Spring for the Cloud

Buiding Spring Applications to run on a PAAS

Paul Chapman – Senior Consultant Trainer
Lawrence Crowther – Field Engineering Manager

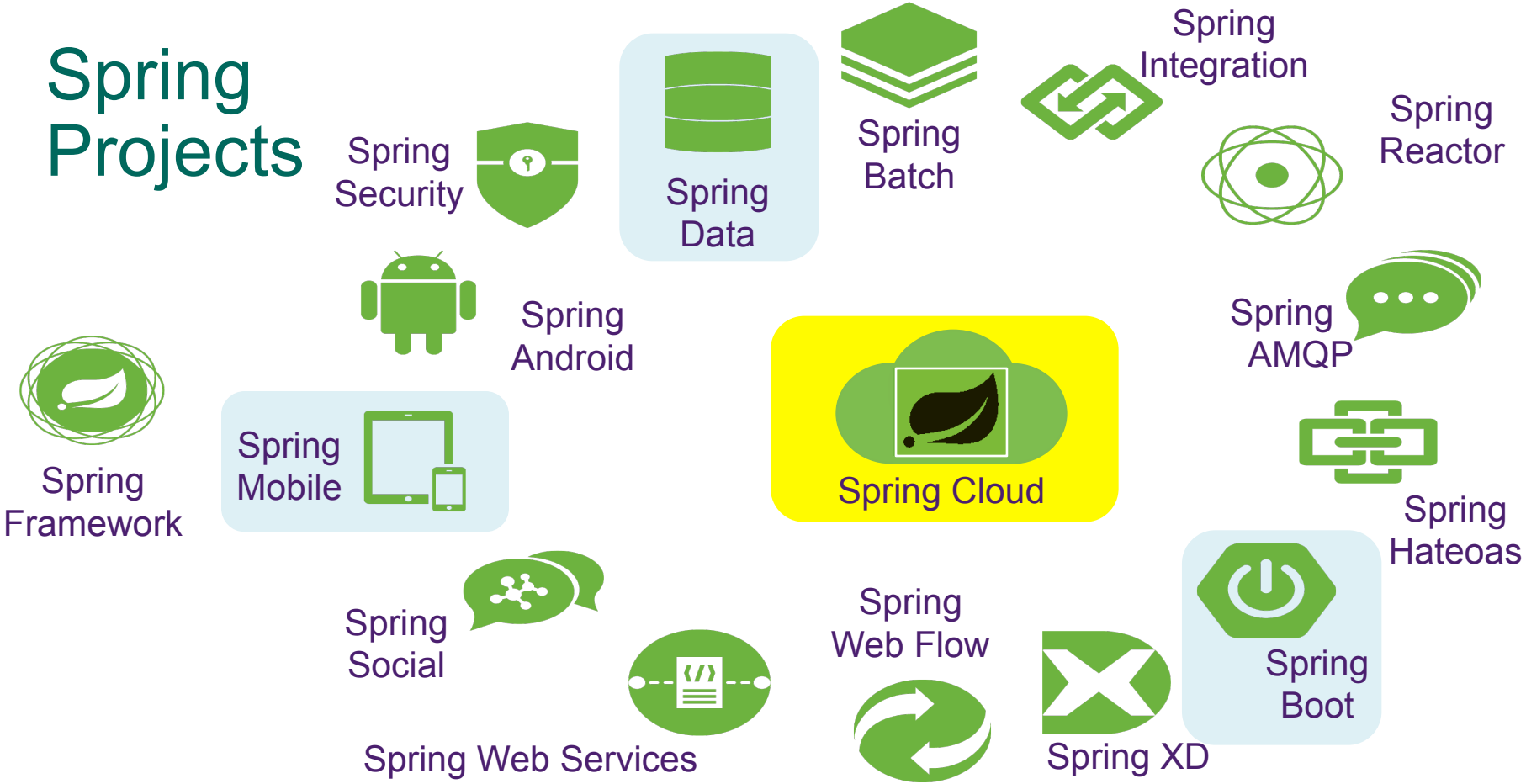
Pivotal

Agenda

- **Spring Projects**
- Spring Data
- Spring Mobile
- Spring Boot
- Spring for the Cloud

