アプリケーションを再起動して、ログイン

「ログイン・ユーザー名」が表示されましたね。

「認可制御」など、ここで挙げた例以外の使い方は、ドキュメント[14]を参照してください。

ノート 「Spring Boot 1.4」からは「Thymeleaf」の最新版である「バージョン3」がサポートされています。

デフォルトでは「バージョン2」を使っていますが、「pom.xml」のタグ内に、以下の設定を追加することで、「バージョン3」に変更できます。

```
<properties>
  <!--(略)-->
  <thymeleaf.version>3.0.1.RELEASE</thymeleaf.version>
  <thymeleaf-layout-dialect.version>2.0.1</thymeleaf-layout-dialect.
version>
  <thymeleaf-extras-springsecurity4.version>3.0.0.RELEASE</thymeleaf-
extras-springsecurity4.version>
</properties>
```

「Thymeleaf2」までは、デフォルトでテンプレートの「HTML」を「XHTML」形式にする必要があり、必ず閉じタグを用意しなければなりません。通常の「HTML」に対応するには、別の「ライブラリ」を追加する必要があります。

「Thymeleaf3」からは、追加ライブラリなしで通常の「HTML」を使えるようになりました。また、「HTML」以外にも「プレーンテキスト」「CSS」「JavaScript」のテンプレート・エンジンとしても使用可能になりました。

「Spring Boot」では、「Thymeleaf2」用の設定がデフォルトで使われるため、起動時に次のようなWARNログが出力されます。

```
[THYMELEAF][restartedMain] Template Mode 'HTML5' is deprecated. Usi
ng Template Mode 'HTML' instead.
```

「Thymeleaf3」用の設定を使うために、「application.properties」に、以下のプロパティを追加してください。

```
spring.thymeleaf.mode=HTML
```

「閉じタグ」を作るのが煩わしいと感じる場合は、「Thymeleaf3」を試してみてはいかがでしょうか。

[14] https://github.com/thymeleaf/thymeleaf-extras-springsecurity4

第4章

PaaS「Cloud Foundry」にデプロイ

これまで作ったアプリケーションを「PaaS」(Platform as a Service)に「デプロイ」してみましょう。

「デプロイ先」の「PasS」として、本書では「Cloud Foundry」を利用します。

*****

「Cloud Foundry」でJavaアプリケーションを実行する場合、「war」を「アプリケーション・サーバ」にデプロイするのではなく、実行可能な形式にしてアプリケーションをプロセスとして起動します。

「Spring Boot」は初めからその形式になっているため、少ない手順で、簡単にデプロイできます。

「Spring Boot」は「Cloud Foundry」と相性がいいと言えるでしょう。

4.1 「PaaS」(Platform as a Service)の重要性

本書では簡単なアプリケーション開発を通じて「Spring Boot」を使ったアプリケーション開発方法を学んできました。

また実行可能な「jar」を作って、スタンドアロンなプロセスとしてアプリケーションを実行できることも学びました。

*

ただし、これまでの方法は、「1インスタンス」での運用しか考えてられていないことに注意してください。

利用者が少なく、可用性も求められない場合はこれでもいいのですが、実際にサービスを運用する場合は、複数の「インスタンス」にスケール・アウトして、ロード・バランサによってリクエストを振り分けるようにすることが多いです。

「インスタンス」が増えると、「死活監視」や「ログ・メトリクスの集約」など考えなければいけないことも増えてきます。

上記の要件に加えて、「セキュリティ」面の考慮もふまえたインフラ環境の構築には、高度なスキルと非常に多くの時間を要します。

このような環境をサービスとして提供するのが「PaaS」であり、「PaaS」を使うことで開発者はアプリケーションの開発に集中でき、新しい機能を素早くリリースすることが可能になります。

ソフトがビジネスを左右する時代になり、このリリース対するスピードの重要度が

増してきています。

「マイクロサービス・アーキテクチャ」という、複数のサービス(アプリケーション)から構成されるアーキテクチャにおいては、各サービスを管理するために、「PaaS」のようなプラットフォームは必須となるでしょう。

4.2 「Cloud Foundry」とは

「PaaS」といえば、「Google App Engine」や「AWS Elastic Beanstalk」、「Heroku」のようなパブリックでブラックボックスなサービスを思い浮かべるかもしれません。

本書で扱う「Cloud Foundry」は、「PaaS」を構築するための「オープンソース・ソフト」です。

「Cloud Foundry Foundation」[1]という財団のもと、Pivotal や IBM、ヒューレットパッカードなどのマルチベンダによって開発されています。

「Cloud Foundry」には、

- アプリケーションのスケール・アウト
- ロード・バランシング
- アプリケーションの自動復旧
- ログのストリーミング
- バックエンド・サービスの提供
- ユーザー管理
- マルチテナント
- 他言語(プログラミング言語)対応

など、プラットフォームに必要な機能が、ほとんど用意されています。

「Cloud Foundry」は「マルチクラウド」に対応しており、「Amazon Web Services」「Microsoft Azure」「Google Cloud Platform」といったパブリック・クラウドや、「Open Stack」「VMware vSphere」といったプライベートな環境で運用することもできます。

これらの環境に自分専用のプライベートな「PaaS」としてデプロイすることもできますし、ベンダが用意しているパブリックな「Cloud Foundry」サービスを利用することできます。

パブリックなサービスとしては、「Pivotal Web Services[2]」「IBM Bluemix[3]」「Swisscom Application Cloud[4]」などが有名です。

[1] https://www.cloudfoundry.org
[2] https://run.pivotal.io
[3] https://console.ng.bluemix.net
[4] https://developer.swisscom.com

4.3 「Pivotal Web Services」のアカウント作成

本書では「Pivotal」が運用するパブリックなサービスである、「Pivotal Web Services」(PWS)を使います。

アカウント作成後、1年間 $87 ぶんの無料期間(2GB メモリのアプリを2か月間利用する使用料に相当)が用意されています。

本書の内容は、どの「Cloud Foundry」でも同じように扱えます。

> **ノート** 「Pivotal」からは、その他、商用パッケージである「Pivotal Cloud Foundry」[5]、ローカル開発向けに仮想マシン上で「Cloud Foundry」を動かせる「PCF Dev」[6]が、提供されています。
> 「PCF Dev」は無償で利用できるので、本章の内容をローカルで試す環境として、とてもよいでしょう。

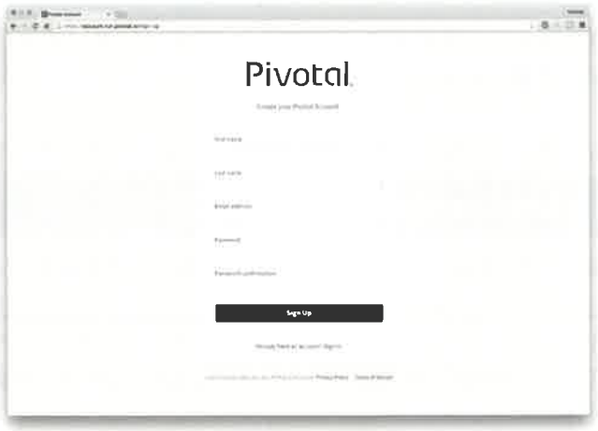
まずは、アカウントを作りましょう。

[1]「https://run.pivotal.io」にアクセスし、「SIGN UP FOR FREE」をクリックしてください。

「Pivotal Web Services」のトップ画面

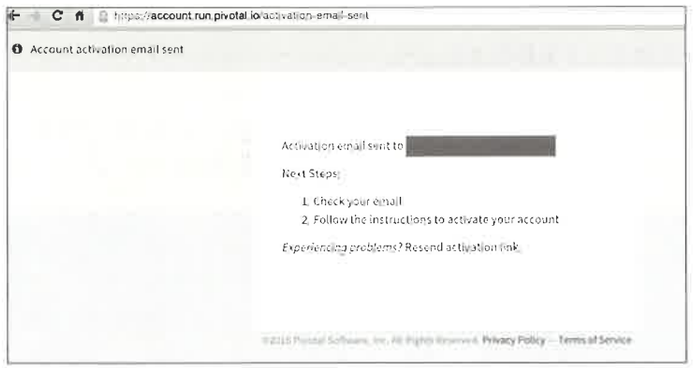
[5] https://pivotal.io/platform
[6] https://pivotal.io/pcf-dev

