Introduction to Enterprise Java Frameworks

jim stafford

Fall 2022 v2019-11-12: Built: 2022-12-07 06:10 EST

Table of Contents

1. Introduction . . . 1
   1.1. Goals . . . 1
   1.2. Objectives . . . 1
2. Code Reuse . . . 2
   2.1. Code Reuse Trade-offs . . . 2
   2.2. Code Reuse Constructs . . . 2
   2.3. Code Reuse Styles . . . 2
3. Frameworks . . . 4
   3.1. Framework Informal Description . . . 4
   3.2. Framework Characteristics . . . 4
4. Framework Enablers . . . 6
   4.1. Dependency Injection . . . 6
   4.2. POJO . . . 6
   4.3. Component . . . 6
   4.4. Bean . . . 6
   4.5. Container . . . 7
   4.6. Interpose . . . 7
5. Language Impact on Frameworks . . . 10
   5.1. XML Configurations . . . 10
   5.2. Annotations . . . 10
   5.3. Lambdas . . . 11
6. Key Frameworks . . . 12
   6.1. CGI Scripts . . . 12
   6.2. JavaEE . . . 12
   6.3. Spring . . . 12
   6.4. Jakarta Persistence API (JPA) . . . 13
   6.5. Spring Data . . . 13
   6.6. Spring Boot . . . 13
7. Summary . . . 15

Chapter 1. Introduction

1.1. Goals

The student will learn:

- constructs and styles for implementing code reuse
- what is a framework
- what has enabled frameworks
- a historical look at Java frameworks

1.2. Objectives

At the conclusion of this lecture, the student will be able to:

1. identify the key difference between a library and framework
2. identify the purpose for a framework in solving an application solution
3. identify the key concepts that enable a framework
4. identify specific constructs that have enabled the advance of frameworks
5. identify key Java frameworks that have evolved over the years

Chapter 2. Code Reuse

Code reuse is the use of existing software to create new software. [1]

We leverage code reuse to help solve either repetitive or complex tasks so that we are not repeating ourselves, we reduce errors, and we achieve more complex goals.

2.1. Code Reuse Trade-offs

On the positive side, we do this because we have confidence that we can delegate a portion of our job to code that has been proven to work. We should not need to again test what we are using.

On the negative side, reuse can add dependencies bringing additional size, complexity, and risk to our solution. *If all you need is a spoon — do you need to bring the entire kitchen?*

2.2. Code Reuse Constructs

Code reuse can be performed using several structural techniques

Method Call

We can wrap functional logic within a method within our own code base. We can make calls to this method from the places that require that task performed.

Classes

We can capture state and functional abstractions in a set of classes. This adds some modularity to related reusable method calls.

Interfaces

Abstract interfaces can be defined as placeholders for things needed but supplied elsewhere. This could be because of different options provided or details being supplied elsewhere.

Modules

Reusable constructs can be packaged into separate physical modules so that they can be flexibly used or not used by our application.

2.3. Code Reuse Styles

There are two basic styles of code reuse and they primarily have to to with **control**.

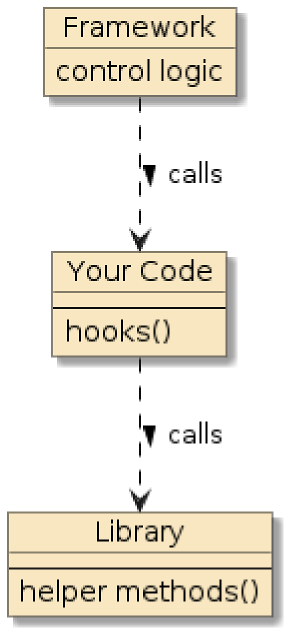

Figure 1. Library/ Framework/Code Relationship [2]

Libraries

Libraries are modules of reusable code that are invoked on-demand by your code base. Your code is in total control of the library call flow.

