



# Deployment to the Cloud

Over the last couple of years the cloud has grown considerably and a lot of companies offer different cloud solutions, such as

- Platform as a Service
- Infrastructure as a Service
- Software as a Service

Platform as a Service is, as the name implies, a full platform to run your applications on, often you can choose some of the services (database, messaging, logging, etc.) you want to use for your applications. These services are provided by companies like Google (Google App Engine), Pivotal (CloudFoundry), and Oracle (Oracle Cloud Solutions).

Infrastructure as a Service provides infrastructure like virtual machines and other resources to build your own platform for deployment. Some examples are VMWare ESX and Oracle Virtualbox.

Software as a Service is a piece of software or pieces of software delivered through the cloud, such as Office365 and Google Apps for Work.

This chapter will focus on the Platform as a Service cloud solution and especially the cloud solution offered by Pivotal named CloudFoundry.

## A-1 Sign Up for CloudFoundry Problem

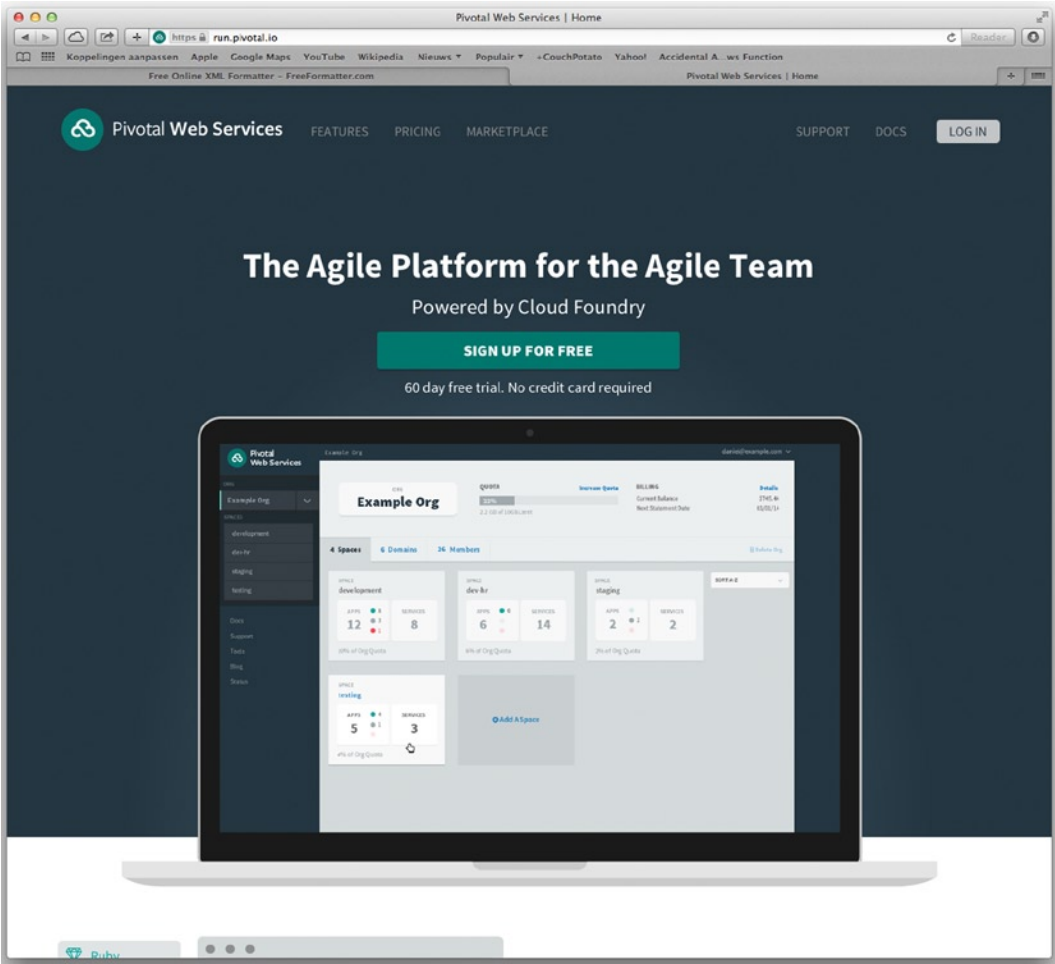
You want to use CloudFoundry for deployment.

### Solution

Sign Up with CloudFoundry on <http://run.pivotal.io>.

