m \
    com.jdk9ex/com.jdk9ex.Main
Hello Modules!
```

基本的な「Hello World」サンプルは、プログラマーにとっておなじみのパターンです。Java 9のモジュール化の現状を完全に理解したい開発者の方には、モジュール・システムを実際に触ってみて、どのように動作するかを確認することを強くお勧めします。

問題が起きる可能性: どんな新技術でも、早期に導入したユーザーに問題が起きることは避けられません。ここでは基本的な問題の事例をいくつか見てゆきます。特に、ビルドに失敗した場合や、モジュール式のJavaを正しく呼び出せなかった場合は、新しい種類の例外が登場します。

```
$ java -modulepath modules -m com.jdk9ex.Main
Error occurred during initialization of VM
java.lang.module.ResolutionException:
    Module com.jdk9ex.Main not found at
java.lang.module.Resolver.fail
    (java.base@9.0/Resolver.java:880)at
java.lang.module.Resolver.resolve
    (java.base@9.0/Resolver.java:193)at
java.lang.module.Resolver.resolve
    (java.base@9.0/Resolver.java:173)at
java.lang.module.Configuration.resolve
    (java.base@9.0/Configuration.java:229)at
jdk.internal.module.ModuleBootstrap.boot
    (java.base@9.0/ModuleBootstrap.java:174)at
java.lang.System.initPhase2
    (java.base@9.0/System.java:1242)
```

[編集部注:スペースの関係上、一部の行を折り返しています。]

この1つのスタック・トレースには、次のような新しい項目が表示されています。

- java.lang.moduleなどのパッケージ
 - ResolutionExceptionなどの例外
 - java.base@9.0/ModuleBootstrap.javaなどのコード上の位置
 - java.lang.System::initPhase2などの起動用エントリ・ポイント
- この内容から、jimages用の新しいモジュール・パスがスタック・トレースに直接存在するようになったこともわかります。

では、内部実装クラスにアクセスできるかどうか試してみましょう。次のコードは、Java 8でコンパイルして実行できます。




```
import sun.invoke.util.BytecodeName;

public class InternalsCheck {
    public static void main(String[] args) {
        String bName =
            BytecodeName
                .toBytecodeName("java.lang.Object");
        System.out.println(bName);
    }
}
```

ご想像のとおり、このコードはjava.lang.Objectの内部名を表示するものです。しかし、モジュールを使用する場合、実装クラスへのアクセスが制限されるため、このコードはコンパイルすらできなくなります。

```
$ javac -d modules/com.jdk9ex \
    src/com.jdk9ex/module-info.java \
    src/com.jdk9ex/com/jdk9ex/*

