Core Spring

Four Day Workshop

Building Enterprise Applications using Spring

Version 4.3.b

Copyright Notice

This Page Intentionally Left Blank

This Page Intentionally Left Blank

spring

Welcome to Core Spring

A 4-day bootcamp that trains you how to use the Spring Framework to create well-designed, testable, business, applications

spring Pivotal

Logistics

- Student introductions
- Self introduction
- Course registration (if needed)
- Courseware
- Internet access
- Phones on silent

- Working hours
- Lunch and breaks
- Toilets/Restrooms
- Fire alarms
- Emergency exits
- Any other questions?

spring Pivotal

How You will Benefit

- Learn to use Spring for web and other applications
- Gain hands-on experience
  - Generous mixture of presentation and labs
- Access to experienced, certified instructors

Covered in this section

- **Agenda**
- Spring and Pivotal

Course Agenda: Day 1

- Introduction to Spring
- Using Spring to configure an application
- Java-based dependency injection
- Annotation-based dependency injection
- XML-based dependency injection
- Spring FactoryBeans

1

spring Pivotal

Course Agenda: Day 2

- Understanding the bean life-cycle
- Testing a Spring-based application using multiple profiles
- Adding behavior to an application using aspects
- Introducing data access with Spring
- Simplifying JDBC-based data access

2

spring Pivotal

Course Agenda: Day 3

- Driving database transactions in a Spring environment
- Introducing object-to-relational mapping (ORM)
- Working with JPA in a Spring environment
- Effective web application architecture
- Getting started with Spring MVC

3

spring Pivotal

Course Agenda: Day 4

- Rapidly start new projects with Spring Boot
- Securing web applications with Spring Security
- Implementing REST with Spring MVC
- Microservices and Cloud Native Applications using Spring Cloud

4

spring Pivotal

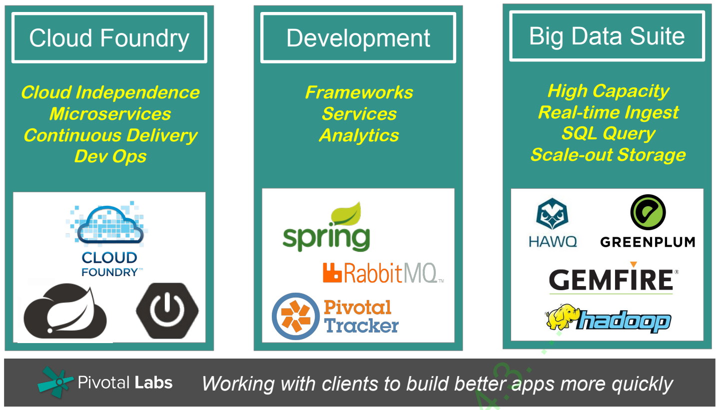
Covered in this section

- Agenda
- **Spring and Pivotal**

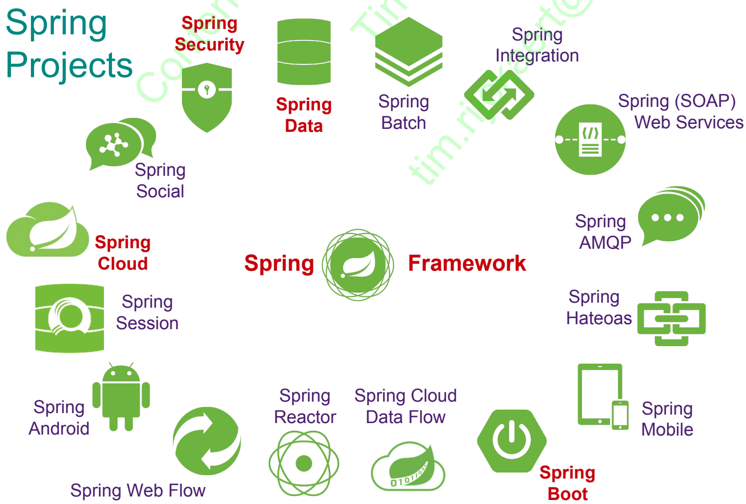
Spring and Pivotal

- SpringSource, the company behind Spring
  - acquired by VMware in 2009
  - transferred to Pivotal joint venture 2013
- Spring projects key to Pivotal's big-data and cloud strategies
  - Virtualize your Java Apps
    - Save license cost
    - Deploy to private, public, hybrid clouds
  - Real-time analytics
    - Spot trends as they happen
    - Spring Data, Spring Hadoop, Spring XD & Pivotal HD

The Pivotal World



Spring Projects



Covered in this section

- Agenda
- Spring and Pivotal

Let's get on with the course..!

Overview of the Spring Framework

Introducing Spring in the Context of Enterprise Application Architecture

What is Spring and why would you use it?

spring Pivotal

Objectives

- After completing this lesson, you should be able to:
  - Define the Spring Framework
  - Understand what Spring is used for
  - Understand why Spring is successful
  - Explain where it fits in your world

spring Pivotal

Topics in this session

- **What is the Spring Framework?**
- Spring is a Container
- Spring Framework history
- What is Spring Used For?

What is the Spring Framework?

- Spring is an Open Source, Lightweight, Container and Framework for building Java enterprise applications

- Open Source
- Lightweight
- Container
- Framework

What is the Spring Framework?
Open Source

- Spring binary and source code is freely available
- Apache 2 license
- Code is available at:
  - https://github.com/spring-projects/spring-framework
- Binaries available at Maven Central
  - http://mvnrepository.com/artifact/org.springframework
- Documentation available at:
  - http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle

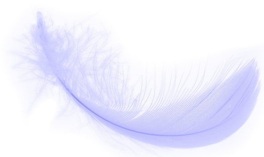
Bookmark This!

The use of a transitive dependency management system (Maven, Gradle, Ant/Ivy) is recommended for any Java application

spring Pivotal

What is the Spring Framework?
Lightweight

- Spring applications do not require a Java EE application server
  - But they can be deployed on one
- Spring is not *invasive*
  - Does not require you to extend framework classes or implement framework interfaces for most usage
  - You write your code as POJOs
- Low overhead
  - Spring jars are relatively small
    - JARs used in this course are < 8 MB



spring Pivotal

What is the Spring Framework?
Container

- Spring serves as a container for your application objects.
  - Your objects do not have to worry about finding / connecting to each other.
- Spring instantiates and dependency injects your objects
  - Serves as a lifecycle manager

What is the Spring Framework?
Framework

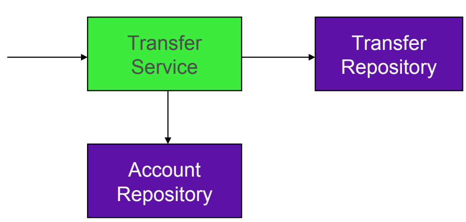
- Enterprise applications must deal with a wide variety of technologies / resources
  - JDBC, JMS, AMQP, Transactions, ORM / JPA, NoSQL, Security, Web, Tasks, Scheduling, Mail, Files, XML/JSON Marshalling, Remoting, REST services, SOAP services, Mobile, Social, ...
- Spring provides framework classes to simplify working with lower-level technologies

Topics in this session

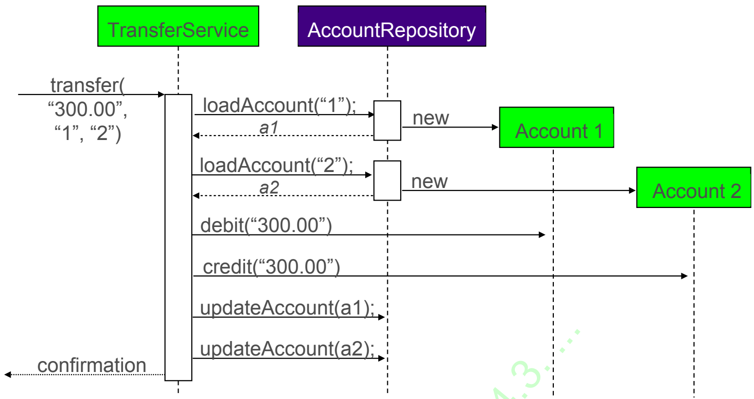
- What is the Spring Framework?
- **Spring is a Container**
- Spring Framework History
- What is Spring Used For?

Application Configuration

- A typical application system consists of several parts working together to carry out a use case



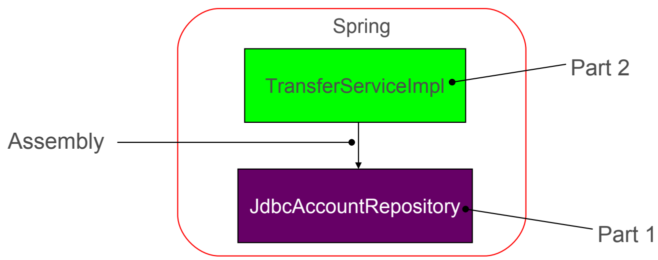
Example: Money Transfer System



Spring's Configuration Support

- Spring provides support for assembling such an application system from its parts
  - Parts do not worry about finding each other
  - Any part can easily be swapped out

Money Transfer System Assembly




```
(1) repository = new JdbcAccountRepository(…);
(2) service = new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

Parts are Just *Plain Old Java Objects* (POJOs)

Part 1

```
public class JdbcAccountRepository implements
    AccountRepository {
    …
}
```

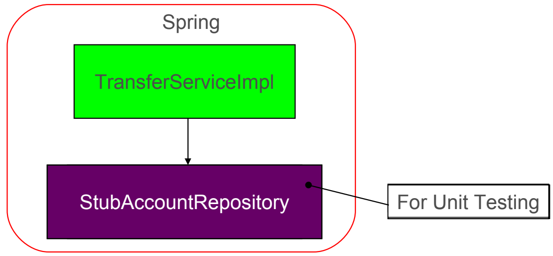
Implements a service (business) interface

Part 2

```
public class TransferServiceImpl implements TransferService {
    private AccountRepository accountRepository;

    public void setAccountRepository(AccountRepository ar) {
        accountRepository = ar;
    }
    …
}
```

Depends on an *interface:*
– conceals complexity of implementation;
– allows for swapping out implementation

Swapping Out Part Implementations




```
(1) new StubAccountRepository();
(2) new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

spring Pivotal

Topics in this session

- What is the Spring Framework?
- Spring is a Container
- **Spring Framework History**
- What is Spring Used For?

spring Pivotal

Why is Spring Successful?
A brief history of Java

- The early years:
  - 1995 – Java introduced, Applets are popular
  - 1997 – Servlets introduced
    - Efficient, dynamic web pages become possible.
  - 1999 – JSP introduced
    - Efficient, dynamic web pages become easy.

- Questions arise regarding “Enterprise” applications
  - How should a Servlet / JSP application handle:
    - Persistence?
    - Transactions?
    - Security?
    - Business Logic?
    - Messaging?
    - Etc.?

spring Pivotal

Introducing J2EE and EJB

- Java's answer: J2EE
  - 1999 – J2EE introduced
    - Featuring Enterprise Java Beans (EJB)
    - Answers the questions of persistence, transactions, business logic, security, etc
- However EJBs prove to be problematic:
  - Difficult to code.
    - Must extend / implement specific classes /interfaces
    - Complicated programming model required
  - Difficult to unit test
  - Expensive to run
    - Must have application server, resource intensive

spring Pivotal

The Birth of Spring

- Rod Johnson publishes J2EE Development without EJB
- 2004 - Spring Framework 1.0 released
  - Champions dependency injection
  - Encourages POJOs
  - Uses XML files to describe application configuration
  - Becomes popular quickly as an EJB alternative

Spring Framework History

- Spring 2.0 (2006):
  - XML simplification, async JMS, JPA, AspectJ support
- Spring 2.5 (2007, last release 2.5.6)
  - Requires Java 1.4+ and supports JUnit 4
  - Annotation DI, @MVC controllers, XML namespaces
- Spring 3.x (3.2.17 released July 2016)
  - Environment & Profiles, @Cacheable, @EnableXXX …
  - Requires Java 1.5+ and JUnit 4.7+
  - REST support, JavaConfig, SpEL, more annotations
- Spring 4.x (released Dec 2013)
  - Support for Java 8, @Conditional, Web-sockets
- Spring 5.x (2017)
  - Reactive programming focus

Topics in this session

- What is the Spring Framework?
- Spring is a Container
- Spring Framework History
- **What is Spring Used For?**

What is Spring Used For?

- Spring provides comprehensive infrastructural support for developing enterprise Java™ applications
  – Spring deals with the plumbing
  – So you can focus on solving the domain problem

- Spring used to build enterprise applications dealing with:

The Current World

- Spring is not simply an alternative to J2EE / EJB
  - Modern application development challenges are different today than 2000
- Spring continues to innovate
  - **Web:** AJAX, WebSockets, REST, Mobile, Social
  - **Data:** NoSQL, Big Data, Stream processing
  - **Cloud:** Distributed systems, Cloud, Microservices
  - **Productivity:** Spring Boot, Spring Cloud Data Flow
  - and many more

More on Spring's Ecosystem

- Visit http://spring.io/projects

Lab

Developing an Application from Plain Java Objects

Dependency Injection Using Spring

Introducing the Spring Application Context and Spring's Java Configuration capability

@Configuration and ApplicationContext

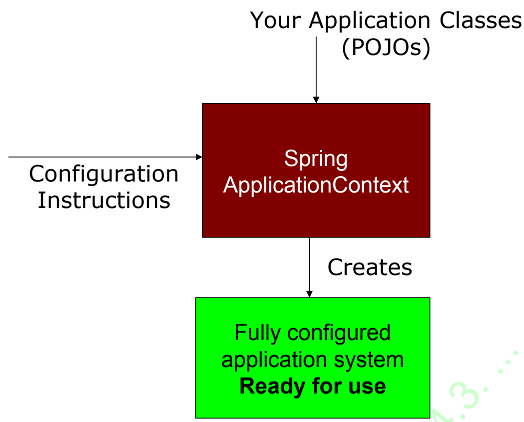
spring Pivotal

Topics in this session

- **Spring quick start**
- Creating an application context
- Bean scope
- Lab

spring Pivotal

How Spring Works



Your Application Classes

```
public class TransferServiceImpl implements TransferService {
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    …
}
```

Needed to load accounts from the database

Configuration Instructions

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService() {
    return new TransferServiceImpl( accountRepository() );
  }
  @Bean public AccountRepository accountRepository()  {
    return new JdbcAccountRepository( dataSource() );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
    dataSource.setUsername("transfer-app");
    dataSource.setPassword("secret45");
    return dataSource;
  }
}
```


Creating and Using the Application

```
// Create the application from the configuration
ApplicationContext context =
   SpringApplication.run( ApplicationConfig.class );

// Look up the application service interface
TransferService service =
   (TransferService) context.getBean(“transferService”);

// Use the application
service.transfer(new MonetaryAmount(“300.00”), “1”, “2”);
```

Bean ID
Based on method name

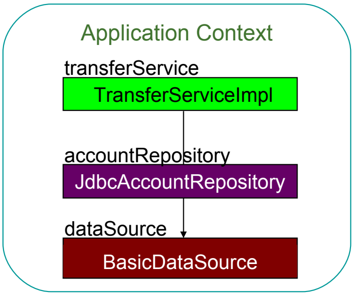
Accessing a Bean

- Multiple ways

```
ApplicationContext context = SpringApplication.run(...);

// Classic way: cast is needed
TransferService ts1 = (TransferService)  context.getBean("transferService");

// Use typed method to avoid cast
TransferService ts2 =  context.getBean("transferService", TransferService.class);

// No need for bean id if type is unique
TransferService ts3 =  context.getBean(TransferService.class );
```

Inside the Spring Application Context

```
// Create the application from the configuration
ApplicationContext context =
    SpringApplication.run( ApplicationConfig.class )
```



Bean Descriptions

- Allows you to provide helpful information about any bean

```
@Bean
@Description("Handles all transfer related use-cases")
public TransferService transferService() { … }

@Bean
@Description("Provides access to data from the Accounts table")
public AccountRepository accountRepository() { … }

@Bean
@Description("Data-source for the underlying RDB we are using")
public DataSource dataSource() { … }
```

spring Pivotal

Quick Start Summary

- Spring separates application configuration from application objects
- Spring manages your application objects
  - Creating them in the correct order
  - Ensuring they are fully initialized before use
- Each bean is given a unique id / name
  - Should reflect service or role the bean provides to clients
  - Bean ids should not contain implementation details

spring Pivotal

Topics in this session

- Spring quick start
- **Creating an application context**
- Multiple Configuration Files
- Bean scope
- Lab

Creating a Spring Application Context

- Spring application contexts can be bootstrapped in any environment, including
  - JUnit system test
  - Web application
  - Standalone application

Application Context Example
Instantiating Within a System (Integration) Test

```
public class TransferServiceTests {
  private TransferService service;

  @Before public void setUp() {
    // Create the application from the configuration
    ApplicationContext context =
      SpringApplication.run( ApplicationConfig.class )
    // Look up the application service interface
    service = context.getBean(TransferService.class);
  }

  @Test public void moneyTransfer() {
    Confirmation receipt =
      service.transfer(new MonetaryAmount("300.00"), "1", "2"));
    Assert.assertEquals("500.00", receipt.getNewBalance());
  }
```

Bootstraps the system to test

Tests the system

spring Pivotal

Topics in this session

- Spring quick start
- Creating an application context
- **Multiple Configuration Files**
- Bean scope
- Lab

spring Pivotal

Creating an Application Context from Multiple Files

- Your @Configuration class can get very long
  - Instead use *multiple* files combined with @Import
  - Defines a single Application Context
    - With beans sourced from multiple files

```
@Configuration
@Import({InfrastructureConfig.class, WebConfig.class })
public class ApplicationConfig {
    ...
}
```

```
@Configuration
public class InfrastructureConfig {
    ...
}
```

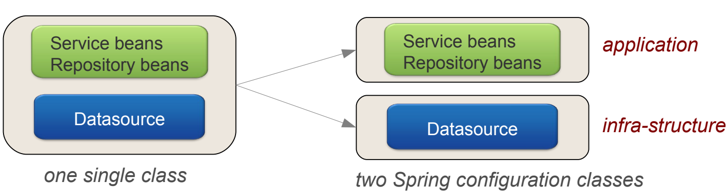
```
@Configuration
public class WebConfig {
    ...
}
```

spring Pivotal

Creating an Application Context from Multiple Files

- Organize your @Configuration classes however you like

- Best practice: separate out “application” beans from “infrastructure” beans
  - Infrastructure often changes between environments



spring Pivotal

Mixed Configuration

```
@Configuration
public class ApplicationConfig {

  @Bean public TransferService transferService()
     { return new TransferServiceImpl( accountRepository() ); }

  @Bean public AccountRepository accountRepository()
     { return new JdbcAccountRepository( dataSource() );  }

  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
    dataSource.setUsername("transfer-app");
    dataSource.setPassword("secret45" );
    return dataSource;
  }
}
```

application beans

Coupled to a local Postgres environment

infrastructure bean

Partitioning Configuration

```
@Configuration
public class ApplicationConfig {
  @Autowired DataSource dataSource;

  @Bean public TransferService transferService() {
     return new TransferServiceImpl ( accountRepository() );
  }

  @Bean public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource );
  }
}
```

application beans

```
@Configuration
public class TestInfrastructureConfig {
  @Bean public DataSource dataSource() {
    ...
  }
}
```

infrastructure bean

Referencing Beans Defined in Another File

- Use @Autowired to reference a bean defined elsewhere

```
@Configuration
@Import( InfrastructureConfig.class )
public class AppConfig {
  private final DataSource dataSource;

  @Autowired
  public AppConfig(DataSource ds) {
    this.dataSource = ds;
  }

  @Bean
  public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource );
  }
}
```

```
@Configuration
public class InfrastructureConfig {
  @Bean
  public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

Pre Spring 4.3: Auto-wire a property setter, can't use a constructor

Referencing Dependencies Via Arguments

- *Alternative:* Define @Bean method arguments
  - Spring finds bean that matches type & populates the argument

```
@Configuration
@Import( InfrastructureConfig.class )
public class ApplicationConfig {
  @Bean
  public AccountRepository accountRepository( DataSource dataSource ) {
    return new JdbcAccountRepository( dataSource );
  }
}
```

```
@Configuration
public class InfrastructureConfig {
  @Bean public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

… But Avoid "Tramp Data"

Bad: dataSource is a "*tramp*"!

```
@Configuration
@Import( InfrastructureConfig.class )
public class ApplicationConfig {
  @Bean public AccountService accountService( DataSource dataSource ) {
    return new accountService( accountRepository(dataSource) );
  }

  @Bean public AccountService accountRepository( DataSource dataSource ) {
    return new accountRepository( dataSource );
  }
}
```

tramp

tramp

Better: Pass *actual* dependency

```
@Configuration
@Import( InfrastructureConfig.class )
public class ApplicationConfig {
  @Bean  public AccountService accountService( AccountRepository repo ) {
    return new accountService( repo );
  }
  @Bean public AccountService accountRepository( DataSource ds ) {
    return new accountRepository( ds );
  }
}
```


Beware Duplicate Beans

Use **@Order** annotation to control which order **@Bean** methods run (since Spring 4.2)

- It is *not* illegal to define the same bean more than once
  - You get the last bean Spring sees defined

```
@Configuration
public class Config1 {
  @Bean
  public String example() {
    return new String("example1");
  }
}
```

```
@Configuration
public class Config2 {
  @Bean
  public String example() {
    return new String("example2");
  }
}
```

```
@Import({ Config1.class, Config2.class })
public class TestApp {
  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(TestApp.class);
    System.out.println("Id=" + context.getBean("example"));
  }
}
```

Console output is *Id=example2*

Topics in this session

- Spring quick start
- Creating an application context
- Multiple Configuration Files
- **Bean scope**
- Lab

spring Pivotal

Bean Scope: default

service1 == service2

- Default scope is *singleton*

```
@Bean
public AccountService accountService() {
  return ...
}
```

```
@Bean
@Scope("singleton")
public AccountService accountService() {
  return ...
}
```

One single instance

```
AccountService service1 = (AccountService)  context.getBean("accountService");
AccountService service2 =  (AccountService)  context.getBean("accountService");
assert service1 == service2;  // True – same object
```

spring Pivotal

Bean Scope: prototype

service1 != service2

- Scope "prototype"
  - New instance created every time bean is referenced

```
@Bean
@Scope("prototype")
public AccountService accountService() {
    return …
}
```

@Scope(scopeName="prototype")

```
AccountService service1 = (AccountService) context.getBean("accountService");
AccountService service2 = (AccountService) context.getBean("accountService");
assert service1 != service2; // True – different objects
```

TWO instances

spring Pivotal

Common Spring Scopes

- The most commonly used scopes are:

singleton	A single instance is used
prototype	A new instance is created each time the bean is referenced
session	A new instance is created once per user session - web environment only
request	A new instance is created once per request – web environment only

spring Pivotal

Other Scopes

- Spring has other more specialized scopes
  - Web Socket scope
  - Refresh Scope
  - Thread Scope (defined but not registered by default)
- Custom scopes (rarely)
  - You define a factory for creating bean instances
  - Register to define a custom scope name
- All are outside the scope of this course

spring Pivotal

Dependency Injection Summary

- Your object is handed what it needs to work
  - Frees it from the burden of resolving its dependencies
  - Simplifies your code, improves code reusability
- Promotes programming to interfaces
  - Conceals implementation details of dependencies
- Improves testability
  - Dependencies easily stubbed out for unit testing
- Allows for centralized control over object lifecycle
  - Opens the door for new possibilities

spring Pivotal

Lab

Using Spring to Configure an Application

spring Pivotal

Dependency Injection Using Spring 2

Deeper Look into Spring's Java Configuration Capability

External Properties, Profiles and Proxies

spring Pivotal

Topics in this session

- **External Properties**
- Profiles
- Spring Expression Language
- Proxying

spring Pivotal

Setting property values

- Consider this bean definition from the last chapter:

```
@Bean
public DataSource dataSource() {
  DataSource ds = new BasicDataSource();
  ds.setDriverClassName("org.postgresql.Driver");
  ds.setUrl("jdbc:postgresql://localhost/transfer" );
  ds.setUser("transfer-app");
  ds.setPassword("secret45" );
  return ds;
}
```

- Unwise to hard-code DB connection parameters
  – "Externalize" these to a properties file

Spring's Environment Abstraction – 1

- **Environment** object used to obtain properties from runtime environment
- Properties from many sources:
  – JVM System Properties
  – Java Properties Files
  – Servlet Context Parameters
  – System Environment Variables
  – JNDI

Spring's Environment Abstraction – 2

```
@Configuration
public class DbConfig {
   private static final String DB_DRIVER = "db.driver";
   private static final String DB_URL = "db.url";
   private static final String DB_USER = "db.user";
   private static final String DB_PWD = "db.password";

  @Autowired public Environment env;

  @Bean public DataSource dataSource() {
     DataSource ds = new BasicDataSource();
     ds.setDriverClassName(  env.getProperty( DB_DRIVER ));
     ds.setUrl( env.getProperty( DB_URL ));
     ds.setUser( env.getProperty( DB_USER ));
     ds.setPassword( env.getProperty( DB_PWD ));
     return ds;
  }
}
```

Property names

Fetch property values from environment

spring Pivotal

Property Sources

- Environment obtains values from "property sources"
  - *Environment Variables* and *Java System Properties* always populated automatically
  - **@PropertySource** contributes *additional* properties
  - Available resource prefixes: classpath: file: http:

```
@Configuration
@PropertySource ( "classpath:/com/organization/config/app.properties" )
@PropertySource ( "file:config/local.properties" )
public class ApplicationConfig {
   ...
}
```

Adds properties from these files *in addition* to environment variables and system properties

spring Pivotal

Accessing Properties using @Value

```
@Configuration
public class DbConfig {

   @Bean
   public DataSource dataSource(
         @Value("${db.driver}") String driver,
         @Value("${db.url}") String url,
         @Value("${db.user}") String user,
         @Value("${db.password}") String pwd) {
      DataSource ds = new BasicDataSource();
      ds.setDriverClassName(  driver);
      ds.setUrl( url);
      ds.setUser( user);
      ds.setPassword( pwd ));
      return ds;
   }
}
```

Convenient alternative to explicitly using Environment

BUT: How are these ${...} variables resolved? Next slide ...

spring Pivotal

Evaluating `${..}` Variables

- ${...} variables are evaluated by a dedicated Spring bean
  - The **`PropertySourcesPlaceholderConfigurer`**
  - **Note:** make this a ***static*** bean
    - Ensures **${..}** placeholder expressions are evaluated *before* any beans are created that might use them

```
@Bean
public static PropertySourcesPlaceholderConfigurer pspc() {
      return new PropertySourcesPlaceholderConfigurer();
}
```

${..} placeholders are *not resolved* <u>unless</u> this bean declared

spring Pivotal

${...} Placeholders

- ${...} placeholders in a `@PropertySource` are resolved against existing properties
  - Such as System properties & Environment variables

```
@PropertySource ( "classpath:/com/acme/config/app-${ENV}.properties" )
```

app-dev.properties
```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://localhost/transfer
db.user=transfer-app
db.password=secret45
```

app-qa.properties
```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://qa/transfer
db.user=transfer-app
db.password=secret88
```

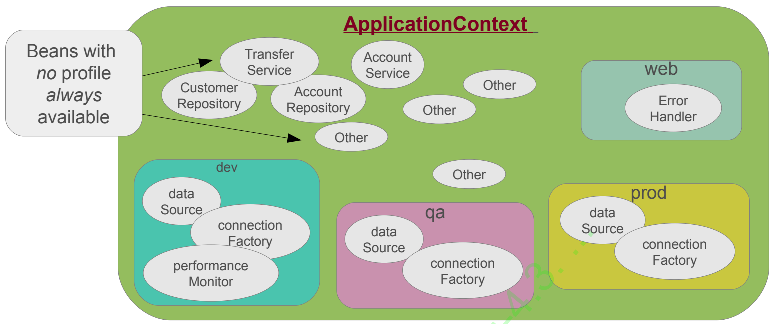
app-prod.properties
```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://prod/transfer
db.user=transfer-app
db.password=secret99
```


Topics in this session

- External Properties
- **Profiles**
- Spring Expression Language
- Proxying

Profiles

- Beans can be grouped into Profiles
  - Profiles can represent purpose: “web”, “offline”
  - Or environment: “dev”, “qa”, “uat”, “prod”
  - Beans included / excluded based on profile membership



Defining Profiles – 1

- Using @Profile annotation on configuration class
  - All beans in Configuration belong to the profile

```
@Configuration
@Profile("dev")
public class DevConfig {

  @Bean
  public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setName("testdb")
              .setType(EmbeddedDatabaseType.HSQL)
              .addScript("classpath:/testdb/schema.db")
              .addScript("classpath:/testdb/test-data.db").build();
  }
  …
```

Defining Profiles - 2

- Using @Profile annotation on @Bean methods

```
@Configuration
public class DataSourceConfig {

  @Bean(name="dataSource")
  @Profile("dev")
  public DataSource dataSourceForDev() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setName("testdb") ...
  }

  @Bean(name="dataSource")
  @Profile("prod")
  public DataSource dataSourceForProd() {
    DataSource dataSource = new BasicDataSource();
    ...
    return dataSource;
  }
```

Explicit bean-name overrides method name

Both profiles define *same* bean id, so only *one* profile should be activated at a time.

spring Pivotal

Ways to Activate Profiles

- Profiles must be activated at run-time
  - System property via command-line

```
-Dspring.profiles.active=dev,jpa
```

  - System property programmatically

```
System.setProperty("spring.profiles.active", "dev,jpa");
SpringApplication.run(AppConfig.class);
```

  - Integration Test: Use `@ActiveProfiles` (later section)
    - **Note:** `@ActiveProfiles` *only* works in a Spring-driven test

spring Pivotal

Quiz:
Which of the Following is/are Selected?

```
-Dspring.profiles.active=jpa
```

?

```
@Configuration
public class
Config { ...}
```

?

```
@Configuration
@Profile("jpa")
public class
JpaConfig
{ ...}
```

?

```
@Configuration
@Profile("jdbc")
public class
JdbcConfig
{ ...}
```

Property Source selection

- @Profile can control which @PropertySources are included in the Environment

```
@Configuration
@Profile("dev")
@PropertySource ( "dev.properties" )
class DevConfig { … }
```

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://localhost/transfer
db.user=transfer-app
db.password=secret45
```

dev.properties

```
@Configuration
@Profile("prod")
@PropertySource ( "prod.properties" )
class ProdConfig { … }
```

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://prod/transfer
db.user=transfer-app
db.password=secret99
```

prod.properties

Topics in this session

- External Properties
- Profiles
- **Spring Expression Language**
- Proxying

spring Pivotal

Spring Expression Language

- SpEL for short
  - Inspired by the Expression Language used in Spring WebFlow
  - Based on Unified Expression Language used by JSP and JSF
- Pluggable/extendable by other Spring-based frameworks

This is just a brief introduction, for full details see
http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html

spring Pivotal

SpEL examples – Using @Value

```
@Configuration
class TaxConfig
{
    @Value("#{ systemProperties['user.region'] }") String region;

    @Bean public TaxCalculator taxCalculator1() {
        return new TaxCalculator( region );
    }

    @Bean public TaxCalculator taxCalculator2
            (@Value("#{ systemProperties['user.region'] }") String region, …) {
        return  new TaxCalculator( region );
    }
    …
```

Option 1: Set an attribute then use it

Option 2: Pass as a bean method argument

spring Pivotal

SpEL – Accessing Spring Beans

```
class StrategyBean {
    private KeyGenerator gen = new KeyGenerator.getInstance("Blowfish");
    public KeyGenerator getKeyGenerator()  { return gen; }
}
```

```
@Configuration
class StrategyConfig
{
    @Bean public StrategyBean strategyBean() {
        return new StrategyBean();
    }
}
```

```
@Configuration
class AnotherConfig
{
    @Value("#{strategyBean.keyGenerator}") KeyGenerator kgen;
    …
}
```

spring Pivotal

Accessing Properties

- Can access properties via the *environment*
  - These are equivalent

```
@Value("${daily.limit}")
int maxTransfersPerDay;
```

```
@Value("#{environment['daily.limit']}")
int maxTransfersPerDay;
```

- Properties are Strings
  - May need to cast in expressions

```
@Value("#{new Integer(environment['daily.limit']) * 2}")
@Value("#{new java.net.URL(environment['home.page']).host}")
```

spring Pivotal

SpEL

- EL Attributes can be:
  - Spring beans (like *strategyBean*)
  - Implicit references
    - Spring's *environment, systemProperties, systemEnvironment* available by default
    - Others depending on context
- SpEL allows to create custom functions and references
  - Widely used in Spring projects
    - Spring Security, Spring WebFlow
    - Spring Batch, Spring Integration
  - Each may add *their own* implicit references

spring Pivotal

Topics in this session

- External Properties
- Profiles
- Spring Expression Language
- **Proxying**

Quiz

Which is the best implementation?

```
@Bean
public AccountRepository accountRepository() {
    return new JdbcAccountRepository();
}

@Bean
public TransferService transferService1() {
    TransferServiceImpl service = new TransferServiceImpl();
    service.setAccountRepository(accountRepository());
    return service;
}

@Bean
public TransferService transferService2() {
    return new TransferServiceImpl( new JdbcAccountRepository() );
}
```

1. Method call?

2. New instance?

Prefer call to *dedicated* method. Let's discuss why ...

Working with Singletons

```
@Bean
public AccountRepository accountRepository() {
    return new JdbcAccountRepository();
}

@Bean
public TransferService transferService() {
    TransferServiceImpl service = new TransferServiceImpl();
     service.setAccountRepository(accountRepository());
     return service;
}

@Bean
public AccountService accountService() {
    return new AccountServiceImpl( accountRepository() );
}
```

Singleton??

Method called *twice* more

HOW IS IT POSSIBLE?

Singletons Require Proxies

- At startup time, a proxy subclass is created
  - Subclass performs *scope-control*
    - Only calls *super* on *first* invocation of singleton bean method
    - Singleton instance is cached by the *ApplicationContext*

```
@Configuration
public class AppConfig {
   @Bean public AccountRepository accountRepository() { ... }
   @Bean public TransferService transferService() { ... }
}
```

inherits from

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {
   public AccountRepository accountRepository() {   // ... }
   public TransferService transferService() {   // ... }
}
```

Inheritance-based Proxies

- Child class is the entry point

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {

  public AccountRepository accountRepository() {
    // if bean is in the applicationContext, then return bean
    // else call super.accountRepository(), store bean in context, return bean
  }

  public TransferService transferService() {
    // if bean is in the applicationContext, then return bean
    // else call super.transferService(), store bean in context, return bean
  }
```

Java Configuration uses *cglib* for inheritance-based proxies

Summary

- Property values are easily externalized using Spring's Environment abstraction
- Profiles are used to group sets of beans
- Spring Expression Language
- Spring proxies your @Configuration classes to allow for scope control.

Annotations in Spring

Annotations for Dependency Injection and Interception

Component scanning and auto-injection

spring Pivotal

Topics in this Session

- Fundamentals
  - **Annotation-based Configuration**
  - Best practices: when to use what?
  - @PostConstruct and @PreDestroy
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - @Resource
  - Standard annotations (JSR 330

spring Pivotal

Before – Explicit Bean Definition

- Configuration is external to bean-class
  - *Separation of concerns*
  - Java-based dependency injection

```
@Configuration
public class TransferModuleConfig {

    @Bean public TransferService transferService() {
        return new TransferServiceImpl( accountRepository() );
    }

    @Bean public AccountRepository accountRepository() {
        …
    }
}
```

Dependency Injection

After - Implicit Configuration

- Annotation-based configuration *within* bean-class

Bean id derived from classname: *transferServiceImpl*

```
@Component
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository repo) {
        this.accountRepository = repo;
    }
}
```

Annotations embedded with POJOs

```
@Configuration
@ComponentScan ( “com.bank” )
public class AnnotationConfig {
   // No bean definition needed any more
}
```

Find @Component classes within designated (sub)packages

Usage of @Autowired

Unique dependency of correct **type** *must* exist

- Constructor-injection

```
@Autowired
public TransferServiceImpl(AccountRepository a) {
    this.accountRepository = a;
}
```

- Method-injection

```
@Autowired
public void setAccountRepository(AccountRepository a) {
    this.accountRepository = a;
}
```

- Field-injection

Even when field is private!! – *but* hard to unit test, see URL

```
@Autowired
private AccountRepository accountRepository;
```

spring http://olivergierke.de/2013/11/why-field-injection-is-evil/ Pivotal

@Autowired dependencies: required or not?

- Default behavior: required

Exception if no dependency found

```
@Autowired
public void setAccountRepository(AccountRepository a) {
    this.accountRepository = a;
}
```

- Use required attribute to override default behavior

```
@Autowired(required=false)
public void setAccountRepository(AccountRepository a) {
    this.accountRepository = a;
}
```

Only inject *if* dependency exists

spring Pivotal

Java 8 Optional<T>

- Another way to inject optional dependencies
  - `Optional<T>` introduced to reduce null pointer errors

```java
@Autowired(required=false)
AccountService accountService;

public void doSomething() {
  if (accountService != null) {
     // do something
  }
}
```

```java
@Autowired
Optional<AccountService> accountService;

public void doSomething() {
  accountService.ifPresent( s -> {
    // s is the AccountService instance,
    // use s to do something
  });
}
```

Note the use of the lamda

Constructor vs Setter Dependency Injection

- Spring doesn't care – can use either
  - But which is best?

Constructors	Setters
Mandatory dependencies	Optional / changeable dependencies
Immutable dependencies	Circular dependencies
Concise (pass several params at once)	Inherited automatically
	If constructor needs too many params

- Follow the same rules as standard Java
  - Be consistent across your project team
  - Many classes use both

Autowiring and Disambiguation – 1

- What happens here?

```
@Component
public class TransferServiceImpl implements TransferService {
  @Autowired
  public TransferServiceImpl(AccountRepository accountRepository)  { … }
}
```

```
@Component
public class JpaAccountRepository implements AccountRepository {..}
```

Which one should get injected?

```
@Component
public class JdbcAccountRepository implements AccountRepository {..}
```

At startup: *NoSuchBeanDefinitionException*, no unique bean of type [AccountRepository] is defined: expected single bean but found 2...

spring Pivotal

Autowiring and Disambiguation – 2

- Use of the @Qualifier annotation

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
  @Autowired
  public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
     AccountRepository accountRepository) { … }
```

qualifier

bean ID

```
@Component("jdbcAccountRepository")
public class JdbcAccountRepository implements AccountRepository {..}
```

```
@Component("jpaAccountRepository")
public class JpaAccountRepository implements AccountRepository {..}
```

@Qualifier also available with method injection and field injection

Component names should *not* show implementation details *unless* there are 2 implementations of the *same* interface (as here)

spring Pivotal

Autowiring and Disambiguation – 3

- Autowired resolution rules
  - Look for unique bean of required *type*
  - Use @Qualifier if supplied
  - Try to find a matching bean by *name*
- Example
  - We have multiple *Queue* beans
  - Spring finds bean with id matching what is being set: “**ack**”

```
@Autowired
public myBean(Queue ack) {
    …
}
```

```
@Autowired
public void setQueue(Queue ack) {
    …
}
```

```
@Autowired
private Queue ack;
```

Looks for Queue bean with id = “**ack**”

spring Pivotal

Using @Value to set Attributes

Can use $ variables *or* SpEL

- Constructor-injection

```
@Autowired
public TransferServiceImpl(@Value("${daily.limit}") int max) {
   this.maxTransfersPerDay = max;
}
```

- Method-injection

```
@Autowired
public void setDailyLimit(@Value("${daily.limit}") int max) {
   this.maxTransfersPerDay = max;
}
```

- Field-injection

```
@Value("#{environment['daily.limit']}")
int maxTransfersPerDay;
```

Not private so we can initialize in a unit-test

spring Pivotal

More on @Value

- Providing a fall-back value
  - If `daily.limit` undefined, use colon **:**

```
@Autowired
public TransferServiceImpl(@Value("${daily.limit : 100000}") int max) {
    this.maxTransfersPerDay = max;
}
```

Equivalent operators

  - For SpEL, use the “Elvis” operator **?:**

```
@Autowired
public setLimit(@Value("#{environment['daily.limit'] ?: 100000}") int max) {
    this.maxTransfersPerDay = max;
}
```

`x ?: y` is short for `x != null ? x : y`

Elvis lives!

Component Names

- When not specified
  - Names are auto-generated
    - De-capitalized non-qualified classname by default
    - *But* will pick up implementation details from classname
  - *Recommendation:* never rely on generated names!
- When specified
  - Allow disambiguation when 2 bean classes implement the same interface

> Common strategy: avoid using qualifiers when possible.
> *Usually rare to have 2 beans of same type in ApplicationContext*

Java Config vs Annotations syntax

- Similar options are available

```java
@Component("transferService")
@Scope("prototype")
@Profile("dev")
@Lazy(false)
public class TransferServiceImpl
      implements TransferService {
  @Autowired
  public TransferServiceImpl
        (AccountRepository accRep) { … }

}
```

Annotations

```java
@Configuration
public class TransferConfiguration
   @Bean(name="transferService")
   @Scope("prototype")
   @Profile("dev")
   @Lazy(false)
   public TransferService tsvc() {
      return
         new TransferServiceImpl(
                  accountRepository());
   }
}
```

Java Configuration

Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - **Best practices: when to use what?**
  - @PostConstruct and @PreDestroy
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - @Resource
  - Standard annotations (JSR 330)

About Component Scanning

- Components are scanned at startup
  - JAR dependencies also scanned!
  - Could result in slower startup time if too many files scanned
    - Especially for large applications
    - A few seconds slower in the worst case

- What are the best practices?

Best practices

- Really bad:

```
@ComponentScan ( { "org", "com" } )
```

All “org” and “com” packages in the classpath will be scanned!!

- Still bad:

```
@ComponentScan ( "com" )
```

- OK:

```
@ComponentScan ( "com.bank.app" )
```

- Optimized:

```
@ComponentScan ( { "com.bank.app.repository",
   "com.bank.app.service", "com.bank.app.controller" } )
```

When to use what?

Java

Java Configuration

- Pros:
  - Is centralized in one (or a few) places
  - Write any Java code you need
  - Strong type checking enforced by compiler (and IDE)
  - Can be used for all classes (not just your own)
- Cons:
  - More verbose than annotations

spring Pivotal

When to use what?

Annotations

- Nice for frequently changing beans
- Pros:
  - Single place to edit (just the class)
  - Allows for very rapid development
- Cons:
  - Configuration spread across your code base
    - Harder to debug/maintain
  - Only works for your own code
  - Merges configuration and code (bad sep. of concerns)

spring Pivotal

Mixing Java Config and Annotations

- You can mix and match in many ways
- Common approach:
  - Use annotations whenever possible
    - Your classes
  - But still use Java Configuration for
    - Third-party beans that aren't annotated
    - Legacy code that can't be changed

Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - **@PostConstruct and @PreDestroy**
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - @Resource
  - Standard annotations (JSR 330)

@PostConstruct and @PreDestroy

- Add behavior at startup and shutdown

```
public class JdbcAccountRepository {
   @PostConstruct
   void populateCache() { }

  @PreDestroy
   void clearCache() { }
}
```

Method called at startup after dependency all injection

Method called at shutdown prior to destroying the bean instance

Annotated methods can have any visibility but *must* take *no* parameters and *only* return *void*

spring Pivotal

About @PostConstuct & @PreDestroy

- Beans are created in the usual ways:
  - Returned from @Bean methods
  - Found and created by the component-scanner
- Spring then invokes these methods *automatically*
  - During bean-creation process

- These are not Spring annotations
  - Defined by JSR-250, part of Java since Java 6
  - In **`javax.annotation`** package
  - Supported by Spring, *and* by JEE

spring Pivotal

@PostConstruct

- Called after setter methods are called

```
public class JdbcAccountRepository {
    private DataSource dataSource;
    @Autowired
    public void setDataSource(DataSource dataSource)       (1)
       {  this.dataSource = dataSource;   }

    @PostConstruct
    public void populateCache()                              (2)
       {   Connection conn = dataSource.getConnection(); //...   }
}
```

@PreDestroy

- Called when a *ConfigurableApplicationContext* is closed
  - *If* application (JVM) exits *normally*
  - Useful for releasing resources & 'cleaning up'
  - ***Not*** called for prototype beans

```
ConfigurableApplicationContext context = SpringApplication.run(...);
// Triggers call of all @PreDestroy annotated methods
context.close();
```

Causes Spring to invoke this method

```
public class JdbcAccountRepository {
    @PreDestroy
    public void clearCache() { … }
    ...
}
```


Lifecycle Methods via @Bean

- Alternatively, **`@Bean`** has options to define these *life-cycle* methods

```
@Bean (initMethod="populateCache", destroyMethod="clearCache")
public AccountRepository accountRepository() {
    // ...
}
```

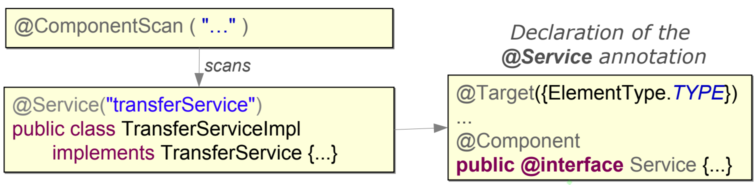
- Common Usage:
  - **`@PostConstruct`**/**`@PreDestroy`** for your own classes
  - **`@Bean`** properties for classes you didn't write and can't annotate

Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - @PostConstruct and @PreDestroy
  - **Stereotypes and meta annotations**
- Lab
- Advanced features
  - @Resource
  - Standard annotations (JSR 330)

Stereotype Annotations

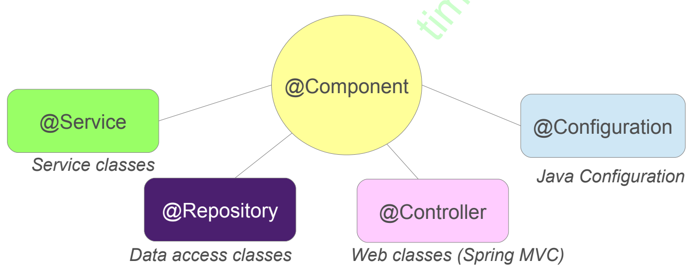
- Component scanning also checks for annotations that are themselves annotated with @Component
  - So-called *stereotype annotations*



@Service annotation is part of the Spring framework

Predefined Stereotype Annotations

- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations (Spring Web-Services, Spring Integration...)