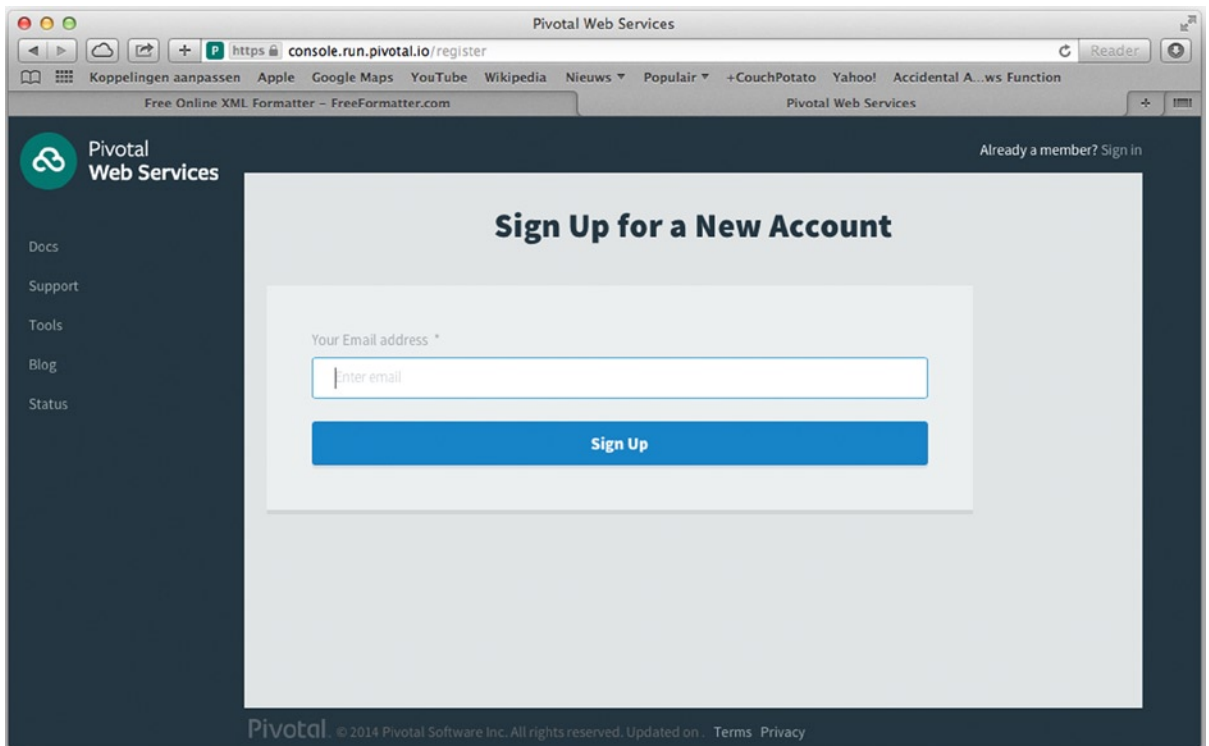
### How It Works

To be able to deploy to CloudFoundry an account is required. Navigate to <http://run.pivotal.io> and press the *Sign Up For Free* button (see Figure A-1).



**Figure A-1.** Sign Up / Sign In screen for CloudFoundry

Signing up for cloudfoundry is as easy as entering your e-mail address and pressing the Sign Up button (see Figure A-2).



**Figure A-2.** *Sign Up form for CloudFoundry*

After pressing the button you will receive a confirmation e-mail with a link, after clicking this link you will be transferred to the confirmation page (see Figure A-3).

Pivotal Web Services

Already a member? Sign in

## Sign Up for your free trial

Username

Choose a password \*

Confirm password \*

☒ Keep me up to date about Pivotal Web Services.

☒ Send me occasional news about related Pivotal products

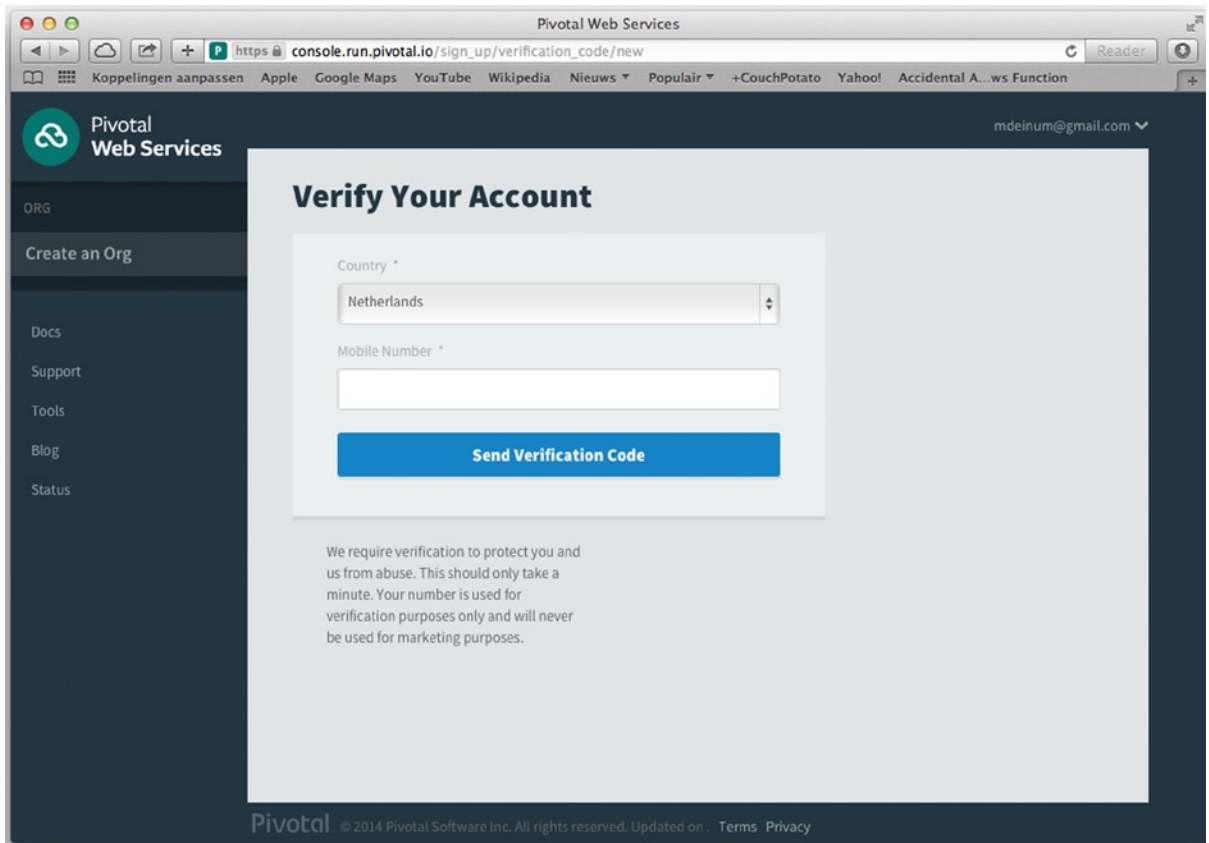
☒ I have read and agree to the [Terms of Service](#) for Pivotal Web Services