src/com.jdk9ex/com/jdk9ex/InternalsCheck.java:8:
  error: package sun.invoke.util does not exist
  import sun.invoke.util.BytecodeName;
                          ^
src/com.jdk9ex/com/jdk9ex/InternalsCheck.java:12:
  error: cannot find symbol
      String bName = BytecodeName.
toBytecodeName("java.lang.Object");
                  ^
    symbol:   variable BytecodeName
    location: class InternalsCheck
2 errors
```

この2つのエラーは、Javaモジュール・システムの目標の1つである、API整合性が強力に保証されていることを証明しています。現在のところ、javacは実装クラスBytecodeNameへのアクセスを防ぐようにアップグレードされていますが、IDEやJava環境のその他のツールはまだこの新たな現実を認識していません。これも、JDK 9が一般の開発者向けに公開されるまでの道のりはまだ長いことを示す印の1つです。

よい知らせなのは、すべての主要なIDEベンダーが積極的に新し

いバージョンの開発に取り組んでおり、Java 9を完全にサポートしようとしていることです。たとえば、OracleはNetBeans 9のJDK 9サポートに注力することを発表し、EclipseのJDTもすべての新機能の開発をJDK 9ブランチに向けています。

Unsafeについて: Javaプラットフォーム、特にOpenJDKプロジェクトに関わる開発者の立場から見れば、「ショットガン・プライバシー」は重大な問題です。javaパッケージやjavaxパッケージは「禁止パッケージ」としてSecurityExceptionメカニズムによって保護されていますが、この仕組みは内部実装クラスには適用されません。

Javaのライブラリやフレームワークでは、そういった内部実装の詳細に直接アクセスすることは想定されていません。Javaの幹事は、そのようなことをするのは危険な行為だと言いつけてきましたが、このルールを破って内部実装の詳細にリンクしようとする開発者に対応しようとはしてきませんでした。

警告が繰り返されてきたにもかかわらず、多くの有名なJavaライブラリがこのルールを破り、プラットフォームの内部仕様にアクセスしています。これによってJDK開発者たちは非常に難しい状況に追い込まれていることから、大きな問題となっています。

1つの選択肢は、内部仕様へのアクセスを禁止する変更は現状では非常に難しいと認め、いくつかの内部仕様を標準APIの一部に取り入れ、サポート対象に加えることです。もう1つの選択肢は、内部仕様を変更し、無数の本番アプリケーションが依存しているライブラリやフレームワークに影響を与えるか、または動かなくなってしまうことです。

もっとも有名で広く使用されており、もっとも危険だと言える直接アクセスの例が、sun.misc.Unsafeクラスです。このクラスには、モジュール・システム採用の圧力が大きくかけられています。JDK 9でUnsafeへのアクセスを単純に遮断してしまうと、ほぼすべてのJavaアプリケーションをアップグレードできなくなってしまうでしょう。一般的なアプリケーションは、ほぼ例外なく直接的または間接的にUnsafeの機能に依存したライブラリを

多くの有名なJavaライブラリがこのルールを破り、プラットフォームの内部仕様にアクセスしています。これによってJDK開発者たちは非常に難しい状況に追い込まれていることから、大きな問題となっています。



使用しているからです。

しかし、無制限にUnsafeへのアクセスというリスクを認めてしまうと、それが非公式な業界標準になってしまい、内部プラットフォーム・メカニズムが進化する可能性が奪われてしまいます。OracleとJavaコミュニティが合意した中間案は、Java 9では暫定的にアクセスを認めるものの、Java 10ではアクセス不可とする(そして、サポート対象に含まれる同等の標準機能と置き換える)というものです。

つまり、既存のフレームワークはそのままだ動作するため、Java開発者はJava 9にアップグレードしやすいということになります。ライブラリやフレームワークの開発者は、Java 10に間に合うように新しいメカニズムに移行する必要があります。その時期は、早くても2018年になる見込みです。

開発者はどう適応すべきか: Java開発者は、新機能を最大限に活用するために、コードのパッケージ化やデプロイの新しい方法に適応しなければならなくなるでしょう。しかし、よい知らせなのは、すぐに対応する必要はないことです。モジュール式での作業を採用する準備が整うまでは、JARファイルを使用する従来型の方式も継続されるでしょう。

シンプルなアプローチ(しかも、Java 9の登場前でも使用可能)の1つは、Java 8の一部であるCompactプロファイルを活用することです。Compactプロファイルは、モジュール化へ向けた第一歩であり、JREの真のサブセットである3つの個別のプロファイルが定義されています。もっとも小さなプロファイルであるcompact1は11MBほどであり、かなりの容量を節約できます。

完全なJREよりも小さなプロファイルで実行できるアプリケーションは、はるかに簡単にモジュールに対応できるでしょう。多くの場合、今のうちに先回りして作業しておけば、後でかなり楽になります。さらに、アプリケーションの依存性を理解して記録しておけば、技術的な負債を減らす上で貴重な作業になるでしょう。

Early Access版には予想されるすべての機能が含まれていますが、**現在進行中の作業を試す余地は十分にあります。**

まとめ

モジュールの到来は、Java環境にとって大きな変化となります。今までで最大の変化となると言っても過言ではないでしょう。Early Access版には予想されるすべての機能が含まれてはおりませんが、現在進行中の作業を試す余地は十分にあります。Java 9のリリースはまだ何か月も先であり、実際のテストでJava 9 EAを試した開発者のフィードバックも歓迎されています。興味をお持ちの開発者の方は、ぜひ[OpenJDK Adoption Group](#)をご覧ください。フィードバックの提供や、Java 9モジュールを推進するためにできることの発見には、理想的な場所です。

</article>

Ben Evans: London Java Communityの運営を助けるとともに、ユーザー・コミュニティの代表としてJavaの運営組織であるJCP Executive Committeeに投票権を持つメンバーとして参加。『The Well-Grounded Java Developer』、新版『Java in a Nutshell』、近刊『Optimizing Java』の著者。

learn more

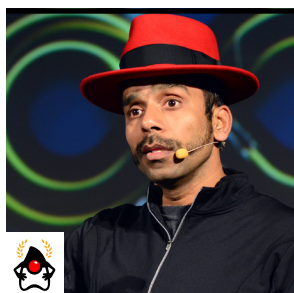
[JDK 9 Outreach](#)

[モジュールの使用 \(Devoxxのビデオ・プレゼンテーション\)](#)

[JDK 9のリリース・スケジュール](#)

[JDK 9のバージョン文字列](#)





ARUN GUPTA

複数のDockerコンテナの使用

Dockerコンテナのクラスタを組み合わせ、ハウスキーピングの手間がかからないJava EEアプリケーションを実行する

全2回シリーズの第1回の記事では、Dockerの基本概念について解説し、Toolboxを使用してDockerを開始する最初のステップ、初めてのDockerイメージの構築方法、JavaアプリケーションをDockerイメージとしてパッケージ化する方法、イメージをコンテナとして実行する方法について説明しました。そして最後に、簡単なJava EEアプリケーションをWildFlyアプリケーション・サーバーにデプロイしました。

本記事では、複数のホスト上に複数のコンテナを持つアプリケーションをデプロイする方法について説明します。具体的には、WildFlyにデプロイしたJava EEアプリケーションからデータベース（今回はCouchbase）に対するCRUD操作を行ってみます。さらに、マルチコンテナ・アプリケーションを作成するDocker Composeを実際に操作し、マルチホスト・アプリケーション環境を作成するDocker Swarmについても説明します。

Docker Compose

Docker Composeは、マルチコンテナDockerアプリケーションの定義や実行を担うツールで、本シリーズの第1回で説明したように、Docker Toolboxとともにインストールされます。

通常、Dockerコンテナを使用するアプリケーションは、複数のコンテナで構成されます。たとえば、アプリケーション・サーバー用に1つのコンテナ、データベース用にも1つのコンテナ、場合によってはメッセージング・サーバー用にもう1つ、などという具合です。各コンテナは、さまざまな構成オプション（ポート・フォワーディング、環境変数、ボリューム・マッピングなど）を指定して起動するのが一般的です。

Docker Composeを使用した場合、コンテナを起動するためのシェルスクリプトを書く必要はなくなります。すべてのコンテナは、YAML構成ファイルでサービスを使用して定義します。各サービスは、1つまたは複数のコンテナで構成されます。アプリケーションに加え、アプリケー

ション内のすべてのサービス、各サービス内にあるすべてのコンテナの起動、停止、再起動には、Docker Composeスクリプトを使用します。このアプローチでは、すべてのオプションをあらかじめファイルで指定しておくことができるため、大きなサービス内の単一のコンテナを起動する場合に便利です。

次のコマンドは、コンテナのポート8091（管理ポート）、ポート8093（クエリー・ポート）などのポートをホストの同じポートに転送し、コンテナのCouchbaseログ・ディレクトリをホストのディレクトリにマッピングしたCouchbaseサーバーを起動します。

```
docker run -d -v ~/couchbase/:/opt/couchbase/var
-p 8091:8091 -p 8092:8092 -p 8093:8093 -p
11210:11210 couchbase/server
```

[編集注：上記のコマンドは1行で入力します]

Docker Composeファイルでコマンドを定義した場合、このコンテナを起動するオプションをすべて覚えておく必要はなくなります。デフォルトでは、このファイルはカレント・ディレクトリのdocker-compose.ymlという名前になっています。先ほどのコマンドと同等の働きをするファイルは次のようになります。