Meta-annotations

- Annotation which can be used to annotate other annotations
  - e.g. all service beans should be configurable using component scanning and be transactional

```
@ComponentScan ( "...." )
```

scans

recognizes

```
@MyTransactionalService
public class TransferServiceImpl
  implements TransferService {...}
```

Custom annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Service
@Transactional(timeout=60)
public @interface
MyTransactionalService {
   String value() default "";
}
```


Summary

- Spring beans can be defined:
  - Explicitly using @Bean methods
  - Implicitly using @Component and component-scanning
- Most applications use both
  - Implicit for your classes
  - Explicit for the rest
- Can perform initialization and clean-up
  - Use @PostConstruct and @PreDestroy
- Use Spring's stereotypes and/or define your own meta annotations

Lab

Using Spring Annotations
To Configure and Test an application

Coming Up: Other Annotations for Dependency Injection

spring Pivotal

Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - @PostConstruct and @PreDestroy
  - Stereotypes and meta annotations
- Lab
- **Advanced features**
  - **@Resource**
  - Standard annotations (JSR 330)

spring Pivotal

Using @Resource

- From JSR-250, supported by EJB 3.0 and Spring
  - Identifies dependencies by ***name***, not by ***type***
    - *Name is Spring bean-name*
    - @Autowired matches by ***type***
  - Supports setter and field injection *only*

```
@Resource(name="jdbcAccountRepository")
public void setAccountRepository(AccountRepository repo) {
  this.accountRepository = repo;
}
```

Setter Injection

```
@Resource(name="jdbcAccountRepository")
private AccountRepository accountRepository;
```

Field injection

spring Pivotal

Qualifying @Resource

@Autowired: type *then* name
@Resource: name *then* type

- When no name is supplied
  - Inferred from property/field name
  - Or falls back on injection by type
- Example
  - Looks for bean called *accountRepository*
    - because method is ***set****AccountRepository*
  - Then looks for bean of type *AccountRepository*

```
@Resource
public void setAccountRepository(AccountRepository repo) {
  this.accountRepository = repo;
}
```

spring Pivotal

Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - @PostConstruct and @PreDestroy
  - Stereotypes and meta annotations
- Lab
- **Advanced features**
  - @Resource
  - **Standard annotations (JSR 330)**

spring Pivotal

JSR 330

- Java Specification Request 330
  - Also known as @Inject
  - Joint JCP effort by Google and SpringSource
  - Standardizes internal DI annotations
  - Published late 2009
    - Spring is a valid JSR-330 implementation
- Subset of functionality compared to Spring's @Autowired support
  - @Inject has 80% of what you need
  - Rely on @Autowired for the rest

spring Pivotal

JSR 330 annotations

Also scans JSR-330 annotations

```
@ComponentScan ( "...." )
```

Should be specified for component scanning (even without a name)

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class TransferServiceImpl implements TransferService {
   @Inject
   public TransferServiceImpl( @Named("accountRepository")
        AccountRepository accountRepository)    { … }
}
```

```
import javax.inject.Named;

@Named("accountRepository")
public class JdbcAccountRepository implements
                          AccountRepository {..}
```

spring Pivotal

From @Autowired to @Inject

Spring	JSR 330	Comments
@Autowired	@Inject	@Inject always mandatory, has no required option
@Component	@Named	Spring also scans for @Named
@Scope	@Scope	JSR 330 Scope for meta-annotation and injection points only
@Scope (“singleton”)	@Singleton	JSR 330 default scope is like Spring's '*prototype*'
@Qualifer	@Named	
@Value	No equivalent	SpEL specific
@Required	Redundant	@Inject *always* required
@Lazy	No equivalent	Useful when needed, often abused

spring Pivotal

Dependency Injection Using XML

Spring's XML Configuration Language

Using <bean> definitions and namespaces

spring Pivotal

Topics in this session

Note: XML *is* in the certification exam

- **Writing bean definitions in XML**
- Creating an application context
- Controlling Bean Behavior
- Factory Beans
- Namespaces
- Lab
- Advanced Topics

spring Pivotal

XML Configuration

- Original form of Configuration / Dependency Injection
  - Dating back to before 2004
  - Still fully supported
- Most commonly seen in existing applications
  - … and in older blogs, books, etc.
- External *explicit* configuration as with Java Config
  - Uses custom XML instead of Java

@Configuration Comparison

Java configuration class

```java
@Configuration
@Profile("prod")
public class AppConfig {

  @Bean
  public TransferService transferService()
  {
    TransferService service
           = new TransferServiceImpl();
    service.setRepository(repository());
    return service;
  }

  @Bean(name="accountRepository")
  public AccountRepository repository()
    { //... }
}
```

XML configuration file

```xml
<beans profile="prod">

  <bean id="transferService"
    class="com.acme.TransferServiceImpl">
    <property name="repository"
              ref="accountRepository" />
  </bean>

<bean id="accountRepository"
  class="com.acme.JdbcAccountRepository">
    ...
  </bean>

</beans>
```

profile

bean id

Dependency injection

bean id

Constructor Injection Configuration

- One parameter

```
<bean id="transferService" class="com.acme.TransferServiceImpl">
   <constructor-arg ref="accountRepository"/>
</bean>

<bean id="accountRepository" class="com.acme.AccountRepositoryImpl"/>
```

- Multiple parameters

```
<bean id="transferService" class="com.acme.TransferServiceImpl">
   <constructor-arg ref="accountRepository"/>
   <constructor-arg ref="customerRepository"/>
</bean>

<bean id="accountRepository" class="com.acme.AccountRepositoryImpl"/>
<bean id="customerRepository" class="com.acme.CustomerRepositoryImpl"/>
```

Parameters injected according to their type

spring Pivotal

Constructor Injection 'Under the Hood'

```
<bean id="service" class="com.acme.ServiceImpl">
   <constructor-arg ref="repository"/>
</bean>

<bean id="repository" class="com.acme.RepositoryImpl"/>
```

Equivalent to:

```
@Bean public Repository repository() {
   return new RepositoryImpl();
}

@Bean public Service service() {
   return new ServiceImpl( repository() );
}
```

spring Pivotal

Setter Injection

```
<bean id="service" class="com.acme.ServiceImpl">
    <property name="repository" ref="repository"/>
</bean>

<bean id="repository" class="com.acme.RepositoryImpl"/>
```

Convention: implicitly refers to method *setRepository(...)*

Equivalent to:

```
@Bean public Repository repository() {
    return new RepositoryImpl();
}

@Bean public Service service() {
    Service svc = new ServiceImpl();
    svc.setRepository( repository() );
    return svc;
}
```

spring Pivotal

Combining Constructor and Setter Injection

```
<bean id="service" class="com.acme.ServiceImpl">
    <constructor-arg ref="required" />
    <property name="optional" ref="optional" />
</bean>

<bean id="required" class="com.acme.RequiredImpl" />
<bean id="optional" class="com.acme.OptionalImpl" />
```

Equivalent to:

```
@Bean public RequiredImpl required() { … }
@Bean public OptionalImpl optional() { … }
@Bean public Service service() {
    Service svc = new ServiceImpl( required() );
    svc.setOptional( optional() );
    return svc;
}
```

spring Pivotal

Injecting Scalar Values

```
<bean id="service" class="com.acme.ServiceImpl">
    <property name="stringProperty" value="foo" />
</bean>
```

Equivalent

```
<property name="stringProperty">
    <value>foo</value>
</property>
```

```
public class ServiceImpl {
    public void setStringProperty(String s) { ... }
    // ...
}
```

Equivalent to:

```
@Bean
public Service service() {
    Service svc = new ServiceImpl( );
    svc.setStringProperty( "foo" );
    return svc;
}
```

spring Pivotal

Automatic Value Type Conversion

```
<bean id="service" class="com.acme.ServiceImpl">
    <property name="intProperty" value="29" />
</bean>
```

```
public class ServiceImpl {
    public void setIntProperty(int i) { ... }
    // ...
}
```

Equivalent to:

```
@Bean public Service service() {
    Service svc = new ServiceImpl( );
    int val = // Integer parsing logic, 29.
    svc.setIntProperty( val );
    return svc;
}
```

Spring can convert:
- Numeric types
- BigDecimal,
- boolean: "true", "false"
- Date
- Locale
- Resource

spring Pivotal

Topics in this session

- Writing bean definitions in XML
- **Creating an application context**
- Controlling Bean Behavior
- Factory Beans
- Namespaces
- Lab
- Advanced Topics

spring Pivotal

Creating an ApplicationContext using XML

- Use a Java Configuration class
  - @ImportResource to define XML file(s):

```
SpringApplication.run(MainConfig.class);
```

```
@Configuration
@ImportResource( {
   "classpath:com/acme/application-config.xml",
   "file:C:/Users/alex/application-config.xml" } )
@Import(DatabaseConfig.class)
public class MainConfig {  … }
```

- Multiple files possible.
- Valid prefixes are classpath: (default), file:, http:

Can combine with @Configuration imports

spring Pivotal

Remember @Import?

```
@Configuration
@Import(DatabaseConfig.class)
public class MainConfig {
    ...
}
```

- Use <import /> to import other XML configuration files

```
<beans>
  <import resource="db-config.xml" />
</beans>
```

- Uses relative path by default
  - Same prefixes available (file, classpath, http)

spring Pivotal

Creating the ApplicationContext – I

- So far, you have seen the ApplicationContext created like this:

```
ApplicationContext context = SpringApplication.run(MainConfig.class);
```

- This is actually a Spring Boot class
  - But it works well for *any* Spring application
  - More general purpose than previous alternatives

spring Pivotal

Creating the ApplicationContext

- Older "classic" techniques available as well
  - Context type defines *where* XML files are loaded from
  - Existing code (and many online examples) do it this way

```
// Load Java Configuration class
new AnnotationConfigApplicationContext(MainConfig.class);

// Load from $CLASSPATH/com/acme/application-config.xml
new ClassPathXmlApplicationContext("com/acme/application-config.xml");

// Load from absolute path: C:/Users/alex/application-config.xml
new FileSystemXmlApplicationContext("C:/Users/alex/application-config.xml");

// Load from path relative to the JVM working directory
new FileSystemXmlApplicationContext("./application-config.xml");
```

spring Pivotal

Topics in this session

- Writing bean definitions
- Creating an application context
- **Controlling Bean Behavior**
- Factory Beans
- Namespaces
- Lab
- Advanced Topics

spring Pivotal

Remember @PostConstruct?

```
@PostConstruct
public void setup() {
    ...
}
```

- Same option available in XML
  - But called “init-method”:

```
<bean id=“accountService” class=“com.acme.ServiceImpl” init-method=“setup”>
   ...
</bean>
```

Same rules: method can have any visibility, *must* take *no* parameters, must return *void*. Called after dependency injection.

spring Pivotal

Remember @PreDestroy?

```
@PreDestroy
public void teardown() {
    ...
}
```

- Same option available in XML
  - But called “destroy-method”:

```
<bean id=“Service” class=“com.acme.ServiceImpl” destroy-method=“teardown”>
   ...
</bean>
```

Same rules: method can have any visibility, *must* take *no* parameters, must return *void.*

spring Pivotal

Remember Bean Scope?

```
@Bean
@Scope("prototype")
public AccountService accountService() {
    return ...
}
```

```
@Component
@Scope("prototype")
public class AccountServiceImpl {
  ...
}
```

- Same options available in XML
  - singleton, prototype, request, session, (custom)

```
<bean id="accountService" class="com.acme.ServiceImpl" scope="prototype">
  ...
</bean>
```

spring Pivotal

Remember @Lazy?

```
@Bean
@Lazy("true")
public AccountService accountService() {
    return ...
}
```

```
@Component
@Lazy("true")
public class AccountServiceImpl {
  ...
}
```

- Same option available in XML
  - Still not recommended, often misused

```
<bean id="accountService" class="com.acme.ServiceImpl" lazy-init="true">
  ...
</bean>
```

spring Pivotal

Profile Configuration in XML

- All bean definitions

```xml
<beans xmlns=http://www.springframework.org/schema/beans …
       profile="dev">  … </beans>
```

Profile applies to *all* Beans in the file

- Subset of bean definitions

```xml
<beans xmlns=http://www.springframework.org/schema/beans ...>
       <bean id="rewardNetwork" … />  <!-- Available to all profiles -->
       ...
       <beans profile="dev">  ... </beans>
       <beans profile="prod"> ... </beans>
</beans>
```

Different subset of beans for each profile, plus some shared beans

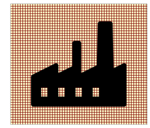
spring Pivotal

Topics in this session

- Writing bean definitions in XML
- Creating an application context
- Controlling Bean Behavior
- **Factory Beans**
- Namespaces
- Lab

spring Pivotal

Advanced XML Bean Instantiation



- Conditional configuration
  - @Bean methods can use *any* Java you need
    - Do property lookups
    - Use if-then-else and iterative logic
- No equivalent in XML
  - We did *not* implement <if>, <for-each>
- Instead Spring XML relies on the *Factory Pattern*
  - Use a factory to create the bean(s) we want
  - Use *any* complex Java code we need in the factory's internal logic

spring Pivotal

The Spring FactoryBean interface

- Fall-back for complex configuration in XML
  - Used long before @Bean methods introduced

Note: even Java Configuration may use factory beans

```
public class AccountServiceFactoryBean
      implements FactoryBean <AccountService>
{
    public AccountService getObject() throws Exception {
        // Conditional logic – for example: selecting the right
        //  implementation or sub-class of AccountService to create
        return accountService;
    }

    public boolean  isSingleton() { return true; }
    public Class<?> getObjectType() { return AccountService.class; }
}
```

```
<bean id="accountService" class="com.acme.AccountServiceFactoryBean" />
```

spring Pivotal

The FactoryBean interface

- Beans implementing *FactoryBean* are *auto-detected*
- Dependency injection using the factory bean id causes *getObject()* to be invoked transparently

```
<bean id="accountService"
        class="com.acme.AccountServiceFactoryBean"/>

<bean id="customerService" class="com.acme.CustomerServiceImpl">
    <property name="service" ref="accountService" />
</bean>
```

getObject() called by Spring *internally*

Using FactoryBeans with Java Configuration

- Works exactly the same way

getObject() called by Spring *internally*

```
@Configuration
public class ServiceConfig {

    @Bean
    public AccountServiceFactoryBean accountService() {
        return new AccountServiceFactoryBean();
    }

    @Bean
    public CustomerService customerService(AccountService accountService) {
        return new CustomerService(accountService);
    }
```

FactoryBeans in Spring

- FactoryBeans are widely used within Spring
  - EmbeddedDatabaseFactoryBean**
  - JndiObjectFactoryBean
    - One option for looking up JNDI objects
  - Creating Remoting proxies
  - Creating Caching proxies**
  - For configuring data access technologies**
    - JPA, Hibernate or MyBatis
- In XML, often hidden behind *namespaces*

** These will appear later in the course

spring Pivotal

Topics in this session

- Writing bean definitions in XML
- Creating an application context
- Controlling Bean Behavior
- Factory Beans
- **Namespaces**
- Lab

spring Pivotal

Default Namespace

- The default namespace in a Spring configuration file is typically the “beans” namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd>
    <!-- ... -->
</beans>
```

dozens of other namespaces are available!

Other Namespaces

- Defined for subsets of framework functionality*
  - aop (Aspect Oriented Programming)
  - tx (transactions)
  - util
  - jms
  - context
  - …
- They allow hiding of actual bean definitions
  - Define “programming instructions” for bean files
  - Greatly reduce size of bean files (see next slides)

*See http://www.springframework.org/schema/ for complete list

Namespace Example 1
In-Memory DataStore using Bean XML

- Creating an in-memory test database

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.
                                   embedded.EmbeddedDatabaseFactoryBean">
   <property name="databasePopulator" ref="populator"/>
</bean>

<bean id="populator" class="org.springframework.jdbc.datasource.
                                          init.ResourceDatabasePopulator">
   <property name="scripts">
      <list>
         <value>classpath:rewards/testdb/schema.sql</value>
         <value>classpath:rewards/testdb/data.sql</value>
      </list>
   </property>
</bean>
```

FactoryBean

Populate with test-data

Bean XML requires two beans and knowledge of the classes being used

spring Pivotal

Namespace Example 1
In-Memory DataStore using jdbc Namespace

- Simplify using jdbc namespace

```
<jdbc:embedded-database id="dataSource" type="HSQL">
      <jdbc:script location="classpath:rewards/testdb/schema.db"/>
      <jdbc:script location=""classpath:rewards/testdb/test-data.db"/>
</jdbc:embedded-database>
```

Equivalent to ...

```
@Bean public DataSource dataSource() {
   EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
   return builder.setName("testdb")
              .setType(EmbeddedDatabaseType.HSQL)
              .addScript("classpath:rewards/testdb/schema.db")
              .addScript("classpath:rewards/testdb/test-data.db").build();
}
```

spring Pivotal

Namespace Example 2
Property Placeholders

- Property Placeholders define property sources
  - XML Equivalent of @PropertySource
- Namespace just an elegant way to hide the underlying bean declaration
  - Same functionality, less typing

```
<context:property-placeholder location="db-config.properties" />
```

```
<bean class="org.springframework...PropertySourcesPlaceholderConfigurer">
    <property name="location" value="db-config.properties"/>
</bean>
```

spring Pivotal

Accessing Properties in XML

```
<beans  ...>
   <context:property-placeholder location="db-config.properties" />

   <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource">
      <property name="URL" value="${dbUrl}" />
      <property name="user" value="${dbUserName}" />
   </bean>
</beans>
```

```
dbUrl=jdbc:oracle:...
dbUserName=moneytransfer-app
```

db-config.properties

```
<bean id="dataSource"
         class="com.oracle.jdbc.pool.OracleDataSource">
   <property name="URL" value="jdbc:oracle:..." />
   <property name="user" value="moneytransfer-app" />
</bean>
```

spring Pivotal

XML Profiles and Properties

```
<import resource="classpath:config/${current.env}-config.xml"/>
<context:property-placeholder properties-ref="configProps"/>
<beans profile="dev">
  <util:properties id="configProps" location="config/app-dev.properties">
</beans>
<beans profile="prod">
  <util:properties id="configProps" location="config/app-prod.properties">
</beans>
```

```
current.env=dev
database.url=jdbc:derby:/test
database.user=tester
```

```
current.env=prod
database.url=jdbc:oracle:thin:...
database.user=admin
```

Equivalent to ...

```
@PropertySource ( "classpath:/com/acme/config/app-${ENV}.properties" )
```

spring Pivotal

Typical Profiles & Namespaces Example

```
<beans xmlns=http://www.springframework.org/schema/beans
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jdbc=http://www.springframework.org/schema/jdbc
      xmlns:jee=http://www.springframework.org/schema/jee
      xsi:schemaLocation="...">
 <beans profile="dev">
   <jdbc:embedded-database id="dataSource">
     <jdbc:script location="classpath:com/bank/sql/schema.sql"/>
     <jdbc:script location="classpath:com/bank/sql/test-data.sql"/>
   </jdbc:embedded-database>
 </beans>
 <beans profile="production">
   <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource" />
 </beans>
</beans>
```

spring Pivotal

Power of Namespaces

- Greatly simplifies Spring configuration
  - Many advanced features of Spring need to declare a large number of beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <context:property-placeholder location="db-config.properties" />

    <aop:aspectj-autoproxy />

    <tx:annotation-driven />

</beans>
```

hides **1** bean definition

AOP configuration: hides **5+** bean definitions

Transactions configuration: hides **more than 15** bean definitions!

Transactions and *AOP* will be discussed later

spring Pivotal

Remember @ComponentScan?

```
@Configuration
@ComponentScan ( { "com.acme.app.repository",
    "com.acme.app.service", "com.acme.app.controller" } )
public class MainConfig {
    ...
}
```

Array of Strings

- Available in the context namespace

Single String

```
<context:component-scan base-package="com.acme.app.repository,
    com.acme.app.service, com.acme.app.controller" />
```

spring Pivotal

Namespaces Declarations are Tedious!

- What you need for beans, context and jdbc:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jdbc="http://www.springframework.org/schema/jdbc"
        xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/jdbc
         http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd">
```

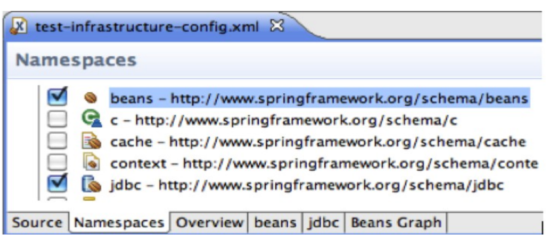
- A typo waiting to happen!
  - Fortunately there is an easier way … (next slide)

spring Pivotal

Adding namespace declaration

xsi:schemaLocation="..."

- XML syntax is error-prone
  - Use the dedicated STS XML editor Namespaces tab!



Click here and select appropriate namespaces

spring Pivotal

Schema Version Numbers

spring-beans-4.2.xsd OR spring-beans.xsd ?

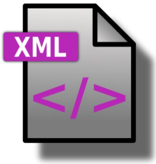
- Common practice: *do not* use a version number
  - Triggers use of most recent schema version
  - Easier migration
    - Will make it easier to upgrade to the next version of Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
   <!-- ... -->
</beans>
```

Not needed!

Summary

- Spring's XML definition language provides *explicit* bean definitions using <bean/> syntax
  - Pre-dates Java Configuration
  - Provides many of the same options via attributes
- Factory Beans
  - Important configuration device
  - Understand how *getObject()* works
- Namespaces reduce verbosity, hide internal details



Spring XML Best Practices

- XML has been around for a long time
  - Many shortcuts and useful techniques exist
    - Singleton and Factory Beans
    - Bean Definition Inheritance
    - Inner Beans
    - p and c namespaces
    - Using collections as Spring beans
- Optional Section at back of handout
  - **XML Dependency Injection Best Practices**
    - Optional lab also

Lab (optional)

Using XML to Configure an Application

spring

Understanding the Bean Lifecycle

An In-Depth Look "Under the Hood"

Using Bean Pre- and Post-Processors

spring Pivotal

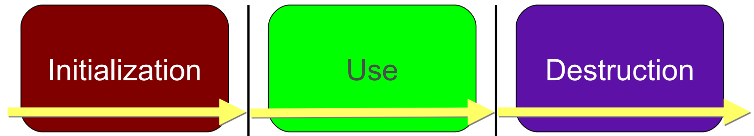
Topics in this session

- **Introduction**
- The initialization phase
- The use phase
- The destruction phase

The content of this chapter is a *much simplified* view of Spring's inner workings

spring Pivotal

Phases of the Application Lifecycle



Initialization

- Prepares for use
- Application services
  - Are created
  - Configured
  - May allocate system resources
- Application is not usable until this phase is complete

Use

- Used by clients
- Application services
  - Process client requests
  - Carry out application behaviors
- 99.99% of time is spent in this phase

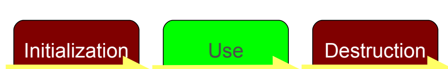
Destruction

- Shuts down
- Application services
  - Release any system resources
  - Are eligible for garbage collection

spring Pivotal

Spring's Role as a Lifecycle Manager

- Spring fits in to manage your application lifecycle
  - May play an important role in *all* phases
- Lifecycle applies to
  - *Any* class of application
    - Standalone Java or Spring Boot application
    - Integration/System Test
    - Java EE™ (web or full profile)
  - *All* 3 dependency injection styles
    - XML, annotations and Java Configuration



spring Pivotal

Topics in this session

- Introduction
- **The initialization phase**
- The use phase
- The destruction phase

Lifecycle of a Spring Application Context

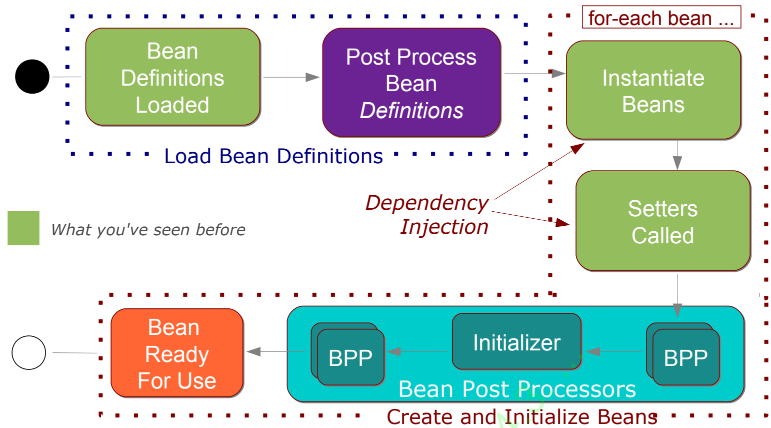
(1) The Initialization Phase

- When a context is created the initialization phase completes

```
// Create the application from the configuration
ApplicationContext context =
    SpringApplication.run(AppConfig.class);
```

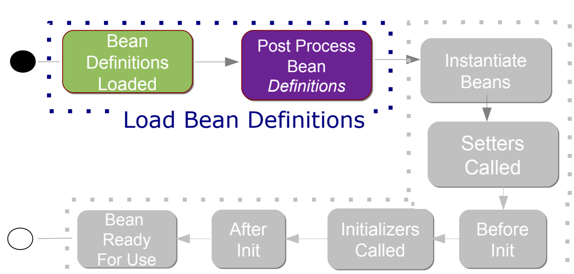
- But what exactly happens in this phase?

Bean Initialization Steps



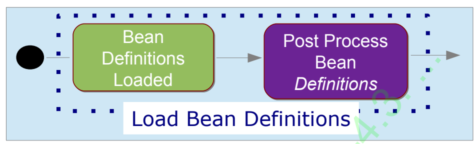
Inside The Application Context – Initialization Lifecycle (1)

- **Load bean definitions**
- Initialize bean instances

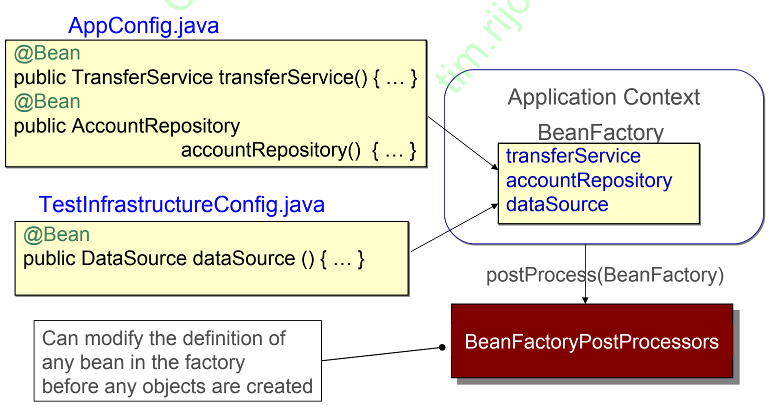


Load Bean Definitions

- The @Configuration classes are processed
  - And/or @Components are scanned for
  - And/or XML files are parsed
- Bean definitions added to a *BeanFactory*
  - Each indexed under its id
- Special *BeanFactoryPostProcessor* beans invoked
  - Can modify the *definition* of *any* bean



Load Bean Definitions



BeanFactoryPostProcessor Extension Point

- Applies transformations to bean *definitions*
  - Before objects are actually created
- Several useful implementations provided in Spring
  - Reading properties, registering a custom scope ...
- You can write your own (not common)
  - Implement BeanFactoryPostProcessor interface

```
public interface BeanFactoryPostProcessor {
   public void postProcessBeanFactory
                      (ConfigurableListableBeanFactory beanFactory);
}
```

spring Pivotal

Most Common Example of Using a **`BeanFactoryPostProcessor`**

- Recall @Value and `${..}` variables
  - Need a `PropertySourcesPlaceholderConfigurer` to evaluate them
  - *This is* a `BeanFactoryPostProcessor`

```
@Configuration
@PropertySource ( "classpath:/config/app.properties" )
public class ApplicationConfig {

  @Value("${max.retries}")
  int maxRetries;

  ...
}
```

spring Pivotal

Java Configuration uses *Static* beans

- To ensure these beans are created *without side-effects*, they must be defined as *static* methods
- Example:
  - **PropertySourcesPlaceholderConfigurer**

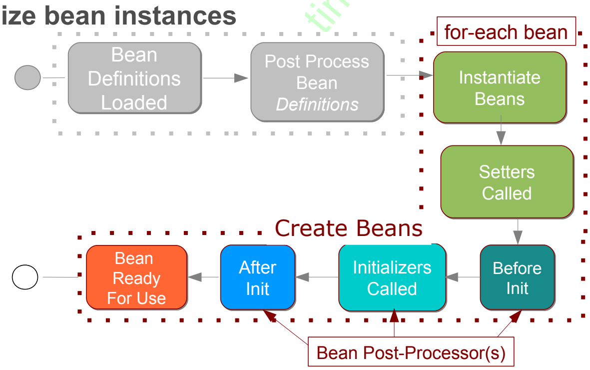
```
@Bean
public static PropertySourcesPlaceholderConfigurer
                    propertySourcesPlaceholderConfigurer() {
       return new PropertySourcesPlaceholderConfigurer();
}
```

Or if using XML, *<context:property-placeholder/>* creates a *PropertySourcesPlaceholderConfigurer* for you

spring Pivotal

Inside the Application Context – Initialization Lifecycle (2)

- Load bean definitions
- **Initialize bean instances**



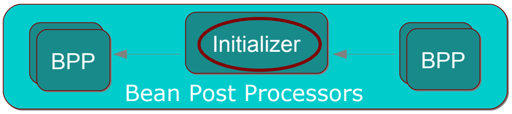
spring Pivotal

Bean Creation Summary

- Each bean is eagerly instantiated by default
  - Created in right order with its dependencies injected
    - *Unless* marked as lazy
- Next each bean goes through a *post-processing* phase
  - *BeanPostProcessors*
- Now the bean is fully initialized and ready for use
  - Tracked by its id until the context is destroyed
    - *Except* prototype beans

The Initializer Extension Point

- Special case of a bean post-processor
  - Causes init (@PostConstruct) methods to be called
- Internally Spring uses several BPPs
  - *CommonAnnotationBeanPostProcessor* enables initialization



The Initializer Extension Point - XML

- BPPs must be enabled when using *just* XML
  - Specify either `<context:annotation-config/>`
  - Or `<context:component-scan/>`
- Either enables same BPPs

```
<bean id="accountRepository" class="com.acme.JdbcAccountRepo"
        init-method="populateCache">
   …
</bean>

<context:annotation-config/>
```

init-method ignored *unless* explicitly enabled

The BeanPostProcessor Extension Point

- An important extension point in Spring
  - Can modify bean *instances* in any way
  - *Powerful* enabling feature
- Must implement the BeanPostProcessor interface
  - Spring provides several implementations
  - You can write your own (not common)

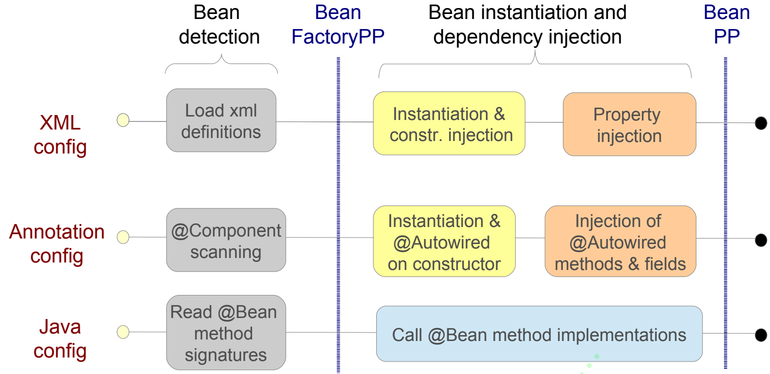
Course will show several BPPs

```
public interface BeanPostProcessor {
  public Object postProcessAfterInitialization(Object bean, String beanName);
  public Object postProcessBeforeInitialization(Object bean,String beanName);
}
```

Post-processed bean

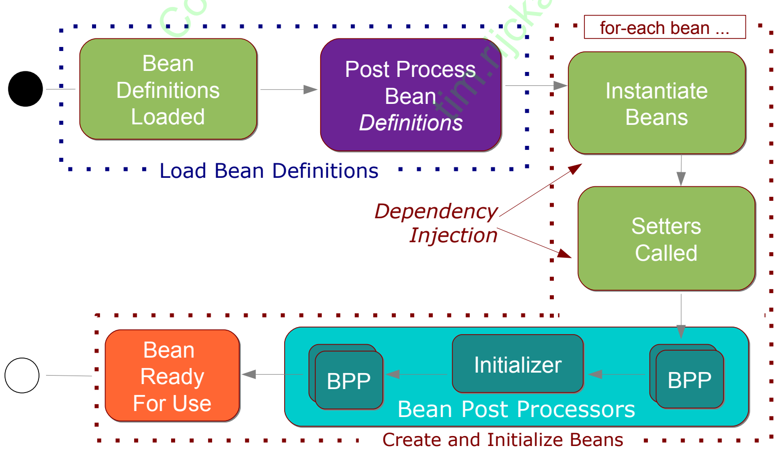
Original bean

Configuration Lifecycle



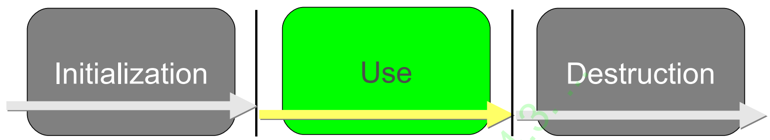
BeanFactoryPP → BeanFactoryPostProcessor
BeanPP → BeanPostProcessor

The Full Initialization Lifecycle



Topics in this session

- Introduction
- The initialization phase
- **The use phase**
- The destruction phase



Lifecycle of a Spring Application Context
(2) The Use Phase

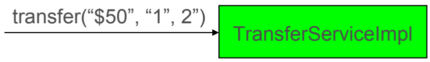
- When you invoke a bean obtained from the context the application is used

```
ApplicationContext context = // get it from somewhere
// Lookup the entry point into the application
TransferService service =
    (TransferService) context.getBean("transferService");
// Use it!
service.transfer(new MonetaryAmount("50.00"), "1", "2");
```

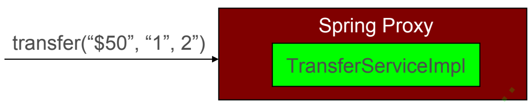
- But exactly what happens in this phase?

Inside The Bean Request (Use) Lifecycle

- The bean is just your raw object
  – it is simply invoked directly (nothing special)



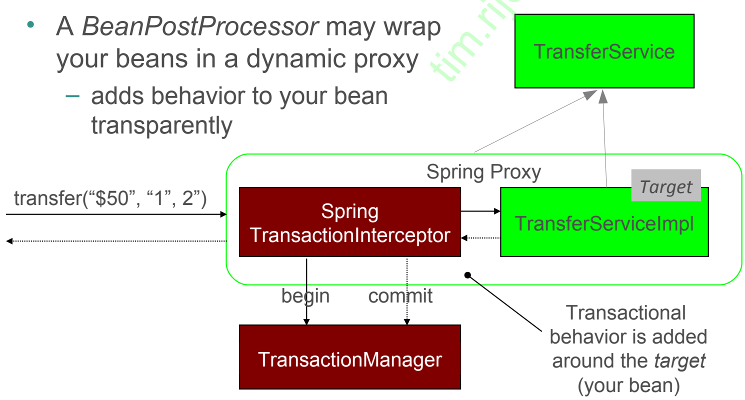
- Your bean has been wrapped in a *proxy*
  – things become more interesting



Proxy classes are created in the init phase by dedicated *BeanPostProcessors*

Proxy Power

- A *BeanPostProcessor* may wrap your beans in a dynamic proxy
  – adds behavior to your bean transparently



Kinds of Proxies

- Spring will create either JDK or CGLib proxies

JDK Proxy

- Also called *dynamic* proxies
- API is built into the JDK
- Requirements: Java interface(s)
- *All* interfaces proxied

CGLib Proxy

- NOT built into JDK
- Included in Spring jars
- Used when interface not available
- Cannot be applied to final classes or methods

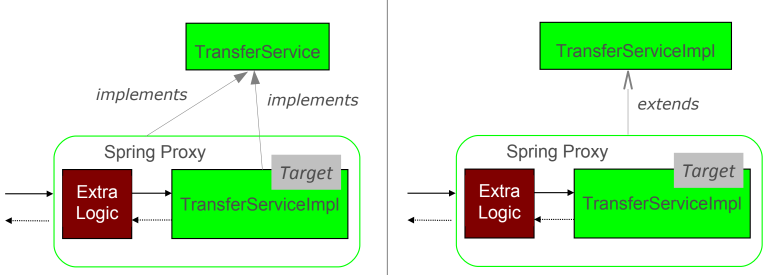
Recommendation: Code to interfaces / Use JDK proxies (default)

See Spring Reference - 10.5.3 JDK- and CGLIB-based proxies

spring Pivotal

JDK vs CGLib Proxies

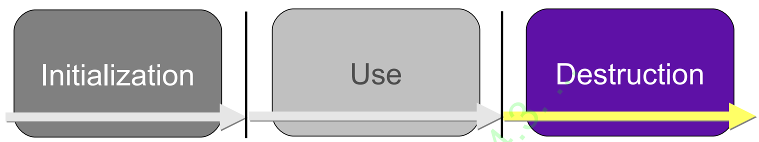
- JDK Proxy
  - Interface based
- CGLib Proxy
  - subclass based



spring Pivotal

Topics in this session

- Introduction
- The initialization phase
- The use phase
- **The destruction phase**



Lifecycle of a Spring Application Context
(3) The Destruction Phase

- When you close a context the destruction phase completes

```
// Need a configurable ApplicationContext
ConfigurableApplicationContext context =
      SpringApplication.run( ApplicationConfig.class );

// Destroy the application
context.close();
```

- *But exactly what happens in this phase?*

Application Context Destruction Lifecycle (1)

- Destroy bean instances if instructed
  - Call their destroy (clean-up) methods
  - Beans must have a *destroy method* defined
    - A no-arg method returning void
- Context then destroys (cleans-up) itself
  - The context is not usable again

Remember: *only GC actually destroys objects*

A method on the AccountRepository

```
@Bean (destroyMethod="clearCache")
public AccountRepository accountRepository() {
   // ...
}
```

- Called only when ApplicationContext / JVM exit *normally*
- *Not* called for *prototype* beans

spring Pivotal

Application Context Destruction Lifecycle (2)

- Can do the same using XML or annotations
  - Annotations require *annotation-config* or the component scanner to be activated

Using XML

```
<bean id="accountRepository"
    class="app.impl.AccountRepository"
    destroy-method="clearCache">
  …
</bean>
```

By Annotation

```
public class AccountRepository {

  @PreDestroy
  void clearCache() {
    // close files, connections ...
    // remove external resources ...
  }
}
```

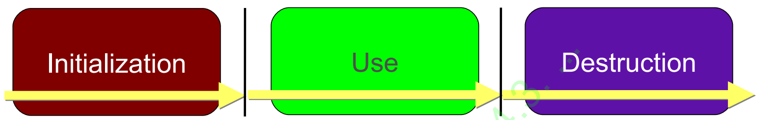
```
<context:annotation-config/>
```

```
<context:component-scan … />
```

spring Pivotal

Topics Covered

- Spring Lifecycle
  - The initialization phase
    - Bean Post Processors for *initialization* and *proxies*
  - The use phase
    - Proxies at Work – most of Spring's “magic” uses a proxy
  - The destruction phase
    - Allow application to terminate cleanly



Testing Spring Applications

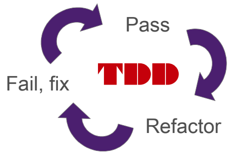
Unit Testing without Spring
Integration Testing with Spring

Testing in General, Spring and JUnit,
Profiles, Database Testing

Topics in this Session

- **Test Driven Development**
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

What is TDD?



- TDD = Test Driven Development
  – Is it writing tests before the code?
  – Is it writing tests at the same time as the code?
  – Ultimately that is not what is most important
- TDD is about:
  – Writing automated tests that verify code actually works
  – Driving development with well defined requirements in the form of tests

"But We Don't Have Time to Write Tests!"

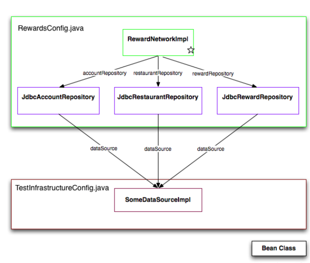
- Every development process includes testing
  – Either automated or manual
- Automated tests result in a faster development cycle overall
  – Your IDE is better at this than you are
- Properly done TDD is faster than development without tests

TDD and Agility

- Comprehensive test coverage provides confidence
- Confidence enables refactoring
- Refactoring is essential to agile development

TDD and Design

- Testing makes you think about your design
- If your code is hard to test then the design should be reconsidered

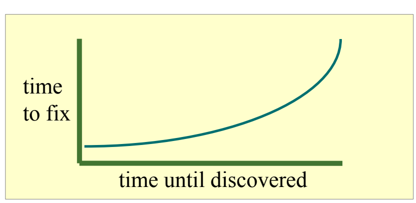


TDD and Focus

- A test case helps you focus on what matters
- It helps you not to write code that you don't need
- Find problems early

Benefits of Continuous Integration

- The cost to fix a bug grows exponentially in proportion to the time before it is discovered



- Continuous Integration (CI) focuses on reducing the time before the bug is discovered
  – Effective CI requires automated tests

Topics in this Session

- Test Driven Development
- **Integration Testing with Spring**
- Testing with Profiles
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

Unit Testing

Unit Testing
Without Spring

- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
  - Uses simplified alternatives for dependencies
    - Stubs and/or Mocks
    - *See Appendix for more details*

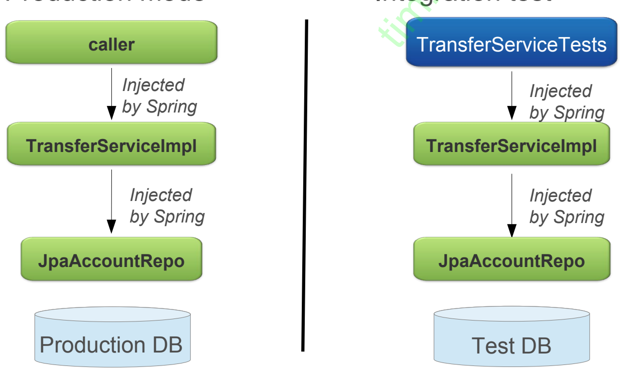
Integration Testing

Integration Testing *With* Spring

- Integration (System) Testing
  - Tests the interaction of multiple units working together
    - All should work individually (unit tests showed this)
- Tests application classes in context of their surrounding infrastructure
  - Out-of-container testing, no need to run up full JEE system
  - Infrastructure may be “scaled down”
    - Use Apache DBCP connection pool instead of container-provider pool obtained through JNDI
    - Use ActiveMQ to save expensive commercial JMS licenses

Integration test example

- Production mode
- Integration test



Spring’s Integration Test Support

- Packaged as a separate module
  - spring-test.jar
- Consists of several JUnit test support classes
- Central support class is *SpringJUnit4ClassRunner*
  - Caches a *shared* ApplicationContext across test methods

See: Spring Framework Reference – Integration Testing
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#integration-testing

Using Spring's Test Support

Run with Spring support

Point to system test configuration file

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void shouldTransferMoneySuccessfully() {
        TransferConfirmation conf = transferService.transfer(...);
        ...
    }
}
```

Inject bean to test

Test the system as normal

No need for @Before method

Including Configuration as an inner class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class JdbcAccountRepoTest {

  private JdbcAccountRepo repo = ...;

  @Test
  public void shouldUpdateDatabaseSuccessfully() {...}

  @Configuration
  @Import(SystemTestConfig.class)
  static  class TestConfiguration {
    @Bean public DataSource dataSource() { … }
  }
}
```

Don't specify config classes

Looks for configuration *embedded* in the class

Override a bean with a test alternative

spring Pivotal

@ContextConfiguration – XML

- Tests when using XML based configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:com/acme/system-test-config.xml")
public final class TransferServiceTests { … }
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"classpath:config-1.xml", "file:db-config.xml"})
public final class TransferServiceTests { … }
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class TransferServiceTests { … }
```

Defaults to ${classname}-context.xml in same package

Loads `TransferServiceTests-context.xml`

spring Pivotal

Multiple test methods

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() {
        ...
    }

    @Test
    public void failedTransfer() {
        ...
    }
}
```

The ApplicationContext is instantiated only *once* for all tests that use the same set of config files (even across test classes)

Annotate test method with @DirtiesContext to force recreation of the cached ApplicationContext *if* method changes the contained beans

spring Pivotal

Test Property Sources

- Custom properties *just* for testing
  - Specify one or more properties
    - Will override any existing properties of same name
  - Specify location of one or more properties files to load
    - Defaults to looking for ***[classname]*.properties**

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
@TestPropertySource(properties = { "username=foo", "password=bar" }
                    locations = "classpath:/transfer-test.properties")
public final class TransferServiceTests {
    ...
}
```

spring Pivotal

Spring 4.3 Simplification

Spring 4.3

- Can use **`SpringRunner`** as an alternative to the **`SpringJUnit4ClassRunner`**
  - Simply a sub-class with a nicer name

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public final class TransferServiceTests {
    ...
}
```

spring Pivotal

Benefits of Testing with Spring

- No need to deploy to an external container to test application functionality
  - Run everything quickly inside your IDE
  - Supports *Continuous Integration* testing
- Allows reuse of your configuration between test and production environments
  - Application configuration logic is typically reused
  - Infrastructure configuration is environment-specific
    - DataSources
    - JMS Queues

spring Pivotal

Topics in this Session

- Test Driven Development
- Integration Testing with Spring
- **Testing with Profiles**
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

Activating Profiles For a Test

- @ActiveProfiles inside the test class
  - Define one or more profiles
  - Beans associated with that profile are instantiated
  - Also beans not associated with *any* profile

- Example: Two profiles activated – *jdbc* **and** *dev*

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DevConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )

public class TransferServiceTests { … }
```

Profiles Activation with JavaConfig

- @ActiveProfiles inside the test class
- @Profile inside the @Configuration class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DevConfig.class)
@ActiveProfiles("jdbc")
public  class TransferServiceTests
{...}
```

```
@Configuration
@Profile("jdbc")
public class DevConfig {

    @Bean
    {...}
}
```

Remember: only @Configurations matching an active profile or with *no* profile are loaded

spring Pivotal

Profiles Activation with Annotations

- @ActiveProfiles inside the test class
- @Profile inside the Component class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DevConfig.class)
@ActiveProfiles("jdbc")
public  class TransferServiceTests
{...}
```

```
@Repository
@Profile("jdbc")
public class
JdbcAccountRepository
{ ...}
```

Only beans with current profile / no profile are component-scanned

spring Pivotal

Profiles Activation with XML

- @ActiveProfiles inside the test class
- profile attribute inside <bean> tag

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("infra-test-conf.xml")
@ActiveProfiles("jdbc")

public class TransferServiceTests
{...}
```

```
<beans xmlns=...>
  <!-- Available to all profiles →
  <bean id="rewardNetwork" … />
  ...
  <beans profile="jdbc"> ... </beans>
  <beans profile="jpa"> ... </beans>
</beans>
```

Only beans with current profile / no profile are loaded

Topics in this Session

- Test Driven Development
- Integration Testing with Spring
- Testing with Profiles
- **Testing with Databases**
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

Spring 4.1

Testing with Databases

- Integration testing against SQL database is common.
- In-memory databases useful for this kind of testing
  - No prior install needed
- Common requirement: populate DB before test runs
  - Use the @Sql annotation:

```
@Test
@Sql ( "/testfiles/test-data.sql" )
public void successfulTransfer() {
    ...
}
```

Run this SQL command
Before this test method executes.

See: Spring Framework Reference, Executing SQL Scripts
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testcontext-executing-sql

spring Pivotal

Spring 4.1

@Sql Examples

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(...)
@Sql({ "/testfiles/schema.sql", "/testfiles/general-data.sql" } )
public final class MainTests {

    @Test
    @Sql
    public void success() { … }

    @Test
    @Sql ( "/testfiles/error.sql" )
    @Sql ( scripts="/testfiles/cleanup.sql",
           executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )
    public void transferError() { … }
}
```

Run these scripts before *each* **@Test** method

Run script named (by default) ***MainTests.success.sql*** in same package

Run before **@Test** method...