**Sign Up Now**

Pivotal © 2014 Pivotal Software Inc. All rights reserved. Updated on . [Terms](#) [Privacy](#)

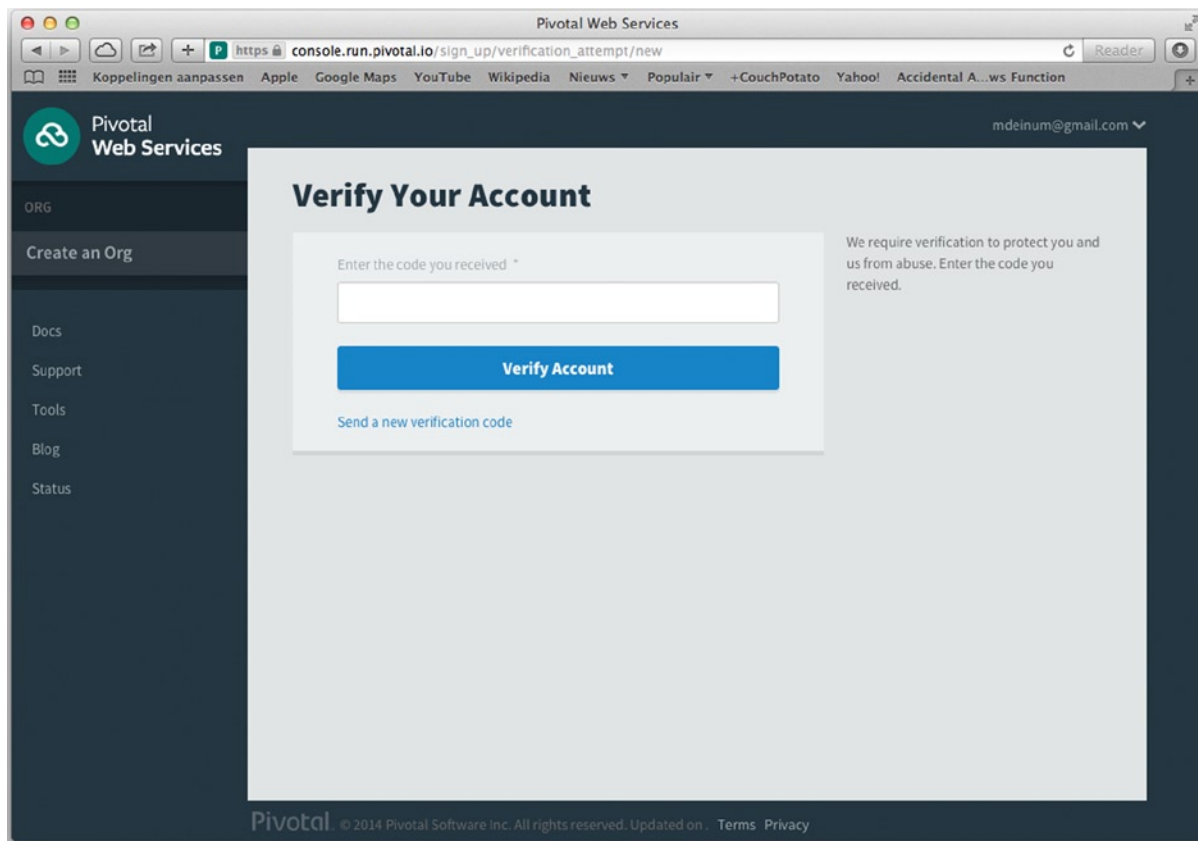
**Figure A-3.** CloudFoundry confirmation page

After pressing the *Sign Up Now* button you will be transferred to the account verification page (see Figure A-4) in which you are requested to enter your mobile number.



**Figure A-4.** CloudFoundry Account verification page

After pressing the *Send Verification Code* button you will receive a text message on your mobile phone with a verification code. This code can be entered in the next verification page (see Figure A-5).