```
mycouchbase:
  image: couchbase/server
  volumes:
    - ~/couchbase:/opt/couchbase/var
  ports:
    - 8091:8091
    - 8092:8092
    - 8093:8093
```



42

- **スケジューラ戦略:** コンテナを実行する最適なノードを決定するために適用できるもので、spread (デフォルト)、binpack、randomから選択できます。デフォルトの戦略では、実行するコンテナの数が最小になるようにノードが最適化されます。
- **ノード検出サービス:** Swarmマネージャは、ホストされている検出サービスと通信します。検出サービスは、Swarmクラスタ内のIPアドレスの一覧を保持しています。Docker/ハブでは、開発用の検出サービスがホストされています。本番環境では、デフォルトの検出サービスをetcd、Consul、ZooKeeperなどの別のサービスで置き換えます。静的なファイルを使用することも可能です。
- **標準Docker API:** Docker Swarmは、標準Docker APIに対応しています。そのため、単一のDockerホストと通信するツールであればどのようなものでも、複数のホストにシームレスに拡張できます。したがって、Docker CLIで複数のDockerホストを構成するシェル・スクリプトを使用していた場合、同じCLIでDocker Swarmクラスタと通信できます。

Couchbaseを使用するマルチコンテナJavaアプリケーションを複数のホストにまたがるDocker Swarmクラスタにデプロイするためには、次の手順が必要になります。

1. 検出サービスの作成
 2. Docker Swarmクラスタの作成
 3. アプリケーション環境の起動
 4. Couchbaseサーバーの構成とサンプル・データのロード
 5. Docker Swarmクラスタへのアプリケーションのデプロイ
- デプロイされたアプリケーションのノード全体への配布は、Docker Swarmが行います。

図2は、アプリケーション内の各コンポーネントの概念図です。

- クラスタには2つのノードがあります。1つはマスター、もう1つはワーカー・ノードです。
- クラスタでアプリケーション環境を起動するために、Docker Composeを使用します。
- 検出サービスは、クラスタの外部に存在する別のマシンにホストされています。

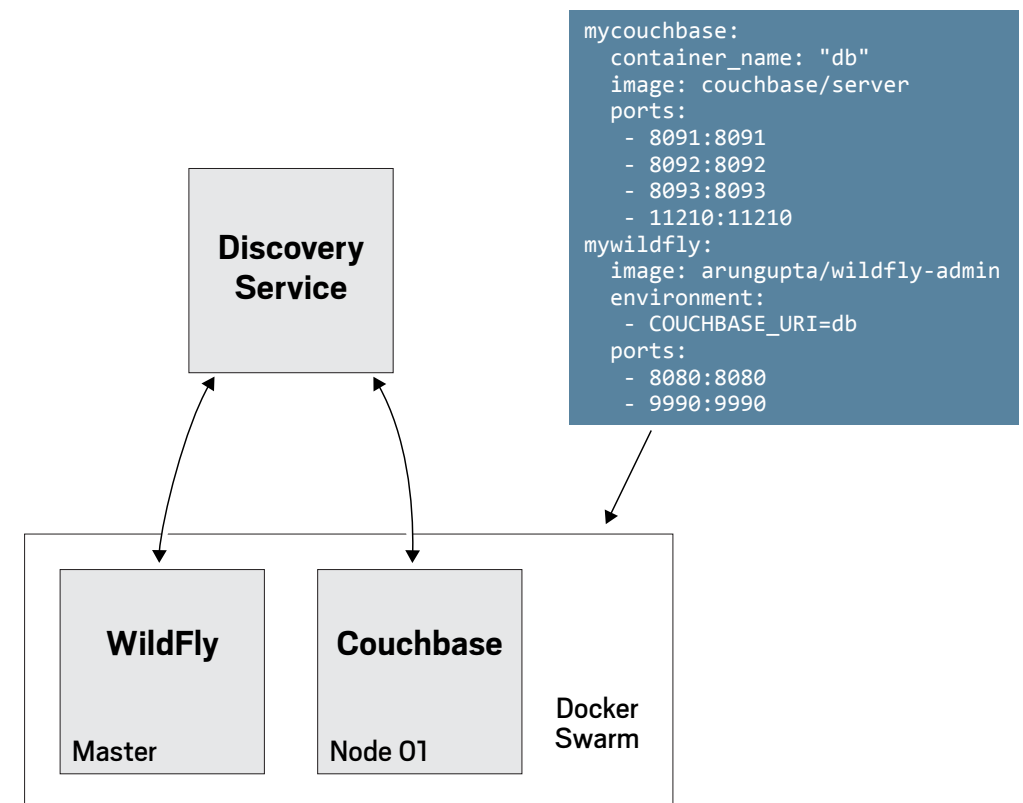


図2: Docker Swarmで実行されるシンプルなサービス

Couchbaseの構成とWildFlyへのアプリケーションのデプロイには、Mavenを使用します。**注:** 本記事では、WildFlyとCouchbaseを使用していますが、Tomcat、GlassFishなどのアプリケーション・サーバーや、MySQL、JavaDBなどのデータベース・サーバーも簡単に使用できます。通常、イメージとそれに含まれるアプリケーションは適切に更新する必要があります。

それでは、この構成の詳しい設定方法について見てゆきましょう。

検出サービスの作成

この例では、検出サービスにConsulを使用しますが、etcdやZooKeeperなどの同様のツールを使用することもできます。

まず、検出サービスをホストするDockerマシンを新しく作成します。



NAME	ACTIVE	DRIVER	STATE	URL	SWARM
consul-machine	-	virtualbox	Running	tcp://192.168.99.100:2376	
swarm-master	*	virtualbox	Running	tcp://192.168.99.101:2376	swarm-master (master)
swarm-node-01	-	virtualbox	Running	tcp://192.168.99.102:2376	swarm-master

表2: サンプルのDocker Swarmの内容

```
eval "$(docker-machine env --swarm swarm-master)"
docker info
```

Swarmクラスタに接続するために、--swarmを指定している点に注意してください。このオプションを指定しない場合、swarm-masterマシンのみ

に接続されます。
docker infoコマンドを実行すると、各ノードについての詳しい情報と、各ノードで実行されているコンテナ数が出力されます。出力の最初の部分には、次のような概要が表示されます。

```
docker info
Containers: 3
Images: 2
Role: primary
Strategy: spread
Filters: health, port,
        dependency, affinity, constraint
```

クラスタには、ノードが2つ (Swarmマスターが1つ、Swarmワーカー・ノードが1つ) あり、3つのコンテナ (マスターとノードにSwarmエージェントが1つずつと、マスター上で実行されているswarm-agent-master) が実行されています。

アプリケーション環境の起動

以上でクラスタの準備が整い、アプリケーションをデプロイできる状態になりました。今回WildFlyにデプロイするJava EEアプリケーションは、Couchbaseのデータに対してCRUD/RESTインタフェースを提供するものです。

アプリケーション環境の起動には、先ほど紹介したComposeファイルを使用しますが、その前にクラスタのネットワークの状態を見てみましょう。

デフォルトで、Dockerでは1つのホストにつき、表3に示す3つのネットワークが作成されます。

Dockerでは、ブリッジ・ネットワークとオーバーレイ・ネットワークを作成できます。ブリッジ・ネットワークは単一のホストを、オーバーレイ・ネットワークは複数のホストをカバーします。Docker Composeアプリケーションを起動する際に--x-networkingスイッチを指定すると、単一のホストに対してはブリッジ・ネットワークが、Swarmクラスタに対してはオーバーレイ・ネットワークが作成されます。

次のコマンドを実行し、クラスタに接続されていることを確認します。

