Applause from Michelé Clarke, Tariq Ellis, and 2 others

Adrian D. Finlay
@thewiprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ adriandavid.me/network
Oct 1, 2017 · 16 min read

How To Become a Java Web Developer (2017)—Part II: Enterprise Fundamentals, Data, & The Web

Java Banner [10]

Java EE—Building an Enterprise System

> *"Since that first release, the platform has become the foundation of server-side development for much of the corporate world, being adopted by and improved by industry heavyweights such as IBM and Oracle."*—Danny Coward, Principal Architect, Oracle Co. [14].

The Java Enterprise Edition is a platform for developing and deploying enterprise software—that is web and networking software; **It is defined by specification**. This is unlike the other editions of Java, namely, ME and SE. By providing a specification for EE components, this leaves the software vendor with more control over how said components are managed and implemented by the application sever.

My opinion: Knowledge of the Java Enterprise Edition is **compulsory knowledge** to Java Web & Enterprise developers, even if they wind up developing in Web/Enterprise Frameworks such as Struts II, Spring MVC, Grails, Play, etc.

There are several core components of the platform that you should be aware of. Everyone will not agree as to what should or should or not be known. In the end, one should probably invest the majority of one's time into the things that one will actually use. Nevertheless these are the **core ideas.** I will break them down into categories:

Web Components: Servlets, JSP/JSTL/EL, JSF

Servlets are the foundation of the Web Components of the Java EE Platform. Java Servlets provide the functionality to do next to anything with a client-server protocol such as HTTP and a markup language like HTML. A Java servlet is an Java object that processes the server side of HTTP interactions[14].

JavaServer Pages (JSPs) are inside-out servlets. This model came from the observation that wrapping static HTML inside the output stream of a Java servlet was counterintuitive for web designers and led to large, messy, code. Instead of putting static HTML into Java Code, JSPs dynamically put Java code into HTML. This, among many improvements led to a more intuitive web view approach and provided a way to dynamically inject Java components such as beans into static HTML. The **JavaServer Pages Standard Tag Library (JSTL)** is a Java EE Web Component that adds to the JSP specification by providing a standard tag library of JSP tags for various tasks.

Unified Expression Language (EL) is a small language that was design to read and process the values of Enterprise Java Bean but is now used to generally process data in a JSP. EL can use Java Beans, JSP EL Literals, and JSP implicit objects.

JavaServer Faces (JSFs) take JavaServer Pages to the next level by providing a MVC Framework based on reusable components.

Web Services: REST via JAX-RS + JSON-P, Jersey, Familiarity with SOAP

JAX-RS is a Java EE specification for the creating of web services according Representational State Transfer (REST) architecture. **Jersey** is a framework for creating RESTful Web Services. Unlike JAX-RS, it is not a Java EE specification but rather the JAX-RS reference implementation. These are not the same—**without an implementation, Java EE specifications can't do anything, they are merely a definition provided so as to ensure common core functionality across Java EE compliant Application servers**. As mentioned earlier, designing the EE platform by specification has allowed Java EE compliant vendors more freedom. Consequently, if you limit yourself to the JAX-RS API, it should run the same (though it may be implemented differently) on all Java EE Compliant Application Servers. I mention this to say that Jersey extends the core functionality described by the JAX-RS specification. Hence, all JAX-RS is valid Jersey but not all Jersey is valid JAX-RS. The reason for listing Jersey here is that it has a very large market share among Java developers for REST development and it further simplifies the development of RESTful web services. **JSON-P** is Java's API for **JSON (JavaScript Object Notation)** processing. The reason for mentioning JSON-P is that RESTful Web Resources are commonly exposed as JSON.

SOAP stands for Simple Object Access Protocol. SOAP is neither a specification implementation nor is it a part of the Java EE Platform. SOAP is an architectural specification. Like REST, SOAP is a means by which people approach the development of Web Services. Like REST, Java provides support for SOAP via **JAX-WS**. Modern Software Development has largely abandoned SOAP oriented approaches for the development of web services in favour of REST oriented approaches. This is controversial, however, and **it is indeed true that SOAP is still used for new work**. However, the trends show that RESTful approaches are overwhelmingly used for new web services work above SOAP oriented approaches. Nevertheless, **because developers are often working on older systems and SOAP is still used to some degree for new work, I have listed Java's SOAP support here for completeness**. Web Resources exposed via the SOAP protocol are commonly exposed as XML.