Pivotal

Spring Projects



Pivotal

Agenda

- Spring Projects
- **Spring Data**
- Spring Mobile
- Spring Boot
- Spring for the Cloud

Pivotal

Spring Data – Instant Repositories

- Just one feature of Spring Data
- How?
  - Annotate domain classes to define keys and enable persistence
    - You do this for JPA anyway
    - Spring Data provides similar annotations for Mongo DB, Neo4J, Gemfire …
  - Define your repository as an *interface*
- Spring will implement it at run-time
  - Scans for interfaces extending **Repository <T, K>**

@NodeEntity

@Document

@Entity

@Region

Pivotal

Define Your Repository Interface

- Auto-generated finders obey naming convention
  - findBy*<DataMember><Op>* : <Op> is Gt, Lt, Ne, Between, Like …

```
public interface CustomerRepository
            extends Repository<Customer, Long> {

    public <S extends T> save(S entity);
    public Customer findByEmail(String someEmail);     // No <Op> for Equals
    public Customer findByFirstOrderDateGt(Date someDate);
    public Customer findByFirstOrderDateBetween(Date d1, Date d2);

    @Query("SELECT c FROM Customer c WHERE c.email NOT LIKE '%@%'")
    public List<Customer> findInvalidEmails();
}
```

ID

Predefined signatures for save, and delete

Custom query uses query-language of underlying product (here JPQL)

Pivotal

Spring Data – Instant Repositories – 1

- Get Spring to scan for interfaces
  - Implemented on-the-fly

```
<jpa:repositories base-package="com.acme.**.repository" />

<mongo:repositories base-package="com.acme.**.repository" />

<gfe:repositories base-package="com.acme.**.repository" />
```

- Now just inject a dependency of type `CustomerRepository`

```
@Autowired CustomerRespository custRepo;
```

Pivotal

Making it Restful

- Repositories can be automatically exported as RESTful
  - Exposes `…/orders`, `…/orders/{id}` to `GET`, `POST`, `PUT` and `DELETE`
    - ***If*** underlying methods exist on the repository

GET http://localhost:8080/myapp

```
{
  "links" : [ {
      "rel" : ”orders",
      "href" : "http://localhost:8080/myapp/orders"
  }, "content" : [ ]
}
```

```
@RestResource(path = ”customers", rel = ”customers")
public interface CustomerRepository
                extends Repository<Customer, Long> { ... }
```

Pivotal

Agenda

- Spring Projects
- Spring Data
- **Spring Mobile**
- Spring Boot
- Spring for the Cloud

Pivotal

Spring Mobile and Social

- Spring Mobile
  - MVC interceptor to enable device detection and site-preferences
  - Automatically redirect to “mobile” site
  - Device object available to MVC controllers

SPRING MOBILE

- Spring Social
  - Handle user-accounts
  - Login using Spring Security OAuth
  - Sub-projects for accessing Facebook, Twitter, LinkedIn data

SPRING SOCIAL

Pivotal

Mobile Web Strategies

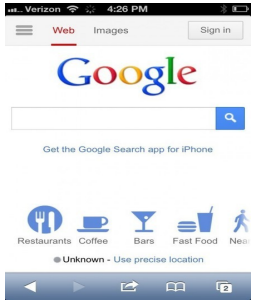
- There are many mobile web strategies:
  - Separate web sites for normal vs mobile.
  - Responsive Web Design
  - Mobile First
- Often a *separate mobile-facing site* is used
  - Navigation, resource needs very different.
  - No time or budget to do a full RWD approach.

