Spring Development for Java

Frank Moley
Internet Architect - Garmin International

Who am I?

- Java Developer and Architect
- Internet Architect with Garmin International
- Spring Core Certified Professional on 3.0 and 4.0
- 13+ years of software development experience
- 6 years of professional Spring Experience

What this workshop is

- High intensity introduction to the Spring Framework
- Modeled from the Spring Core class
- Opinionated view of how to use Spring Platform, not a complete view
- Focused on the areas I feel are most important to understand

What this workshop isn't

- Not a complete Spring Core class, that class is 32 hours long
- Not valid for certification
- Not a complete introduction to Spring

Agenda

- Introduction to Spring and IoC
- Configuration of Spring Application Context using Java Config
- Spring bean lifecycle (one of the most important areas)
- Testing with Spring*
- Aspecting with Spring*

Agenda (cont)

- Data Access with Spring
- Spring MVC
- Spring ReSTful Services
- Spring JMS/JMX *
- Spring Boot
- * will be covered quickly if at all

Welcome to Spring

Goals of Spring

- Spring has a goal of providing support for developing enterprise Java applications
    - They deal with the “plumbing” of your applications
        - You focus on the business needs of your customers
- Provide consistently framed abstractions to many of the common enterprise needs, remove boiler plate code

Spring Core Support

- Application Configurations
- Enterprise Integration (J2EE) and Integration Patterns (Fowler)
- Testing frameworks (leveraging IoC in tests with Mock objects)
- Data access -> read not just database access, but many ways to get data

IoC

Inversion of Control

- To fully understand Spring, one must first understand Inversion of Control or IoC (also called dependency injection)

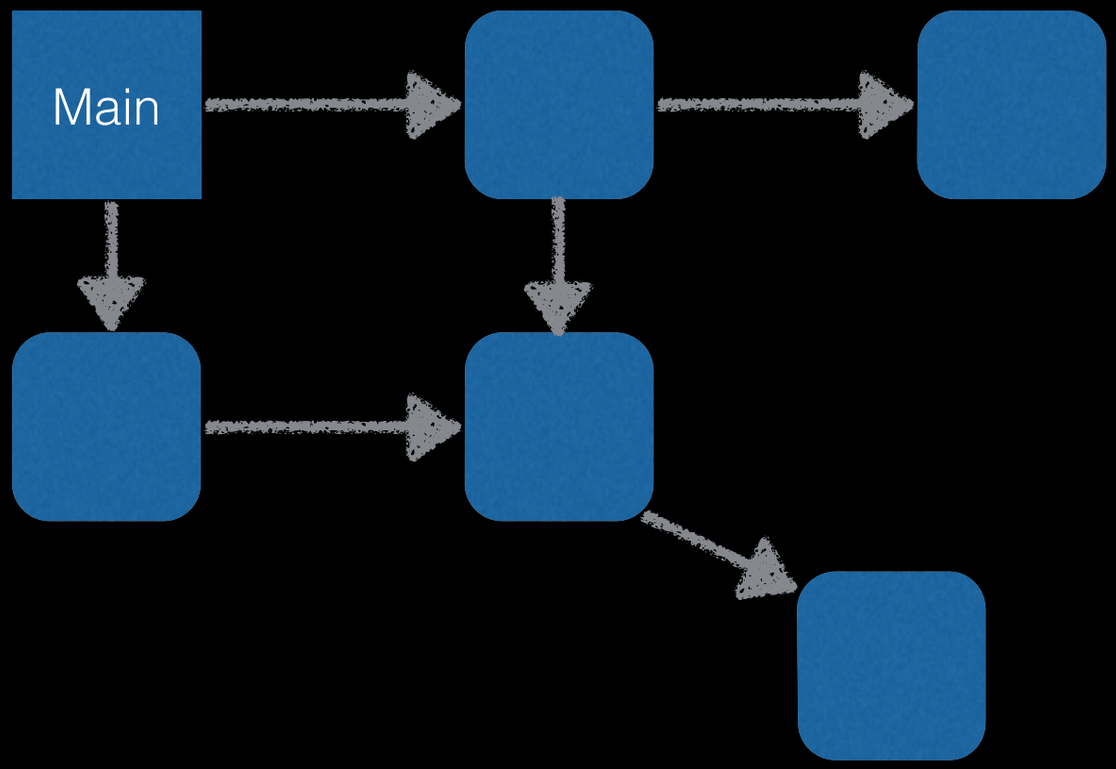
- IoC has the purpose of letting the container serve you dependencies to your class instead of you class building its own dependencies.

- Reduces the noise in your classes and your applications.