- Examples: JSON or XML parser

Frameworks

Frameworks are different from callable libraries — in that they provide some level of control or orchestration. Your code base is under the control of the framework. This is called **"Inversion of Control"**.

- Examples: Spring/Spring Boot, JakartaEE (formerly JavaEE)

Its not always a one-or-the-other style. Libraries can have mini frameworks within them. Even the JSON/XML parser example can be a mini-framework of customizations and extensions.

[1] *"Code reuse"*,"Wikipedia"

Chapter 3. Frameworks

3.1. Framework Informal Description

A successful software framework is a body code that has been developed from the skeletons of successful and unsuccessful solutions of the past and present within a common domain of challenge. A framework is a generalization of solutions that provides for key abstractions, opportunity for specialization, and supplies default behavior to make the on-ramp easier and also appropriate for simpler solutions.

- *"We have done this before. This is what we need and this is how we do it."*

A framework is much bigger than a pattern instantiation. A pattern is commonly at the level of specific object interactions. We typically have created or commanded something at the completion of a pattern — but we have a long way to go in order to complete our overall solution goal.

- *Pattern Completion: "that is not enough — we are not done"*
- *Framework Completion: "I would pay (or get paid) for that!"*

A successful framework is more than many patterns grouped together. Many patterns together is just a sea of calls — like a large city street at rush hour. There is a pattern of when people stop and go, make turns, speed up, or yield to let someone into traffic. Individual tasks are accomplished, but even if you could step back a bit — there is little to be understood by all the interactions.

- *"Where is everyone going?"*

A framework normally has a complex purpose. We have typically accomplished something of significance or difficulty once we have harnessed a framework to perform a specific goal. Users of frameworks are commonly not alone. Similar accomplishments are achieved by others with similar challenges but varying requirements.

- *"This has gotten many to their target. You just need to supply …"*

Well designed and popular frameworks can operate at different scale — not just a one-size-fits-all all-of-the-time. This could be for different sized environments or simply for developers to have a workbench to learn with, demonstrate, or develop components for specific areas.

- *"Why does the map have to be actual size?"*

3.2. Framework Characteristics

The following distinguishing features for a framework are listed on Wikipedia. [1] I will use them to structure some further explanations.

Inversion of Control (IoC)

Unlike a procedural algorithm where our concrete code makes library calls to external components, a framework calls our code to do detailed things at certain points. All the complex but reusable logic has been abstracted into the framework.

- *"Don’t call us. We’ll call you."* is a very common phrase to describe inversion of control

Default Behavior

Users of the framework do not have to supply everything. One or more selectable defaults try to do the common, right thing.

- *Remember — the framework developers have solved this before and have harvested the key abstractions and processing from the skeletal remains of previous solutions*

Extensibility

To solve the concrete case, users of the framework must be able to provide specializations that are specific to their problem domain.

- *Framework developers — understanding the problem domain — have pre-identified which abstractions will need to be specialized by users. If they get that wrong, it is a sign of a bad framework.*

Non-modifiable Framework code

A framework has a tangible structure; well-known abstractions that perform well-defined responsibilities. That tangible aspect is visible in each of the concrete solutions and is what makes the product of a framework immediately understandable to other users of the framework.

- *"This is very familiar."*

[1] *"Software framework"*, Wikipedia

Chapter 4. Framework Enablers

4.1. Dependency Injection

A process to enable Inversion of Control (IoC), whereby objects define their dependencies [1] and the manager (the "Container") assembles and connects the objects according to definitions.

The "manager" can be your setup code ("POJO" setup) or in realistic cases a "container" (see later definition)

4.2. POJO

A Plain Old Java Object (POJO) is what the name says it is. It is nothing more than an instantiated Java class.

A POJO normally will address the main purpose of the object and can be missing details or dependencies that give it complete functionality. Those details or dependencies are normally for specialization and extensibility that is considered outside of the main purpose of the object.