Web Sockets: Java WebSockets API

Java API for WebSockets is a specification that provides APIs to handle WebSocket connections. WebSockets are unique among the Web Components in the Java EE API in that they can they push data to the client without the client having to request it (i.e. an HTTP GET request).

In deeper detail: “Java WebSockets are a departure from the HTTP-based interaction model, providing a way for Java EE applications to update browser and non-browser clients asynchronously. For many kinds of web applications, having the user always in the driver’s seat is not desirable.
From financial applications with live market data, to auction applications where people around the world bid on items, to the lowly chat and presence applications, web developers have long sought means by which the server side of the web application can push data out to the client. The WebSocket protocol is a TCP-based protocol that provides a full duplex communication channel over a single connection. In simple terms, this means that it uses the same underlying network protocol as does HTTP and that over a single WebSocket connection both parties can send messages to the other at the same time.”—Java EE 7, The Big Picture ~ Danny Coward[14]

Security: Web Component Security, Bean Security

Web Component & Enterprise Java Bean Security mainly involve two notions: the **declarative security model** and the **programmatic model.** To be brief, the declarative security model describes (in metadata) the protection model that you want the web container to apply to the application. This metadata can be defined in the deployment descriptor and can also be specified via certain annotations in source code. The fundamental concepts of the declarative security model involve authentication, authorization, and establishing data privacy based on user roles. The programmatic model involves building against the Java Security API’s to define you own security framework.

Security skills are critical to all Software Developers. The recent Equifax [31] and Yahoo! [32] breaches further cement this very serious fact. Globally it is estimated that 4 billion data records were stolen in 2016 [33].

While Enterprise Developers should not be expected to possess the skills of a a security engineer, they should **always write code with security in mind**. A detailed conversation regarding security could fill many books and is beyond the scope of this article. However, you cannot afford to code in ignorance of security. The evidence is daunting. The risks are very real.

Middle Tier: Enterprise Java Beans, Accessing EJBs

Enterprise Java Beans (EJBs) are Java's groundbreaking middleware (between the client and the database) components. EJBs are a Java EE specification concerned with representing the business logic of an Enterprise Application. Enterprise Java Beans run in the Enterprise Bean Container. Once hailed as the golden child of server-side development in the corporate world, The Enterprise Java Bean API provides stock functionality for a number of tasks, simplifying web development. EJBs are trusted, reliable, scalable, proven technology for sever-side development. The EJB is one of Java's most significant technologies.

> ***How Enterprise Beans are accessed:***

Remote Method Invocation (RMI) over the **Internet Inter-Orb Protocol** is the standard, TCP based way for clients to communicate with an EJB in *a different Java Virtual Machine.*

Enterprise Java Beans can be accessed over **HTTP** if said EJBs are exposed as **RESTful** Web Services. Needless to say, **EJBs** can be accessed via local method calls as well.

Last but not least, a special kind of Enterprise Java Bean called a **Message Driven Bean**, can be accessed by making a call to the **Java Message Service**.

Persistence: JDBC (SQL based), JPA/JPQL (ORM)

The **Java Database Connectivity API (JDBC)** is a Java SE API for interacting with a database. It is primarily concerned with relational databases, however an Open Database Connectivity API bridge exists to allow interacting with an ODBC compliant data source. In practice, the most common use of the JDBC is to query or create relational databases via SQL queries processed via the Statement.executeUpdate() method. The JDBC is the *de facto* way to use *SQL* to deal with relational databases in Java programming.

The **Java Persistence API (JPA)** is a Java EE API specification that provides the means to persist Java objects in a relational database. You might ask: “Why do I need the JPA when the JDBC is available?”. The Answer: The JPA simplifies the common process of storing application data in a relational database. Instead of the ad-hoc method of creating SQL query statements to map a Java object’s properties to a relational database and restructuring said persisted objects back to Java objects, the JPA eases the pain of these common tasks. In addition, the JPA API includes it’s own query language called the **Java Persistence Query Language (JPQL)**. The JPA API also includes APIs such as the EntityManager API for dealing with the transition of objects between the Application & Persistence layers. The JPA API provides support for *Object Relational Mapping*.