```
eval "$(docker-machine env --swarm swarm-master)"
```

WildFlyとCouchbaseの起動には、前述のComposeファイルを使用します。WildFlyサービスでは、COUCHBASE_URI環境変数をdbに設定しています。dbはCouchbaseコンテナの名前です。次のコマンドで、アプリケーションを起動します。

ネットワーク名	目的
bridge	コンテナの接続先となるデフォルトのネットワーク
none	コンテナ固有のネットワーク・スタック
host	ホストのネットワーク・スタックにコンテナを追加

表3: Dockerで作成される3つのネットワーク



```
docker-compose --x-networking up -d
```

アプリケーション固有のオーバーレイ・ネットワークが作成されます。次のコマンドで、結果を確認できます。

```
docker network ls
```

Swarmノード1つにつき、3つのネットワークが表示されます。それらのネットワークに加えて、コンテナがクラスタ外部に接続するために、wildflycouchbasejavaee7というオーバーレイ・ネットワークと、docker_gwbridgeというブリッジ・ネットワークが作成されます。

docker psコマンドを使用すると、WildFlyコンテナとCouchbaseコンテナを確認できます。コマンドの出力(掲載は省略します)により、swarm-masterマシン上でCouchbaseサーバーが、swarm-node-01マシン上でWildFlyが実行されていることがわかります。

Couchbaseサーバーの構成とサンプル・データの導入

CouchbaseサーバーがDockerコンテナとして起動する際に、データの格納に使用できるよう構成する必要があります。ちょうどよいことに、Couchbaseサーバーにはその目的に使用できるREST APIがあります。このAPIを使用してサンプル・データをアップロードすることも可能です。

次に示すように、gitを使用してオンライン・アプリケーションをクローンし、Mavenを使ってインストールします。このアプリケーションには、REST APIでCouchbaseサーバーを構成できるMavenプロファイルがあります。

```
git clone https://github.com/
    arun-gupta/couchbase-javaee.git
mvn install -Pcouchbase -Ddocker.host=
    $(docker-machine ip swarm-master)
```

このスニペットで、\$(docker-machine ip swarm-master)は先ほど作成したSwarmマスター・マシンのIPアドレスになります。\$(docker-machine ip swarm-master)と記述する代わりに、実際のIPアドレス(docker-machine ip swarm-masterを実行すると、実際のIPアドレスがわかります)を使用することもできます。

Docker Swarmクラスタへのアプリケーションのデプロイ

アプリケーションをデプロイするためには、WildFlyが実行されているホストのIPアドレスと、WildFlyの管理レلم・ユーザーのユーザー名/パスワードを指定する必要があります。今回の例では、IPアドレスはswarm-node-01マシンのIPアドレスで、ユーザー名/パスワードの値はarungupta/wildfly-adminイメージのビルドに使用したものです。

正確なコマンド(1行で入力する必要があります)は、次のとおりです。

```
mvn install -Pwildfly -Dwildfly.hostname=
    $(docker-machine ip swarm-node-01)
    -Dwildfly.username=admin
    -Dwildfly.password=Admin#007
```

\$(docker-machine ip swarm-node-01)でWildFlyが実行されているマシンのIPアドレスを取得して使用することもできます。

アプリケーションへのアクセス

以上でWildFlyサービスとCouchbaseサービスが起動しました。いよいよアプリケーションにアクセスしてみます。アクセスするためには、WildFlyが実行されているマシンのIPアドレスを指定する必要があります。アプリケーションへのアクセスは、おなじみのコマンドライン・ユーティリティcurlを使用するのが簡単です。

