In short, Spring framework helps you build web applications. It takes care of dependency injection, handles transactions, implements an MVC framework and provides foundation for the other Spring frameworks (including Spring Boot)

While you can do everything in Spring without Spring Boot, Spring Boot helps you get things done faster:

* simplifies your Spring dependencies, no more version collisions
* can be run straight from a command line without an application container
* build more with less code - no need for XML, not even web.xml, auto-configuration
* useful tools for [running in production](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints), [database initialization](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-database-initialization),  [environment](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \l "boot-features-profiles" \t "_blank)specific [config files](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \l "boot-features-external-config" \t "_blank), [collecting metrics](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics)

URI

Uniform Resource Identifier (URI) is a string of characters used to identify a name or a resource on the Internet

A URI identifies a resource either by location, or a name, or both. **A URI has two specializations known as URL and URN.**

A Uniform Resource Locator (URL) is a subset of the Uniform Resource Identifier (URI) that specifies where an identified resource is available and the mechanism for retrieving it.URL defines how the resource can be obtained. It does not have to be HTTP URL (http://), a URL can also be (ftp://) or (smb://)

A Uniform Resource Name (URN) is a Uniform Resource Identifier (URI) that uses the URN scheme, and **does not imply availability of the identified resource**. Both URNs (names) and URLs (locators) are URIs, and a particular URI may be both a name and a locator at the same time.

The URNs are part of a larger Internet information architecture which is composed of URNs, URCs and URLs.

**bar.html is not a URN**. A URN is similar to a person's name, while a URL is like a street address. The URN defines something's identity, while the URL provides a location. Essentially, "what" vs. "where". A URN has to be of this form <URN> ::= "urn:" <NID> ":" <NSS> where <NID> is the Namespace Identifier, and <NSS> is the Namespace Specific String.

**To put it differently:**

* A URL is a URI that identifies a resource and also provides the means of locating the resource by describing the way to access it
* A URL is a URI
* A URI is not necessarily a URL

I'd say the only thing left to make it 100% clear would be to have an example of an URI that is not an URL. We can use the examples in the [RFC3986](http://www.ietf.org/rfc/rfc3986.txt):

URL: ftp://ftp.is.co.za/rfc/rfc1808.txt

URL: http://www.ietf.org/rfc/rfc2396.txt

URL: ldap://[2001:db8::7]/c=GB?objectClass?one

URL: mailto:John.Doe@example.com

URL: news:comp.infosystems.www.servers.unix

URL: telnet://192.0.2.16:80/

URN (not URL): urn:oasis:names:specification:docbook:dtd:xml:4.1.2

URN (not URL): tel:+1-816-555-1212 (?)

**Statelessness**

A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently. HTTP is a stateless protocol by design and you need to do something extra to implement a stateful service using HTTP. But it is really easy to implement stateful services with current technologies. We need a clear understanding of a stateless and stateful design so that we can avoid misinterpretation.

A stateless design looks like so:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

Each of these requests can be treated separately.

A stateful design, on the other hand, looks like so:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/NextPerson HTTP/1.1

To process the second request, the server needs to remember the last PersonID that the client fetched. In other words, the server needs to remember the current state — otherwise Request2 cannot be processed. Design your service in a way that a request never refers to a previous request. Stateless services are easier to host, easy to maintain, and more scalable. Plus, such services can provide better response time to requests, as it is much easier to load balance them.