**Figure A-5.** Enter verification code page

When you have entered the verification code you are asked to create an org. This can be the name of the project or your organization name. The organization name has to be unique and you will receive an error when you try to reuse an existing org name.

## A-2 Installing and Using CloudFoundry CLI Problem

You want to use the CloudFoundry CLI tooling to push your application.

### Solution

Download and install the CloudFoundry CLI tools.

## How It Works

To be able to manipulate your CloudFoundry instance you need tooling, there is tooling support deployment in different IDEs like Spring Source Toolsuite and IntelliJ IDEA. However the most powerful are the command line tools. To install the tools, first download the version for your system from <https://github.com/cloudfoundry/cli/releases>. Select and download the installer for your system, after installation the tools are available on the command line.

Now the command line tools need to be setup, open a terminal and type `cf login`. This will prompt for the api, e-mail, and password. The URL for the API is the URL to your cloudfoundry instance. For this recipe you are using the public API the URL is <https://api.run.pivotal.io>. The e-mail address and password are the ones you used for registration.

Next it will ask for the org and space to use, those you can skip as you only have a single org and single space at the moment.

After filling out the questions the configuration is written to a `config.json` file in the `.cf` directory in the user's home directory.

Let's create a simple hello world and deploy that to CloudFoundry.

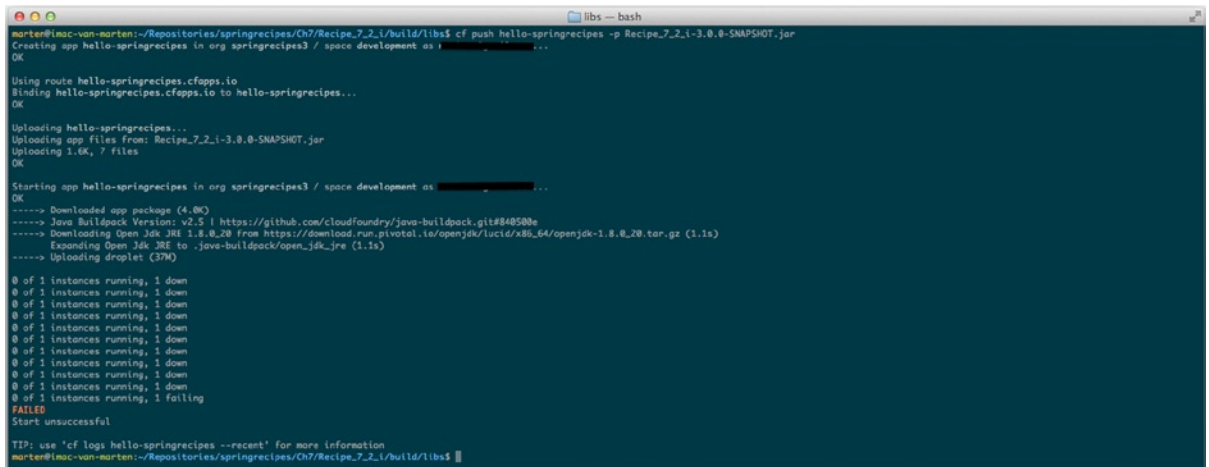
```
package com.apress.springrecipes.cloud;

public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World from CloudFoundry.");
    }

}
```

The class is a basic java class with a main method. The only thing that is being printed, is a message to the console. After compiling the file and creating a jar for it, it can be deployed to CloudFoundry. To deploy type `cf push <application-name> -p Recipe_a_2_i-3.0.0.jar` on the command line, where <application-name> is a nice name you can make up for your application. During deployment the output should look like that of Figure A-6.



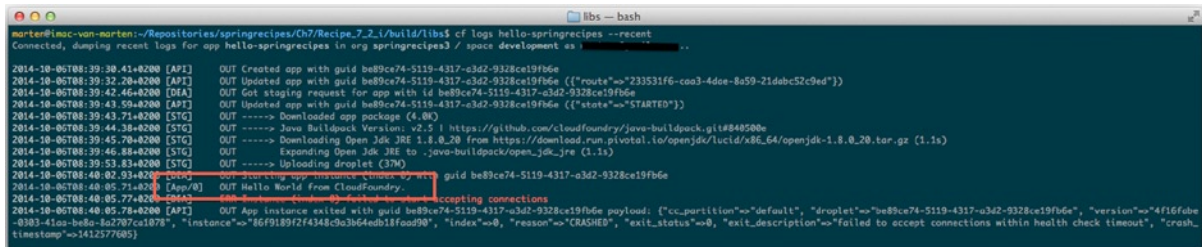
**Figure A-6.** *Output for application deployment*

First thing to notice is that, apparently, the starting of the application failed. Actually it did start but it only printed a message in the console and quit right after that. For the CloudFoundry tooling it looks like it failed to start because it already shut down before it was able to detect that it was up.

The first thing in the output is the creation of the application on CloudFoundry. It reserves some space and assigns memory to it (default 1Gb). Next it creates a route <application-name>.cfapps.io for public applications. This route is the entry point for application with a web interface. For the current application it has little use (adding the --no-route option to the cf push command prevents a route from being created).

