www.mas-ict.ch

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

Java EE & Web Services

Distributed Architecture
Service-Oriented Architecture

Omar ABOU KHALED

Stefano CARRINO

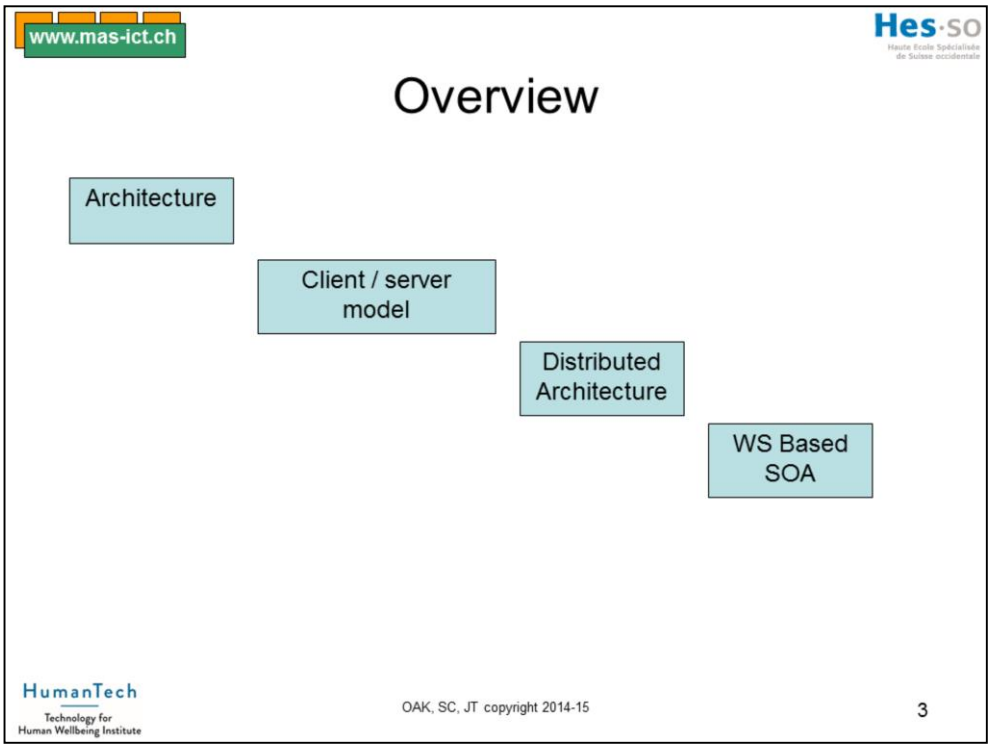
Julien Tscherrig

University of Applied Sciences of Western Switzerland
EIA-FR, Bd de Pérolles 80 - CP 32, CH-1705 Fribourg
omar.aboukhaled@hefr.ch |Tél: +41 26 429 65 89
stefano.carrino@hefr.ch| Tél: +41 26 429 67 48
http://humantech.eia-fr.ch | Fax: +41 26 429 66 00

HumanTech
Technology for Human Wellbeing Institute

Plan

- Day 1
  - Introduction to architecture evolution
    - 2 tiers, 3 tiers, distributed, etc.
  - Introduction to Java EE
    - Servlet and JSP
- Day 2 & 3
  - Java EE: EJB, JPA and JSF
  - Web Services: UDDI, WSDL, SOAP
  - An Overview of Java Web Services
  - Service-Oriented Architecture (SOA)
  - RESTful web services

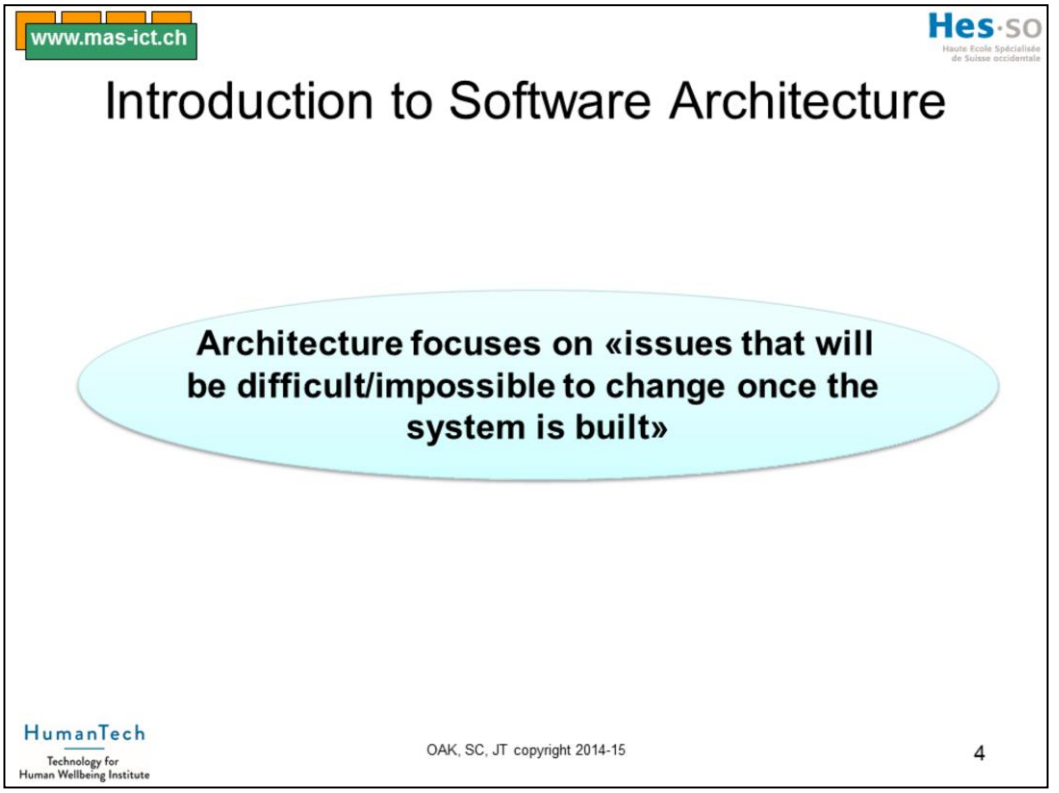


Motivation

To understand the problems that Web services try to solve it is helpful to understand how distributed information systems evolved. While technology has changed, problems stayed the same.

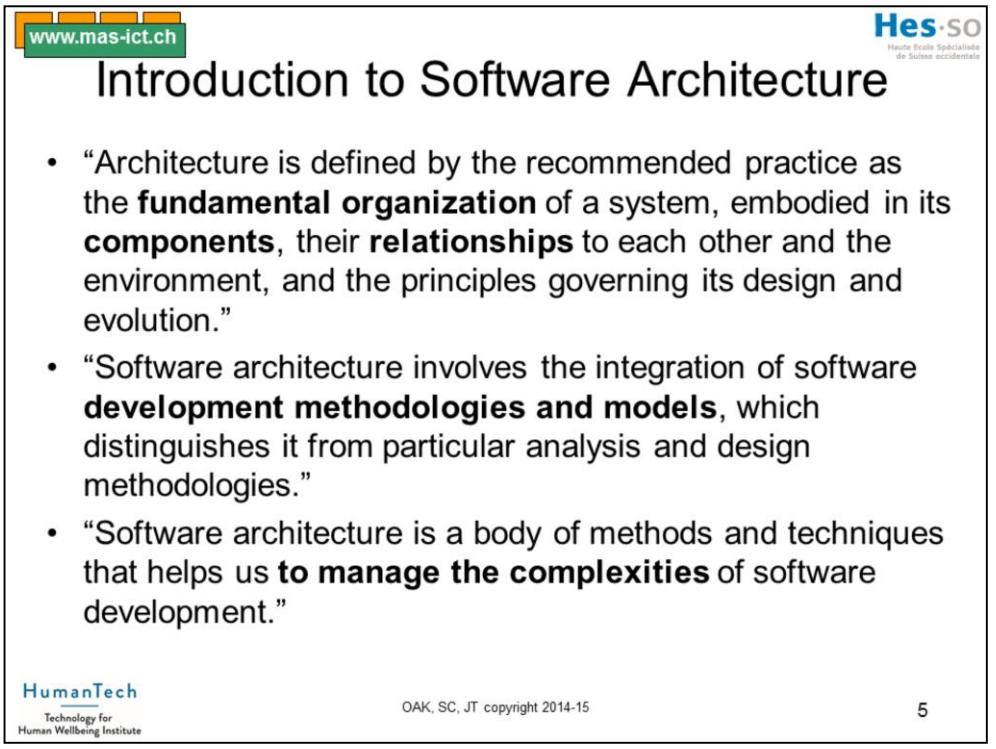
The architect assumes success and structures the system accordingly. Typical architectural questions are:

- How do I partition my system?
- Should I support diverse user bases?
- How do I enable the system to integrate with third-party applications?
- What standards should the design and development teams adhere to?
- What tools best meet these needs?



Garlan and Shaw:

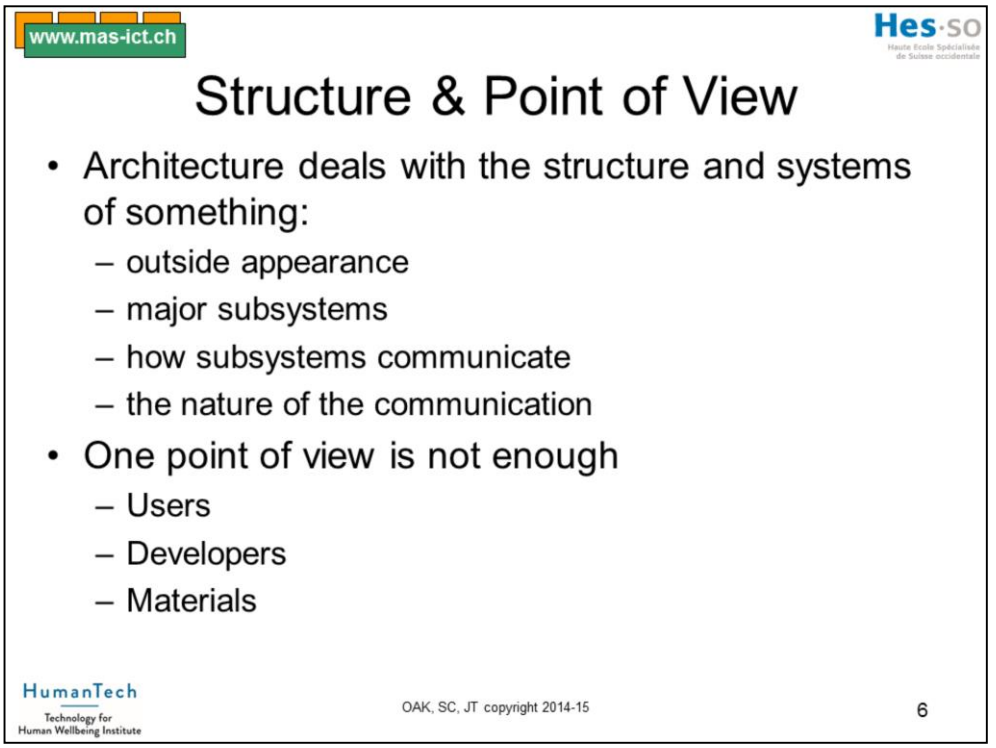
Structural issues include gross organization and global control structure;
-Protocols for communication, synchronization and data access;
-Assignment of functionalities to design elements
-Physical distribution
-Composition of design elements;
-Scaling and performance;
-Selection among design alternatives.



Sources: [ANSI/IEEE Std 1471-2000*, Recommended Practice for Architectural Description of Software-Intensive Systems]*

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally

visible properties of those elements, and the relationships among them." [L.Bass, P.Clements, R.Kazman, Software Architecture in Practice (2nd edition), Addison-Wesley 2003]



Architecture Defines Structure

Much of an architect's time is concerned with how to sensibly partition an application into a set of inter-related components, modules, objects or whatever unit of software partitioning works for you.

Architecture Specifies Component Communication

When an application is divided into a set of components, it becomes necessary to think about how these components communicate data and control information. The components in an application may exist in the same address space, and communicate via straightforward method calls. They may execute in different threads or processes, and communicate through synchronization mechanisms. Or multiple components may need to be simultaneously informed when an event occurs in the application's environment.

Architecture Addresses Non-functional Requirements

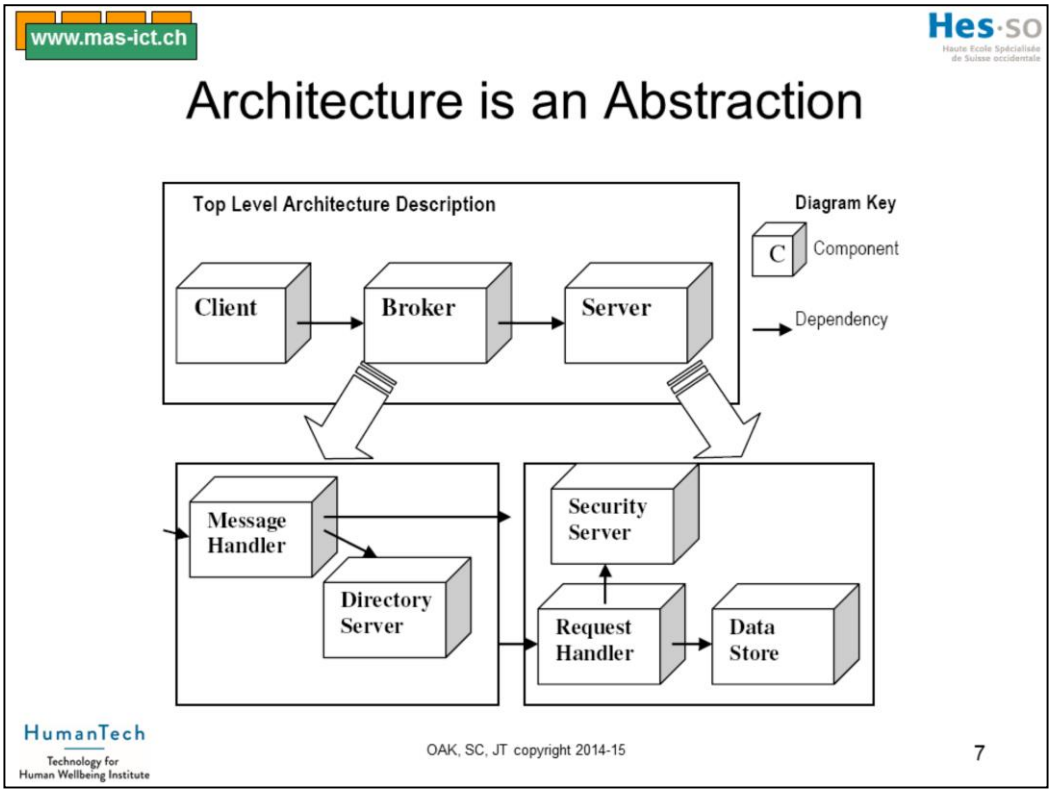
Non-functional requirements are the ones that don't appear in use cases. Rather than define *what the application does, they are concerned with how* the application provides the required functionality.

There are three distinct areas of non-functional requirements:

• **Technical constraints:** These will be familiar to everyone. They constrain design options by specifying certain technologies that the application must use. "We only have Java developers, so we must develop in Java." "The existing database runs on Windows XP only." These are usually non-negotiable.

• **Business constraints:** These too constraint design options, but for business, not technical reasons. For example, "In order to widen our potential customer base, we must interface with XYZ tools." Another example is "The supplier of our middleware has raised prices prohibitively, so we're moving to an open source version." Most of the time, these too are non-negotiable.

• **Quality attributes** These define an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor.



Architecture is an Abstraction

One of the most useful, but often non-existent, descriptions from an architectural perspective is something that is colloquially known as a *marketecture.*

This is one page, typically informal depiction of the system's structure and interactions. It shows the major components, their relationships and has a few well chosen labels and text boxes that portray the design philosophies embodied in the architecture. A *marketecture is an excellent* vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process. It's easy to understand and explain, and serves as a starting point for deeper analysis.

One of the most powerful mechanisms for describing an architecture is hierarchical decomposition. Components that appear in one level of description are decomposed in more detail in accompanying design documentation. As an example, Fig. depicts a very simple two level hierarchy using an informal notation, with two of the components in the top-level diagram decomposed further. Different levels of description in the hierarchy tend to be of interest to different developers in a project. In Fig. it's likely that the three components in the top level description will be designed and built by different teams working on the application. The architecture clearly partitions the responsibilities of each team, defining the dependencies between them.