Pivotal

Separate Web Sites

Request: google.com

Redirect:
m.google.com



Spring Mobile

- Subproject within Spring Family
  - Plugs into Spring MVC
  - Can be used with other frameworks

spring.io/spring-mobile

- Features
  - Server-side device detection / resolution
    - *Is the current request from a mobile device? Tablet?*
  - Site preference management
    - *Does the user wish to use the "Full Site"?*
  - Site switching
    - *Redirect this request to m.myapp.com*

Pivotal

Interceptor to Determine Device

- Determines if request is mobile, tablet, or other
  - Based on UserAgent request header
  - Built-in strategies: Simple, WURFL (web-device registry)
  - Places a *Device* object in request attributes.

```
<mvc:interceptors>
  <!-- On pre-handle, resolve the originating device  -->
  <bean class="org.sf.mobile.device.
                 DeviceResolverHandlerInterceptor" />
</mvc:interceptors>
```

Pivotal

Using Device in Controller

- Also available to @ControllerAdvice or your own interceptor

```
@Controller
public class HomeController {
    @RequestMapping("/")
    public void home(Device device) {

        if (device.isMobile()) {
            ...
        } else if (device.isTablet()) {
            ...
        } else {
            ...
    }
}
```

```
public interface Device {

    // True if not a mobile or tablet
    boolean isNormal();

    boolean isMobile();

    boolean isTablet();
}
```

Pivotal

Interceptor for Site Switching

- Redirects mobile-originating requests to mobile-specific URL
  - Create using factory-methods
    - mDot – Redirects to m.${serverName}
    - dotMobi – Redirects to ${serverName}.mobi.com
    - urlPath – Redirects to ${serverName}/${rootPath}

```
<mvc:interceptors>  <!-- Redirects mobile users to "m.myapp.com" -->
  <bean class="org.springframework.mobile.device.switcher.
          SiteSwitcherHandlerInterceptor" factory-method="mDot">
    <constructor-arg value="myapp.com"/>  <!- Server name -->
  </bean>
</mvc:interceptors>
```

Pivotal

Interceptor for Site Preference

- Allow user to override site switcher
  - Switch between mobile and full-site
  - Uses Cookie by default, could get from session if users login

```
<mvc:interceptors>
 <bean class="...mobile.device.site.SitePreferenceHandlerInterceptor/>
</mvc:interceptors>
```

Same mechanism as theme and Locale switching in Spring MVC

Agenda

- Spring Projects
- Spring Data
- Spring Mobile
- **Spring Boot**
- Spring for the Cloud

Pivotal

Spring Boot

- Write a deployable Spring application in just a few lines
  - Many defaults out of the box
  - Override any/all as needed
  - Still need to setup your build environment (ant, maven, gradle)
  - Can run as a Java application
    - Deploys to embedded container (Tomcat by default)
  - Or deploy as a war in usual way
  - Java Configuration or XML supported plus autowiring and component-scanning

SPRING BOOT

Pivotal

Simplest Program (Standalone)

- Just one class – an MVC Controller

```
@Controller
@EnableAutoConfiguration
public class SampleController {

  @RequestMapping("/")
  @ResponseBody
  String home() {
    return "<body><h1>Hello World!</h1><p>Instant web-app</p></body>";
  }

  public static void main(String[] args)    throws Exception {
    SpringApplication.run(SampleController.class, args);
  }
}
```

This annotation tells Spring Boot to use all its defaults

Initial class to run

Run as Java application and got to http://localhost:8080/

Pivotal

Extend: Run as WAR, Pull-in extra code

- Extend `SpringBootServletInitializer`
  - Use component scanning, Java and/or XML to define setup
    - Controllers, Services, Repositories, Messaging, DataSource …

```
@EnableAutoConfiguration
@ComponentScan({ "org.project.myapp" })
@ImportResource(value = "classpath:config/security.xml")
public class Main extends SpringBootServletInitializer {

  @Override
  protected SpringApplicationBuilder configure
                        (SpringApplicationBuilder app) {
    return new app.sources(Main.class);
  }
}
```

Tells Spring Boot to run up as a war

Initialising class (itself)

Pivotal

Agenda

- Spring Projects
- Spring Data
- Spring Mobile
- Spring Boot
- **Spring for Cloud**

Pivotal

Other Spring Projects You Might Need

- Spring Integration
  - Orchestrate interactions with local and remote services
- Spring XD and Batch
  - Import/export large data-volumes
    - Data analysis, personalization

- Spring AMQP
  - Open-source messaging to get data in/out of your cloud

- Spring Security, Spring Social
  - Application security, OAuth, Integrate with Facebook, Twitter …

Pivotal

What are Services?