- *Example: POJO may assume inputs are valid but does not know validation rules.*

4.3. Component

A component is a fully assembled set of code (one or more POJOs) that can perform its duties for its clients. A component will normally have a well-defined interface and a well-defined set of functions it can perform.

A component can have zero or more dependencies on other components, but there should be no further **mandatory** assembly once your client code gains access to it.

4.4. Bean

A generalized term that tends to refer to an object in the range of a POJO to a component that encapsulates something. A supplied "bean" takes care of aspects that we do not need to have knowledge of.

> In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container. [1]
>
> — Spring.io, Introduction to the Spring IoC Container and Beans

You will find that I commonly use the term "component" in the lecture notes — to be a bean that is fully assembled and managed by the container.

4.5. Container

A container is the assembler and manager of components.

Both Docker and Spring are two popular containers that work at two different levels but share the same core responsibility.

4.5.1. Docker Container Definition

- Docker supplies a container that assembles and packages software so that it can be generically executed on remote platforms.

> A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. [2]
>
> — Docker.com, Use containers to Build Share and Run your applications

4.5.2. Spring Container Definition

- Spring supplies a container that assembles and packages software to run within a JVM.

> (The container) is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects. [3]
>
> — Spring.io, Container Overview

4.6. Interpose

Containers do more than just configure and assemble simple POJOs. Containers can apply layers of functionality onto beans when wrapping them into components. Examples:

- Perform validation
- Enforce security constraints
- Manage transaction for backend resource
- Perform Method in a separate thread

4.6.1. POJO Calls

The following two examples are examples of straight POJO calls. There is no interpose going on here.

In the first example, method `m1()` and `m2()` are in the same class (aka "buddy methods"). Method `m1()` calls sibling buddy method `m2()`. This call will be a straight POJO call. No container is involved between two methods of the same class unless there is a chance for sub-classing.

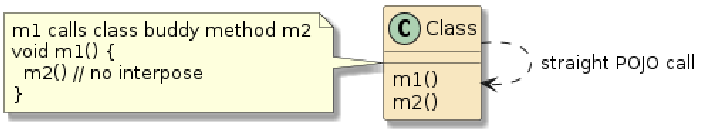

Figure 2. POJO Buddy Call

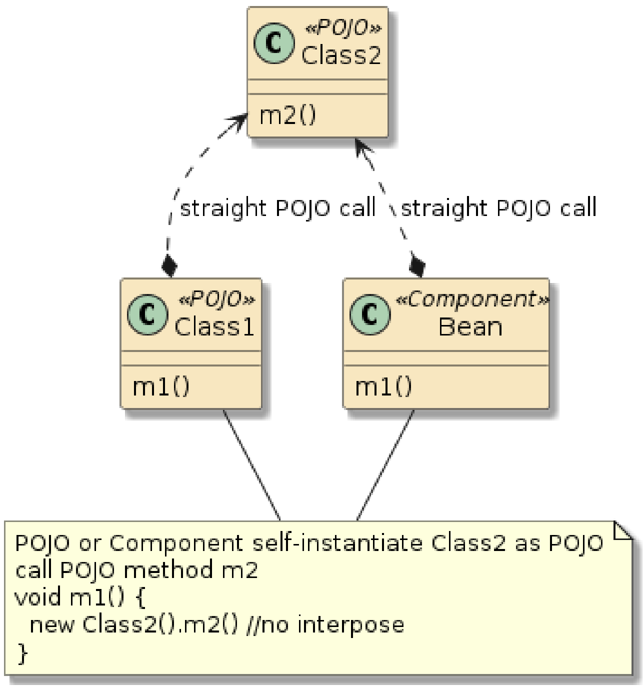

Figure 3. Self-Instantiated POJO Call

In the second example, method `m1()` and `m2()` are in two separate classes. Method `m2()` is inside `Class2`. Method `m1()` instantiates `Class2` and calls method `m2()`. This call will also be a straight POJO call no matter whether `m1()` is a POJO or component because `Class2` was instantiated outside the control of the container.