```
curl http://$(docker-machine ip swarm-node-01):8080/
    couchbase-javaee/resources/airline
```

[編集注:このコマンドは1行で入力する必要があります]コマンドを実行すると、先ほどCouchbaseで構成したサンプル・データである10社の航空会社の一覧が表示されます。

または、docker-machine ip swarm-node-01コマンドでWildFlyのIPアドレスを取得し、ブラウザでアプリケーションにアクセスしても構いません。

このアプリケーションは、Couchbaseサンプル・データのリソースに対するGET、PUT、POST、DELETEの各APIをサポートしています。このアプリケーションのREST APIの完全な一覧は、[オンライン・ドキュメント](#)に掲載されています。



まとめ

DockerにJavaアプリケーションをデプロイする方法を全2回のシリーズで説明してきました。第1回ではDockerやDocker Toolboxの基本概念について紹介し、イメージのビルド方法やコンテナの実行方法について説明しました。こういった基本は、ほぼすべてのDockerプロジェクトに応用できます。

第2回となる本記事では、複数のホストにデプロイできるマルチコンテナ・アプリケーションの構築に役立つDocker ComposeとDocker Swarmを紹介しました。また、説明の中で、異なるアプリケーションが分離された環境を作成できるDockerのネットワーク設定についても触れました。本記事によって、複数のDockerコンテナを使用するアプリケーションの構築を始めるために必要なものが得られたはずです。

Dockerは、Javaの「Write once, run anywhere」(一度書けばどこでも実行できる)という原則をうまく補完しています。なぜなら、Dockerは「Package once, deploy anywhere」(一度パッケージ化すればどこでもデプロイできる)を実現しているからです。ここ数年でDockerを中心とする巨大なエコシステムが構築されてきたのは、Dockerがシンプルであるがゆえです。ぜひDockerに触れてみてください。そうすれば、DockerでJavaアプリケーションをパッケージ化する楽しさがわかることでしょう。

</article>

Arun Gupta (@arungupta) : Couchbaseの開発者支援部門担当バイス・プレジデント。Java Championで、Javaユーザー・グループのリーダーも務める。かつてのJava EEチーム設立時のメンバー。

learn more

Dockerを試してみる



TILEN FAGANEL,
MATJAZ B. JURIC

KumuluzEE:Java EEでの マイクロサービスの構築

オープンソースのKumuluzEEフレームワークを使用して、標準Java EE APIで自己完結型マイクロサービスを開発する

本記事では、標準Java EE APIでマイクロサービスを開発する方法について説明します。マイクロサービスは、個々のプロセスとしてデプロイされる独立した小さなサービスにより構成される複雑なアプリケーションを開発するための、有力なアーキテクチャ・スタイルになりました。マイクロサービスは、詳細に定義されたAPI経由で通信します。各マイクロサービスは自己完結型で、単一のタスクを担当し、RESTなどの軽量API経由でアクセスでき、独自のライフサイクルを備えています。マイクロサービスが行き着くところは、複合型アプリケーション向けに高度に分離された再利用可能なモジュール式アーキテクチャです。

Javaでマイクロサービスを開発するということは、アプリケーションをRESTサービス中心に設計するだけでなく、各サービス、フロントエンド、その他のコンポーネントを個別に、かつ独立的にデプロイすることにもなります。言い換えれば、マイクロサービスでは、すべてのサービス、ビジネス・ロジック、フロントエンド、その他のモジュールを1つのEARファイルにまとめてモノリシック・アプリケーションとしてデプロイするのではなく、コンポーネントごとに別々のアーカイブ(JARファイルなど)を使用します。[KumuluzEEフレームワーク](#)によって、後述のとおり、Java EEで稼働するマイクロサービスの構成とデプロイに関連するタスクが自動化されます。**[注:KumuluzEEは2015年のJavaOneカンファレンスでDuke's Choice Awardを受賞しました]**

マイクロサービスとJava EE

鉄道乗車券のオンライン予約サービスの作成を考えてみます。このサービスを使用し、乗車する路線の計画を立て、乗車券を購入できるというものです。説明を簡潔にするため、本記事ではこの例の単純なバージョンを

見ていきます。このバージョンでは、単純なユーザー・インタフェースを使用して、路線の参照と単一路線の予約のみを実行できるものとします。

従来のアプローチを使用する場合、ビジネス・ロジック、サービスを含み、さらにフロントエンドを格納した1つ以上のWARパッケージも含むモノリシックなEARパッケージを作成することになるでしょう。そのEARファイルを、Oracle WebLogic ServerやGlassFishなどのアプリケーション・サーバーにデプロイすることになります。

それに対して、マイクロサービス・アーキテクチャを使用する場合は、まず責務を切り分けることから始めます。「路線」、「予約サービス」、「UI」を、マイクロサービスとしてそれぞれ個別に開発、構成、デプロイすることになります。これらのマイクロサービスはステートレスであり、RESTインタフェース経由で通信します（ただし、通信のためにSOAPやRemote Method Invocation (RMI) を使用しても構いません）。このようにして、ドメインごとに定義され、明示的なインタフェースに関して「単一の責務」の原則に従うマイクロサービスを作成します。これで、分離された高度なモジュール式アーキテクチャができました。このアーキテクチャでは、各サービスが専門的な単一の機能について責務を担います。また、各マイクロサービスには、それぞれ独自のライフサイクルがあります。本記事での、マイクロサービスを使用するアーキテクチャの案を図1に示します。

以降では、この単純な例を使用しますが、マイクロサービスは本来、複雑なアプリケーションやシステムにもっとも適していることに注意してください。

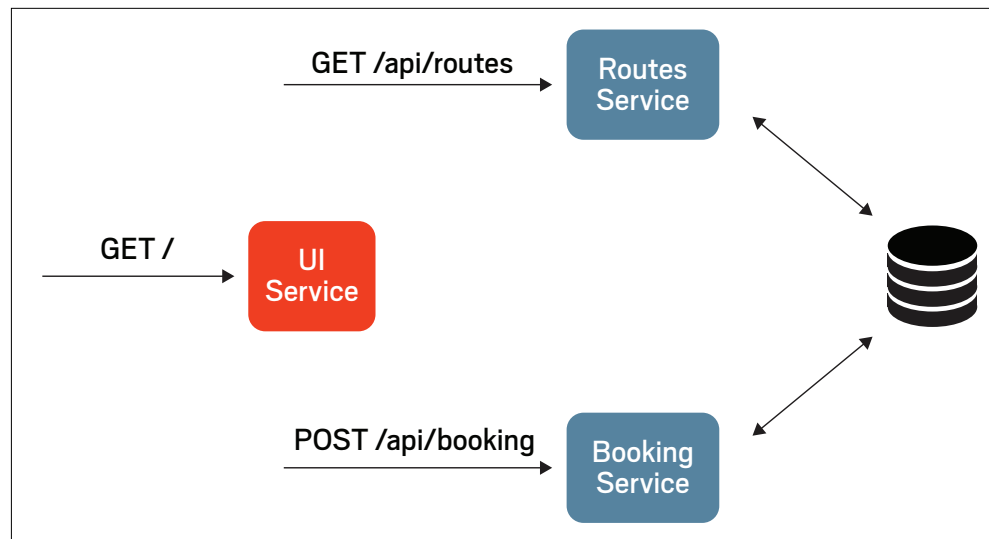


図1: マイクロサービス・アーキテクチャの案の概要

利点:モノリシックなアプローチと比較したマイクロサービスの利点について、詳しく見ていきましょう。明らかな利点は、マイクロサービスによって、より柔軟で分離されたアーキテクチャになることです。各マイクロサービスを個別に開発できるため、ライフサイクルや変更の管理が容易になります。たとえば、前述の予約サービスをアップグレードする必要がある場合に、アプリケーション全体の再デプロイは不要です。すべてのマイクロサービスは、個別のリポジトリとデプロイメント構成を持つ別々のプロジェクトとなります。新機能を反復的に展開できるため、俊敏性が向上します。

長期の運用後に、新しいJavaのバージョンにアップグレードしようとなった場合に、各マイクロサービスを個別にアップグレードできます。複雑なアプリケーションの場合、これは大きなメリットになります。一般に、複雑なJava EEアプリケーションのアップグレード(アプリケーション・サーバーの新バージョンやJava EEおよびJava SEの新リリースへのアップグレード)には困難が伴うためです。さらに良いことに、マイクロサービスでは新技術を段階的に取り込むことができます。

また、これも重要なことですが、マイクロサービスによってスケーラビリティが格段に向上します。たとえば、路線計算の方が予約よりも使用頻度が高く、路線計算サービスのトラフィック量が予約サービスを上回る場合に、マイクロサービスを使用すれば、予約サービスのスケーリングをその他のアプリケーション要素から独立して行うことができます。このアプローチは、クラウド環境およびPaaS (Platform as a Service) 環境に最適で

す。これらの環境では、柔軟なスケーラビリティを容易に構成できます。Docker環境内では、マイクロサービス・アプリケーションのスケーリングに手間がかかりません。つまり、マイクロサービス・アーキテクチャは、クラウド対応型アプリケーションと相性が良いと言えます。

これらの利点を存分に活用するためには、マイクロサービスをステートレスにする必要があります。マイクロサービスが使用するすべてのリソース(データベース、オブジェクト・ストレージなど)は、個別に構成できるものである必要があります(一般に、接続文字列または環境パラメータによって個別に構成します)。

欠点:マイクロサービスには、避けようがない欠点があります。通常、この種のアーキテクチャを伴うJava EEプロジェクトでは、セットアップや構成が面倒です。実際、手動による構成や依存性追跡のための多くの作業が必要で、結果的に運用が大幅に複雑化することもあります。デプロイメントもテストも複雑になります。

KumuluzEE

KumuluzEEはこれらの欠点の一部を解決します。マイクロサービスの構成とデプロイに関連するタスクがシームレスに自動化されます。基本的には、KumuluzEEは各マイクロサービスとJava EE APIのランタイムを単純なスタンドアロン・パッケージ(JARファイル)に集めます。この際に、実際に使用されるAPI(とランタイム)のみを追加することで、フットプリントを最小限に抑えます。そのため、開発者はスタンドアロンのステートレスな自己完結型マイクロサービスを構築して、各マイクロサービスにJava EEアプリケーション・サーバー全体を含めるオーバーヘッドを避け、効率的にそれらのマイクロサービスをパッケージ化できます。この方法で作成されたマイクロサービスはJREから直接実行でき、フットプリントは最小限で、起動とシャットダウンの時間が短くなります。

例について

これから、KumuluzEE、標準Java EE、Mavenを使用してマイクロサービスを作成します。まず、それぞれ独自のマイクロサービスが格納された、3つのMavenプロジェクトを作成します。説明を簡潔にするため、本記事ではこれら3つのプロジェクトを同一リポジトリ内に作成します。作成するのは、路線計算サービス用のroutesプロジェクト、予約サービス用のbookingsプロジェクト、そしてフロントエンド用のuiプロジェクトです。ま



動作確認のために、maven packageを実行します。その後、以下のコマンドでこのマイクロサービスを起動できます。

ブラウザで<http://localhost:8080/>にアクセスすると、index.xhtml HTMLファイルの内容が表示されます。もちろん、JSPやサーブレットを使用することや、JSFサポートを追加してフロントエンドとして使用することも可能です。後ほど、このフロントエンドを拡張します。

前項の手順を各マイクロサービスに対して実行します。各マイクロサービスはそれぞれ独自のプロジェクトであるため、必要なだけカスタマイズできます。

路線サービスと予約サービスの開発を始める前に、先にJPAモジュールとエンティティを定義しましょう。JPAを含めるために、[EclipseLink JPA](#) 実装を追加します。また、データベース・ドライバも追加します。この例では、PostgreSQLデータベースを使用しますが、実際には任意のデータベースを使用できます。./models/pom.xmlに、以下のように必要な依存性を記述します。

次に、2つのマイクロサービスが共有するpersistence.xmlファイルとエ

次に、エンティティ・クラスを作成します。この例では標準JPAを使用します。標準JPAを使用する場合、KumuluzEEを導入するための変更は必要ありません。RouteおよびBookingのJPAエンティティ・クラスの例を次に示します。1つ目のファイル.../models/Route.javaの内容は以下のとおりです。

}



次に、予約用のRESTサービスを実装します。ここでは、`createBooking(Booking)`と`getBooking()`という2つのメソッドを含む予約リソース(`BookingsResource`)を作成します。前述の説明に従って、適切な依存性を追加してください。また、RESTサービスが使用されるすべてのマイクロサービスに対して、JPAエンティティにアクセスできるように、`models`モジュールを追加してください。コードは以下のようになります。