...run after **@Test** method

spring Pivotal

@Sql Options

- When does the SQL run?
  - *executionPhase:* before (default) or after the test method
  - *config:* SqlConfig has many options to control SQL scripts
    - What to do if script fails? **FAIL_ON_ERROR**, **CONTINUE_ON_ERROR**, **IGNORE_FAILED_DROPS**, **DEFAULT***
    - SQL syntax control: comments, statement separator

```
@Sql(  scripts = "/test-user-data.sql",
       executionPhase = ExecutionPhase.AFTER_TEST_METHOD,
       config = @SqlConfig(errorMode = ErrorMode.FAIL_ON_ERROR,
                           commentPrefix = "//", separator = "@@")  )
```

***DEFAULT** = whatever @Sql defines at class level, otherwise **FAIL_ON_ERROR**

Summary

- Testing is an *essential* part of any development
- Unit testing tests a class in isolation
  - External dependencies should be minimized
  - Consider creating stubs or mocks to unit test
  - *You don't need Spring to unit test*
- Integration testing tests the interaction of multiple units working together
  - Spring provides good integration testing support
  - Profiles for different test & deployment configurations
  - Built-in support for testing with Databases

Lab

Testing Spring Applications

Coming Up: Appendix on Unit Testing using Stubs or Mocks

spring Pivotal

Topics in this Session

- Test Driven Development
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- **Appendix**
  - **Unit Testing (Stubs & Mocks)**

spring Pivotal

Unit Testing vs. Integration Testing

- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
- Integration Testing
  - Tests the interaction of multiple units working together
  - Integrates infrastructure
- Discussed Integration Testing earlier
  - Let's discuss Unit Testing here
  - Remember: *Unit Testing does not use Spring*

spring Pivotal

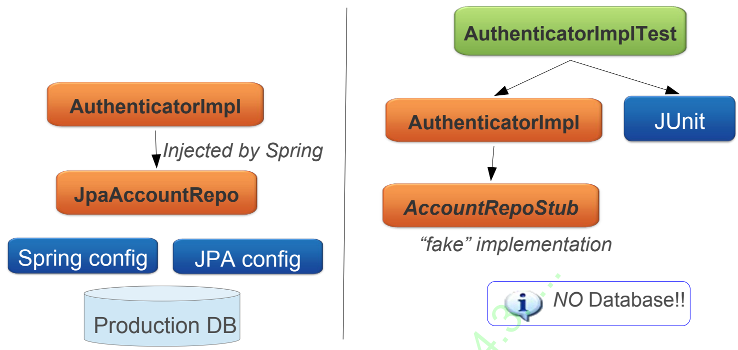
Unit Testing

- Remove links with dependencies
  - The test shouldn't fail because of external dependencies
  - Spring is also considered as a dependency
  -
- 2 ways to create a "testing-purpose" implementation of your dependencies:
  - Stubs Create a simple test implementation
  - Mocks Dependency class generated at startup-time using a "Mocking framework"

spring Pivotal

Unit Testing example

- Production mode
- Unit test with Stubs



spring Pivotal

Example Unit to be Tested

```
public class AuthenticatorImpl implements Authenticator {
  private AccountRepository accountRepository;

  public AuthenticatorImpl(AccountRepository accountRepository) {
    this.accountRepository = accountRepository;
  }

  public boolean authenticate(String username, String password) {
     Account account = accountRepository.getAccount(username);

     return account.getPassword().equals(password);
  }
}
```

External dependency

Unit *business* logic
– 2 paths: success or fail

Note: Validation failure paths ignored for simplicity

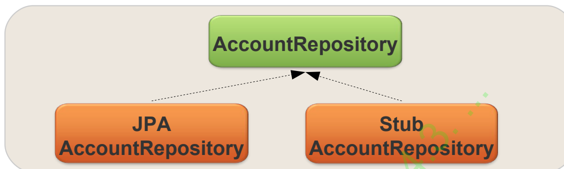
spring Pivotal

Implementing a Stub

- Class created manually
  - Implements Business interface

```
class StubAccountRepository implements AccountRepository {
  public Account getAccount(String user) {
    return "lisa".equals(user) ? new Account("lisa", "secret") : null;
  }
}
```

Simple state



Unit Test using a Stub

```
import org.junit.Before; import org.junit.Test; ...

public class AuthenticatorImplTests {

  private AuthenticatorImpl authenticator;

  @Before public void setUp() {
    authenticator = new AuthenticatorImpl( new StubAccountRepository() );
  }

  @Test public void successfulAuthentication() {
    assertTrue(authenticator.authenticate("lisa", "secret"));
  }

  @Test public void invalidPassword() {
    assertFalse(authenticator.authenticate("lisa", "invalid"));
  }
}
```

Spring **not** in charge of injecting dependencies

OK scenario

KO scenario

Unit Testing with Stubs

- Advantages
  - Easy to implement and understand
  - Reusable
- Disadvantages
  - Change to an interface requires change to stub
  - Your stub must implement all methods
    - even those not used by a specific scenario
  - If a stub is reused refactoring can break other tests

spring Pivotal

Steps to Testing with a Mock

1. Use a mocking library to generate a mock object
   - Implements the dependent interface on-the-fly
2. Record the mock with expectations of how it will be used for a scenario
   - What methods will be called
   - What values to return
3. Exercise the scenario
4. Verify mock expectations were met

spring Pivotal

EasyMock

Example: Using a Mock - I

- Setup
  - A Mock class is created at startup time

```
import static org.easymock.classextensions.EasyMock.*;

public class AuthenticatorImplTests {
   private AccountRepository accountRepository
               = createMock(AccountRepository.class);

   private AuthenticatorImpl authenticator
          = new AuthenticatorImpl(accountRepository);

   // continued on next slide ...
```

static import

Implementation of interface *AccountRepository* is created

spring Pivotal

EasyMock

Example: Using a Mock - II

```
   // … continued from previous slide

 @Test public void validUserWithCorrectPassword() {
   expect(accountRepository.getAccount("lisa")).
      andReturn(new Account("lisa", "secret"));


   replay(accountRepository);


   assertTrue( authenticator.
        authenticate("lisa", "secret") );


   verify(accountRepository);
  }
}
```

Recording

What behavior to expect?

Recording *Playback*

"playback" mode

Mock now fully available

Verification

No planned method call has been omitted

spring Pivotal

Mockito

Same Example using Mockito

```
import static org.mockito.Mockito.*;

public class AuthenticatorImplTests {
   private AccountRepository accountRepository
         = mock( AccountRepository.class );            // Create a mock object
   private AuthenticatorImpl authenticator
         = new AuthenticatorImpl(accountRepository);  // Inject the mock object

 @Test public void validUserWithCorrectPassword() {
   when(accountRepository.getAccount("lisa")).              // Train the mock
       thenReturn(new Account("lisa", "secret"));

   assertTrue( authenticator.authenticate("lisa", "secret") );      // Run test
    verify(accountRepository);                         // Verify getAccount() was
   }                                                   // invoked on the mock
}
```

No replay() step with Mockito

spring Pivotal

Mock Considerations

- Several mocking libraries available
  - Mockito, JMock, EasyMock
- Advantages
  - No additional class to maintain
  - You only need to setup what is necessary for the scenario you are testing
  - Test behavior as well as state
    - Were all mocked methods used? If not, why not?
- Disadvantages
  - A little harder to understand at first

spring Pivotal

Mocks or Stubs?

- You will probably use both
- General recommendations
  - Favor mocks for non-trivial interfaces
  - Use stubs when you have simple interfaces with repeated functionality
  - Always consider the specific situation
- Read "Mocks Aren't Stubs" by Martin Fowler
  - http://www.martinfowler.com/articles/mocksArentStubs.html

spring Pivotal

Developing Aspects with Spring AOP

Aspect Oriented Programming For Declarative Enterprise Services

Using and Implementing Spring Proxies

Topics in this session

- **What Problem Does AOP Solve?**
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics

What Problem Does AOP Solve?

- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns

spring Pivotal

What are Cross-Cutting Concerns?

- Generic functionality that is needed in many places in your application
- Examples
  - Logging and Tracing
  - Transaction Management
  - Security
  - Caching
  - Error Handling
  - Performance Monitoring
  - Custom Business Rules

spring Pivotal

An Example Requirement

- Perform a role-based security check before every application method

A sign this requirement is a cross-cutting concern

Implementing Cross Cutting Concerns Without Modularization

- Failing to modularize cross-cutting concerns leads to two things
  - Code tangling
    - A coupling of concerns
  - Code scattering
    - The same concern spread across modules

Symptom #1: Tangling

```
public class RewardNetworkImpl implements RewardNetwork {
   public RewardConfirmation rewardAccountFor(Dining dining) {
      if (!hasPermission(SecurityContext.getPrincipal()) {
         throw new AccessDeniedException();
      }

      Account a = accountRepository.findByCreditCard(…
      Restaurant r = restaurantRepository.findByMerchantNumber(…
      MonetaryAmount amt = r.calculateBenefitFor(account, dining);
      …
   }
}
```

Mixing of concerns

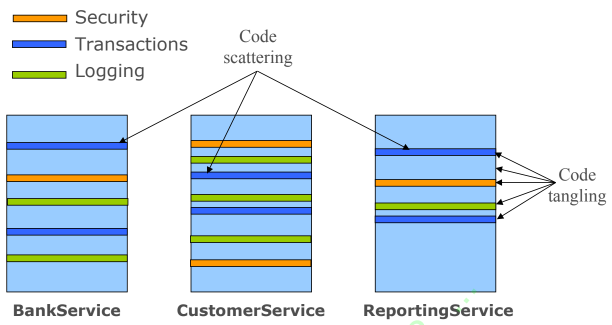
Symptom #2: Scattering

```
public class JpaAccountManager implements AccountManager {
   public Account getAccountForEditing(Long id) {
      if (!hasPermission(SecurityContext.getPrincipal()) {
         throw new AccessDeniedException();
      }
      …
```

Duplication

```
public class JpaMerchantReportingService
        implements MerchantReportingService {
   public List<DiningSummary> findDinings(String merchantNumber,
                                          DateInterval interval) {
      if (!hasPermission(SecurityContext.getPrincipal()) {
         throw new AccessDeniedException();
      }
      …
```

System Evolution Without Modularization



Aspect Oriented Programming (AOP)

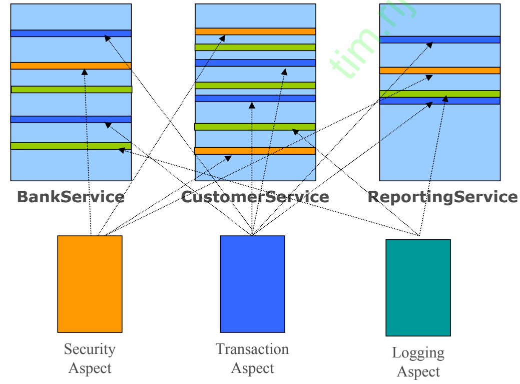
- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns
  - To avoid tangling
  - To eliminate scattering

spring Pivotal

How AOP Works

- Implement your mainline application logic
  - Focusing on the core problem
- Write aspects to implement your cross-cutting concerns
  - Spring provides many aspects out-of-the-box
- Weave the aspects into your application
  - Adding the cross-cutting behaviours to the right places

System Evolution: AOP based



Leading AOP Technologies

- AspectJ
  - Original AOP technology (first version in 1995)
  - A full-blown Aspect Oriented Programming language
    - Uses byte code modification for aspect weaving
- Spring AOP
  - Java-based AOP framework with AspectJ integration
    - Uses dynamic proxies for aspect weaving
  - Focuses on using AOP to solve enterprise problems
  - The focus of this session

See: Spring Framework Reference – Aspect Oriented Programming
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop

spring Pivotal

Topics in this session

- What Problem Does AOP Solve?
- **Core AOP Concepts**
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics

spring Pivotal

Core AOP Concepts

- Join Point
  - A point in the execution of a program such as a method call or exception thrown
- Pointcut
  - An expression that selects one or more Join Points
- Advice
  - Code to be executed at each selected Join Point
- Aspect
  - A module that encapsulates pointcuts and advice
- Weaving
  - Technique by which aspects are combined with main code

spring Pivotal

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- **Quick Start**
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics

spring Pivotal

AOP Quick Start

- Consider this basic requirement

Log a message every time a property is about to change

- How can you use AOP to meet it?

An Application Object Whose Properties Could Change

```
public class SimpleCache implements Cache
{
  private int cacheSize;
  private DataSource dataSource;
  private String name;

  public SimpleCache(String beanName) { name = beanName; }

  public void setCacheSize(int size) { cacheSize = size; }

  public void setDataSource(DataSource ds) { dataSource = ds; }

   …

  public String toString() { return name; }       // For convenience later
}
```

```
public interface Cache {
  public void setCacheSize(int size);
}
```

Implement the Aspect

```java
@Aspect
@Component
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

   @Before(“execution(void set*(*))”)
    public void trackChange() {
        logger.info(“Property about to change…”);
    }
}
```

Configure Aspect as a Bean

Using Java

```java
@Configuration
@EnableAspectJAutoProxy
@ComponentScan(basePackages=“com.example”)
public class AspectConfig {
   …
}
```

Configures Spring to apply @Aspect to your beans

OR

Using XML

```xml
<beans>
   <aop:aspectj-autoproxy />

   <context:component-scan  base-package=“com.example” />
</beans>
```

Include the Aspect Configuration

```
@Configuration
@Import(AspectConfig.class)
public class MainConfig {

    @Bean
    public Cache cacheA() { return new SimpleCache("cacheA"); }

    @Bean
    public Cache cacheB() { return new SimpleCache("cacheB"); }

    @Bean
    public Cache cacheC() { return new SimpleCache("cacheC"); }
}
```

Include aspect configuration

spring Pivotal

Test the Application

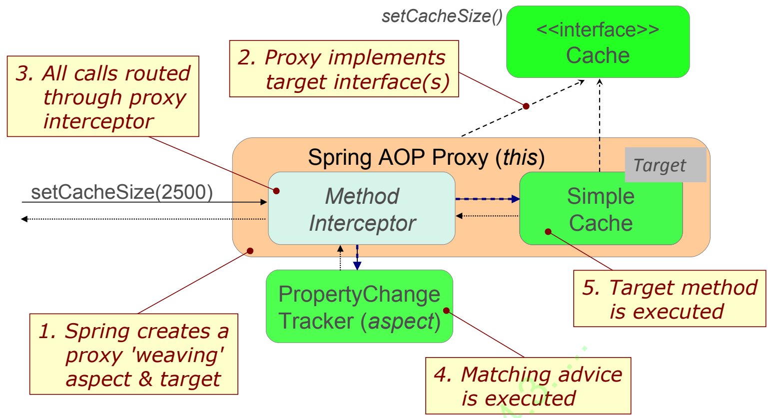
```
ApplicationContext context = SpringApplication.run(MainConfig.class);
```

```
@Autowired @Qualifier("cacheA");
private Cache cache;
…
cache.setCacheSize(2500);
```

```
INFO: Property about to change…
```

spring Pivotal

How Aspects are Applied



Tracking Property Changes – With Context

- Context provided by the *JoinPoint* parameter

```
@Aspect
public class PropertyChangeTracker {
   private Logger logger = Logger.getLogger(getClass());


   @Before("execution(void set*(*))")
   public void trackChange(JoinPoint point) {
      String name = point.getSignature().getName();
      Object newValue = point.getArgs()[0];
      logger.info(name + " about to change to " +
                  newValue + " on " +
                  point.getTarget());
   }
}
```

Context about the intercepted point

toString() returns bean-name

INFO: setCacheSize about to change to 2500 on cacheA

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- **Defining Pointcuts**
- Implementing Advice
- Lab
- Advanced Topics

spring Pivotal

Defining Pointcuts

- Spring AOP uses AspectJ's pointcut expression language
  – For selecting where to apply advice
- Complete expression language reference available at
  – http://www.eclipse.org/aspectj/docs.php
- Spring AOP supports a practical subset

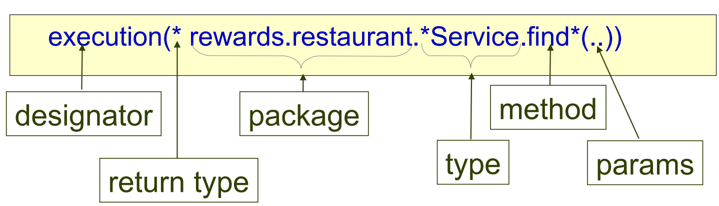
See: Spring Framework Reference – Declaring a Pointcut
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop-pointcuts

spring Pivotal

Common Pointcut Designator

- execution(<method pattern>)
  - The method must match the pattern
- Can chain together to create composite pointcuts
  - && (and), || (or), ! (not)
- Method Pattern
  - [Modifiers] ReturnType [ClassType]
    MethodName ([Arguments]) [throws ExceptionType]

Writing Expressions



Execution Expression Examples
Any Class or Package

execution(void send*(String))

- Any method starting with send that takes a single String parameter and has a void return type

execution(* send(*))

- Any method named send that takes a single parameter

execution(* send(int, ..))

- Any method named send whose first parameter is an int (the ".." signifies 0 or more parameters may follow)

spring Pivotal

Execution Expression Examples
Implementations vs Interfaces

- Restrict by *class*

  execution(void example.MessageServiceImpl.*(..))

  – Any void method in the *MessageServiceImpl* class
    - Including any sub-class

  – But will be ignored if a different implementation is used
- Restrict by *interface*

  execution(void example.MessageService.send(*))

  – Any void *send* method taking one argument, in any object implementing *MessageService*

  – More flexible choice – works if implementation changes

spring Pivotal

Execution Expression Examples
Using Annotations

execution(@javax.annotation.security.RolesAllowed void send*(..))

- Any void method whose name starts with "*send*" that is annotated with the *@RolesAllowed* annotation

```
public interface Mailer {
    @RolesAllowed("USER")
    public void sendMessage(String text);
}
```

- Ideal for your own classes
  - Matches if annotation is present
  - Not if it isn't

spring Pivotal

Execution Expression Examples
Working with Packages

execution(* rewards.*.restaurant.*.*(..))

- There is one directory between rewards and restaurant

execution(* rewards..restaurant.*.*(..))

- There may be several directories between rewards and restaurant

execution(* *..restaurant.*.*(..))

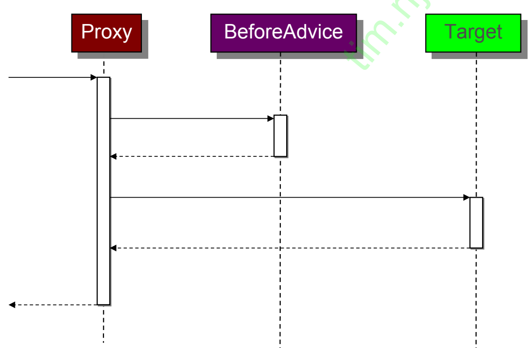
- Any sub-package called restaurant

spring Pivotal

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- **Implementing Advice**
- Lab
- Advanced Topics

Advice Types: Before



Before Advice Example

- Use @*Before* annotation

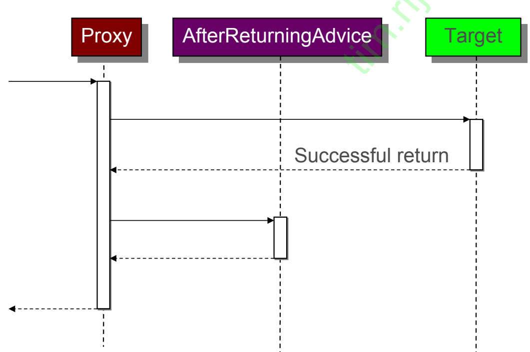
Track calls to all setter methods

```
@Aspect
public class PropertyChangeTracker {
   private Logger logger = Logger.getLogger(getClass());

   @Before(“execution(void set*(*))”)
   public void trackChange() {
      logger.info(“Property about to change…”);
   }
}
```

- **Note:** if the advice throws an exception, target will not be called – this is a valid use of a *Before Advice*

spring Pivotal

Advice Types:After Returning



spring Pivotal

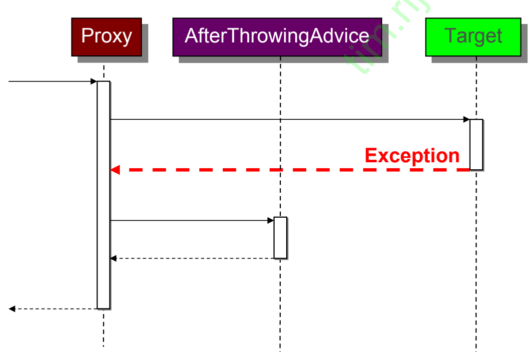
After Returning Advice - Example

- Use *@AfterReturning* annotation with the *returning* attribute

Audit all operations in the *service* package that return a *Reward* object

```
@AfterReturning(value="execution(* service..*.*(..))",
                        returning="reward")
public void audit(JoinPoint jp, Reward reward) {
    auditService.logEvent(jp.getSignature() +
        " returns the following reward object :"  + reward.toString() );
}
```


Advice Types: After Throwing



After Throwing Advice - Example

- Use @*AfterThrowing* annotation with the *throwing* attribute
  - Only invokes advice if the right exception type is thrown

Send an email every time a Repository class throws an exception of type DataAccessException

```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
   mailService.emailFailure("Exception in repository", jp, e);
}
```

spring Pivotal

After Throwing Advice - Propagation

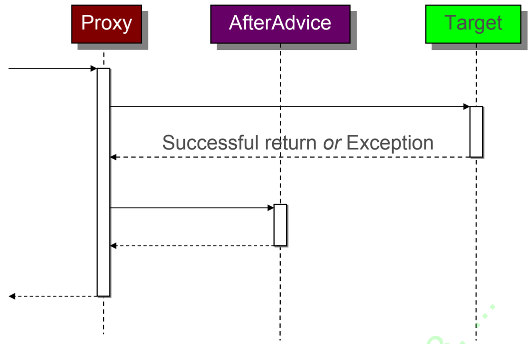
- The @AfterThrowing advice will not stop the exception from propagating
  - However it can throw a different type of exception

```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
   mailService.emailFailure("Exception in repository", jp, e);
   throw new RewardsException(e);
}
```

If you wish to stop the exception from propagating any further, you can use an @Around advice (see later)

spring Pivotal

Advice Types: After



After Advice Example

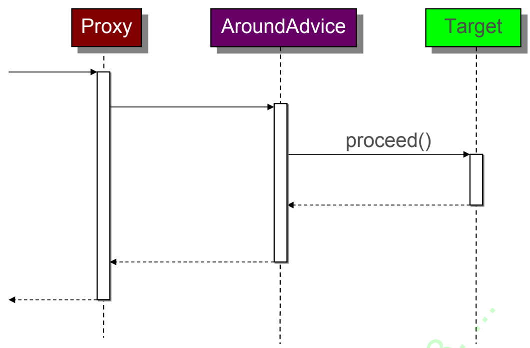
- Use @*After* annotation
  - Called regardless of whether an exception has been thrown by the target or not

Track calls to all update methods

```
@Aspect
public class PropertyChangeTracker {
   private Logger logger = Logger.getLogger(getClass());

   @After("execution(void update*(..))")
   public void trackUpdate() {
      logger.info("An update has been attempted …");
   }
}
```

We don't know how the method terminated

Advice Types: Around



Around Advice Example

- Use *@Around* annotation *and* a *ProceedingJoinPoint*
  - Inherits from *JoinPoint* and adds the *proceed()* method

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point)  throws Throwable {
    Object value = cacheStore.get(CacheUtils.toKey(point));

    if (value == null) {
        value = point.proceed();
        cacheStore.put(CacheUtils.toKey(point), value);
    }

    return value;
}
```

Proceed *only* if not already cached

Cache values returned by *cacheable* services

Limitations of Spring AOP

- Can only advise *non-private* methods
- Can only apply aspects to *Spring Beans*
- Limitations of weaving with proxies
  - When using proxies, suppose method a() calls method b() on the *same* class/interface
    - advice will *never* be executed for method b()

Summary

AOP

- Aspect Oriented Programming (AOP) *modularizes cross-cutting concerns*
- An aspect is a module (Java class) containing the cross-cutting behavior
  - Annotated with @*Aspect*
  - Behavior is implemented as an "*advice*" method
  - Pointcuts select *joinpoints* (methods) where *advice* applies
  - Five advice types
    - Before, AfterThrowing, AfterReturning, After and Around

Lab

Developing Aspects using Spring AOP

Note: The lab is working when
1) your unit test is green *and*
2) you get console logging output

Coming Up: Named pointcuts, context selection, annotations in pointcuts

spring Pivotal

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
  - **XML Configuration**
  - Named Pointcuts
  - Context-Selecting Pointcuts
  - Working with Annotations

spring Pivotal

Alternative Spring AOP Syntax - XML

- XML Based Alternative to @Annotations
  - More centralized configuration
- Approach
  - Aspect logic defined Java
  - Aspect configuration in XML
    - Uses the aop namespace

Tracking Property Changes - Java Code

```
public class PropertyChangeTracker {
    public void trackChange(JoinPoint point) {
        …
    }
}
```

Aspect is a Plain Java Class with no annotations

Tracking Property Changes - XML Configuration

- XML configuration uses the *aop* namespace

```
<aop:config>
   <aop:aspect ref="propertyChangeTracker">
      <aop:before pointcut="execution(void set*(*))" method="trackChange"/>
   </aop:aspect>
</aop:config>

<bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />
```


Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
  - XML Configuration
  - **Named Pointcuts**
  - Context-Selecting Pointcuts
  - Working with Annotations

Named Pointcuts in XML

- A pointcut expression can have a name
  - Reuse it in multiple places

```
<aop:config>
  <aop:pointcut id="setterMethods" expression="execution(void set*(*))"/>

  <aop:aspect ref="propertyChangeTracker">
    <aop:after-returning pointcut-ref="setterMethods" method="trackChange"/>
    <aop:after-throwing pointcut-ref="setterMethods" method="logFailure"/>
  </aop:aspect>
</aop:config>

<bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />
```

spring Pivotal

Named Pointcut Annotation

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("serviceMethod() || repositoryMethod()")
    public void monitor() {
        logger.info("A business method has been accessed…");
    }

    @Pointcut("execution(* rewards.service..*Service.*(..))")
    public void serviceMethod() {}

    @Pointcut("execution(* rewards.repository..*Repository.*(..))")
    public void repositoryMethod() {}
}
```

The method *name* becomes the pointcut ID.
The method is *not* executed.

spring Pivotal

Named Pointcuts

- Expressions can be externalized

```
public class Pointcuts {
    @Pointcut("execution(* rewards.service..*Service.*(..))")
    public void serviceMethods() {}
}
```

```
@Aspect
public class ServiceMethodInvocationMonitor {
    private Logger logger = Logger.getLogger(getClass());

    @Before( "com.acme.Pointcuts.serviceMethods()" )
    public void monitor() {
        logger.info("A service method has been accessed…");
    }
}
```

Fully-qualified pointcut name

Named Pointcuts - Summary

- Can break one complicated expression into several sub-expressions
- Allow pointcut expression reusability
- Best practice: consider externalizing expressions into one dedicated class
  - When working with many pointcuts
  - When writing complicated expressions

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
  - XML Configuration
  - Named Pointcuts
  - **Context Selecting Pointcuts**
  - Working with Annotations

spring Pivotal

Context Selecting Pointcuts

- Pointcuts may also select useful join point context
  - The target object
  - Method arguments
  - Annotations associated with the method, target, or arguments
  - The currently executing object (proxy)
- Allows for simple POJO advice methods
  - Alternative to working with a JoinPoint object directly

spring Pivotal

Context Selecting Example

- Consider this basic requirement

Log a message every time Server is about to start

```
public interface Server {
   public void start(Map input);
   public void stop();
}
```

In the advice, how do we access Server? Map?

spring Pivotal

Without Context Selection

- All needed info must be obtained from *JoinPoint* object
  - No type-safety guarantees
  - Write advice *defensively*

```
@Before("execution(void example.Server.start(java.util.Map))")
public void logServerStartup(JoinPoint jp) {
      // A 'safe' implementation would also check target type
      Server server = (Server) jp.getTarget();
      // Don't assume args[0] exists
      Object[] args= jp.getArgs();
      Map map = args.length > 0 ? (Map) args[0] : new HashMap();
      logger.info( server + " starting – params: " + map);
}
```

spring Pivotal

With Context Selection

- Best practice: use context selection
  - Method attributes are bound automatically
  - Types must match or advice skipped

```
@Before("execution(void example.Server.start(java.util.Map))
  && target(server) && args(input)")
public void logServerStartup(Server server, Map input) {
   …
}
```

- target(server) selects the target of the execution (your object)
- this(server) would have selected the proxy

spring Pivotal

Context Selection - Named Pointcut

```
@Before("serverStartMethod(server, input)")
public void logServerStartup(Server server, Map input) {
   …
}
```

'target' binds the server starting up

'args' binds the argument value

```
@Pointcut("execution(void example.Server.start(java.util.Map))
  && target(server) && args(input)")
public void serverStartMethod (Server server, Map input) {}
```

spring Pivotal

Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
  - XML Configuration
  - Named Pointcuts
  - Context-Selecting Pointcuts
  - **Working with Annotations**

Pointcut Expression Examples using Annotations

- Can match annotations everywhere
  - annotated methods, methods with annotated arguments, returning annotated objects, on annotated classes
- execution(@org..transaction.annotation.Transactional * *(..))
  - Any method marked with the @*Transactional* annotation
- execution((@example.Sensitive *) *(..))
  - Any method that returns a type marked as @*Sensitive*

```
@Sensitive
public class MedicalRecord { … }

public class MedicalService {
   public MedicalRecord lookup(...) { … }
}
```

AOP and Annotations - Example

- Use of the *annotation()* designator

```
@Around("execution(* *(..)) && @annotation(txn)")
public Object execute(ProceedingJoinPoint jp, Transactional txn) {
   TransactionStatus tx;

   try {
     TransactionDefinition defintion = new DefaultTransactionDefinition();
     definition.setTimout(txn.timeout());
     definition.setReadOnly(txn.readOnly());
     ...
     tx = txnMgr.getTransaction(definition);
     return jp.proceed();
   }
  … // commit or rollback
}
```

No need for @Transactional in *execution* expression – the *@annotation* matches it instead

AOP and Annotations – Named pointcuts

- Same example using a named-pointcut

```
@Around("transactionalMethod(txn)")
public Object execute(ProceedingJoinPoint jp, Transactional txn) {
   ...
}

@Pointcut("execution(* *(..)) && @annotation(txn)")
public void transactionalMethod(Transactional txn) {}
```

Advanced Topics Summary

- Topics covered were:
  - XML Configuration
  - Named Pointcuts
  - Context-Selecting Pointcuts
  - Working with Annotations

spring Pivotal

Introduction to Data Management with Spring

Implementing Data Access and Caching

Spring's Role in Supporting Data Access in an Enterprise Application

spring Pivotal

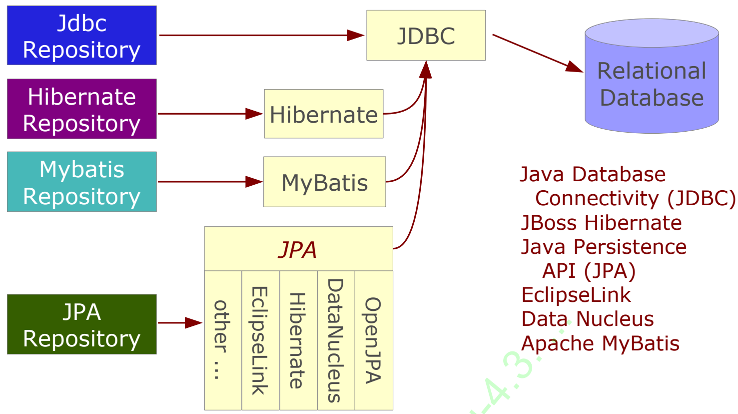
Topics in this Session

- **The Role of Spring in Enterprise Data Access**
- Spring's DataAccessExceptionHierarchy
- Using Test Databases
- Implementing Caching
- NoSQL databases

spring Pivotal

Spring Resource Management Works Everywhere

Works *consistently* with leading data access technologies

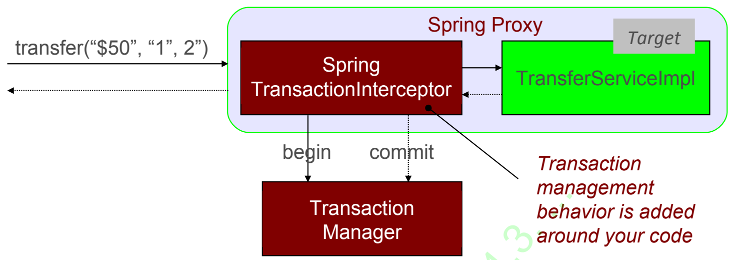


The Resource Management Problem

- Steps Required
  - Access a data source and establish a connection
  - Begin a transaction
  - Do the work – execute business logic
  - Commit or rollback the transaction
  - Close the connection
- Spring Advantages
  - No code to implement (classic cross-cutting concern)
  - No connection or session leakage
  - Throws own exceptions, independent of underlying API

Declarative Transaction Management

```
public class TransferServiceImpl implements TransferService {
   @Transactional   // marks method as needing a txn
    public void transfer(…) {   // your application logic  }
}
```



Template Design Pattern

- Widely used and useful pattern
  - http://en.wikipedia.org/wiki/Template_method_pattern
- Define the outline or skeleton of an algorithm
  - Leave the details to specific implementations later
  - Hides away large amounts of *boilerplate* code
- Spring provides many template classes
  - JdbcTemplate
  - JmsTemplate, RestTemplate, WebServiceTemplate …
  - Most hide low-level resource management

Where are my Transactions?

- Every thread needs its own transaction
  - Typically: a web-driven request
- Spring transaction management
  - Transaction manager handles transaction
    - Puts it into thread-local storage
  - Data-access code, like JdbcTemplate, finds it automatically
    - Or you can get it yourself:

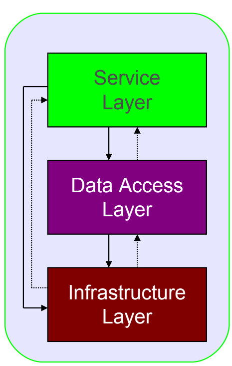
```
DataSourceUtils.getConnection(dataSource)
```

  - Hibernate sessions, JTA (Java EE) work similarly

spring Pivotal

Data Access in a Layered Architecture

- Many enterprise applications consist of three logical layers
  - *Service Layer* (or application layer)
    - Exposes high-level application functions
    - Use-cases, business logic defined here
  - *Data access Layer*
    - Defines interface to the application's data repository (such as a Relational or NoSQL database)
  - *Infrastructure Layer*
    - Exposes low-level services to the other layers



A classic Separation of Concerns

spring Pivotal

Topics in this Session

- The Role of Spring in Enterprise Data Access
- **Spring's DataAccessExceptionHierarchy**
- Using Test Databases
- Implementing Caching
- NoSQL databases

spring Pivotal

Exception Handling

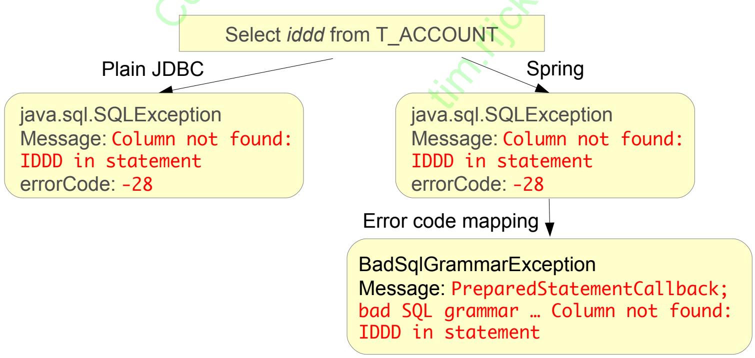
- Checked Exceptions
  - Force developers to handle errors
    - But if you can't handle it, must declare it
  - Bad: intermediate methods must declare exception(s) from *all* methods below
    - A form of tight-coupling
- Unchecked Exceptions
  - Can be thrown up the call hierarchy to the best place to handle it
  - Good: Methods in between don't know about it
    - Better in an Enterprise Application
  - Spring throws Runtime (unchecked) Exceptions

spring Pivotal

Data Access Exceptions

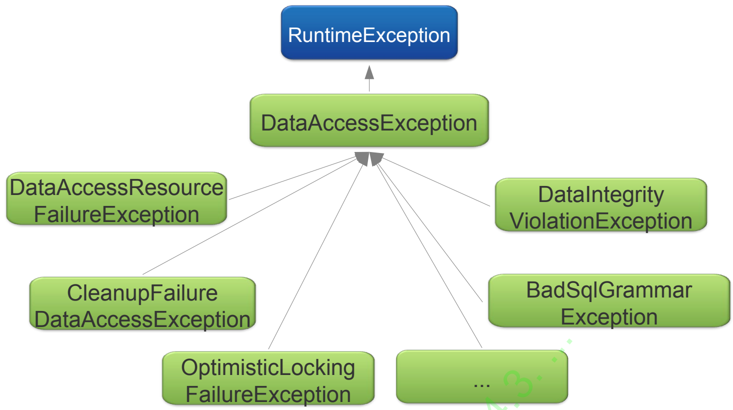
- SQLException
  - Too general – one exception for every database error
  - Calling class 'knows' you are using JDBC
  - Tight coupling
- Spring provides DataAccessException hierarchy
  - Hides whether you are using JPA, Hibernate, JDBC …
  - Actually a hierarchy of sub-exceptions
    - Not just one exception for everything
  - Consistent across all supported Data Access technologies
  - Unchecked

Example: *BadSqlGrammarException*



For more details on error codes: see
spring-jdbc.jar/ org/springframework/jdbc/support/sql-error-codes.xml

Spring Data Access Exceptions



spring Pivotal

Topics in this Session

- The Role of Spring in Enterprise Data Access
- Spring's DataAccessExceptionHierarchy
- **Using Test Databases**
- Implementing Caching
- NoSQL databases

spring Pivotal

Embedded Database Builder

- Conveniently define a new (empty) in-memory database
  - And run script(s) to initialize it
  - HSQL, H2 and Derby are supported

```
@Bean
public DataSource dataSource() {
   EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
   return builder.setName("testdb")
               .setType(EmbeddedDatabaseType.HSQL)
               .addScript("classpath:/testdb/schema.db")
               .addScript("classpath:/testdb/test-data.db").build();
}
```

spring Pivotal

JDBC Namespace Equivalent

- Especially useful for testing
  - Supports H2, HSQL and Derby

```
<bean class="example.order.JdbcOrderRepository" >
   <property name="dataSource" ref="dataSource" />
</bean>

<jdbc:embedded-database id="dataSource" type="H2">
   <jdbc:script location="classpath:schema.sql" />
   <jdbc:script location="classpath:test-data.sql" />
</jdbc:embedded-database>
```

In memory database (created at startup)

spring Pivotal

Initializing an Existing Test Database
XML provides jdbc:initialize-database

- Namespace supports populating other DataSources, too

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="url" value="${dataSource.url}" />
   <property name="username" value="${dataSource.username}" />
   <property name="password" value="${dataSource.password}" />
</bean>

<jdbc:initialize-database data-source="dataSource">
   <jdbc:script location="classpath:schema.sql" />
   <jdbc:script location="classpath:test-data.sql" />
</jdbc:initialize-database>
```

Initializes an ***external*** database

spring Pivotal

Initializing and Existing Test Database
Java Configuration is not so Easy

Explicitly define a database Populator to run the init scripts

```
@Configuration
public class DatabaseInitializer {
    @Value("classpath:schema.sql") private Resource schemaScript;
    @Value("classpath:test-data.sql") private Resource dataScript;

    private DatabasePopulator databasePopulator() {
      final ResourceDatabasePopulator populator =
                       new ResourceDatabasePopulator();
      populator.addScript(schemaScript);
      populator.addScript(dataScript);
      return populator;
    }                       // continued on next slide
```

spring http://stackoverflow.com/questions/16038360 Pivotal

Initializing and Existing Test Database
Java Configuration is not so Easy … (continued)

– Bean can have any name, we won't use it (but Spring will)

```
                    // continued from previous slide
   @Bean
   public DataSourceInitializer anyName(final DataSource dataSource) {
     final DataSourceInitializer initializer = new DataSourceInitializer();
     initializer.setDataSource(dataSource);
     initializer.setDatabasePopulator(databasePopulator());
     return initializer;
   }
}
```

Explicitly create a database initializer which will do the work in its post-construct method

spring http://stackoverflow.com/questions/16038360 Pivotal

Topics in this Session

- The Role of Spring in Enterprise Data Access
- Spring's DataAccessExceptionHierarchy
- Using Test Databases
- **Implementing Caching**
- NoSQL databases

spring Pivotal

About Caching

- What is a cache?
  - In this context: a key-value store = Map
- Where do we use this caching?
  - Any method that always returns the same result for the same argument(s)
    - This method could do anything
      - Calculate data on the fly
      - Execute a database query
      - Request data via RMI, JMS, a web-service ...
  - A unique key must be generated from the arguments
    - That's the cache key

Caching Support

- Transparently applies caching to Spring beans (AOP)
  - Mark methods cacheable
    - Indicate caching key(s)
    - Name of cache to use (multiple caches supported)
  - Define one or more caches in Spring configuration

See: Spring Framework Reference – Cache Abstraction
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#cache

Caching with @Cacheable

- @Cacheable marks a method for caching
  - its result is stored in a cache
  - subsequent invocations (with the *same arguments)*
    - fetch data from cache using key, method not executed
- @Cacheable attributes
  - value: name of cache to use
  - key: the key for each cached data-item
    - Uses SpEL and argument(s) of method

```
@Cacheable(value="topBooks", key="#refId.toUpperCase()")
public Book findBook(String refId) {...}
```

spring Pivotal

Caching via Annotations

Use 'topBooks' cache

Only cache if condition true

use object property

custom key generator

clear cache *before* method invoked

```
public class BookService {

  @Cacheable(value="topBooks", key="#title", condition="#title.length < 32")
  public Book findBook(String title, boolean checkWarehouse) { ... }

  @Cacheable(value="topBooks", key="#author.name")
  public Book findBook2(Author author, boolean checkWarehouse) { ... }

  @Cacheable(value="topBooks", key="T(example.KeyGen).hash(#author)")
  public Book findBook3(Author author, boolean checkWarehouse) { ... }

  @CacheEvict(value="topBooks")
  public void loadBooks() { ... }

  ...
```

spring Pivotal

Enabling Caching Proxy

- Caching must be enabled ...