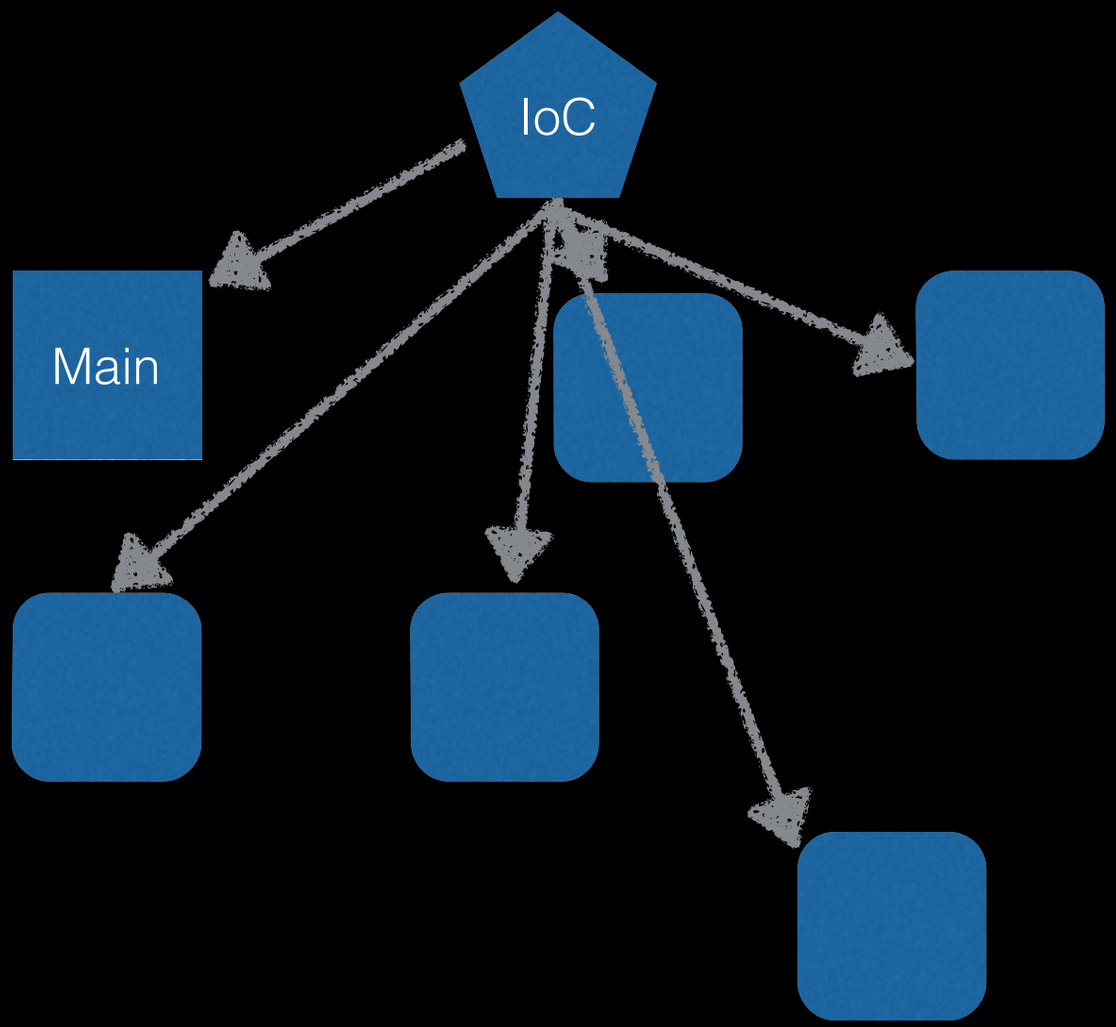
Why is IoC so important?

- Reduces code complexity, your code only focuses on the business needs
- Reduces defects in inappropriately constructing dependencies
- Provides for consistent behavior of dependencies
- IoC allows us to focus on the contract of the API, instead of the inner workings of the dependency.

Tradition Dependency Management



IoC Dependency Management



Application Context

Purpose of AC

- The Application Context serves several primary purposes
  - Provides the necessary meta data to load all of the bean instances
  - Provides a framework for creating beans in the correct order
  - Provides the mechanism to inject the beans
  - Maintains state of the application beans

Creating AC - Standalone

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

    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(ExampleConfig.class);
    WidgetOrderService widgetOrderService = applicationContext.getBean(WidgetOrderService.class);
}
```

Creating AC - war (pre 3.0)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:/example-config.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>rest</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Creating AC - war 3.0

```java
public class WebInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext();
        applicationContext.register(ExampleConfig.class);
        servletContext.addListener(new ContextLoaderListener(applicationContext));

        applicationContext.setServletContext(servletContext);

        ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(applicationContext));
        servlet.addMapping("/");
        servlet.setLoadOnStartup(1);
    }
}
```

Configuring the Application Context

How to Configure?

- Traditional XML
- Annotation Based
- Java Config

XML Based Configuration