```
@ApplicationPath("/")
public class BookingApplication extends
    javax.ws.rs.core.Application {
}
```

```
@Path("/bookings")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@RequestScoped
public class BookingsResource {
    @PersistenceContext(unitName = "trains")
    private EntityManager em;
```

...



ご覧のとおり、KumuluzEEを使用すればマイクロサービスの実装を変更する必要はなく、他のJava EEアプリケーションとまったく同じ実装になります。残りのマイクロサービスも同じように実装するか、ダウンロードしたサンプル・コードを参照してください。

この例では、フロントエンドUIモジュールにJavaServer Faces (JSF) を使用することにします。JSFからREST経由でマイクロサービスを呼び出して、実際のデータの受信や保存を行います。ここでもっとも重要な問題は、マイクロサービスの正確なURLを把握する必要があります。環境変数を使用してURLを指定することもできますが、その場合、URLが変更されるたびに手動で更新しなければなりません。クラウド環境では、URLの変更がかなり頻繁に起こる可能性があります。

もっと便利な方法は、リクエストするマイクロサービスのアドレスを動的に問い合わせるというものです。この例ではサービス検出のために [Apache ZooKeeper](#) を使用しますが、他の類似ツールを使用しても構いません。ZooKeeper について簡単に説明すると、構成情報を管理するための一元化されたサービスです。ここでは、ZooKeeper を使用して各マイクロサービスに正しい URL を保存します。例では、ヘルパー・クラスの [ServiceRegistry](#) を `utils` モジュールに追加し、このクラスで ZooKeeper を使用してエンドポイントの動的な登録、登録解除、取得の各処理を行います。

```

public ZooKeeperServiceRegistry()
    throws IOException {
    try {
        String zookeeperUri =
            System.getenv("ZOOKEEPER_URI");
        zookeeper = CuratorFrameworkFactory
            .newClient(zookeeperUri,
                new RetryNTimes(5, 1000));
        zookeeper.start();
        zonePaths = new ConcurrentHashMap<>();
    } catch (IOException) {
        ...
    }
}

public void registerService(
    String name, String uri) {
    try {
        String node = "/services/" + name;
        if (zookeeper.checkExists()
            .forPath(node) == null) {
            zookeeper.create()
                .creatingParentsIfNeeded()
                .forPath(node);
        }
        String nodePath =
            zookeeper.create().withMode(
                CreateMode.EPHEMERAL_SEQUENTIAL)
                .forPath(node + "/" + uri.getBytes());
        zonePaths.put(uri, nodePath);
    } catch (Exception ex) {
        ...
    }
}