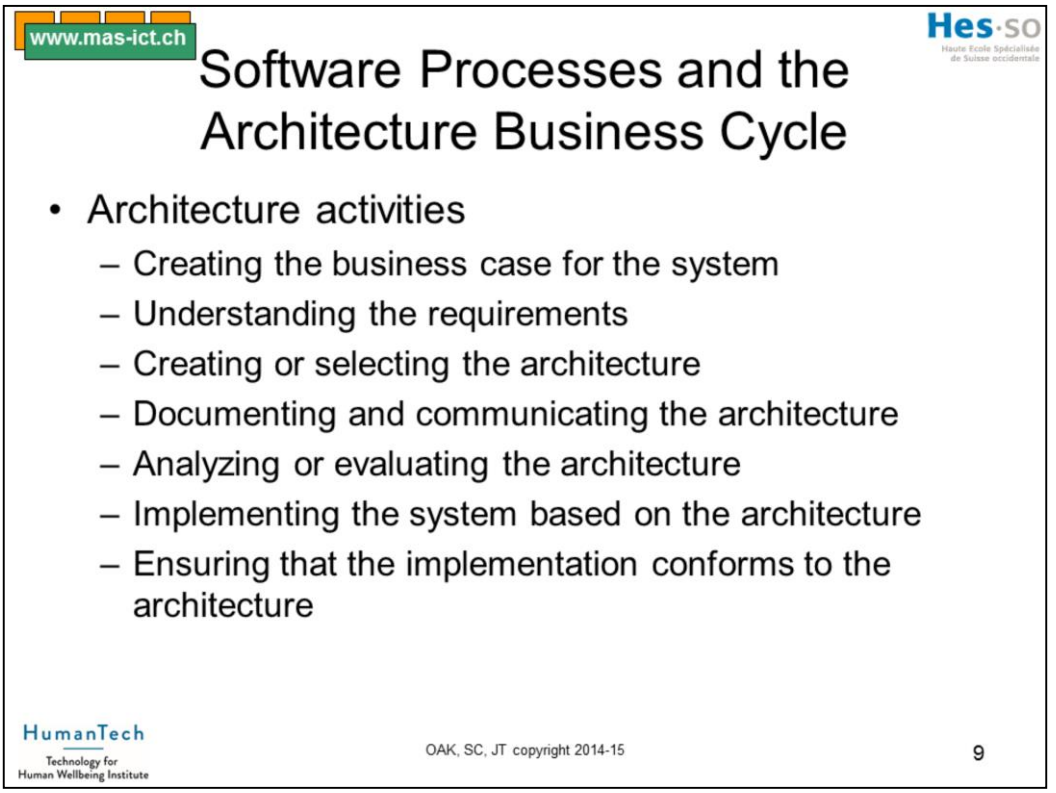
Source: http://www.sei.cmu.edu/activities/architecture/arch_duties.html

•**Liaison:** Architects play many liaison roles. They liaise between the customers or clients of the application and the technical team, often in conjunction with the business and requirements analysts. They liaise between the various engineering teams on a project, as the architecture is central to each of these. They liaise with management, justifying designs, decisions and costs. They liaise with the sales force, to help promote a system to potential purchasers or investors. Much of the time, this liaison takes the form of simply translating and explaining different terminology between different stakeholders.

• **Software Engineering:** Excellent design skills are what get a software engineer to the position of architect. They are an essential pre-requisite for the role. More broadly though, architects must promote good software engineering practices. Their designs must be adequately documented and communicated and their plans must be explicit and justified. They must understand the downstream impact of their decisions, working appropriately with the application testing, documentation and release teams.

•**Technology Knowledge:** Architects have a deep understanding of the technology domains that are relevant to the types of applications they work on. They are influential in evaluating and choosing third party components and technologies. They track technology developments, and understand how new standards, features and products might be usefully exploited in their projects. Just as importantly, good architects know what they don't know.

•**Risk Management** Good architects tend to be cautious. They are constantly enumerating and evaluating the risks associated with the design and technology choices they make. They document and manage these risks in conjunction with project sponsors and management. They develop and instigate risk mitigation strategies, and communicate these to the relevant engineering teams. They try to make sure no unexpected disasters occur.



Software process is the term given to the organization, and management of software development activities.

Creating the Business Case for the System

Creating a business case is broader than simply assessing the market need for a system. It is an important step in creating and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market? Will it need to interface with other systems? Are there system limitations that it must work within?

Understanding the Requirements

There are a variety of techniques for eliciting requirements. For example, object-oriented analysis uses scenarios, or "use cases" to embody requirements. Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly catalyze decisions on the system's design and the design of its user interface.

Analyzing or Evaluating the Architecture

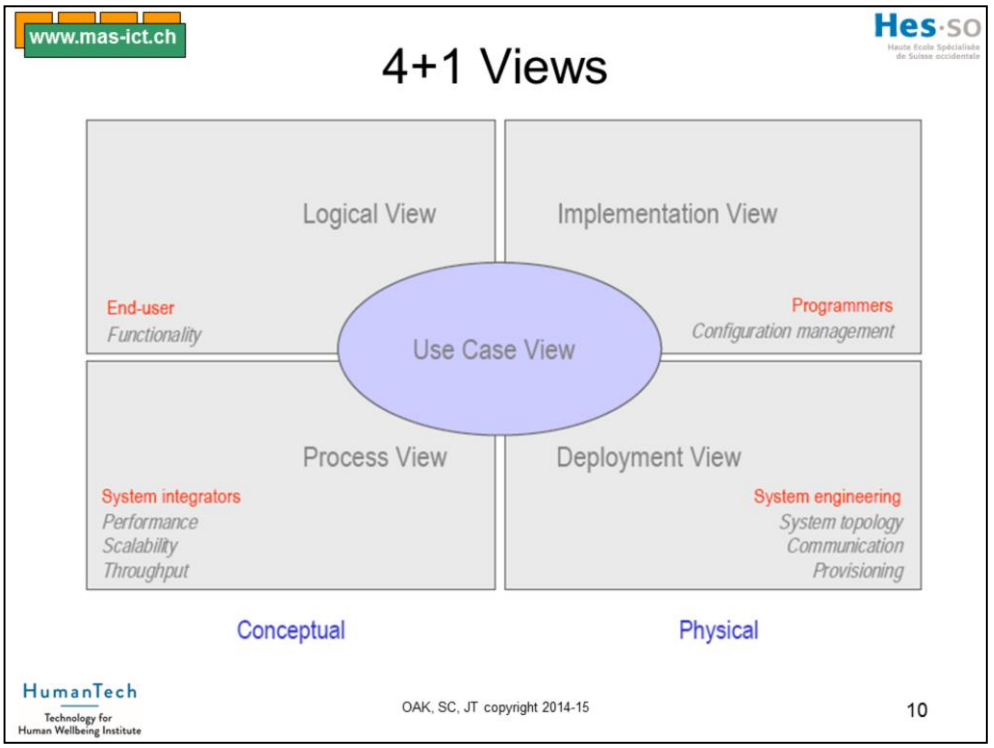
Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture.

Implementing Based on the Architecture

This activity is concerned with keeping the developers faithful to the structures and interaction protocols constrained by the architecture. Having an explicit and well-communicated architecture is the first step toward ensuring architectural conformance. Having an environment or infrastructure that actively assists developers in creating and maintaining the is better.

Ensuring Conformance to an Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Constant vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase.



Architecture Views

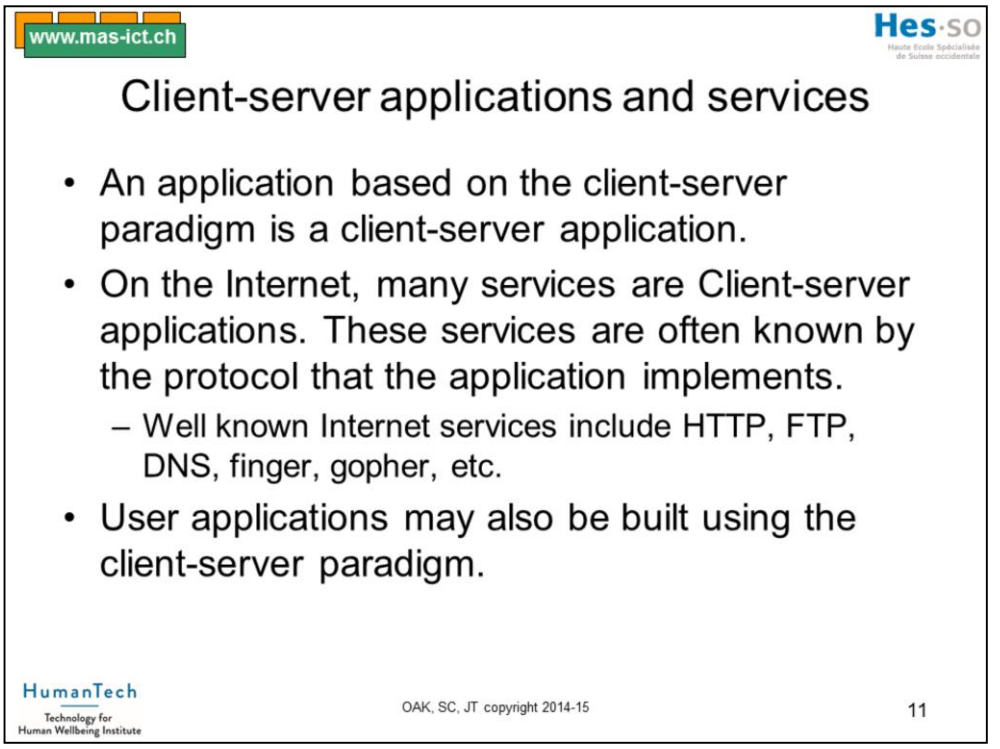
To completely describe a software architecture, four views are needed:

Logical view: This describes the architecturally significant elements of the architecture and the relationships between them. The logical view essentially captures the structure of the application using class diagrams or equivalents.

Process view: This focuses on describing the concurrency and communications elements of an architecture. In IT applications, the main concerns are describing multi-threaded or replicated components, and the synchronous or asynchronous communication mechanisms used.

Physical (Deployment) view: This depicts how the major processes and components are mapped on to the applications hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines.

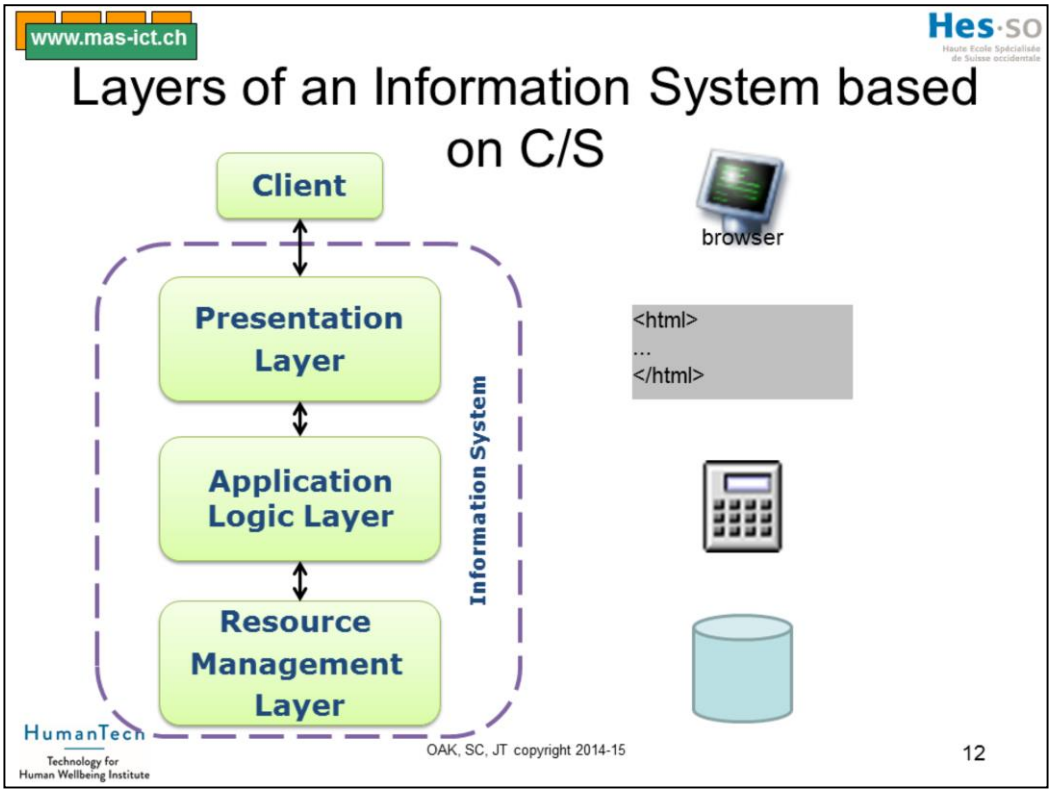
Development (implementation) view: This captures the internal organization of the software components, typically as they are held in a development environment or configuration management tool. For example, the depiction of a nested package and class hierarchy for a Java application would represent the development view of an architecture.



Client / Server model

Client/server refers to either or both of hardware and software

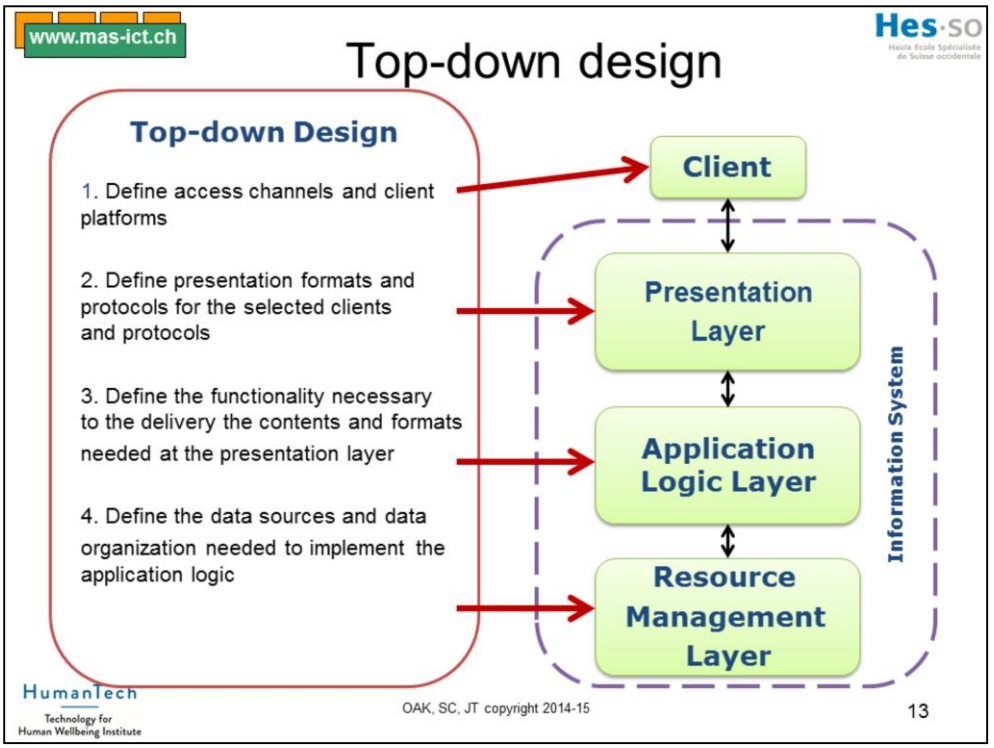
In this case, one thing is a client which makes requests to another thing which is a server. Requests go from client to server. Responses go from server to client



Presentation Layer: here is decided HOW data should appear to the user

Application Logic Layer: Data Processing ('The actual Program'), here the algorithms are implemented. This Layer is often referred as services (business logic, business rules, server)

Resource Management Layer: deals with and implements different data sources of IS. It is the 'data layer' in a restricted interpretation (Database Management System). Can also be an external system, which recursively uses other ISs.



top-down design

-usually created to run in homogenous environments

-way of distribution has to be specified

-results in tightly coupled components:

-functionality of each component heavily depends on functionality of other components

-design is component based, but components are not standalone

Advantages & disadvantages

advantages:

design emphasises final goals of the system

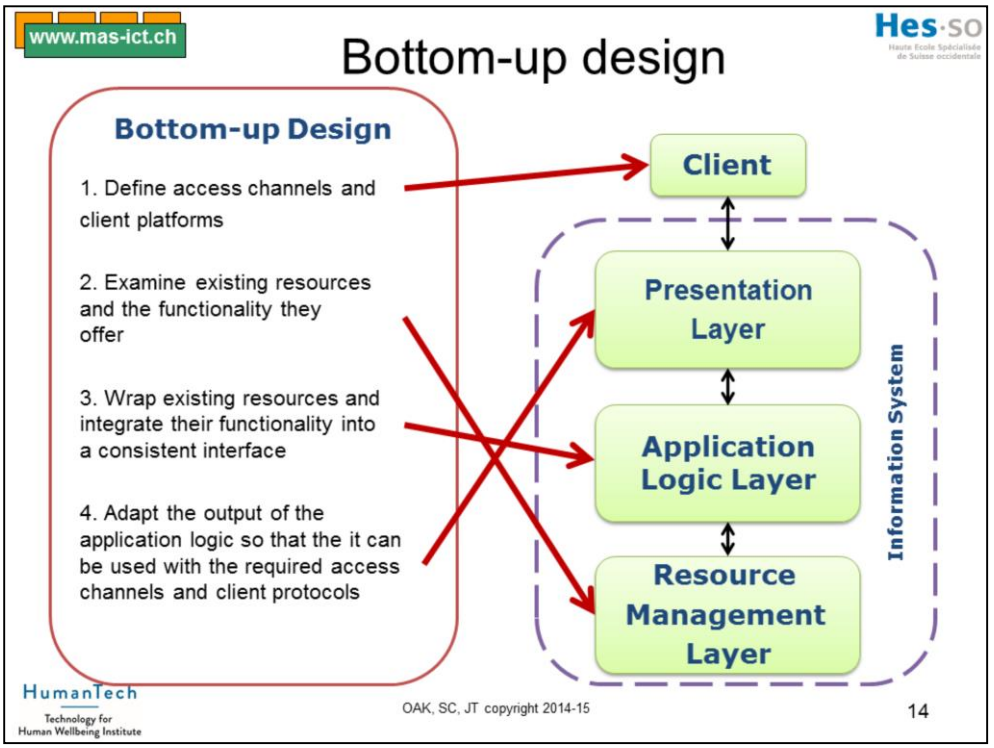
can be optimized for: functional and non-functional(performance, availability,..) issues

disadvantages

can only be designed from scratch

legacy systems cannot be integrated

today few ISs are designed purely top-down



bottom-up design

-out of necessity rather than choice

-need to integrate legacy systems and/or applications

-results in loosely coupled systems

independent and

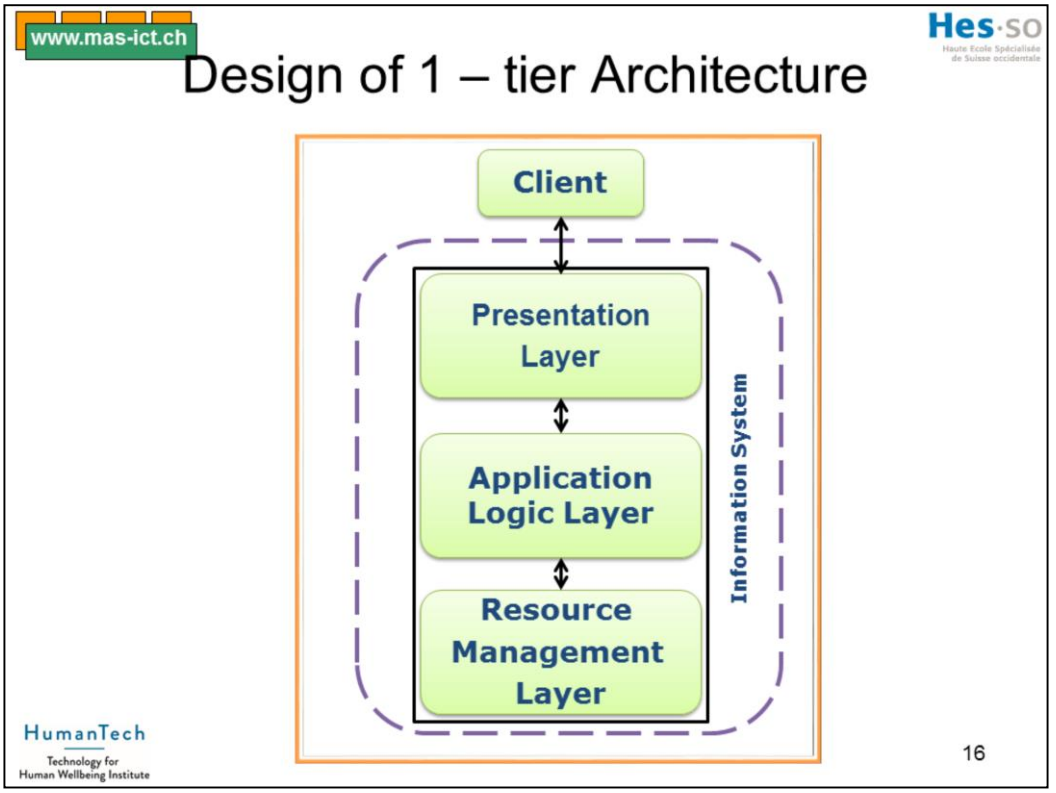
standalone components

-most distributed IS are result of a bottom-up design

Web services can make those designs more efficient, cost-effective and simpler to design