[2]「アカウント情報」を入力してください。



「アカウント情報」の入力

[3] 入力したメール・アドレスに、「アクティベーション用リンク」が送信されます。

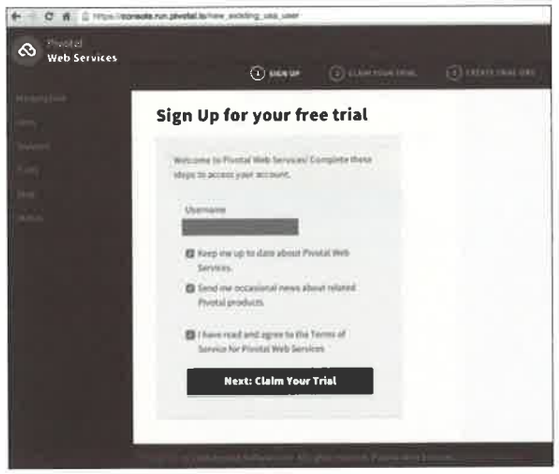


「アクティベーション・メール」の送信

[4] メールを確認し、「Verify your email address」をクリックしてください。

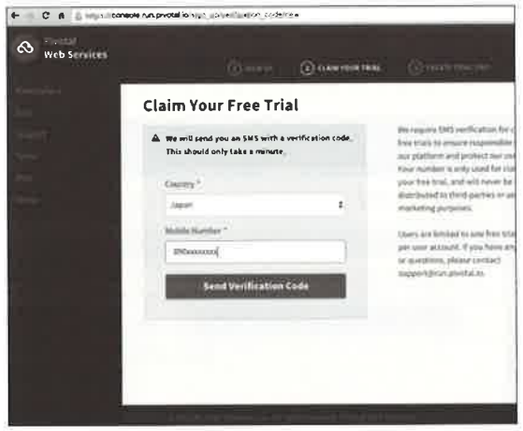
「アクティベーション・メール」の確認

[5]「I Have read and agree to the Terms of Service for Pivotal Web Services」にチェックを入れ、「Next: Claim Your Trial」をクリックしてください。



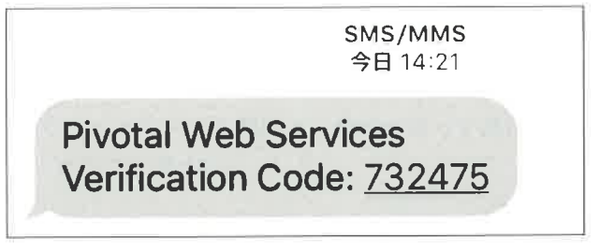
トライアルの開始

[6]「電話番号」を入力してください。



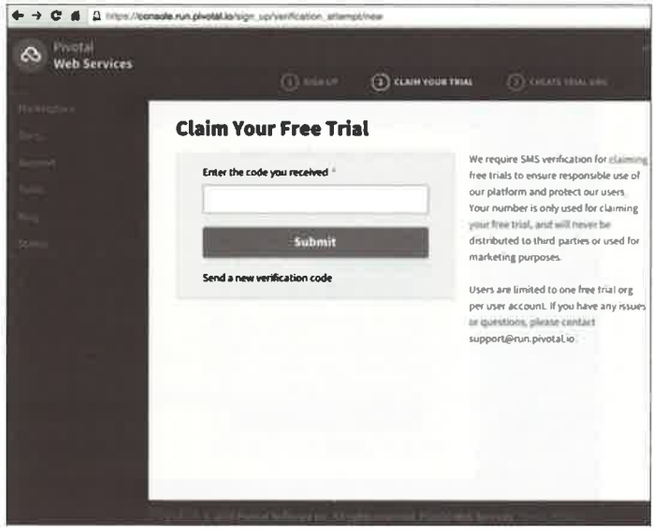
「電話番号」の入力

[7] SMS に「Verification Code」（確認コード）が送信されます。



「Verification Code」（確認コード）の確認

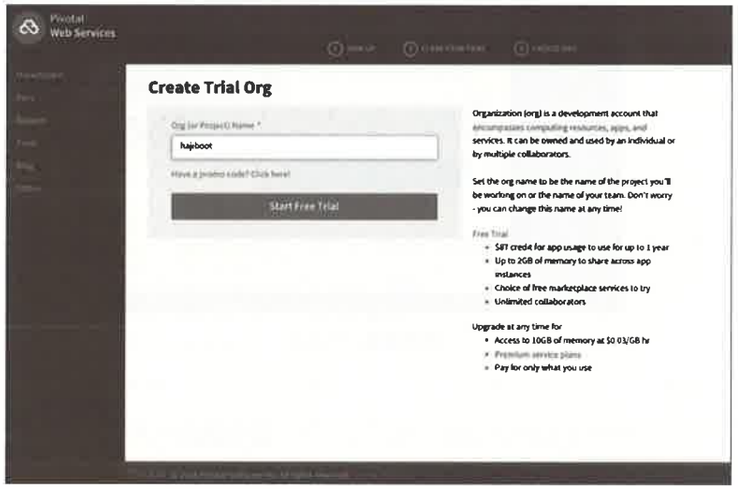
[8]「Verification Code」（確認コード）を入力して「Submit」をクリックしてください。



「Verification Code」（確認コード）の入力

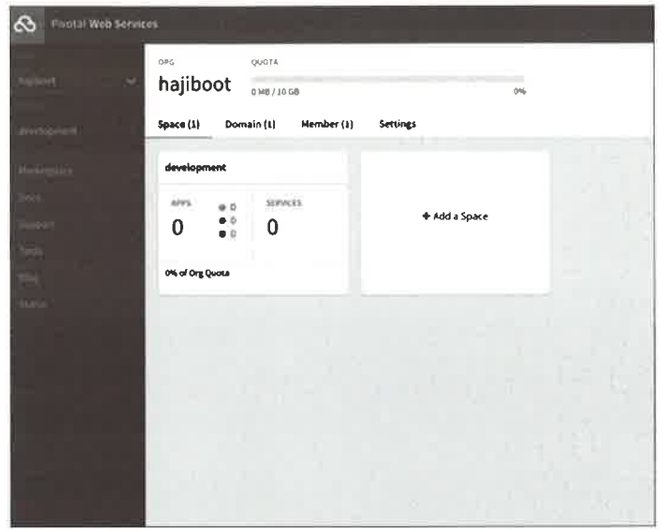
[9]「Organization 名」という、「プロジェクト名」に相当する項目を入力して、「Start Free Trial」をクリックしてください。

「Organization 名」はイニシャルなどを使ってグローバルで一意の名前にしてください。



「Organization 名」の入力

[10] コンソール画面が表示されます。

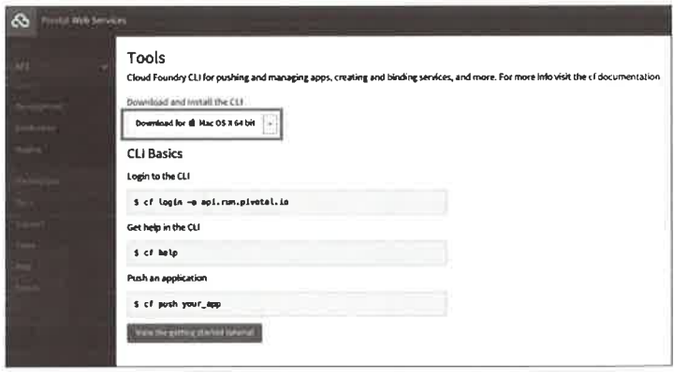


コンソール画面

4.4 「Cloud Foundry CLI」のインストール

「Cloud Foundry」では、アプリケーションのデプロイや管理のためのさまざまな操作に「CLI」（コマンドライン・インターフェイス）を使っています。

コンソール画面左の「Tools」リンクをクリックし、使う OS を選択しダウンロードボタンをクリックしてください。



「Cloud Foundry CLI」のダウンロード

ダウンロードしたインストーラを開いて、指示に従って操作し、「Cloud Foundry CLI」のインストールしてください。

インストールが完了したら、ターミナルを立ち上げて、「cf -v」コマンドを実行してください。

【ターミナル】「cf」コマンドの確認

```
$ cf -v
cf version 6.21.1+cd086c8-2016-08-18
```

●「Pivotal Web Services」へログイン

「cf login」コマンドでPWSにログインできます。

【ターミナル】PWSにログイン

```
$ cf login -a api.run.pivotal.io
API endpoint: api.run.pivotal.io

Email> メールアドレス

Password> パスワード
Authenticating...
OK

Targeted org hajiboot
Targeted space development

API endpoint:   https://api.run.pivotal.io (API version: 2.58.0)
User:           メールアドレス
Org:            hajiboot
Space:          development
```