4.6.2. Container Interpose

In this third example, method `m1()` and method `m2()` are in two separate classes (`Class1` and `Class2`) — but those classes have been defined as beans to the container.

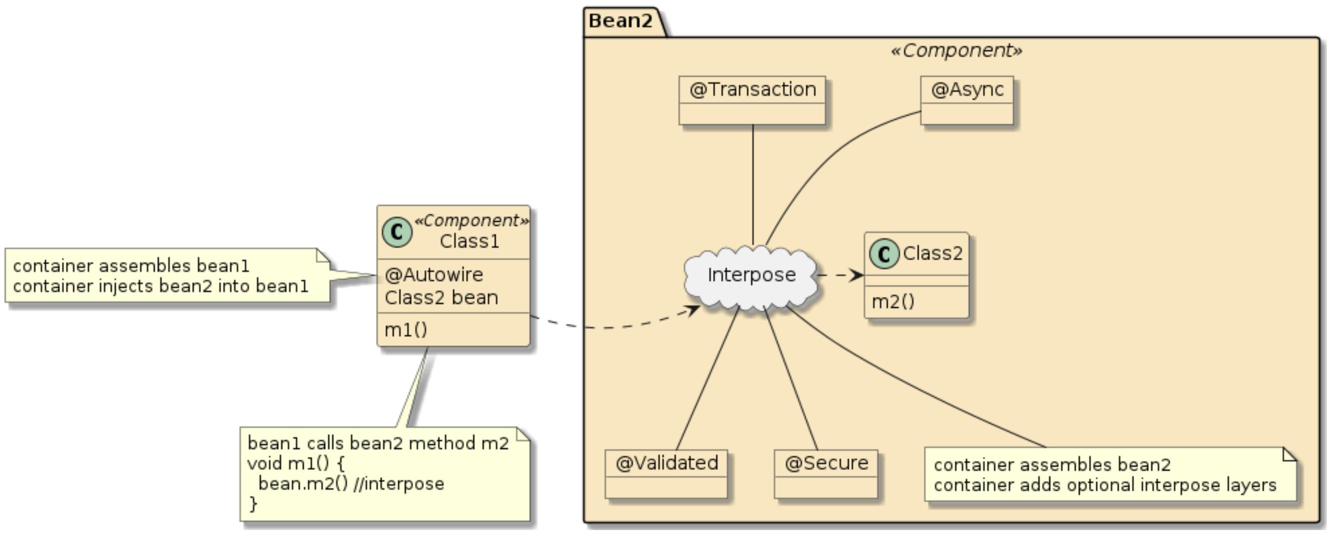
Bean2
«Component»
@Transaction
@Async
«Component»
Class1
container assembles bean1
container injects bean2 into bean1
@Autowire
Class2 bean
m1()
Interpose
Class2
m2()
bean1 calls bean2 method m2
void m1() {
bean.m2() //interpose
}
@Validated
@Secure
container assembles bean2
container adds optional interpose layers

Figure 4. Container Interpose

That means both `Class1` and `Class2` will be instantiated as beans by the container. `Bean2` will be augmented with zero or more layers of functionality — called interpose — to implement the full bean component definition. `Bean1` will have the `Bean2` injected to satisfy its `Class2` dependency and be augmented with whatever functionality its is required to complete its bean component definition

This is how features can be added to simple looking POJOs when we make them into beans.

[1] *"Spring Framework Documentation, The IoC Container"*, Spring.io

[2] *"Use containers to Build, Share and Run your applications"*, Docker.com

[3] *"The IOC Container, Container Overview"*, Spring Framework Documentation

Chapter 5. Language Impact on Frameworks

As stated earlier, frameworks provide a template of behavior — allowing for configuration and specialization. Over the years, the ability to configure and to specialize has gone through significant changes with language support.