After the creation the jar file is uploaded, after uploading CloudFoundry does detection on what kind of application it has to deal with. When it has determined that, the corresponding buildpack is downloaded and added. In our case that means the Java buildpack is installed. After that the application is started and the tooling will try to detect the successful start of the application. In this case it appears to have failed.

Type `cf logs <application-name> --recent` will give you the last logging of the application (see Figure A-7).



```

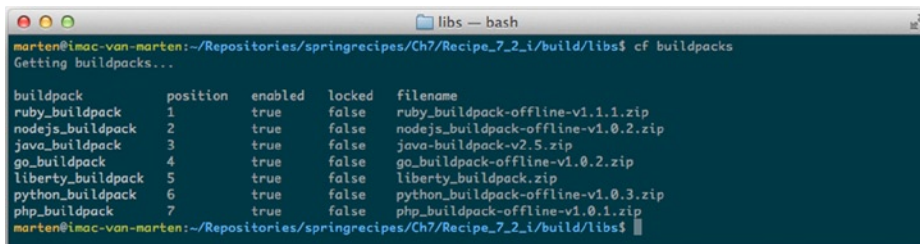
marten@mac-van-marten:~/Repositories/springrecipes/Ch7/Recipe_7_2_i/build/libs$ cf logs hello-springrecipes --recent
Connected, dumping recent logs for app hello-springrecipes in org springrecipes3 / space development as ...
2014-10-06T08:39:38.41+0200 [API] OUT Created app with guid be89ce74-5119-4317-a3d2-9328ce19f6de
2014-10-06T08:39:32.20+0200 [API] OUT Updated app with guid be89ce74-5119-4317-a3d2-9328ce19f6de ("route"=>"23531f6-cna3-4dce-8a59-21dabc52c9ed")
2014-10-06T08:39:42.46+0200 [DEA] OUT Got staging request for app with id be89ce74-5119-4317-a3d2-9328ce19f6de
2014-10-06T08:39:43.50+0200 [API] OUT Updated app with guid be89ce74-5119-4317-a3d2-9328ce19f6de ("state"=>"STARTED")
2014-10-06T08:39:43.71+0200 [STG] OUT -----> Downloaded app package (4.8K)
2014-10-06T08:39:44.38+0200 [STG] OUT -----> Java Buildpack Version: v2.5 | https://github.com/cloudfoundry/java-buildpack.git#848500e
2014-10-06T08:39:45.78+0200 [STG] OUT -----> Downloading Open Jdk JRE 1.8.0_20 from https://download.run.pivotal.io/openjdk/1.8.0_20.tar.gz (1.1s)
2014-10-06T08:39:46.88+0200 [STG] OUT Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.1s)
2014-10-06T08:39:53.83+0200 [STG] OUT -----> Uploading droplet (370K)
2014-10-06T08:40:02.93+0200 [DEA] OUT Starting app instance with guid be89ce74-5119-4317-a3d2-9328ce19f6de
2014-10-06T08:40:05.71+0200 [App/0] OUT Hello World from CloudFoundry.
2014-10-06T08:40:05.77+0200 [DEA] OUT App instance exited with guid be89ce74-5119-4317-a3d2-9328ce19f6de payload: {"cc_partition"=>"default", "droplet"=>"be89ce74-5119-4317-a3d2-9328ce19f6de", "version"=>"4f16fde-8303-41aa-be8a-8a2707ca1078", "instance"=>"86f9189f2f4348c0a3b64eab18fad98", "index"=>0, "reason"=>"CRASHED", "exit_status"=>0, "exit_description"=>"Failed to accept connections within health check timeout", "crash_timestamp"=>1412377085}

```

**Figure A-7.** Logging output for application

The logging contains the line you have put in the System.out. So it actually did start but ended right after that, which made it unavailable to the health check which signaled that it crashed.

As mentioned before CloudFoundry uses so-called buildpacks to (optionally) build and run applications. CloudFoundry supports a variety of different buildpacks. Typing `cf buildpacks` on the commandline will give a list of default, supported buildpacks (see Figure A-8).



```

marten@mac-van-marten:~/Repositories/springrecipes/Ch7/Recipe_7_2_i/build/libs$ cf buildpacks
Getting buildpacks...

```

buildpack	position	enabled	locked	filename
ruby_buildpack	1	true	false	ruby_buildpack-offline-v1.1.1.zip
nodejs_buildpack	2	true	false	nodejs_buildpack-offline-v1.0.2.zip
java_buildpack	3	true	false	java_buildpack-v2.5.zip
go_buildpack	4	true	false	go_buildpack-offline-v1.0.2.zip
liberty_buildpack	5	true	false	liberty_buildpack.zip
python_buildpack	6	true	false	python_buildpack-offline-v1.0.3.zip
php_buildpack	7	true	false	php_buildpack-offline-v1.0.1.zip

**Figure A-8.** Default supported buildpacks

As you can see multiple languages like Ruby, Python, PHP, and Java are supported, which means CloudFoundry isn't limited to just Java based applications.



## A-3 Deploying a Spring MVC Application Problem

You want to deploy your Spring MVC based application to CloudFoundry.