これでPWSにアクセスできます。

4.5 「Hello Worldアプリケーション」のデプロイ

まずは「[1.3] はじめての「Spring Boot」」で作ったアプリケーションをデプロイしましょう。

「hajiboot」フォルダで、次の「mvnw」コマンドで実行可能な「jarファイル」を作ってください。

[4.5.1] アプリケーションのデプロイ

【ターミナル】実行可能な「jarファイル」の作成

```
$ ./mvnw clean package -DskipTests=true
```

実行可能な「jarファイル」が出来たら、「cf push <アプリケーション名> -p <実行可能jarファイルのパス>」コマンドで「Cloud Foundry」にデプロイできます。

ここでは<アプリケーション名>は重複しないように、「自分の名前」や「イニシャル」をつけてください。

また、「PWS」の無償期間で利用できるメモリには制約があるので、メモリ使用量を節約するために「-m」オプションで、アプリケーションに割り振るメモリも指定

します。

ここでは「256MB」を指定します。

【ターミナル】アプリケーションのデプロイ

```
$ cf push hajiboot-maki -p target/hajiboot-0.0.1-SNAPSHOT.jar -m 256m
Creating app hajiboot-maki in org hajiboot / space staging as メールアドレス ...
OK

Creating route hajiboot-maki.cfapps.io...
OK

Binding hajiboot-maki.cfapps.io to hajiboot-maki...
OK

(略)
Downloading binary_buildpack... <--- (1)
Downloading python_buildpack...
Downloading go_buildpack...
Downloading nodejs_buildpack...
Downloading java_buildpack...
(略)
Staging...
-----> Java Buildpack Version: v3.8.1 (offline) | https://github.com/cloudfoundry/java-buildpack.git#29c79f2 <--- (2)
-----> Downloading Open Jdk JRE 1.8.0_91-unlimited-crypto from https://java-buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_91-unlimited-crypto.tar.gz (found in cache)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.0s)
-----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from https://java-buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-2.0.2_RELEASE.tar.gz (found in cache)
       Memory Settings: -Xms104169K -XX:MetaspaceSize=64M -Xss228K -Xmx104169K -XX:MaxMetaspaceSize=64M
-----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-1.10.0_RELEASE.jar (found in cache)
Exit status 0
Staging complete
(略)
1 of 1 instances running

App started

OK

App hajiboot-maki was started using this command `CALCULATED_MEMORY=$($PWD/.java-buildpack/open_jdk_jre/bin/java-buildpack-memory-calculator-2.0.2_RELEASE -memorySizes=metaspace:64m..,stack:228k.. -memoryWeights=heap:65,metaspace:10,native:15,stack:10 -memoryInitials=heap:100%,metaspace:100% -stackThreads=300 -totMemory=$MEMORY_LIMIT) && JAVA_OPTS="-Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/.java-buildpack/open_jdk_jre/bin/killjava.sh $CALCULATED_MEMORY" && SERVER_PORT=$PORT eval exec $PWD/.java-buildpack/open_jdk_jre/bin/java $JAVA_OPTS -cp $PWD/. org.springframework.boot.loader.JarLauncher`

Showing health and status for app hajiboot-maki in org hajiboot / space staging as メールアドレス ...
OK
```

```
requested state: started
instances: 1/1
usage: 256M x 1 instances
urls: hajiboot-maki.cfapps.io  <--- (3)
last uploaded: Sat Aug 20 09:18:54 UTC 2016
stack: cflinuxfs2
buildpack: java-buildpack=v3.8.1-offline-https://github.com/cloudfoun
dry/java-buildpack.git#29c79f2 java-main open-jdk-like-jre=1.8.0_91-
unlimited-crypto open-jdk-like-memory-calculator=2.0.2_RELEASE
spring-auto-reconfiguration=1.10.0_RELEASE

     state     since                    cpu    memory    disk     details
#0   running   2016-08-20 06:19:34 PM   0.0%   856K of 256M   1.3M of 1G
```

プログラム解説

項　番	説　明
(1)	「Buildpack」というアプリケーションに対して「フレームワーク」や「ランタイム」を提供する仕組みをダウンロードしている。 ここで他言語対応が行なわれている。今回はアップロードしたファイルの形式から「Java/Spring」が検出され、「java_buildpack」が使われるが、「-b」オプションをつけることで、明示的に指定することもできる。
(2)	「ステージング」というフェーズで、「アプリケーション＋ランタイム」（ここではJREなど）を合わせた「Droplet」という「実行可能形式」が作られている。
(3)	「hajioot-maki」というアプリケーションに「hajiboot-maki.cfapps.io」というURLがマッピングされている。

実 行

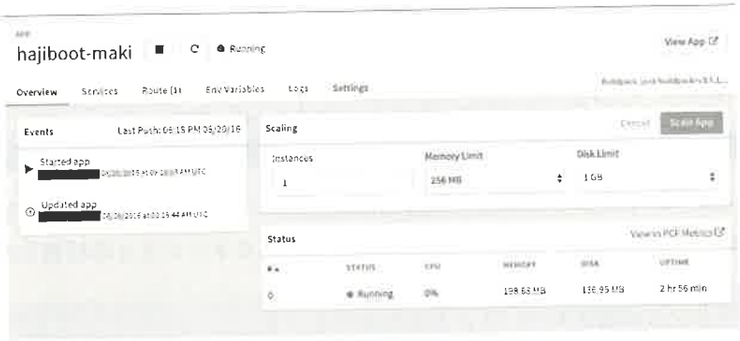
「http://<アプリケーション名>.cfapps.io」または「https://<アプリケーション名>.cfapps.io」にアクセスしてください。

デプロイされたアプリケーションにアクセス

これで「Cloud Foundry」に「HelloWorld アプリケーション」がデプロイされ、インターネットに公開されました。
とても簡単ではないでしょうか。

*

PWSの管理コンソール「https://console.run.pivotal.io」にアクセスして、デプロイされたアプリケーションの状態を見ることもできます。



デプロイされたアプリケーションの状態

[4.5.1] アプリケーションの「スケール・アウト」

次に「Cloud Foundry」にデプロイされたアプリケーションが簡単に「スケール・アウト」できることを確認しましょう。

「スケール・アウト」されていることが分かるように、アプリケーションを少し変更します。

[1]「HajibootApplication.java」の「hello」メソッドを修正して、環境変数「CF_INSTANCE_INDEX」に設定されているインスタンスのインデックスを表示するようにします。

```
@GetMapping("/")
String hello() {
    return "Hello World! " + " (" + System.getenv("CF_INSTANCE_INDEX")
+ ")";
}
```

[2] 再度ビルドして「cf push」でデプロイし直してください。
今回は「-b」オプションをつけて「buildpack」を明示します。

[ターミナル] アプリケーションの再デプロイ

```
$ ./mvnw clean package -DskipTests=true
$ cf push hajiboot-maki -p target/hajiboot-0.0.1-SNAPSHOT.jar -m 256m
-b java_buildpack
```