```
@Configuration
@EnableCaching
public class MyConfig {
    @Bean
    public BookService bookService() { … }
}
```

OR

```
<cache:annotation-driven />

<bean id="bookService" class="example.BookService" />
```

spring Pivotal

Pure XML Cache Setup

- Or use XML instead (for example with third-party class)

```
<bean id="bookService" class="example.BookService">
<aop:config>
  <aop:advisor advice-ref="bookCache"
               pointcut="execution(* *..BookService.*(..))"/>
</aop:config>
<cache:advice id="bookCache" cache-manager="cacheManager">
  <cache:caching cache="topBooks">
    <cache:cacheable method="findBook" key="#refId"/>
    <cache:cache-evict method="loadBooks" all-entries="true" />
  </cache:caching>
</cache:advice>
```

XML Cache Setup – no @Cachable

spring Pivotal

Setup Cache Manager

- Must specify a cache-manager
  - Some provided, or write your own
  - See `org.springframework.cache` package
- SimpleCacheManager
  - For each cache name, it creates a **`ConcurrentHashMap`**

```
@Bean
public CacheManager cacheManager() {
      SimpleCacheManager cacheManager =
           new SimpleCacheManager("topAuthors", "topBooks");
      return cacheManager;
}
```

Keep *cacheManager* bean name

Concurrent Map Cache

spring Pivotal

Third-Party Cache Implementations

- Simple Cache is OK for testing
  - But has no cache control options (overflow, eviction)
- Third-party alternatives
  - Terracotta's EhCache
  - Google's Guava and Caffeine
  - Pivotal's Gemfire

spring Pivotal

Third-Party Cache Manager – EHCache

EHcache

```
@Bean
public CacheManager cacheManager(CacheManager ehCache) {
    EhCacheCacheManager cmgr = new EhCacheCacheManager();
    cmgr.setCacheManager(ehCache);
    return cmgr;
}

@Bean EhCacheManagerFactoryBean
        ehCacheManagerFactoryBean(String location) {
    EhCacheManagerFactoryBean eh = new EhCacheManagerFactoryBean();
    eh.setConfigLocation(context.getResource(location));
    return eh;
}
```

factory creates

Wrapper for an EhCache cache

ehcache.xml

EHCACHE

spring Pivotal

Third-Party Cache Managers – Gemfire

- Gemfire: A distributed, shared nothing data-grid
  - Can be used to setup a distributed cache
  - Caches (regions) replicated across multiple nodes
    - Consistent updates occur on all copies in parallel
    - No loss of data if a storage node fails
    - Automatic recovery and rebalancing

GEMFIRE

Pivotal Gemfire Cache

```
<gfe:cache-manager p:cache-ref="gemfire-cache"/>

<gfe:cache id="gemfire-cache"/>

<gfe:replicated-region id="topAuthors" p:cache-ref="gemfire-cache"/>
<gfe:partitioned-region id="topBooks" p:cache-ref="gemfire-cache"/>
```

spring Pivotal

Spring Gemfire Project

- GemFire configuration in Spring config files
  - Also enables configuration injection for environments
- Features
  - Exception translation
  - GemfireTemplate
  - Transaction management (*GemfireTransactionManager*)
  - Injection of transient dependencies during deserialization
  - *Gemfire Cache Manager class*

spring Pivotal

Topics in this Session

- The Role of Spring in Enterprise Data Access
- Spring's DataAccessExceptionHierarchy
- Using Test Databases
- Implementing Caching
- **NoSQL databases**

spring Pivotal

Not Only Relational

SPRING DATA

- NoSQL
  - Relational databases only store some data
    - LDAP, data-warehouses, files
    - Most documents and spreadsheets aren't in *any* database
- Other database products exist
  - Have strengths where RDB are weak
    - Non-tabular data
      - Hierarchical data: parts inventory, org chart
      - Network structures: telephone cables, roads, molecules
      - Documents: XML, spreadsheets, contracts, ...
      - Geographical data: maps, GPS navigation
      - Many more ...

So Many Choices ...

- Many options – each has a particular strength
  - Document databases
    - MongoDB, *CouchDB coming*
  - Distributed key-value Stores (smart caches)
    - Redis, Riak
  - Network (graph) database
    - Neo4j
  - Big Data
    - Apache Hadoop (VMware Serengeti)
  - Data Grid
    - Gemfire
  - Column Stores *coming*: *HBase*, *Cassandra*

Summary

- Data Access with Spring
  - Enables layered architecture principles
    - Higher layers should not know about data management below
  - Isolate via Data Access Exceptions
    - Hierarchy makes them easier to handle
  - Provides consistent transaction management
  - Supports most leading data-access technologies
    - Relational and non-relational (NoSQL)
  - A key component of the core Spring libraries
  - Automatic caching facility

spring Pivotal

Introduction to Spring JDBC

Using JdbcTemplate

Simplifying JDBC-based data-access with Spring

spring Pivotal

Topics in this Session

- **Problems with traditional JDBC**
  – Results in redundant, error prone code
  – Leads to poor exception handling
- Spring's JdbcTemplate
  – Configuration
  – Query execution
  – Working with result sets
  – Exception handling

See: Spring Framework Reference – Data access with JDBC
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jdbc

spring Pivotal

Redundant, Error Prone Code

```
public List findByLastName(String lastName) {
  List personList = new ArrayList();
  Connection conn = null;
  String sql = "select first_name, age from PERSON where last_name=?";
  try {
    DataSource dataSource = DataSourceUtils.getDataSource();
    conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, lastName);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
      personList.add(new Person(rs.getString("first_name"), ...));
    }
  } catch (SQLException e) { /* ??? */ }
  finally {
    try {
      conn.close();
    } catch (SQLException e) { /* ??? */ }
  }
  return personList;
}
```

Redundant, Error Prone Code

```
public List findByLastName(String lastName) {
  List personList = new ArrayList();
  Connection conn = null;
  String sql = "select first_name, age from PERSON where last_name=?";
  try {
    DataSource dataSource = DataSourceUtils.getDataSource();
    conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, lastName);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
      personList.add(new Person(rs.getString("first_name"), ...));
    }
  } catch (SQLException e) { /* ??? */ }
  finally {
    try {
      conn.close();
    } catch (SQLException e) { /* ??? */ }
  }
  return personList;
}
```

The bold matters - the rest is boilerplate

Poor Exception Handling

```
public List findByLastName(String lastName) {
  List personList = new ArrayList();
  Connection conn = null;
  String sql = "select first_name, age from PERSON where last_name=?";
  try {
    DataSource dataSource = DataSourceUtils.getDataSource();
    conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, lastName);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
      personList.add(new Person(rs.getString("first_name"), ...));
    }
  } catch (SQLException e) { /* ??? */ }
  finally {
    try {
      conn.close();
    } catch (SQLException e) { /* ??? */ }
  }
  return personList;
}
```

What can you do?

Topics in this session

- Problems with traditional JDBC
  - Results in redundant, error prone code
  - Leads to poor exception handling
- **Spring's JdbcTemplate**
  - Configuration
  - Query execution
  - Working with result sets
  - Exception handling

Spring's JdbcTemplate

- Greatly simplifies use of the JDBC API
  - Eliminates repetitive boilerplate code
  - Alleviates common causes of bugs
  - Handles SQLExceptions properly
- Without sacrificing power
  - Provides full access to the standard JDBC constructs

JdbcTemplate in a Nutshell

```
int count = jdbcTemplate.queryForObject(
    "SELECT COUNT(*) FROM CUSTOMER", Integer.class);
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

All handled by Spring

JdbcTemplate Approach Overview

```
List<Customer> results = jdbcTemplate.query(someSql,
  new RowMapper<Customer>() {
    public Customer mapRow(ResultSet rs, int row) throws SQLException {
      // map the current row to a Customer object
    }
  });
```

```
class JdbcTemplate {
  public List<Customer> query(String sql, RowMapper rowMapper) {
    try {
      // acquire connection
      // prepare statement
      // execute statement
      // for each row in the result set
      results.add(rowMapper.mapRow(rs, rowNumber));
      return results;
    } catch (SQLException e) {
      // convert to root cause exception
    }  finally {
      // release connection
    }
  }
```


Creating a JdbcTemplate

- Requires a DataSource

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

- Create a template once and re-use it
  - Do not create one for each thread
  - Thread safe after construction

When to use JdbcTemplate

- Useful standalone
  - Anytime JDBC is needed
  - In utility or test code
  - To clean up messy legacy code
- Useful for implementing a repository in a layered application
  - Also known as a data access object (DAO)

spring Pivotal

Implementing a JDBC-based Repository

```
public class JdbcCustomerRepository implements CustomerRepository {

   private JdbcTemplate jdbcTemplate;

   public JdbcCustomerRepository(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }

   public int getCustomerCount() {
      String sql = "select count(*) from customer";
      return jdbcTemplate.queryForObject(sql, Integer.class);
   }
}
```

No try / catch needed
(unchecked exception)

spring Pivotal

Querying with JdbcTemplate

- JdbcTemplate can query for
  - Simple types (int, long, String, Date, ...)
  - Generic Maps
  - Domain Objects

Query for Simple Java Types

- Query with no bind variables: *queryForObject*

```
public Date getOldest() {
    String sql = "select min(dob) from PERSON";
    return jdbcTemplate.queryForObject(sql, Date.class);
}

public long getPersonCount() {
    String sql = "select count(*) from PERSON";
    return jdbcTemplate.queryForObject(sql, Long.class);
}
```

queryForInt, queryForLong deprecated since Spring 3.2, just as easy to queryForObject instead (API improved in Spring 3)

Query With Bind Variables

- Can query using bind variables: ?
  - Note the use of a variable argument list

```
private JdbcTemplate jdbcTemplate;

public int getCountOfNationalsOver(Nationality nationality, int age) {
    String sql = "select count(*) from PERSON " +
                 "where age > ? and nationality = ?";

    return jdbcTemplate.queryForObject
             (sql, Integer.class, age, nationality.toString());
}
```

Bind to first ? Bind to second ?

spring Pivotal

Generic Queries

- *JdbcTemplate* returns each row of a ResultSet as a **`Map`**
- When expecting a single row
  - Use queryForMap(..)
- When expecting multiple rows
  - Use queryForList(..)
- Useful for *ad hoc* reporting, testing use cases
  - The data fetched does not need mapping to a Java object

ad hoc – created or done for a particular purpose as necessary
– sometimes called "window-on-data" queries

spring Pivotal

Querying for Generic Maps (1)

- Query for a single row

```
public Map<String,Object> getPersonInfo(int id) {
   String sql = "select * from PERSON where id=?";
   return jdbcTemplate.queryForMap(sql, id);
}
```

- returns:

  Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }

A Map of [Column Name | Field Value] pairs

spring Pivotal

Querying for Generic Maps (2)

- Query for multiple rows

```
public List<Map<String,Object>> getAllPersonInfo() {
   String sql = "select * from PERSON";
   return jdbcTemplate.queryForList(sql);
}
```

- returns:

```
List {
   0 - Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }
   1 - Map { ID=2, FIRST_NAME="Jane", LAST_NAME="Doe" }
   2 - Map { ID=3, FIRST_NAME="Junior", LAST_NAME="Doe" }
}
```

A List of Maps of [Column Name | Field Value] pairs

spring Pivotal

Domain Object Queries

- Often it is useful to map relational data into domain objects
  - e.g. a ResultSet to an Account
- Spring's JdbcTemplate supports this using a callback approach
- You may prefer to use ORM for this
  - Need to decide between JdbcTemplate queries and JPA (or similar) mappings
  - Some tables may be too hard to map with JPA

RowMapper

- Spring provides a RowMapper interface for mapping a single row of a ResultSet to an object
  - Can be used for both single and multiple row queries
  - Parameterized as of Spring 3.0

```
public interface RowMapper<T> {
   T mapRow(ResultSet rs, int rowNum)
     throws SQLException;
}
```

Querying for Domain Objects (1)

- Query for single row with JdbcTemplate

```
public Person getPerson(int id) {
    return jdbcTemplate.queryForObject(
        "select first_name, last_name from PERSON where id=?",
        new PersonMapper(), id);
}
```

No need to cast

Maps rows to Person objects

Parameterizes return type

```
class PersonMapper implements RowMapper<Person> {
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException
{
        return new Person(rs.getString("first_name"),
                 rs.getString("last_name"));
    }
}
```

spring Pivotal

Querying for Domain Objects (2)

- Query for multiple rows

No need to cast

```
public List<Person> getAllPersons() {
    return jdbcTemplate.query(
        "select first_name, last_name from PERSON",
        new PersonMapper());
```

Same row mapper can be used

```
class PersonMapper implements RowMapper<Person> {
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Person(rs.getString("first_name"),
                 rs.getString("last_name"));
    }
}
```

spring Pivotal

Querying for Domain Objects (3)

- Simplify using Java 8 Lambda Expressions
  - No need for Mapper class
  - Use inline code instead

```
public List<Person> getAllPersons() {
   return jdbcTemplate.query(
      "select first_name, last_name from PERSON",
      (rs, rowNum) -> {
         return new Person(rs.getString("first_name"),
                           rs.getString("last_name"));
   });
```

Replace RowMapper by a *lambda*

```
public interface RowMapper<T> {
  public T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

spring Pivotal

RowCallbackHandler

- Spring provides a simpler RowCallbackHandler interface when there is no return object
  - Streaming rows to a file
  - Converting rows to XML
  - Filtering rows before adding to a Collection
    - *but filtering in SQL is <u>much</u> more efficient*
  - Faster than JPA equivalent for big queries
    - avoids result-set to object mapping

```
public interface RowCallbackHandler {
   void processRow(ResultSet rs)  throws SQLException;
}
```

spring Pivotal

Using a RowCallbackHandler (1)

```
public class JdbcOrderRepository {
    public void generateReport(Writer out) {
        // select all orders of year 2009 for a full report
        jdbcTemplate.query("select * from order where year=?",
                   new OrderReportWriter(out), 2009);
    }
}
```

returns "void"

```
class OrderReportWriter implements RowCallbackHandler {
    public void processRow(ResultSet rs) throws SQLException {
         // parse current row from ResultSet and stream to output
    }
  // May also be a stateful object: you could accumulate data and add
  //    convenience methods like getTotalOrderValue(), getCount() ...
}
```

spring Pivotal

Using a RowCallbackHandler (2)

- Might use a Lambda – if *no* state needed
  - Need to cast, tells Java which callback lambda replaces

Or using a *lambda*

```
public class JdbcOrderRepository {
  public void generateReport(final PrintWriter out) {
     // select all orders of year 2009 for a full report
     jdbcTemplate.query("select * from order where year=?",
            (RowCallbackHandler)(rs) ->
                  { out.write( rs.getString("customer") … ); },
            2016);
  }
}
```

Cast needed

```
public interface RowCallbackHandler {
   void processRow(ResultSet rs)  throws SQLException;
}
```

spring Pivotal

ResultSetExtractor

- Spring provides a ResultSetExtractor interface for processing an entire ResultSet at once
  - You are responsible for iterating the ResultSet
  - e.g. for mapping entire ResultSet to a single object

```
public interface ResultSetExtractor<T> {
    T extractData(ResultSet rs)  throws SQLException,
                                        DataAccessException;
}
```

You may need this for the lab!

spring Pivotal

Using a ResultSetExtractor (1)

```
public class JdbcOrderRepository {
   public Order findByConfirmationNumber(String number) {
      // execute an outer join between order and item tables
      return jdbcTemplate.query(
              "select...from order o, item i...conf_id = ?",
              new OrderExtractor(), number);
   }
}
```

```
class OrderExtractor implements ResultSetExtractor<Order> {
   public Order extractData(ResultSet rs) throws SQLException {
      Order order = null;
      while (rs.next()) {
         if (order == null) {
            order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);
         }
         order.addItem(mapItem(rs));
      }
      return order;
   }
}
```

spring Pivotal

Using a ResultSetExtractor (2)

Or using a *lambda*

```
public class JdbcOrderRepository {
  public Order findByConfirmationNumber(String number) {
    // execute an outer join between order and item tables
    return jdbcTemplate.query(
      "select...from order o, item i...conf_id = ?",
      (ResultSetExtractor<Order>)(rs) -> {
        Order order = null;
        while (rs.next()) {
          if (order == null)
            order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);

          order.addItem(mapItem(rs));
        }
        return order;
      },
      number);
  }
}
```

```
public interface ResultSetExtractor<T> {
  T extractData(ResultSet rs)
    throws SQLException, DataAccessException;
}
```

spring Pivotal.

Summary of Callback Interfaces

- RowMapper
  - Best choice when *each* row of a ResultSet maps to a domain object
- RowCallbackHandler
  - Best choice when *no value* should be returned from the callback method for *each* row, especially large queries
- ResultSetExtractor
  - Best choice when *multiple* rows of a ResultSet map to a *single* object

spring Pivotal.

Inserts and Updates (1)

- Inserting a new row
  – Returns number of rows modified

```
public int insertPerson(Person person) {
   return jdbcTemplate.update(
      “insert into PERSON (first_name, last_name, age)” +
      “values (?, ?, ?)”,
      person.getFirstName(),
      person.getLastName(),
      person.getAge());
}
```

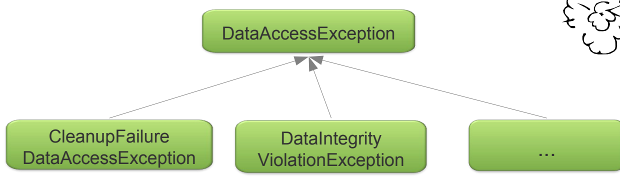

Inserts and Updates (2)

- Updating an existing row

```
public int updateAge(Person person) {
   return jdbcTemplate.update(
         “update PERSON set age=? where id=?”,
         person.getAge(),
         person.getId());
}
```

Exception Handling

- The JdbcTemplate transforms SQLExceptions into DataAccessExceptions



DataAccessException hierarchy was discussed in module “Introduction to Data Access”. You can refer to it for more information on this topic.

Summary

- JDBC is useful
  - But using JDBC API directly is tedious and error-prone
- *JdbcTemplate* simplifies data access and enforces consistency
  - DRY principle hides most of the JDBC
  - Many options for reading data
- *SQLExceptions* typically cannot be handled where thrown
  - Should not be *checked* Exceptions
  - Spring provides *DataAccessException* instead

Lab

Reimplementing repositories using Spring's JdbcTemplate

Transaction Management with Spring

Spring's Consistent Approach

Transactional Proxies and @Transactional

Topics in this session

- **Why use Transactions?**
- Java Transaction Management
- Spring Transaction Management
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

What is a Transaction?

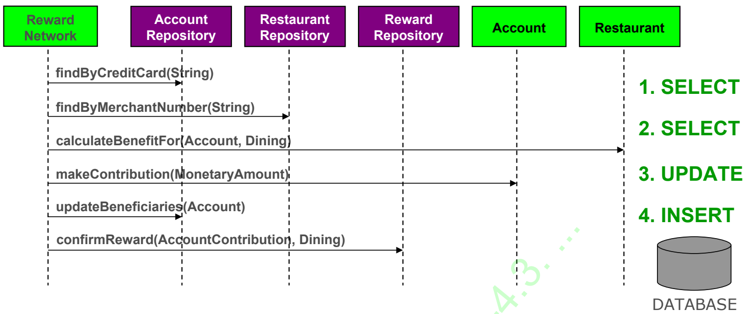
- A set of tasks which take place as a single, indivisible action
  - An *Atomic*, *Consistent*, *Isolated*, *Durable* operation
  - Acronym: **ACID**

Why use Transactions?
To Enforce the ACID Principles

- **A**tomic
  - Each unit of work is an all-or-nothing operation
- **C**onsistent
  - Database integrity constraints are never violated
- **I**solated
  - Isolating transactions from each other
- **D**urable
  - Committed changes are permanent

Transactions in the RewardNetwork

- The *rewardAccountFor(Dining)* method represents a unit-of-work that should be atomic

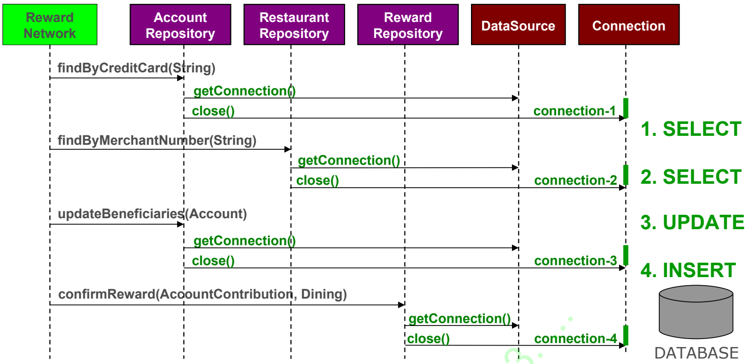


Naïve Approach

Connection per Data Access Operation

- This unit-of-work contains 4 data access operations
  - Each acquires, uses, and releases a distinct Connection
- The unit-of-work is ***non-transactional***

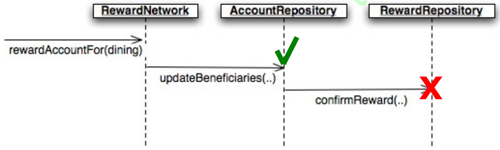
Running non-Transactionally



spring Pivotal

Partial Failures

- Suppose an Account is being rewarded



- If the beneficiaries are updated…
- But the reward confirmation fails…
- There will be no record of the reward!

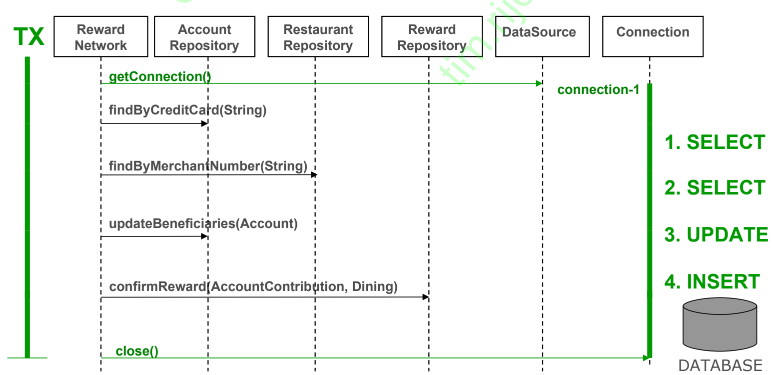
The unit-of-work is **not** *atomic*

spring Pivotal

Correct Approach
Connection per Unit-of-Work

- More efficient
  - Same Connection reused for each operation
- Operations complete as an atomic unit
  - Either all succeed or all fail
- The unit-of-work can run in a ***transaction***

Running in a Transaction



Topics in this session

- Why use Transactions?
- **Java Transaction Management**
- Spring Transaction Management
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

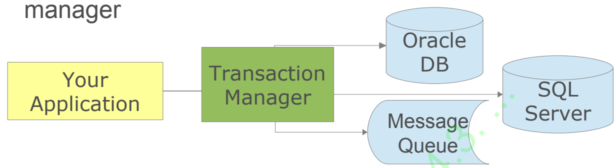
Java Transaction Management

- Java has several APIs which handle transactions differently
  - JDBC, JMS, JTA, Hibernate, JPA, etc.
- Each uses program code to mark the start and end of the transaction
  - Transaction Demarcation
- Different APIs for Global vs Local transactions

Local and Global Transaction Management

- Local Transactions – Single Resource
  - Transactions managed by underlying resource

- Global (distributed) Transactions – Multiple
  - Transaction managed by separate, dedicated transaction manager



JDBC Transaction Management Example

```
try {
    conn = dataSource.getConnection();
    conn.setAutoCommit(false);

    …

    conn.commit();
} catch (Exception e) {
    conn.rollback();
    …
}
```

Specific To JDBC API

Programatic Transaction Demarcation

Checked Exceptions

Code cannot 'join' a transaction already in progress
Code cannot be used with global transaction

JMS Transaction Management Example

```
try {
      session = connection.createSession ( true, 0 );

      …

      session.commit();
   } catch (Exception e) {
      session.rollback();
      …
   }
```

Specific To JMS API

Programatic Transaction Demarcation

Checked Exceptions

Code cannot 'join' a transaction already in progress
Code cannot be used with global transaction

spring Pivotal

JPA / Hibernate Transaction Management Example

API Specific

```
// Hibernate
Transaction tx =
   session.beginTransaction();
…

tx.commit();
```

Programatic Transaction Demarcation

```
// JPA
Transaction tx =
   entityManager.getTransaction();
tx.begin();

…

tx.commit();
```

spring Pivotal

Java Transaction API (JTA) Example

```
try {
      UserTransaction ut =
         (UserTransaction) new InitialContext()
                .lookup("java:comp/UserTransaction");
     ut.begin();
     …

      ut.commit();
   } catch (Exception e) {
      ut.rollback();
      …
   }
```

Programatic Transaction Demarcation

Checked Exceptions

Requires a JTA implementation:
- "Full" application server (WebSphere, WebLogic, JBoss, etc.)
- Standalone implementation (Atomikos, JTOM, etc.)

spring Pivotal

Problems with Java Transaction Management

- Multiple APIs for different local resources
- Programatic transaction demarcation
  - Typically performed in the repository layer (wrong place)
  - Usually repeated (cross-cutting concern)
- Service layer more appropriate
  - Multiple data access methods often called within a single transaction
  - But: don't want data-access code in service-layer
- Orthogonal concerns
  - Transaction demarcation should be independent of transaction implementation

spring Pivotal

Topics in this session

- Why use Transactions?
- Java Transaction Management
- **Spring Transaction Management**
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

spring Pivotal

Spring Transaction Management – 1

- Spring separates transaction *demarcation* from transaction *implementation*
  - Demarcation expressed declaratively via AOP
    - Programatic approach also available
  - PlatformTransactionManager abstraction hides implementation details.
    - Several implementations available
- Spring uses the same API for global vs. local.
  - Change from local to global is minor
    - Just change the transaction manager

spring Pivotal

Spring Transaction Management – 2

- There are only 2 steps
  – Declare a PlatformTransactionManager bean
  – Declare the transactional methods
    - Using Annotations, XML, Programmatic
    - Can mix and match

spring Pivotal

PlatformTransactionManager

- Spring's **PlatformTransactionManager** is the base interface for the abstraction
- Several implementations are available
  – DataSourceTransactionManager
  – HibernateTransactionManager
  – JpaTransactionManager
  – JtaTransactionManager
  – WebLogicJtaTransactionManager
  – WebSphereUowTransactionManager
  – *and more*

Spring allows you to configure whether you use JTA or not.
It does not have *any* impact on your Java classes

spring Pivotal

Deploying the Transaction Manager

- Create the required implementation
  - Just like any other Spring bean
    - Configure as appropriate
  - Here is the manager for a DataSource

A DataSource bean must be defined elsewhere

```
@Bean
public PlatformTransactionManager
            transactionManager(DataSource dataSource) {
   return new DataSourceTransactionManager(dataSource);
}
```

Bean id "*transactionManager*" is default name. Can change it, but must specify alternative name everywhere – easier not to!

spring Pivotal

@Transactional Configuration

In your code

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional
  public RewardConfirmation rewardAccountFor(Dining d) {
    // atomic unit-of-work
  }
}
```

In your Spring configuration

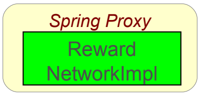
```
@Configuration
@EnableTransactionManagement
public class TxnConfig {
  @Bean
  public PlatformTransactionManager transactionManager(DataSource ds);
     return new DataSourceTransactionManager(ds) {
  }
```

Defines a Bean Post-Processor – proxies @Transactional beans

Or use **<tx:annotation-driven/>** - see Advanced Section

spring Pivotal

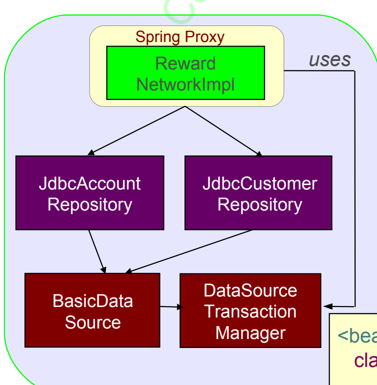
@Transactional: What Happens Exactly?



- Target object wrapped in a proxy
  - Uses an Around advice
- Proxy implements the following behavior
  - Transaction started before entering the method
  - Commit at the end of the method
  - Rollback if method throws a RuntimeException
    - Default behavior
    - Can be overridden (see later)
- Transaction context bound to current thread.
- All controlled by *configuration*

spring Pivotal

Local JDBC Configuration



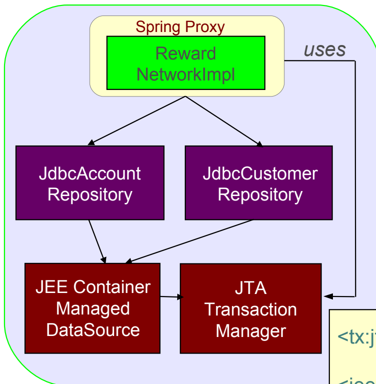
- How?
  - Define local data source
  - DataSource Transaction Manager
- Purpose
  - Integration testing
  - Deploy to Tomcat or other servlet container

```
<bean id="transactionManager"
  class="...DataSourceTransactionManager"> ...

<jdbc:embedded-database id="dataSource"> ...
```

spring Pivotal

JDBC Java EE Configuration

No code changes
Just configuration



- How?
  - Use container-managed datasource (JNDI)
  - JTA Transaction Manager
- Purpose
  - Deploy to JEE container

```
<tx:jta-transaction-manager/>

<jee:jndi-lookup id="dataSource" ... />
```

spring Pivotal

@Transactional – Class Level

- Applies to all methods declared by the interface(s)

```
@Transactional
public class RewardNetworkImpl implements RewardNetwork {

  public RewardConfirmation rewardAccountFor(Dining d) {
    // atomic unit-of-work
  }

  public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
    // atomic unit-of-work
  }
}
```

Alternatively *@Transactional* can be declared on the interface instead
– if *not* using Spring Boot nor CGLIB proxies

spring Pivotal

@Transactional
– Class *and* method levels

- Combining class and method levels

default settings

```
@Transactional(timeout=60)
public class RewardNetworkImpl implements RewardNetwork {

  public RewardConfirmation rewardAccountFor(Dining d) {
    // atomic unit-of-work
  }
  @Transactional(timeout=45)
  public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
    // atomic unit-of-work
  }
}
```

override attributes at method level

Topics in this session

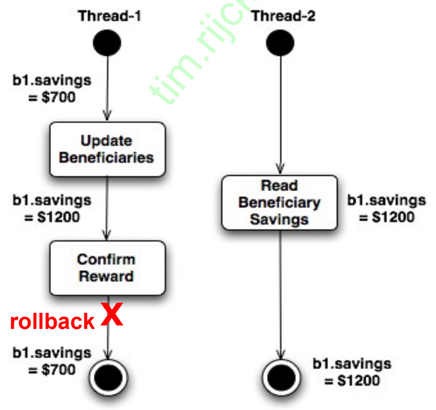
- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- **Isolation Levels**
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

Isolation levels

- 4 isolation levels can be used:
  - READ_UNCOMMITTED
  - READ_COMMITTED
  - REPEATABLE_READ
  - SERIALIZABLE
- Some DBMSs do not support all isolation levels
- Isolation is a complicated subject
  - DBMS all have differences in the way their isolation policies have been implemented
  - We just provide general guidelines

Dirty Reads

Transactions should be isolated – unable to see the results of another uncommitted unit-of-work



READ_UNCOMMITTED

- Lowest isolation level – allows *dirty reads*
- Current transaction can see the results of another uncommitted unit-of-work
- Typically used for large, intrusive read-only transactions
- And/or where the data is constantly changing

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional (isolation=Isolation.READ_UNCOMMITTED)
  public BigDecimal totalRewards(String merchantNumber, int year)
    // Calculate total rewards for a restaurant for a whole year
  }
}
```

spring Pivotal

READ_COMMITTED

- Does not allow dirty reads
  - Only committed information can be accessed
- Default strategy for most databases

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional (isolation=Isolation.READ_COMMITTED)
  public RewardConfirmation rewardAccountFor(Dining dining)
    // atomic unit-of-work
  }
}
```

spring Pivotal

Highest isolation levels

- REPEATABLE_READ
  - Does not allow dirty reads
  - Non-repeatable reads are prevented
    - If a row is read twice in the same transaction, result will always be the same
      - Might result in locking depending on the DBMS
- SERIALIZABLE
  - Prevents non-repeatable reads and dirty-reads
  - Also prevents phantom reads

spring Pivotal

Topics in this session

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Isolation Levels
- **Transaction Propagation**
- Rollback rules
- Testing
- Advanced topics

spring Pivotal

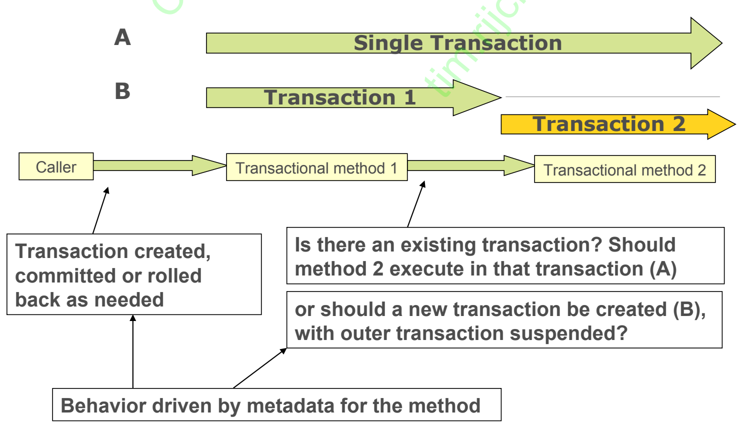
Understanding Transaction Propagation

- Consider the sample below. What should happen if ClientServiceImpl calls AccountServiceImpl?
  - Should everything run into a single transaction?
  - Should each service have its own transaction?

```java
public class ClientServiceImpl
        implements ClientService {
  @Autowired
  private AccountService accountService;

  @Transactional
  public void updateClient(Client c)
  {  // …
    this.accountService.update(c.getAccounts());
  }
}
```

```java
public class AccountServiceImpl
        implements AccountService
{
  @Transactional
  public void update(List <Account> l)
  { // … }
}
```


Understanding Transaction Propagation



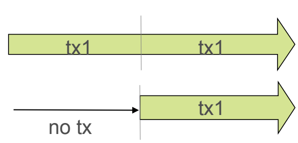
Transaction Propagation with Spring

- 7 levels of propagation
- The following examples show *REQUIRED* and *REQUIRES_NEW*
  - *Check the documentation for other levels*
- Can be used as follows:

```
@Transactional( propagation=Propagation.REQUIRES_NEW )
```


REQUIRED

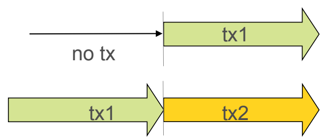
- REQUIRED
  - Default value
  - Execute within a current transaction, create a new one if none exists




```
@Transactional(propagation=Propagation.REQUIRED)
```

REQUIRES_NEW

- REQUIRES_NEW
  - Create a new transaction, suspending the current transaction if one exists



@Transactional(propagation=Propagation.*REQUIRES_NEW*)

Topics in this session

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Isolation Levels
- Transaction Propagation
- **Rollback rules**
- Testing
- Advanced topics

Default Behavior

- By default, a transaction is rolled back if a RuntimeException has been thrown
  - Could be any kind of RuntimeException: DataAccessException, HibernateException etc.

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional
  public RewardConfirmation rewardAccountFor(Dining d) {
    // ...
    throw new RuntimeException();

  }
}
```

Triggers a rollback

spring Pivotal

rollbackFor and noRollbackFor

- Default settings can be overridden with *rollbackFor* and/or *noRollbackFor* attributes

```
public class RewardNetworkImpl implements RewardNetwork {

  @Transactional(rollbackFor=MyCheckedException.class,
                 noRollbackFor={JmxException.class, MailException.class})
  public RewardConfirmation rewardAccountFor(Dining d) throws Exception {
      // ...
  }

}
```

spring Pivotal

Topics in this session

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Isolation Levels
- Transaction Propagation
- Rollback rules
- **Testing**
- Advanced topics

@Transactional within Integration Test

- Annotate test method (or class) with @Transactional
  - Runs test methods in a transaction
  - Transaction will be *rolled back* afterwards
    - No need to clean up your database after testing!

```
@ContextConfiguration(classes=RewardsConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
public class RewardNetworkTest {
    @Test @Transactional
    public void testRewardAccountFor() {
        ...
    }
}
```

This test is now transactional

Controlling Transactional Tests

```
@ContextConfiguration(classes=RewardsConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
@Transactional
public class RewardNetworkTest {

    @Test
    @Commit
    public void testRewardAccountFor() {
        ...    // Whatever happens here will be committed
    }
}
```

Make *all* tests transactional

Commit transaction at end of test

spring Pivotal

Lab

Managing Transactions Declaratively using Spring Annotations

Coming Up: Programmatic transactions, read-only and multiple transactions, Global transactions, Propagation options

spring Pivotal

Topics in this session

- Advanced topics
  1. XML Configuration
  2. Programmatic transactions
  3. Read-only transactions
  4. More on Transactional Tests
  5. Multiple and Global Transactions
  6. Propagation Options

1. Deploying the Transaction Manager

- Declare as a Spring Bean

```xml
<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

A dataSource must be defined elsewhere

Bean id "*transactionManager*" is default name. Can change it but must specify alternative name everywhere – easier not to!

Automatic JTA Implementation Resolution

- For JTA, also possible to use custom XML tag:

```
<tx:jta-transaction-manager/>
```

- Resolves to appropriate implementation for the environment
  - `WebLogicJtaTransactionManager`
  - `WebSphereUowTransactionManager`
  - `JtaTransactionManager`
  - `OC4JJtaTransactionManager`
    - Obsolete, removed after Spring 3.2

@Transactional Configuration Using XML

- Annotate classes and methods with @Transactional in usual way
- Enable using tx namespace in the configuration:

Defines a Bean Post-Processor – proxies @Transactional beans

```
<tx:annotation-driven/>

<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>

<jdbc:embedded-database id="dataSource"> … </jdbc:embedded-database>
```


100% XML-based Spring Transactions

- @Transactional not always an option
  - Someone else may have written the service (without annotations)
  - Legacy code written before @Transactional
- Spring provides support for 100% XML
  - Predates annotations
  - An AOP pointcut declares what to advise
  - Spring's **`tx`** namespace enables a concise definition of transactional advice
  - Can add transactional behavior to any class used as a Spring Bean

Declarative Transactions: XML

AspectJ *named* pointcut expression

```
<aop:config>
  <aop:pointcut id="rewardNetworkMethods"
            expression="execution(* rewards.RewardNetwork.*(..))"/>
  <aop:advisor pointcut-ref="rewardNetworkMethods" advice-ref="txAdvice"/>
</aop:config>
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" timeout="10"/>
    <tx:method name="find*" read-only="true" timeout="10"/>
    <tx:method name="*" timeout="30"/>
  </tx:attributes>
</tx:advice>
<bean id="transactionManager"
   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Method-level configuration for transactional advice

Includes rewardAccountFor(..) and updateConfirmation(..)

2. Programmatic Transactions with Spring

- Declarative transaction management is highly recommended
  - Clean code
  - Flexible configuration
- Spring does enable programmatic transaction
  - Works with local or JTA transaction manager
  - **`TransactionTemplate`** plus callback

> Can be useful inside a technical framework that would not rely on external configuration

Programmatic Transactions: example

Method *not* @Transactional

```
public RewardConfirmation rewardAccountFor(Dining dining) {
  ...
  return  new TransactionTemplate(txManager).execute( (status) -> {
      try {
        ...
        accountRepository.updateBeneficiaries(account);
        confirmation = rewardRepository.confirmReward(contribution, dining);
      }
      catch (RewardException e) {
        status.setRollbackOnly();
        confirmation = new RewardFailure();
      }
      return confirmation;
    }
  );
}
```

Lambda syntax

Method no longer throws exception, using status to perform *manual* rollback

```
public interface TransactionCallback<T> {
   public T doInTransaction(TransactionStatus status)
               throws Exception;
}
```


3. Read-only Transactions – Faster

- Why use transactions if you're only planning to read data?
  - Performance: allows Spring to optimize the transactional resource for read-only data access

```
public void rewardAccount1() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}

@Transactional(readOnly=true)
public void rewardAccount2() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}
```

2 connections

1 single connection

spring Pivotal

Read-only Transactions – Isolation

- Why use transactions if you're only planning to read data?
  - With a high isolation level, a read-only transaction prevents data from being modified until the transaction commits

```
@Transactional(readOnly=true, isolation=Isolation.REPEATABLE_READ)
public void importantAccounts() {
    List accounts = jdbcTemplate.queryForList
                ("SELECT * FROM Accounts WHERE balance > 1000000");
    process(accounts);
    int nAccounts = jdbcTemplate.queryForInt
                ("SELECT count(*) FROM Accounts WHERE balance > 1000000");
    assert accounts.size() == nAccounts;
}
```

spring Pivotal

4. Transactional Tests
@Before vs @BeforeTransaction

```
@ContextConfiguration(locations={"/rewards-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class RewardNetworkTest {

    @BeforeTransaction
    public void verifyInitialDatabaseState() {…}

    @Before
    public void setUpTestDataInTransaction() {…}

    @Test  @Transactional
    public void testRewardAccountFor() { … }
```

Run before transaction is started

Within the transaction

@After and *@AfterTransaction* work in the same way as *@Before* and *@BeforeTransaction*

spring Pivotal.

@Sql and Transaction Control

- Transaction control options
  - *ISOLATED:* Uses *own* txn, a PTM *must* exist
  - *INFERRED:* If PTM exists, txn started using default propagation (so it uses txn around test method) otherwise a DataSource *must* exist (used with *no* txn)
  - *DEFAULT*: Whatever @Sql defines at class level, INFERRED otherwise

```
@Sql(  scripts = "/test-user-data.sql",
       config = @SqlConfig
             ( transactionMode = TransactionMode.ISOLATED,
               transactionManager = "myTxnMgr",
               dataSource= "myDataSource" )
```

Optionally specify bean ids

PTM = PlatformTransactionManager, txn = transaction

spring Pivotal.

5. Multiple Transaction Managers

- @Transactional can declare the id of the transaction manager that should be used

```
@Transactional("myOtherTransactionManager")
public void rewardAccount1() {
  jdbcTemplate.queryForList(…);
  jdbcTemplate.queryForInt(…);
}

@Transactional
public void rewardAccount2() {
  jdbcTemplate.queryForList(…);
  jdbcTemplate.queryForInt(…);
}
```

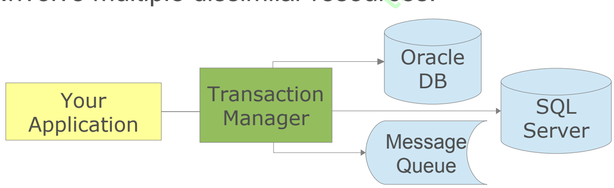
Uses the bean with id "myOtherTransactionManager"

Defaults to use the bean with id "transactionManager"

Important: Separate transaction managers = separate transactions!

Global Transactions

- Also called distributed transactions
- Involve multiple dissimilar resources:



- Global transactions typically require JTA and specific drivers (XA drivers)
  - Two-phase commit protocol

Global Transactions → Spring Integration

- Many possible strategies
  - Spring allows you to switch easily from a non-JTA to a JTA transaction policy
  - Just change the type of the transaction manager
- Reference:
  - *"Distributed transactions with Spring, with and without XA"* by Dr. Dave Syer

http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html

Learn More: ***Enterprise Spring***
– 4 day course, including *global transactions*

6. Propagation Levels and their Behaviors

Propagation Type	If NO current transaction	If there is a current transaction
MANDATORY	throw exception	use current transaction
NEVER	don't create a transaction, run method outside any transaction	throw exception
NOT_SUPPORTED	don't create a transaction, run method outside any transaction	suspend current transaction, run method outside any transaction
SUPPORTS	don't create a transaction, run method outside any transaction	use current transaction
REQUIRED(default)	create a new transaction	use current transaction
REQUIRES_NEW	create a new transaction	suspend current transaction, create a new independent transaction
NESTED	create a new transaction	create a new nested transaction

JPA with Spring and Spring Data

Object Relational Mapping with Spring & Java Persistence API

Using JPA with Spring, Spring Data Repositories

spring Pivotal

Topics in this session

- **Introduction to JPA**
  - **General Concepts**
  - **Mapping**
  - **Querying**
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- Optional and Advanced Topics

spring Pivotal

Introduction to JPA

- The Java Persistence API is designed for operating on domain objects
  – Defined as POJO entities
  – No special interface required
- Replaces previous persistence mechanisms
  – EJB Entity Beans
  – Java Data Objects (JDO)
- A common API for object-relational mapping
  – Derived from the experience of existing products such as JBoss Hibernate and Oracle TopLink

About JPA

- Java Persistence API
  – Current version: 2.1 May 2013
  – 2.2 Under discussion
- Configuration
  – Persistence Unit
- Key Features
  – Entity Manager
  – Entity Manager Factory
  – Persistence Context

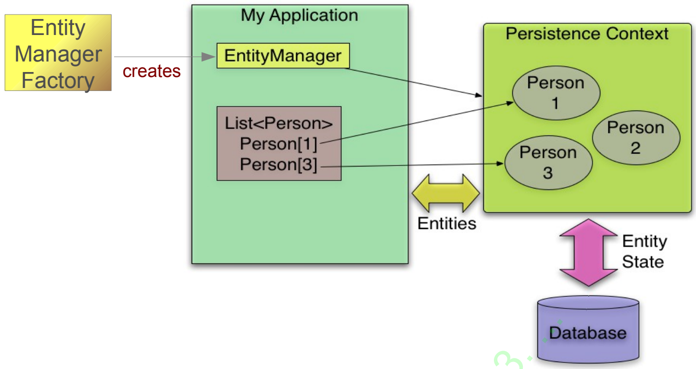
JPA Configuration

- **Persistence Unit**
  - Describes a group of persistent classes (entities)
  - Defines provider(s)
  - Defines transactional types (local vs JTA)
  - Multiple Units per application are allowed
  - Defined by the file: `persistence.xml`

JPA General Concepts

- **EntityManager**
  - Manages a unit of work and persistent objects therein: the *PersistenceContext*
  - Lifecycle often bound to a Transaction (usually container-managed)
- **EntityManagerFactory**
  - thread-safe, shareable object that represents a single data source / persistence unit
  - Provides access to new application-managed EntityManagers

Persistence Context and EntityManager



The EntityManager API

persist(Object o)	Adds the entity to the Persistence Context: *SQL: insert into table …*
remove(Object o)	Removes the entity from the Persistence Context: *SQL: delete from table …*
find(Class entity, Object primaryKey)	Find by primary key: *SQL: select * from table where id = ?*
Query createQuery(String jpqlString)	Create a JPQL query
flush()	Force changed entity state to be written to database immediately

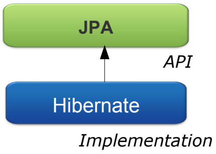
Plus other methods …

JPA Providers

- Several major implementations of JPA spec
  - Hibernate EntityManager
    - Used inside Jboss
  - EclipseLink (RI)
    - Used inside Glassfish
  - Apache OpenJPA
    - Used by Oracle WebLogic and IBM Websphere
  - Data Nucleus
    - Used by Google App Engine
- **Can all be used without application server as well**
  - Independent part of EJB 3 spec

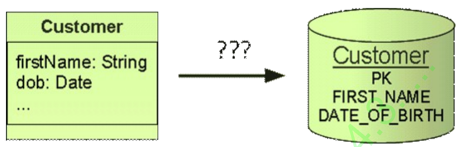
Hibernate JPA

- Hibernate adds JPA support through an additional library
  - The *Hibernate EntityManager*
  - Hibernate *sessions* used behind JPA *interfaces*
  - Custom annotations for Hibernate specific extensions not covered by JPA
    - less important since JPA version 2



JPA Mapping

- JPA requires metadata for mapping classes/fields to database tables/columns
  - Usually provided as annotations
  - XML mappings also supported (orm.xml)
    - Intended for overrides only – not shown here
- JPA metadata relies on defaults
  - No need to provide metadata for the obvious



What can you Annotate?

- Classes
  - Applies to the entire class (such as table properties)
- Fields
  - Typically mapped to a column
  - By default, *all* treated as persistent
    - Mappings will be defaulted
    - Unless annotated with @Transient (non-persistent)
  - Accessed directly via Reflection
- Properties (getters)
  - Also mapped to a column
  - Annotate getters instead of fields

Mapping Using Fields (Data-Members)

```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
   @Id
   @Column(name="cust_id")
   private Long id;

   @Column(name="first_name")
   private String firstName;

  @Transient
   private User currentUser;

   ...
```

Mark as an *entity*

Optionally override *table name*

Mark *id-field* (primary key)

Optionally override *column names*

Not stored in database

Only *@Entity* and *@Id* are mandatory

Data members set *directly*
- *using reflection*
- *"field" access*
- *no setters needed*

spring Pivotal

Mapping Using Accessors (Properties)

Must place @Id on the *getter* method

Other annotations now also placed on *getter* methods

```
@Entity @Table(name= "T_CUSTOMER")
public class Customer {
   private Long id;
   private String firstName;

   @Id
   @Column (name="cust_id")
   public Long getId()
      { return this.id; }

   @Column (name="first_name")
   public String getFirstName()
      { return this.firstName; }

   public void setFirstName(String fn)
      { this.firstName = fn; }
}
```

spring Pivotal

Relationships

- Common relationship mappings supported
  - Single entities and entity collections both supported
  - Associations can be uni- or bi-directional

```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
    @Id
    @Column (name="cust_id")
    private Long id;

    @OneToMany
    @JoinColumn (name="cid")
    private Set<Address> addresses;
    …
```

```
@Entity
@Table(name= "T_ADDRESS")
public class Address {
    @Id private Long id;
    private String street;
    private String suburb;
    private String city;
    private String postcode;
    private String country;
}
```

Foreign key in Address table

spring Pivotal

Embeddables

- Map a table row to multiple classes
  - Address fields also columns in T_CUSTOMER
  - @AttributeOverride overrides mapped column name

```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
    @Id
    @Column (name="cust_id")
    private Long id;

    @Embedded
    @AttributeOverride
        (name="postcode", column=@Column(name="ZIP"))
    private Address office;
    …
```

```
@Embeddable
public class Address {
    private String street;
    private String suburb;
    private String city;
    private String postcode;
    private String country;
}
```

Maps to ZIP column in T_CUSTOMER

spring Pivotal

JPA Querying

- JPA provides several options for accessing data
  - Retrieve an object by primary key
  - Query for objects using JPA Query Language (JPQL)
    - Similar to SQL and HQL
  - Query for objects using Criteria Queries (appendix)
    - API for creating ad hoc queries
    - Only in JPA 2
  - Execute SQL directly to underlying database (appendix)
    - “Native” queries, allow DBMS-specific SQL to be used
    - Consider JdbcTemplate instead when not using managed objects – more options/control, more efficient

JPA Querying: By Primary Key

- To retrieve an object by its database identifier simply call *find()* on the EntityManager

```
Long customerId = 123L;
Customer customer = entityManager.find(Customer.class, customerId);
```

returns **null** if no object exists for the identifier

No cast required – JPA uses generics

JPA Querying: JPQL

- SELECT clause required
- can't use *

- Query for objects based on properties or associations ...