```xml
<?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">

    <bean id="widgetOrderServiceBean"
          class="com.frankmoley.intro.example.impl.WidgetOrderServiceImpl"></bean>

</beans>
```

This is considered dead by SpringSource

XML Configuration

- Original form on configuration of the application context, precedes JDK 1.5 (5.0) when annotations where introduced
- Supports inheritance, namespaces, and “verbose” bean definition
- Still supported fully in Spring 4.0, however spring.io examples are limited in favor of Java Configuration

Annotation Configuration

- Component Scanning requires a hook (XML or Java Config)
- Classes in package are scanned for @Component or stereotype of it
- Beans can be injected at Constructor (preferred), field, or method
- Ambiguous definitions require @Qualifier
- JSR 330 Annotations supported

Java Configuration

- Java class based configuration, class annotated with @Configuration
- @Bean defines bean definitions
- Roughly similar to XML, few minor features removed, several gained

Java Config - Example Project

Dependency Injection

Field Based Injection

- Simple and easy, however has many downfalls
- Inability to test cleanly is the primary reason you should avoid it
- Can tie your code to a DI framework, not usually the best practice
- Breaks OOP paradigm
- Only supported by annotation based configs

Constructor Based Injection

- Simple, and preferred method for required dependencies
- Can support XML, Annotation, or Java Config
- Maintains OOP paradigm

Property Based Injection

- Preferred method for optional dependencies (or those modified during runtime)
- Can be used with Annotations, XML, or Java Config

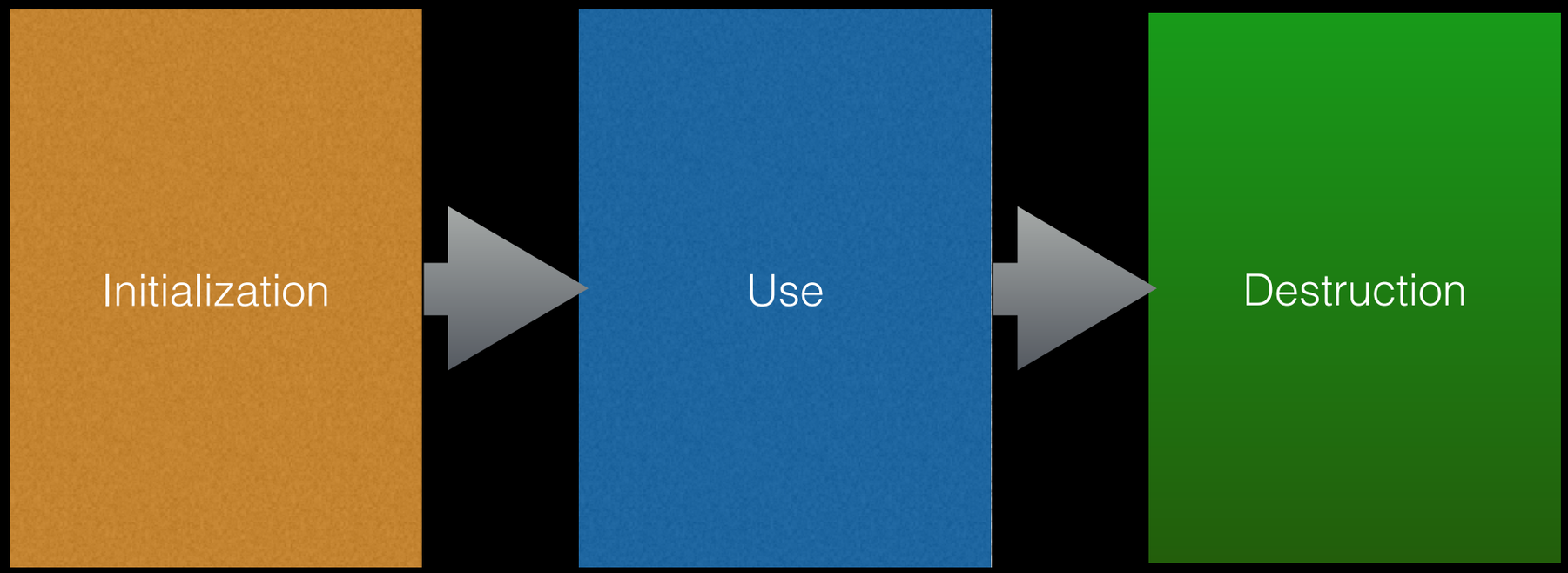
Take a Break

Be back in 10 minutes please

Bean Lifecycle

aka - the most important topic today

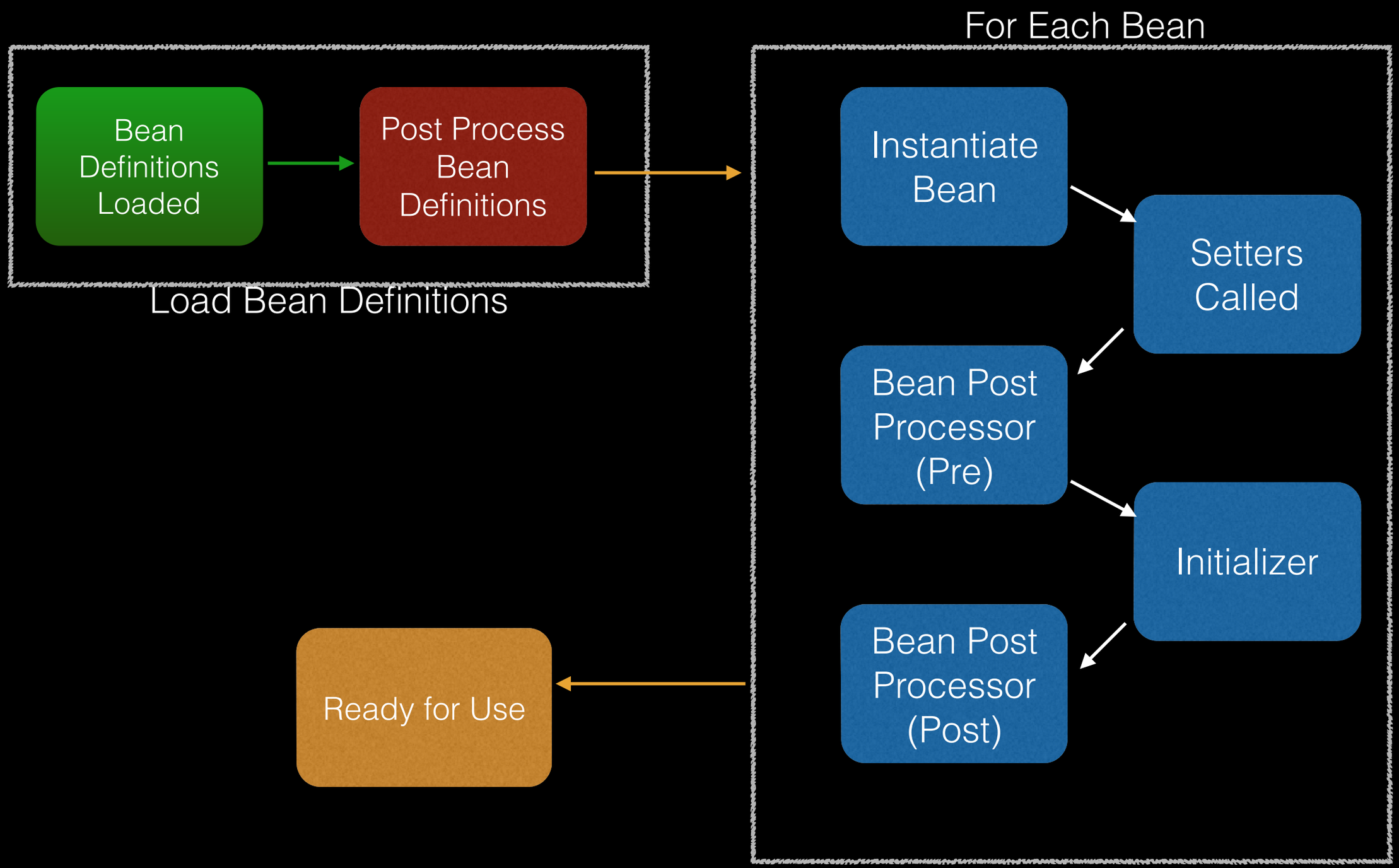
Introduction to the Lifecycle



Initialization Introduction