この段階ではインスタンス数は「1」なので、インデックスは常に「0」を示します。

[3] 次に「cf scale <アプリケーション名> -i <インスタンス数>」コマンドでこのアプリケーションをスケール・アウトさせましょう。

【ターミナル】3インスタンスにスケール・アウト

```
$ cf scale hajiboot-maki -i 3
```

「cf app <アプリケーション名>」コマンドでアプリケーションの状態を確認できます。

3インスタンス起動されていることが確認できます。

【ターミナル】アプリケーションの状態確認

```
$ cf app hajiboot-maki
Showing health and status for app hajiboot-maki in org hajiboot / spa
ce staging as メールアドレス ...
OK

requested state: started
instances: 3/3
usage: 256M x 3 instances
urls: hajiboot-maki.cfapps.io
last uploaded: Sat Aug 20 13:08:00 UTC 2016
stack: cflinuxfs2
buildpack: java_buildpack

     state   since                  cpu    memory    disk    details
#0   running 2016-08-20 10:08:33 PM  0.1%   184.2M of 256M    136.9M of 1G
#1   running 2016-08-20 10:18:05 PM  0.1%   181.8M of 256M    .9M of 1G
#2   running 2016-08-20 10:18:06 PM  0.3%   186.7M of 256M    .9M of 1G
```

デプロイされたアプリケーションに何回かアクセスすると、インデックスは「0 → 1 → 2 → 0 → ...」というように順番に表示されることが分かります。

「Cloud Foundry」内部のルータによって「ロード・バランシング」されています。

「スケール・アウト」したアプリケーションにアクセス

「Cloud Foundry」では、アプリケーションは仮想マシンにデプロイされるのではなく、(仮想マシン上の)「コンテナ」にデプロイされます。

「コンテナ」の起動は仮想マシンの起動に比べて非常に高速であるため、「スケール・アウト」は瞬時に行なわれます。

[4.5.3] アプリケーションの自動復旧

「Cloud Foundry」のアプリケーションは状態が監視されており、ダウンした場合には自動で復旧(再起動)します。

この挙動を確認するために、「HajibootApplication.java」に以下の「kill」メソッドを追加し、わざとダウンするようにしましょう。

```
@GetMapping("kill")
void kill() {
    System.exit(1);
}
```

再度「cf push」でデプロイしてください。デプロイ後、「https://<アプリケーション名>.cfapps.io/kill」に二回アクセスしてください。

「cf app <アプリケーション名>」コマンドで2つのインスタンスの状態が"crashed"になります。

2インスタンスがcrashed状態

```
     state    since                  cpu    memory disk    details
#0   crashed  2016-08-20 11:29:47 PM  0.0%   0 of 256M      0 of 1G
#1   crashed  2016-08-20 11:29:52 PM  0.0%   0 of 256M      0 of 1G
#2   running  2016-08-20 11:27:14 PM  0.1%   186.8M of 256M 136.9M of 1G
```

しばらくして、状態を再確認すると状態が"starting"になり、再起動し始めたことが分かります。

2インスタンスがstarting状態

```
     state     since                  cpu    memory  disk   details
#0   starting  2016-08-20 11:30:52 PM  0.1%   186.3M of 256M    136.9M of 1G
#1   starting  2016-08-20 11:30:54 PM  0.1%   182.6M of 256M    136.9M of 1G
#2   running   2016-08-20 11:27:14 PM  0.1%   186.8M of 256M    136.9M of 1G
```

再起動が完了すると、すべてのインスタンスの状態が「running」に戻ります。各インスタンスが自動で復旧したことが確認できました。

全インスタンスがruning状態に復旧

```
     state   since                  cpu    memory  disk   details
#0   running 2016-08-20 11:30:03 PM  0.2%   191.9M of 256M    136.9M of 1G
#1   running 2016-08-20 11:30:09 PM  0.1%   193.5M of 256M    136.9M of 1G
#2   running 2016-08-20 11:27:14 PM  0.1%   186.8M of 256M    136.9M of 1G
```

なお、ダウンしたインスタンスは、自動でルータの「ロード・バランス」対象から外れます。

そのため、この例ではすべてのインスタンスが復旧するまで、インデックスは「2」だけが表示されます。

[4.5.4] アプリケーションのログ

これまで見てきた「Spring Boot」アプリケーションのログは、すべて標準出力に出力されていました。

伝統的なシステムでは、ファイルにログを出力し、「ログ・ローテーション」の設定をすることが多く、これらの設定方法を知りたいという方も多いのではないでしょうか。

「Spring Boot」は「logging.*」プロパティで、これらの設定に対応していますが、これらの設定は、クラウド上では望ましくありません。

これまで見てきたように、「PaaS」上ではインスタンスがスケール・アウトや自動復旧がしばしば行なわれるため、「ローカル・ファイルシステム」にログを出力しても、ログ・ファイルが分散したり、消失してしまいます。

そのため、「PaaS」上では、アプリケーションは標準出力にログを出力し、プラットフォーム側で標準出力のログを集約して「ストリーム」として扱うことが、「ベスト・プラクティス」[7]として知られています。

「Cloud Foundry」もこの機能に対応しています。

「cf logs <アプリケーション名>」コマンドで、アプリケーションのログを追跡できます。

3インスタンスに「スケール・アウト」している場合は、3インスタンスぶんが集約されます。

【ターミナル】ログの追跡

```
$ cf logs hajiboot-maki
```

このコマンドでは、コマンド実行以降のログしか確認できません。

直近のログを確認するには「--recent」オプションをつけてください。

【ターミナル】直近のログの確認

```
$ cf logs hajiboot-maki --recent
```

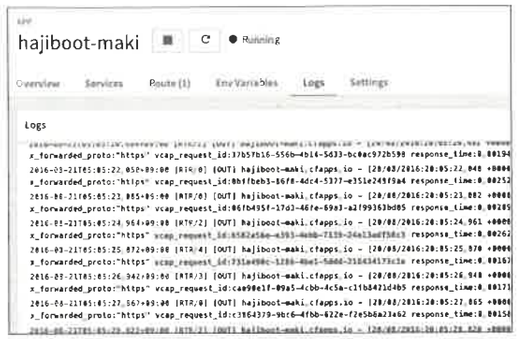
「Cloud Foundry」内に蓄積されるログの量は限られており、通常は「Logstash」「Splunk」「Papertrail」といったサード・パーティの「ログ・マネージャー」に転送します。

「ログ・マネージャー」との連携方法は、本書では扱いませんので、ドキュメント[8]を参照してください。

「PWS」では、「管理コンソール」から、アプリケーションのログを確認することもできます。

[7] https://12factor.net/ja/logs

[8] https://docs.cloudfoundry.org/devguide/services/log-management-thirdparty-svc.html



「管理コンソール」上のアプリケーション・ログ

[4.5.5] 「PCF Metrics」によるアプリケーションのモニタリング

「PWS」の独自機能として、「PCF Metrics」というアプリケーションの「メトリクス」(品質測定値)や「ログ」を蓄積して視覚化する仕組みが用意されています。

2016年8月時点で「BETA版」として無償で利用可能です。

*

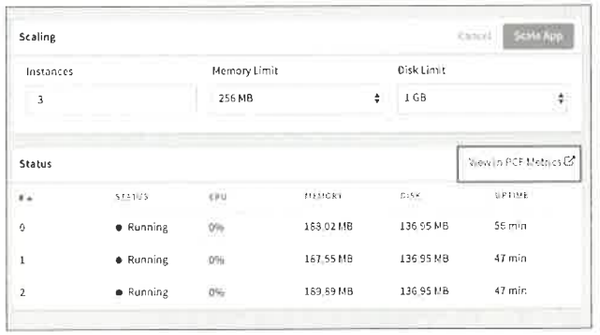
「PCF Metrics」を利用することで、直近1日間の、

- コンテナの「CPU」「メモリ」「ディスク使用率」
- 単位秒あたりの「リクエスト数」「エラー数」「リクエストのレイテンシー」
- アプリケーションの「イベント」
- アプリケーションの「ログ」