CDI: Contexts & Dependency Injection (CDI) API

The **Contexts & Dependency Injection (CDI) API** is a Java EE Specification for modularizing applications. The need for the CDI API grew from the need to decouple Java EE Application components in a type-safe manner. The motivation for the CDI API sprung from the desire to integrate bind web and enterprise bean components by way of injection, leaving the responsibility of determining which instance you need to the injection service.

Contexts & Dependency Injection is a pivotal tool in contemporary Enterprise Development across the board. You should understand how to use this well. **Every JEE project you will encounter will use it in some way.**

Structure: Packaging & Deployment,WAR, JAR,EAR, JNDI, etc.

Important to Java EE development is understanding the containers in which Java EE Components operate and how they are grouped together and structured. Several concepts are important in understanding this. Among them: **WAR, JAR, EAR, Deployment Descriptors, RAR, Application Manifest, Modules etc.** A Java EE Application is a single executable containing any of the following:

> ***Brief Descriptions:***

Web Application Archives (WARs)—"A web application [archive] is a collection of static content, markup pages, and images, together with dynamic web components such as Java servlets, JSPs, JavaServer Faces, JAX-RS resources, and Java WebSocket endpoints, with any resources that these web components need such as managed beans or tag libraries and their collective deployment information." [14] A **WAR** is a special type of file format build on top of the JAR archive.

Java ARchive (JARs)—An archive file format for grouping Java class files and application resources for deployment in a single executable unit. JAR files build on the ZIP archive format and include a Java specific Application Manifest**.** The **Application Manifest—**"a metadata file contained within a JAR.It defines extension and package-related data. It contains name-value pairs organized in sections. If a JAR file is intended to be used as an executable file, the manifest file specifies the main class of the application. The manifest file is named MANIFEST.MF. The manifest directory has to be the first entry of the compressed archive."[15]

Enterprise Application aRchive (EARs)—An archive file format for packaging Java EE modules for deployment on an Application Server. In Java EE the **Deployment Descriptors** are XML files used to specify deployment configuration information of Java EE modules by an Application Server or Enterprise Container. **w*eb.xml*** is the deployment descriptor for Web Applications while ***application.xml*** is the deployment descriptor for Enterprise Applications.

Resource Adapter Archives (RARs)—An archive file format that implements the Java EE Connector Architecture for to foreign enterprise or non-java systems.[14]

In Java EE, all of the aforementioned archives are known as **modules.**

Awareness of various JEE Components:

Lastly, you should be aware of the overall environment in which you are working in. While you may or may not need to use many of these tools (*Your Mileage May Vary)*, you should at least be aware that they exist and have (at least) a basic, superficial understanding of what they are used for. There are several JSRs and several Java EE APIs and I will not name them all.