- Begins with the creation of the Application Context
- Bean Factory Initialization Steps
- Bean Initialization and Instantiation Steps

Overall Picture



Load Bean Definitions

- Starts with creation of ApplicationContext
- Context configures the bean factory and the beans contained in it
- Bean definitions identified via parsing or discovery
- Operations at this phase apply to all beans

Bean Factory Initialization Steps

- Parse Bean Definitions, as appropriate
- Load Bean Definitions into Bean Factory

Bean Definitions Loaded

Bean Factory Initialization Steps (Cont)

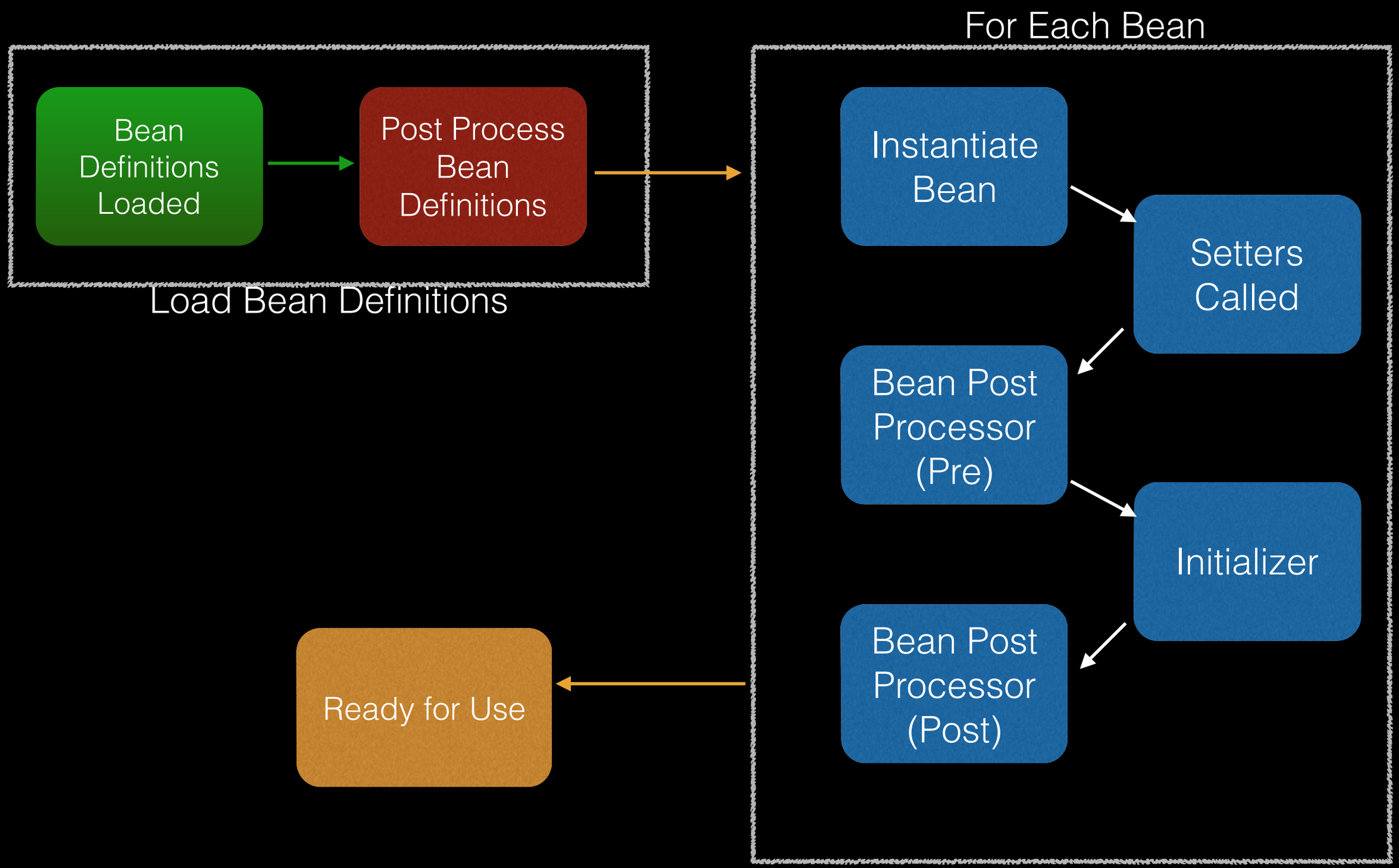
- Post Processing of Bean Definitions occurs here
- First major extension point -> BeanFactoryPostProcessor Interface
- Most Common one used = PropertyPlaceholderConfigurer

Post Process Bean Definitions

What has happened so far?

- Beans definitions have been loaded into factory, indexed by ID not name
- Component Scanning
- XML Parsing
- Java Config class loading and parsing
- Beans have been Initialized

Overall Picture



Bean Creation Operations

- Occurs after bean factory is configured
- Occurs on each bean in the factory, beans definitions already in heap
- Bean instances are created and maintained by the container for use

Instantiate Beans

- Beans are Instantiated -> singleton instances
- Dependency graph was identified during bean factory phase, used to determine order of instantiation
- Constructor injections occurs here, since the constructor is used :)

Soap Box Time

- Notice that at this phase, dependencies that are part of the constructor are injected, not property based dependencies
- Spring promotes good Object Oriented Practices
- If your dependency is required for your bean to operate, it should be injected in the constructor
- Properties should be used for optional or conditional dependencies only

Prototype & Lazy Beans

- Take note that prototype beans are not instantiated at this point by default