を確認することができます。

*

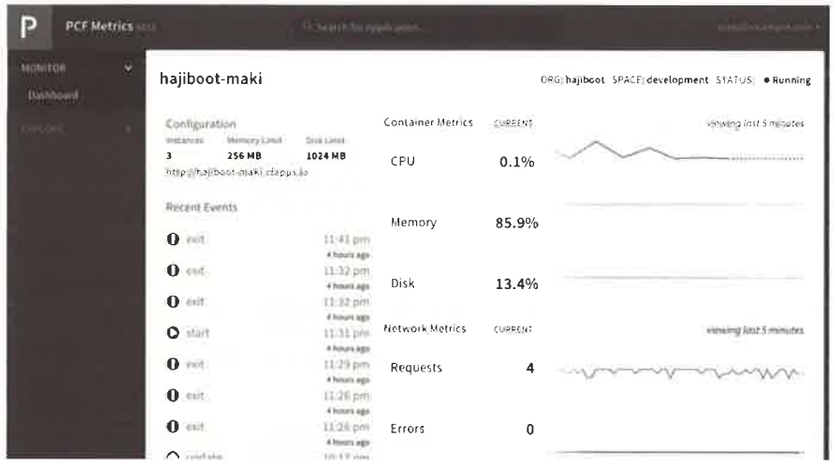
「管理コンソール」から「View in PCF Metrics」リンクをクリックしてください。



「PCF Metrics」へのリンク

「ダッシュボード」にアクセスできます。

ここでは、アプリケーションに発生した「イベント」と、直近5分間の「メトリクス」(品質測定値)を確認することができます。



「PCF Metrics」のダッシュボード

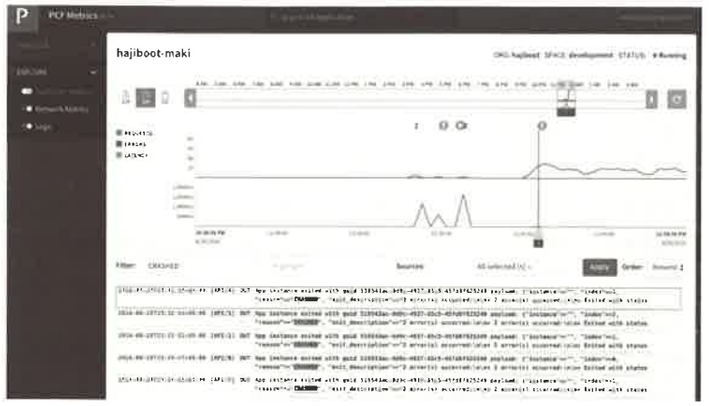
左側のメニューの「EXPLORE」をクリックしてください。

ネットワークに関する「メトリクス」のグラフと「アプリケーション・ログ」が表示されます。

「1 分」「1 時間」「1 日」という単位で状態を確認できます。

「ネットワーク・メトリクス」と「ログ」

「ログ」は、「キーワード」で検索することもできます。



ログの検索

「Container Metrics」にチェックを入れることで、コンテナに関する「メトリクス」のグラフも表示できます。

コンテナ・メトリクス

「PCF Metrics」を利用することで、直近でアプリケーションに何が起きたのか、どこでリクエストや CPU 使用率のスパイクが発生したのかを把握することが可能になります。

アプリケーション開発者が、このような便利な機能をセットアップすることなく使えるようになるのは、「PaaS」のメリットのひとつです。

[4.5.6] アプリケーションの「ブルー・グリーンデプロイ」

「ブルー・グリーンデプロイ」とは、できるだけ「ダウン・タイム」なしに、アプリケーションをアップデートする手法です。

*

「Cloud Foundry」は「ルータ」を内蔵しているため、「ルーティング情報」を制御することで、簡単に「ブルー・グリーンデプロイ」を実現できます。

「ブルー・グリーンデプロイ」で「HajibootApplication.java」のアップデートしてみましょう。

[1] まずは「cf app」コマンドで現在デプロイされているアプリケーション一覧を確認してください。
ここでは「cf scale ＜アプリケーション名＞ -i 1」コマンドで、インスタンス数を「1」に戻しています。

【ターミナル】アプリケーション一覧

```
$ cf apps

name          requested state instances  memory   disk     urls
hajiboot-maki  started       1/1        256M     1G       hajiboot-maki.cfapps.io
```

[2] 次に、「HajibootApplication.java」を以下のように変更します。

```
@GetMapping
String home() {
    return "Hello World!" + " (" + System.getenv("CF_INSTANCE_INDEX")
 + ") Ver.2";
}
```

[3] 変更後にアプリケーションをビルドし、こんどはアプリケーションを別名でデプロイしてください。

【ターミナル】別名でアプリケーションをデプロイ

```
$ ./mvnw clean package -DskipTests=true
$ cf push hajiboot-maki-v2 -p target/hajiboot-0.0.1-SNAPSHOT.jar -m
 256m -b java_buildpack
```

【ターミナル】アプリケーション一覧

```
$ cf apps

name           requested state  instances  memory  disk   urls
hajiboot-maki    started        1/1        256M    1G     hajiboot-maki.cfapps.io
hajiboot-maki-v2 started        1/1        256M    1G     hajiboot-maki-v2.cfapps.io
```

新しくデプロイされたアプリケーションにアクセスすると、変更後の内容を確認できます。

この段階では変更前のアプリケーションに対しては何も影響はありません。

[4] 次に「cf map-route ＜アプリケーション名＞＜ドメイン名＞ -n ＜ホスト名＞」コマンドで「hajiboot-maki.cfapps.io」へのリクエストが「hajiboot-maki-v2」にもルーティングされるようにします。

【ターミナル】ルーティングの追加

```
$ cf map-route hajiboot-maki-v2 cfapps.io -n hajiboot-maki
```

「cf apps」コマンドでアプリケーション一覧を確認すると「hajiboot-maki-v2」に対して、URL が 2 つ割り当てられていることが分かります。

【ターミナル】アプリケーション一覧

```
$ cf apps

name           requested state  instances  memory  disk   urls
hajiboot-maki    started        1/1        256M    1G     hajiboot-maki.cfapps.io
hajiboot-maki-v2 started        1/1        256M    1G     hajiboot-maki.cfapps.io,
                                                          hajiboot-maki-v2.cfapps.io
```

この時点で「https://hajiboot-maki.cfapps.io」にアクセスすると、「ロード・バランサ」によって、新旧両方のバージョンにルーティングされます。

[5] 次に「cf unmap-route ＜アプリケーション名＞＜ドメイン名＞ -n ＜ホスト名＞」コマンドで、「hajiboot-maki」を「hajiboot-maki.cfapps.io」からのルーティングを除きましょう。

【ターミナル】ルーティングの削除

```
$ cf unmap-route hajiboot-maki cfapps.io -n hajiboot-maki
```

[6] 同様に「hajiboot-maki-v2」を「hajiboot-maki-v2.cfapps.io」からのルーティングを削除しましょう。

【ターミナル】ルーティングの削除

```
$ cf unmap-route hajiboot-maki-v2 cfapps.io -n hajiboot-maki-v2
```

「cf apps」コマンドでアプリケーション一覧を確認すると「hajiboot-maki-v2」に対して「hajiboot-maki.cfapps.io」のみが割り当てられ、「hajiboot-maki」に対してはURLが割り当てられていないことが分かります。

【ターミナル】アプリケーション一覧

```
$ cf apps

name           requested state  instances  memory  disk  urls
hajiboot-maki  started          1/1        256M    1G
hajiboot-maki-v2 started        1/1        256M    1G    hajiboot-maki.cfapps.io
```

これで「https://hajiboot-maki.cfapps.io」にアクセスすると、新バージョンにのみルーティングされるので、無事にアプリケーションをバージョンアップできました。

旧バージョンのアプリケーションは「cf delete <アプリケーション名>」コマンドで削除してもいいですし、「cf stop <アプリケーション名>」コマンドで停止だけしておき、新バージョンに問題が発生した時に切り戻せるように残しておいてもいいでしょう。

ノート PWSを無償で利用できるのは1年間で$87分です。1時間単位で使用しているメモリ1GBあたり$0.03がかかります。無償枠を長く使いたい場合は、不要になったアプリケーションを削除または停止するとよいでしょう。