Architecture of an Information System - 4 types:

- 1 – tier
- 2 – tier
- 3 – tier
- n – tier



1 – tier Architectures were used decades ago.. monolithic Information Systems. Presentation, application logic, and resource management were merged into a single tier many of these 'old' Systems are still in use!

advantages: easy to optimize performance. no context switching. no compatibility issues.

disadvantages: monolithic pieces of code (high maintenance). hard to modify. lack of qualified programmers for these systems. no client development. maintenance and deployment cost

Mainframe Architecture

With mainframe software architectures all intelligence is within the central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host. One of the advantages of Mainframe is that Mainframe software architectures are not tied to a hardware platform. User can interact using PCs and UNIX workstations.

•Limitation of mainframe software architectures is that:

•they do not easily support graphical user interfaces (GUI) or access to multiple databases.

•produces substantial network traffic

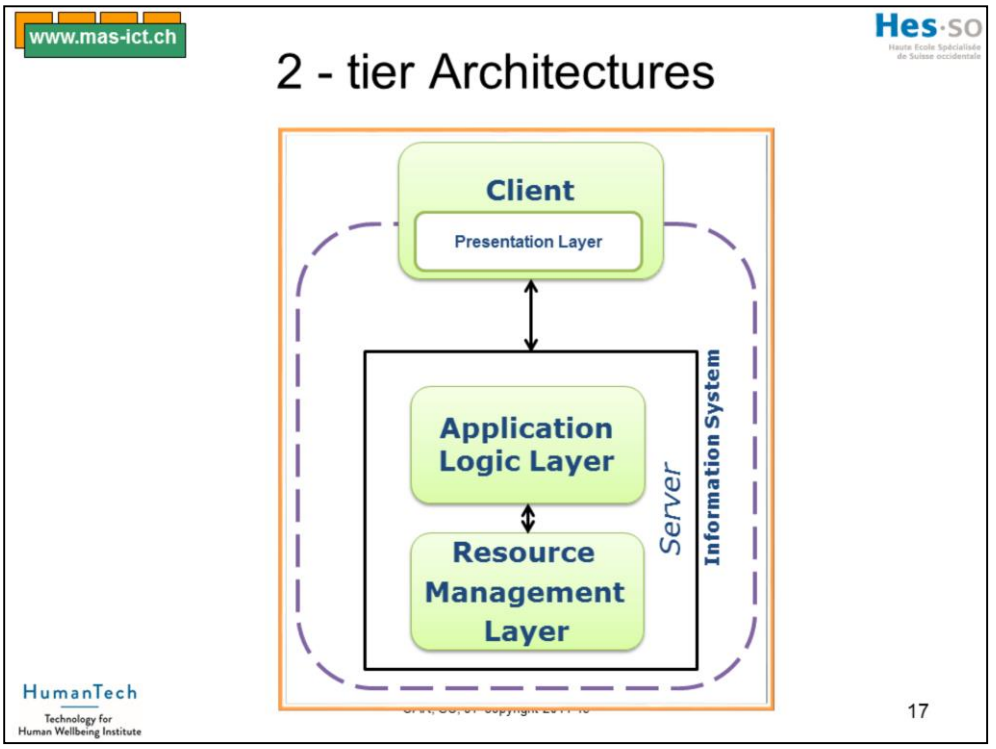
•Requires a complex operating System

•Expensive to maintain

•do not easily support access to multiple databases from geographically dispersed sites

•Difficult to scale up or adapt them to increasing workloads

However In the last few years, mainframes have found a new use as a server in client/server architectures .



2 - tier Architectures

Separation of presentation layer from other 2 layers (app + resource)

•became popular as 'server/client' systems

•thin clients/fat clients

•RPC (Remote Procedure Call)

•API (Application Program Interface)

•need for standardization

advantages & disadvantages

advantages

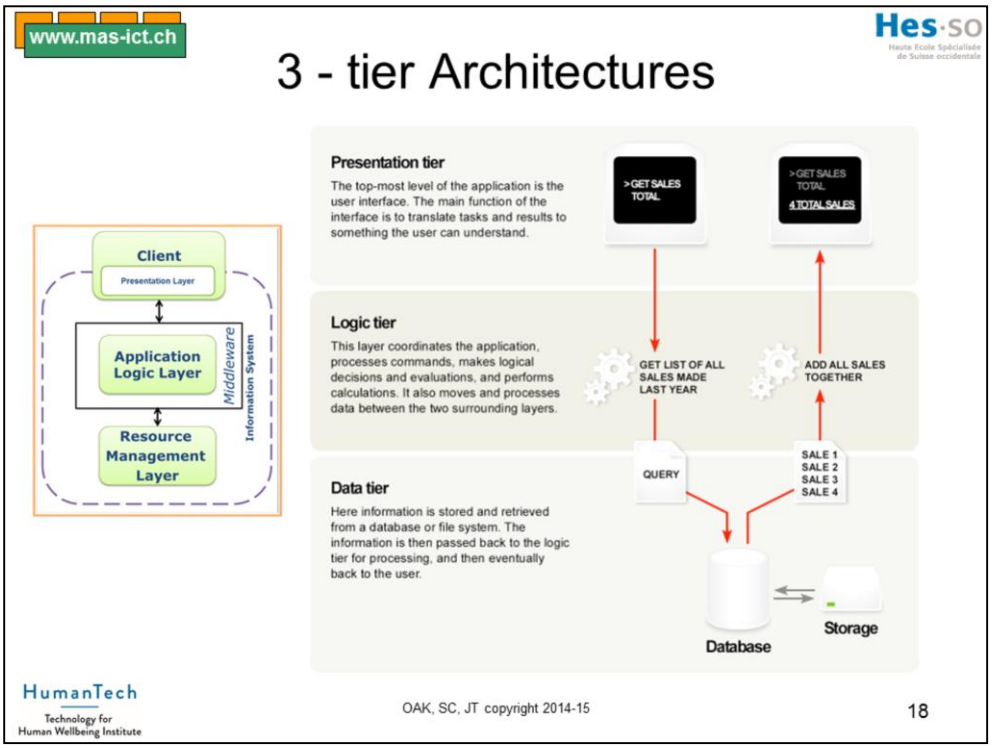
•portability

•no need for context switches or calls between component for key operations

disadvantages

•limited scalability

•legacy problems (blown up clients)



Source:http://en.wikipedia.org/wiki/Multitier_architecture

3 - tier Architectures

•can be achieved by separating RM (resource management) from application logic layer

•additional middleware layer between client and server: *integration logic & application logic*

•lead to the introduction of clear RM layer interfaces

•good at dealing with integration of different resources

advantages & disadvantages

advantages

•scalability by running each layer on a different server

•scalability by distributing AL (application logic layer) across many nodes

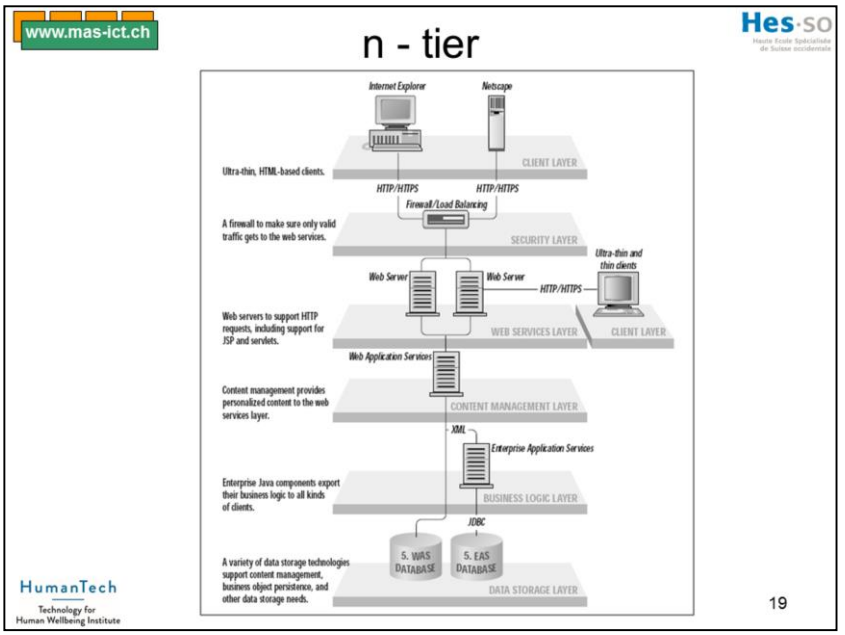
•additional tier for integration logic

•flexibility

disadvantages

•performance loss if distributed over the internet

•problem when integrating different 3 – tier systems



advantages & disadvantages

•Advantages: better scalability, higher fault tolerance, higher throughput for less cost

•Disadvantages: too much middleware involved, redundant functionality, difficulty and cost of development

gains and losses

with growing number of tiers one gains: flexibility, functionality, possibilities for distribution

but: each tier increases communication costs, complexity rises, higher complexity of management and tuning

Benefits of an n-tiered model

Separating the responsibilities of an application into multiple tiers makes it easier to scale the application. An n-tiered architecture allows you to separate the workload better for developers. By breaking design into tiers, developers with different specialties can focus on a tier that best suits their skill set. An n-tiered model also makes an application more readable and its components more reusable. By separating an application into tiers you make it much harder to fall into the trap of writing spaghetti code-which refers to huge line counts of code with a complex, tangled control structure and deeply nested if-then statements. Finally, an n-tiered approach makes your applications more robust by eliminating a single point of failure. For example, if you decide to change database vendors you won't have to hunt through every single template of your application to make the necessary changes. Simply replace the data tier and adjust the applicable portions of the integration tier to query the new database. You will not break the business logic or, more importantly, presentation tier code.

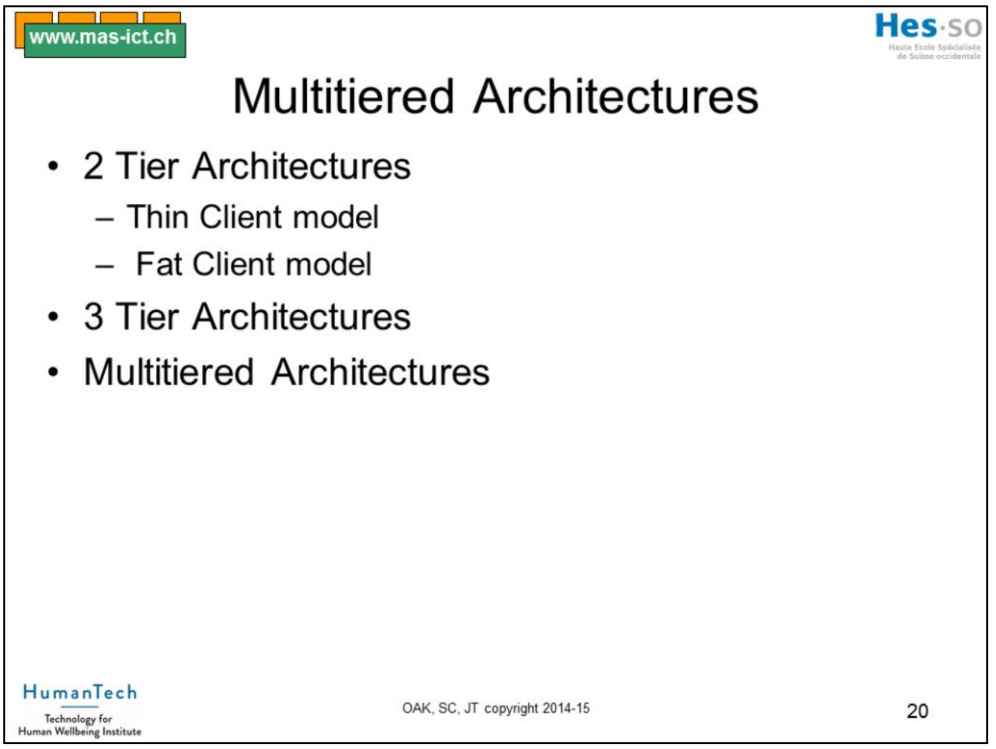
The Client/Server architecture

The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing. Clients and servers are separate logic logical entities that work together over a network to accomplish a task. They all have the following characteristics:

Service*:* Client/server is primarily a relationship between processes running on separate machines. The server process is a provider of services. The client is a consumer of services. In essence, client/server provides a clean separation of function based on the idea of service.

Shared resources*:* A server can service many clients at the same time and regulate their access to shared resources.

Asymmetrical protocols*:* There is a many-to-one relationship between clients and server. Clients always initiate the dialog by requesting a service. Servers are passively awaiting requests from the clients.



FAT SERVERS OR FAT CLIENTS?

Client/server models can be distinguished by the service they provide but Client/server applications can also be differentiated by how the distributed application is split between the client and the server.

•*The fat server model* places more function on the server.

•*The fat client model* does the reverse.

•Groupware, transaction, and Web servers are examples of fat servers

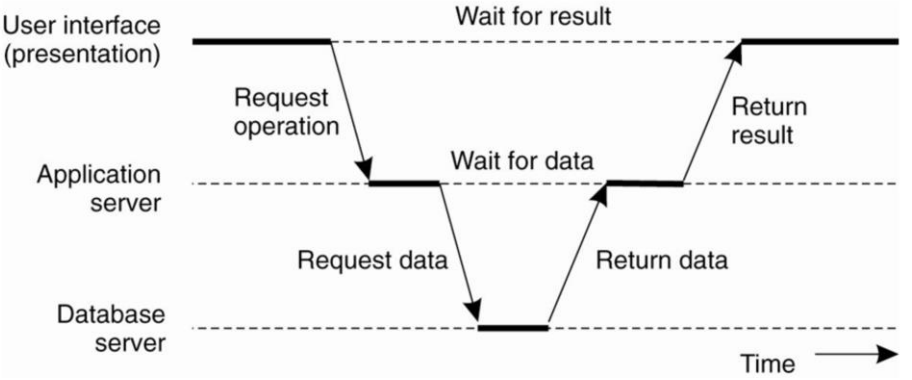
•database and file servers are examples of fat clients.

Fat clients are the more traditional form of client/server. The bulk of the application runs on the client side of the equation. In both the file server and database server models, the clients know how the data is organized and stored on the server side. fat clients are used for decision support and personal software. They provide flexibility and opportunities for creating front-end tools that let end-users create their own applications.

Fat server applications are easier to manage and deploy on the network because most of the code runs on the servers. Transaction and object servers, for example, encapsulate the database. Instead of exporting raw data, they export the procedures (or methods in object-oriented terminology) that operate on that data. The client in the fat server model provides the GUI and interacts with the server through remote procedure calls. (or method invocations).

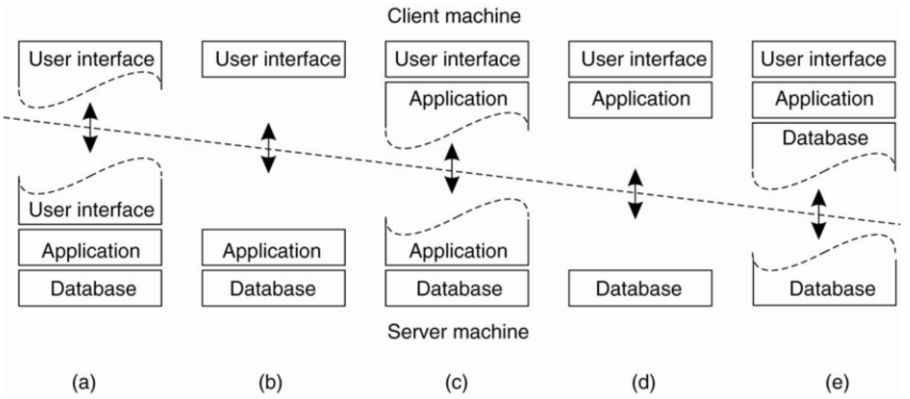
Multitiered Architectures (3 Tier Architecture)

- An example of a server acting as client.