```
// Query with named parameters
TypedQuery<Customer> query = entityManager.createQuery(
        "select c from Customer c where c.address.city = :city", Customer.class);
query.setParameter("city", "Chicago");
List<Customer> customers = query.getResultList();

// … or using a single statement
List<Customer> customers2 = entityManager.
       createQuery("select c from Customer c ...", Customer.class).
       setParameter("city", "Chicago").getResultList();

// … or if expecting a single result
Customer customer = query.getSingleResult();
```

Specify Class to Populate / return

Can also use bind ? Variables – indexed from 1 *like JDBC*

spring Pivotal

Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- **Configuring JPA in Spring**
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- Optional and Advanced Topics

spring Pivotal

Quick Start – Spring JPA Configuration

Steps to using JPA with Spring

1. Define an EntityManagerFactory bean.
2. Define a DataSource bean
3. Define a Transaction Manager bean
4. Define Mapping Metadata (already covered)
5. Define DAOs

Note: There are many configuration options for EntityManagerFactory, persistence.xml, and DataSource. See the optional section for details.

spring Pivotal

Define the EntityManagerFactory

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setShowSql(true);
    adapter.setGenerateDdl(true);
    adapter.setDatabase(Database.HSQL);

    Properties props = new Properties();
    props.setProperty("hibernate.format_sql", "true");

    LocalContainerEntityManagerFactoryBean emfb =
            new LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource);
    emfb.setPackagesToScan("rewards.internal");
    emfb.setJpaProperties(props);
    emfb.setJpaVendorAdapter(adapter);

    return emfb;
}
```

NOTE: no persistence.xml needed when using *packagesToScan* property

spring Pivotal

Configuration – XML Equivalent

```
<bean id="entityManagerFactory"
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="packagesToScan" value="rewards.internal"/>

 <property name="jpaVendorAdapter">
   <bean class="org.sfwk.orm.jpa.vendor.HibernateJpaVendorAdapter">
     <property name="showSql" value="true"/>
     <property name="generateDdl" value="true"/>
     <property name="database" value="HSQL"/>
   </bean>
 </property>

 <property name="jpaProperties">
    <props> <prop key="hibernate.format_sql">true</prop> </props>
 </property>
</bean>
```

spring Pivotal

Define DataSource & Transaction Manager

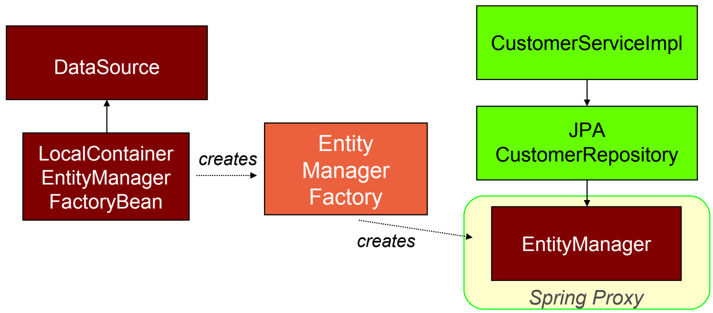
```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emfb =
         new LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource());
    ...
    return emfb;
}

@Bean
public PlatformTransactionManager
                   transactionManager(EntityManagerFactory emf) {
   return new JpaTransactionManager(emf);
}

@Bean
public DataSource dataSource() {  // Lookup via JNDI or create locally.  }
```

Method returns a *FactoryBean*...

...Spring calls getObject() on the FactoryBean to obtain the *EntityManagerFactory*:

spring Pivotal

EntityManagerFactoryBean Configuration



Proxy automatically finds entity-manager for current transaction

Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- Configuring JPA in Spring
- **Implementing JPA DAOs**
- Spring Data – JPA
- Lab
- Optional and Advanced Topics

Implementing JPA DAOs

- JPA provides configuration options so Spring can manage transactions via the EntityManager
- There are no Spring dependencies in your DAO implementations

spring Pivotal

Spring-Managed Transactions & EntityManager (1)

- To transparently participate in Spring-driven transactions
  - Define a transaction manager
    - Either `JpaTransactionManager`
    - Or `JtaTransactionManager`
  - Use FactoryBean for building the *EntityManagerFactory*
  - Inject *EntityManager* “proxy” with @PersistenceContext
    - JPA's equivalent to *@Autowired*
    - At runtime the proxy resolves to current *EntityManager* for current transaction in current thread

spring Pivotal

Spring-Managed Transactions & EntityManager (2)

- The code – no Spring dependencies

```
public class JpaCustomerRepository implements CustomerRepository {
    private EntityManager entityManager;

    @PersistenceContext
    public void setEntityManager (EntityManager entityManager) {
        this. entityManager = entityManager;
    }

    public Customer findById(long orderId) {
        return entityManager.find(Customer.class, orderId);
    }
}
```

Automatic injection of EM Proxy

Proxy resolves to EM when used

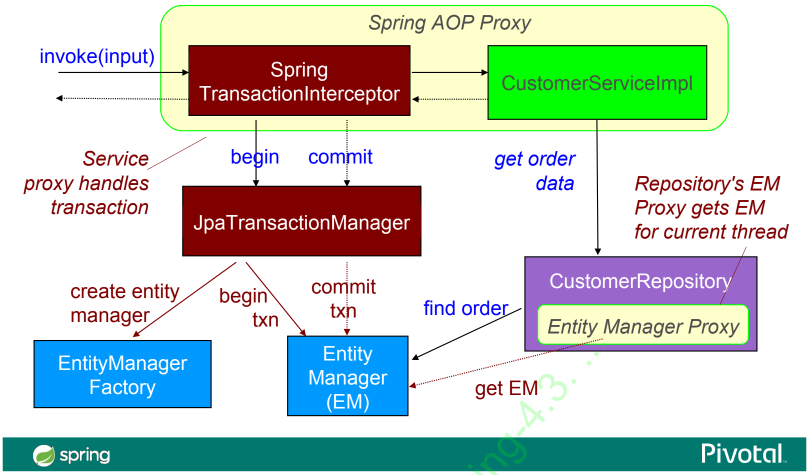
Spring-managed Transactions and EntityManager (3)

- The Configuration

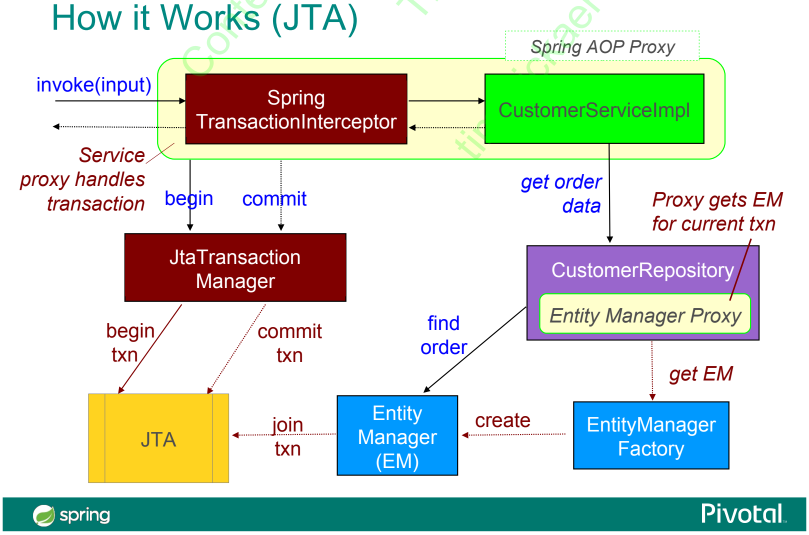
```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    ...
}

@Bean
public CustomerRepository jpaCustomerRepository() {
    return new JpaCustomerRepository();
}

@Bean
public PlatformTransactionManager
        transactionManager(EntityManagerFactory emf) throws Exception {
    return new JpaTransactionManager(emf);
}
```

Automatic injection of entity-manager proxy

How it Works (JPA)



How it Works (JTA)

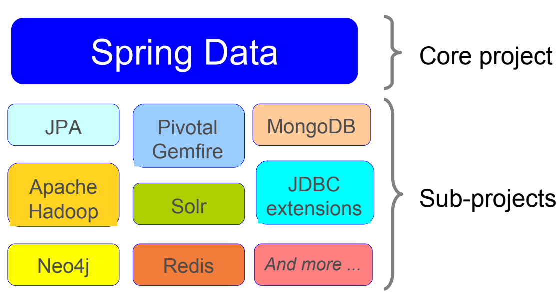


Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- Configuring JPA in Spring
- Implementing JPA DAOs
- **Spring Data – JPA**
- Lab
- Optional and Advanced Topics

What is Spring Data?

- Reduces boiler plate code for data access
  - Works in many environments



Instant Repositories

- How?
  - **Step 1:** Annotate domain class
    - define keys & enable persistence
  - **Step 2:** Define your repository as an *interface*
- Spring will implement it at run-time
  - Scans for interfaces extending Spring's `Repository<T, K>`
  - CRUD methods auto-generated
  - Paging, custom queries and sorting supported
  - Variations exist for most Spring Data sub-projects

spring Pivotal

Step 1: Annotate Domain Class

Here we are using JPA

- Annotate JPA Domain object as normal
  - Nothing to see here!

Domain Class

```
@Entity
@Table(...)
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private Date oderDate;
  private String email;

  // Other data-members and getters and setters omitted
}
```

Note: Key is a *Long*

spring Pivotal

Domain Objects: Other Data Stores

- Spring Data provides similar annotations to JPA
  - *@Document*, *@Region*, *@NodeEntity* ...
- Templates (like *JdbcTemplate*) for basic CRUD access
  - MongoTemplate, GemfireTemplate, RedisTemplate ...

MongoDB – map to a JSON document

```
@Document
public class Account {
  ...
```

```
@NodeEntity
public class Account {
  @GraphId
  Long id;
  ...
```

Neo4J – map to a graph

Gemfire – map to a region

```
@Region
public class Account {
  ...
```

spring Pivotal

Step 2: Define a Repository Interface

Must extend Repository<T, ID>

```
public interface Repository<T, ID> { }
```

Marker interface – add any methods from CrudRepository and/or add custom finders

```
public interface CrudRepository<T, ID
    extends Serializable> extends Repository<T, ID> {

  public <S extends T> save(S entity);
  public <S extends T> Iterable<S> save(Iterable<S> entities);

  public T findOne(ID id);
  public Iterable<T> findAll();

  public void delete(ID id);
  public void delete(T entity);
  public void deleteAll();
}
```

You get all these methods automatically

```
PagingAndSortingRepository<T, K>
- adds Iterable<T> findAll(Sort)
- adds Page<T> findAll(Pageable)
```

spring Pivotal

Generating Repositories

- Spring scans for Repository interfaces
  - Implements them and creates as a Spring bean
- Java Configuration

```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.**.repository")
@EnableMongoRepositories(...)
public class MyConfig { ... }
```

- XML

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

Defining a JPA Repository

- Auto-generated finders obey naming convention
  - find(First)By*<DataMember><Op>*
  - <Op> can be GreaterThan, NotEquals, Between, Like …

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

  public Customer findFirstByEmail(String someEmail);    // No <Op> for Equals
  public List<Customer> findByOrderDateLessThan(Date someDate);
  public List<Customer> findByOrderDateBetween(Date d1, Date d2);

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

id

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

Convention over Configuration

Extend **Repository** and build your own interface using conventions.

- Note: Repository is an *interface* (*not a class!*)

```
import org.springframework.data.repository.Repository;
import org.springframework.data.jpa.repository.Query;

public interface CustomerRepository extends Repository<Customer, Long> {

   <S extends Customer> save(S entity);   // Definition as per CrudRepository
   Customer findOne(long i);               // Definition as per CrudRepository

   Customer findFirstByEmailIgnoreCase(String email);  // Case insensitive search

   @Query("select u from Customer u where u.emailAddress = ?1")
   Customer findByEmail(String email);      // ?1 replaced by method param
}
```


Internal Behavior – Another Spring Proxy

- Before startup
- After startup


```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig { … }
```

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

Accessing the Repository

- Use Spring to inject *CustomerRepository* dependency

```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig {

   @Autowired
   public CustomerRepository customerRepository;

   @Bean
   public CustomerService customerService() {
      return new CustomerService( customerRepository );
   }
}
```

spring Pivotal

Summary

- Use 100% JPA to define entities and access data
  - Repositories have no Spring dependency
  - Spring Data Repositories need no code!
- Use Spring to configure JPA entity-manager factory
  - Smart proxy works with Spring-driven transactions
  - Optional translation to DataAccessExceptions (see advanced section)

spring Pivotal

Lab

Reimplementing Repositories using Spring and JPA

Coming Up: Optional topics on JPA queries, connection factories, DataAccessExceptions, custom Spring Data repositories

spring Pivotal

Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml
  - Exception Translation
  - Customized Spring Data Repositories

spring Pivotal

JPA Querying: Typed Queries

- Criteria Query API (JPA 2)
  - Build type safe queries: fewer run-time errors
  - Much more verbose

```
public List<Customer> findByLastName(String lastName) {
   CriteriaBuilder builder = entityManager.getCriteriaBuilder();
   CriteriaQuery<Customer> cq = builder.createQuery(Customer.class);
   Predicate condition =
        builder.equal( cq.from(Customer.class).get(Customer_.name), lastName);
   cq.where(condition);

   return entityManager.createQuery(cq).getResultList();
}
```

Meta-data class created by JPA (note underscore)

JPA Querying: SQL

- Use a *native* query to execute raw SQL

```
// Query for multiple rows
Query query = entityManager.createNativeQuery(
      "SELECT cust_num FROM T_CUSTOMER c WHERE cust_name LIKE ?");
query.setParameter(1, "%ACME%");
List<String> customerNumbers = query.getResultList();

// … or if expecting a single result
String customerNumber = (String) query.getSingleResult();

// Query for multiple columns
Query query = entityManager.createNativeQuery(
         "SELECT ... FROM T_CUSTOMER c WHERE ...", Customer.class);
List<Customer> customers = query.getResultList();
```

No *named* parameter support

Indexed from 1 – *like JDBC*

Specify Class to Populate / return

Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - JPA Typed Queries / Native Queries
  - **EntityManagerFactoryBean alternatives / persistence.xml**
  - Exception Translation
  - Customized Spring Data Repositories

Setting up an EntityManagerFactory

- Three ways to set up an EntityManagerFactory:
  - LocalEntityManagerFactoryBean
  - LocalContainerEntityManagerFactoryBean
  - Use a JNDI lookup
- persistence.xml required for configuration
  - From version 3.1, Spring allows no *persistence.xml* with LocalContainerEntityManagerFactoryBean

persistence.xml <?xml?>

- Always stored in META-INF
- Specifies "persistence unit":
  - optional vendor-dependent information
  - DB Connection properties often specified here.

```xml
<persistence version="1.0"
      xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
   <persistence-unit name="rewardNetwork"/>
   ...
</persistence>
```

- File is <u>required</u> in JPA, but <u>optional</u> when using Spring with JPA!

spring Pivotal

LocalEntityManagerFactoryBean

- Useful for standalone apps, integration tests
- Cannot specify a DataSource
  - Useful when only data access is via JPA
  - Uses standard JPA service location (SPI) mechanism
    /META-INF/services/javax.persistence.spi.PersistenceProvider

```java
@Bean
public LocalEntityManagerFactoryBean entityManagerFactory() {
   LocalEntityManagerFactoryBean em =
         new LocalEntityManagerFactoryBean();
   em.setPersistenceUnitName("rewardNetwork");
   return em;
}
```

spring Pivotal

LocalContainer EntityManagerFactoryBean

- Provides full JPA capabilities
- Integrates with existing DataSources
- Useful when fine-grained customization needed
  - Can specify vendor-specific configuration

We saw this earlier using 100% Spring configuration In both XML and Java

Configuration – Spring *and* Persistence Unit

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emfb =
        new LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource);
    emfb.setPersistenceUnitName("rewardNetwork");
    return emfb;
}
```

Minimal Spring config

```
<persistence-unit name="rewardNetwork">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
</persistence-unit>
```

Do JPA config in *persistence.xml*

If using JTA – declare <jta-data-source> in the persistence-unit

JNDI Lookups

- A jee:jndi-lookup can be used to retrieve *EntityManagerFactory* from application server
- Useful when deploying to JEE Application Servers (WebSphere, WebLogic, etc.)

```
@Bean
public EntityManagerFactory entityManagerFactory() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("persistence/rewardNetwork");
}
```

OR

```
<jee:jndi-lookup id="entityManagerFactory"
                 jndi-name="persistence/rewardNetwork"/>
```

spring Pivotal

Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml
  - **Exception Translation**
  - Customized Spring Data Repositories

spring Pivotal

Transparent Exception Translation (1)

- Used as-is, the DAO implementations described earlier will throw unchecked JPA PersistenceExceptions
  - Not desirable to let these propagate up to the service layer or other users of the DAOs
  - Introduces dependency on the specific persistence solution that should not exist
- AOP allows translation to Spring's rich, vendor-neutral DataAccessException hierarchy
  - Hides access technology used

Transparent Exception Translation (2)

- Spring provides this capability out of the box
  - Annotate with @Repository
  - Define a Spring-provided BeanPostProcessor

```
@Repository
public class JpaCustomerRepository implements CustomerRepository {
  ...
}
```

```
<bean class="org.springframework.dao.annotation.
                    PersistenceExceptionTranslationPostProcessor"/>
```

Transparent Exception Translation (3)

- Or use XML configuration:

```
public class JpaCustomerRepository implements CustomerRepository {
   ...
}
```

No annotations

```
<bean id="persistenceExceptionInterceptor"
         class="org.springframework.dao.support.
                   PersistenceExceptionTranslationInterceptor"/>

<aop:config>
   <aop:advisor pointcut="execution(* *..CustomerRepository+.*(..))"
                    advice-ref="persistenceExceptionInterceptor" />
</aop:config>
```

spring Pivotal

Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  – JPA Typed Queries / Native Queries
  – EntityManagerFactoryBean alternatives / persistence.xml
  – Exception Translation
  – **Customized Spring Data Repositories**

spring Pivotal

JPA Specific Interface

- Adds EntityManager specific options

```
public interface  JpaRepository<T, ID extends Serializable>
        extends PagingAndSortingRepository<T, ID> {

   <S extends T> S saveAndFlush(S entity);
   void flush();

   // Implemented as a single DELETE
   void deleteInBatch(Iterable<T> entities);
   void deleteAllInBatch();

   // Returns a lazy-loading proxy, using JPA's EntityManager.getReference()
   //  – equivalent to Hibernate's Session.load()
   T getOne(ID id);
}
```

spring Pivotal

Adding Custom Behavior (1)

- Not all use cases satisfied by automated methods
  - Enrich with custom repositories: *mix-ins*
- **Step 1:** Create normal interface and implementation

```
public class CustomerRepositoryImpl implements CustomerRepositoryCustom {
   Customer  findDeadbeatCustomers() {
      //  Your custom implementation to find unreliable
      //  and bad-debt customers
   }
}
```

```
public interface CustomerRepositoryCustom {
   Customer  findDeadbeatCustomers();
}
```

spring Pivotal

Adding Custom Behavior (2)

- **Step 2**: Combine with an automatic repository:

```
public interface CustomerRepository
      extends CrudRepository<Account, Long>, CustomerRepositoryCustom {
}
```

- Spring Data looks for implementation beans
  - ID = repository interface + “Impl” (configurable)
  - In this example: “*CustomerRepositoryImpl*”

- Result: *CustomerRepository* bean contains automatic and custom methods!

Topics Covered

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Advanced
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives and persistence.xml
  - Exception Translation
  - Customized Spring Data Repositories

Overview of Spring Web

Developing Modern Web Applications

Servlet Configuration, Product Overview

spring Pivotal

Topics in this Session

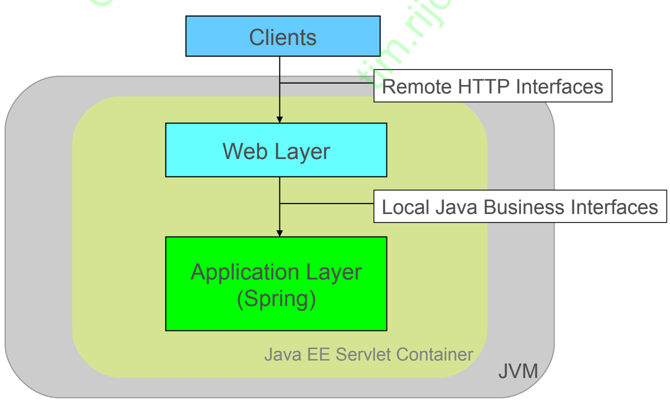
- **Introduction**
- Using Spring in Web Applications
- Overview of Spring Web
- Spring and other Web frameworks

spring Pivotal

Web Layer Integration

- Spring provides support in the Web layer
  - Spring MVC, Spring WebFlow...
- However, you are free to use Spring with any Java web framework
  - Integration might be provided by Spring or by the other framework itself
  - Spring also integrates with many of the common REST frameworks

Effective Web Application Architecture



Topics in this Session

- Introduction
- **Using Spring in Web Applications**
- Overview of Spring Web
- Spring and other Web frameworks

spring Pivotal

Spring Application Context
Lifecycle in Web Applications

- Spring can be initialized within a webapp
  - start up business services, repositories, etc.
- Uses a standard servlet listener
  - initialization occurs before any servlets execute
  - application ready for user requests
  - ApplicationContext.close() is called when the application is stopped

spring Pivotal

Configuration via WebApplicationInitializer

```java
public class MyWebAppInitializer
   extends AbstractContextLoaderInitializer {

@Override
protected WebApplicationContext createRootApplicationContext() {

     // Create the 'root' Spring application context
     AnnotationConfigWebApplicationContext rootContext =
       new AnnotationConfigWebApplicationContext();

     rootContext.getEnvironment().setActiveProfiles("jpa");  // optional
     rootContext.register(RootConfig.class);
     return rootContext;
}
...
```

Implements *WebApplicationInitializer* Automatically detected by servlet container.

Multiple classes may be listed

Available in Servlet 3.0+ Environments, no more web.xml!

spring Pivotal

Configuration via web.xml

- Only option prior to servlet 3.0
  - Just add a Spring-provided servlet listener

web.xml

```xml
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>
      /WEB-INF/merchant-reporting-webapp-config.xml
   </param-value>
</context-param>

<listener>
   <listener-class>
      org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

The application context's configuration file(s)

Loads the ApplicationContext into the ServletContext *before* any Servlets are initialized

spring Pivotal

web.xml Configuration Options

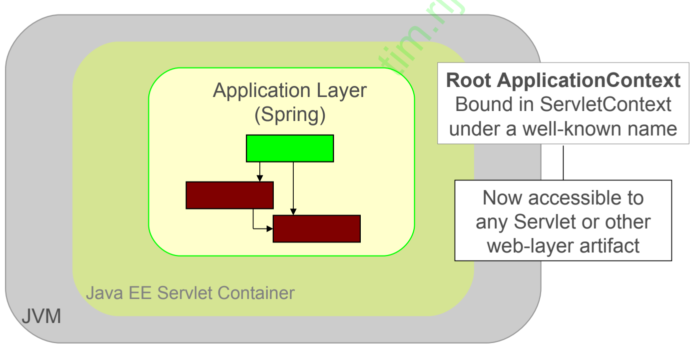
- Default resource location is document-root
  - Can use *classpath:* designator
  - Defaults to WEB-INF/applicationContext.xml

web.xml

```xml
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:/rewards/internal/application-config.xml
    /WEB-INF/merchant-reporting-webapp-config.xml
  </param-value>
</context-param>
<context-param>
  <param-name>spring.profiles.active</param-name>
  <param-value>jpa</param-value>
</context-param>
```

Optionally specify profile(s) to use

Servlet Container After Starting Up



Instantiating Servlets

- Override *onStartup()* method to define servlets
  - Warning: Can't access Spring beans, not available yet

```
public class MyWebAppInitializer extends AbstractContextLoaderInitializer {

  protected WebApplicationContext createRootApplicationContext() {
     // ...Same configuration as previous slide...
  }

  public void onStartup(ServletContext container) {
     super.onStartup(container);
     // Register and map a servlet
     ServletRegistration.Dynamic svlt =
          container.addServlet("myServlet", new TopSpendersReportGenerator());
     svlt.setLoadOnStartup(1);
     svlt.addMapping("/");
  }
```

calls

No beans are loaded yet at this point in the lifecycle...

Dependency Injection of Servlets

- Suitable for *web.xml* or *AbstractContextLoaderInitializer*
- Use *WebApplicationContextUtils*
  - gets Spring *ApplicationContext* from *ServletContext*

```
public class TopSpendersReportGenerator extends HttpServlet {
   private ClientService clientService;

   public void init() {
      ApplicationContext context = WebApplicationContextUtils.
            getRequiredWebApplicationContext(getServletContext());
      clientService  = (ClientService) context.getBean("clientService");
   }
   …
}
```

Spring MVC Supports Dependency Injection

- Example using Spring MVC

```
@Controller
public class TopSpendersReportController {
   private ClientService clientService;

   @Autowired
   public TopSpendersReportController(ClientService service) {
      this.clientService = service;
   }

   …
}
```

Dependency is automatically injected by type

No need for *WebApplicationContextUtils* anymore

Topics in this Session

- Introduction
- Using Spring in Web Applications
- **Overview of Spring Web**
- Spring and other Web frameworks

Spring Web

- Spring MVC
  - Web framework bundled with Spring
- Spring WebFlow
  - Plugs into Spring MVC
  - Implements navigation flows
- Spring Mobile
  - Routing between mobile / non-mobile versions of site
- Spring Social
  - Easy integration with Facebook, Twitter, etc.

spring Pivotal

Spring Web MVC

- Spring's web framework
  - Uses Spring for its own configuration
  - Controllers are Spring beans
  - testable artifacts
- Annotation-based model since Spring 2.5
- Builds on the Java Servlet API
- The core platform for developing web applications with Spring
  - All higher-level modules such as WebFlow build on it

spring Pivotal

Spring Web Flow

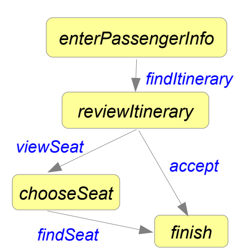
- Plugs into Spring Web MVC as a Controller technology for implementing stateful "flows"
  - Checks that users follow the right navigation path
  - Manages back button and multiple windows issues
  - Provides scopes beyond request and session
    - such as the *flow* and *flash* scope
  - Addresses the double-submit problem elegantly

Example Flow Definition

Online Check-in

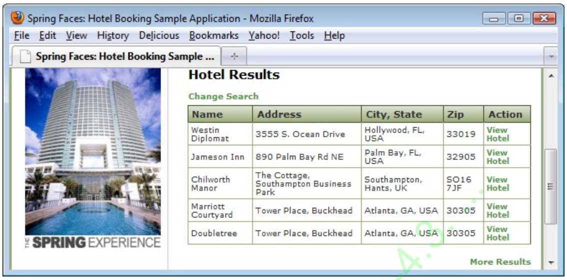
- Flows are declared in Xml

```
<flow ...>
   <view-state id="enterPassengerInfo">
      <transition on="findItinerary” to=”reviewItinerary" />
   </view-state>

   <view-state id="reviewItinerary">
       <transition on="viewSeat" to="chooseSeat" />
       <transition on="accept"     to="finish" />
   </view-state>

   <view-state id="chooseSeat">
       <transition on="findSeat" to="finish" />
   </view-state>

   <end-state id="finish"/>
</flow>
```



More about WebFlow

- Online sample application is available here:
  http://richweb.springsource.org/swf-booking-faces/spring/intro
- Sample applications can be downloaded here:
  http://projects.spring.io/spring-webflow/



spring Pivotal

Topics in this Session

- Introduction
- Using Spring in Web Applications
- Overview of Spring Web
- **Spring and other Web frameworks**

spring Pivotal

Spring – Struts 2 Integration
Java Configuration

```
@Action("/doLogin")
@ResultPath("/WEB-INF/views")
@Result(name = "success", location = "LoginSuccess.jsp"),
@Result(name = "error", location = "LoginError.jsp")
public class DoLoginAction extends ActionSupport {
    @Autowired
    private UserDAO userDAO;
    private User user;
    ...

    public String execute() {
        if (userDAO.checkLogin(user)) { return SUCCESS; }

        return ERROR;
    }
}
```

Inject Spring Beans in
The normal way

Spring – Struts 2 Integration
XML Configuration

- Integration plug-in provided by Struts 2 framework

```
<struts>
    <!-- Define Spring as the object factory -->
    <constant name="struts.objectFactory" value="spring" />
     ...
</struts>
```

```
<struts>
    <include file="struts-default.xml"/>

    <package name="secure" namespace="/secure" extends="default">
        <action name="example" class="myBean">
            <result>example.ftl</result>
        </action>
    </package>
</struts>
```

class actually specifies
a Spring bean name

Spring – JSF Integration

- Two options
  - Spring-centric integration
    - Provided by Spring Faces sub-project of Spring Web Flow
  - JSF-centric integration
    - Spring plugs in as as the JSF managed bean provider

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.springsource.web.ClientController</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

JSF-centric integration

Integration with Other Frameworks

- Wicket
  - Comes with an integration to Spring
    https://cwiki.apache.org/confluence/display/WICKET/Spring

- Tapestry 5
  - Provides a dedicated integration module for Spring
    https://tapestry.apache.org/integrating-with-spring-framework.html

Summary

- Spring can be used with any web framework
  - Spring provides the ContextLoaderListener that can be declared in web.xml
- Spring MVC is a lightweight web framework where controllers are Spring beans
  - More about Spring MVC in the next module
- WebFlow plugs into Spring MVC as a Controller technology for implementing stateful "flows"

Spring Web MVC Essentials

Getting Started With Spring MVC

Implementing a Simple Controller

spring Pivotal

What is Spring MVC?

- Web framework based on the Model/View/Controller pattern
  - Alternative to JSF, Struts 1, Struts 2 (WebWork), Tapestry, Wicket …
- Based on Spring principles
  - POJO programming
  - Testable components
  - Uses Spring for configuration
- Supports a wide range of view technologies
  - JSP, XSLT, PDF, Excel, Velocity, Freemarker, Thymeleaf, etc.

spring Pivotal

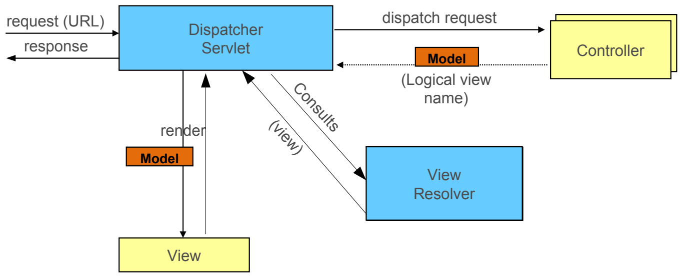
Topics in this Session

- **Request Processing Lifecycle**
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - Views
- Quick Start

Web Request Handling Overview

- Web request handling is rather simple
  - Based on an incoming URL...
  - ...we need to call a method...
  - ...after which the return value (if any)...
  - ...needs to be rendered using a view

Request Processing Lifecycle



spring Pivotal

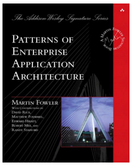
Topics in this Session

- Request Processing Lifecycle
- **Key Artifacts**
  - DispatcherServlet
  - Controllers
  - Views
- Quick Start

spring Pivotal

DispatcherServlet: The Heart of Spring Web MVC

- A “front controller”
  - coordinates all request handling activities
  - analogous to Struts ActionServlet / JSF FacesServlet
- Delegates to Web infrastructure beans
- Invokes user Web components
- Fully customizable
  - interfaces for all infrastructure beans
  - many extension points



DispatcherServlet Configuration

- Defined by **`WebApplicationInitializer`** or **`web.xml`**
- Uses Spring for its configuration
  - programming to interfaces + dependency injection
  - easy to swap parts in and out
- Creates separate “servlet” application context
  - configuration is private to DispatcherServlet
- Full access to the parent “root” context
  - instantiated via ContextLoaderListener
    - shared across servlets

Dispatcher Servlet
Java Configuration Example

Servlet 3.0+

```java
public class MyWebInitializer
      extends AbstractAnnotationConfigDispatcherServletInitializer  {

  // Tell Spring what to use for the Root context:
  @Override protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[]{ RootConfig.class };
  }

  // Tell Spring what to use for the DispatcherServlet context:
  @Override protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{ MvcConfig.class };
  }

  // DispatcherServlet mapping:
  @Override protected String[] getServletMappings() {
    return new String[]{"/rewardsadmin/*"};
  }
```

"Root" configuration

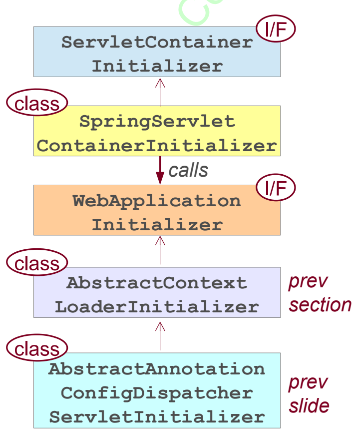
MVC configuration

Beans defined in MVC Context have access to beans defined in Root Context.

spring Pivotal

About Web Initializer Classes

Key ↑ inherit ↓ invoke



- Interface from *Servlet 3 specification*, implement to initialize servlet system
- Spring's implementation which, in turn, delegates to one or more ...
- Base-class for all Spring MVC apps to implement for servlet configuration *without* `web.xml`
- Sets up a `ContextLoaderListener`, you provide *root* `ApplicationContext`
- Also defines a `DispatcherServlet`, assumes Java Config. You provide *root* and *servlet* Java config classes

spring Pivotal

Dispatcher Servlet

XML Configuration Example

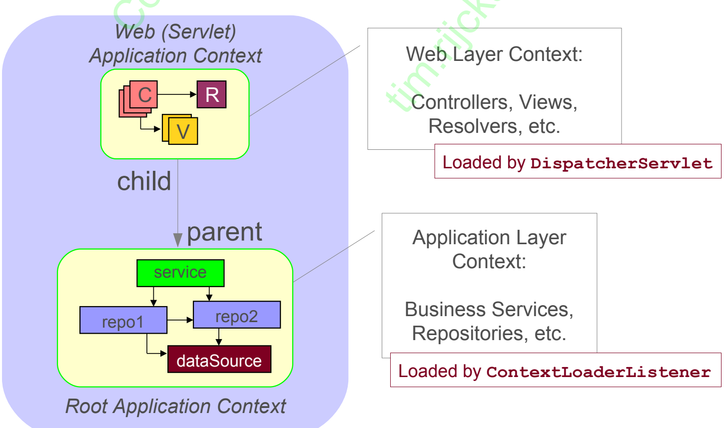
```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/web-config.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/rewardsadmin/*</url-pattern>
</servlet-mapping>
```

web.xml

Pre-Servlet 3.0

Beans defined in web context have access to beans defined in RootApplicationContext

Servlet Container After Starting Up



Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - **Controllers**
  - Views
- Quick Start

Controller Implementation

- Annotate controllers with `@Controller`
  - `@RequestMapping` tells Spring what method to execute when processing a particular request

```
@Controller
public class AccountController {

   @RequestMapping("/listAccounts")
   public String list(Model model) {...}
}
```

Example of calling URL:
http://localhost:8080 / mvc-1 / rewardsadmin / listAccounts

application server | webapp | servlet mapping | request mapping

URL-Based Mapping Rules

- Mapping rules typically URL-based, optionally using wild cards:
  - /accounts
  - /accounts/edit
  - /editAccount
  - /listAccounts.htm
  - /accounts/*

Suffixes *ignored* by default

Controller Method Parameters

- Extremely flexible!
- You pick the parameters you need, Spring provides them
  - HttpServletRequest, HttpSession, Principal …
  - Model for sending data to the view.
  - See Spring Reference, Handler Methods

```
@Controller
public class AccountController {

  @RequestMapping("/listAccounts")
  public String list(Model model) {
    ...
  }
}
```

View name

Model holds data for view

Extracting Request Parameters

- Use @RequestParam annotation
  - Extracts parameter from the request
  - Performs type conversion

```
@Controller
public class AccountController {

   @RequestMapping("/showAccount")
   public String show(@RequestParam("entityId") long id,
                              Model model) {
      ...
   }
```

Example of calling URL:
http://localhost:8080/mvc-1/rewardsadmin/showAccount.htm?entityId=123

spring Pivotal

URI Templates

- Values can be extracted from request URLs
  - *Based on URI Templates*
  - not Spring-specific concept, used in many frameworks
  - Use {…} placeholders and @PathVariable
- Allows clean URLs without request parameters

```
@Controller
public class AccountController {

   @RequestMapping("/accounts/{accountId}")
   public String show(@PathVariable("accountId") long id,
                              Model model) {
      ...
   }
   ...
```

Example of calling URL:
http://localhost:8080/mvc-1/rewardsadmin/accounts/123

spring Pivotal

Method Signature Examples

Example URLs

http://.../accounts

```
@RequestMapping("/accounts")
public String show(HttpServletRequest request, Model model)
```

http://.../orders/1234/items/2

```
@RequestMapping("/orders/{id}/items/{itemId}")
public String show(@PathVariable("id") Long id,
                   @PathVariable int itemId,
                   Model model, Locale locale,
                   @RequestHeader("user-agent")) String agent )
```

http://.../orders?id=1234&itemId=2

```
@RequestMapping("/orders")
public String show(@RequestParam Long id,
                   @RequestParam("itemId") int itemId,
                    Principal user, Map<String,Object> model,
                    HttpSession session )
```

View name

spring Pivotal

Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - **Views**
- Quick Start

spring Pivotal

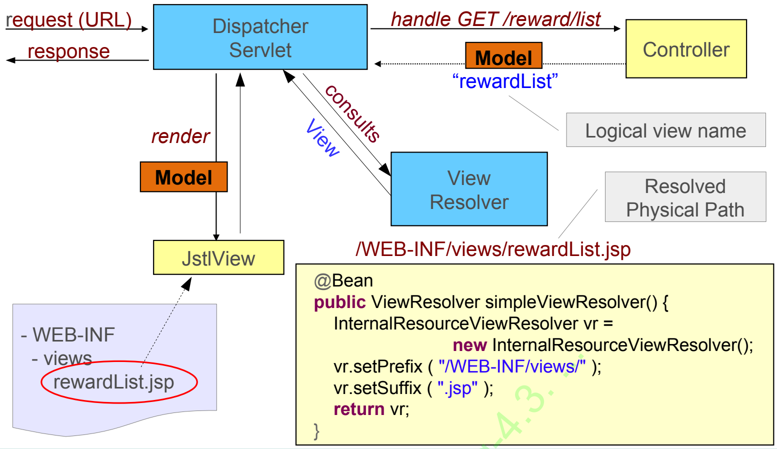
Views

- A View renders web output.
  - Many built-in views available for JSPs, XSLT, templating approaches (Velocity, FreeMarker), etc.
  - View support classes for creating PDFs, Excel spreadsheets, etc.
- Controllers typically return a 'logical view name' String.
- ViewResolvers select View based on view name.

View Resolvers

- The DispatcherServlet delegates to a ViewResolver to obtain View implementation based on view name.
- The default ViewResolver treats the view name as a Web Application-relative file path
  - i.e. a JSP: `/WEB-INF/reward/list.jsp`
- Override this default by registering a ViewResolver bean with the DispatcherServlet
  - We will use InternalResourceViewResolver
  - Several other options available.

Internal Resource View Resolver Example




```
@Bean
public ViewResolver simpleViewResolver() {
    InternalResourceViewResolver vr =
                    new InternalResourceViewResolver();
    vr.setPrefix ( "/WEB-INF/views/" );
    vr.setSuffix ( ".jsp" );
    return vr;
}
```


Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - Views
- **Quick Start**

Quick Start

Steps to developing a Spring MVC application

1. Deploy a Dispatcher Servlet (one-time only)
2. Implement a controller
3. Register the Controller with the DispatcherServlet
4. Implement the View(s)
5. Register a ViewResolver (optional, one-time only)
6. Deploy and test

Repeat steps 2-6 to develop new functionality

1a. Deploy DispatcherServlet

```java
public class WebInitializer
      extends AbstractAnnotationConfigDispatcherServletInitializer  {

  // Root context:
  @Override protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[]{ RootConfig.class };
  }

  // DispatcherServlet context:
  @Override protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{ MvcConfig.class };
  }

  // DispatcherServlet mapping:
  @Override protected String[] getServletMappings() {
    return new String[]{"/rewardsadmin/*"};
  }
```

Services, Repositories ...

Contains Spring MVC configuration

1b. Deploy DispatcherServlet

- Can handle URLs like …

```
http://localhost:8080/mvc-1/rewardsadmin/reward/list
http://localhost:8080/mvc-1/rewardsadmin/reward/new
http://localhost:8080/mvc-1/rewardsadmin/reward/show?id=1
```

  – We will implement *show*

spring Pivotal

Initial Spring MVC Configuration

Sets up Spring MVC with convenient defaults – see Advanced section for full details

```java
@Configuration
@EnableWebMvc
public class MvcConfig {

    //  No beans required for basic Spring MVC usage.

}
```

Spring MVC *automatically* defines several beans.

Only need to provide additional beans and/or overrides to default beans (if desired)

- For example: view resolvers

spring Pivotal

2. Implement the Controller

```
@Controller
public class RewardController {
    private RewardLookupService lookupService;

    @Autowired
    public RewardController(RewardLookupService svc) {
        this.lookupService = svc;
    }

    @RequestMapping("/reward/show")
    public String show(@RequestParam("id") long id,
                                Model model) {
        Reward reward = lookupService.lookupReward(id);
        model.addAttribute("reward", reward);
        return "rewardView";
    }
}
```

Depends on application service

Automatically filled in by Spring

Selects the "rewardView" to to render the reward

3. Register the Controller

```
@Configuration
@EnableWebMvc
@ComponentScan("accounts.web")
public class MvcConfig() {

}
```

- Component-scanning very effective for MVC controllers!
- **Be specific** when indicating base package, avoid loading non-web layer beans
- Feel free to use <bean /> or @Configuration approaches as desired

4. Implement the View

```
<html>
   <head><title>Your Reward</title></head>
   <body>
      Amount=${reward.amount} <br/>
      Date=${reward.date} <br/>
      Account Number=${reward.account} <br/>
      Merchant Number=${reward.merchant}
   </body>
</html>
```

References result model object by name

/WEB-INF/views/rewardView.jsp

Note: no references to Spring object / tags required in JSP.

spring Pivotal

5. Register ViewResolver

```
@Configuration
@EnableWebMvc
@ComponentScan("accounts.web")
public class MvcConfig {

  @Bean
  public ViewResolver simpleViewResolver() {
     InternalResourceViewResolver vr =
            new InternalResourceViewResolver();
     vr.setPrefix ( "/WEB-INF/views/" );
     vr.setSuffix ( ".jsp" );
     return vr;
  }
}
```

- Controller returns rewardList
- ViewResolver converts to /WEB-INF/views/rewardList.jsp

spring Pivotal

6. Deploy and Test

http://localhost:8080/rewardsadmin/reward/show?id=1

Your Reward

Amount = $100.00
Date = 2006/12/29
Account Number = 123456789
Merchant Number = 1234567890

Summary

- Spring MVC is Spring's web framework
  - @Controller classes handle HTTP requests
  - URL information available
    - @RequestParam, @PathVariable
  - Data returned via the Model
  - Output (HTML) generated by Views
- Multiple View technologies supported
  - ViewResolvers define where Views can be found

Learn More:
Spring-Web – 4 day course on Spring Web Modules

Lab

Adding a Web Interface

Coming Up: @EnableWebMvc

MVC Additions in Spring 3.x and 4.x

- @MVC and legacy Controllers enabled by default
  - Appropriate Controller Mapping and Adapters registered out-of-the-box
- Newer features *not* enabled by default
  - Stateless converter framework for binding & formatting
  - Support for JSR-303 declarative validation for forms
  - HttpMessageConverters (for RESTful web services)
- *How do you use these features?*

@EnableWebMvc

- Registers Controller Mapping/Adapter for @MVC only
  - You lose legacy default mappings and adapters!
  - Enables custom conversion service and validators
  - Beyond scope of this course

```
@Configuration
@EnableWebMvc
public class RewardConfig {

    @Bean
    public rewardController(RewardLookupService service) {
        return new RewardController(service);
    }
    ...
```

spring Pivotal

WebMvcConfigurerAdapter

- Optionally extend WebMvcConfigurerAdapter
  - Override methods to define/customize web-beans

```
@Configuration
@EnableWebMvc
public class RewardConfig extends WebMvcConfigurerAdapter {

    @Bean  public rewardController(RewardLookupService service) { … }

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // Register your own type converters and formatters...
    }

    ...
```

Example: add custom formatters

spring Pivotal

MVC Namespace

- XML Equivalent to @EnableWebMvc

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xsi:schemaLocation="...">

   <!-- Provides default conversion service, validator and message converters -->
   <mvc:annotation-driven/>
```

Learn More:
Spring-Web – 4 day course on Spring Web Modules

Older Versions of Spring MVC

- Spring MVC is highly backwards compatible
  - Most default settings have remained unchanged since Spring 2.5 (versions 3.0, 3.1, 3.2, 4.0, 4.1, 4.2, 4.3 … !)
- However, old default settings are no longer recommended
  - Newer styles of controllers, adapters, message convertors, validators ...
- To enable the *more modern set of defaults*
  - **Use Spring Boot**
  - *Or* use `@EnableWebMvc` or `<mvc:annotation-config/>` explicitly

Spring Boot - Basics

Getting started with Spring Boot

Starter POMs, Auto-Configuration

spring Pivotal

What is Spring Boot?

- Spring Applications typically require a lot of setup
  - Consider working with JPA. You need:
    - Datasource, TransactionManager, EntityManagerFactory ...
  - Consider a web MVC app. You need:
    - WebApplicationInitializer / web.xml, ContextLoaderListener, DispatcherServlet, ...
  - An MVC app using JPA would need all of this
- *BUT: Much of this is predictable*
  - Spring Boot can do most of this setup for you

spring Pivotal

What is Spring Boot?

- An opinionated runtime for Spring Projects
- Supports different project types, like Web and Batch
- Handles most low-level, predictable setup for you

- It is not:
  - A code generator
  - An IDE plugin

See: Spring Boot Reference
http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle

Topics in this Session

- **What is Spring Boot?**
  - **Definition and Hello World example**
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Packaging
- Web Applications with Spring Boot
- Ease of Use Features

Opinionated Runtime?

- Spring Boot uses sensible defaults, *“opinions”*, mostly based on the classpath contents.
- For example
  - Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.
  - Creates a default Spring MVC setup, if Spring MVC is on the classpath.
- Everything can be overridden easily
  - But most of the time not needed

Hello World example

- Just a few files to get a running Spring Web application

pom.xml

Setup Spring Boot dependencies

HelloController class

Basic Spring MVC controller

application.properties + hello.jsp

View setup

Application class

Application launcher

Hello World (1) – Maven descriptor

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

parent

Spring MVC
Embedded Tomcat
Jackson...

pom.xml

Maven is just one option. You can also use Gradle or Ant/Ivy

spring Pivotal

Hello World (2) – Spring MVC controller

- A minimal controller to keep this example simple
  - Returns a JSP view-name

```java
@Controller
public class HelloController {
   @RequestMapping("/")
   public String hello() {
      return "hello";
   }
}
```

Nothing to do, just return view name

Controller.java

spring Pivotal

Hello World (3) – Implement the View

```
<html>
  <head><title>Hello</title></head>
  <body>
    <p>Hello Spring Boot</p>
  </body>
</html>
```

/WEB-INF/views/hello.jsp

application.properties

```
spring.mvc.view.prefix=/WEB-INF/views
spring.mvc.view.suffix=.jsp
```

Configure an *InternalResourceViewResolver*

Hello World (4) – Application Class

- *@SpringBootApplication* annotation enables Spring Boot
  - Runs Tomcat *embedded*

```
@SpringBootApplication
public class Application {

   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }

}
```

application.java

Main method will be used to run the packaged application from the command line – *old style!*

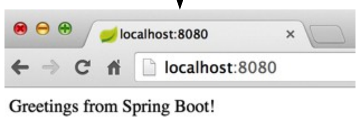
Putting it all together

`mvn package` — Maven command to generate an archive file

`helloApp-0.0.1-SNAPSHOT.jar` — Generated file

generated file

`java -jar helloApp-0.0.1-SNAPSHOT.jar` — App started using command line



App runs on port 8080

Deployment

- Our “Hello World” example bundles Tomcat inside the application
  - Runs as an executable *JAR*
- Spring Boot apps can also be deployed into an existing app server
  - As a familiar *WAR* file
  - PROs and CONs to be discussed later

Topics in this Session

- What is Spring Boot?
  - Definition and Hello World example
- **Spring Boot Explained**
  - **Dependency Management**
  - Auto Configuration
  - Packaging
- Web Applications with Spring Boot
- Ease of Use Features

Spring Boot Needs Dependencies

- Auto-configuration works by analyzing the classpath
  - If you forget a dependency, Spring Boot can't configure it
  - A dependency management tool is recommended
  - Spring Boot parent and starters make it much eaiser
- Spring Boot works with Maven, Gradle, Ant/Ivy
  - Our content here will show Maven

Spring Boot Parent POM

- Parent POM defines key versions of dependencies and Maven plugins

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.0.RELEASE</version>
</parent>
```

Defines properties for dependencies, for example: `${spring.version} = 4.3`

Core Spring *“Starter”* Dependencies

- Easy way to bring in multiple coordinated dependencies
  - Including *“Transitive”* Dependencies

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

Version not needed!
Defined by parent.