5.1. XML Configurations

Prior to Java 5, the primary way to identify components was with an XML file. The XML file would identify a bean class provided by the framework user. The bean class would either implement an interface or comply with JavaBean getter/setter conventions.

5.1.1. Inheritance

Early JavaEE EJB defined a set of interfaces that represented things like stateless and stateful sessions and persistent entity classes. End-users would implement the interface to supply specializations for the framework. These interfaces had many callbacks that were commonly not needed but had to be tediously implemented with noop return statements — which produced some code bloat.

5.1.2. Java Reflection

Early Spring bean definitions used some interface implementation, but more heavily leveraged compliance to JavaBean setter/getter behavior and Java reflection. Bean classes listed in the XML were scanned for methods that started with "set" or "get" (or anything else specially configured) and would form a call to them using Java reflection. This eliminated much of the need for strict interfaces and noop boilerplate return code.

5.2. Annotations

By the time Java 5 and annotations arrived in 2005 (late 2004), the Java framework worlds were drowning in XML. During that early time, everything was required to be defined. There were no defaults.

Although changes did not seem immediate, the JavaEE frameworks like EJB 3.0/JPA 1.0 provided a substantial example for the framework communities in 2006. They introduced "sane" defaults and a primary (XML) and secondary (annotation) override system to give full choice and override of how to configure. Many things just worked right out of the box and only required a minor set of annotations to customize.

Spring went a step further and created a Java Configuration capability to be a 100% replacement for the old XML configurations. XML files were replaced by Java classes. XML bean definitions were replaced by annotated factory methods. Bean construction and injection was replaced by instantiation and setter calls within the factory methods.

Both JavaEE and Spring supported class level annotations for components that were very simple to instantiate and followed standard injection rules.

5.3. Lambdas

Java 8 brought in lambdas and functional processing, which from a strictly syntactical viewpoint is primarily a shorthand for writing an implementation to an interface (or abstract class) with only one abstract method.

You will find many instances in modern libraries where a call will accept a lambda function to implement core business functionality within the scope of the called method. Although — as stated — this is primarily syntactical sugar, it has made method definitions so simple that many more calls take optional lambdas to provide convenient extensions.

Chapter 6. Key Frameworks

In this section I am going to list a limited set of key Java framework highlights. In following the primarily Java path for enterprise frameworks, you will see a remarkable change over the years.

6.1. CGI Scripts

The Common Gateway Interface (CGI) was the cornerstone web framework when Java started coming onto the scene. [1] CGI was created in 1993 and, for the most part, was a framework for accepting HTTP calls, serving up static content and calling scripts to return dynamic content results. [2]

The important parts to remember is that CGI was 100% stateless relative to backend resources. Each dynamic script called was a new, heavyweight operating system process and new connection to the database. Java programs were shoehorned into this framework as scripts.

6.2. JavaEE

Jakarta EE, formerly the Java Platform, Enterprise Edition (JavaEE) and Java 2 Platform, Enterprise Edition (J2EE) is a framework that extends the Java Platform, Standard Edition (Java SE) to be an end-to-end Web to database functionality and more. [3] Focusing only on the web and database portions here, JakartaEE provided a means to invoke dynamic scripts — written in Java — within a process thread and cached database connections.

The initial versions of Jakarta EE aimed big. Everything was a large problem and nothing could be done simply. It was viewed as being overly complex for most users. Spring was formed initially as a means to make J2EE simpler and ended up soon being an independent framework of its own.

J2EE first was released in 1999 and guided by Sun Microsystems. The Servlet portion was likely the most successful portion of the early release. The Enterprise Java Beans (EJB) portion was not realistically usable until JavaEE 5 / post 2006. By then, frameworks like Spring had taken hold of the target community.