4.6 「顧客管理システム」のデプロイ

次に、[3.2.3]で開発した「顧客管理システム」のREST Webサービスをデプロイしましょう。

[4.6.1] アプリケーションのデプロイ

まずはデータベースを「H2データベース」のまま、デプロイしましょう。
前節と同じくビルドして「cf push」コマンドを実行するだけです。
今回は、メモリ・サイズを「512MB」にします。

```
$ ./mvnw clean package -DskipTests=true
$ cf push hajiboot-rest-maki -p target/hajiboot-rest-0.0.1-SNAPSHOT.
jar -m 512m -b java_buildpack
```

「hajiboot-rest-maki.cfapps.io」に「顧客管理システム」のREST Webサービスがデプロイされたので「curl」コマンドでアクセスしましょう。

```
$ curl https://hajiboot-rest-maki.cfapps.io/api/customers -i -XGET
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Sun, 21 Aug 2016 14:29:17 GMT
X-Application-Context: hajiboot-rest-maki:cloud:0
X-Vcap-Request-Id: 828ccf91-4687-4506-60bf-256394bb6ed9
Content-Length: 326
Connection: keep-alive

{"content":[{"id":1,"firstName":"Nobita","lastName":"Nobi"},{"id":4,"
firstName":"Shizuka","lastName":"Minamoto"},{"id":3,"firstName":"Sune
o","lastName":"Honekawa"},{"id":2,"firstName":"Takeshi","lastName":"G
oda"}],"last":true,"totalPages":1,"totalElements":4,"sort":null,"firs
t":true,"numberOfElements":4,"size":10,"number":0}j
```

ここではローカル開発と同様に組込の「H2データベース」が使われています。

*

次に、使うデータベースとして「MySQL」で使うための準備を行ないます。

[4.6.2] 「cloudプロファイル」の利用

「Spring」では「プロファイル」という概念があり、環境ごとに「プロパティ」や「Bean定義」のグループを定義して実行時に切り替えることが可能です。

たとえば、①「dev」プロパティには開発用に「組み込みデータベース」の設定を行ない、②「prod」プロファイルには商用環境で使う「MySQL」の設定を行ないます。
「プロファイル」は、システムプロパティ「spring.profiles.active」または環境変数「SPRING_PROFILES_ACTIVE」で指定可能なので、商用環境で「-Dspring.profiles.active=prod」を指定して、Javaを実行すれば「prod」プロファイルが適用されます。

「Spring Boot」では、「プロファイル」を指定した場合に、クラスパス直下の「application-(プロファイル名).properties」が読み込まれ、「application.properies」のプロパティ値を上書きすることができます。

そのため、「特定の環境でのみ有効にしたいプロパティ」を設定するのに便利です。

*

「Cloud Foundry」に Spring Boot アプリケーションをデプロイした場合は、デフォルトで「cloud プロファイル」が適用されます。

したがって「Cloud Foundry」環境下でのみ適用したいプロパティは「application-cloud.properties」に設定すればよいです。

*

「src/main/resources/application-cloud.proerties」を作って次の内容を設定してください。

```
# (1)
spring.jpa.hibernate.ddl-auto=update
# (2)
spring.datasource.initialize=false
```

プログラム解説

項 番	説 明
(1)	「H2 データベース」のような組込データベースを使う場合は、「spring.jpa.hibernate.ddl-auto」の値にはスキーマを毎回作成、破棄する「create-drop」がデフォルトで使われ、「MySQL」や「PostgreSQL」のようなデータベース・サーバを使用する場合は、何も行なわない「none」が使われる。 「Cloud Foundry」環境で「MySQL」を使用する場合に、スキーマが存在しなければ、スキーマが作られるように「update」を指定する。
(2)	「data.sql」は開発環境でのみ使うテスト・データであることが多く、クラウド環境にデプロイする際には無効にするため、「spring.datasource.initialize」を false に設定する。

[4.6.3] 「MySQL」の利用

「Cloud Foundry」では、データベースや Key-Value ストアのような「バックエンド・サービス」は、「マネージド・サービス」として用意されています。

「マーケットプレース」から、「バックエンド・サービス」を選択し、サービス・インスタンスを作ってアプリケーションにバインドすることで、アプリケーションには「サービス・インスタンス」に接続するための情報が、環境変数として設定されます。

*

「Cloud Foundry」の「バックエンド・サービス」を使ってみましょう。

[1] 「cf marketplace」コマンドで利用可能な「バックエンド・サービス一覧」を表示できます。

【ターミナル】PWS に用意されているバックエンド・サービス一覧

```
$ cf marketplace

service      plans                                    description
cleardb      spark, boost*, amp*, shock*
                                                      Highly available MySQL for your Apps.
cloudamqp    lemur, tiger*, bunny*, rabbit*, panda*
                                                      Managed HA RabbitMQ servers in the cloud
(略)
rediscloud   100mb*, 250mb*, 500mb*, 1gb*, 2-5gb*, 5gb*, 10gb*, 50gb*, 30mb
                                                      Enterprise-Class Redis for Developers
(略)
```

バックエンド・サービスの種類やプランは、「Cloud Foundry」のプロバイダによって異なります。

「Pivotal Web Services」では、たとえば「MySQL」のサービスとして「cleardb」、「RabbitMQ」のサービスとして「cloudamqp」、「Redis」のサービスの「rediscloud」が利用可能です。

[2] 今回は「cleardb」を使い、プランとしてはフリープラン[9]である「spark」を利用します。

「cf create-service <サービス名> <プラン名> <サービス・インスタンス名>」コマンドで「MySQL サービス・インスタンス」を作りましょう。

【ターミナル】「MySQL サービス・インスタンス」の作成

```
$ cf create-service cleardb spark customer-db
```

[3] 次に、作成したサービス・インスタンス「cf bind-service <アプリケーション名> <サービス・インスタンス名>」コマンドで、アプリケーションをバインドします。

[4] バインドする前に「application-cloud.properties」を含んだアプリケーションを再度ビルドして、デプロイします。

「サービス・インスタンス」は、バインド先のアプリケーションのステージング前にバインドされている必要があるので、今回は「--no-start」オプションをつけて「cf push」コマンドを実行します。

【ターミナル】「--no-start」オプションをつけて再デプロイ

```
$ ./mvnw clean package -DskipTests=true
$ cf push hajiboot-rest-maki -p target/hajiboot-rest-0.0.1-SNAPSHOT.
jar -m 512m -b java_buildpack --no-start
```

[5] そして「custoemr-db」サービス・インスタンスを「hajiboot-rest-maki」アプリケーションにバインドします。

[9] プラン名の右に * 印が付いているものは有料です。PWS にクレジットカード情報を登録しないと利用できません。

【ターミナル】「MySQL サービス・インスタンス」のバインド

```
$ cf bind-service hajiboot-rest-maki customer-db
```

[6] サービス・インスタンスをアプリケーションにバインドするとサービス・インスタンスへの接続情報が環境変数「VCAP_SERVICES」に埋め込まれます。
この環境変数は「cf env ＜アプリケーション名＞」コマンドで確認できます。

【ターミナル】環境変数の確認

```
$ cf env hajiboot-rest-maki

System-Provided:
{
 "VCAP_SERVICES": {
  "cleardb": [
   {
    "credentials": {
     "hostname": "us-cdbr-iron-east-04.cleardb.net",
     "jdbcUrl": "jdbc:mysql://us-cdbr-iron-east-04.cleardb.net/ad_655
34bc92cbcbcc?user=b568f82bc8e0a0\u0026password=7b4cee48",
     "name": "ad_65534bc92cbcbcc",
     "password": "7b4cee48",
     "port": "3306",
     "uri": "mysql://b568f82bc8e0a0:7b4cee48@us-cdbr-iron-east-04.cle
ardb.net:3306/ad_65534bc92cbcbcc?reconnect=true",
     "username": "b568f82bc8e0a0"
    },
    "label": "cleardb",
    "name": "customer-db",
    "plan": "spark",
    "provider": null,
    "syslog_drain_url": null,
    "tags": [
     "(略)",
     "mysql",
     "relational"
    ],
    "volume_mounts": []
   }
  ]
 }
}
(略)
```