// ...Similar for the other methods.
// See full example available for download.
}

```



ZOOKEEPER_URL環境変数にZooKeeperのURLを設定する必要があります。この例ではDockerを使用して、マイクロサービスと共にZooKeeperサービスを起動します。その方法については次項で説明します。すべてのマイクロサービスにutilsパッケージへの依存性を追加することも忘れないでください。

次に、マイクロサービスに前述のServiceRegistryサービスをインジェクションして、起動時にURLを登録し、シャットダウン時にURLの登録を解除するようにします。その方法の1つとして、bookingマイクロサービスに、このタスクを動的に処理するためのBeanを追加します。

```
@ApplicationScoped
public class BookingService {

    @Inject
    ServiceRegistry services;

    private String serviceName = "trains-booking";
    private String endpointURI;

    public BookingService() {
        endpointURI =
            System.getenv("BASE_URI");
    }

    @PostConstruct
    public void registerService() {
        services.registerService(
            serviceName, endpointURI);
    }

    @PreDestroy
    public void unregisterService() {
        services.unregisterService(
            serviceName, endpointURI);
    }
}
```

@PostConstruct、@PreDestroy、@ApplicationScopedの各アノテーションを使用して、ライフサイクルの中で一度だけサービスの登録と登録解除を行うようにしています。BASE_URI環境変数に、マイクロサービスのパブリックURIを設定する必要があります。さらに、それぞれのマイクロサービスに、同様のBeanを作成してください。

フロントエンドへのマイクロサービスのインジェクション

フロントエンドからRESTサービスに接続するために、前項のインジェクションの手法をフロントエンドでも使用できます。以下のようにServiceDiscovery BeanをJSFバックングBeanにインジェクションして、呼び出し先のマイクロサービスのURLを取得します。

```
@Model
public class BookingsBean {

    @Inject
    ServiceRegistry services;

    public List getAllBookings() {
        return ClientBuilder.newClient()
            .target(services.discoverServiceURI(
                "trains-booking"))
            .path("bookings")
            .request().get(List.class);
    }
}
```



ご覧のとおり、Dockerfileはかなり単純でわかりやすい内容です。必要な依存先ファイルをインストールし、マイクロサービスをビルドし、さらにマイクロサービスを実行するためのコマンドを指定しています。本記事の例の場合、このDockerfileをすべてのマイクロサービスに対して使用できます。ただし最終行のコマンドでは、個々のDockerコンテナで実行するマイクロサービスを指定してください。


```
$ docker build -t trains/ui \
  -f ui/Dockerfile .
$ docker build -t trains/routes \
  -f routes/Dockerfile .
$ docker build -t trains/bookings \
  -f bookings/Dockerfile .
```

ビルド後のイメージは、DockerまたはDocker APIが使用される場所に加えて、Docker Swarm、Kubernetes、各種PaaSプロバイダ、および同様のサービスが使用される場所でも実行できます。イメージをテストするために、ローカルで実行できます。また、既存のイメージを使用してZooKeeperインスタンスを起動する必要もあります。忘れずに、マイクロサービスに対して必要な環境変数を渡してください(-eスイッチを使用します)。以下の例では、デフォルトの172.17.42.1をDockerホストとして使用します。必要に応じて、このアドレスを変更してください。

```
$ docker run -p 2181:2181 -d fabric8/zookeeper
$ docker run --name ui -p 3000:8080 -d \
  -e BASE_URI='http://172.17.42.1:3000' \
  -e ZOOKEEPER_URI=172.17.42.1:2181 trains/ui
$ docker run --name routes -p 3001:8080 -d \
  -e BASE_URI='http://172.17.42.1:3001' \
  -e ZOOKEEPER_URI=172.17.42.1:2181 trains/routes
$ docker run --name bookings -p 3002:8080 -d \
  -e BASE_URI='http://172.17.42.1:3002' \
  -e ZOOKEEPER_URI=172.17.42.1:2181 trains/bookings
```

これで、今回作成したマイクロサービス・アプリケーションにはhttp://localhost:3000からアクセスでき、またroutesモジュールとbookingsモジュールのRESTサービスには、それぞれhttp://localhost:3001とhttp://localhost:3002からアクセスできるようになります。

まとめ

本記事では、オープンソースのKumuluzEEフレームワークを使用して、標準Java EEでマイクロサービスを開発する方法について確認しました。これまで説明したとおり、KumuluzEEフレームワークによって、マイクロサービスの構成とデプロイに関連するタスクが自動化されます。Mavenでのシンプルな依存性定義により、必要な実行環境が含まれる自己完結

型のスタンドアロンJARファイルを作成できます。そのため、アプリケーション・サーバーを必要とせずに、標準JRE内で実行できます。このように、KumuluzEEを使用したマイクロサービスは最小限のフットプリントとなるため、クラウド、PaaS、Dockerスタイルの環境での実行に適しています。本記事の例では、Dockerスタイルの環境を使用しました。また、本記事では、ZooKeeperを使用したサービス検出の処理方法も確認しました。

KumuluzEEは、Java EEでのマイクロサービス・アーキテクチャ構成を可能にするツールです。その主な利点は、標準Java EE APIを使用してマイクロサービスを開発できることにあります。既存のアプリケーションをマイクロサービスに移植することも可能です。一見すると、マイクロサービスは従来のアプリケーション・サーバーを利用する開発者に奇妙に映るかもしれませんが、多くの技術者が、マイクロサービスがクラウドでのJavaの未来像であると考えています。どうぞ気軽に[KumuluzEE](#)をダウンロードして試してみてください。</article>

Tilen Faganel: Sunesisのリード・ソフトウェア・アーキテクト。Java用のKumuluzEEフレームワークのリード開発者。同フレームワークは、2015年のDuke's Choice Awardの最優秀Javaイノベーション部門で受賞を果たしている。

Matjaz B. Juric: PhD。Java Champion、Oracle ACE Director。Java、BPM、SOAに関する15冊以上の書籍の著者(または共著者)。複数の雑誌や論文にも寄稿している。

learn more

[Getting Started with KumuluzEE](#)



クイズに挑戦

Java認定試験の作者による出題

SIMON ROBERTS

[編集注：このセクションを拡充してさらに役立つものとするために、たくさんの方の認定試験を作成している Simon Roberts 氏に問題と解答を作成していただきました。以前のコラムの内容とは異なり、正解の理由について詳しく掘り下げた解説をしていただきます。この新しい形式を気に入っていただけると幸いです。ご意見、ご感想をお待ちしています]

今回の問題は、1Z0-809 Programmer II試験と同等の難易度を想定しています。出題には、少なくとも2つの目的があります。1点目は、多くの読者に新しいことを知っていただき、Java言語の微妙な側面を明確にさせていただくことです。2点目は、言うまでもなく、試験問題の全体像や難易度、解答方法などについて理解させていただくことです。

設問 1 次のコードについて：