Resolves ~ 16 JARs!

spring-boot-.jar*	*spring-core-*.jar*
spring-context-.jar*	*spring-aop-*.jar*
spring-beans-.jar*	*aopalliance-*.jar*
...	

Spring Web Dependencies

- Everything you need to develop a web application with Spring

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

Resolves
spring-web-.jar*
spring-webmvc-.jar*
tomcat-.jar*
jackson-databind-.jar*
…

spring | Pivotal

Test Dependencies

- Common test libraries

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

Resolves
spring-test-.jar*
junit-.jar*
mockito-.jar*
…

spring | Pivotal

Available Starter POMs

- Not essential but *strongly* recommended
- Coordinated dependencies for common Java enterprise frameworks
  – Pick the starters you need in your project
- To name a few:
  – **`spring-boot-starter-jdbc`**
  – **`spring-boot-starter-jpa`**
  – **`spring-boot-starter-batch`**

See: Spring Boot Reference, Starter POMs
http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter-poms

spring Pivotal

Topics in this Session

- What is Spring Boot?
  – Definition and Hello World example
- **Spring Boot Explained**
  – Dependency Management
  – **Auto Configuration**
  – Packaging
- Web Applications with Spring Boot
- Ease of Use Features

spring Pivotal

Spring Boot @EnableAutoConfiguration

- *@EnableAutoConfiguration* annotation on a Spring Java configuration class
  - Causes Spring Boot to automatically create beans it thinks you need
  - Usually based on classpath contents, can easily override

```
@Configuration
@EnableAutoConfiguration
public class MyAppConfig {
    public static void main(String[] args) {
            SpringApplication.run(MyAppConfig.class, args);
    }
}
```

Spring Boot class

Shortcut @SpringBootApplication

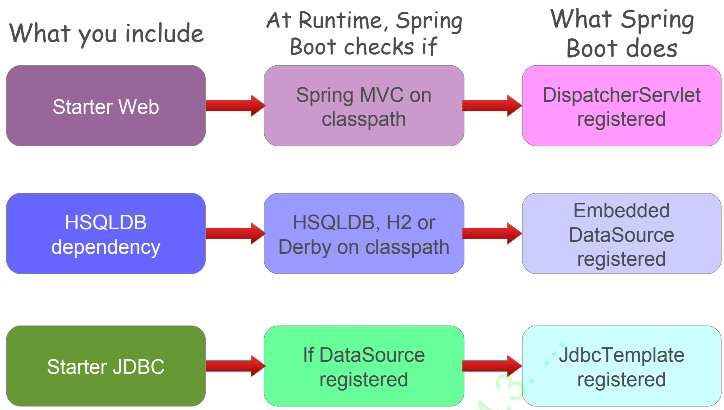
- Very common to use @EnableAutoConfiguration, @Configuration, and @ComponentScan together
  - @ComponentScan, with no arguments, scans the current package *and* its sub-packages

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class MyAppConfig {
   ...
}
```

```
@SpringBootApplication
public class MyAppConfig {
   ...
}
```

@SpringBootApplication was available from Spring Boot 1.2

Auto-configuration: Examples



Topics in this Session

- What is Spring Boot?
  – Definition and Hello World example
- **Spring Boot Explained**
  – Dependency Management
  – Auto Configuration
  – **Packaging**
- Web Applications with Spring Boot
- Ease of Use Features

Packaging

- Spring Boot creates a single archive
  - JAR (or WAR)
- Gradle and Maven plugins available
  - Generate an *executable* JAR

    `java -jar yourapp.jar`

Maven Packaging

- Add Boot Maven plugin to pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Packaging Result

- "**`mvn package`**" execution produces (in **`target`**)

```
22M  yourapp-0.0.1-SNAPSHOT.jar
 5K  yourapp-0.0.1-SNAPSHOT.jar.original
```

  - **`.jar.original`** contains only your code (a traditional JAR file)
  - **`.jar`** contains your code *and* all libs – executable
    - *Notice that it is much bigger*

spring Pivotal

Topics in this Session

- What is Spring Boot?
  - Definition and Hello World example
- Spring Boot Explained
- **Web Applications with Spring Boot**
  - **Containerless Applications**
  - Spring Boot inside of a Servlet Container
- Ease of Use Features

spring Pivotal

Spring Boot as a Runtime

- Spring Boot can startup an *embedded* web server
  - You can run a web application from a JAR file!
  - Tomcat included by Web Starter

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

> Simpler for running and testing, may be preferred when deploying *Cloud Native* applications

Jetty Support

- Jetty can be used instead of Tomcat

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Excludes Tomcat

Adds Jetty

Jetty automatically detected and used!

JAR vs WAR?

- Why Run a Web Application Outside of a Container?
  - No separation of container config and app config
    - Depend on each other anyway (JNDI DS names, security config ...)
  - Projects usually know which container will be used
    - Why not just include it?
  - No special IDE support needed
    - Easier debugging and profiling, hot code replacement
  - Familiar model for non-Java developers
  - Recommended for Cloud Native applications
    - 12-Factor applications #7 (see http://12factor.net)

Topics in this Session

- What is Spring Boot?
  - Definition and Hello World example
- Spring Boot Explained
- **Web Applications with Spring Boot**
  - Containerless Applications
  - **Spring Boot inside of a Servlet Container**
- Ease of Use Features

Spring Boot in a Servlet Container

- Spring Boot can also run in any Servlet 3.x container
  - e.g. Tomcat 7+, Jetty 8+
- Only small changes required
  - Change artifact type to WAR (instead of JAR)
  - Extend **`SpringBootServletInitializer`**
  - Override configure method
- Still no *web.xml* required

spring Pivotal

Spring Boot WAR file

- Spring Boot produces hybrid WAR file
- Can still be executed with embedded Tomcat
  - using "**`java -jar yourapp.war`**"
- Traditional WAR file is produced as well
  - without embedded Tomcat
  - just drop it in your application server web app directory

spring Pivotal

Spring Boot in a Servlet Container

Sub-classes Spring's *WebApplicationInitializer* – called by the web container (Servlet 3.0)

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    protected SpringApplicationBuilder configure(
                SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

}
```

The above requires no *web.xml* file

spring Pivotal

WAR Packaging Result

- “mvn package” execution produces:

```
22M  yourapp-0.0.1-SNAPSHOT.war
20M  yourapp-0.0.1-SNAPSHOT.war.original
```

  - `.war.original` is a traditional WAR file
  - `.war` is a hybrid WAR file, additionally containing the embedded Tomcat

spring Pivotal

Servlet Container *and* Containerless

- Can execute: `java -jar yourapp-0.0.1-SNAPSHOT.war`

```java
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    protected SpringApplicationBuilder configure(
            SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

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

WAR support

Provides main method too

Warning: Embedded tomcat version must match version used with WAR

spring Pivotal

It's Your Choice

- There is no force to go containerless
  - Embedded container is just one feature of Spring Boot
- Traditional WAR also benefits a lot from Spring Boot
  - Automatic Spring MVC setup, including DispatcherServlet
  - Sensible defaults based on the classpath content
  - Embedded container can be used during development

spring Pivotal

Topics in this session

- What is Spring Boot?
  - Definition and Hello World example
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- **Ease of Use Features**

spring Pivotal

Externalized Properties – 1

application.properties

- Developers commonly externalize properties to files
  - Easily consumable via Spring PropertySource
  - But developers name / locate their files different ways

- Spring Boot looks for **`application.properties`** in well-known locations
  - You can put any properties you need in here and Boot will automatically find and load them
  - Available to **`Environment`** and **`@Value`** in usual way

spring Pivotal

Externalized Properties – 2

application.properties

- Spring Boot can be easily configured by setting any of its many, many properties
  - Set these in application.properties also
- Example: configure default **`DataSource`** bean

```
database.host=localhost
database.user=admin
```

application.properties

See Appendix A of Spring Boot documentation:
http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

spring Pivotal

Externalized Properties – 3

YAML Alternative

- Spring Boot also supports YAML configuration
  - More concise, indented text format (similar to JSON)
  - By default it looks for ***`application.yml`***
  - *Do not use tabs*

```
database:
  host: localhost
  user: admin
```

application.yml

spring Pivotal

Controlling Logging Level

- Boot can control the logging level
  - Just set it in **`application.properties`**
- Works with most logging frameworks
  - Java Util Logging, Logback, Log4J, Log4J2

```
logging.level.org.springframework=DEBUG
logging.level.com.acme.your.code=INFO
```

application.properties

> Try to stick to SLF4J in the application.
> The *advanced* section covers how to change the logging framework

spring Pivotal

DataSource Configuration

- Use either spring-boot-starter-jdbc or spring-boot-starter-data-jpa and include a JDBC driver on classpath
- Declare properties

application.properties

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- That's It!
  - Spring Boot will create a DataSource with properties set
  - Will even use a connection pool if the library is found on the classpath!

spring Pivotal

Web Application Convenience – 1

- Boot automatically configures
  - A **`DispatcherServlet`** & **`ContextLoaderListener`**
  - Spring MVC using same defaults as **`@EnableWebMvc`**
- *Plus* many useful *extra features*:
  - Static resources served from classpath
    - /static, /public, /resources or /META-INF/resources
  - Templates served from /templates
    - If Velocity, Freemarker, Thymeleaf, or Groovy on classpath
  - Provides default /error mapping
    - Easily overridden
  - Default **`MessageSource`** for I18N

Web Application Convenience – 2

- Spring Boot automatic web configuration
  - Because ***`spring-webmvc*.jar`*** on classpath
    - Such as when **`spring-boot-starter-web`** is used
  - *You did not specify* **`@EnableWebMvc`**
- **`@EnableWebMvc`** in a Spring Boot application
  - *Only* sets up what **`@EnableWebMvc`** normally does
  - *No* Spring Boot web extras

Bottom Line: Most Spring Boot web applications *do not* specify **`@EnableWebMvc`**

Summary

- Spring Boot speeds up Spring application development
- You always have full control and insight
- Nothing is generated
- No special runtime requirements
- No servlet container needed (if you want)
  - E.g. ideal for microservices

Advanced Section: More customization, Configuration using Properties or YAML files, Logging control, Boot-driven testing

Spring-Boot Developer – 2 day in-depth Spring Boot workshop

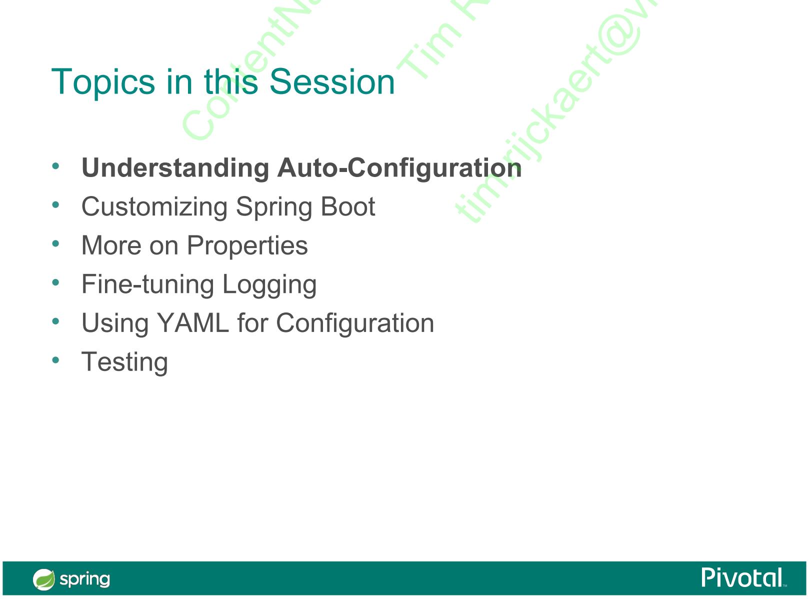
spring Pivotal

Lab

Simplification using Spring Boot

spring Pivotal

Spring Boot – Going Further
Going beyond default settings
Customization, Logging, YAML Properties, Testing
spring
Pivotal



Topics in this Session
Understanding Auto-Configuration
Customizing Spring Boot
More on Properties
Fine-tuning Logging
Using YAML for Configuration
Testing
spring
Pivotal

How Does Spring Boot Work?

- Extensive use of pre-written @Configuration classes
- Conditional on
  - The contents of the classpath
  - Properties you have set
  - Beans already defined
- @Profile is an example of conditional configuration
  - Spring Boot takes this idea to the next level

spring Pivotal

@Conditional Annotations

- Allow conditional bean creation
  - Only create if other beans exist (or don't exist)

```
@Bean
@ConditionalOnBean(name={"dataSource"})
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```

  - Or by type: `@ConditionalOnBean(type={DataSource.class})`
- Many others:
  - `@ConditionalOnClass`, `@ConditionalOnProperty`, ... `@ConditionalOnMissingBean`, `@ConditionalOnMissingClass`

> @Profile is a special case of @Conditional

spring Pivotal

What are AutoConfiguration Classes

- Pre-written Spring configurations
  - `org.springframework.boot.autoconfigure` package
  - See `spring-boot-autoconfigure` JAR file
    - Best place to check what they exactly do

```
@Configuration
public class DataSourceAutoConfiguration
    implements EnvironmentAware {
    …
    @Conditional(...)
    @ConditionalOnMissingBean(DataSource.class)
    @Import(...)
    protected static class EmbeddedConfiguration { ... }
    …
}
```

Spring Boot defines many of these configurations. They activate in response to dependencies on the classpath

spring Pivotal

Topics in this Session

- Understanding Auto-Configuration
- **Customizing Spring Boot**
- More on Properties
- Fine-tuning Logging
- Using YAML for Configuration
- Testing

spring Pivotal

Controlling What Spring Boot does

- There are several options
  - Set some of Spring Boot's properties
  - Define certain beans yourself so Spring Boot won't
  - Explicitly disable some auto-configuration
  - Changing dependencies

spring Pivotal

1. Using Properties

- Spring Boot looks for application.properties in these locations (in this order):
  - **/config** sub-directory of the working directory
  - The working directory
  - **config** package in the classpath
  - classpath root
- Creates a *PropertySource* based on these files
- Many, many configuration properties available

See: Spring Boot Reference, Appendix A. Common Application Properties
http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties

spring Pivotal

2. Replacing Generated Beans

- Normally beans you declare *explicitly* disable any auto-created ones.
  - *Example:* Your DataSource stops Spring Boot creating a default DataSource
  - Bean name often not important
  - Works with XML, Component Scanning and/or Java Config

```
@Bean
public DataSource dataSource() {
   return new EmbeddedDatabaseBuilder().
                           setName("RewardsDb").build();
}
```

spring Pivotal

3. Selectively Disabling Auto Configuration

- Can disable some AutoConfiguration classes
  - If they don't suit your needs
- Use the *@EnableAutoConfiguration* annotation
  - List the auto-configuration classes to exclude in the “exclude” attribute

```
@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
public class ApplicationConfiguration {
    …
}
```

spring Pivotal

4a. Overriding Dependency Versions

- Spring Boot POMs preselect the versions of frameworks
  - Ensures the versions of all frameworks are consistent
  - Avoids "*dependency hell*"
- Should I override the version of a given framework?
  - Ideally no, it makes your life more complicated
- But there are good reasons to override it sometimes
  - A bug in the given version
  - Company policies

4b. Overriding Dependency Versions

- Set the appropriate Maven property in your pom.xml

```
<properties>
    <spring.version>4.2.0.RELEASE</spring.version>
</properties>
```

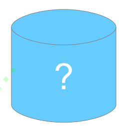
- Check this POM to know all the properties names
  - https://github.com/spring-projects/spring-boot/blob/master/spring-boot-dependencies/pom.xml

This only works if you *inherit* from the starter. You need to redefine the artifact if you directly import the dependency

Configuration Example: DataSource (1)

- A common example of how to control or override Spring Boot's default configuration

- Typical customizations
  - Use the predefined properties
  - Change the underlying data source connection pool implementation
  - Define your own DataSource bean (shown earlier)



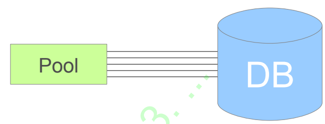
Example: DataSource Configuration (2)

- Common properties configurable from properties file

```
spring.datasource.url=             # Connection settings
spring.datasource.username=
spring.datasource.password=
spring.datasource.driver-class-name=

spring.datasource.schema=          # SQL scripts to execute
spring.datasource.data=

spring.datasource.initial-size=    # Connection pool settings
spring.datasource.max-active=
spring.datasource.max-idle=
spring.datasource.min-idle=
```


Example: DataSource Configuration (3)

- Spring Boot creates a pooled DataSource by default
  - If a known pool dependency is available
    - *spring-boot-starter-jdbc* or *spring-boot-starter-jpa* starters pull in *tomcat-jdbc* connection pool by default
  - *Alternatives:* Tomcat, HikariCP, Commons DBCP 1 & 2
    - Simply use relevant dependency



Example: Web Container Configuration

- Many settings accessible from the configuration file

```
server.port=9000
server.address=192.168.1.20
server.session-timeout=1800
server.context-path=/rewards
server.servlet-path=/admin
```

- Also available
  - SSL (keystore, truststore for client authentication)
  - Tomcat specifics (access log, compression, etc)

Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- **More on Properties**
- Fine-tuning Logging
- Using YAML for Configuration
- Testing

spring Pivotal

Overriding Properties

Applies to Spring *or* Spring Boot

- Order of evaluation of the properties (non-exhaustive)
  - Command line arguments
  - Java system properties
  - OS environment variables
  - Property file(s) – including `application.properties`
- Can access any of them using `@Value` in the usual way
- *Recommendation:*
  - Use Property files to define defaults
  - Override *externally* using one of the other 3 options

spring Pivotal

Renaming `application.properties`

- You can override the name of this file
  - *Example:* to use **`myserver.properties`**

Application.java

```
public static void main(String[] args) {
    System.setProperty("spring.config.name", "myserver");
    SpringApplication.run(Application.class, args);
}
```

Note: `myserver` ***not*** `myserver.properties`

spring Pivotal

Relaxed Property Binding

- No need for exact match between desired properties and names
- Intuitive mapping between java-style properties and environment variables
  - **`path`** equivalent to **`PATH`**
  - **`java.home`** equivalent to **`JAVA_HOME`**
  - Easy overriding of property without changing the name!

```
@Configuration
class AppConfig {

  @Value("${java.home}")
  String javaInstallDir;
  ...
}
```

spring Pivotal

The Problem with Property Placeholders

- Using property placeholders is sometimes cumbersome
  - Many properties, prefix has to be repeated

```
@Configuration
public class RewardsClientConfiguration {

    @Value("${rewards.client.host}") String host;
    @Value("${rewards.client.port}") int port;
    @Value("${rewards.client.logdir}") String logdir;
    @Value("${rewards.client.timeout}") int timeout;

    ...
}
```

spring Pivotal

Use @ConfigurationProperties

- Add @ConfigurationProperties to *dedicated* container bean
  - Will hold the externalized properties
  - Avoids repeating the prefix
  - Data-members automatically set from corresponding properties

```
@Component
@ConfigurationProperties(prefix="rewards.client")
public class ConnectionSettings {

    private String host;
    private int port;
    private String logdir;
    private int timeout;
    ...    // getters/setters
}
```

```
rewards.client.host=192.168.1.42
rewards.client.port=8080
rewards.client.logdir=/logs
rewards.client.timeout=2000
```

application.properties

spring Pivotal

Use @EnableConfigurationProperties

- *@EnableConfigurationProperties* on configuration class
  - Specify and auto-inject the container bean

```
@Configuration
@EnableConfigurationProperties(ConnectionSettings.class)
public class RewardsClientConfiguration {
    // Spring initialized this automatically
    @Autowired ConnectionSettings connectionSettings;

    @Bean public RewardClient rewardClient() {
        return new RewardClient(
            connectionSettings.getHost(),
            connectionSettings.getPort(), ...
        );
    }
}
```

Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- **Fine-tuning Logging**
- Using YAML for Configuration
- Testing

Logging frameworks

- Spring Boot includes by default
  - SLF4J: logging facade
  - Logback: SLF4J implementation
- Best practice: stick to this in your application
  - Use the SLF4J abstraction the application code
- Other logging frameworks are supported
  - Java Util Logging, Log4J, Log4J2

spring Pivotal

Using another logging framework

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
    <exclusions>
        <exclusion>                                  Excludes Logback
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </exclusion>
    </exclusions>                                    Includes Log4J
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

spring Pivotal

Logging Output

- Spring Boot logs by default to the console
- Can also log to rotating files
  – Specify file OR path in application.properties

```
# Use only one of the following properties

# absolute or relative file to the current directory
logging.file=rewards.log

# will write to a spring.log file
logging.path=/var/log/rewards
```

> Spring Boot can also configure logging by using the appropriate configuration file of the underlying logging framework.

spring Pivotal

Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- Fine-tuning Logging
- **Using YAML for Configuration**
- Testing

spring Pivotal

What is YAML?

- *Yaml Ain't a Markup Language*
  - Recursive acronym
- Created in 2001
- Alternative to .properties files
  - Allows hierarchical configuration
- Java parser for YAML is called SnakeYAML
  - Must be in the classpath
  - Provided by spring-boot-starters

YAML for Properties

- Spring Boot support YAML for Properties
  - An alternative to properties files

```
database.host = localhost
database.user = admin
```

application.properties

equals

```
database:
  host: localhost
  user: admin
```

application.yml

colon

- YAML is convenient for hierarchical configuration data
  - Spring Boot properties are organized in groups
  - *Examples:* server, database, etc

Multiple Profiles Inside a Single YAML File

- YAML file can contain configuration for multiple profiles
  - '---' implies a separation between profiles

```
logging.level:
  org.springframework: INFO
---
spring.profiles: development
database:
  host: localhost
  user: dev
---
spring.profiles: production
database:
  host: 198.18.200.9
  user: admin
```

Used for all profiles

“development” profile only

“production” profile only

application.yml

spring Pivotal

Multiple Profiles Inside Multiple Files

```
logging.level:
  org.springframework: INFO
---
spring.profiles: development
database:
  host: localhost
  user: dev
---
spring.profiles: production
database:
  host: 198.18.200.9
  user: admin
```

application.yml

```
logging.level:
  org.springframework: INFO
```

application.yml

```
database:
  host: localhost
  user: dev
```

application-development.yml

```
database:
  host: 198.18.200.9
  user: admin
```

application-production.yml

Alternatively `application-development.properties` and `application-production.properties` can be used in same way

spring Pivotal

Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- Fine-tuning Logging
- Using YAML for Configuration
- **Testing**

spring Pivotal

Testing: @SpringBootTest

Alias for *SpringJUnit4ClassRunner*

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes=TransferApplication.class)
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() {
        TransferConfirmation conf = transferService. transfer(...);
        ...
    }

}
```

Sets up *same* configuration for the tests that the application would use

```
@SpringBootApplication
public class TransferApplication {
    public static void main(String[] args) {
        SpringApplication.run(TransferApplication.class, args);
    }
}
```

spring Pivotal

Web Application Testing

Can use *with or without* Spring Boot

- Spring Unit test with **`@WebAppConfiguration`**
  - Creates a **`WebApplicationContext`**
  - Can test code that uses web features
    - *ServletContext*, *Session* and *Request* bean scopes
  - Configures the location of resources
    - Defaults to **`src/main/webapp`**
      - Override using annotation's value attribute
    - For classpath resources use **`classpath:`** prefix

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
public final class TransferServiceTests { ... }
```

spring Pivotal

Summary

- Spring Boot takes care of boilerplate configuration
  - Auto-configuration can be overridden/disabled
  - Frameworks versions can be overridden too
- Spring Boot enhances Spring configuration externalization mechanisms
  - Properties/YAML files
  - Easier to override using env/Java system variables

spring Pivotal

Spring Security

Web Application Security

Addressing Common Security Requirements

Topics in this Session

- **High-Level Security Overview**
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Security Concepts

- Principal
  - User, device or system that performs an action
- Authentication
  - Establishing that a principal's credentials are valid
- Authorization
  - Deciding if a principal is allowed to perform an action
- Secured item
  - Resource that is being secured

spring Pivotal

Authentication

- There are many authentication mechanisms
  - e.g. basic, digest, form, X.509
- There are many storage options for credential and authority information
  - e.g. Database, LDAP, in-memory (development)

spring Pivotal

Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
  - ADMIN can cancel orders
  - MEMBER can place orders
  - GUEST can browse the catalog

Topics in this Session

- High-Level Security Overview
- **Motivations of Spring Security**
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

See: Spring Security Reference
http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/

Motivations - I

- Spring Security is portable across containers
  - Secured archive (WAR, EAR) can be deployed as-is
  - Also runs in standalone environments
  - Uses Spring for configuration
- Separation of Concerns
  - Business logic is decoupled from security concerns
  - Authentication and Authorization are decoupled
    - Changes to the authentication process have *no impact* on authorization

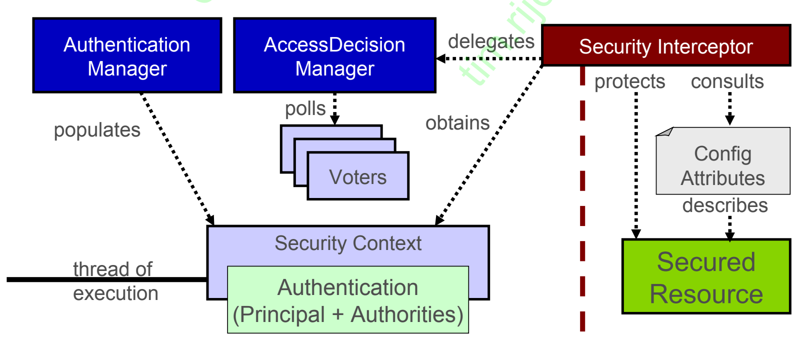
Motivations: II

- Flexibility
  - Supports all common authentication mechanisms
    - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
  - Configurable storage options for user details (credentials and authorities)
    - RDBMS, LDAP, custom DAOs, properties file, etc.
- Extensible
  - All the following can be customized
    - How a principal is defined
    - How authorization decisions are made
    - Where security constraints are stored

Consistency of Approach

- The goal of authentication is *always* the same *regardless* of the mechanism
  - Establish a security context with the authenticated principal’s information
  - Out-of-the-box this works for web applications
- The *process* of authorization is *always the same* regardless of resource type
  - Consult the attributes of the secured resource
  - Obtain principal information from security context
  - Grant or deny access

Spring Security – the Big Picture



Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- **Spring Security in a Web Environment**
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Setup the Filter Chain

- Implementation is a *chain* of Spring configured filters
  - Requires a **`DelegatingFilterProxy`** which *must* be called *springSecurityFilterChain*
- Use *one* of these options
  - Spring Boot does it automatically
  - Use **`@EnableWebSecurity`**
    - See next slide
    - For total control, subclass Spring Security's web-initializer
      - **`AbstractSecurityWebApplicationInitializer`**
  - Declare as a **`<filter>`** in **`web.xml`** in usual way

Configuration in the Application Context

- Java Configuration (XML also available)
  - Extend *WebSecurityConfigurerAdapter* for more control

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {

  }

  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

  }
}
```

Web-specific security settings

General security settings (authentication manager, ...).

authorizeRequests()

- Adds specific authorization requirements to URLs
- Evaluated in the order listed
  - first match is used, put specific matches first

```java
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/css/**","/images/**","/javascript/**").permitAll()
      .antMatchers("/accounts/edit*").hasRole("ADMIN")
      .antMatchers("/accounts/account*").hasAnyRole("USER","ADMIN")
      .antMatchers("/accounts/**").authenticated()
      .antMatchers("/customers/checkout*").fullyAuthenticated()
      .antMatchers("/customers/**").anonymous();
```

Specifying login and logout

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/aaa*").hasRole("ADMIN")
      .and()                          //  method chaining!

    .formLogin()                      //  setup form-based authentication
      .loginPage("/login.jsp")        //  URL to use when login is needed
      .permitAll()                    //  any user can access
      .and()                          //  method chaining!

    .logout()                         //  configure logout
      .permitAll();                   //  any user can access
}
```


An Example Login Page

URL that indicates an authentication request.
Default: POST against URL used to display the page.

```
<c:url var='loginUrl'value='/login.jsp' />
<form:form action="${loginUrl}" method="POST">
  <input type="text" name="username"/>
  <br/>
  <input type="password" name="password"/>
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form:form>
```

The expected keys for generation of an authentication request token

login-example.jsp

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- **Configuring Web Authentication**
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Configure Authentication

- DAO Authentication provider is default
  - Expects a *UserDetailsService* implementation to provide credentials and authorities
    - Built-in: In-memory (properties), JDBC (database), LDAP
    - Custom
- Or define your *own* Authentication provider
  - *Example:* to get pre-authenticated user details when using single sign-on
    - CAS, TAM, SiteMinder ...
  - See online examples

Authentication Provider

- Use a *UserDetailsManagerConfigurer*
  - Three built in options:
    - LDAP, JDBC, in-memory (for quick testing)
  - Or use your own UserDetailsService implementation

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
            .withUser("hughie").password("hughie").roles("GENERAL").and()
            .withUser("dewey").password("dewey").roles("ADMIN").and()
            .withUser("louie").password("louie").roles("SUPPORT");
}
```

Not web-specific

Adds a UserDetailsManagerConfigurer

login

password

Supported roles

Sourcing Users from a Database

- Configuration:

```
@Autowired DataSource dataSource;

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

  auth.jdbcAuthentication().dataSource(dataSource);

}
```

Can customize queries using methods:
usersByUsernameQuery()
authoritiesByUsernameQuery()
groupAuthoritiesByUsername()

Sourcing Users from a Database

Queries RDBMS for users and their authorities

- Provides default queries
  - SELECT username, password, enabled FROM users WHERE username = ?
  - SELECT username, authority FROM authorities WHERE username = ?
- Groups also supported
  - groups, group_members, group_authorities tables
  - See online documentation for details
- Advantage
  - Can modify user info while system is running

Password Encoding

- Can encode passwords using a hash
  - sha, md5, bcrypt, ...

```
auth.jdbcAuthentication()
  .dataSource(dataSource)
  .passwordEncoder(new StandardPasswordEncoder());
```

SHA-256 encoding

- Secure passwords using a well-known string
  - Known as a 'salt', makes brute force attacks harder

```
auth.jdbcAuthentication()
  .dataSource(dataSource)
  .passwordEncoder(new StandardPasswordEncoder("sodium-chloride"));
```

encoding with salt

Other Authentication Options

- Implement a custom UserDetailsService
  - Delegate to an existing User repository or DAO
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
  - OAuth, SAML
  - SiteMinder, Kerberos
  - JA-SIG Central Authentication Service

Authorization is *not* affected by changes to Authentication!

spring Pivotal

@Profile with Security Configuration

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/resources/**").permitAll();
  }
}
```

Use in-memory provider

```
@Configuration
@EnableWebSecurity
@Profile("development")
public class SecurityDevConfig extends SecurityBaseConfig {
  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("hughie").password("hughie").roles("GENERAL");
  }
}
```

spring Pivotal

@Profile with Security Configuration

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/resources/**").permitAll();
  }
}
```

```
@Configuration
@EnableWebSecurity
@Profile("production")
public class SecurityProdConfig extends SecurityBaseConfig {
  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
      auth.jdbcAuthentication().dataSource(dataSource);
  }
}
```

Use database provider

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- **Using Spring Security's Tag Libraries**
- Method security
- Advanced security: working with filters

spring Pivotal

Tag library declaration

- The Spring Security tag library is declared as follows

available since Spring Security 2.0

```jsp
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags" %>
```

- Facelet tags for JSF are also available
  - You need to define and install them manually
  - See *"Using the Spring Security Facelets Tag Library"* in the Spring Webflow documentation
  - Principal available in SpEL: **`#{principal.username}`**
- Equivalent functionality for other View technologies

Spring Security's Tag Library

- Display properties of the Authentication object

```jsp
You are logged in as:
<security:authentication property="principal.username"/>
```

- Hide sections of output based on role
  - Not recommended, roles in JSP page (see next slide)

```jsp
<security:authorize access="hasRole('ADMIN')">
   TOP-SECRET INFORMATION
   Click <a href="/admin/deleteAll">HERE</a> to delete all records.
</security:authorize>
```

Authorization in JSP based on intercept-url

- Role declaration can be centralized in Spring config files

URL to protect

```
<security:authorize url="/admin/deleteAll">
   TOP-SECRET INFORMATION
   Click <a href="/admin/deleteAll">HERE</a>
</security:authorize>
```

jsp

Pattern that matches the URL to be protected

```
.antMatchers("/admin/*").hasAnyRole("MANAGER","ADMIN")
```

Matching roles

Recommended over <security:authorize access="hasRole('ADMIN')"> example on previous slide

spring Pivotal

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- **Method security**
- Advanced security: working with filters

spring Pivotal

Method Security

- Spring Security uses AOP for security at the method level
  - annotations based on Spring annotations or JSR-250 annotations
  - Java configuration to activate detection of annotations
- Typically secure your services
  - Do not access repositories directly, bypasses security (and transactions)

spring Pivotal

Method Security - JSR-250

- JSR-250 annotations should be enabled

```
@EnableGlobalMethodSecurity(jsr250Enabled=true)
```

```
import javax.annotation.security.RolesAllowed;

public class ItemManager {
  @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
  public Item findItem(long itemNumber) {
    ...
  }
}
```

*Only supports **role-based** security – hence the name*

spring Pivotal

Method Security - @Secured

- Secured annotation should be enabled

```
@EnableGlobalMethodSecurity(securedEnabled=true)
```

```
import org.springframework.security.annotation.Secured;

public class ItemManager {
  @Secured("IS_AUTHENTICATED_FULLY")
  public Item findItem(long itemNumber) {
    ...
  }
}
```

```
@Secured("ROLE_MEMBER")
@Secured({"ROLE_MEMBER", "ROLE_USER"})
```

*Spring 2.0 syntax, so **not** limited to roles. SpEL **not** supported.*

spring Pivotal

Method Security with SpEL

- Use Pre/Post annotations for SpEL

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

```
import org.springframework.security.annotation.PreAuthorize;

public class ItemManager {
  @PreAuthorize("hasRole('MEMBER')")
  public Item findItem(long itemNumber) {
    ...
  }
}
```

spring Pivotal

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- **Advanced security: working with filters**

spring Pivotal

Spring Security in a Web Environment

- *SpringSecurityFilterChain*
  - Spring Boot does this automatically
  - Or declare in servlet configuration
- This single proxy filter delegates to a chain of Spring-managed filters
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain SecurityContext in HttpSession
  - *and more*

spring Pivotal

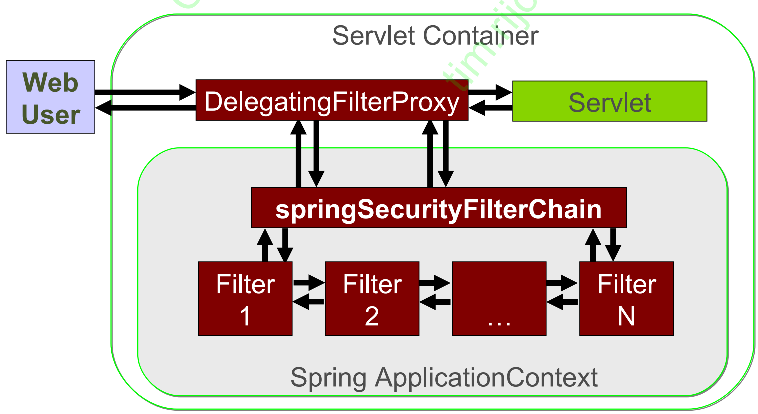
Example: Configuration in web.xml

- Define the single proxy filter
  - *springSecurityFilterChain* is a mandatory name
    - Refers to an existing Spring bean with same name

web.xml

```xml
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

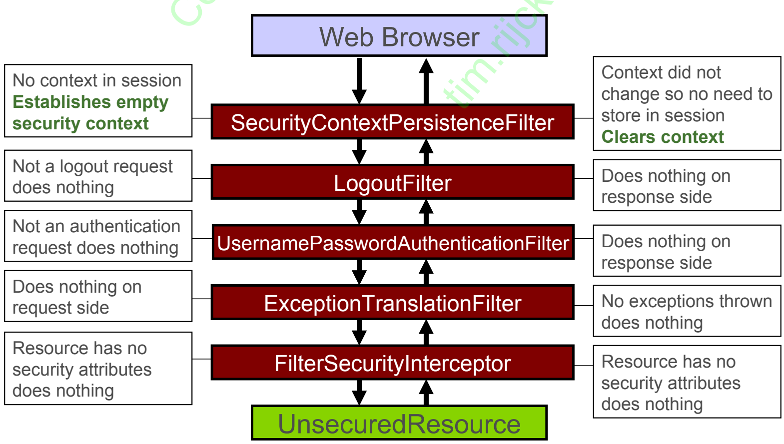

Web Security Filter Configuration



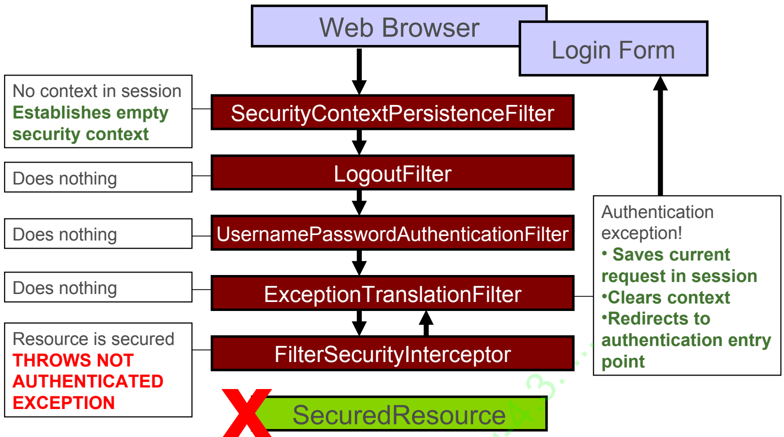
The Filter Chain

- With ACEGI Security 1.x
  - Filters were manually configured as individual <bean> elements
  - Led to verbose and error-prone XML
- Spring Security 2.x, 3.x, 4.x
  - Filters are initialized with correct values by default
  - Manual configuration is not required **unless you want to customize Spring Security's behavior**
  - It is still important to understand how they work underneath

Access Unsecured Resource Prior to Login

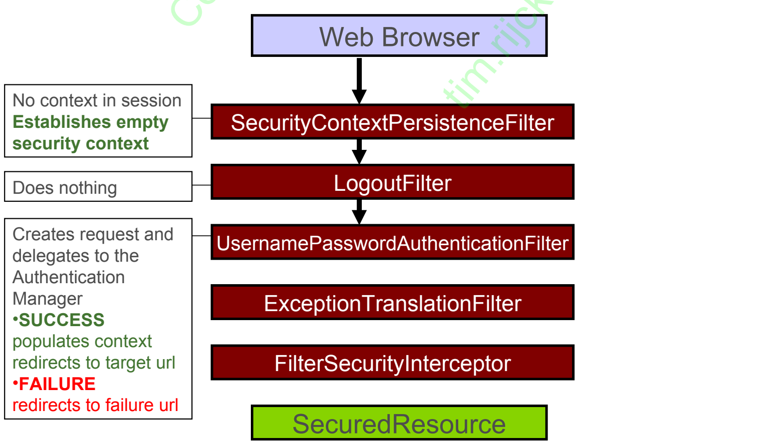


Access Secured Resource Prior to Login



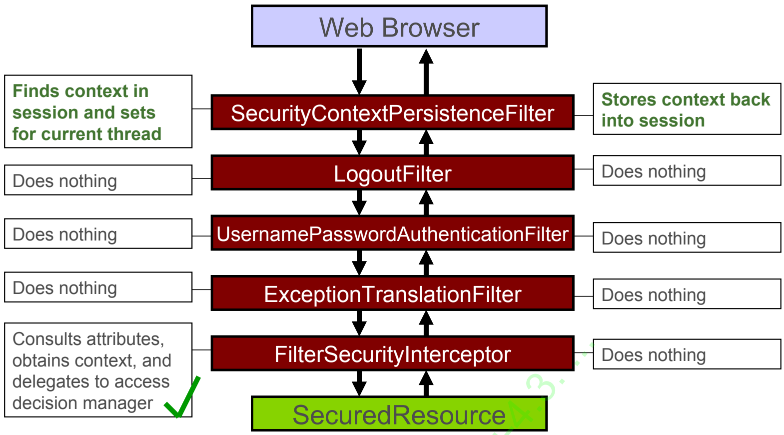
spring Pivotal

Submit Login Request

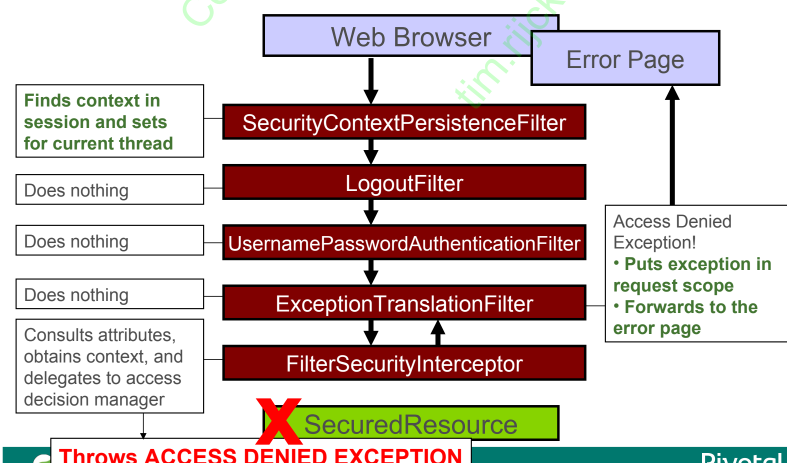


spring Pivotal

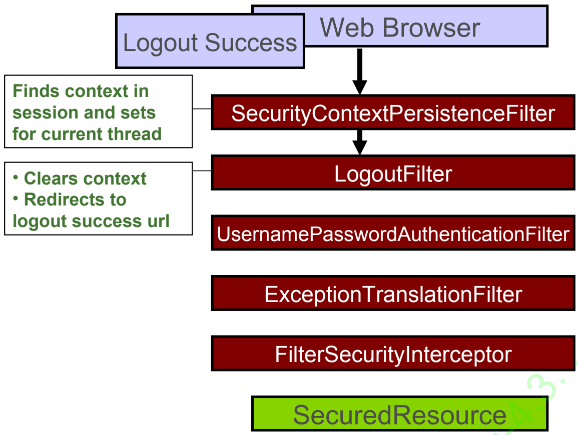
Access Resource With Required Role



Access Resource Without Required Role



Submit Logout Request



The Filter Chain: Summary

#	Filter Name	Main Purpose
1	SecurityContext IntegrationFilter	Establishes SecurityContext and maintains between HTTP requests *formerly: HttpSessionContextIntegrationFilter*
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword AuthenticationFilter	Puts Authentication into the SecurityContext on login request *formerly: AuthenticationProcessingFilter*
4	Exception TranslationFilter	Converts SpringSecurity exceptions into HTTP response or redirect
5	FilterSecurity Interceptor	Authorizes web requests based on on config attributes and authorities

Custom Filter Chain

- Filter can be **added** to the chain
  - Before or after existing filter

```
http.addFilterAfter ( myFilter, UsernamePasswordAuthenticationFilter.class );
…
@Bean
public Filter myFilter() {  return new MySpecialFilter(); }
```

- Filter on the stack may be **replaced** by a custom filter
  - Replacement must *extend* the filter being replaced.

```
http.addFilter ( myFilter );
…
@Bean
public Filter myFilter() {
      return new MySpecialFilter();
}
```

```
public class MySpecialFilter
      extends UsernamePasswordAuthenticationFilter {}
```


Summary

- Spring Security
  - Secure URLs using a chain of Servlet filters
  - And/or methods on Spring beans using AOP proxies
- Out-of-the-box setup usually sufficient – you define:
  - URL and/or method restrictions
  - How to login (typically using an HTML form)
  - Supports in-memory, database, LDAP credentials (and more)
  - Password encryption using familiar hashing techniques
  - Support for security tags in JSP views

Lab

Applying Spring Security to a Web Application

spring

Pivotal

Practical REST Web Services

Using Spring MVC to build RESTful Web Services

Extending Spring MVC to handle REST

Topics in this Session

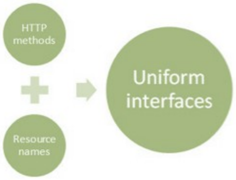
- **REST introduction**
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Accessing Request/Response Data
  - Putting it all together
- RESTful Clients with the RestTemplate
- Advanced Topics

REST Introduction

- Web apps not just usable by browser clients
  - Programmatic clients can also connect via HTTP
  - Such as: mobile applications, AJAX enabled web-pages
- REST is an *architectural style* that describes best practices to expose web services over HTTP
  - REpresentational State Transfer, term by Roy Fielding
  - HTTP as *application* protocol, not just transport
  - Emphasizes scalability
  - *Not* a framework or specification

REST Principles (1)

- Expose *resources* through URIs
  - Model nouns, not verbs
  - http://springbank.io/banking/accounts/123456789
- Resources support limited set of operations
  - GET, PUT, POST, DELETE in case of HTTP
  - All have well-defined semantics
- Example: update an order
  - PUT to /orders/123
  - don't POST to /order/edit?id=123



REST Principles (2)

- Clients can request particular representation
  - Resources can support multiple representations
  - HTML, XML, JSON, …
- Representations can link to other resources
  - Allows for extensions and discovery, like with web sites
- Hypermedia As The Engine of Application State
  - *HATEOAS*: Probably the world's worst acronym!
  - RESTful responses contain the links you need – just like HTML pages do

REST Principles (3)

- Stateless architecture
  - No HttpSession usage
  - GETs can be cached on URL
  - Requires clients to keep track of state
  - Part of what makes it scalable
  - Looser coupling between client and server
- HTTP headers and status codes communicate result to clients
  - All well-defined in HTTP Specification

Why REST?