- Services:
  - Provide *external* functionality to your applications
    - Examples - Databases (MySQL, Mongo, Redis), Messaging (Rabbit/MQ)
    - Provisioned alongside an application.
    - May be shared among many applications
  - Are *bound* to (associated with) an application
    - Using a “Service Broker”
  - Application provides *connection information* via properties
  - Most PAAS provide “out of the box” or “marketplace” services
    - CF provides MySQL, Postgres, Rabbit/MQ, MongoDB, Redis …
    - And you can add your own (such as Oracle)

Pivotal

What is Spring Cloud?

- A simple way for JVM apps to
  - Access cloud services
  - Discover their own information during runtime
  - Special support for Spring apps
- Works with multiple clouds and cloud services
  - *Without* modifying the application
  - Currently Cloud Foundry and Heroku, more coming

Core Components – I

- **Cloud Connector**
  - An interface between application and cloud
  - Implemented by a cloud provider for their PAAS
  - Currently two connectors exist: CloudFoundry and Heroku
- **Service Connector**
  - An object, such as javax.sql.DataSource, that represents a connection to a service offered by the PAAS

Pivotal

Core Components – II

- **Service information**
  - Information about the underlying service such as host, port, and credentials.
- **Application information**
  - Information about application and instance in which these libraries are embedded.

Pivotal

Tying it all Together

- Integrate application with your Cloud and its Services

CloudConfig.java

Default profile

Java Config

```
@Bean
@Profile("cloud")
public DataSource dataSource() {
    CloudEnvironment ce = new CloudEnvironment();
    RdbmsServiceInfo si =
         ce.getServiceInfo("mysql", RdbmsServiceInfo.class);
    return (new RdbmsServiceCreator()).createService(si);
}
```

XMLConfig

```
<beans profile="cloud">
    <cloud:data-source id="dataSource" service="mysql" />
```

cloud.xml

Pivotal

Now Integrate with Your Repositories

```
<jpa:repositories base-package="com.acme.**.repository”
                  transaction-manager-ref="emf"
                  entity-manager-factory-ref="emf" />

<bean id="emf" class="...LocalContainerEntityManagerFactoryBean”>
    <property name="dataSource" ref=“dataSource” />
    ...
</bean>

<bean id="transactionManager" class="...JpaTransactionManager" >
    <property name="entityManagerFactory" ref="emf"/>
</bean>

<tx:annotation-driven/>
```

app-config.xml

Pivotal

Or Use Java Configuration

```
@Configuration   @EnableJpaRepositories   @EnableTransactionManagement
class ApplicationConfig {

   @Autowired DataSource dataSource;

   @Bean public EntityManagerFactory entityManagerFactory() {
      LocalContainerEntityManagerFactoryBean factory =
         new LocalContainerEntityManagerFactoryBean();
      ...
      return factory.getObject();
   }

   @Bean public PlatformTransactionManager transactionManager() {
      JpaTransactionManager txManager = new JpaTransactionManager();
      txManager.setEntityManagerFactory(entityManagerFactory());
      return txManager;
   }
}
```

AppConfig.java

Run Using Spring Boot

```
@EnableAutoConfiguration
@ComponentScan({ "org.project.myapp" })
@Import({ AppConfig.class, CloudConfig.class })
@ImportResource({ "classpath:config/app-config.xml”,
                  "classpath:config/app-cloud.xml”})
public class MyApp extends SpringBootServletInitializer {

  @Override
  protected SpringApplicationBuilder configure
                      (SpringApplicationBuilder app) {
    return new app.sources(MyApp );
  }
}
```

Java Config

OR
XML Config

Next step?

Deployment to your Cloud

Snack break first …

Next Spring training 07-Apr 2014:

http://gopivotal.com/training#spring

Pivotal

BUILT FOR THE SPEED OF BUSINESS