Multitiered Architectures

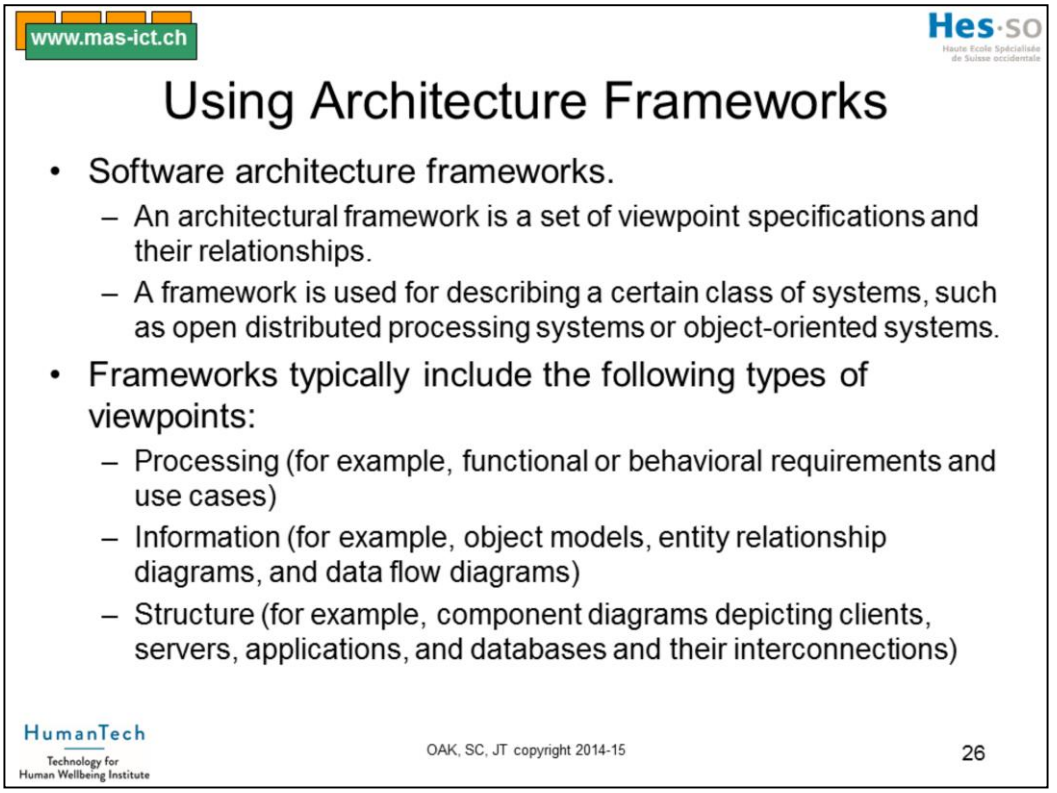
- Alternative client-server organizations (a)–(e).



Thin Client ⟷ Fat Client

When to use a two-tier design

- If you can answer "yes" to each of the questions in the checklist, then a two-tier architecture is likely your best solution. **Otherwise you might consider a three-tier design**.
  - Does your application emphasize time-to-market over architecture?
  - Does your application use a single database?
  - Is your database engine located on a single host?
  - Is your database likely to stay approximately the same size over time?
  - Is your user base likely to stay approximately the same size over time?
  - Are your requirements fixed with little or no possibility of change?
  - Do you expect minimal maintenance after you deliver the application?



Architecture Framework Goals

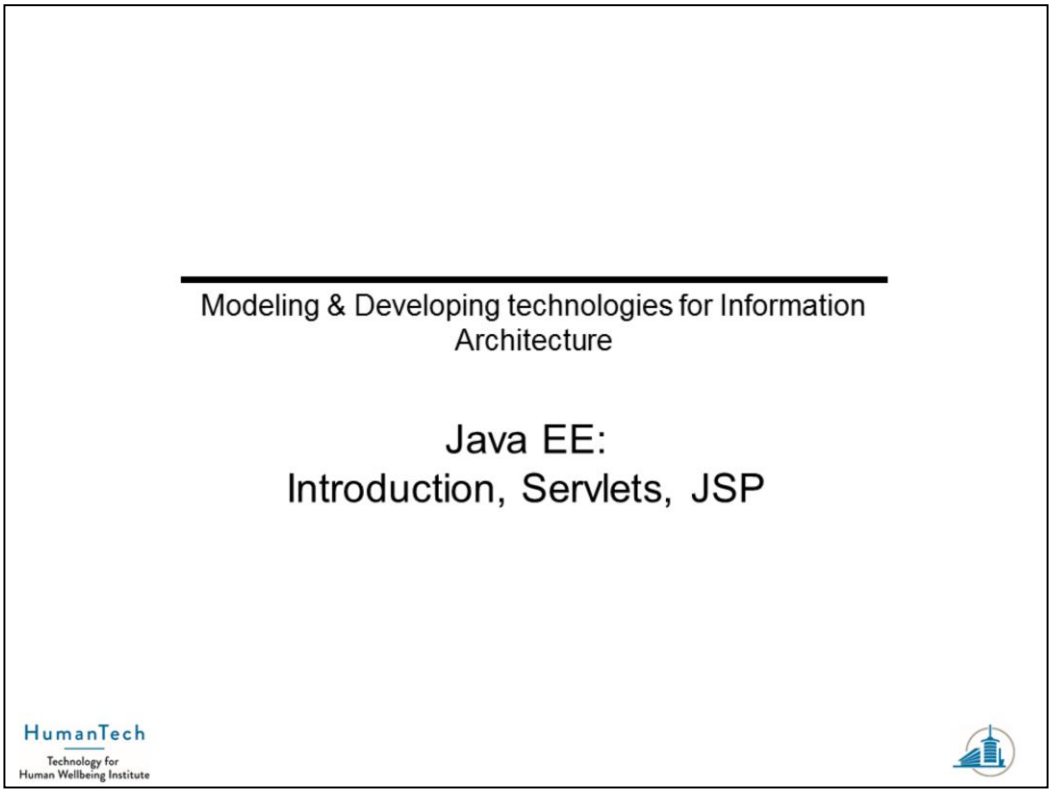
•Codify best practices for architectural description (to improve the state of the practice).

•Ensure that the framework sponsors receive architectural information in the format they want.

•Facilitate architecture assessment.

•Improve the productivity of software development teams by using standardized means for design representation.

•Improve interoperability of information systems.

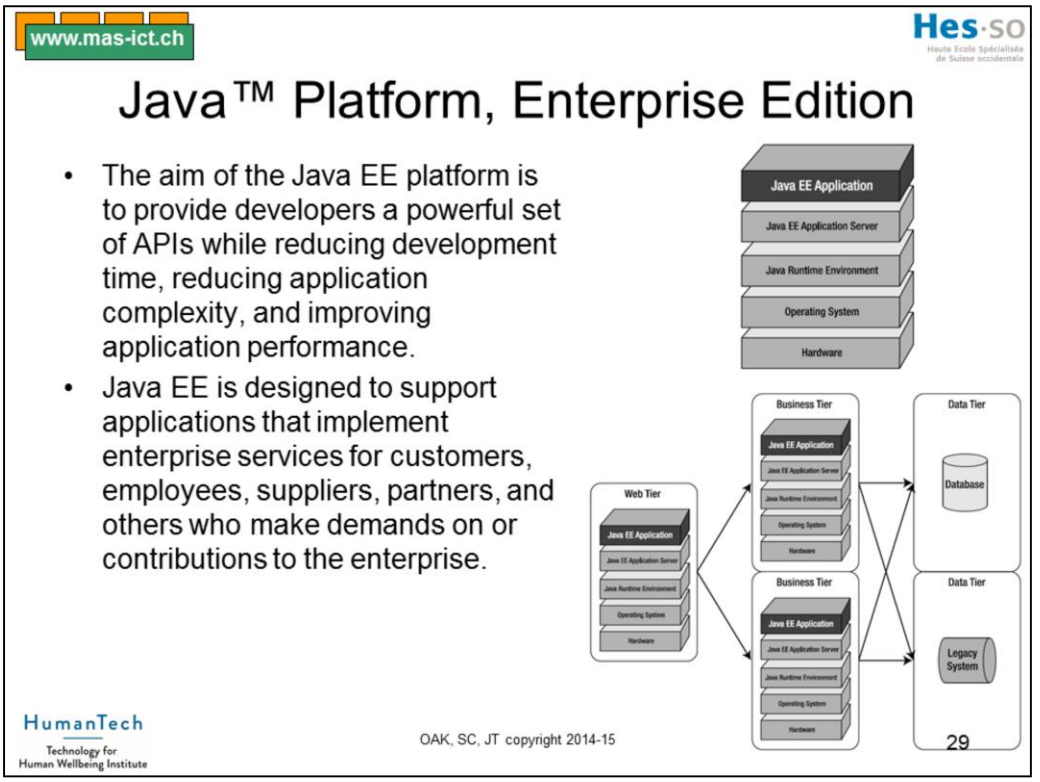


Java™ Platform, Enterprise Edition 5 (Java EE 5) Specification (http://jcp.org/en/jsr/detail?id=244)

Java Technology Levels

- Java Platform, Standard Edition
  - Java SE (J2SE)
  - core language
- Java Platform, Micro Edition
  - Java ME (J2ME)
  - targeted at small devices
    - PDAs
    - cell phones
- Java Platform, Enterprise Edition
  - Java EE
  - targeted at enterprise deployments
    - persistence
    - distributed systems
    - web-based applications
    - transactions
    - security

Current Version: Java EE 7



As the 10/2013 the java platform is at the version 7.

Source: The Java™ EE 5 Tutorial, Third Edition: For Sun Java System Application Server Platform Edition 9 By Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, Kim Haase, Publisher: Addison Wesley Professional Pub Date: November 03, 2006 Print ISBN-10: 0-321-49029-0 Print ISBN-13: 978-0-321-49029-2 Pages: 1360

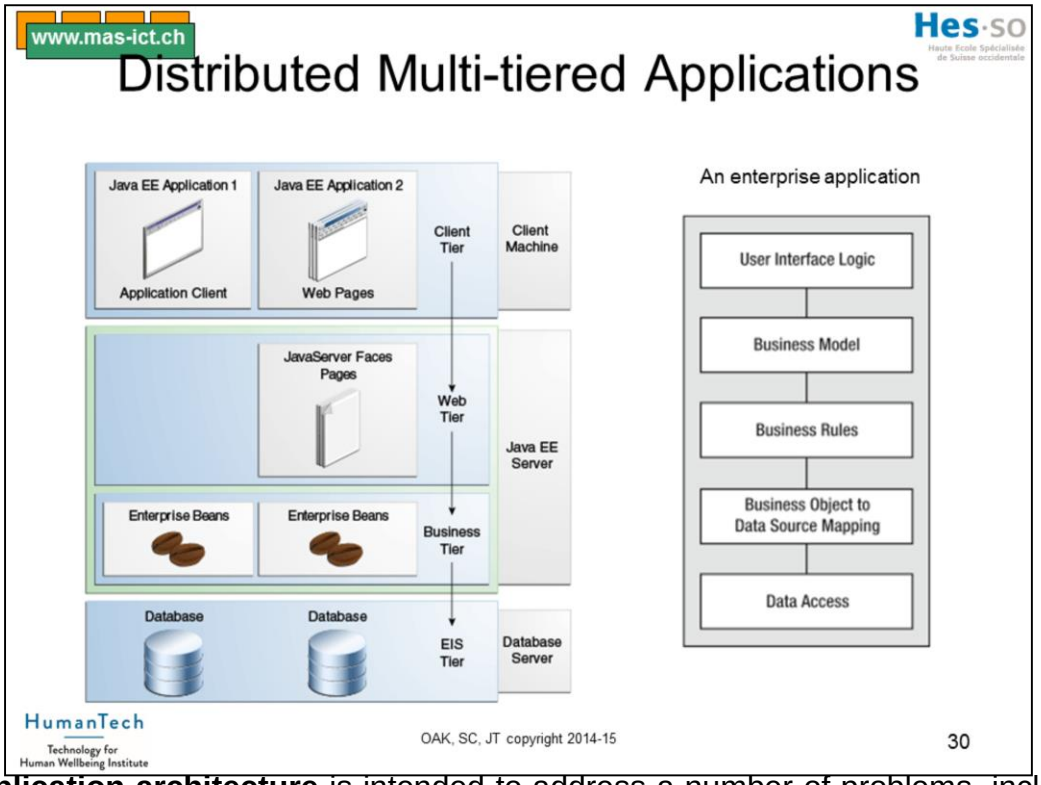
Java EE Application Model

The Java EE 5 platform introduces a simplified programming model. With Java EE 5 technology, XML deployment descriptors are now optional. Instead, a developer can simply enter the information as an annotation directly into a Java source file, and the Java EE server will configure the component at deployment and runtime.

Java EE is an open standard for building web-based enterprise applications. Containers provide several features so that application developers won't have to write everything from scratch. Java EE provides the consumer with several choices between platform and vendor. Open source application servers are gaining a lot of momentum due to low cost of entry and improving quality.

Layered Execution Model:

The benefit gained by embracing Java EE as a deployment platform is hardware and operating system independence. To utilize these benefits, we write our applications to adhere to formal specifications and deploy them to an application server running in a Java Virtual Machine (JVM) on an operating system on a physical computer (hardware). In its simplest form, a Java EE application requires all of these components running on a single machine, shown in Figure (slide).



n-tier application architecture is intended to address a number of problems, including the following:

• The high cost of maintenance when business rules change. N-tier applications have improved maintainability.

• Inconsistent business rule implementation between applications. N-tier applications provide consistency.

• Inability to share data or business rules between applications. N-tier applications offer interoperability.

• Inability to provide web-based front ends to line-of-business applications. N-tier applications are flexible.

• Poor performance and inability to scale applications to meet increased user load. N-tier applications are scalable.

• Inadequate or inconsistent security across applications. N-tier applications can be designed to be secure.

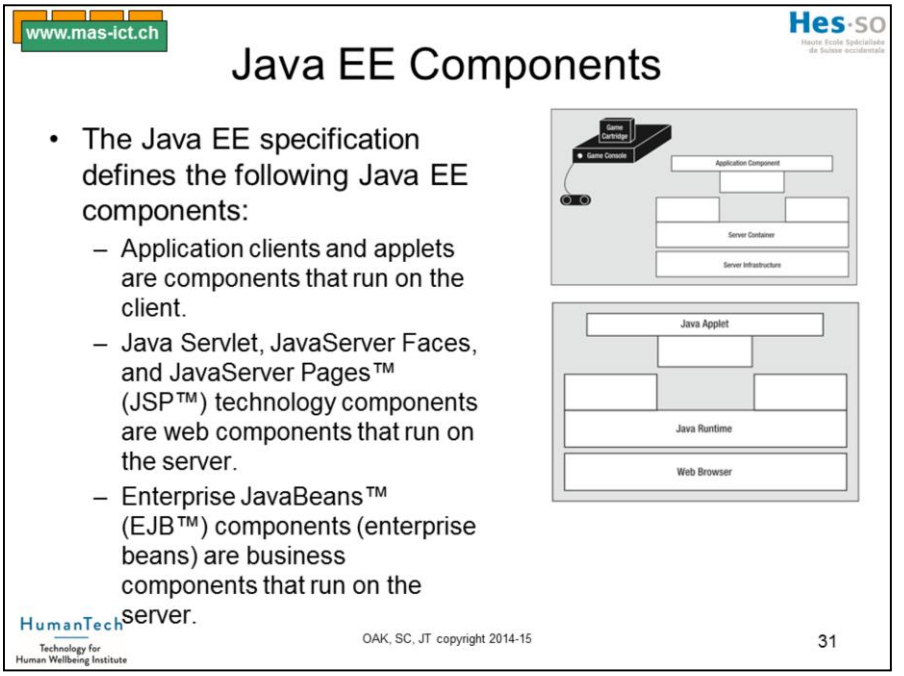
The Java EE architecture is based on the notion of n-tier applications. Java EE makes it very easy to build industrial-strength applications based on two, three, or more application layers, and provides all of the plumbing and wiring to make that possible. The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a Java EE application are installed on different machines depending on the tier in the multitiered Java EE environment to which the application component belongs. Figure shows two multitiered Java EE applications divided into the tiers described in the following list:

•Client-tier components run on the client machine.

•Web-tier components run on the Java EE server.

•Business-tier components run on the Java EE server.

•Enterprise information system (EIS)-tier software runs on the EIS server.



Java EE Components

Java EE applications are made up of components. A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.

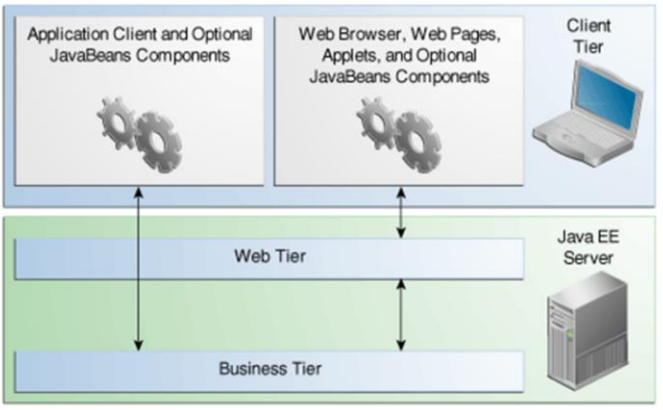
Java EE Clients: A Java EE client can be a web client or an application client.