「VCAP_SERVICES」というJSONの中に「credentials」というキーがあり、「MySQL」の接続情報を取得できます。

[7] サービスがバインドされたので「cf start ＜アプリケーション名＞」コマンドでアプリケーションを起動しましょう。

【ターミナル】「MySQL サービス・インスタンス」がバインドされたアプリケーションの起動

```
$ cf start hajibot-rest-maki
(略)
Staging...
-----> Java Buildpack Version: v3.8.1 (offline) | https://github.com/
cloudfoundry/java-buildpack.git#29c79f2
-----> Downloading Open Jdk JRE 1.8.0_91-unlimited-crypto from htt
ps://java-buildpack.cloudfoundry.org/openjdk/trusty/x86_64/
openjdk-1.8.0_91-unlimited-crypto.tar.gz (found in cache)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.1s)
-----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from
 https://java-buildpack.cloudfoundry.org/memory-calculator/trusty/
x86_64/memory-calculator-2.0.2_RELEASE.tar.gz (found in cache)
       Memory Settings: -XX:MaxMetaspaceSize=64M -Xss228K -Xms317161K
-XX:MetaspaceSize=64M -Xmx317161K
-----> Downloading Maria Db JDBC 1.4.6 from https://java-buildpack.cl
oudfoundry.org/mariadb-jdbc/mariadb-jdbc-1.4.6.jar (found in cache)
 <--- (1)
-----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from
 https://java-buildpack.cloudfoundry.org/auto-reconfiguration/auto-
reconfiguration-1.10.0_RELEASE.jar (found in cache)
Exit status 0
Staging complete
(略)
```

プログラム解説

項　番	説　明
(1)	ステージング・フェーズにおいて、「Droplet」作成中にMySQLの使用を検出し、自動でMySQLのJDBCドライバ（ここではMaria DB JDBCドライバ）が取り込まれる。

[8]「顧客新規作成」と「顧客全件取得」のAPIを実行して、データベースにアクセスできることを確認しましょう。

【ターミナル】「顧客新規作成」と「顧客全件取得」の API を実行

```
$ curl http://hajiboot-rest-maki.cfapps.io/api/customers -i -XPOST -H
 "Content-Type: application/json" -d "{¥"firstName¥":¥"Nobita¥",¥"las
tName¥":¥"Nobi¥"}"
HTTP/1.1 201 Created
Content-Type: application/json;charset=UTF-8
Date: Sun, 21 Aug 2016 14:42:43 GMT
Location: http://hajiboot-rest-maki.cfapps.io/api/customers/2
X-Application-Context: hajiboot-rest-maki:cloud:0
X-Vcap-Request-Id: bcfa8695-243b-48ec-5657-a0dd11801530
Content-Length: 47
Connection: keep-alive

{"id":2,"firstName":"Nobita","lastName":"Nobi"}

$ curl https://hajiboot-rest-maki.cfapps.io/api/customers -i -XGET
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Sun, 21 Aug 2016 14:43:18 GMT
X-Application-Context: hajiboot-rest-maki:cloud:0
X-Vcap-Request-Id: 75cb9a54-7882-4ce9-4647-c767ef3a28f6
```

```
Content-Length: 173
Connection: keep-alive

{"content":[{"id":2,"firstName":"Nobita","lastName":"Nobi"}],"last":t
rue,"totalPages":1,"totalElements":1,"size":10,"number":0,"sort":null
,"first":true,"numberOfElements":1}j
```

一件の顧客データが登録されていることが分かります。

[9] アプリケーションには「MySQL」に関する設定をまったく行ないませんでしたが、何が起きているのでしょうか。
「cf logs <アプリケーション名> --recent」コマンドでログを見てみましょう。

【ターミナル】出力されたログ

```
2016-08-21T23:36:44.10+0900 [APP/0]
OUT 2016-08-21 14:36:44.106  INFO 28 --- [           main] urceClou
dServiceBeanFactoryPostProcessor : Auto-reconfiguring beans of type
javax.sql.DataSource
2016-08-21T23:36:44.12+0900 [APP/0]      OUT 2016-08-21 14:36:44.124
INFO 28 --- [           main] o.c.r.o.s.c.s.r.PooledDataSourceCreator
: Found Tomcat high-performance connection pool on the classpath. Us
ing it for DataSource connection pooling.
2016-08-21T23:36:44.18+0900 [APP/0]      OUT 2016-08-21 14:36:44.180
INFO 28 --- [           main] urceCloudServiceBeanFactoryPostProces
sor : Reconfigured bean dataSource into singleton service connector
org.apache.tomcat.jdbc.pool.DataSource@646be2c3{ConnectionPool[defa
ultAutoCommit=null; defaultReadOnly=null; defaultTransactionIsolati
on=-1; defaultCatalog=null; driverClassName=org.mariadb.jdbc.Driver;
maxActive=4; maxIdle=100; minIdle=0; initialSize=0; maxWait=30000;
testOnBorrow=true; testOnReturn=false; timeBetweenEvictionRunsMill
is=5000; numTestsPerEvictionRun=0; minEvictableIdleTimeMillis=60000;
testWhileIdle=false; testOnConnect=false; password=********; url=jdb
c:mysql://us-cdbr-iron-east-04.cleardb.net/ad_65534bc92cbcbcc?user=
b568f82bc8e0a0&password=7b4cee48; username=null; validationQuery=/*
ping */ SELECT 1; validationQueryTimeout=-1; validatorClassName=nu
ll; validationInterval=30000; accessToUnderlyingConnectionAllowed=tr
ue; removeAbandoned=false; removeAbandonedTimeout=60; logAbandoned=
false; connectionProperties=null; initSQL=null; jdbcInterceptors=nu
ll; jmxEnabled=true; fairQueue=true; useEquals=true; abandonWhenPerc
entageFull=0; maxAge=0; useLock=false; dataSource=null; dataSourceJN
DI=null; suspectTimeout=0; alternateUsernameAllowed=false; commitOnR
eturn=false; rollbackOnReturn=false; useDisposableConnectionFacade=t
rue; logValidationErrors=false; propagateInterruptState=false; ignor
eExceptionOnPreLoad=false; }
```

「Auto-reconfiguring beans of type javax.sql.DataSource」という出力があり、バインドされた「MySQL」サービス・インスタンスにアクセスするためのデータソースが作られていることが分かります。

「Cloud Foundry」環境ではステージング段階で用意された「Spring Auto Reconfiguration」が、アプリケーションにバインドされた「サービス・インスタンス」を検出し、その「サービス・インスタンス」に対応した「Bean」(ここでは「DataSource」)定義を作って、「DIコンテナ」中の既存の定義を書き換えます[10]。

今回は環境変数「VCAP_SERVICES」中の「tags」キーに「mysql」が含まれていることが「MySQL」を検出するポイントでした。

これによって、ローカル環境向けに作った実行可能な「jarファイル」をそのまま「Cloud Foundry」にデプロイして、「バックエンド・サービス」を使うこともできるため、設定を減らすことができます。

*

「バックエンド・サービス」を使うBean定義を明示的に行ないたい場合は、「Spring Cloud Connector」を利用すると便利です。

> **ノート** 今回使用した「MySQL」のサービスである「ClearDB」の「spark」プランは無償ですが、「データ容量は5MB」「同時接続可能なコネクション数は4」と小さく、検証用途以外には利用するのは難しいでしょう。

[4.6.4] 「Spring Cloud Connectors」の利用

「PaaS」では「バックエンド・サービス」の接続情報が、環境変数を経由してアプリケーションに渡されるのが一般的です[11]。
これまで見てきたように「Cloud Foundry」では、環境変数「VCAP_SERVICES」が使われていましたし、「Heroku」では「バックエンド・サービス」ごとに決まった環境変数が用意されています。

これらプラットフォームによって異なる「バックエンド・サービス」へのアクセスを透過的に扱い、接続するためのオブジェクト(データベースであれば「DataSource」オブジェクト)を生成するためのプロジェクトが、「Spring Cloud Connectors」[12]です。
[4.6.3]で使われた「Spring Auto Reconfiguration」でも「Spring Cloud Connectors」が利用されています。