- Benefits of REST
  - Every platform/language supports HTTP
    - Unlike for example SOAP + WS-* specs
  - Easy to support many different clients
    - Scripts, Browsers, Applications
  - Scalability
  - Support for redirect, caching, different representations, resource identification, …
  - Support for XML, but also other formats
    - JSON and Atom are popular choices

REST and Java: JAX-RS

JAX-RS

- JAX-RS is a Java EE 6 standard for building RESTful applications
  - Focuses on programmatic clients, not browsers
- Various implementations
  - Jersey (RI), RESTEasy, Restlet, CXF
  - All implementations provide Spring support
- Good option for full REST support using a standard
- No support for building clients in standard
  - Although some implementations do offer it

REST and Java: Spring-MVC

- Spring-MVC provides REST support as well
  - Since version 3.0
  - Using familiar and consistent programming model
  - Spring MVC does not implement JAX-RS
- Single web-application for everything
  - Traditional web-site: HTML, browsers
  - Programmatic client support (RESTful web applications, HTTP-based web services)
- RestTemplate for building programmatic clients in Java

Topics in this Session

- REST introduction
- **Spring MVC support for RESTful applications**
  - Request/Response Processing
  - Accessing Request/Response Data
  - Putting it all together
- RESTful Clients with the RestTemplate
- Advanced Topics

Spring-MVC and REST

- Will now extend Spring MVC to support REST
  - Map requests based on HTTP method
  - Define response status
  - Message Converters
  - Access request and response body data
  - Build valid Location URIs

spring Pivotal

Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
  - **Request/Response Processing**
  - Accessing Request/Response Data
  - Putting it all together
- RESTful Clients with the RestTemplate
- Advanced Topics

spring Pivotal

Request Mapping Based on HTTP Method

- Can map HTTP requests based on method
  - Allows same URL to be mapped to multiple methods
  - Often used for form-based controllers (GET & POST)
  - Essential to support RESTful resource URLs
    - incl. PUT and DELETE

```
@RequestMapping(path="/orders", method=RequestMethod.GET)
public void listOrders( ... ) {
   // find all Orders and add them to the model
}

@RequestMapping(path="/orders", method=RequestMethod.POST)
public void createOrder( ... ) {
   // process the order data from the request
}
```

spring Pivotal

Simpler Mapping Annotations

Spring 4.3

- Alternative handler mapping shortcuts
  - `@RequestMapping(path="/accounts", method=GET)`
  - Or `@GetMapping("/accounts");`

- Exist for each HTTP method
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`

spring Pivotal

HTTP Status Code Support

- Web apps just use a handful of status codes
  - Success: 200 OK
  - Redirect: 30x for Redirects
  - Client Error: 404 Not Found
  - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many additional codes
  - Created Successfully: 201
  - HTTP method not supported: 405
  - Cannot generate response body requested: 406
  - Request body not supported: 415

For a full list: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

spring Pivotal

@ResponseStatus

Spring 4.3 – Can place at class-level, applies to *all* methods

- To return a status code *other* than 200
  - Use `HttpStatus` enumerator
- **Note:** `@ResponseStatus` on ***void*** methods
  - No longer want to return a view name - *no View at all!*
  - Method returns a response with empty body (*no-content*)

```
@RequestMapping(path="/orders/{id}",
                method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(HttpServletRequest request) {
   Order order = getOrder(request);    // Extract from request
   orderService.updateOrder(order);
}
```

Can also set error response codes – see Advanced section

spring Pivotal

Accessing Request/Response Data

- *The Problem*
  - HTTP **`GET`** needs to return data in response body
  - HTTP **`POST`** and **`PUT`** receive data in request body
  - Typically XML or JSON
  - Want to deal in Java objects
  - Want to avoid converting between formats manually
- *The Solution*
  - Use *Marshaling* via dedicated *message-converters*
  - Annotate incoming data with **`@RequestBody`**
  - Annotate outgoing data with **`@ResponseBody`**

spring Pivotal

`HttpMessageConverter`

- Converts HTTP request/response body data
  - XML (using JAXP Source, JAXB2 mapped object, Jackson-Dataformat-XML*)
  - Jackson JSON*, GSON*Feed data* such as Atom/RSS
  - Google protocol buffers *
  - Form-based data
  - **`Byte[]`**, **`String`**, **`BufferedImage`**

* Requires 3rd party libraries on classpath

- **Must** enable otherwise no convertors defined at all!
  - Automatic with Spring Boot
  - `@EnableWebMvc` or `<mvc:annotation-driven/>`
  - Or define explicitly (allows you to register extra convertors)
    - Using **`WebMvcConfigurer`** or **`<mvc/>`**

spring Pivotal

@RequestBody

- To use converters for incoming request data
  - Correct converter chosen automatically
    - Based on content type of request
  - **updatedOrder** could be mapped from XML (with JAXB2) or from JSON (with Jackson)
    - Annotate Order class to help JAXB/Jackson work

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)   // 204
public void updateOrder(@RequestBody Order updatedOrder,
                        @PathVariable("id") long id) {
   // process updated order data and return empty response
   orderManager.updateOrder(id, updatedOrder);
}
```

@ResponseBody

- Use converters for response data by annotating return data with `@ResponseBody`
- Converter handles rendering a response
  - Again, no ViewResolver and View involved any more!

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)   // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
   // Order class is annotated with JAXB2's @XmlRootElement
   Order order= orderService.findOrderById(id);
   // results in XML response containing marshalled order:
   return order;
}
```

What Return Format? *Accept* Header

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)   // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderService.findOrderById(id);
}
```

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/xml, ...
…
```

```
HTTP/1.1 200 OK
Date: …
Content-Length: 1456
Content-Type: application/xml
<order id="123">
…
</order>
```

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json, ...
…
```

```
HTTP/1.1 200 OK
Date: …
Content-Length: 756
Content-Type: application/json
{
   "order": {"id": 123,
             "items": [ … ], … }
}
```

spring Pivotal

@RestController Simplification

Spring 4.0

```
@Controller
public class OrderController {
  @RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
  public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderService.findOrderById(id);
  }

  …
}
```

```
@RestController
public class OrderController {
  @RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
  public Order getOrder(@PathVariable("id") long id) {
     return orderService.findOrderById(id);
  }

  …
}
```

No need for **@ResponseBody** on **GET** methods

spring Pivotal

Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - **Accessing Request/Response Data**
  - Putting it all together
- RESTful Clients with the RestTemplate
- Advanced Topics

Accessing Request/Response Data

- Can explicitly inject `HttpServletRequest` (and `HttpServletResponse`)
  - But makes Controller methods hard to test
  - Consider Spring's `MockHttp`... classes
- Spring can *automatically* inject part of the request
  - `@RequestParam`, `@PathVariable`, `Principal`, `Locale`, `@Value`, `@RequestHeader`,
  - `@RequestBody`, `@ResponseBody`
- To perform REST we may also use
  - ***`HttpEntity`, `ResponseEntity`***

HttpEntity and ResponseEntity

- To build responses explicitly
  - Set headers, control content returned
  - Use `HttpEntity` or `ResponseEntity`

```
// Want to return a String as the response-body
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.TEXT_PLAIN);
HttpEntity<String> entity =
            new HttpEntity<String>("Hello Spring", headers);

// ResponseEntity (since Spring 4.1) supports a "fluent" API
ResponseEntity<String> response =
      ResponseEntity.ok()
                    .contentType(MediaType.TEXT_PLAIN)
                    .body("Hello Spring");
```


Setting Response Data

- Can use `HttpEntity` to generate a Response
  - Avoids use of `HttpServletResponse` (easier to test)

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
public HttpEntity<Order> getOrder(@RequestParam("id") long id) {
    String order = orderService.find(id);

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("Content-Type", "text/xml");

    return new HttpEntity<Order>(order, responseHeaders);
}
```

Response body type

Response body

Building URIs

- An HTTP POST typically returns location of newly created resource in the response header
- How to create a URI?
  - **UriComponentsBuilder**
    - Allows explicit creation of URI
    - *But* uses hard-coded URLs
  - **ServletUriComponentsBuilder**
    - Provides access to the URL that invoked the current controller method

Building URIs: `UriComponentsBuilder`

- Support for building URIs from templated strings
  - Escapes illegal characters – such as %20 for a space

Use of hard-coded URL *not* recommended

```
String templateUrl =
   "http://store.spring.io/orders/{orderId}/items/{itemId}";

URI location = UriComponentsBuilder.
                   fromHttpUrl(templateUrl).
                   buildAndExpand("123456","item A").
                   toUri();

return ResponseEntity.created(location).build();

// http://store.spring.io/orders/123456/items/item%20A
```

Convenient way to build POST response

ServletUriComponentsBuilder

- Use in a Controller method
  - Avoids hard-coding URL

Framework puts request URL in current thread – which builder can access

```
// Must be in a Controller method
//    Example:  POST /orders/{id}

URI location = ServletUriComponentsBuilder
      .fromCurrentRequestUri()
      .path("items/{itemId}")
      .buildAndExpand("item A")
      .toUri(); ;

return ResponseEntity.created(location).build();

       // http://.../items/item%20A
```

spring Pivotal

Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Accessing Request/Response Data
  - **Putting it all together**
- RESTful Clients with the RestTemplate
- Advanced Topics

spring Pivotal

Putting it all Together

- We have covered many new concepts
  - **`@ResponseStatus`**
  - HTTP Message Converters
  - **`@RequestBody`**, **`@ResponseBody`**
  - **`HttpEntity`**, **`ResponseEntity`**
  - **`UriComponentsBuilder`**
- Let's see some typical Controller implementations for each of **`GET`**, **`POST`**, **`PUT`** and **`DELETE`**
  - Useful reference when you are writing a new Controller
    - *And for the lab!*

spring Pivotal

Retrieving a Representation: GET

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/xml,
...
...
```

```
HTTP/1.1 200 OK
Date: …
Content-Length: 1456
Content-Type: application/xml

<order id="123">
  …
</order>
```

```
@GetMapping(path="/orders/{id}")
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
   return orderService.findOrderById(id);
}
```

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
```

spring Pivotal

Creating a new Resource: **POST** (1)

```
POST /store/orders/123/items
Host: shop.spring.io
Content-Type: application/xml

<item>
…
</item>
```

```
HTTP/1.1 201 Created
Date: …
Content-Length: 0
Location: http://shop.spring.io/
          store/orders/123/items/abc
```

- The most complicated to implement
  - *"201 Created"* requires *Location* header for new resource
  - Will need **`ServletUriComponentsBuilder`** *and* **`ResponseEntity`**

Creating a new Resource: **POST** (2)

```
@PostMapping(path="/orders/{id}/items")
@ResponseStatus(HttpStatus.CREATED)  // 201
public ResponseEntity<Void>
createItem(@PathVariable("id") long id, @RequestBody Item newItem) {
  // Add the new item to the order
  orderService.findOrderById(id).addItem(newItem);

  // Build the location URI of the new item
  URI location = ServletUriComponentsBuilder
        .fromCurrentRequestUri()
        .path("{itemId}")
        .buildAndExpand(newItem.getId())
        .toUri();

  return ResponseEntity.created(location).build();
}
```

Assume added id to item

```
@RequestMapping(path="/orders/...", method=RequestMethod.POST)
```

Updating existing Resource: **PUT**

```
PUT /store/orders/123/items/abc
Host: shop.spring.io
Content-Type: application/xml

<item>
…
</item>
```

```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```java
@PutMapping(path="/orders/{orderId}/items/{itemId}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void updateItem(@PathVariable("orderId") long orderId,
                       @PathVariable("itemId") String itemId,
                       @RequestBody Item item) {
   orderService.findOrderById(orderId).updateItem(itemId, item);
}
```

```java
@RequestMapping(path="/orders/...", method=RequestMethod.PUT)
```

Deleting a Resource: **DELETE**

```
DELETE /store/orders/123/items/abc
Host: shop.spring.io
...
```

```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```java
@DeleteMapping(path="/orders/{orderId}/items/{itemId}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void deleteItem(@PathVariable("orderId") long orderId,
                       @PathVariable("itemId") String itemId) {
   orderService.findOrderById(orderId).deleteItem(itemId);
}
```

```java
@RequestMapping(path="/orders/...", method=RequestMethod.DELETE)
```

Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
- **RESTful Clients with the RestTemplate**
- Advanced Topics

spring Pivotal

RestTemplate

- Provides access to RESTful services
  - Supports all the HTTP methods

HTTP Method	RestTemplate Method
DELETE	delete(String url, Object… urlVariables)
GET	getForObject(String url, Class<T> responseType, Object… urlVariables)
HEAD	headForHeaders(String url, Object… urlVariables)
OPTIONS	optionsForAllow(String url, Object… urlVariables)
POST	postForLocation(String url, Object request, Object… urlVariables)
	postForObject(String url, Object request, Class<T> responseType, Object… uriVariables)
PUT	put(String url, Object request, Object… urlVariables)

spring Pivotal

Defining a RestTemplate

- Just call constructor in your code
  - Setups default *HttpMessageConverters* internally
    - Same as on the server, depending on classpath

```
RestTemplate template = new RestTemplate();
```

RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";

// GET all order items for an existing order with ID 1:
OrderItem[] items =
        template.getForObject(uri, OrderItem[].class, "1");

// POST to create a new item
OrderItem item = // create item object
URI itemLocation = template.postForLocation(uri, item, "1");

// PUT to update the item
item.setAmount(2);
template.put(itemLocation, item);

// DELETE to remove that item again
template.delete(itemLocation);
```

{id} = 1

{id} = 1

Also supports HttpEntity, for example to set HTTP request headers

Summary

- REST is an architectural style that can be applied to HTTP-based applications
  - Useful for supporting diverse clients and building highly scalable systems
- Spring-MVC adds REST support using a familiar programming model (but *without* Views)
  - **`@ResponseStatus`**, **`@RequestBody`**, **`@ResponseBody`**
  - **`HttpEntity`**, **`ResponseEntity`**, **`UriComponentsBuilder`**
  - HTTP Message Converters
- Clients use *RestTemplate* to access RESTful servers

Lab

Restful applications with Spring MVC

Coming Up: More on Spring MVC REST
Introduction to Spring HATEOAS

Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
- RESTful clients with the RestTemplate
- **Advanced Topics**
  - **More on Spring REST**
  - Spring HATEOAS

spring Pivotal

@ResponseStatus & Exceptions

- Can also annotate exception classes with this
  - Given status code used when an unhandled exception is thrown from *any* controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND)  // 404
public class OrderNotFoundException extends RuntimeException {
   …
}
```

```
@GetMapping(value="/orders/{id}")
public String showOrder(@PathVariable("id") long id, Model model) {
   Order order = orderService.findOrderById(id);
   if (order == null) throw new OrderNotFoundException(id);
   model.addAttribute(order);
   return "orderDetail";
}
```

NOTE: this is *not* a RESTful method, it returns a view-name.

spring Pivotal

@ExceptionHandler

- For existing exceptions you cannot annotate, use @ExceptionHandler method on controller
  - Method signature similar to request handling method
  - Also supports @ResponseStatus

```
@ResponseStatus(HttpStatus.CONFLICT)  // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception, response, etc. as method params
}
```

Spring MVC offers several ways to handle exceptions, for more details see http://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

spring Pivotal

Mixing Views *and* Annotations - 1

- REST methods do not return HTML, PDF, …
  - No message converter
  - Views better for presentation-rich representations
- How to distinguish between representations?
  - Or a RESTful POST from a HTML form submission
- Use *produces* and *consumes* attributes

```
@GetMapping(value="/orders/{id}", produces = {"application/json"})

@PostMapping(value="/orders/{id}", consumes = {"application/json"})
```

spring Pivotal

Mixing Views *and* Annotations – 2

- Need two methods on controller for same URL
  - One uses a converter, the other a View
  - Identify using *produces* attribute
- Recommendation
  - Mark RESTful method with *produces*
    - To avoid returning XML to normal browser request
  - Call RESTful method from View method
    - Implement all data-access logic *once* in RESTful method

spring Pivotal

Mixing Views *and* Annotations - 3

- Recommendation

RESTful Method

```
@GetMapping(path="/orders/{id}",
            produces = {"application/json","application/xml"})
@ResponseStatus(HttpStatus.OK)  // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
   // Access data here ...
   return orderService.findOrderById(id);
}

@GetMapping(path="/orders/{id}")
public String getOrder(Model model, @PathVariable("id") long id) {
   // Invoke RESTful method, use result to populate model
   model.addAttribute(getOrder(id));
   return "orderDetails"; // View name
}
```

View method calls RESTful method

View Method

spring Pivotal

HttpMethodFilter

- HTML forms do not support PUT or DELETE
  - Not even in HTML 5
- So use a POST
  - Put PUT or DELETE in a *hidden* form field
- Deploy a special filter to intercept the message
  - Restores the HTTP method you wanted to send
  - Appear to Spring MVC as a PUT or a DELETE

See *HttpMethodFilter* in online documentation

spring Pivotal

Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- **Advanced Topics**
  - More on Spring REST
  - **Spring HATEOAS**

spring Pivotal

HATEOAS - Concepts

- REST clients need *no* prior knowledge about how to interact with a particular application/server
  – SOAP web-services need a WSDL
  – SOA processes require a fixed interface defined using interface description language (IDL)
- Clients interact entirely through hypermedia
  – Provided dynamically by application servers
- Serves to *decouple* client and server
  – Allows the server to evolve functionality independently
  – Unique compared to other architectures

HATEOAS Account Example

Spring HATEOAS provides an API for generating these links in MVC Controller responses

```
<account>
   <account-number>12345</account-number>
   <balance currency="usd">100.00</balance>
   <link rel="self" href="/account/12345" />
   <link rel="deposit" href="/account/12345/deposits" />
   <link rel="withdraw" href="/account/12345/withdrawls" />
   <link rel="transfer" href="/account/12345/transfers" />
   <link rel="close" href="/account/12345/close" />
</account>
```

Note: links *change* as *state* changes

```
<account>
   <account-number>12345</account-number>
   <balance currency="usd">-25.00</balance>
   <link rel="self" href="/account/12345" />
   <link rel="deposit" href="/account/12345/deposits" />
</account>
```

There is no standard for links yet. This example uses the link style from the *Hypertext Application Language* (HAL), one possible representation

From: http://restcookbook.com/Basics/hateoas/

Managing Links

- Use Link class
  - Holds an href and a rel (relationship)
  - Self implies the current resource
  - Link builder derives URL from Controller mappings

```
// A link can be built with a relationship name
//   Use withSelfRel() for a self link
Link link = ControllerLinkBuilder.linkTo(AccountController.class)
    .slash(accountId).slash("transfer")).withRel("transfer");

link.getRel();    // => transfer
link.getHref();   // => http://.../account/12345/transfer
```


Converting to a Resource

- Wrap return value of REST method in Resource
  - Converted by @ResponseBody to XML/JSON with links
    - Only HAL supported currently

```
@Controller
@EnableHypermediaSupport(type=HypermediaType.HAL)
public class OrderController {

   @GetMapping(value="/orders/{id}")
   public @ResponseBody Resource<Order>
                        getOrder(@PathVariable("id") long id) {
      Links[] = ...;  // Some links (see previous slide)
      return new Resource<Order>
           (orderService.findOrderById(id), links);
   }
}
```

Spring HATEOAS

- Spring Data sub-project for REST
  - For generating links in RESTful responses
  - Supports ATOM (newsfeed XML) and HAL (Hypertext Application Language) links
  - Many other features besides examples shown here

- For more information see
  - http://projects.spring.io/spring-hateoas/
  - http://spring.io/guides/gs/rest-hateoas/

spring Pivotal

Microservices with Spring

Building Cloud Native Applications

Introduction to Spring Cloud

spring Pivotal

Roadmap

- **What is Microservices Architecture?**
- Pros and Cons of Microservices
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System

spring Pivotal

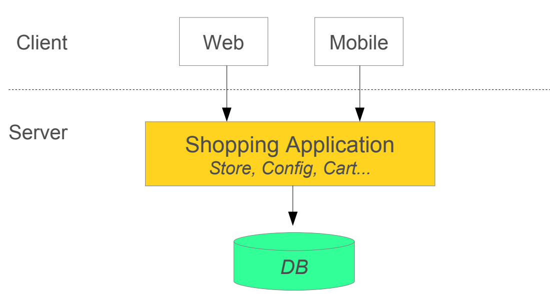
Introduction

- “Microservices” is not a new word
  - Term coined in 2005 by Dr Peter Rodgers
    - Then called “*micro web services*” and based on SOAP
  - Term started to become popular since 2010
    - Proposed by a group of architects in Venice during 2011
    - Used in 2012 in a presentation from James Lewis
    - Meanwhile Adrian Cockcroft (Netflix), was describing this approach as "fine grained SOA"

http://martinfowler.com/articles/microservices.html

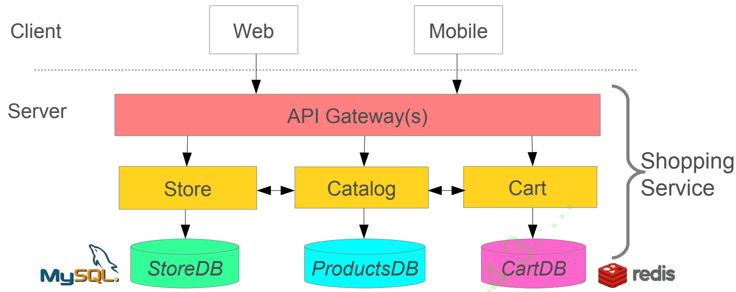
Without Microservices

- Using a “monolith” architecture
  - All-in-One application



With Microservices

- Main application has been divided in a set of sub-applications
  - Called microservices



Roadmap

- What is Microservices Architecture?
- **Pros and Cons of Microservices**
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System

Core Spring Concepts
Applied to Application Architecture

- Spring enables *separation-of-concerns*
  - *Loose Coupling:* Effect of change is isolated
  - *Tight Cohesion:* Code performs a single well-defined task
- Microservices exhibit the same strengths
  - *Loose Coupling*
    - Applications are built from collaborating services (processes)
    - Can change independently so long as protocols unchanged
  - *Tight Cohesion*
    - An application (service) that deals with a *single* view of data
    - Also known as "Bounded Contexts" (*Domain-Driven Design*)

Microservice Benefits

- Smaller code base is easy to maintain
- Easy to scale
  - Scale individual component
- Technology diversity
  - Mix libraries, frameworks, data storage, languages
- Fault Isolation
  - Component failure should not bring whole system down
- Better support for smaller, parallel teams
- Independent deployment

Microservice Challenges

- Difficult to achieve strong consistency across services
  - ACID transactions *do not* span multiple processes
  - Eventual consistency, Compensating transactions
- Distributed system
  - Harder to debug/trace
  - Greater need for end-to-end testing
  - Expect, test for and handle the failure of any process
  - More components to maintain: redundancy, HA
- Typically requires “cultural” change (*Dev Ops*)
  - How applications are developed and deployed
  - Devs and Ops working *together*, even on same team!

Use a Platform to Support This

- Platforms* like *Pivotal Cloud Foundry* aid deployment
  - Easily run, scale, monitor and recover multiple processes
  - Run up a complete dev system for end-to-end testing
- Support for
  - Continuous Deployment
  - Rolling upgrades of new versions of code
    - Also termed: Blue/Green or Canary rollout
    - Quick rollback in case of defects
  - Running multiple versions of same service at same time
    - Makes migration easier for downstream projects

*Platform as a Service (PaaS)

Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- **Developing Microservices**
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System

Microservice Infrastructure

We cover these today

- Multiple processes working together
- Issues that now arise:
  - How do they find each other? Service Discovery
  - How do we decide which instance to use? Client-side Load Balancing
  - What happens if a microservice is not responding Fault Tolerance
  - How to we control access? OAuth, ...
  - How do they communicate? Messaging REST
  - To just name a few!

Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- Managing Microservices
- **Tooling: Spring, Spring Cloud, Netflix**
- Building a Simple Microservice System

spring Pivotal

Microservices made-easy by Spring

- Setup a new service using Spring Boot
- Expose resources via a RestController
- Consume remote services using RestTemplate
- Will leverage capabilities from *Spring Cloud Project*

spring Pivotal

What is Spring Cloud?

- Building blocks for Cloud and Microservice applications
  - Microservices Infrastructure
    - Provides useful services such as Service Discovery, Configuration Server and Monitoring
    - Several based on other Open Source projects
      - Netfix OSS, HashiCorp's Consul, ...
  - Platform Support
    - Access platform-specific information and services
      - Available for Cloud Foundry, AWS and Heroku
  - Uses Spring Boot style starters
    - *Requires Spring Boot to work*

HASHICORP
NETFLIX | OSS

spring http://projects.spring.io/spring-cloud Pivotal

Spring Cloud Ecosystem

** Today we only have time to look at the Netflix project*

Spring Cloud

IaaS Integration	*Dynamic Reconfiguration*	***Service Infrastructure***	*Utilities*	*Data Ingestion*
Spring Cloud Connectors	Spring Cloud Config	**Spring Cloud Netflix ***	Spring Cloud Security	Spring Cloud Data Flow
Spring Cloud Cloud Foundry	Spring Cloud Bus	Spring Cloud Consul	Spring Cloud CLI	Spring Cloud Modules
Spring Cloud For AWS		Spring Cloud Cluster	Spring Cloud Stream	

Spring Cloud is at http://projects.spring.io/spring-cloud/

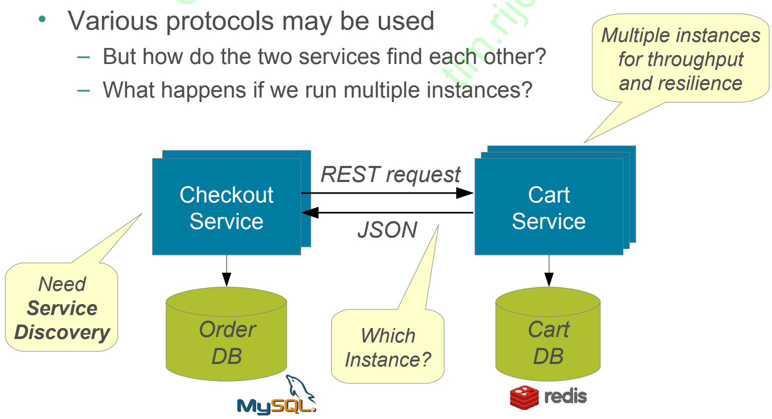
spring Pivotal

Spring Cloud Usage Examples

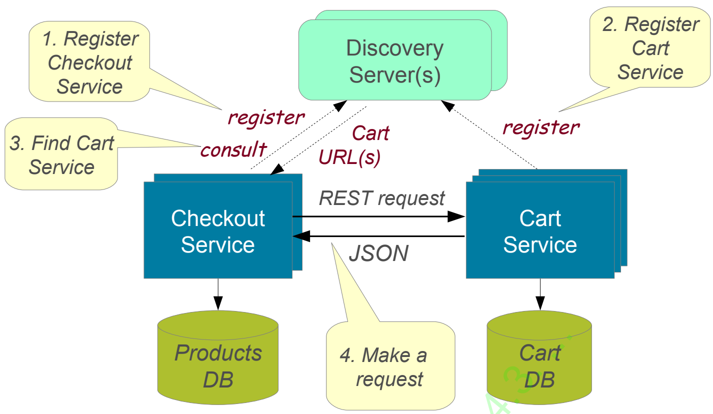
- There are *many* use-cases supported by Spring Cloud
  - Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Data Ingestion
- Today we concentrate on *microservices* support
  - **Service Discovery**
    - How do the services find each other?
  - **Client-side Load Balancing**
    - Each service typically deployed as multiple instances
      - For fault-tolerance and load-sharing
    - How do we decide which service *instance* to use?

Why We Need Service Discovery – 1

- Various protocols may be used
  - But how do the two services find each other?
  - What happens if we run multiple instances?



Why We Need Service Discovery – 2



Implementing Service Discovery

- Spring Cloud supports several
  - We will use Eureka
    - Created by Netflix
  - Consul.io is another option
    - Hashicorp project (inventors of Vagrant)
- Spring Cloud makes it easy
  - To utilize either of those servers
  - Hiding their internal complexity

See also: http://spring.io/blog/2015/07/14/microservices-with-spring

Implementing Client-Side Load Balancing

- Discovery server may return the location of *multiple* instances
  - *Recall:* multiple instances for resilience and load-sharing
  - Client needs to pick one
- We will use Netflix Ribbon
  - Provides several algorithms for client-side load-balancing
- Spring provides a smart RestTemplate
  - Service-discovery and load-balancing built-in
  - @LoadBalanced

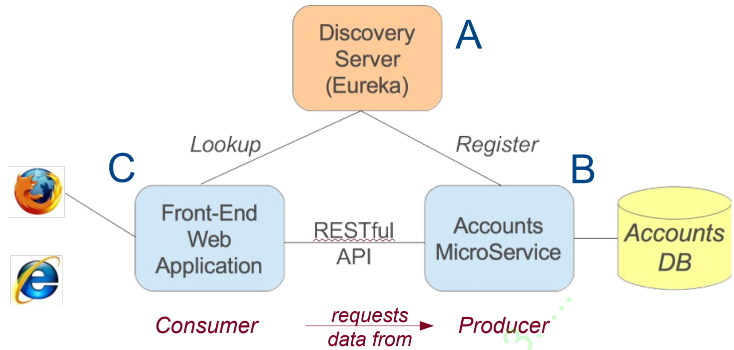
spring Pivotal

Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- **Building a Simple Microservice System**

spring Pivotal

Our Simple Microservice System



Building our Simple Microservice System

- A) Run a Discovery Service
  – We will see how to create a Eureka Discovery Service
- B) Run a Microservice (the Producer)
  – Ensure it registers itself with the Discovery Service
  – Registers its *logical* service name with A
- C) How do Microservice Consumers find service B?
  – Discovery client using a "smart" RestTemplate
    - Spring performs service lookup for you
    - Uses *logical* service names in URLs

Maven Dependencies

```xml
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR4</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
     <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter</artifactId>
     </dependency>
     <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
</dependencies>
```

Parent

"Release train" = a consolidated set of releases

Spring Cloud

Access Eureka registry server

pom.xml

Dependencies for A, B & C. Spring Cloud is based on Spring Boot

(A) Eureka Server using Spring Cloud

- All you need to implement your own registry service!

application.yml

```yaml
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:  # Not a client
    registerWithEureka: false
    fetchRegistry: false
```

```java
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

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

main.java

pom.xml

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

(B) Accounts Producer Microservice

Performs Service Registration

- Microservice declares itself as an available service
  - Using *@EnableDiscoveryClient*
  - *Registers* using its *application name*

```
@SpringBootApplication
@EnableDiscoveryClient
public class AccountsApplication {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

```
spring:
  application:
    name: accounts-microservice
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Service name

Eureka Server URL

spring Pivotal

(C) Consumer Service – Step 1

Enable our consumer to find the producer

- Same annotation *also* allows service *lookup*

Same annotation as (B) – allows us to query Discovery Server to *find* microservices

```
@SpringBootApplication
@EnableDiscoveryClient
public class FrontEndApplication {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    // Will use this template to access the microservice
    //   Spring will enhance this to do service discovery
    @Bean @LoadBalanced
    public RestTemplate restTemplate() {
        return new  RestTemplate();
    }
}
```

spring Pivotal

(C) Consumer Service – Step 2

Let our consumer use the producer

```java
@Service
public class RemoteAccountManager implements AccountService {

    // Spring injects the “smart” service-aware template
    // defined on previous slide
    // It performs a load-balanced lookup (see next slide)
    @Autowired
    @LoadBalanced
    RestTemplate restTemplate;

    public Account findAccount(String id) {
        // Fetch data
        return restTemplate.getForObject
            ("http://accounts-microservice/accounts/{id}",
                                Account.class, id);
    }
}
```

Service name

Load Balanced RestTemplate

- Create using @LoadBalanced – an @Qualifier
  - Spring enhances it to do service lookup & load-balancing

```java
@Bean @LoadBalanced
public RestTemplate restTemplate() {
    return new  RestTemplate();
}
```

- Must inject using same qualifier
  - If there are multiple RestTemplates you get the right one
  - Can be used to access multiple microservices

```java
@Autowired
@LoadBalanced
RestTemplate restTemplate;
```


Load Balancing with Ribbon

- Our "smart" RestTemplate automatically integrates *two* Netflix utilities
  - "Eureka" service-discovery
  - "Ribbon" client-side load-balancer
- End result
  - Eureka returns the URL of all available instances
  - Ribbon determines the best available service to use
- Just inject the load-balanced **`RestTemplate`**
  - Automatic lookup by *logical* service-name

spring Pivotal

Spring Cloud Resources

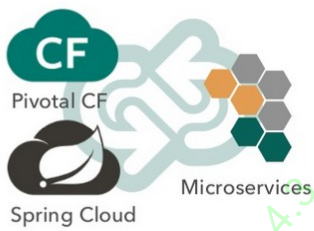
- We have only covered a *few* Spring Cloud features
  - Project Home page
    - http://projects.spring.io/spring-cloud
  - Matt Stine's *free* book on Cloud Native Architectures
    - https://pivotal.io/platform/migrating-to-cloud-native-application-architectures-ebook
  - Spring Blog article
    - https://spring.io/blog/2015/07/14/microservices-with-spring

Consider taking our ***Spring Cloud Services*** course
See: *http://pivotal.io/academy*

spring Pivotal

Summary

- After completing this lesson, you should have learned:
  - What is a Microservice Architecture?
  - Advantages and Challenges of Microservices
  - A little bit about Spring Cloud projects



Finishing Up

Course Completed

What's Next?

spring Pivotal

What's Next

- Congratulations, we've finished the course
- What to do next?
  - Certification
  - Other courses
  - Resources
  - Evaluation
- Check-out optional sections on Remoting and SOAP web-services

spring Pivotal

Certification

- Computer-based exam
  - 50 multiple-choice questions
  - 90 minutes
  - Passing score: 76% (38 questions answered successfully)

- Preparation
  - Review all the slides
  - Redo the labs
  - Study Guide available online

    https://pivotal.io/academy/certification

spring Pivotal

Certification: Questions

Sample question

- Statements
  a. An application context holds Spring beans
  b. An application context manages bean scope
  c. Spring provides many types of application context

- Pick the correct response:
  1. Only a. is correct
  2. Both a. and c. are correct
  3. All are correct
  4. None are correct

spring Pivotal

Certification: Logistics

Voucher is valid for 3 months – *do it soon!*

- Where?
  - Online at PSI (Innovative Exams)
    - https://www.examslocal.com
- How?
  - You should receive a certification voucher by email
  - Register/sign-in and book an exam using the voucher
    - http://it.psionline.com/exam-faqs/pivotal-faq
  - Take the test from *any* location
- For more information, email
  - education@pivotal.io

Other courses

- Many courses available
  - Spring Web
  - Enterprise Spring
  - Spring Boot Developer
  - Spring Cloud Services (Microservices with Spring)
  - Pivotal Cloud Foundry Developer
  - Pivotal Cloud Foundry Administration
  - Big Data and Analytics, Hadoop, Gemfire, …
- See https://pivotal.io/academy

Spring Web

- Four-day workshop
- Making the most of Spring in the web layer
  - Spring MVC using Spring Boot
  - REST using MVC and AJAX, CORS
  - Security of Web applications
  - Mock MVC testing framework
  - Spring Web Sockets
- Spring Web Application Developer certification

Enterprise Spring

- Building loosely coupled event-driven architectures
  - Separate processing, communications & integration
- 4 day course covering
  - Tasks, Scheduling and Concurrency
  - Advanced transaction management
  - REST Web Services with Spring MVC
  - Spring Integration
  - Spring Batch
  - Data Ingestion and Transformation

Spring Boot Developer

- 2 day workshop covering
  - Getting started with Spring Boot
  - Spring Boot CLI
  - Configuration, auto-configuration and profiles
  - Web development and REST
  - Data Access: JDBC, JPA, Spring Data, NoSQL
  - Testing
  - Security, Messaging
  - Deployment, Metrics, Actuator
  - Microservices

spring Pivotal

Spring Cloud Services
Microservices With Spring

- Course topics:
  - Introduction to Spring Boot
    - Underpins all Spring Cloud projects
  - Pushing Applications to a PaaS
    - Using Pivotal Cloud Foundry
  - What are Microservices?
    - Architecting a microservices solution
  - Cloud infrastructure services and Netflix OSS
    - Service Configuration
    - Service Registration
    - Load-balancing and fault tolerance
    - Security using OAuth

spring Pivotal

Cloud Foundry Developer

CLOUDFOUNDRY

- 3 day course covering
  - Application deployment to Cloud Foundry
    - Deployment using *cf* tool or an IDE
    - Using the PCF Application Manager
  - Cloud Foundry Concepts
    - Logging, Continuous Integration, Monitoring
    - Accessing and defining Services
    - Using and customizing Buildpacks
  - Design considerations: “12 Factor”
    - JVM application specifics, using Spring Cloud

Formerly: Developing Applications with Cloud Foundry

spring Pivotal

Cloud Foundry Administration

CLOUDFOUNDRY

- 5 day course covering
  - Application deployment to Cloud Foundry
    - Logging, scaling, blue-green deployments
  - “Day 1 Operations”
    - Installation of PCF Ops Manager and Elastic Runtime
    - Configuring users, roles, and quotas
    - Capturing and reading logs
  - “Day 2 Operations”
    - Backing up and restoring an installation
    - Using BOSH
    - Upgrading Ops Manager and tiles.

spring Pivotal

Pivotal Support Offerings

- Global organization provides 24x7 support
  - How to Register: http://tinyurl.com/piv-support

- Premium and Developer support offerings:
  - http://www.pivotal.io/support/offerings
  - http://www.pivotal.io/support/oss
  - Both Pivotal App Suite *and* Open Source products
- Support Portal: https://support.pivotal.io
  - Community forums, Knowledge Base, Product documents

Pivotal Consulting

- Custom consulting engagement?
  - Contact us to arrange it
    - http://www.pivotal.io/contact/spring-support
    - Even if you don't have a support contract!
- Pivotal Labs
  - Agile development experts
  - Mentoring: design, development and product management
    - http://www.pivotal.io/agile
    - http://pivotallabs.com

Resources

- The Spring reference documentation
  - http://spring.io/docs
  - http://projects.spring.io/spring-boot
  - http://projects.spring.io/spring-data
  - http://projects.spring.io/spring-security
  - http://projects.spring.io/spring-cloud
- The official technical blog
  - http://spring.io/blog
- Stack Overflow – Active Spring Forums
  - http://stackoverflow.com

spring Pivotal

Resources (2)

- You can register issues on our Jira repository
  - https://jira.spring.io
- The source code is available here
  - https://github.com/spring-projects/spring-framework

spring Pivotal

Thank You!

- We hope you enjoyed the course
- Please fill out the evaluation form
  - Americas: http://tinyurl.com/usa-eval
  - EMEA: http://tinyurl.com/emea-eval
  - Asia-Pac: http://tinyurl.com/apj-eval
- Once you've done, login to *Pivotal Academy*
  - You can download your Attendance Certificate

***If** your course is registered at Pivotal Academy*

Don't forget the optional sections

XML Dependency Injection

Advanced Features & Best Practices

Techniques for Creating Reusable and Concise Bean Definitions

Topics in this session

- **Singletons and Factory Beans**
- Constructor Arguments
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

Advanced Bean Instantiation

- Four techniques:
  - @Bean method in @Configuration class
    - 100% Java code available, write whatever code you need
  - Beans implementing Spring's FactoryBean interface
  - Use XML factory-method attribute for Singletons
  - Define your own factories as Spring Beans in XML

Using a Java Singleton

- How can Spring instantiate the following?
  - Classes with private constructors (such as Singleton pattern below)

```
public class AccountServiceSingleton implements  AccountService {
   private static AccountServiceSingleton inst = new AccountServiceSingleton();

   private AccountServiceSingleton() { ... }

   public static AccountService getInstance() {
      // ...
      return inst;
   }
}
```

The factory-method Attribute

- Non-intrusive
  - Useful for existing Singletons or Factories

```
public class AccountServiceSingleton implements AccountService {
   …
   public static AccountService getInstance() { // …  }
}
```

```
<bean id="accountService" class="com.acme.AccountServiceSingleton"
      factory-method="getInstance" />
```

Spring configuration

```
AccountService service1 = (AccountService) context.getBean("accountService");
AccountService service2 = (AccountService) context.getBean("accountService");
```

Test class

Spring uses getInstance() method – so *service1* and *service2* point to the *same* object

Using Your Own Factories

- Spring allows one bean to create another
  - Create an instance of your factory as a bean
  - Use it to create another bean

Spring configuration

```
<bean id="accountServiceFactory" class="com.acme.AccountServiceFactory">
  <!-- any constructor-arg or property elements you need -->
</bean

<bean id="accountService" factory-bean="accountServiceFactory"
                          factory-method="create" />
```

The *class* attribute is *illegal* here
Will be determined by the factory

Topics in this session

- Singletons and Factory Beans
- **Constructor Arguments**
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

spring Pivotal

More on Constructor Args

- Constructor args matched by type
  - <constructor-arg> elements can be in *any* order
  - When ambiguous: indicate order with *index*

Both are Strings

```
class MailService {
    public MailService(String username, String email) { … }
```

```
<bean name="example" class="com.app.MailService">
  <constructor-arg index="0" value="foo"/>
  <constructor-arg index="1" value="foo@foo.com"/>
</bean>
```

Index from *zero*

spring Pivotal

Using Constructor Types

- Can also specify the type
  - Typically when class has multiple ambiguous constructors

```
class MailService {
    public MailService(String username) { … }
    public MailService(int maxMessages) { … }
```

```
<bean name="example" class="com.app.MailService">
  <constructor-arg type="int" value="2000"/>
</bean>
```

Is a String in XML

Force use of second consturctor
Without *type*, Spring passes “2000” as the username

spring Pivotal

Using Constructor Names

- Constructor args can have names for matching
- Must be using Java 8 or later
  - OR: Need to compile with debug-symbols enabled
  - OR: Use **`@java.beans.ConstructorProperties`**

```
class MailService {
    @ConstructorProperties( { "username", "email" } )
    public MailService(String username, String email) { … }
```

Specify arg names *in order*

```
<bean name="example" class="com.app.MailService">
  <constructor-arg name="username" value="foo"/>
  <constructor-arg name="email" value="foo@foo.com"/>
</bean>
```

No *index* needed

spring Pivotal

Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- **'p' and 'c' namespaces**
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

The *c* and *p* namespaces

- Before

```
<bean id="transferService" class="com.acme.BankServiceImpl">
   <constructor-arg name="bankRepository" ref="bankRepository"  />
   <property  name="accountService" ref="accountService" />
   <property  name="customerService" ref="customerService" />
</bean>
```

- After

```
<bean id="transferService" class="com.acme.BankServiceImpl"
   c:bankRepository-ref="bankRepository"
   p:accountService-ref="accountService"
   p:customer-service-ref="customerService"  />
```

Use camel case *or* hyphens

c namespace is newer, introduced in Spring 3.1

The c and p namespaces

- *c* and *p* namespaces should be declared on top
  - Use '-ref' suffix for references

Namespace declaration

```
<beans xmlns:c="http://www.springframework.org/schema/c"
          xmlns:p="http://www.springframework.org/schema/p"
            ...>

    <bean id="transferService" class="com.acme.ServiceImpl"
                p:url="jdbc://..." p:service-ref="otherService"  />
</beans>
```

Inject *value* for property 'url'

Inject *reference* to bean 'otherService'

No schemaLocation needed

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
        xmlns:c="http://www.springframework.org/schema/c"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">

       <!-- ... -->

</beans>
```

p and *c* namespace definitions

no extra *schemaLocation* entry required (no xsd)

'c' and 'p' Pros and Cons

- Pros
  - More concise
  - Well supported in STS
    - CTRL+space works well
- Cons
  - Less widely known than the usual XML configuration syntax

Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- **Using Bean definition inheritance**
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

Bean Definition Inheritance (1)

- Sometimes several beans need to be configured in the same way
- Use bean definition inheritance to define the common configuration once
  - Inherit it where needed

Without Bean Definition Inheritance

```
<beans>
  <bean id="pool-A" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-a/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>

  <bean id="pool-B" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-b/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>

  <bean id="pool-C" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-c/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>
</beans>
```

Can you find the duplication?

Abstract Parent bean

```xml
<beans>
   <bean id="abstractPool"
      class="org.apache.commons.dbcp.BasicDataSource" abstract="true">
     <property name="user" value="moneytransfer-app" />
   </bean>

   <bean id="pool-A" parent="abstractPool">
     <property name="URL" value="jdbc:postgresql://server-a/transfer" />
   </bean>
   <bean id="pool-B" parent="abstractPool">
     <property name="URL" value="jdbc:postgresql://server-b/transfer" />
   </bean>
   <bean id="pool-C" parent="abstractPool">
     <property name="URL" value="jdbc:postgresql://server-c/transfer" />
     <property name="user" value="bank-app" />
   </bean>
</beans>
```

Will not be instantiated

Can override

Each pool inherits its *parent* configuration

Default Parent Bean

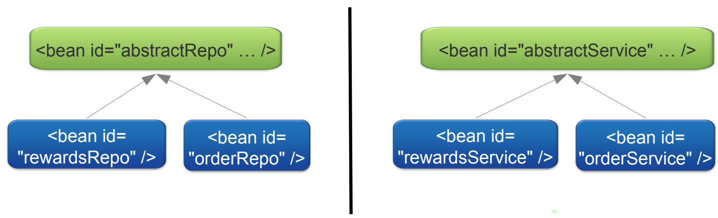
```xml
<beans>

   <bean id="defaultPool" class="org.apache.commons.dbcp.BasicDataSource">
     <property name="URL" value="jdbc:postgresql://server-a/transfer" />
     <property name="user" value="moneytransfer-app" />
   </bean>

   <bean id="pool-B" parent="defaultPool">
     <property name="URL" value="jdbc:postgresql://server-b/transfer" />
   </bean>

   <bean id="pool-C" parent="defaultPool" class="example.SomeOtherPool">
     <property name="URL"
         value="jdbc:postgresql://server-c/transfer" />
   </bean>

</beans>
```

Overrides URL property

Overrides class as well

Inheritance for service and repository beans

- Bean inheritance commonly used for definition of Repository (or DAO) beans and (less often) Services



Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- **Inner Beans**
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

Inner beans

- Inner bean only visible from surrounding bean

```
<bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
   <property name="benefitAvailabilityPolicy">
      <bean class="rewards...DefaultBenefitAvailabilityPolicyFactory" />
   </property>
</bean>
```

No bean id

- Cannot be accessed from the applicationContext

```
applicationContext.getBean(RestaurantRepository.class);
applicationContext.getBean(DefaultBenefitAvailabilityPolicyFactory.class);
```

OK

NoSuchBeanDefinitionException!!

spring Pivotal

Without an Inner Bean

```
<beans>

   <bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
      <property name="dataSource" ref="dataSource" />
      <property name="benefitAvailabilityPolicyFactory" ref="factory" />
   </bean>

   <bean id="factory"
      class="rewards.internal.restaurant.availability.
            DefaultBenefitAvailabilityPolicyFactory">
      <constructor-arg ref="rewardHistoryService" />
   </bean>
   …
</beans>
```

Can be referenced by other beans
(even if it should not be)

spring Pivotal

With an Inner Bean

```
<beans>

  <bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
    <property name="dataSource" ref="dataSource" />
    <property name="benefitAvailabilityPolicyFactory">
      <bean class="rewards.internal.restaurant.availability.
                 DefaultBenefitAvailabilityPolicyFactory">
        <constructor-arg ref="rewardHistoryService" />
      </bean>
    </property>
  </bean>
  …
</beans>
```

Inner bean has no id (it is anonymous)
Cannot be referenced outside this scope

spring Pivotal

Multiple Levels of Nesting

```
<beans>
  <bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
    <property name="dataSource" ref="dataSource" />
    <property name="benefitAvailabilityPolicyFactory">
      <bean class="rewards.internal.restaurant.availability.
                 DefaultBenefitAvailabilityPolicyFactory">
        <constructor-arg>
          <bean class="rewards.internal.rewards.JdbcRewardHistory">
            <property name="dataSource" ref="dataSource" />
          </bean>
        </constructor-arg>
      </bean>
    </property>
  </bean>
</beans>
```

spring Pivotal

Inner Beans: pros and cons

- Pros
  – You only expose what needs to be exposed
  – Very commonly used technique in online examples
- Cons
  – Can be harder to read
  – Avoid really deep nesting
- General recommendation
  – Use them when it makes sense
    - As for inner classes in Java
    - Complex "infrastructure beans" configuration

Lab (Optional)

Using Bean Definition Inheritance, Property Placeholders and Namespaces

Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- More on Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - **SpEL**, Autowiring, Collections

Spring Expression Language

- Can also be used in bean XML files
  - Same syntax that you have seen with @Value
  - Expressions in {} preceded by #
- Recall:
  - Can access System properties and environment
  - Properties of Spring beans

SpEL examples – XML

```
<bean id="rewardsDb" class="com.acme.RewardsTestDatabase">
      <property name="keyGenerator"
                value="#{strategyBean.databaseKeyGenerator}" />
</bean>

<bean id="strategyBean" class="com.acme.DefaultStrategies">
   <property name="databaseKeyGenerator" ref="myKeyGenerator"/>
</bean>

<bean id="taxCalculator" class="com.acme.TaxCalculator">
  <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>
</bean>
```

Can refer a nested property

Equivalent to System.getProperty(...)