Web Clients: A web client consists of two parts: (1) dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier, and (2) a web browser, which renders the pages received from the server. A web client is sometimes called a thin client. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

Applets: A web page received from the web tier can include an embedded applet. An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file for the applet to successfully execute in the web browser. Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

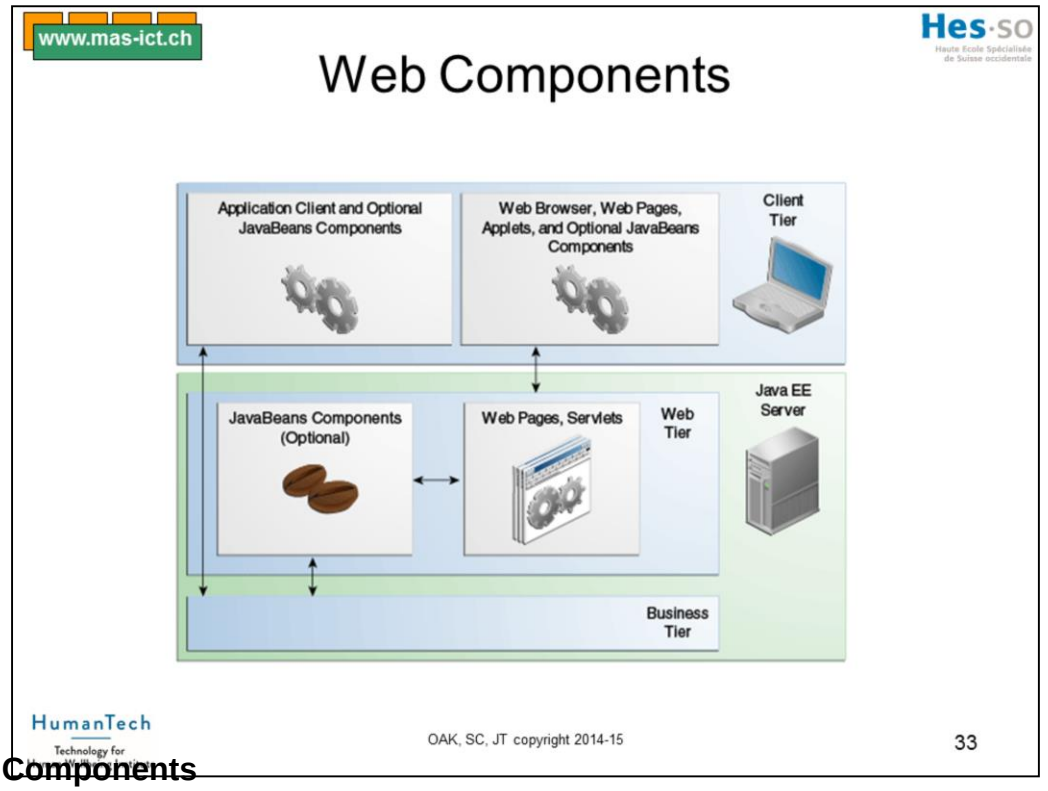
Application Clients: An application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible. Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier. Application clients written in languages other than Java can interact with Java EE servers, enabling the Java EE platform to interoperate with legacy systems, clients, and non-Java languages.

Java EE Server Communications



Java EE Server Communications

Figure shows the various elements that can make up the client tier. The client communicates with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the web tier. Your Java EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.



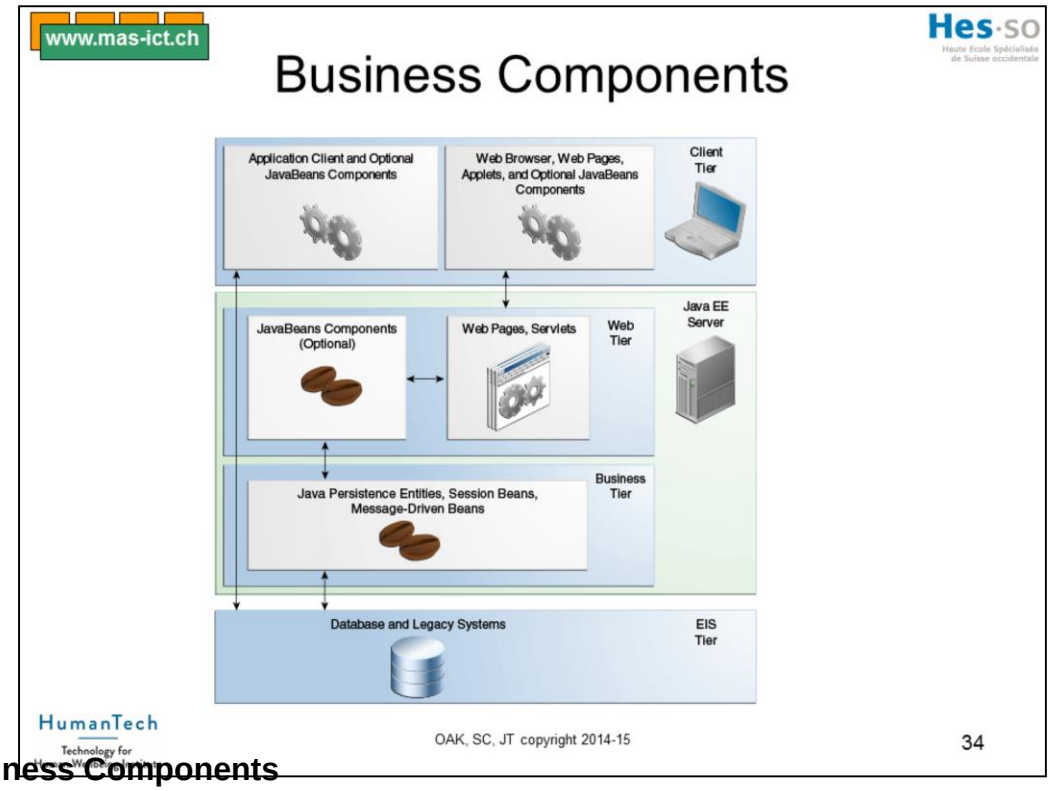
Web Components

Java EE web components are either **servlets** or pages created using **JSP technology** (JSP pages) and/or **JavaServer Faces technology**. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. JavaServer Faces technology builds on servlets and JSP technology and provides a user interface component framework for web applications.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the Java EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

JavaBeans are reusable software components for Java that can be manipulated visually in a builder tool. Practically, they are classes written in the Java programming language conforming to a particular convention. They are used to encapsulate many objects into a single object (the bean), so that they can be passed around as a single bean object instead of as multiple individual objects.

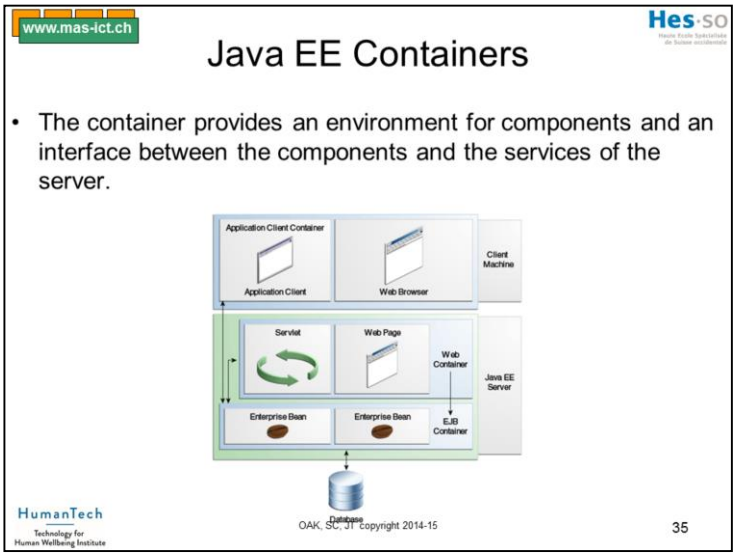


Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, Java EE application components might need access to enterprise information systems for database connectivity.



Java EE Containers

Containers provide the runtime support for Java EE application components. Containers provide a federated view of the underlying Java EE APIs to the application components. Java EE application components never interact directly with other Java EE application components. They use the protocols and methods of the container for interacting with each other and with platform services. Interposing a container between the application components and the Java EE services allows the container to transparently inject the services required by the component, such as declarative transaction management, security checks, resource pooling, and state management.

The Java EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand:

•Provide Java EE runtime environments with J2SE

•Provide required services to application components: example services – JMS (Java Message Service), JTA (Java Transaction API); application components: Applets, Java applications, Servlets/JSPs, and EJBs

•Understand application component deployment formats

•Interposes between application components to transparently inject services required: transactions, security checks, resource pooling

•Synonymous with application server from the application developer point of view

Containers are like the rooms in the house. People and things exist in the rooms, and interface with the infrastructure through well-defined interfaces. In an application server, web and business components exist inside containers and interface with the Java EE infrastructure through well-defined interfaces.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, **it must be assembled into a Java EE module and deployed into its container.**

Container settings customize the underlying support provided by the Java EE server, including services such as security, transaction management, Java Naming and Directory Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

•The Java EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.

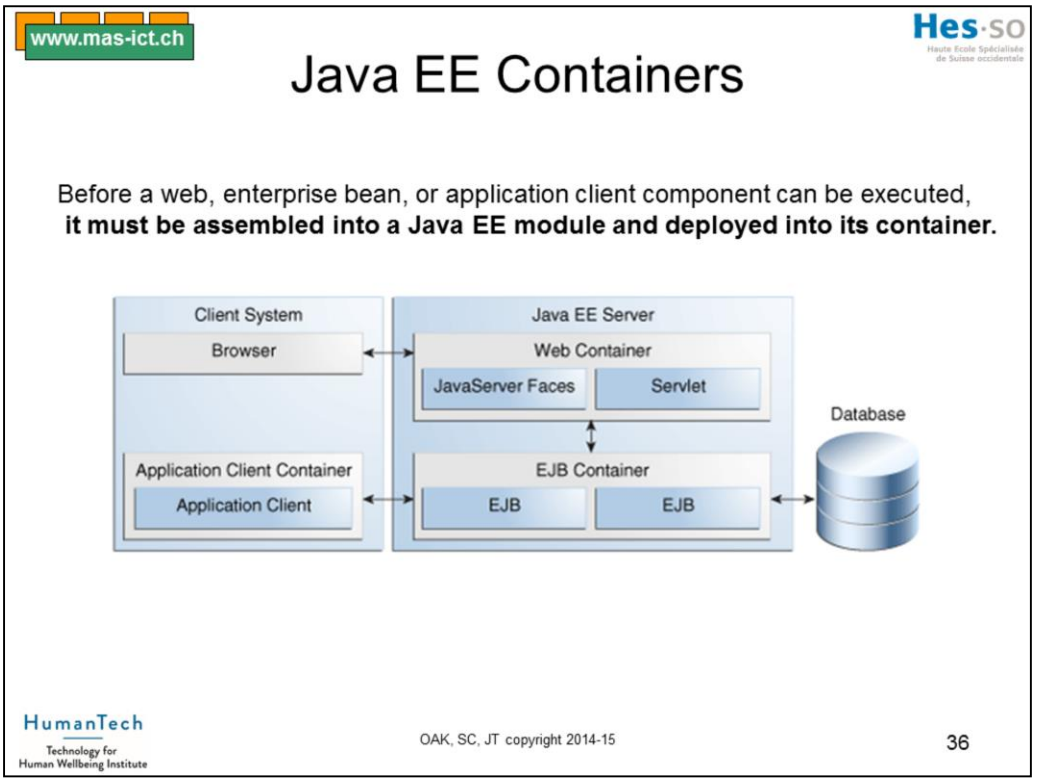
•The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.

•JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.

•The Java EE remote connectivity model manages low-level communications between clients and

enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the Java EE architecture provides configurable services, application components within the same Java EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.



Container Types

The deployment process installs Java EE application components in the Java EE containers as illustrated in Figure:

•Java EE server: The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.

•Enterprise JavaBeans (EJB) container: Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.

•Web container: Manages the execution of JSP page and servlet components for Java EE applications. Web components and their container run on the Java EE server.

•Application client container: Manages the execution of application client components. Application clients and their container run on the client.

•Applet container: Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The Java EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

www.mas-ict.ch

Java EE 7 Architecture / APIs

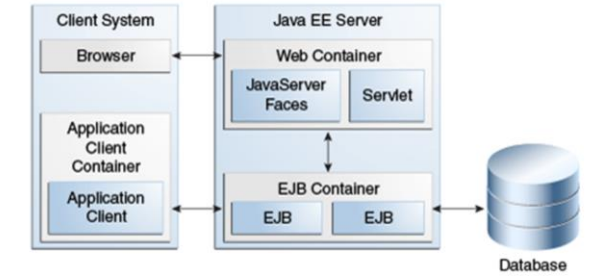
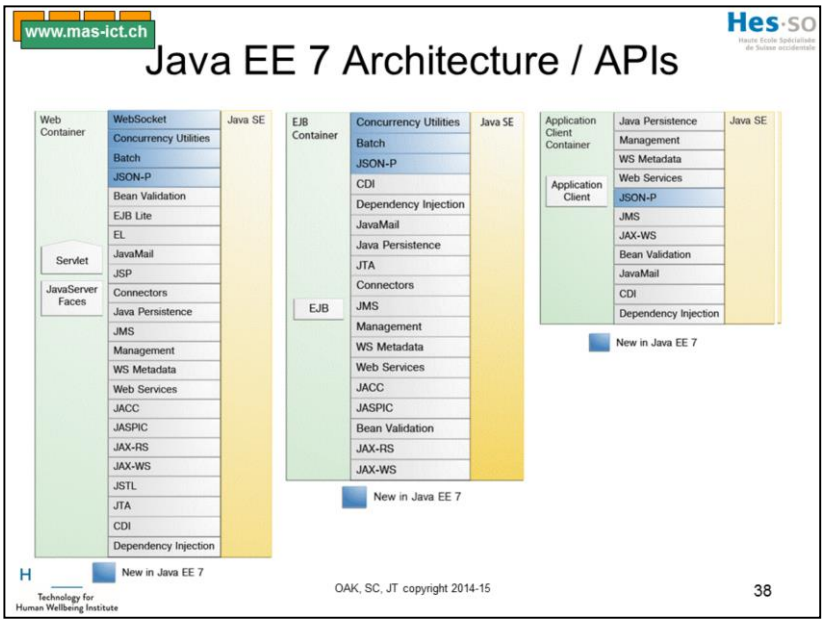


Diagram of Java EE containers and their relationships

Source: http://docs.oracle.com/javaee/7/tutorial/doc/overview008.htm



Source: http://docs.oracle.com/javaee/7/tutorial/doc/overview008.htm

Enterprise JavaBeans Technology

An Enterprise JavaBeans (EJB) component, or **enterprise bean**, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server. Enterprise beans are either session beans or message-driven beans.

Java Servlet Technology

Java Servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer Faces Technology

JavaServer Faces technology is a user interface framework for building web applications.

JavaServer Pages Technology

JavaServer Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text:

Static data, which can be expressed in any text-based format such as HTML or XML

JSP elements, which determine how the page constructs dynamic content

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications.

Java Persistence API, Java Transaction API, Java API for RESTful Web Services, Managed Beans, Contexts and Dependency Injection for the Java EE Platform (JSR 299), Dependency Injection for Java (JSR 330), Bean Validation, Java Message Service API, EE Connector Architecture, JavaMail API, Java Authorization Contract for Containers, Java Authentication Service Provider Interface for Containers

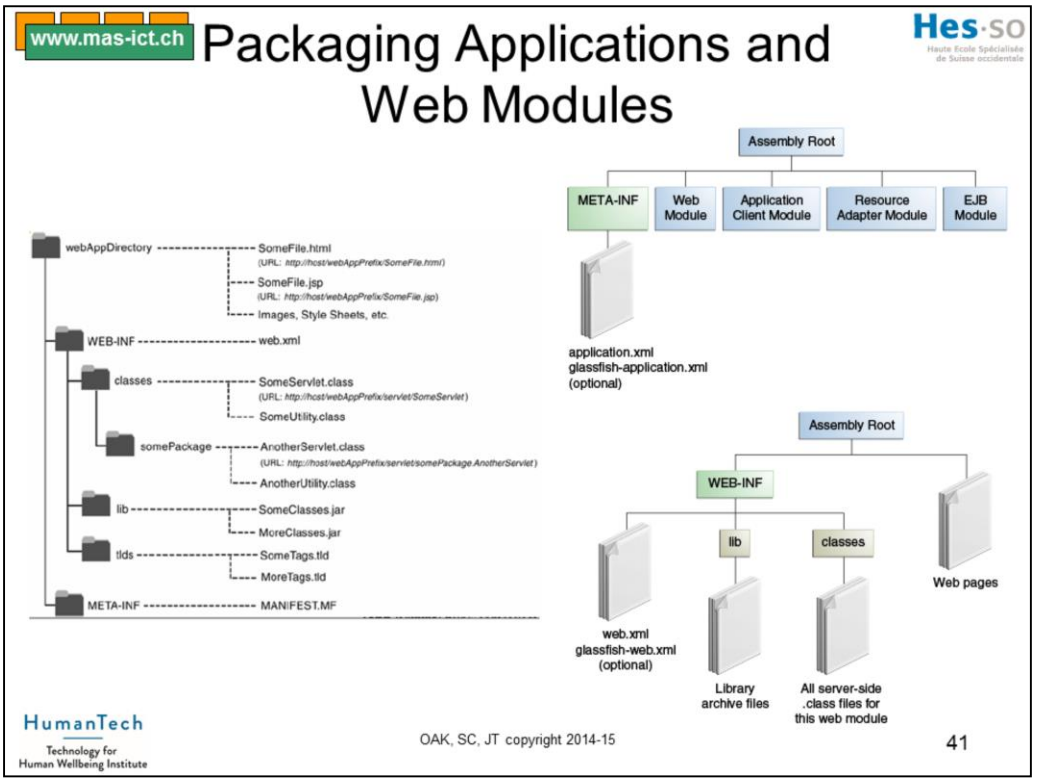
www.mas-ict.ch

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

Web Application Life Cycle

- The process for creating, deploying, and executing a web application can be summarized as follows:
  - Develop the web component code.
  - Develop the web application deployment descriptor.
  - Compile the web application components and helper classes referenced by the components.
  - Optionally package the application into a deployable unit.
  - Deploy the application into a web container.
  - Access a URL that references the web application.