「Spring Cloud Connectors」は、「MySQL」や「PostgreSQL」などのRDMSだけでなく、「MongoDB」「Redis」「RabbitMQ」などのデータストアへのアクセスにも対応しています。
また、プラットフォームとして「Cloud Foundry」「Heroku」、ローカル環境で試すための「Local」に対応しています。

*

「Cloud Foundry」に対応した「Spring Cloud Connectors」を使うには、以下の依存関係を追加してください。

---

[10] 対象のBeanが複数定義ある場合(2つのDataSourceなど)や異なる同種サービスを同時に使う場合(「MySQL」と「PostgreSQL」など)は、Bean定義の書き換えは発生しません。
[11] https://12factor.net/ja/config
[12] http://cloud.spring.io/spring-cloud-connectors/

【pom.xml】依存関係の追加

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-spring-service-connector</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
</dependency>
```

次に「Spring Cloud Connectors」を使った「JavaConfig」を作成しましょう。

```
package com.example;

import org.springframework.cloud.config.java.AbstractCloudConfig;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import javax.sql.DataSource;

@Configuration
@Profile("cloud") // (1)
public class CloudConfig extends AbstractCloudConfig /* (2) */ {
    @Bean
    DataSource dataSource() {
        return connectionFactory().dataSource(); // (3)
    }
}
```

プログラム解説

項　番	説　明
(1)	「JavaConfig」に「@Profile」アノテーションを付与することにより、本クラスによるBean定義は指定したプロファイルでのみ有効になる。今回の設定は「Cloud Foundry」でのみ使いたいので「cloud」プロファイルを指定する。
(2)	「AbstractCloudConfig」クラスを継承することにより、「Spring Cloud Connectors」を使用するための設定を簡単に行なえる。
(3)	アプリケーションにバインドされたサービス・インスタンスにアクセスするための「DataSource」オブジェクトを作るための設定。

実　行

この「JavaConfig」を含んだアプリケーションをビルドして、「Cloud Foundry」にデプロイし、[4.6.3]と同様にデータベースにアクセスできることを確認してください。

＊

「Cloud Foundry」上で「MySQL」以外に対応しているサービスはドキュメント[13]を参照してください。

なお、「Spring Cloud Connectors」がクラスパスに存在する場合は前述の「Spring Auto Reconfiguration」は無効になります。

[13] http://cloud.spring.io/spring-cloud-connectors/spring-cloud-cloud-foundry-connector.html

[4.6.5] Spring Cloud Connectors のコネクションプールに関する設定

「AbstractCloudConfig」のデフォルト設定ではコネクションプールに関する設定が固定されてしまいます。コネクションプールに関する設定を行なう場合は、次のように「PooledServiceConnectorConfig.PoolConfig クラス」を使います。

PooledServiceConnectorConfig.PoolConfig クラス

```
package com.example;

import javax.sql.DataSource;

import org.springframework.cloud.config.java.AbstractCloudConfig;
import org.springframework.cloud.service.PooledServiceConnectorConfig;
import org.springframework.cloud.service.relational.DataSourceConfig;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("cloud")
public class CloudConfig extends AbstractCloudConfig {
    @Bean
    DataSource dataSource() {
        PooledServiceConnectorConfig.PoolConfig poolConfig = new
PooledServiceConnectorConfig.PoolConfig(
                5 /* 最小プール数 */, 30 /* 最大プール数 */, 3000 /* 最大待機時間 */);
        return connectionFactory().dataSource(new DataSourceConfig
(poolConfig, null));
    }
}
```

実は[4.6.3]で説明したAuto-Reconfigurationを利用すると次のログが出力されていました。

【ターミナル】出力されたログ

```
  org.apache.tomcat.jdbc.pool.ConnectionPool          WARNING maxIdle
is larger than maxActive, setting maxIdle to: 4
```

これはAuto-Reconfiguration側で最大接続数を4に指定しているからです（バックエンドサービスの無償枠向け）[14]。

基本的にはspring-cloud-connectorを使って、コネクションプールの設定をすべきです。

4.7 「Spring Boot Actuator」で「アプリケーションの状態」を監視

「Cloud Foundry」とは直接関係ありませんが、アプリケーションを「監視」や「管理」するための便利なエンド・ポイントを提供する「Spring Boot Actuator」を紹介します。

＊

「Spring Boot Actuator」は、依存関係を追加するだけで使用可能であり、「JVMの状態」などの「メトリクス」や「ヘルス・チェック」「環境変数」「スレッド」などを「HTTP」(JSON形式)や「JMX」で取得できます。

「PaaS」のようなサーバをコントロールするのが難しい環境において、この機能は特に効果的です。

[14] https://discuss.pivotal.io/hc/en-us/articles/221898227-Connection-pool-warning-message-maxIdle-is-larger-than-maxActive-setting-maxIdle-to-4-seen-in-PCF-deployed-Spring-app

使い方は「pom.xml」に以下の依存関係を追加するだけです。

【pom.xml】依存関係の追加

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

この依存関係を追加してアプリケーションを起動すると、以下のようなログが出力されることを確認してください。

【ターミナル】出力されたログ

```
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/heapdump (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/mappings (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/metrics/{name:.*}]
(略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/metrics (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/info (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/configprops (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/env/{name:.*}] (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/env (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/dump (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/health (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/beans (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/autoconfig (略)
(略) o.s.b.a.e.mvc.EndpointHandlerMapping  : Mapped "{[/trace (略)
```

HTTP でアクセスできる「エンド・ポイント」が表示されています。

*

代表的な「エンド・ポイント」について次の表で説明します。

代表的な「エンド・ポイント」

パス	取得内容
/metrics	「アクセス・カウンタ」や「レスポンス・タイム」「JVM の状態」(「ヒープ」「GC 回数」など) など。
/health	各種「データソース」の「ヘルス・チェック」。
/dump	スレッド・ダンプ。
/configprops	プロパティの設定値。
/env	環境変数やシステム・プロパティ。

*

依存関係を追加した後、再ビルドして、「Cloud Foundry」にデプロイし、このアプリケーションに対して、「ヘルス・チェック」の API を実行してみます。

【ターミナル】「ヘルス・チェック」の API を実行

```
$ curl https://hajiboot-rest-maki.cfapps.io/health -XGET
{"status":"UP","diskSpace":{"status":"UP","total":1056858112,"free":8
94726144,"threshold":10485760},"db":{"status":"UP","database":"MySQL"
,"hello":1}}j
```

データベースの状態が確認できました。

正常に起動している場合は「UP」、障害が発生している場合は「DOWN」が返ります。

*

本書では省略しますが、他の「エンド・ポイント」も活用して、運用時の役に立ててください。

第5章 「Spring Boot」におけるテスト

本書の最後に、「Spring Boot」を用いたテストの方法を説明します。

これまで「Spring Boot」を使ったアプリが「組み込みサーバ」を立ち上げて動作することを見てきました。

Java でテストを書くときは「JUnit」を使うのが一般的です。「JUnit」を使ったテストでサーバを立ち上げ、DB まで接続した結合テストが簡単にできたら便利ではないでしょうか。

「Spring Boot」は、このような「組み込みサーバ」を使った「結合テスト」を実施する仕組みを提供しています。

本章では、このような「結合テスト」の実施方法について説明します。

*

なお、「モック」を利用した単体テストについては本書では扱いません。ここでは説明しませんが、「Spring Framework」にはもともと「MockMVC」[1] という、「サーバを立ち上げなくても Controller をテストできる仕組み」が用意されています。

5.1 「Hello World」アプリの結合テスト

「[1.3] はじめての「Spring Boot」」で最初に作った、「HelloWorld アプリ」のテストを書いてみましょう。

「Spring Initializr」で作ったプロジェクトには、あらかじめ「テスト・コード」の雛形が用意されています。

「src/test/java/com/example/HajibootApplicationTests.java」を開いて、次のコードを書いてください。

HajibootApplicationTests クラス

```
package com.example;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
```

[1] http://docs.spring.io/spring/docs/4.3.2.RELEASE/spring-framework-reference/html/testing.html#spring-mvc-test-framework