- Only instantiated when needed, as such they live as initialized instances on the heap, but not instantiated
- Lazy loaded beans are not prototypes, they are not instantiated until needed but once they are defined they are singletons

Setters Called

- After all the beans have been Instantiated, setter methods are called
- Defined by Autowiring setters, autowiring with aspecting, or property definitions in configuration

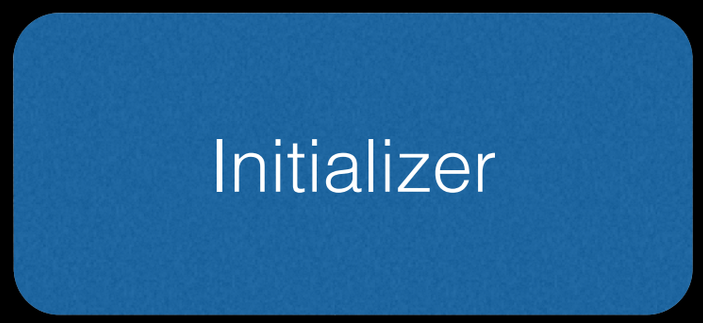
Setters Called

Bean Post Processors

- There are three “phases” of bean post processing
- These are processing on individual beans, not the factory
- 2 basic types in the three phases, Initializers and all the rest (pre and post)

Initializers First
but really Second

- Methods marked with @PostConstruct (1 per class) for annotations
- Methods defined in XML as init-method
- Point to note -> Beans are not ready for use so be very carefully using injected dependencies in the Initializer, proxies may not exist



Bean Post Processor

- BeanPostProcessor Interface
- Supports postProcessAfterInitialization and postProcessBeforeInitialization
- Not a common customer extension point, but used

Bean Post Processor (Pre)

Bean Post Processor (Post)

Ready for Use

- 99% of the time is spent in this phase of the Bean Lifecycle
- Beans can be accessed directly from the ApplicationContext or via Dependency Injection
- ApplicationContextAware interface allows a class controlled by Spring to have a handle on the container it is in

A Note on Proxies

- Proxies are created in the init phase by dedicated Spring Framework BeanPostProcessors

- With 4.0, every bean gets a proxy, no longer based solely on implicit aspects or annotations

- Need to be aware of the implications of calling self with proxies

Destruction

- When the context is closed or goes out of scope, there is a shutdown process
- Remember, only Garbage Collection can actually destroy the beans