### Solution

Use `cf push` to push the war to CloudFoundry.

### How It Works Creating the Application

Let's start by creating a simple Spring MVC based web application. First create a `ContactRepository` interface.

```
package com.apress.springrecipes.cloud;

import java.util.List;

public interface ContactRepository {

    List<Contact> findAll();
    void save(Contact c);
}
```

For now create a Map based implementation for this interface.

```
package com.apress.springrecipes.cloud;

import org.springframework.stereotype.Repository;
import java.util.*;

@Repository
public class MapBasedContactRepository implements ContactRepository {

    private Map<Long, Contact> contacts = new HashMap<>();

    @Override
    public List<Contact> findAll() {
        return new ArrayList(contacts.values());
    }

    @Override
    public void save(Contact c) {
        if (c.getId() <= 0) {
            long generated = UUID.randomUUID().getMostSignificantBits();
            c.setId(generated);
        }
    }
}
```

```

        contacts.put(c.getId(), c);
    }
}

```

Of course there needs to be a `Contact` entity, this is just a simple class with three properties.

```
package com.apress.springrecipes.cloud;
```

```
public class Contact {

    private long id;
    private String name;
    private String email;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```

As this is a web application let's create a controller.

```
package com.apress.springrecipes.cloud.web;

import com.apress.springrecipes.cloud.Contact;
import com.apress.springrecipes.cloud.ContactRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

```

```

@Controller
@RequestMapping("/contact")
public class ContactController {

    private final ContactRepository contactRepository;

    @Autowired
    public ContactController(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String list(Model model) {
        model.addAttribute("contacts", contactRepository.findAll());
        return "list";
    }

    @RequestMapping(value="/new", method=RequestMethod.GET)
    public String newContact(Model model) {
        model.addAttribute(new Contact());
        return "contact";
    }

    @RequestMapping(value="/new", method=RequestMethod.POST)
    public String newContact(@ModelAttribute Contact contact) {
        contactRepository.save(contact);
        return "redirect:/contact";
    }
}

```

The controller is very simple. It has a method for showing a list of currently available contacts and it has a method to add a new contact. The next two views need to be created in the /WEB-INF/views directory. First the list.jsp file.

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!doctype HTML>
<html>
<head>
    <title>Spring Recipes - Contact Sample</title>
</head>
<body>
<h1>Contacts</h1>
<table>
    <tr><th>Name</th><th>Email</th></tr>
    <c:forEach items="${contacts}" var="contact">
        <tr><td>${contact.name}</td><td>${contact.email}</td></tr>
    </c:forEach>
</table>
<a href="<c:url value="/contact/new"/>">New Contact</a>
</body>
</html>

```

Next the `contact.jsp` for adding new contacts.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!doctype HTML>
<html>
<head>
    <title>Spring Recipes - Contact Sample</title>
</head>
<body>
<h1>Contact</h1>
<form:form method="post" modelAttribute="contact">
    <fieldset>
        <legend>Contact Information</legend>
        <div>
            <div><form:label path="name">Name</form:label></div>
            <div><form:input path="name"/></div>
        </div>
        <div>
            <div><form:label path="email">Email Address</form:label></div>
            <div><form:input path="email" type="email"/></div>
        </div>
        <div><button>Save</button></div>
    </fieldset>
</form>
</form:form>
</html>
```

That is all the application code. What remains is some application configuration and a class to start the application. First, the configuration.

```
package com.apress.springrecipes.cloud.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan(basePackages = {"com.apress.springrecipes.cloud"})
@Configuration
public class ContactConfiguration {}
```

The application configuration is quite empty. It only defines a component scan annotation to detect the service and controller. Next a configuration for the web-related part is needed.

```
package com.apress.springrecipes.cloud.web;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
```

```

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
public class ContactWebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurator) {
        configurator.enable();
    }

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("redirect:/contact");
    }

    @Bean
    public InternalResourceViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}

```

The final missing part is the application initializer.

```

package com.apress.springrecipes.cloud.web;

import com.apress.springrecipes.cloud.config.ContactConfiguration;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class ContactWebApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