A web application consists of web components, static resource files such as images, and helper classes and libraries. The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop.



Web Modules

In the Java EE architecture, web components and static web content files such as images are called web resources. A web module is the smallest deployable and usable unit of web resources. A Java EE web module corresponds to a web application as defined in the Java Servlet specification.

A web module has a specific structure. The top-level directory of a web module is the document root of the application. The document root is where JSP pages, client-side classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named /WEB-INF/, which contains the following files and directories:

•web.xml: The web application deployment descriptor

•Tag library descriptor files

•classes: A directory that contains server-side classes: servlets, utility classes, and JavaBeans components

•tags: A directory that contains tag files, which are implementations of tag libraries

•lib: A directory that contains JAR archives of libraries called by server-side classes

Files in WEB-INF not directly accessible to clients. A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a web archive (WAR) file. Because the contents and use of WAR files differ from those of JAR files, WAR file names use a .war extension. The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet Specification.

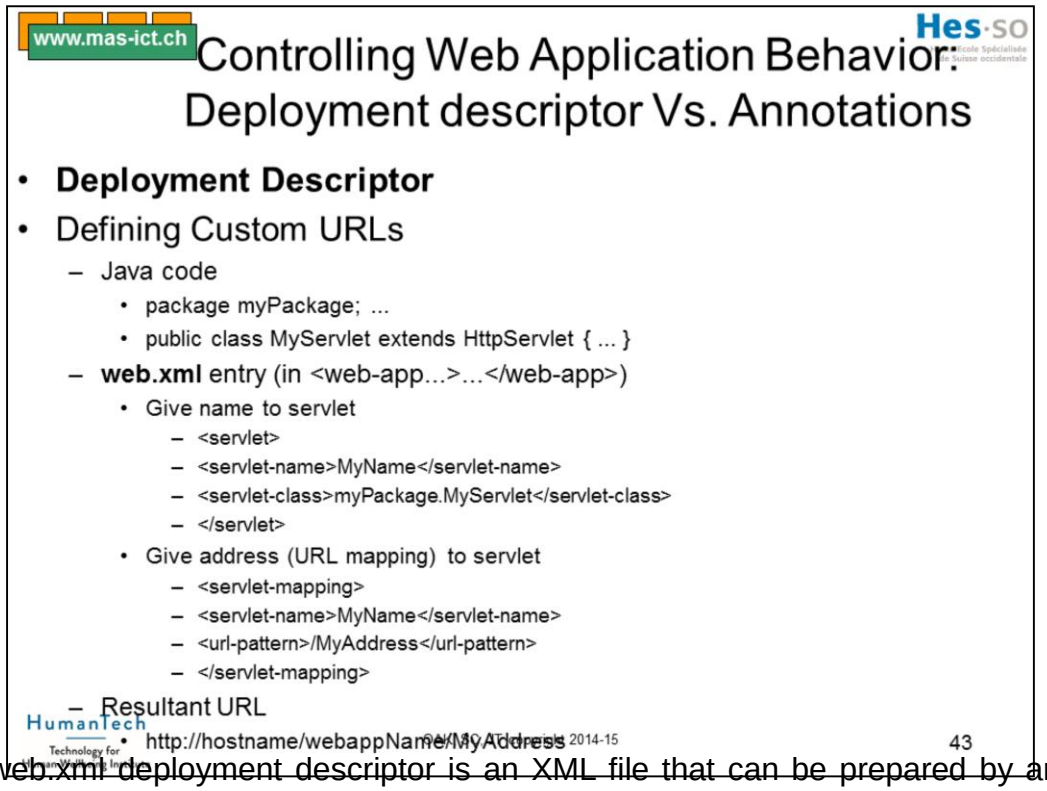
To deploy a WAR on the Application Server, the file must also contain a runtime deployment descriptor. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application's resources to the Application Server's resources.

Controlling Web Application Behavior: Deployment descriptor Vs. Annotations

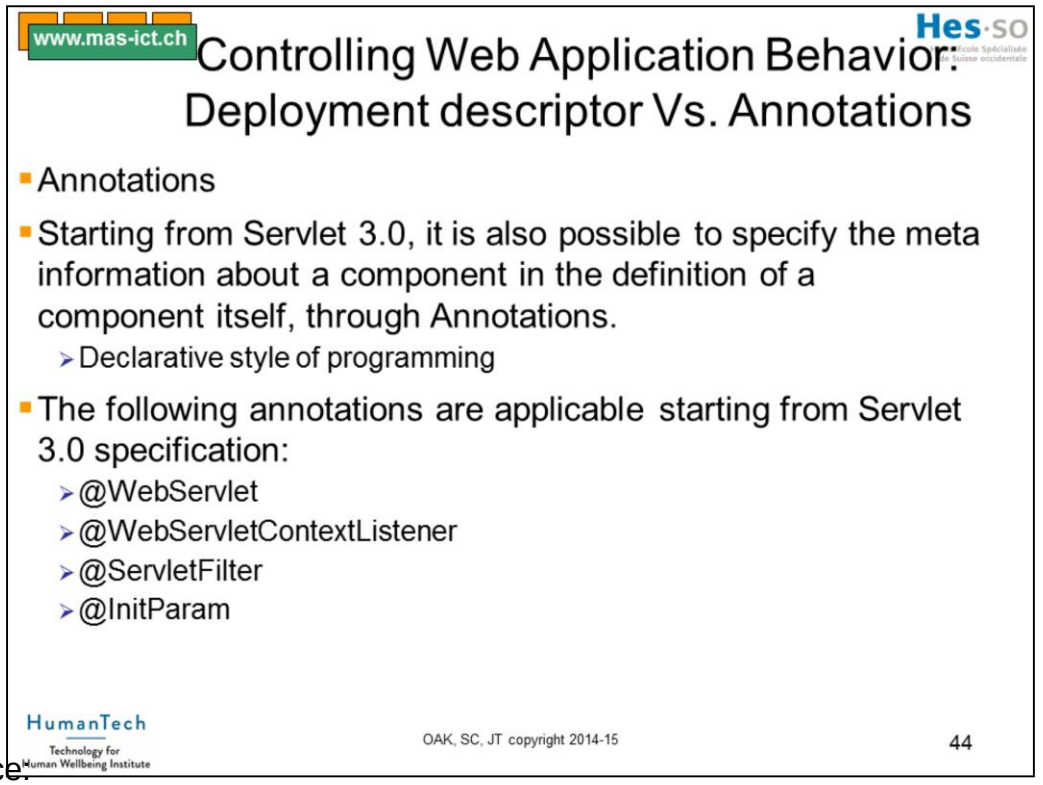
- **Deployment Descriptor**
  - It describes the deployment settings of an application, a module, or a component.
  - Because deployment descriptor information is declarative, it can be changed without the need to modify the source code.
  - At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.
- Deployment information is most commonly specified in the source code by **annotations**.
  - Deployment descriptors, if present, override what is specified in the source code.