- Interaction with the destruction phase can be important for closing connections safely or cleaning up other areas

Destroy Methods

- Methods annotated with @PreDestroy
- Methods listed as destroy-method (destroyMethod in Java Config)
- method must be no-arg void method
- Destroy Method(s) is/are executed then the context itself is destroyed
- Garbage Collection will then clean it all up

Questions so Far?

Take a Break

Be back in 10 minutes please

Quick Note on Testing

- Testing with Spring is as simple as wiring in the Application Context into your jUnit
- Can be more complex using the SpringJUnit4Runner and wiring in the AC there

Quick Note on Aspecting

- Aspecting allows you to add behavior to your Spring Beans without embedding the logic
- Great for logging, transactions, and other non-business logic that is required for your application
- Spring provides many aspecting annotations for common patterns, like transaction management

Data Access with Spring

Data Access Paradigm

- We will talk about 2 ways to access data against two separate data store types
- JDBC Template against RDBMS
- Spring-Data-JPA against RDBMS
- Spring-Data-Mongo against NoSQL database

JDBC Template

- JdbcTemplate handles the connection open, commit, rollback, and closing (with transactional help)
- Also handles ResultSet mapping to objects
- Simple enough, but there is a lot of code

My Strategy

- store sql statements in properties files
  - easier to give to DBA for validation
    - Advanced DBA teams can even modify in SCM and commit changes
- one property file per repository
  - Use common names through inherited classes, makes configuration easier in XML (if using)

My Strategy Cont

- Always use a NamedParameterJdbcTemplate implementation of JdbcTemplate, makes code more readable
- Keep transaction management on the Repository itself, if you need to aggregate transactions, fine, but at least you know where you have defined boundaries for sure

Infrastructure