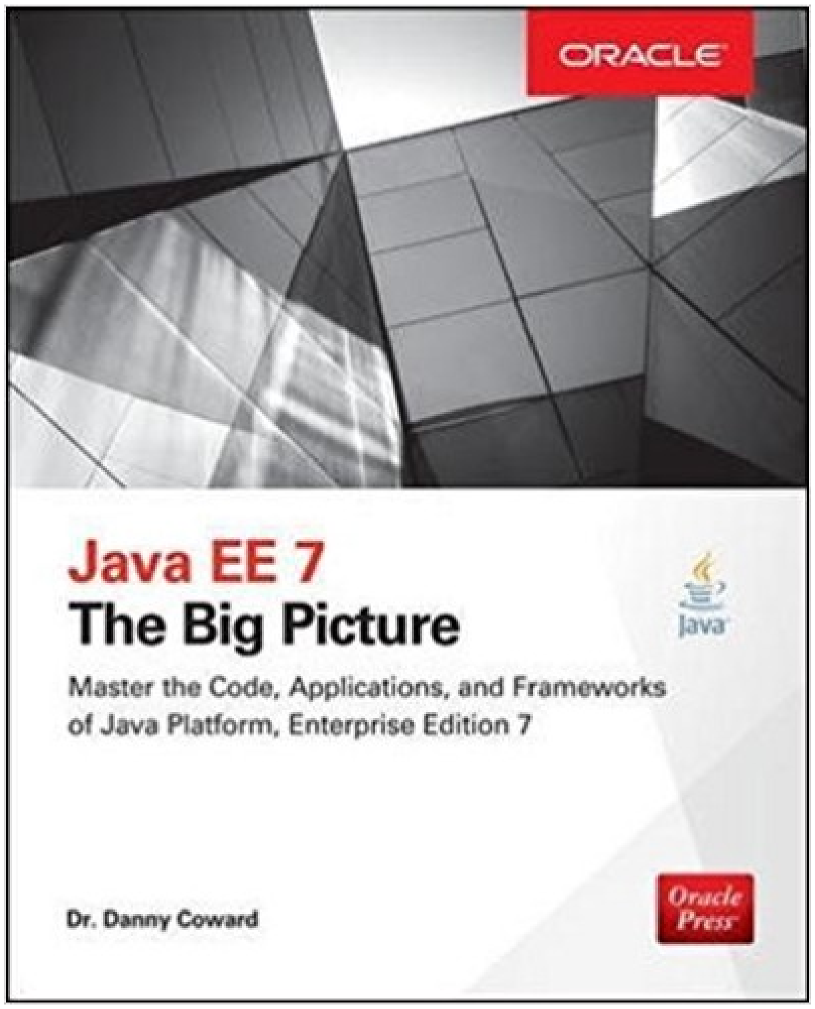
> *Some that you should be aware of:*

How **Concurrency** works in Java EE Applications.
The **Java Message Service** for loosely coupled distributed communication.
The **JAX-WS API** for XML Web Services (SOAP).
The **Java Transaction API(JTA)** for distributed transaction.
The **Java Connector Architecture** for connecting to non-Java EISs.

And so on.

Where to learn, Recommended Resources to learn Java EE:

Java EE 7: The Big Picture
Danny Coward's text is compulsory reading on Java EE. No question about it. I have learned all of the aforementioned Java EE concepts from reading and applying the content of Dr. Coward's book. I strongly recommend it for Java EE beginners and those looking to grasp the core ideas of the Java EE platform specification.


ORACLE
Java EE 7
The Big Picture
Master the Code, Applications, and Frameworks of Java Platform, Enterprise Edition 7
Dr. Danny Coward
Oracle Press

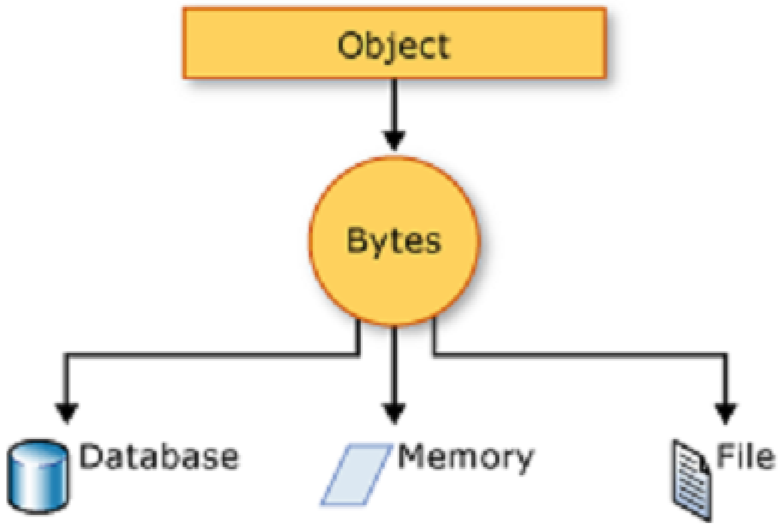
Java Platform, Enterprise Edition: The Java EE Tutorial
This is the Oracle Official Java EE Tutorial.

Java Platform, Enterprise Edition: The Java EE Tutorial Release 7
This a more expansive, Oracle Official reference document on the Java EE platform specification.