Basic format:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance“
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd“
version="2.4">
<!-- "Real" elements go here. All are optional. -->
</web-app>
```

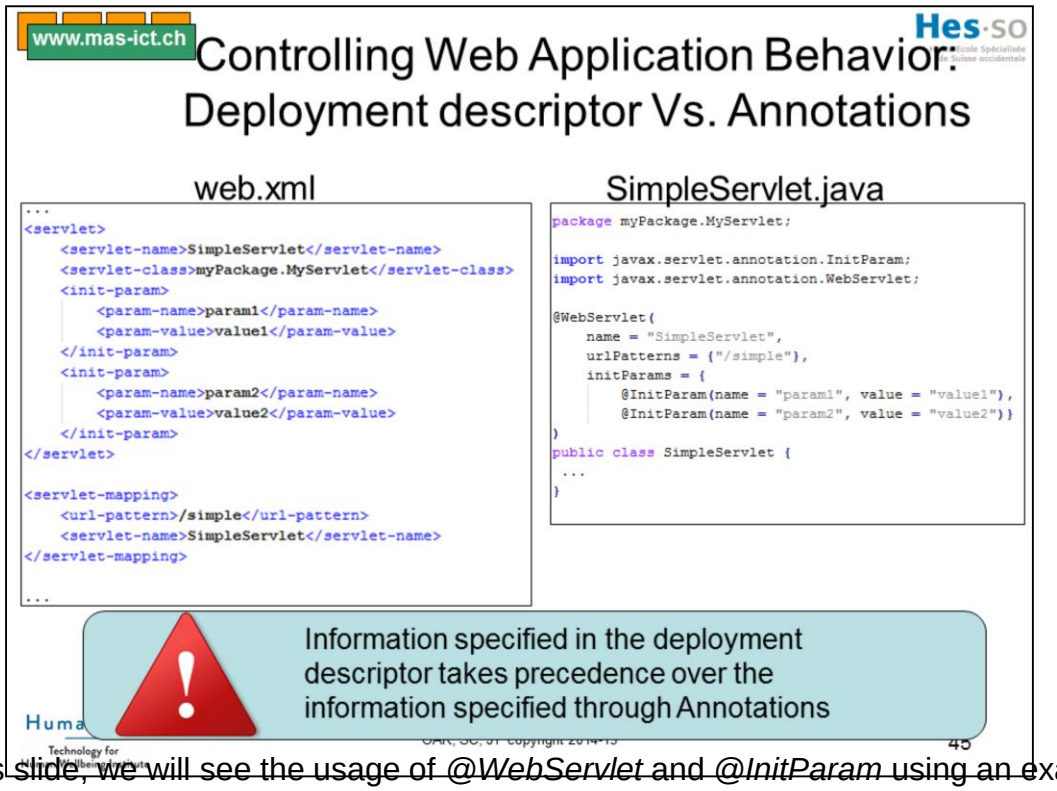


The web.xml deployment descriptor is an XML file that can be prepared by any plain-text editor. The application-server vendor may provide a GUI tool for creation deployment descriptors. While web.xml is a required file for every Web application deployed in a J2EE compliant application server, vendors may also create additional deployment descriptor files. Please refer to your vendor's documentation describing the servlet deployment procedure.



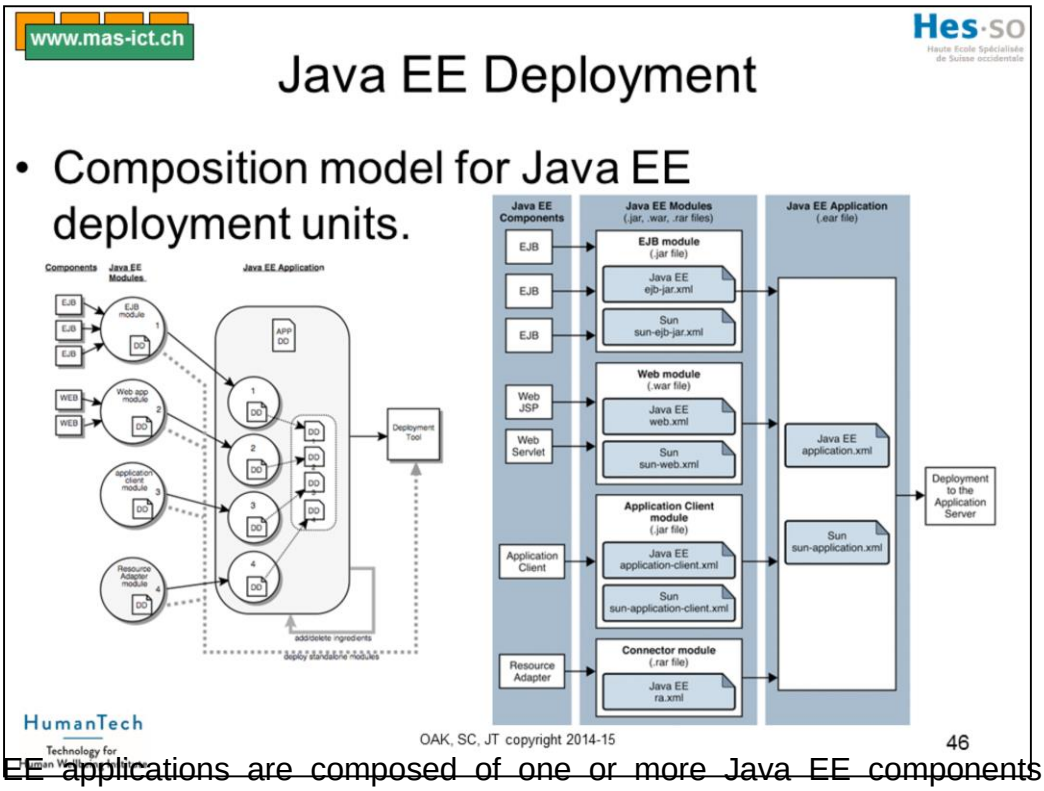
Source:

- http://www.javabeat.net/2008/12/new-features-in-servlets-3-0/



In this slide, we will see the usage of *@WebServlet* and *@InitParam* using an example.

In the example, we have declared a class by name SimpleServlet and this class is not made to extend or implement any of the Servlet/HttpServlet types. Instead, to qualify this class as a Servlet class we have annotated using *@WebServlet* annotation. Note that the name of the servlet is SimpleServlet as specified through the *name* attribute. The attribute *urlPatterns* defines a set of url-patterns that can be used to invoke the Servlet. The **Servlet Container** after scanning this class will generate the deployment descriptor (similar to the one on the left side).



Java EE applications are composed of one or more Java EE components and an optional Java EE application deployment descriptor. The deployment descriptor, if present, lists the application's components as modules. If the deployment descriptor is not present, the application's modules are discovered using default naming rules. A Java EE module represents the basic unit of composition of a Java EE application. Java EE modules consist of one or more Java EE components and an optional module level deployment descriptor. The flexibility and extensibility of the Java EE component model facilitates the packaging and deployment of Java EE components as individual components, component libraries, or Java EE applications.

Enterprise Application Archive (EAR):

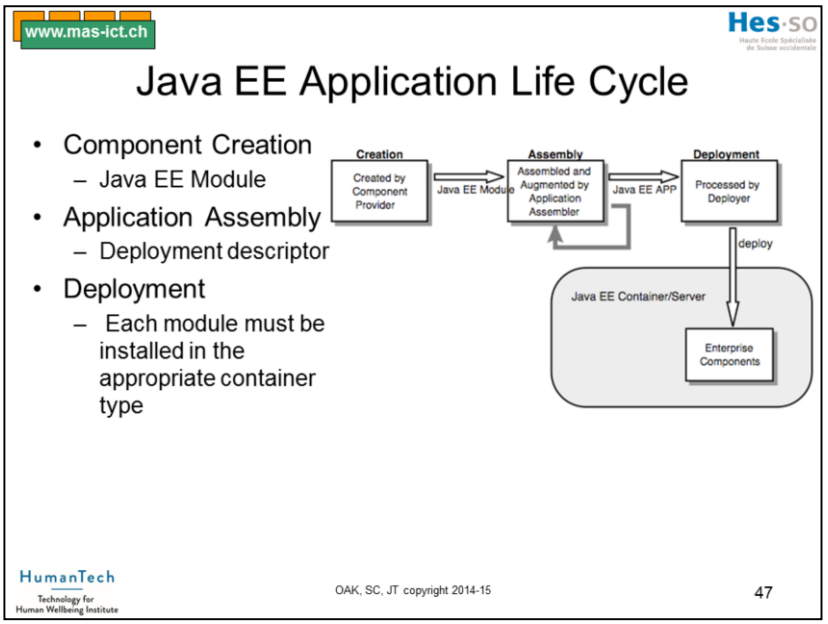
WAR(s): directory or archive

EJB(s). directory or archive

Client JAR(s). client applications

Utility Classes(s). directory or archive, supplies external source utility classes, referenced through MANIFESTs

Resource Adapters(s): custom resource drivers



Application Development Life Cycle

The development life cycle of a Java EE application begins with the creation of discrete Java EE components. These components may then be packaged with a module level deployment descriptor to create a Java EE module. Java EE modules can be deployed as stand-alone units or can be assembled with a Java EE application deployment descriptor and deployed as a Java EE application.

Component Creation

The EJB, servlet, application client, and Connector specifications include the XML Schema definition of the associated module level deployment descriptors and component packaging architecture required to produce Java EE modules. A Java EE module is a collection of one or more Java EE components of the same component type (web, EJB, application client, or Connector) with an optional module deployment descriptor of that type.

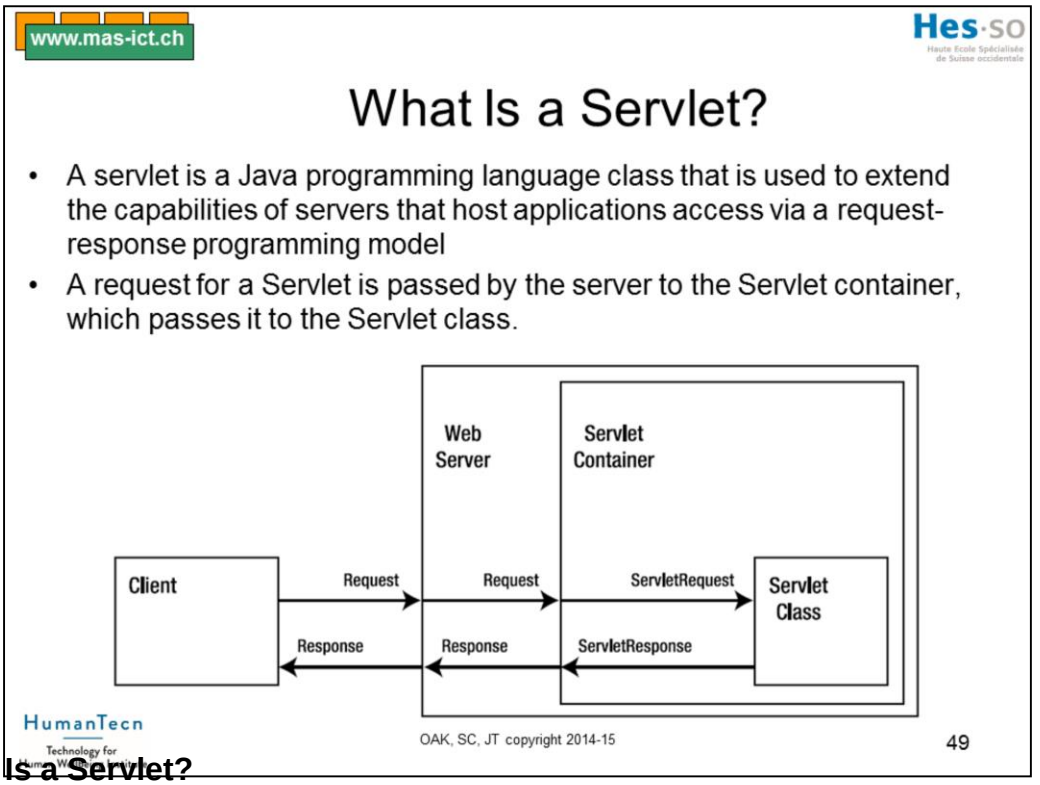
Application Assembly

A Java EE application may consist of one or more Java EE modules and one Java EE application deployment descriptor. A Java EE application is packaged using the Java Archive (JAR) file format into a file with a .ear (Enterprise ARchive) filename extension. A minimal Java EE application package will only contain Java EE modules and the application deployment descriptor. A Java EE application package may also include libraries referenced by Java EE modules, help files, and documentation to aid the deployer. The Java EE application deployment descriptor represents the top level view of a Java EE application's contents. The Java EE application deployment descriptor is specified by an XML schema or document type definition.

Deployment

During the deployment phase of an application's life cycle, the application is installed on the Java EE platform and then is configured and integrated into the existing infrastructure. Each Java EE module listed in the application deployment descriptor (or discovered using the default rules described below) must be deployed according to the requirements of the specification for the respective Java EE module type. Each module listed must be installed in the appropriate container type and the environment properties of each module must be set appropriately in the target container to reflect the values declared by the deployment descriptor element for each component.

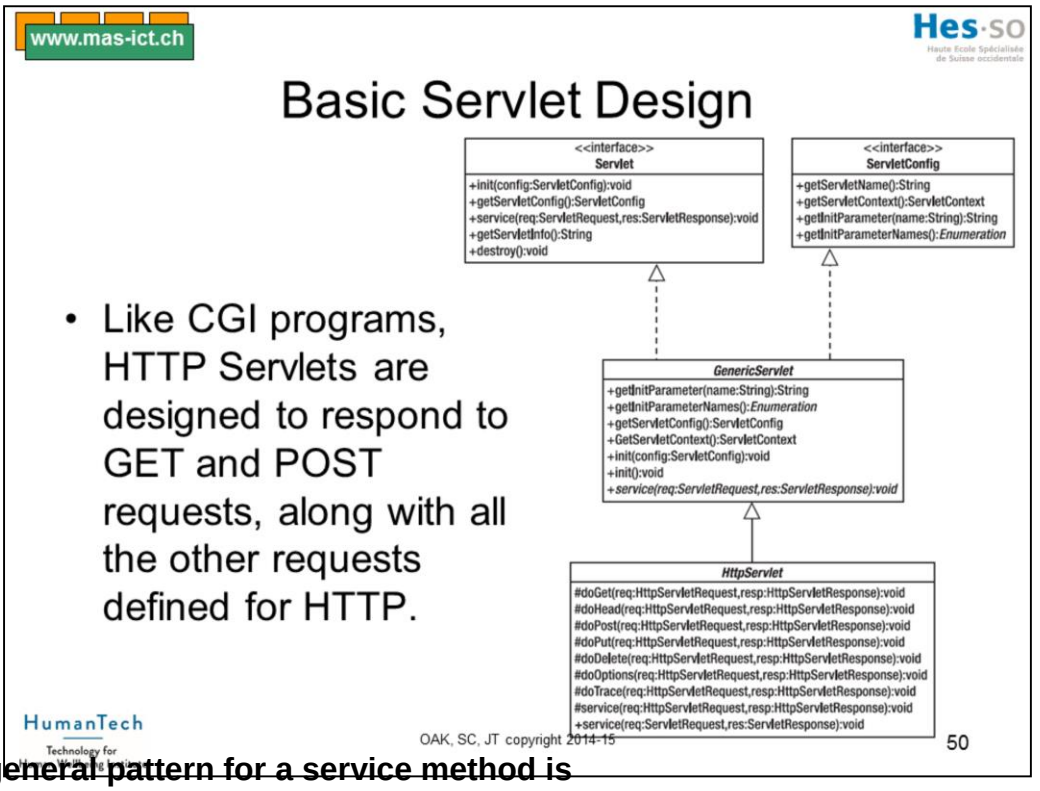
As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java servlet technology was created as a portable way to provide dynamic, user-oriented content.



What Is a Servlet?

A servlet is a Java programming language class that is used to extend the capabilities of servers that host applications access via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

When a client (usually, but not necessarily, a web browser) makes a request to the server, and the server determines that the request is for a Servlet resource, it passes the request to the Servlet container. The container is the program responsible for loading, initiating, calling, and releasing Servlet instances. The Servlet container takes the HTTP request; parses its request URI, the headers, and the body; and stores all of that data inside an object that implements the javax.servlet.ServletRequest interface. It also creates an instance of an object that implements javax.servlet.ServletResponse. The response object encapsulates the response back to the client. The container then calls a method of the Servlet class, passing the request and response objects. The Servlet processes the request and sends a response back to the client.

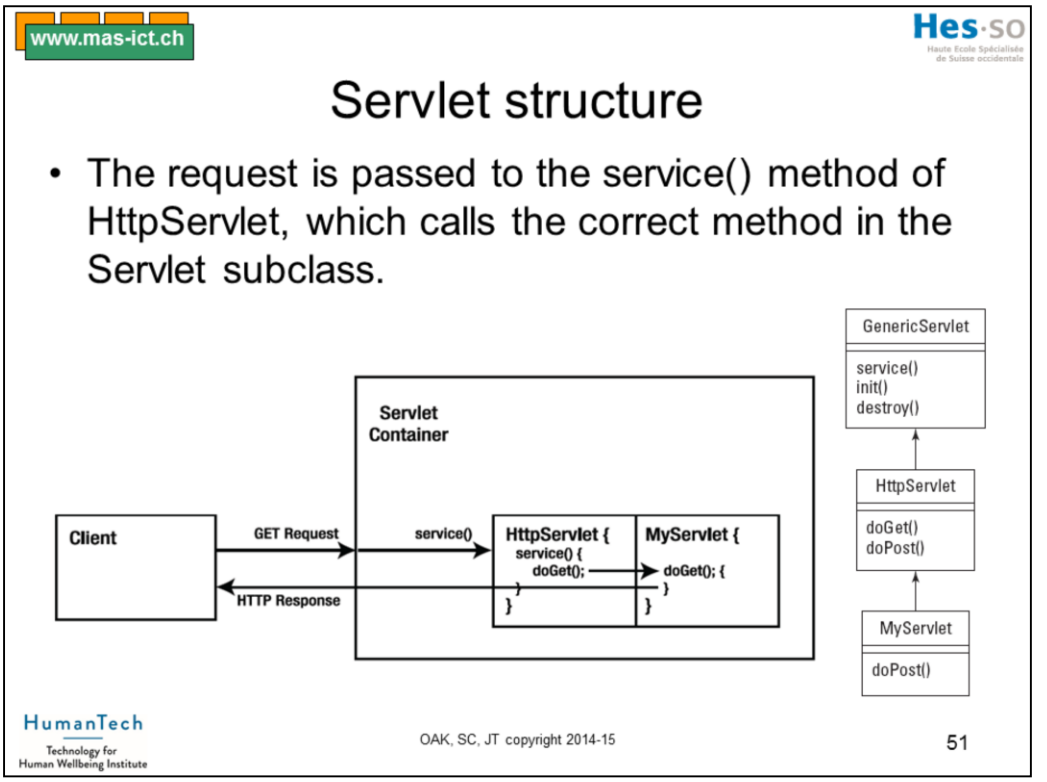


The general pattern for a service method is

- to extract information from the request,
- access external resources,
- and then populate the response based on that information.

When writing Servlets, you will usually extend a class named javax.servlet.http.HttpServlet. This is a base class that provides support for HTTP requests. The HttpServlet class, in turn, extends javax.servlet.GenericServlet, which provides some basic Servlet functionality. Finally, GenericServlet implements the primary Servlet API interface, javax.servlet.Servlet.

It also implements an interface called ServletConfig, which allows it to provide easy access to Servlet configuration information. Notice that Servlet defines only a small number of methods. You can probably guess that init() and destroy() don’t handle any requests. The getServletConfig() and getServletInfo() methods don’t handle requests either. That leaves only service() to handle requests.



The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods. The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps:

a. Loads the servlet class.

b. Creates an instance of the servlet class.

c. Initializes the servlet instance by calling the init method.

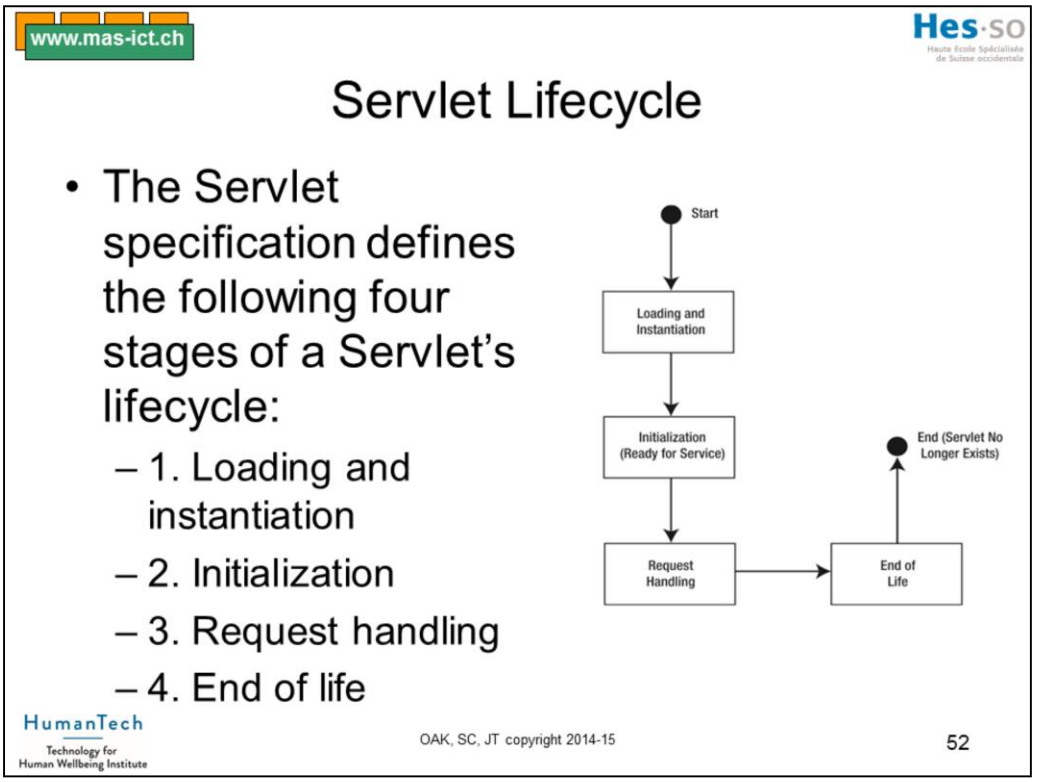
Invokes the service method, passing request and response objects.

The doPost() and doGet() Methods:

HttpServlet is intended to respond to HTTP requests, and it must handle requests for GET,POST, HEAD, and so on. Thus, HttpServlet defines additional methods. It defines a doGet() to handle GET requests, doPost() to handle POST requests, and so on; there is a do*XXX() method for* every HTTP method.

When the Servlet container receives the HTTP request, it maps the URI to a Servlet. It then calls the service() method of the Servlet. Assuming the Servlet extends HttpServlet, and overrides only doPost() or doGet(), the call to service() will go to the HttpServlet parent class. The service() method determines which HTTP method the request used and calls the correct do*XXX() method, as illustrated in Figure.*

Each method—doPost(), doGet(), and so on—accepts two parameters. The HttpServletRequest object encapsulates the request to the server. It contains the data for the request, as well as some header information about the request. Using methods defined by the request object, the Servlet can access the data submitted as part of the request. The HttpServletResponse object encapsulates the response to the client. Using the response object and its methods, you can return a response to the client.

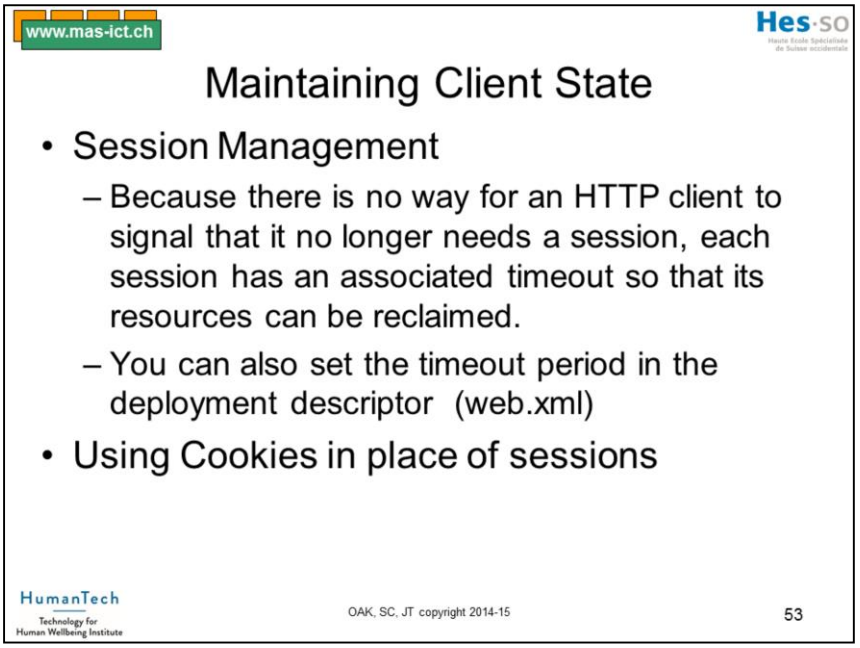


Loading and Instantiation: In the first stage of the lifecycle, the Servlet class is loaded from the classpath and instantiated by the Servlet container. How does the Servlet container know which Servlets to load? It knows by reading the deployment descriptor. The Servlet container reads each web.xml file, and loads the Servlet classes identified in the deployment descriptor. Then the container instantiates each Servlet by calling its no-argument constructor.

Initialization : After the Servlet is loaded and instantiated, the Servlet must be initialized. This occurs when the container calls the init(ServletConfig) method. If your Servlet does not need to perform any initialization, the Servlet does not need to implement this method. The method is provided for you by the GenericServlet parent class. The init() method allows the Servlet to read initialization parameters or configuration data, initialize external resources such as database connections, and perform other one-time activities.

Request Handling: the primary method defined for servicing requests during this phase of the Servlet lifecycle is the service() method. As each request comes to the Servlet container, the container calls the service() method of the appropriate Servlet to handle the request. Since you will almost always be subclassing HttpServlet, however, your Servlet only needs to override doPost() and/or doGet() to handle requests.