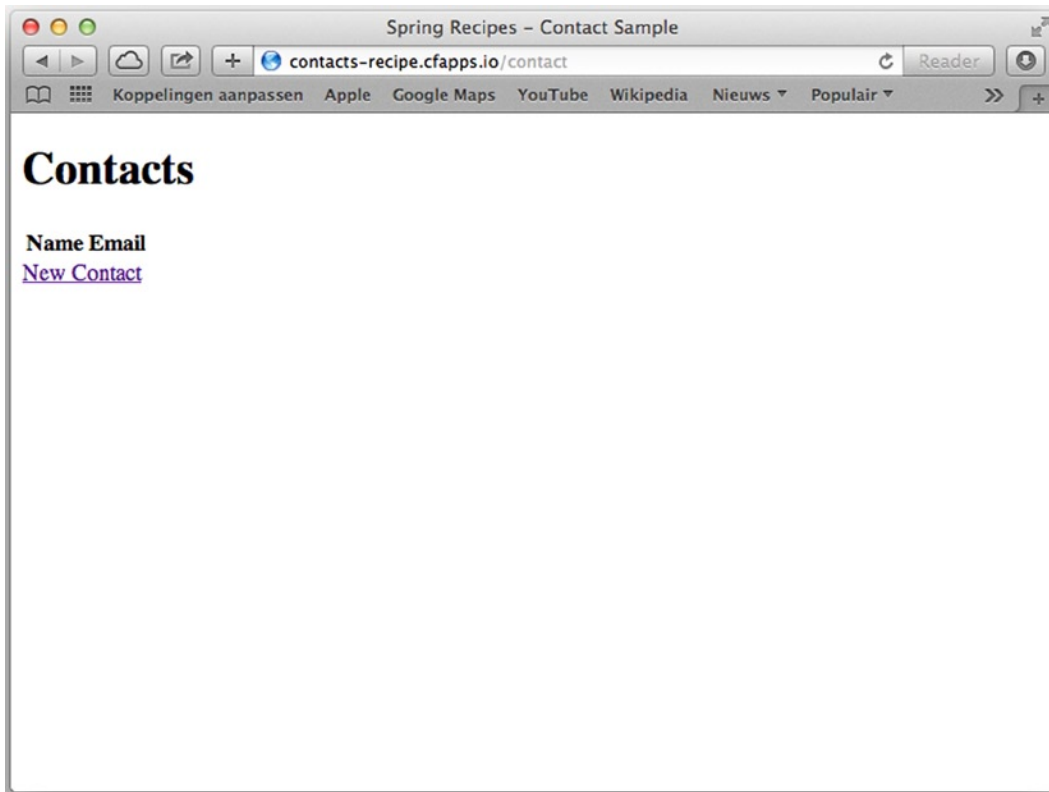
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {ContactConfiguration.class, ContactWebConfiguration.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] {"/"} ;
    }
}

```

Now everything is in place. After building the war file, push it to CloudFoundry by entering `cf push <application-name> -p contact.war` on the command line. This will show the progress of uploading, installation of the buildpack, and installation of Tomcat. After a successful deployment the application is available on `<application-name>.cfapps.io`. (see Figure A-9).



**Figure A-9.** Contact application on CloudFoundry

## Using a DataSource

At the moment the application stores the contact information in a `HashMap`. This is nice for testing purposes but for a real application the data needs to be stored in a database. First create a JDBC driven `ContactRepository` implementation.

```
package com.apress.springrecipes.cloud;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;
```

```

@Service
public class JdbcContactRepository extends JdbcDaoSupport implements ContactRepository {

    @Autowired
    public JdbcContactRepository(DataSource dataSource) {
        super.setDataSource(dataSource);
    }

    @Override
    public List<Contact> findAll() {
        return getJdbcTemplate().query("select id, name, email from contact",
            new RowMapper<Contact>() {
                @Override
                public Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
                    Contact contact = new Contact();
                    contact.setId(rs.getLong(1));
                    contact.setName(rs.getString(2));
                    contact.setEmail(rs.getString(3));
                    return contact;
                }
            });
    }

    @Override
    public void save(Contact c) {
        getJdbcTemplate().update("insert into contact (name, email) values (?, ?)", c.getName(),
            c.getEmail());
    }
}

```

Next update the configuration and add a `DataSource` and `DataSourceInitializer`.

```

package com.apress.springrecipes.cloud.config;

import com.apress.springrecipes.cloud.ContactRepository;
import com.apress.springrecipes.cloud.JdbcContactRepository;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;

import javax.sql.DataSource;

@ComponentScan(basePackages = {"com.apress.springrecipes.cloud"})
@Configuration
public class ContactConfiguration {

```

```

@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2).build();
}

@Bean
public DataSourceInitializer dataSourceInitializer(DataSource dataSource) {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScript(new ClassPathResource("/sql/schema.sql"));
    populator.setContinueOnError(true);