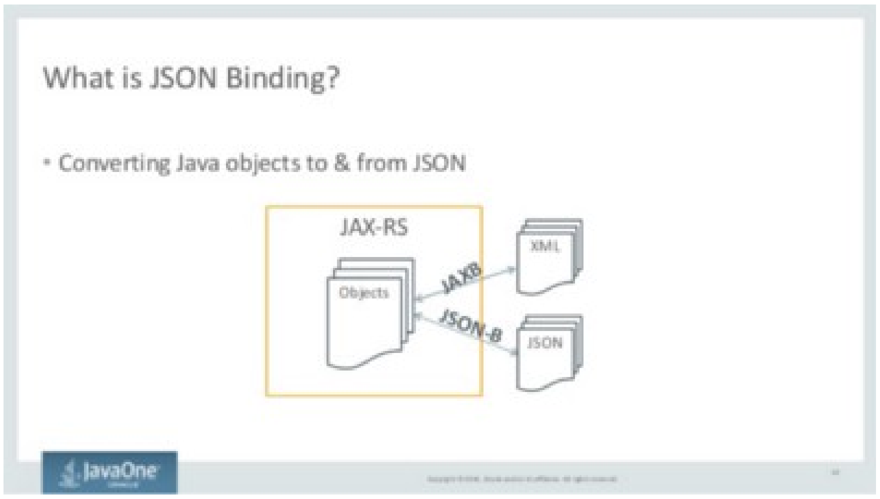
Serialization Formats—Translating Object State

> *Serialization is the process of translation data structures or objects into a format that can be stored or transmitted and then later reconstructed (possibly in a different computing environment).[17] Deserialization is the reverse of this process.*

As an Enterprise Developer you should understand how to perform serialization and deserialization. However, I will leave the topic of how to perform Serialization to another Author and instead briefly touch on the formats used in the process. Commonly, the formats that application objects will be translated to are: **Binary, XML, & JSON.** The two that we will briefly discuss here are XML & JSON. There are several advantages to serialization, of which three may be considered principal: communication, portability, and storage. In the real world, web service resources are commonly transmitted as JSON or XML. Consequently, and for several other reasons, serialization is a critical concept to understand in enterprise development. **Commonly, RESTful web resources are transmitted as JSON, and SOAP oriented web resources are transmitted as XML.**



Binary Serialization[16]



What is JSON Binding?

- Converting Java objects to & from JSON

Java to JSON, XML Slide—JavaOne Talk [18]

Extensible Markup Language (**XML**) “is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable through use of tags that can be created and defined by users.” [19]

One of the key features of XML that make it useful for serialization is the ability to define document elements. The ***XML Information Set*** is used with the SOAP protocol.

JavaScript Object Notation (JSON) “is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format used for asynchronous browser/server communication, including as a replacement for XML in some AJAX-style systems.

JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parseJSON-format data.”[20]

Where to learn, Recommended Resources:

To learn about XML:
https://www.tutorialspoint.com/xml/

To learn about JSON:
http://www.tutorialspoint.com/json/

Java EE Persistence Implementations—Tucking Objects Away For Later Use

> *“Persistence refers to object and process characteristics that continue to exist even after the process that created it ceases or the machine it is running on is powered off.” [22]*

As an Enterprise Developer, strong understanding of Persistence is key. While the JPA provides a very solid core set of features for persistence, contemporary application development has demanded features that exceed the JPA. For this reason, there are several JPA implementations that add to the JPA. Among JPA implementations there are many options: EclipseLink(Reference Implementation), Hibernate, Toplink, Apache OpenJPA, DataNucleus, & ObjectDB.

