Spring was created to address the complexity of enterprise application development, and makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. But Spring’s usefulness isn’t limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

Spring does a lot of things. But when you break it down to its core parts, spring’s primary features are dependency injection (DI) and aspect-oriented programming (AOP).

How does Spring simplify Java development? To back up its attack on Java complexity, Spring employs four key strategies:

* Lightweight and minimally invasive development with plain old Java objects(POJOs)
* Loose coupling through dependency injection and interface orientation
* Declarative programming through aspects and common conventions
* Boilerplate reduction through aspects and templates

**DI:**

Dependency injection involves giving an object its dependencies as opposed to an object having to acquire those dependencies on its own.

**AOP:**

Although DI makes it possible to tie software components together loosely, aspect-oriented programming enables you to capture functionality that’s used throughout your application in reusable components.

Aspect-oriented programming is often defined as a technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often these components also carry additional responsibility beyond their core functionality. System services such as logging, transaction management, and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system. By spreading these concerns across multiple components, you introduce two levels of complexity to your code:

* The code that implements the system wide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you’ll need to visit multiple components. Even if you’ve abstracted the concern to a separate module so that the impact to your components is a single method call, that method call is duplicated in multiple places.
* Your components are littered with code that isn’t aligned with their core functionality. A method to add an entry to an address book should only be concerned with how to add the address and not with whether it’s secure or transactional.

**Spring Security**

Spring Security provides comprehensive security services for J2EE-based enterprise software applications. People use Spring Security for many reasons, but most are drawn to the project after finding the security features of J2EE's Servlet Specification or EJB Specification lack the depth required for typical enterprise application scenarios. They are not portable at a WAR or EAR level. Therefore, if you switch server environments, it is typically a lot of work to reconfigure your application's security in the new target environment. Using Spring Security overcomes these problems, and also brings you dozens of other useful, customisable security features.

As you probably know two major areas of application security are “authentication” and “authorization” (or“access-control”). These are the two main areas that Spring Security targets. “Authentication” is the process of establishing a principal is who they claim to be (a “principal” generally means a user, device or some other system which can perform an action in your application). “Authorization” refers to the process of deciding whether a principal is allowed to perform an action within your application. To arrive at the point where an authorization decision is needed, the identity of the principal has already been established by the authentication process. These concepts are common, and not at all specific to Spring Security.

At an authentication level, Spring Security supports a wide range of authentication models.

* HTTP BASIC authentication headers (an IETF RFC-based standard)
* HTTP Digest authentication headers (an IETF RFC-based standard)
* HTTP X.509 client certificate exchange (an IETF RFC-based standard)
* LDAP (a very common approach to cross-platform authentication needs, especially in large environments)
* Form-based authentication (for simple user interface needs)
* OpenID authentication
* Authentication based on pre-established request headers (such as Computer Associates Siteminder)
* JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign-on system)
* Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol)
* Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time)
* Anonymous authentication (allowing every unauthenticated call to automatically assume a particular security identity)
* Run-as authentication (which is useful if one call should proceed with a different security identity)
* Java Authentication and Authorization Service (JAAS)
* JEE container autentication (so you can still use Container Managed Authentication if desired)
* Kerberos
* Java Open Source Single Sign On (JOSSO) \*
* OpenNMS Network Management Platform \*
* AppFuse \*
* AndroMDA \*
* Mule ESB \*
* Direct Web Request (DWR) \*
* Grails \*
* Tapestry \*
* JTrac \*
* Jasypt \*
* Roller \*
* Elastic Path \*
* Atlassian Crowd \*
* Your own authentication systems