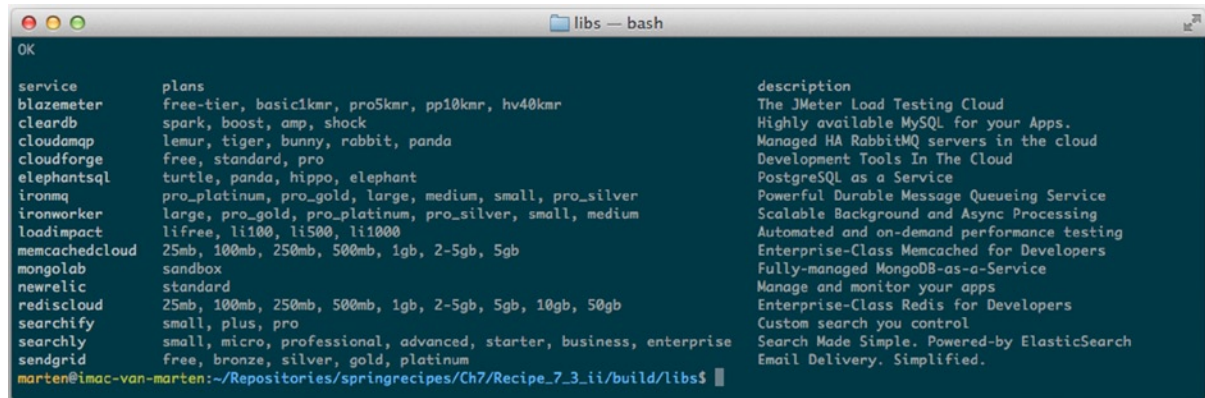
    DataSourceInitializer initializer = new DataSourceInitializer();
    initializer.setDataSource(dataSource);
    initializer.setDatabasePopulator(populator);
    return initializer;
}
}

```

For testing and local deployment the in-memory H2 database is used to configure this instance the `EmbeddedDatabaseBuilder` is used. The `DataSourceInitializer` takes care of executing the create script.

After building the war file again and deploying to CloudFoundry the application should still be running and using the in-memory database. However instead of the in-memory database, you want to use a real database, so that content survives redeployments, outages and so on.

CloudFoundry provides several services which are available in the marketplace. To get an overview type `cf marketplace` (or `cf m`) on the commandline (see Figure A-10).



**Figure A-10.** Overview of CloudFoundry services

As you can see there are different services like database implementations, messaging, and e-mail. For this recipe a database instance is needed, there are 2 database options MySQL and PostgreSQL. Choose which one you like and create an instance. To construct a basic MySQL instance type `cf create-service cleardb spark contacts-db`. After the database has been created you need to bind it (make it available for access) to your application type `cf bind-service contacts-recipe contacts-db`.

Now the database is ready to be used, to use it simply restart or redeploy the application.



CloudFoundry has a feature called auto-reconfiguration and this is enabled by default. It finds a bean of a certain type, in this case a `DataSource`, it will try to replace it with one provided by your configured services. This will however only work when there is a single service and a single bean of that type if you have multiple `DataSources` auto-reconfiguration won't work. It will work for all the provided services, like AMQP connection factories, and MongoDB instances.

## Accessing Cloud Service

Although auto-reconfiguration is a nice feature, as already mentioned, it doesn't always work. However there is an easy way to explicitly tell which service to use when deployed to CloudFoundry. Another nice thing that CloudFoundry does is that it activates a profile called `cloud`. This profile can be used to determine if the application is deployed on CloudFoundry or not and as such certain beans can be accessed specifically when deployed here.

First two additional dependencies are needed

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
  <version>1.1.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-spring-service-connector</artifactId>
  <version>1.1.0.RELEASE</version>
</dependency>
```

These two dependencies make it easy to interact with the CloudFoundry instance and the provided services. Now that these are available it is just a matter of a reconfiguration.

```
package com.apress.springrecipes.cloud.config;

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

@ComponentScan(basePackages = {"com.apress.springrecipes.cloud"})
@Configuration
public class ContactConfiguration {

...
    @Configuration
    @Profile("default")
    public class LocalDatasourceConfiguration {
        @Bean
        public DataSource dataSource() {
            return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2).build();
        }
    }

    @Configuration
    @Profile("cloud")
    public class CloudDatasourceConfiguration extends AbstractCloudConfig {
```

```

@Bean
public DataSource dataSource() {
    return connectionFactory().dataSource("contacts-db");
}
}

```

Notice the two inner configuration classes. Both have the `@Profile` annotation. The `LocalDataSourceConfiguration` is available when there is no cloud profile active. The `CloudDataSourceConfiguration` is only available when deployed to the cloud. The latter extends the `AbstractCloudConfig` class which has convenient methods to access the provided services. To get a reference to the datasource use the `dataSource` lookup method on the `connectionFactory`. For default services (DataSources, MongoDB, Redis, etc.) it provides convenient access methods. If you have developed and deployed custom services those can be accessed using the `generalService()` method.

After rebuilding the war and pushing it to CloudFoundry it will still be working.

## A-3 Removing an Application Problem

You want to remove an application from CloudFoundry.

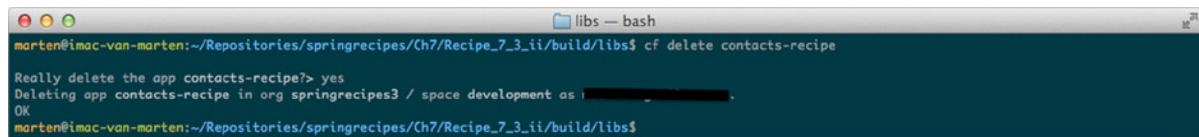
### Solution

Use the CF tools to delete the application from your space.

### How It Works

To remove an application issue a delete command, instead of push, to let CloudFoundry remove the application. To remove the application type `cf delete <application-name>` on the command line. After confirming that you really want to delete the application CloudFoundry will start to remove the application.

The output should look like that in Figure A-11.



**Figure A-11.** Output of removing an application from CloudFoundry

## Summary

In this chapter you explored how to deploy and remove an application to the cloud platform provided by Pivotal, CloudFoundry. First a basic web application without any external connections was deployed.

After that a datasource was added and you learned how to create and bind a service to your application. As soon as the service was available you experienced the auto-reconfiguration feature provided by CloudFoundry. Finally you explored how to interact with the Cloud from within your application configuration and not to rely on auto-reconfiguration.