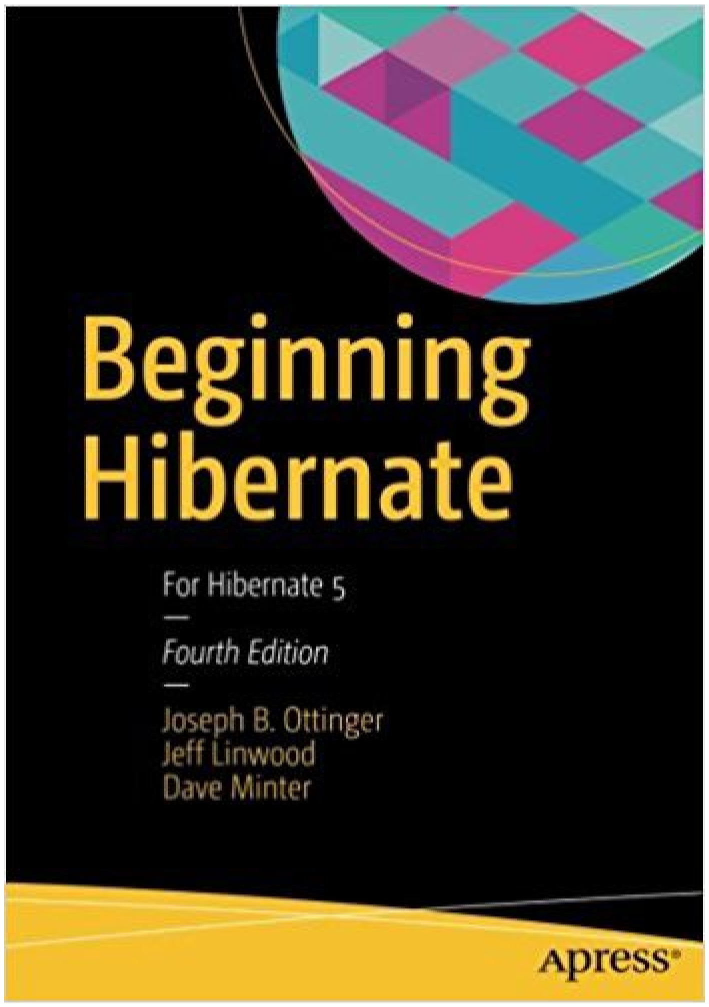
Returning to the topic of persistence, we will touch on the most popular API for persisting application objects—Red Hat's **Hibernate ORM**.

Hibernate ORM—an Object Relational Mapping tool (ORM) for persisting Java Application Objects. So popular is Hibernate that a few polls on the net suggest that it has 70% +/- market share among Java developers as an ORM solution. In 2014, Rebel Labs estimated this market share to be 67.5% [23].

Where to learn, Recommended Resources:

Hibernate 5.2 Official Tutorial & Documentation

Beginning Hibernate: For Hibernate 5, 4th ed. Edition



Spring & Hibernate for Beginners, Chad Darby—Updated 08/2017
The most popular course on uDemy on the topic of Spring/Hibernate, Chad's course is relied upon by thousands. I spent $10 on this course, a small investment that should pay back many times more.

Protocols—How we talk to each other on the net

> *A protocol is a set of rules and guidelines for communicating data. [24]*

HyperText Transfer Protocol (HTTP) is the foundational communication protocol for the World Wide Web (WWW). It is a request-response model between a client and a sever implemented atop a transport protocol, such as TCP/IP.

Basic knowledge of how HTTP works may be considered bar none in importance to web development. The web speaks HTTP. In order to understand the web, you must understand HTTP.

In Java, we deal with HTTP primarily with the **java.net.*** and **javax.servlet.*** packages.

HyperText Transfer Protocol Secure (HTTPS) is an HTTP communications protocol encrypted with ***Transport Layer Security*** (TLS) or ***Secure Socket Layer*** (SSL).

In Java, we deal with HTTPS primarily with the package **java.net.ssl.*** and **Java EE platform**.

Transmission Control Protocol/Information Protocol (TCP/IP) is the foundation of the Internet, commonly referred to as the **Internet Protocol Suite.** While it was named after the two primary protocols, it is more than just TCP and IP.

"TCP/IP is a [four-layered] suite of protocols designed to establish a network of networks to provide a host with access to the Internet. TCP/IP is responsible for full-fledged data connectivity and transmitting the data end-to-end.."[25]

In Java, we interact with TCP/IP in none other than the **java.net.*** packages.

You can learn more, here. TCP is the foundation of the **WebSocket protocol**. The truth is, however, TCP/IP is the engine by which we most often conduct HTTP communication—TCP/IP is everywhere, and is by far the most dominant protocol of internet communication. It is important to understand, that HTTP functions on a layer underneath TCP/IP.

TCP/IP are in the transport and internet layers, while HTTP is much higher, at the Application layer. The **four layers,** in hierarchical order are: the link layer, the internet layer, the transport layer, and the application layer. Consequently, **we most commonly use HTTP atop TCP/IP.**

User Datagram Protocol (UDP) is a unreliable, stateless transport layer protocol that communicates by way of messages called datagrams. Datagrams are not guaranteed to be delivered and there are no error checking mechanisms in place. It finds common application in domains such as video conferencing.