In 2010, Sun Microsystems and control of both JavaSE and JavaEE was purchased by Oracle and seemed to progress but on a slow path. By JavaEE 8 in 2017, the framework had become very Spring-like with its POJO-based design. In 2017, Oracle transferred ownership of JavaEE to Jakarta. The framework seems to have paused for a while for naming changes and compatibility releases. [3]

6.3. Spring

Spring 1.0 was released in 2004 and was an offshoot of a book written by Rod Johnson "Expert One-on-One J2EE Design and Development" that was originally meant to explain how to be successful with J2EE. [4]

In a nutshell, Rod Johnson and the other designers of Spring thought that rather than starting with a large architecture like J2EE, one should start with a simple bean and scale up from there without boundaries. Small Spring applications were quickly achieved and gave birth to other frameworks like the Hibernate persistence framework (first released in 2003) which significantly influenced the EJB3/JPA standard. [5]

6.4. Jakarta Persistence API (JPA)

The Jakarta Persistence API (JPA), formerly the Java Persistence API, was developed as a part of the JavaEE community and provided a framework definition for persisting objects in a relational database. JPA fully replaced the original EJB Entity Beans standards of earlier releases. It has an API, provider, and user extensions. [6] The main drivers of JPA where EclipseLink (formerly TopLink from Oracle) and Hibernate.

> *Frameworks should be based on the skeletons of successful implementations*
>
> Early EJB Entity Bean standards (< 3) were not thought to have been based on successful implementations. The persistence framework failed to deliver, was modified with each major release, and eventually replaced by something that formed from industry successes.

JPA has been a wildly productive API. It provides simple API access and many extension points for DB/SQL-aware developers to supply more efficient implementations. JPA's primary downside is likely that it allows Java developers to develop persistent objects without thinking of database concerns first. One could hardly blame that on the framework.

6.5. Spring Data

Spring Data is a data access framework centered around a core data object and its primary key — which is very synergistic with Domain-Driven Design (DDD) Aggregate and Repository concepts. [7]

- Persistence models like JPA allow relationships to be defined to infinity and beyond.
- In DDD the persisted object has a firm boundary and only IDs are allowed to be expressed when crossing those boundaries.
- These DDD boundary concepts are very consistent with the development of microservices — where large transactional, monoliths are broken down into eventually consistent smaller services.

By limiting the scope of the data object relationships, Spring has been able to automatically define an extensive CRUD (Create, Read, Update, and Delete), query, and extension framework for persisted objects on multiple storage mechanisms.

We will be working with Spring Data JPA and Spring Data Mongo in this class. With the bounding DDD concepts, the two frameworks have an amazing amount of API synergy between them.

6.6. Spring Boot

Spring Boot was first released in 2014. Rather than take the "build anything you want, any way you want" approach in Spring, Spring Boot provides a framework for providing an opinionated view of how to build applications. [8]

- By adding a dependency, a default implementation is added with "sane" defaults.
- By setting a few properties, defaults are customized to your desired settings.
- By defining a few beans, you can override the default implementations with local choices.

There is no external container in Spring Boot. Everything gets boiled down to an executable JAR and launched my a simple Java main (and a lot of other intelligent code).

Our focus will be on Spring Boot, Spring, and lower-level Spring and external frameworks.

[1] *"Write CGI programs in Java"*, InfoWorld 1997

[2] *"Common Gateway Interface"*, Wikipedia

[3] *"Jakarta EE"*, Wikipedia

[4] *"Spring Framework"*, Wikipedia

[5] *"Hibernate (framework)"*,Wikipedia

[6] *"Jakarta Persistence"*, JPA

[7] *"Domain-Driven Design Reference"*,Eric Evans Domain Language, Inc. 2015

[8] *"History of Spring Framework and Spring Boot"*,Quick Programming Tips

Chapter 7. Summary

In this module we:

- identified the key differences between a library and framework
- identify the purpose for a framework in solving an application solution
- identify the key concepts that enable a framework
- identify specific constructs that have enabled the advance of frameworks
- identify key Java frameworks that have evolved over the years