```java
public abstract class PiiRepository {

    protected NamedParameterJdbcTemplate jdbcTemplate;
    protected Properties sqlProperties;
    protected final static String INSERT = "insert";
    protected final static String GET = "get";
    protected final static String UPDATE = "update";
    protected final static String DELETE = "delete";
    protected final static String FIND_ALL = "get_all";

    PiiRepository(DataSource dataSource, Properties sqlProperties){
        super();
        this.jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
        this.sqlProperties = sqlProperties;
    }
```

Example Implementation

```
public class PersonEntityRepository extends PiiRepository {

    private final static String PARAM_PERSON_ID = "personId";
    private final static String PARAM_PREFIX = "prefix";
    private final static String PARAM_FIRST_NAME = "firstName";
    private final static String PARAM_MIDDLE_NAME = "middleName";
    private final static String PARAM_LAST_NAME = "lastName";
    private final static String PARAM_SUFFIX = "suffix";

    public PersonEntityRepository(DataSource dataSource, Properties sqlProperties) {
        super(dataSource, sqlProperties);
    }
```

Example Method

```
@Transactional(propagation = Propagation.REQUIRED)
public PersonEntity addPerson(PersonEntity model){
    String personId = UUID.randomUUID().toString();
    String sql = this.sqlProperties.getProperty(INSERT);
    Map<String, Object> namedParameters = new HashMap<>(6);
    namedParameters.put(PARAM_PERSON_ID, personId);
    namedParameters.put(PARAM_PREFIX, StringUtils.trimToNull(model.getPrefix()));
    namedParameters.put(PARAM_FIRST_NAME, StringUtils.trimToNull(model.getFirstName()));
    namedParameters.put(PARAM_MIDDLE_NAME, StringUtils.trimToNull(model.getMiddleName()));
    namedParameters.put(PARAM_LAST_NAME, StringUtils.trimToNull(model.getLastName()));
    namedParameters.put(PARAM_SUFFIX, StringUtils.trimToNull(model.getSuffix()));
    jdbcTemplate.update(sql, namedParameters);
    return this.getPerson(personId);
}
```

Example XML Config

```
<context:property-placeholder location="classpath*:/piiDataServices.properties"/>
<context:component-scan base-package="com.frankmoley.services.pii"/>
<mvc:annotation-driven />

<jdbc:embedded-database id="piiDataSource" type="H2">
    <jdbc:script location="classpath*:/databaseCreate.sql"/>
    <jdbc:script location="classpath*:/preloadData.sql"/>
</jdbc:embedded-database>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg name="dataSource" ref="piiDataSource"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="piiRepository" class="com.frankmoley.services.pii.data.repository.PiiRepository" abstract="true">
    <constructor-arg name="dataSource" ref="piiDataSource"/>
</bean>

<bean id="personRepository" class="com.frankmoley.services.pii.data.repository.PersonEntityRepository" parent="piiRepository">
    <constructor-arg name="sqlProperties">
        <util:properties location="classpath:/com.frankmoley.services.pii.data.repository/person.properties"/>
    </constructor-arg>
</bean>
```

Benefits

- Spring handles all of the boilerplate code, setting up connections, tearing them down, rollback segments, etc.
- You focus on your business needs only
- Significant control over operations, I actually prefer this for RDBMS type systems because I can better control the SQL generated

Spring Data

Introduction

Why Spring Data

- Provides common interface for most data access technologies
- Allows the developer to swap out data sources with limited code changes
- Focus on the business logic, not on the data access technology

Repository Pattern

- Core object of Spring Data is the Repository
- Interface based on the Repository design pattern
- Get a piece of data, come back and get the next piece
- No massive join/aggregation paths in complex DAO methods

CRUD Repository

- The Crud Repository is the primary “implementation” of the Repository Pattern in Spring Data - it is an interface that Spring proxies with an implementation
- Extends the Repository marker interface
- Provides the expected operations; save, findOne, delete. Also supports findAll, exists, save (in batch mode), count, and delete (in batch mode)

PagingAndSortingRepository

- Extension of CrudRepository adds the ability to sort or page results from the findAll method
- Very useful in Restful Repository based micro services among other locations

Extending Repositories

- You can use standard language to add dynamic functionality to your repository
- Uses reflection based on bean notation to create queries
- Supported syntax is based on datastore technology, but all follows same syntax

Requirements

- Repository is based on generics, each definition requires two object definitions
- Entity definition
- Id definition
- The major difference between technologies at this point comes from the Entity definition

Example Entity

```java
@Entity
public class Foo implements Serializable {
    private static long serialVersionUID = 1L;

    @Id
    private long id;
    private String name;
    private long partNumber;
    private String serialNumber;
    private boolean active;

    public long getId() { return id; }

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

    public String getName() { return name; }

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

    public long getPartNumber() { return partNumber; }

    public void setPartNumber(long partNumber) { this.partNumber = partNumber; }

    public String getSerialNumber() { return serialNumber; }

    public void setSerialNumber(String serialNumber) { this.serialNumber = serialNumber; }

    public boolean isActive() { return active; }

    public void setActive(boolean active) { this.active = active; }
}
```

Example Repository

```
@Repository
public interface FooRepository extends PagingAndSortingRepository<Foo, Long> {

    Foo findByName(String name);

    List<Foo> findByNameLike(String name);

    Foo findByPartNumber(long partNumber);

    List<Foo> findByNameOrSerialNumber(String name, String serialNumber);