In Java, we interact with UDP via the **java.net.*** packages, particularly **java.net.DatagramSocket**.

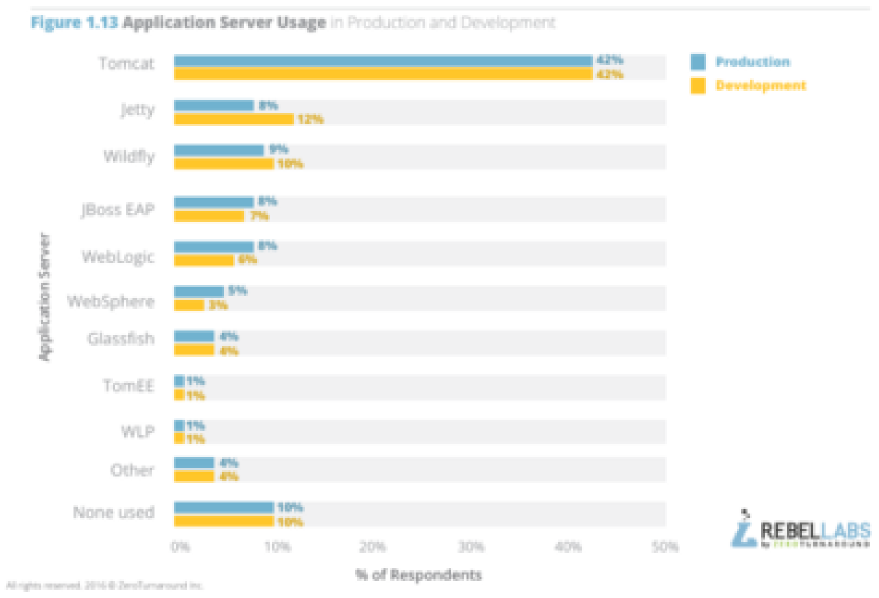
Application Server—It's all talk until it's deployed

> *An Application Server provides the the tools to create Web Applications as well as a sever to run them [26].*

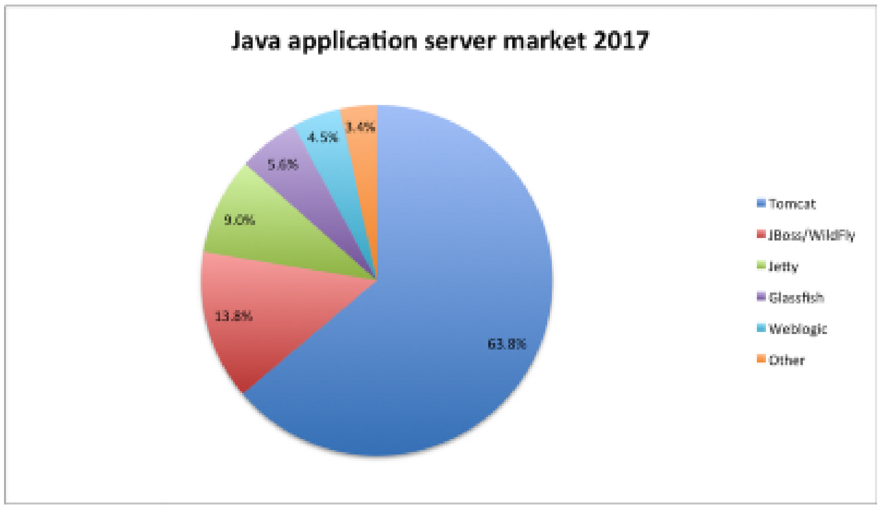
Remember how we said that Java EE was merely a specification? Java EE Application Servers provide the implementation for Java EE specifications as well as a whole host of resources needed to build and deploy web applications. The reference implementation of the Java EE Specification is Oracle's **GlassFish Application Server.** Additionally, it is important to understand that not all Java EE Application Servers implement the full breadth of the platform. Apache Tomcat (arguably the most popular one), for example, is mostly a Servlet Container implementation. Also, key in understanding Java EE is understanding that each implementation of the Java EE Specification may add to or build on the API specification, and this is commonly the case. One such example is found in Jersey, the JAX-RS reference implementation that we touched on earlier. While Jersey provides a reference implementation for the JAX-RS API specification, it additionally provides APIs for other things, such as GUICE integration.