```
public class Outer {
    private int x = 99;

    private class Inner {
        private int x = 100;

        public void show() {
            System.out.println(x); } // line n1
        public void show0() {
            System.out.println(Outer.this.x); } // line n2
    }

    public void show(Inner i) {
        System.out.println(i.x); } // line n3
    public void show() {
        System.out.println(x); } // line n4

    public static void main(String [] args) {
```

```
Outer o = new Outer();
Inner i = o.new Inner(); // line n5
o.show(); o.show(i);
i.show(); i.show0();
}
```

次のそれぞれについて、正しいものはどれですか。2つ選んでください。

- a. line n2 の式 Outer.this.x の値は 99 である。
- b. line n2 の Outer.this.x は、Inner クラスが Outer の x にアクセスできないため、失敗する。
- c. line n2 の式 Outer.this.x は構文エラーであり、super.x と読み替えるべきである。
- d. line n3 の式 i.x の値は 100 である。
- e. line n3 での i.x へのアクセスは、Outer クラスから Inner の x にアクセスできないため、失敗する。

設問 2 次のインタフェースについて：

```
public interface Clothing {
    int getSize();
    String getColor();
}
```

次のクラスを **Clothing** アイテムのジェネリック型コンテナとして使用するには、どのような変更が必要ですか。

```
public class Pair {           // line n1
    private E left, right;    // line n2

    public Pair(E left, E right) {
        this.left = left;
        this.right = right;
    }
}
```

```
public boolean isMatched() {
    return left != null && right != null
        && left.getSize() == right.getSize()
        && left.getColor().equals(right.getColor());
}
```

- a. 型 E を Object に変更する。
- b. line n1 を次のように変更する。

```
public class Pair<? extends Clothing> {
```
- c. line n1 を次のように変更する。

```
public class Pair<E super Clothing> {
```
- d. line n1 を次のように変更する。

```
public class Pair<E extends Clothing> {
```
- e. line n2 の変数宣言を次のように変更する。

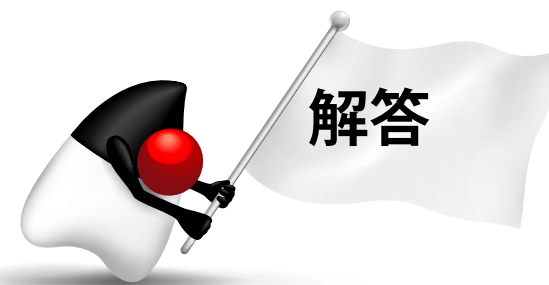
```
private Clothing left, right;
```

設問 3 次のラムダ式で、ジェネリック引数が適切に定義されているとします。

```
s->s.length()
```

length() メソッドが int を返す場合、このラムダ式はどの標準関数インタフェースに割り当てることができますか。2つ選んでください。

- a. Function
- b. IntFunction
- c. IntSupplier
- d. ToIntFunction
- e. UnaryOperator



設問 1 正解は選択肢 A と D です。この問題は、`private` のアクセス可否の意味がテーマとなっています。`private` の意味を言葉にすると、「`private` なものは、同じクラスの中からだけアクセスできる」というような表現になることがよくあります。ただし、厳密に言えばこれは正しくなく、この問題はその点に着目しています。

実際は、`private` な要素はトップレベルの波括弧で囲まれた中であれば、どこからでも参照できます。単純なケースであれば、この2つの表現は同じ意味になることがあります。しかし、ネスト・クラス（インナー・クラス）が存在する場合は、内部クラスと外部クラスの `private` 要素はどちらも両方のクラスから参照できます。『Java 言語仕様』の[セクション 6.6.1](#)には、「`private` と宣言されると、その宣言を囲んでいるトップ・レベルのクラスの本体内に現れる場合に限りアクセスが許可される」と記載されています。

Outer.this.x という構文は、明示的に外側のインスタンスのフィールド x にアクセスする際の正しい記述です。もちろん、今回のように、内部クラスと外部クラスの両方で x という同じ名前を使用するのはお勧めできないスタイルです。変数が両方とも x という名前でなければ、変数名が衝突することはないため、曖昧さを回避する構文を使用する必要はなくなります。ただし、この試験は、誰もが知っていることを知っているかどうかをテストするものではなく、平均よりも優れていることを証明するためのものです。また、時間の経過とともに場当たり的な対応が繰り返されてきた古いコードをメンテナンスする場合などには、この構文がときどき現れることもあります。[セクション 15.8.4](#) には、この形式についての説明があります。

以上の内容を踏まえると、選択肢 A と D が正解であることがわかります。選択肢 B と E は、本質的に `private` フィールドは自身を定義するクラスの中だけでアクセスできると言っているため、誤りです。

選択肢 C に記載されている構文も不適切です。通常、`super.x` という形式は有効ですが、これは外側のクラスではなく、親クラスの変数 `x` に明示的にアクセスすることを示します（[セクション 15.11.2](#)）。以上の内容を踏まえると、選択肢 C も誤りであることがわかります。

したがって、選択肢 A と D が正解となります。


```
//fix this /
```

まります。これは戻り値のオートボクシングを避けることができるため、ある意味で「最適」な選択肢であることを覚えておくといよいでしょう。もちろん、この問題ではそこまでの価値評価を行う必要はありません。いずれにしても、選択肢 D は明らかに正解です。

`UnaryOperator` では、引数と戻り値の型が同じである必要があります。戻り値が `int` であることはわかっているため、このラムダ式に `UnaryOperator` を当てはめるには、`UnaryOperator<Integer>` とする必要があります。これは、`Integer` オブジェクトに `length()` メソッドがあれば動作します。もちろん、これは該当しないので、選択肢 E も誤りです。

</article>

Simon Roberts : Sun Microsystems がイギリスで初めて Java の研修を行う少し前に Sun に入社し、Sun 認定 Java プログラマーと Sun 認定 Java 開発者の試験の作成に携わる。何冊かの Java 認定ガイドを執筆し、現在はフリーランスでシリコン・バレーや世界中のたくさんの大企業の研修を行う。Oracle の Java 認定プロジェクトにも継続的に関わっている。