SpEL Examples – Other Spring Projects

- In Spring Security

```
<security:intercept-url pattern="/accounts/**"
           access="isAuthenticated() and hasIpAddress('192.168.1.0/24')" />
```

- In Spring Batch

```
<bean id="flatFileItemReader" scope="step"
     class="org.springframework.batch.item.file.FlatFileItemReader">
   <property name="resource" value="#{jobParameters['input.file.name'] }" />
</bean>
```

Spring Security will be discussed later in this course. *Spring Batch* is part of the "Spring Enterprise" course

Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  – SpEL, **Autowiring**, Collections

Autowiring in XML

- XML had automatic wiring (setting) of dependencies before @Autowired – it's where the name comes from
- Can select *byType* or *byName* or *byConstructor*
  – *Cannot autowire both properties **and** constructor-args*
  – *Is inherently confusing and limited due to this difference*

```
<!-- Autowire properties (setters) by type matching just like @Autowired -->
<bean id="rewardsDb" autowire="byType" … />
<!-- Autowire properties by name – just like @Resource -->
<bean id="accountManager" autowire="byName"/  … />
<!-- Autowire constructors only by type – just like @Autowired -->
<bean id="accountManager" autowire="byConstructor" … />
```

Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - SpEL, Autowiring, **Collections**

spring Pivotal

beans and *util* collections

- *beans* collections
  - From the default *beans* namespace
  - Simple and easy, legacy from Spring V1
- *util* collections
  - From the *util* namespace
    - Requires additional namespace declaration
  - More features available, since Spring V2

Both offer support for *set*, *map*, *list* and *properties* collections

spring Pivotal

Using the *beans* namespace

```
<bean id="service" class="com.acme.service.TransferServiceImpl">
   <property name="customerPolicies">
      <list>
            <ref bean="privateBankingCustomerPolicy"/>
            <ref bean="retailBankingCustomerPolicy"/>
            <bean class="com.acme.DefaultCustomerPolicy"/>
      </list>
   </property>
</bean>
```

```
public void setCustomerPolicies(java.util.List policies) { .. }
```

Equivalent to:

```
TransferServiceImpl service = new TransferServiceImpl();
service.setCustomerPolicies(list);   // create list with bean references
```

ApplicationContext

service -> instance of TransferServiceImpl

spring Pivotal

beans collections limitation

- Can't specify the collection type
  - Example: *java.util.List* implementation is always *ArrayList*
- Collection has no bean id
  - Can't be accessed from the ApplicationContext
  - *Only valid as inner beans*

```
<bean id="service" class="com.acme.service.TransferServiceImpl">
   <property name="customerPolicies">
      <list> ... </list>
   </property>
</bean>
```

OK

NoSuchBeanDefinitionException!!

```
applicationContext.getBean("service");
applicationContext.getBean("customerPolicies");
```

spring Pivotal

Injecting a Set or Map

- Similar support available for Set

```
<property name="customerPolicies">
  <set>
    <ref bean="privateBankingCustomerPolicy"/>
    <ref bean="retailBankingCustomerPolicy"/>
  </set>
</property>
```

- Map (through map / entry / key elements)

```
<property name="customerPolicies">
  <map>
    <entry key="001-pbcp" value-ref="privateBankingCustomerPolicy"/>
    <entry key-ref="keyBean" value-ref="retailBankingCustomerPolicy"/>
  </map>
</property>
```

Key can use primitive type or ref to bean

value also supported

spring Pivotal

Injecting a collection of type *Properties*

- Convenient alternative to a dedicated properties file
  - Use when property values are unlikely to change

```
<property name="config">
  <value>
    server.host=mailer
    server.port=1010
  </value>
</property>
```

```
<property name="config">
  <props>
    <prop key="server.host">mailer</prop>
    <prop key="server.port">1010</prop>
  </props>
</property>
```

```
public void setConfig(java.util.Properties props) { .. }
```

spring Pivotal

util collections

- util: collections allow:
  - specifying collection implementation-type and scope
  - declaring a collection as a top-level bean

```
<bean id="service" class="com.acme.service.TransferServiceImpl"
      p:customerPolicies-ref="customerPolicies"/>

<util:set id="customerPolicies" set-class="java.util.TreeSet">
  <ref bean="privateBankingCustomerPolicy"/>
   <ref bean="retailBankingCustomerPolicy"/>
</util:set>
```

bean id

Implementation class

Also: util:list, util:map, util:properties

beans or *util* collections?

- In most cases, the default collection elements in the beans namespace will suffice
  - But can *only* be inner beans
- Just remember the additional collection features in the **<util/>** namespace, in case you might need them
  - Declare a collection as a top-level bean
  - Specify collection implementation-type
- In the long-run, simpler to always use the **<util/>** namespace collection elements

Summary

- Spring offers many techniques to simplify XML configuration
  - We've seen just a few here
  - It's about expressiveness and elegance, just like code
- Best practices we've discussed are used widely by many existing Spring XML projects
  - Imports, Bean Inheritance, Inner Beans ...
- Other features are more specialized

spring Pivotal

Object Relational Mapping

Using OR Mapping in the Enterprise

Fundamental Concepts and Concerns

spring Pivotal

Topics in this session

- **The Object/Relational mismatch**
- ORM in context
- Benefits of O/R Mapping

spring Pivotal

The Object/Relational Mismatch (1)

- A domain object model is designed to serve the needs of the application
  - Organize data into abstract concepts that prove useful to solving the domain problem
  - Encapsulate behavior specific to the application
  - Under the control of the application developer

spring Pivotal

The Object/Relational Mismatch (2)

- Relational models relate business data and are typically driven by other factors:
  - Performance
  - Space
- Furthermore, a relational database schema often:
  - Predates the application
  - Is shared with other applications
  - Is managed by a separate DBA group

spring Pivotal

Object/Relational Mapping

- Object/Relational Mapping (ORM) engines exist to mitigate the mismatch
- Spring supports all of the major ones:
  – Hibernate
  – EclipseLink
  – Other JPA (Java Persistence API) implementations, such as OpenJPA
- This session will focus on Hibernate

spring Pivotal

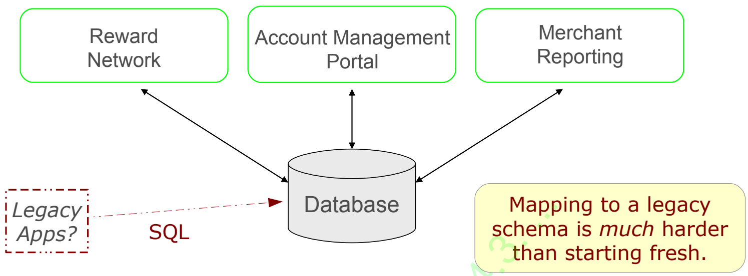
Topics in this session

- The Object/Relational Mismatch
- **ORM in context**
- Benefits of modern-day ORM engines

spring Pivotal

ORM in context

- For the **Reward Dining** domain
  - The database schema already exists
  - Several applications share the data

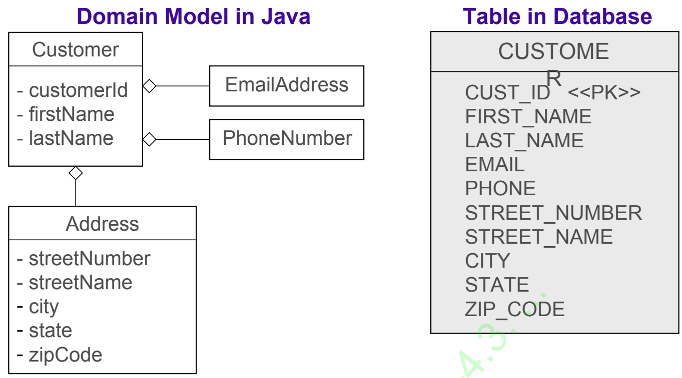


O/R Mismatch: Granularity (1)

- In an object-oriented language, cohesive fine-grained classes provide encapsulation and express the domain naturally
- In a database schema, granularity is typically driven by normalization and performance considerations

O/R Mismatch: Granularity (2)

just one example…



O/R Mismatch: Identity (1)

- In Java, there is a difference between Object identity and Object equivalence:
  - x == y *identity* (same memory address)
  - x.equals(y) *equivalence*
- In a database, identity is based solely on primary keys:
  - x.getEntityId().equals(y.getEntityId())

O/R Mismatch: Identity (2)

- When working with persistent Objects, the identity problem leads to difficult challenges
  - Two different Java objects may correspond to the same relational row
  - But Java says they are *not* equal
- Some of the challenges:
  - Implement equals() to accommodate this scenario
  - Determine when to update and when to insert
  - Avoid duplication when adding to a Collection

spring Pivotal

O/R Mismatch: Inheritance and Associations (1)

- In an object-oriented language:
  - *IS-A* relations are modeled with inheritance
  - *HAS-A* relations are modeled with composition
- In a database schema, relations are limited to what can be expressed by *foreign keys*

spring Pivotal

O/R Mismatch: Inheritance and Associations (2)

- Bi-directional associations are common in a domain model (e.g. Parent-Child)
  - This can be modeled naturally in each Object
- In a database:
  - One side (parent) provides a primary-key
  - Other side (child) provides a foreign-key reference
- For many-to-many associations, the database schema requires a *join table*

Topics in this session

- The Object/Relational Mismatch
- ORM in Context
- **Benefits of O/R Mapping**

Benefits of ORM

- Object Query Language
- Automatic Change Detection
- Persistence by Reachability
- Caching
  - Per-Transaction (1st Level)
  - Per-DataSource (2nd Level)

Object Query Language

- When working with domain objects, it is more natural to query based on objects.
  - Query with SQL:

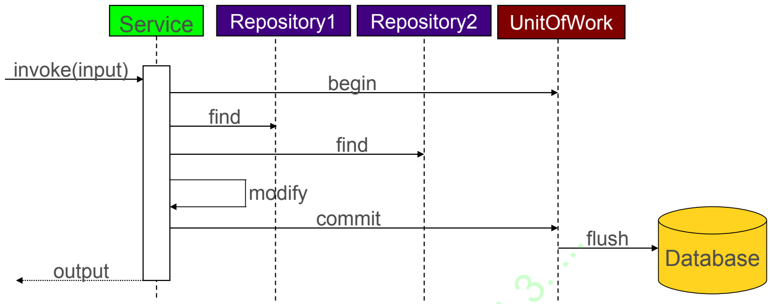
```
SELECT c.first_name, c.last_name, a.city, ...
   FROM customer c, customer_address ca, address a
WHERE ca.customer_id = c.id
   AND   ca.address_id = a.id
   AND   a.zip_code = 12345
```

  - Query with object properties and associations:

```
SELECT c FROM Customer c WHERE c.address.zipCode = 12345
```

Automatic Change Detection

- When a unit-of-work completes, all modified state will be synchronized with the database.



Persistence by Reachability

- When a persistent object is being managed, other associated objects may become managed transparently:

```
Order order = orderRepository.findByConfirmationId(cid);
// order is now a managed object – retrieved via ORM

LineItem item = new LineItem(..);
order.addLineItem(item);
// item is now a managed object – reachable from order
```

(Un)Persistence by Reachability
= Make Transient

- The same concept applies for deletion:

```
Order order = orderRepository.findByConfirmationId(cid);
// order is now a managed object – retrieved via ORM

List<LineItem> items = order.getLineItems();
for (LineItem item : items) {
    if (item.isCancelled()) { order.removeItem(item); }
    // the database row for this item will be deleted
}

if (order.isCancelled()) {
    orderRepository.remove(order);
    // all item rows for the order will be deleted
}
```

Item becomes transient

Order and all its items now transient

spring Pivotal

Caching

- The first-level cache (1LC) is scoped at the level of a unit-of-work
  - When an object is first loaded from the database within a unit-of-work it is stored in this cache
  - Subsequent requests to load that same entity from the database will hit this cache first
- The second-level cache (2LC) is scoped at the level of the SessionFactory
  - Reduce trips to database for read-heavy data
  - Especially useful when a single application has exclusive access to the database

spring Pivotal

Summary

- Managing persistent objects is hard
  - Especially if caching is involved
  - Especially on a shared, legacy schema with existing applications
- The ORM overcomes *some* of these problems
  - Automatic change detection, queries, caching
  - Ideal if your application *owns* its database
  - It is *not* a magic-bullet
    - JDBC may still be better for some tables/queries
    - True distributed cache coherency is *very* hard
    - *Design* for it and *test* performance

Spring Security
XML Configuration

Classic configuration options for Web Application Security

Addressing Common Security Requirements

Spring Security – XML Configuration

- Spring Security is also configurable via XML
  - Most common in older code bases
  - Some default behaviors are different.

Configuration in web.xml

- web.xml configuration remains the same
  - springSecurityFilterChain
  - May also use Servlet 3.0 initializers

web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```


intercept-url

- intercept-urls are evaluated in the order listed
  - first match is used, put specific matches first

```
<beans>
  <security:http>

    <security:intercept-url pattern="/accounts/edit*"
                  access="ROLE_ADMIN" />
    <security:intercept-url pattern="/accounts/account*"
                  access="ROLE_ADMIN,ROLE_USER" />
    <security:intercept-url pattern="/accounts/**"
                  access="IS_AUTHENTICATED_FULLY" />
    <security:intercept-url pattern="/customers/**"
                  access="IS_AUTHENTICATED_ANONYMOUSLY" />

  </security:http>
</beans>
```

Syntax available since Spring Security 2.0

Security EL expressions

- hasRole('role')
  - Checks whether the principal has the given role
- hasAnyRole('role1', 'role2', …)
  - Checks whether the principal has any of the given roles
- isAnonymous()
  - Allows access for unauthenticated principals
- isAuthenticated()
  - Allows access for authenticated or remembered principals

Available from Spring Security 3.0
Previous syntax still works in Spring Security 3.0

spring Pivotal

Intercept-url and Expression Language

- Expression Language provides more flexibility
  - Many built-in expressions available

Spring configuration file

```
<beans>
  <security:http use-expressions="true">

  <security:intercept-url pattern="/accounts/edit*"
                access="hasRole('ROLE_ADMIN')" />
  <security:intercept-url pattern="/accounts/account*"
                access="hasAnyRole('ROLE_ADMIN', 'ROLE_USER')" />
  <security:intercept-url pattern="/accounts/**"
                access="isAuthenticated() and hasIpAddress('192.168.1.0/24')" />

  </security:http>
</beans>
```

Expression Language needs to be enabled explicitly

Syntax available from Spring Security 3.0

spring Pivotal

Working with roles

- Checking if the user has one single role

```
<security:intercept-url pattern="/accounts/update*" access="hasRole('ROLE_ADMIN')" />
```

- "or" clause

```
<security:intercept-url pattern="/accounts/update*"
    access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')" />
```

- "and" clause

```
<security:intercept-url pattern="/accounts/update*"
    access="hasRole('ROLE_ADMIN') and hasRole('ROLE_MANAGER')" />
```

- Previous and new syntax can't be mixed

Not correct!!

```
<security:intercept-url pattern="/accounts/update*"
                        access="hasRole('ROLE_MANAGER')" />
<security:intercept-url pattern="/accounts/update*" access="ROLE_ADMIN" />
```

spring Pivotal

Specifying login and logout

Exempt login page (Spring Security 3.1)

Specify login options

Must be declared explicitly or no logout possible

```
<beans ...>
  <security:http pattern="/accounts/login" security="none"/>

  <security:http use-expressions="true">

      <security:form-login login-page="/accounts/login"
                           default-target-url="/accounts/home"/>

      <security:intercept-url pattern="/accounts/update*"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')" />

      <security:intercept-url pattern="/accounts/**"
                        access="hasRole('ROLE_ADMIN')" />

      <security:logout logout-success-url="/home.html"

  </security:http>
  ...
```

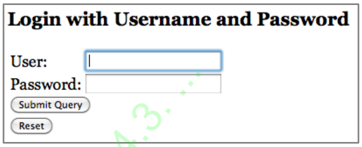
Spring configuration file

spring Pivotal

Setting up User Login

- Default auth. provider assumes form-based login
  - This is *web* security after all
  - *Must* specify form-login element
  - A basic form is provided
  - Configure to use your own login-page

```
<security:http>
    <security:form-login/>
    …
</security:http>

<security:authentication-manager>
    <security:authentication-provider>
     …
    </security:authentication-provider>
<security:authentication-manager>
```



An Example Login Page

URL that indicates an authentication request

```
<form action="<c:url value='j_spring_security_check'/>" method="POST">
  <input type="text" name="j_username"/>
  <br/>
  <input type="password" name="j_password"/>
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form>
```

The expected keys for generation of an authentication request token

login-example.jsp

Above example shows default values (*j_spring_security_check*, *j_username*, *j_password*). All of them can be redefined using <security:form-login/>

The In-Memory User Service

- Useful for development and testing
  - *Note:* must restart system to reload properties

Spring configuration file

```
<security:authentication-manager>
    <security:authentication-provider>
      <security:user-service properties="/WEB-INF/users.properties" />
    </security:authentication-provider>
<security:authentication-manager>
```

```
admin=secret,ROLE_ADMIN,ROLE_MEMBER,ROLE_GUEST
testuser1=pass,ROLE_MEMBER,ROLE_GUEST
testuser2=pass,ROLE_MEMBER
guest=guest,ROLE_GUEST
```

List of roles separated by commas

login

password

The JDBC user service (2/2)

- Configuration:

```
<beans>
  <security:http> … <security:http>

  <security:authentication-manager>
    <security:authentication-provider>
      <security:jdbc-user-service data-source-ref="myDatasource" />
    </security:authentication-provider>
  </security:authentication-manager>
</beans>
```

Spring configuration file

Can customize queries using attributes:
- users-by-username-query
- authorities-by-username-query
- group-authorities-by-username-query

Password Encoding

- Can encode passwords using a hash
  - sha, md5, …

```
<security:authentication-provider>
    <security:password-encoder hash="sha-256" />
    <security:user-service properties="/WEB-INF/users.properties" />
</security:authentication-provider>
```

simple encoding

- Secure passwords using a well-known string
  - Known as a 'salt', makes brute force attacks harder

```
<security:authentication-provider>
    <security:password-encoder hash="sha-256">
        <security:salt-source system-wide="MySalt" />
    </security:password-encoder>
    <security:user-service properties="/WEB-INF/users.properties" />
</security:authentication-provider>
```

encoding with salt

spring Pivotal

Method Security using XML

- Can apply security to multiple beans with only a simple declaration

```
<security:global-method-security>
  <security:protect-pointcut
      expression="execution(* com.springsource..*Service.*(..))"
      access="ROLE_USER,ROLE_MEMBER" />
</security:global-method-security>
```

Spring configuration file

*Spring Security 2 syntax only. SpEL **not** supported here.*

spring Pivotal

Custom Filter Chain

- Filter on the stack may be **replaced** by a custom filter

```
<security:http>
   <security:custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

- Filter can be **added** to the chain

```
<security:http>
   <security:custom-filter after="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialFilter"/>
```

spring Pivotal

Spring JMS

Simplifying Messaging Applications

JmsTemplate and Spring's Listener Container

Topics in this Session

- **Introduction to JMS**
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

Java Message Service (JMS)

- The JMS API provides an abstraction for accessing Message Oriented Middleware
  - Avoid vendor lock-in
  - Increase portability
- JMS does *not* enable different MOM vendors to communicate
  - Need a bridge (expensive)
  - Or use AMQP (standard msg protocol, like SMTP)
    - See RabbitMQ

spring Pivotal

JMS Core Components

- Message
- Destination
- Connection
- Session
- MessageProducer
- MessageConsumer

spring Pivotal

JMS Message Types

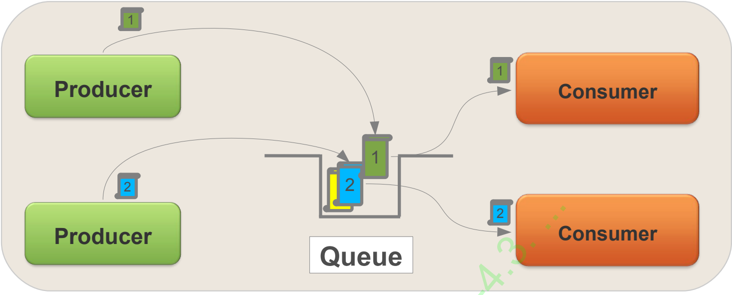
- Implementations of the Message interface
  - TextMessage
  - ObjectMessage
  - MapMessage
  - BytesMessage
  - StreamMessage

JMS Destination Types

- Implementations of the Destination interface
  - Queue
    - Point-to-point messaging
  - Topic
    - Publish/subscribe messaging
- Both support *multiple* producers and consumers
  - Messages are different
  - Let's take a close look ...

JMS Queues: Point-to-point

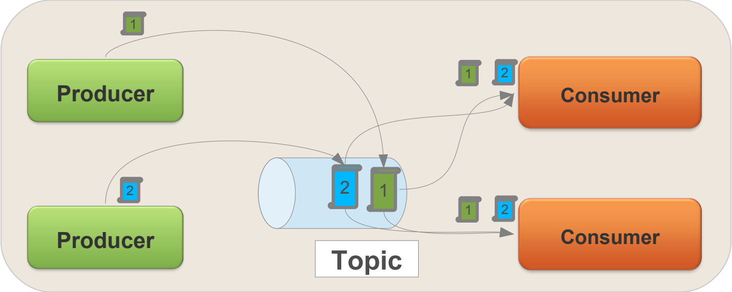
1. Message sent to queue
2. Message queued
3. Message consumed by *single* consumer



spring Pivotal

JMS Topics: Publish-subscribe

1. Message sent to topic
2. Message optionally stored
3. Message distributed to *all* subscribers



spring Pivotal

The JMS Connection

- A JMS Connection is obtained from a factory

```
Connection conn = connectionFactory.createConnection();
```

- Typical enterprise application:
  - ConnectionFactory is a managed resource bound to JNDI

```
Properties env = new Properties();
// provide JNDI environment properties
Context ctx = new InitialContext(env);
ConnectionFactory connectionFactory =
    (ConnectionFactory) ctx.lookup("connFactory");
```

spring Pivotal

The JMS Session

- A Session is created from the Connection
  - Represents a unit-of-work
  - Provides transactional capability

```
Session session = conn.createSession(
        boolean transacted, int acknowledgementMode);

// use session
if (everythingOkay) {
  session.commit();
} else {
  session.rollback();
}
```

spring Pivotal

Creating Messages

- The Session is responsible for the creation of various JMS Message types

```
session.createTextMessage("Some Message Content");

session.createObjectMessage(someSerializableObject);

MapMessage message = session.createMapMessage();
message.setInt("someKey", 123);

BytesMessage message = session.createBytesMessage();
message.writeBytes(someByteArray);
```

spring Pivotal

Producers and Consumers

- The Session is also responsible for creating instances of MessageProducer and MessageConsumer

```
producer = session.createProducer(someDestination);

consumer = session.createConsumer(someDestination);
```

spring Pivotal

Topics in this Session

- Introduction to JMS
- **Apache ActiveMQ**
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

JMS Providers

- Most providers of Message Oriented Middleware (MoM) support JMS
  - WebSphere MQ, Tibco EMS, Oracle EMS, JBoss AP, SwiftMQ, etc.
  - Some are Open Source, some commercial
  - Some are implemented in Java themselves
- The lab for this module uses Apache ActiveMQ

Apache ActiveMQ

- Open source message broker written in Java
- Supports JMS and many other APIs
  - Including non-Java clients!
- Can be used stand-alone in production environment
  - 'activemq' script in download starts with default config
- Can also be used *embedded* in an application
  - Configured through ActiveMQ or Spring configuration
  - *What we use in the labs*

spring Pivotal

Apache ActiveMQ Features

Support for:

- Many cross language clients & transport protocols
  - Incl. excellent Spring integration
- Flexible & powerful deployment configuration
  - Clustering incl. load-balancing & failover, ...
- Advanced messaging features
  - Message groups, virtual & composite destinations, wildcards, etc.
- Enterprise Integration Patterns when combined with Spring Integration or Apache Camel
  - from the book by Gregor Hohpe & Bobby Woolf

spring Pivotal

Topics in this Session

- Introduction to JMS
- Apache ActiveMQ
- **Configuring JMS Resources with Spring**
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

Configuring JMS Resources with Spring

- Spring enables decoupling of your application code from the underlying infrastructure
  - Container provides the resources
  - Application is simply coded against the API
- Provides deployment flexibility
  - use a standalone JMS provider
  - use an application server to manage JMS resources

See: Spring Framework Reference – Using Spring JMS
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms

Configuring a ConnectionFactory

- ConnectionFactory may be standalone

```
@Bean
public ConnectionFactory connectionFactory() {
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    cf.setBrokerURL("tcp://localhost:60606");
    return cf;
}
```

- Or retrieved from JNDI

```
@Bean
public ConnectionFactory connectionFactory() throws Exception {
    Context ctx = new InitialContext();
    return (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");
}
```

```
<jee:jndi-lookup id="connectionFactory" jndi-name="jms/ConnectionFactory"/>
```

spring Pivotal

Configuring Destinations

- Destinations may be standalone

```
@Bean
public Destination orderQueue() {
    return new ActiveMQQueue( "order.queue" );
}
```

- Or retrieved from JNDI

```
@Bean
public Destination orderQueue() throws Exception {
    Context ctx = new InitialContext();
    return (Destination) ctx.lookup("jms/OrderQueue");
}
```

```
<jee:jndi-lookup id="orderQueue" jndi-name="jms/OrderQueue"/>
```

spring Pivotal

Topics in this Session

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- **Spring's JmsTemplate**
- Sending Messages
- Receiving Messages
- Advanced Features

spring Pivotal

Spring's JmsTemplate

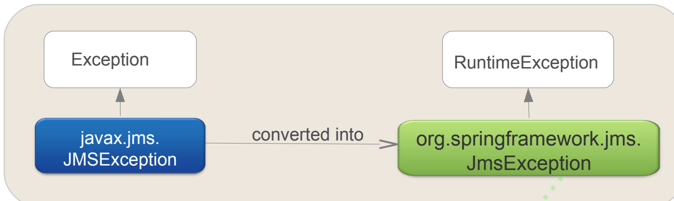
- The template simplifies usage of the API
  - Reduces boilerplate code
  - Manages resources transparently
  - Converts checked exceptions to runtime equivalents
  - Provides convenience methods and callbacks

NOTE: The *AmqpTemplate* (used with RabbitMQ) has an almost identical API to the *JmsTemplate* – they offer similar abstractions over very different products

spring Pivotal

Exception Handling

- Exceptions in JMS are checked by default
- JmsTemplate converts checked exceptions to runtime equivalents



JmsTemplate configuration

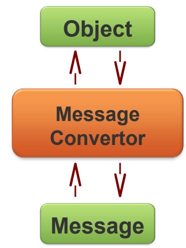
- *Must* provide reference to ConnectionFactory
  - via either constructor or setter injection
- Optionally provide other facilities
  - *setMessageConverter* (1)
  - *setDestinationResolver* (2)
  - *setDefaultDestination* or *setDefaultDestinationName* (3)

```
@Bean
public JmsTemplate jmsTemplate () {
    JmsTemplate template = new JmsTemplate( connectionFactory() );
    template.setMessageConverter ( … );
    template.setDestinationResolver ( … );
    return template;
}
```

(1), (2), (3) – see next few slides

(1) MessageConverter

- The JmsTemplate uses a MessageConverter to convert between objects and messages
  - You only send and receive objects
- The default SimpleMessageConverter handles basic types
  - String to TextMessage
  - Map to MapMessage
  - byte[] to BytesMessage
  - Serializable to ObjectMessage

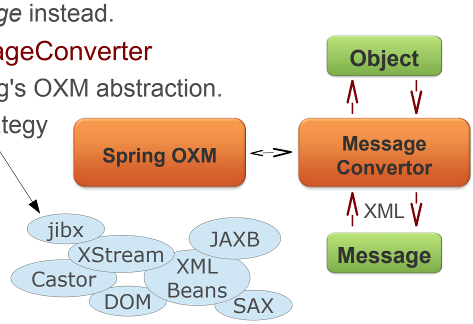


NOTE: It is possible to implement custom converters by implementing the *MessageConverter* interface

spring Pivotal

XML MessageConverter

- XML is a common message payload
  - ...but there is no "XmlMessage" in JMS
  - Use *TextMessage* instead.
- MarshallingMessageConverter
  - Plugs into Spring's OXM abstraction.
  - You choose strategy



spring Pivotal

MarshallingMessageConverter Example

```
@Bean public JmsTemplate jmsTemplate () {
    JmsTemplate template = new JmsTemplate( connectionFactory() );
    template.setMessageConverter ( msgConverter() );
    return template;
}

@Bean public MessageConverter msgConverter() {
    MessageConverter converter = new MarshallingMessageConverter();
    converter.setMarshaller ( marshaller() );
    return converter;
}

@Bean  public Marshaller marshaller() {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setContextPath ( "example.app.schema" );
    return marshaller;
}
```

JAXB2 Illustrated here, other strategies available.

(2) DestinationResolver

- Convenient to use destination names at runtime
- DynamicDestinationResolver used by default
  - Resolves topic and queue names
  - *Not* their Spring bean names
- JndiDestinationResolver also available


```
Destination resolveDestinationName(Session session,
        String destinationName,
        boolean pubSubDomain)
throws JMSException;
```

publish-subscribe?
true q Topic
false q Queue

(3) Default Destination

- Used by default when sending *or* receiving messages

Specify by Object

```
@Bean
public JmsTemplate orderTemplate () {
    JmsTemplate template = new JmsTemplate ( connectionFactory() );
    template.setDefaultDestination ( orderQueue() );
    return template;
}
```

Specify by Name

```
@Bean public JmsTemplate orderTemplate () {
    JmsTemplate template = new JmsTemplate ( connectionFactory() );
    template.setDefaultDestinationName ("order.queue");
    return template;
}
```

spring Pivotal

Topics in this Session

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring’s JmsTemplate
- **Sending Messages**
- Receiving Messages
- Advanced Features

spring Pivotal

Sending Messages

- The template provides options
  - Simple methods to send a JMS message
  - One line methods that leverage the template's MessageConverter
  - Callback-accepting methods that reveal more of the JMS API
- Use the simplest option for the task at hand

Sending POJO

- A message can be sent in one single line

```
public class JmsOrderManager implements OrderManager {
  @Autowired JmsTemplate jmsTemplate;
  @Autowired Destination orderQueue;

  public void placeOrder(Order order) {
    String stringMessage = "New order " + order.getNumber();
    jmsTemplate.convertAndSend("message.queue", stringMessage );
                                // use destination resolver and message converter

    jmsTemplate.convertAndSend(orderQueue, order);   // use message converter

    jmsTemplate.convertAndSend(order);   // use converter and default destination
  }
}
```

No @Qualifier so Destination is wired by *name*

Sending JMS Messages

- Useful when you need to access JMS API
  - eg. set expiration, redelivery mode, reply-to ...

```
public void sendMessage(final String msg) {

   this.jmsTemplate.send( (session) -> {
      TextMessage message = session.createTextMessage(msg);
      message.setJMSExpiration(2000);    // 2 seconds
      return message;
    });
}
```

Lambda syntax

```
public interface MessageCreator {
   public Message createMessage(Session session)
                     throws JMSException;
}
```

spring Pivotal

Topics in this Session

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- **Receiving Messages**
- Advanced Features

spring Pivotal

Receiving Objects

- JmsTemplate can also *receive* data
  - Automatically converted using MessageConverter
  - Underlying messages hidden

```
public void receiveData() {

  // use message converter and destination resolver
  String s =  (String) jmsTemplate.receiveAndConvert("message.queue");
  // use message converter
  Order order1 = (Order) jmsTemplate.receiveAndConvert(orderQueue);
  // use message converter and default destination
  Order order2 = (Order) jmsTemplate.receiveAndConvert(\);
}
```

Receiving Messages

- Or you may access the underlying message
  - Gives you access to message properties

```
public void receiveMessages() {

  // handle JMS native message from default destination
  ObjectMessage orderMessage =  (ObjectMessage) jmsTemplate.receive();
  Order order2 = (Order) orderMessage.getObject();

  // receive(destination) and receive(destinationName) also available
}
```

Synchronous Message Exchange

- JmsTemplate also implements a request/reply pattern
  - Using sendAndReceive()
  - Sending a message and blocking until a reply has been received (also uses receiveTimeout)
  - Manage a temporary reply queue automatically by default

```
public void processMessage(String msg) {

  Message reply =  jmsTemplate.sendAndReceive("message.queue",
      (session) -> {
          return session.createTextMessage(msg);
      });
  // handle reply
}
```


Asynchronous or Synchronous

- Sending messages is asynchronous
  - The send methods return immediately
    - Even if the message takes time to be delivered
    - Recall the acknowledgement modes in createSession()
- But receive() and receiveAndConvert() are blocking
  - Synchronous – will wait for ever for a new message
    - optional timeout: setReceiveTimeout()
- How can we receive data asynchronously?
  - JMS defines *Message Driven Beans*
  - But you normally need a full JEE container to use them

Spring's MessageListener Containers

- Spring provides containers for asynchronous JMS reception
  - *SimpleMessageListenerContainer*
    - Uses plain JMS client API
    - Creates a fixed number of Sessions
  - *DefaultMessageListenerContainer*
    - Adds transactional capability

- Many configuration options available for each container type

spring Pivotal

Quick Start

Steps for Asynchronous Message Handling

(1) Define POJO / Bean to process Message
(2) Define JmsListenerContainerFactory / Enable Annotations
(3) Annotate POJO to be message-driven

spring Pivotal

Spring 4.1

Step (1)

Define POJO / Bean to Process Message

- Define a POJO to process message
  - Note: No references to JMS

```
public class OrderServiceImpl {
        @JmsListener(destination="queue.order")
        @SendTo("queue.confirmation")
        public OrderConfirmation order(Order o) { … }
}
```

  - Define as a Spring bean using XML, JavaConfig, or annotations as preferred
  - @JmsListener enables a JMS message consumer for the method
  - @SendTo defines response destination (optional)

spring Pivotal

Spring 4.1

Step (2)

Define JmsListenerContainerFactory to use

- JmsListenerContainerFactory
  - Separates JMS API from your POJO:

```
@Configuration @EnableJms
public class MyConfiguration {

   @Bean
   public DefaultJmsListenerContainerFactory
       jmsListenerContainerFactory () {
     DefaultJmsListenerContainerFactory cf =
          new DefaultJmsListenerContainerFactory( );
     cf.setConnectionFactory(connectionFactory());
     ...
     return cf;
   }
}
```

Enable annotations

Default container name

Set *ConnectionFactory*

Many settings available:
TransactionManager, TaskExecutor, ContainerType ...

spring Pivotal

Spring 4.1

Step (3)
Define Receiving Method with @JmsListener

- Container with name **jmsListenerContainerFactory** is used by default

```
public class OrderServiceImpl {
      @JmsListener(containerFactory="myFactory",
         destination="orderConfirmation")
      public void process(OrderConfirmation o) { … }
}
```

- Can also set a custom concurrency or a payload selector

```
public class OrderServiceImpl {
      @JmsListener(selector="type = 'Order'",
         concurrency="2-10", destination = "order")
      public OrderConfirmation order(Order o) { … }
}
```

spring Pivotal

Using JMS: Pros and Cons

- Advantages
  - Application freed from messaging concerns
    - Resilience, guaranteed delivery (compare to REST)
  - Asynchronous support built-in
  - Interoperable – languages, environments
- Disadvantages
  - Requires additional third-party software
    - Can be expensive to install and maintain
  - More complex to use – *but not with JmsTemplate!*

Spring Enterprise – 4 day course on application integration

spring Pivotal

Lab

Sending and Receiving Messages in a Spring Application

Coming Up: Spring's Caching Connection Factory

spring Pivotal

Topics in this Session

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- **Optional Features**
  - **Using XML**

spring Pivotal

Alternative Step (2)
Use JMS XML Namespace Support

- Equivalent Capabilities
  - The **containerId** attribute exposes the configuration of the container with that name
  - Same configuration options available
    - task execution strategy, concurrency, container type, transaction manager and more

```
<jms:annotation-driven/>

<jms:listener-container
    containerId="jmsMessageContainerFactory"
    connection-factory="myConnectionFactory"/>

<bean id="orderService" class="org.acme.OrderService"/>
```

spring Pivotal

100% XML Equivalent

- Use *jms:listener-container* with embedded *jms:listeners*
  - Supports multiple listeners in a single declaration
  - Same configuration options available

```
<jms:listener-container connection-factory="myConnectionFactory">
  <jms:listener destination="order.queue"
                ref="orderService"
                method="order"
                response-destination="confirmation.queue"  />
  <jms:listener destination="confirmation.queue"
                ref="orderService"
                method="confirm" />
</jms:listener-container>

<bean id="orderService" class="org.acme.OrderService"/>
```

No need for *@JmsListener*

spring Pivotal

Message-Driven POJO in XML

- Listener unpacks incoming payload
  - Uses the MessageConverter
  - Invokes method on POJO
  - Return value sent to response-destination after conversion

```java
public class OrderService {
      public OrderConfirmation order(Order o) {}
}
```

```xml
<jms:listener
      ref="orderService"
      method="order"
      destination="queue.orders"
      response-destination="queue.confirmation"/>
```

CachingConnectionFactory

- JmsTemplate aggressively closes and reopens resources like Sessions and Connections
  - Lots of overhead and poor performance
  - Normally these are cached by connection factory
- Use our *CachingConnectionFactory* to add caching within the application if needed

```xml
<bean id="connectionFactory"
        class="org.springframework.jms.connection.CachingConnectionFactory">
   <property name="targetConnectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
         <property name="brokerURL" value="vm://embedded?broker.persistent=false"/>
      </bean>
   </property>
</bean>
```

Performance and Operations

Management and Monitoring of Spring Java Applications

Exporting Spring Beans to JMX

spring Pivotal

Topics in this Session

JMX

- **Introduction**
- JMX
- Introducing Spring JMX
- Automatically exporting existing MBeans
- Spring Insight

spring Pivotal

Overall Goals

- Gather information about application during runtime
- Dynamically reconfigure app to align to external occasions
- Trigger operations inside the application
- Even adapt to business changes in smaller scope

spring Pivotal

Topics in this Session

- Introduction
- **JMX**
- Introducing Spring JMX
- Automatically exporting existing MBeans
- Spring Insight

spring Pivotal

What is JMX?

- The **J**ava **M**anagement E**x**tensions specification aims to create a standard API for adding management and monitoring to Java applications

- Management
  - Changing configuration properties at runtime
- Monitoring
  - Reporting cache hit/miss ratios at runtime

spring Pivotal

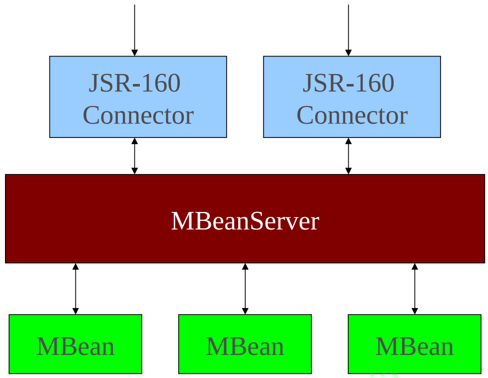
How JMX Works

- To add this management and monitoring capability, JMX instruments application components

- JMX introduces the concept of the MBean
  - An object with management metadata

spring Pivotal

JMX Architecture



JMX Architecture

- MBeanServer acts as broker for communication between
  - Multiple local MBeans
  - Remote clients and MBeans
- MBeanServer maintains a keyed reference to all MBeans registered with it
  - *object name*
- Many generic clients available
  - JDK: jconsole, jvisualvm

JMX Architecture

- An MBean is an object with additional management metadata
  - Attributes (→ properties)
  - Operations (→ methods)
- The management metadata can be defined statically with a Java interface or defined dynamically at runtime
  - Simple MBean or Dynamic MBean respectively

Plain JMX – Example Bean

```
public interface JmxCounterMBean {

    int getCount();// becomes Attribute named 'Count'

    void increment();    // becomes Operation named 'increment'
}
```

```
public class JmxCounter implements JmxCounterMBean {
    ...
    public int getCount() {…}

    public void increment() {…}
}
```


Plain JMX – Exposing an MBean

```
MBeanServer server = ManagementFactory.getPlatformMBeanServer();

JmxCounter bean = new JmxCounter(…);

try {
   ObjectName name = new ObjectName("ourapp:name=counter");
   server.registerMBean(bean, name);
} catch (Exception e) {
   e.printStackTrace();
}
```

spring Pivotal

Topics in this Session

- Introduction
- JMX
- **Introducing Spring JMX**
- Automatically exporting existing MBeans
- Spring Insight

spring Pivotal

Goals of Spring JMX

- Using the raw JMX API is difficult and complex
- The goal of Spring's JMX support is to simplify the use of JMX while hiding the complexity of the API

Goals of Spring JMX

- Configuring JMX infrastructure
  - Declaratively using context namespace or FactoryBeans
- Exposing Spring beans as MBeans
  - Annotation based metadata
  - Declaratively using Spring bean definitions
- Consuming JMX managed beans
  - Transparently using a proxy-based mechanism

Spring JMX Steps

1. Configuring MBean Server
2. Configure Exporter
3. Control Attribute / Operation Exposure.

spring Pivotal

Step 1: Creating an MBeanServer

- Use context namespace to locate or create an MBeanServer

XML
```
<context:mbean-server />
```

- Or declare it explicitly

or JavaConfig
```
@Bean
public MBeanServerFactoryBean mbeanServer () {
    MBeanServerFactoryBean server = new MBeanServerFactoryBean();
    server.setLocateExistingServerIfPossible( true );
    ...
    return server;
}
```

spring Pivotal

Step 2: Exporting a Bean as an MBean

- Start with one or more existing POJO bean(s)

```
<bean id="messageService" class="example.MessageService"/>
```

- Use the MBeanExporter to export it
  - By default: *all public* properties exposed as attributes, *all public* methods exposed as operations.

JavaConfig

```
@Bean
public MBeanExporter mbeanExporter () {
    MBeanExporter exporter = new MBeanExporter();
    exporter.setAutodetect ( true );
    ...
    return exporter;
}
```

or XML

```
<context:mbean-export/>
```

spring Pivotal

Step 1 & 2: JavaConfig Shortcut

- One annotation defines server and exporter:

```
@Configuration
@EnableMBeanExport
public class MyConfig {
    …
}
```

Specific server bean configurable if desired.

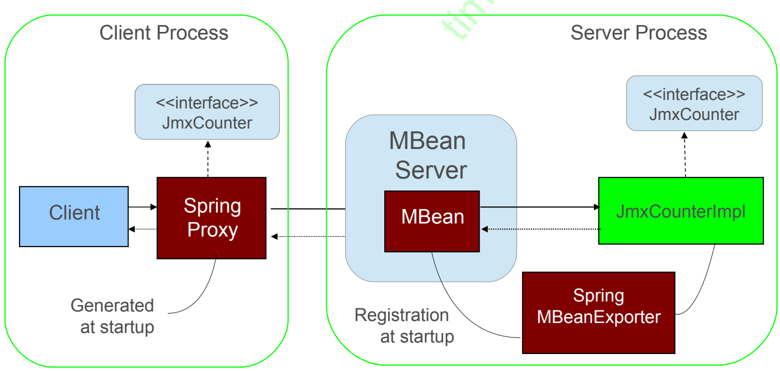
spring Pivotal

3. Control Attribute/Operation Exposure:

- Combine Annotations with Exporter:
  - Only annotated attributes/operations exposed.

```
@ManagedResource(objectName="statistics:name=counter",
                        description="A simple JMX counter")
public class JmxCounterImpl implements JmxCounter {
   @ManagedAttribute(description="The counter value")
   public int getCount() {…}
   @ManagedOperation(description="Increments the counter value")
   public void increment() {…}
}
```


Spring in the JMX architecture



Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- **Automatically exporting existing MBeans**
- Spring Insight

Automatically Exporting Pre-existing MBeans

- Some beans are MBeans themselves
  - Example: Log4j's `LoggerDynamicMBean`
  - Spring will auto-detect and export them for you

```
<context:mbean-export/>

<bean class="org.apache.log4j.jmx.LoggerDynamicMBean">
   <constructor-arg>
      <bean class="org.apache.log4j.Logger"
               factory-method="getLogger"/>
         <constructor-arg value="org.springframework.jmx" />
      </bean>
   </constructor-arg>
</bean>
```

Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- Automatically exporting existing MBeans
- **Spring Insight**

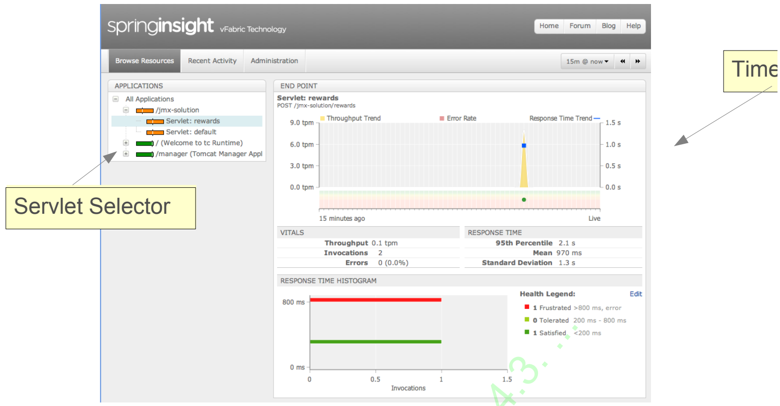
spring Pivotal

Spring Insight Overview

- Part of tc Server Developer Edition
  - Monitors web applications deployed to tc Server
  - http://localhost:8080/insight
- Focuses on what's relevant
  - esp. performance related parts of the application
- Detects performance issues during development
  - Commercial version for production: *vFabric APM*

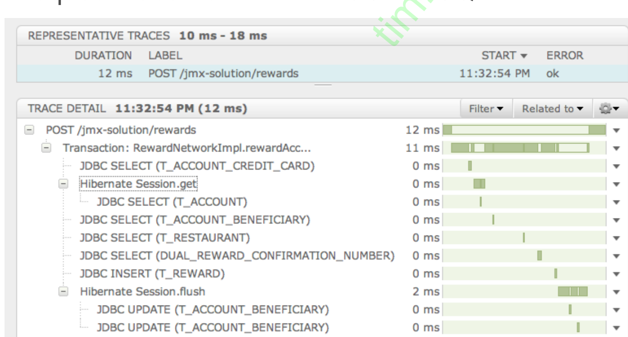
spring Pivotal

Spring Insight Overview



Spring Insight Overview

- A request trace from HTTP POST to SQL



Summary

- Spring JMX
  - Export Spring-managed beans to a JMX MBeanServer
  - Simple value-add now that your beans are managed
- Steps
  - Create MBean server
  - Automatically export annotated and pre-existing Mbeans
    - Use @EnableMBeanExport or <context:mbean-server> and <context:mbean-export>
  - Use Spring annotations to declare JMX metadata
- Spring Insight (tc Server Developer Edition)
  - Deep view into your web-application in STS

Optional Lab

Monitoring and Managing a Java Application