The take away from this is that the situation as it pertains to portability is a little more subtle as compared to Java SE development. **Portable code is accomplished by sticking to the Java EE API libraries.** The various implementations of the Java EE API specification almost always include more than just the Java EE APIs. **This is true even for various Reference Implementations. Consequently, in practice this leads to code that is often not portable across Java EE Application Servers.** Unless you examine the contents of a Enterprise or Web archive, you cannot be sure that it will run on your application server. This is further complicated by the fact that there are partial as well as full Java EE Application Servers.

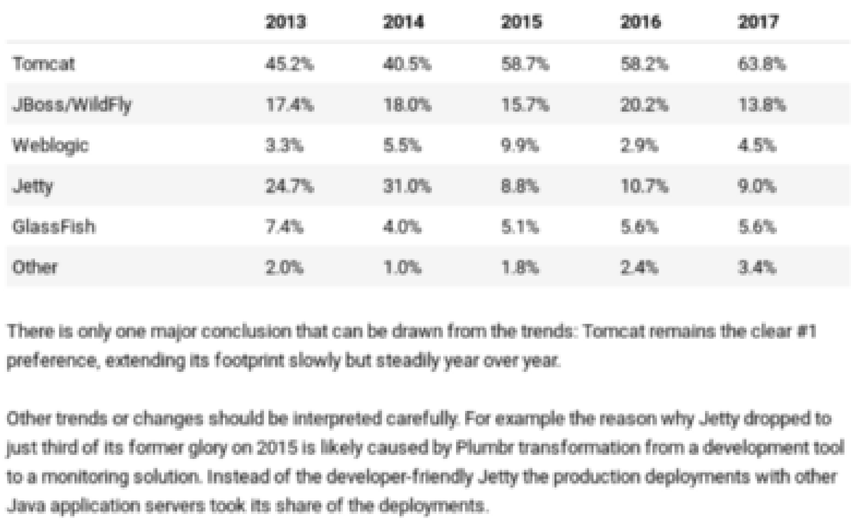
According to Rebel Labs [27], **Apache Tomcat** is a leading Application Server choice with 42% market share followed by a distant **Eclipse Jetty** and **RedHat's Wildfily** (formerly popularly known as JBoss).



According to ***Plumbr*** [28], an application performance monitoring company, **Apache Tomcat** has a 63.8% market share, followed by 13.8% for **Redhat's WildFly** and 9.0% for **Eclipse's Jetty.**



The following data is again from ***Plumbr***, showing trends over time.



	2013	2014	2015	2016	2017
Tomcat	45.2%	40.5%	58.7%	58.2%	63.8%
JBoss/WildFly	17.4%	18.0%	15.7%	20.2%	13.8%
Weblogic	3.3%	5.5%	9.9%	2.9%	4.5%
Jetty	24.7%	31.0%	8.8%	10.7%	9.0%
GlassFish	7.4%	4.0%	5.1%	5.6%	5.6%
Other	2.0%	1.0%	1.8%	2.4%	3.4%

There is only one major conclusion that can be drawn from the trends: Tomcat remains the clear #1 preference, extending its footprint slowly but steadily year over year.

Other trends or changes should be interpreted carefully. For example the reason why Jetty dropped to just third of its former glory on 2015 is likely caused by Plumbr transformation from a development tool to a monitoring solution. Instead of the developer-friendly Jetty the production deployments with other Java application servers took its share of the deployments.

It is a safe bet to say that a contemporary Java Enterprise Developer should understand the way **Apache Tomcat** works. Minimally, a Java Enterprise Developer should be familiar with the environment of at least one Application Server.

Where to learn, Recommended Resources:

Apache Tomcat 9 Official Documentation

Redhat's WildFly 10 Official Documentation

This publication is the second part of a 3-part series. Should you prefer to read the full article it is available here.

Source: ChocolateVanilla.com

The next article in this series, "How To Become a Java Web or Enterprise Developer—Part III: Frameworks, Real World Tools, Better Code & Beyond", is available here.

The works cited for the publication is available in the last article in the series.