    List<Foo> findByActiveIsTrueOrderByPartNumberAsc();
}
```

Benefits

- By utilizing the Repository pattern and interfaces, creating data access is similar if not identical across several different data storage engines
- Swapping out datasources becomes trivial, as we will see

Spring-Data-JPA

ORM Abstraction

- Builds off of the JPA model to handle all aspects of EntityManager
- Transactional boundaries still managed in your code (or preferred through transactional proxies)
- No need to interact with Hibernate or any other JPA implementation, Spring handles the dependencies

First Class Entity Support

- Support of relationship models: @OneToOne, @OneToMany etc
- EntityManager injection through Spring allows for the container to manage the EntityManager as well as associated caches
- Full transaction support through EntityManager and Transactional Proxies

Spring Data MongoDB

NoSql Support

- MongoDB is just an example, most NoSql data packages are the same with respect to Spring
- Support for Neo4j, CouchDB, Gemfire, Hadoop, ElasticSearch, and others
- Reduces need to learn or remember the driver details, just implement it with Spring Data

Same Paradigm

- Still just uses a repository and an entity
- Query syntax may require some special considers with the datasource, for instance gemFire requires the entity to be on the server for OQL
- Mongo supports the ability to index fields on the document

Rest Repositories

- Just wanted to mention a quick note on RestRepository annotation
- Takes a Spring-Data-Repository and exposes it as a RESTful service
- Very powerful in a microservices world, talking about it tomorrow

Break Time

Spring Web MVC

What is Spring MVC?

- Web Framework based on the Model-View-Controller Pattern
- Based on common Spring concepts
  - POJOs, Testable Components
- Supports many view technologies, if you want to learn only one, use Thymeleaf

Servlets…

- Request paradigm is as such
  - Request comes in on a specific URL
  - Dispatcher Servlet handles request, converts URL to method via RequestMapping
  - Calls appropriate method, to get view name and model (as appropriate)
  - Consults ViewResolver, then renders the view and the model

Dispatcher Servlet

- The heart of Spring-MVC
- Defined in web.xml
- Uses Spring for its configuration
- Creates a separate private application context for the servlet
- Full access to parent Application Context by default

Basic Config Example

```
public class WebInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext applicationContext = new AnnotationConfigWebApplicationContext();
        applicationContext.register(WebAppConfiguration.class);
        servletContext.addListener(new ContextLoaderListener(applicationContext));

        applicationContext.setServletContext(servletContext);

        ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(applicationContext));
        servlet.addMapping("/");
        servlet.setLoadOnStartup(1);

    }
}
```

Controllers

- @Controller - Stereotype of @Component annotation specific for Web MVC classes
- @RequestMapping indicates where the method maps to in relation to the root servlet context
- Preferred method is to use annotations on Controllers exclusively
- More on mappings later

View Resolver

- Default View Resolver maps to WEB-INF relative to the application
- Maps to NAME.jsp where NAME is the String returned from the controller method
- Can be overridden at will, which we will for Thymeleaf

Break Time

Spring REST

Spring Rest

- Based on Servlets
- Primary Servlet is the Dispatcher Servlet, handles primary job of dispatching requests to lower level controls
- Defined in web.xml as the servlet on record
- Contains handle to WebApplicationContext which is the primary interface for the IoC bean factory managed by Spring

Controllers

- All interactions with the underlying services occur in the Controller
- Defines the RequestMappings
- Dispatcher Servlet delegates requests to the controller that expresses the appropriate mapping
- Handle to controllers comes from WebApplicationContext (BeanFactory)

Request Mapping

- The core component of the Controller in Spring Web MVC
- Maps a specific request to a service method
  - URI specific
  - Verb specific
  - Can be specific to parameters as well

Controller Methods

- Web applications typically return a String object that is the view to load, resolved by a View Resolver (ie return “index” and let the view resolver translate that to /WEB-INF/views/index.html)
- For REST, we can return a first class object and apply a @ResponseBody annotation on the method to allow the Accept header drive marshaling of the object

Difference with Spring MVC (since 4.0)

- Jackson Libraries changed (from 3.2)
- Introduction of @RestController instead of @Controller
- @ResponseBody can apply to class so methods inherit it naturally (included in @RestController annotation)
- Full support for servlet 3.0

Notes on Spring JMS

- Provides a Template class, similar to JdbcTemplate that allows connections to JMS systems
- Reduces a ton of boiler plate code

Notes on JMX

- Does anyone really use this?
- Expose any POJO or its properties through JMX console with Spring
- Execute operations as well, need to use a RefreshableApplicationContext (not advised usually) if you want to change bean behavior

Spring Boot

The New Hotness

- Get applications running very quickly
- Embed an Application Server if desired
- Our examples today used it

Questions?

Lets Code!