End of Life When the Servlet container needs to unload the Servlet, either because it is being shut down or for some other reason such as a ServletException, the Servlet container will call the destroy() method. However, prior to calling destroy(), the container must allow time for any request threads that are still processing to complete their processing. After these threads are finished processing, or after a server-defined timeout period, the container is allowed to call destroy().



Session Management There's one big challenge with relying on HTTP for web applications: HTTP is a stateless protocol. Each request and response stands alone. Without session management, each time a client makes a request to a server, the server doesn't remember any information about the client. From the server's point of view, it's a brand-new user with a brand-new request, with no relation to any other request. To deal with this issue, web applications use the concept of a session. The session could be terminated by the client's request, or the server could automatically close it after a certain period of time. A session can last for just a few minutes, or it could last days or weeks or months (if the application were willing to let a session last that long).

A web application can associate various requests into a single session by using additional information that the client sends with each request. The following are two common ways to do this:

• The application server sends a session ID as a cookie to the client. When the client makes a request, the session ID is sent back to the server, and the application server can use the ID to associate the request with a session.

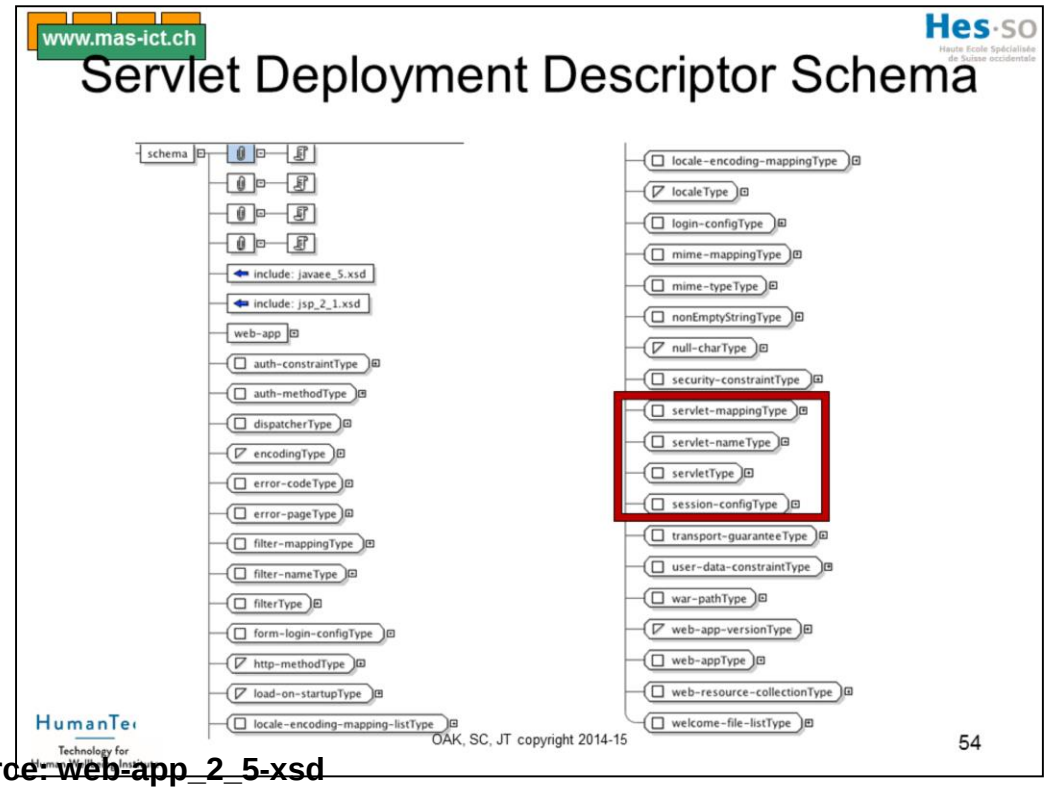
• The web application embeds the session ID in the URLs in a web response. When the client clicks a link in the response web page, the URL sent to the server includes the session ID, and the web application can use the ID to associate the request with the session.

Cookies:

Instead of storing information about a client's request in a session object, you could send all that information as a cookie to the client. For example, in an e-commerce application, you could store all the shopping cart information in cookies that are sent to the client. The session object has two advantages over cookies:

• clients can reject cookies sent by a server; however session objects live on the server, and can always be created, either by setting the session ID in a cookie or through URL rewriting.

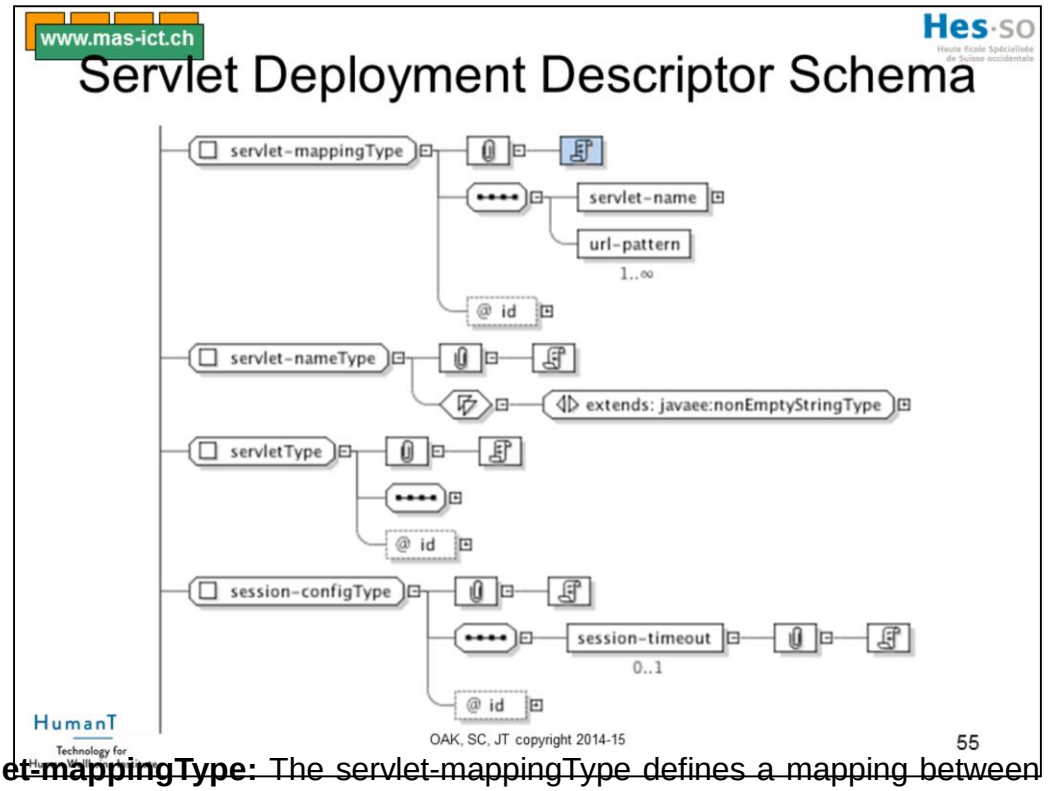
• Cookies can store only text data, so you are limited to storing text information or information that can be represented by text. Using a session object, you can store any Java object in a session.



Source: web-app_2_5-xsd

name="web-app" type="javaee:web-appType: The web-app element is the root of the deployment descriptor for a web application. Note that the sub-elements of this element can be in the arbitrary order. Because of that, the multiplicity of the elements of distributable, session-config, welcome-file-list, jsp-config, login-config, and locale-encoding-mapping-list was changed from "?" to "*" in this schema. However, the deployment descriptor instance file must not contain multiple elements of session-config, jsp-config, and login-config. When there are multiple elements of welcome-file-list or locale-encoding-mapping-list, the container must concatenate the element contents. The multiple occurrence of the element distributable is redundant and the container treats that case exactly in the same way when there is only one distributable.

mime-typeType: The mime-typeType is used to indicate a defined mime type. Example: "text/plain" . Used in: mime-mapping



servlet-mappingType: The servlet-mappingType defines a mapping between a servlet and a url pattern. Used in: web-app.

servlet-nameType: The servlet-name element contains the canonical name of the servlet. Each servlet name is unique within the web application.

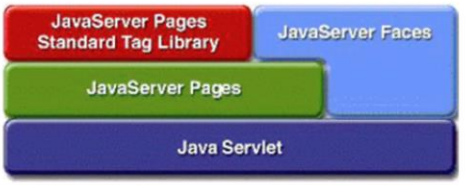
servletType: The servletType is used to declare a servlet. It contains the declarative data of a servlet. If a jsp-file is specified and the load-on-startup element is present, then the JSP should be precompiled and loaded. Used in: web-app.

session-configType: The session-configType defines the session parameters for this web application. Used in: web-app

JSP

Outline

- JSP
  - Servlet problems
  - Directives
  - Scriptlets
  - Expressions
  - Declarations

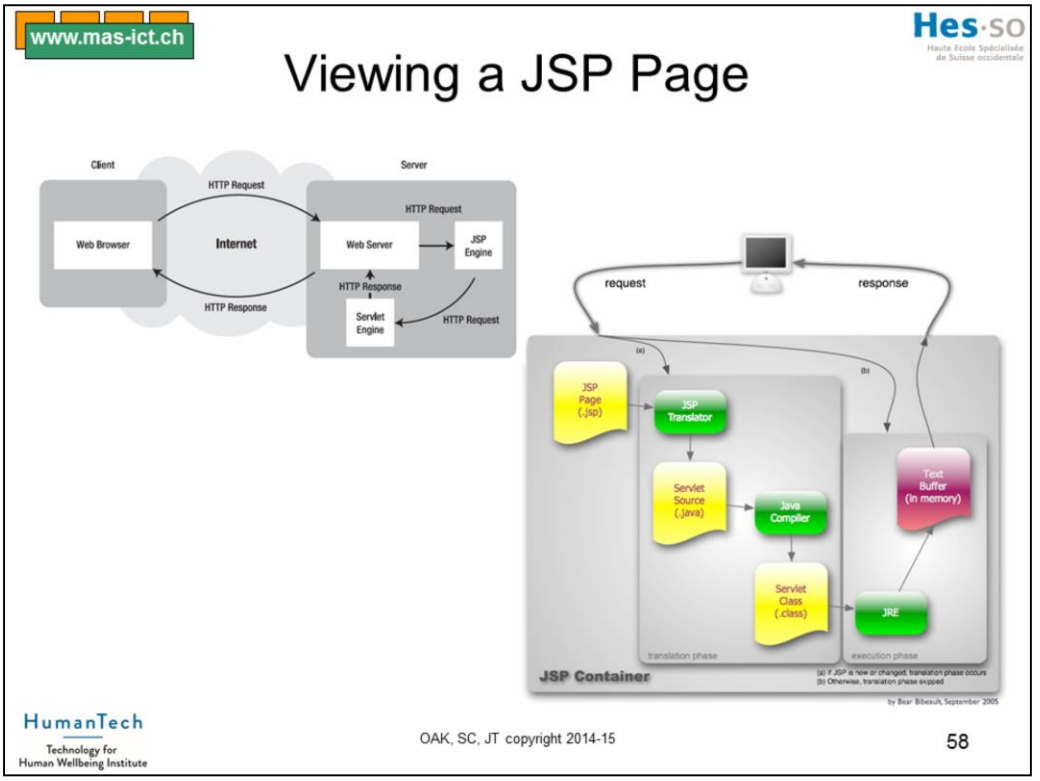


JSP – Java Server Pages

JSTL – JSP Standard Tag Library

EL – Expression Language

JSF – Java Server Faces



Servlet Problems:

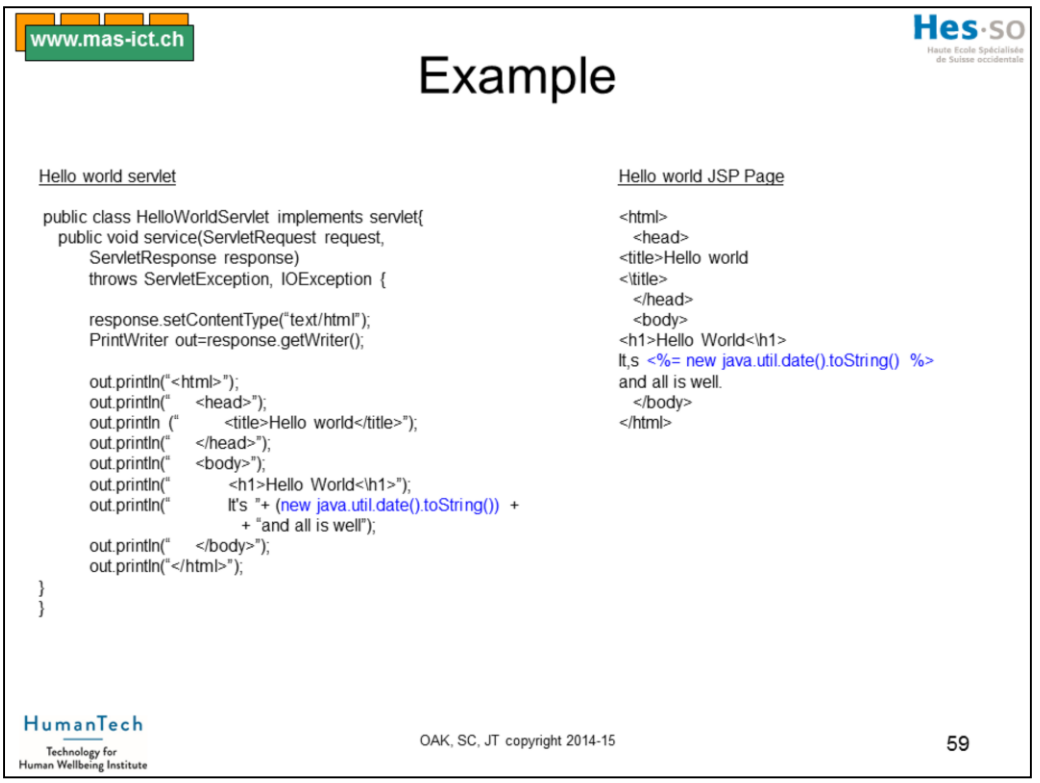
So what problems do servlets pose? Servlets raise problems with respect to the out.println statements. They are good for control, but not for presentation.

1. Detailed Java programming knowledge is needed.
2. To change the look and feel, change the servlet code and recompile.
3. Advantage of the web page development tools are utilized.

The following steps explain how the web server creates the web page:

1. As with a normal page, your browser sends an HTTP request to the web server. This doesn't change with JSP, although the URL probably ends in .jsp instead of .html.
2. The web server is not a normal server, but rather a Java server, with the extensions necessary to identify and handle Java servlets. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine.
3. The JSP engine loads the JSP page from disk and converts it into a Java servlet. From this point on, this servlet is indistinguishable from any other servlet developed directly in Java rather than JSP, although the automatically generated Java code of a JSP servlet is difficult to read, and you should never modify it by hand.
4. The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine. Note that the JSP engine only converts the JSP page to Java and recompiles the servlet if it finds that the JSP page has changed since the last request. This makes the process more efficient than with other scripting languages (such as PHP) and therefore faster.
5. A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
6. The web server forwards the HTTP response to your browser.
7. Your web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page. In fact, static and dynamic web pages are in

the same format.



Anatomy of a JSP Page:

- A JSP page is a regular web page with JSP elements for generating the parts of the page that differ for each request.

-JSP separates the request processing and the business logic code from the presentation. In servlet, HTML is embedded, here JSP elements are added to generate the dynamic content.

JSP Development

The process of developing a JSP page that can respond to client requests involves three main steps:

•Creation: The developer creates a JSP source file that contains HTML and embedded Java code.

•Deployment: The JSP is installed into a server. This can be a full Java EE server or a stand-alone JSP server.

•Translation and compilation: The JSP container translates the HTML and Java code into a Java code source file. This file is then compiled into a Java class that is executed by the server. The class file created from the JSP is known as the JSP page implementation class.

Basic JSP Lifecycle

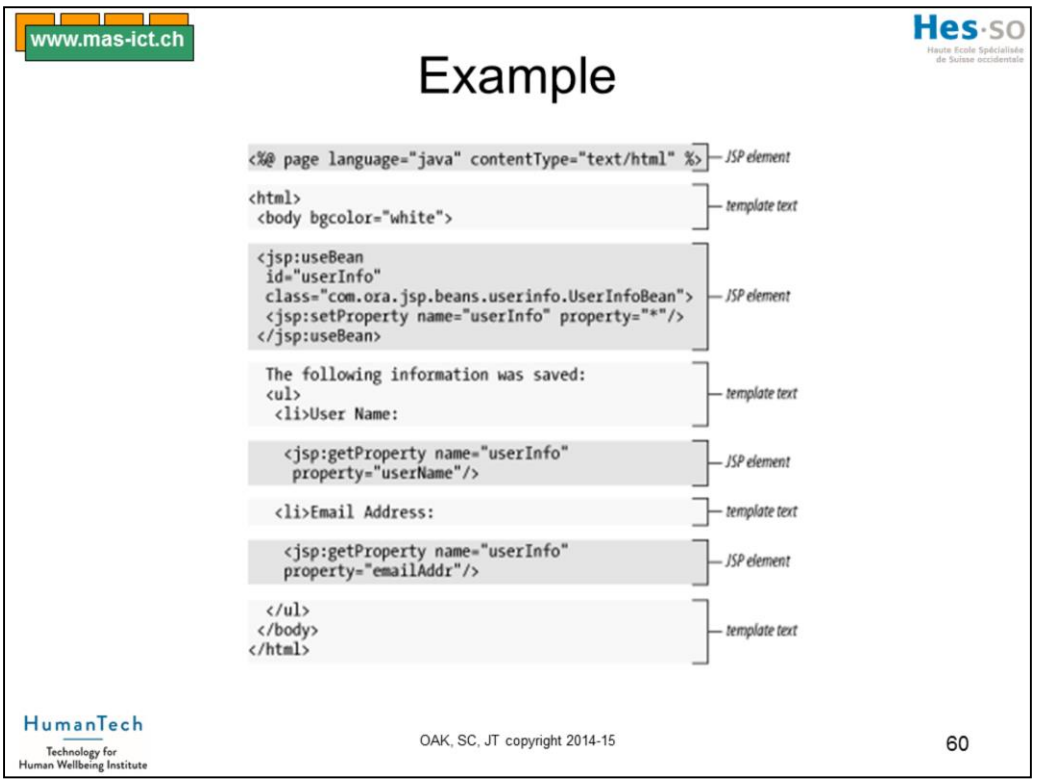
Once compilation is complete, the JSP lifecycle has these phases:

•Loading and instantiation: The server finds or creates the JSP page implementation class for the JSP page and loads it into the JVM. After the class is loaded, the JVM creates an instance of the class. This can occur immediately after loading, or it can occur when the first request is made.

•Initialization: The JSP page object is initialized. If you need to execute code during initialization, you can add a method to the page that will be called during initialization.

•Request processing: The page object responds to requests. Note that a single object instance will process all requests. After performing its processing, a response is returned to the client. The response consists solely of HTML tags or other data; none of the Java source code is sent to the client.

•End of life: The server stops sending requests to the JSP. After all current requests are finished processing, any instances of the class are released. This usually occurs when the server is being shut down, but can also occur at other times, such as when the server needs to conserve resources, when it detects an updated JSP source file, or when it needs to terminate the instance for other reasons. If you need code to execute and perform any cleanup actions, you can implement a method that will be called before the class instance is released.



Anatomy of a JSP Page:

- A JSP page is a regular web page with JSP elements for generating the parts of the page that differ for each request.

-JSP separates the request processing and the business logic code from the presentation. In servlet, HTML is embedded, here JSP elements are added to generate the dynamic content.

JSP Development

The process of developing a JSP page that can respond to client requests involves three main steps:

•Creation: The developer creates a JSP source file that contains HTML and embedded Java code.

•Deployment: The JSP is installed into a server. This can be a full Java EE server or a stand-alone JSP server.

•Translation and compilation: The JSP container translates the HTML and Java code into a Java code source file. This file is then compiled into a Java class that is executed by the server. The class file created from the JSP is known as the JSP page implementation class.

Basic JSP Lifecycle

Once compilation is complete, the JSP lifecycle has these phases:

•Loading and instantiation: The server finds or creates the JSP page implementation class for the JSP page and loads it into the JVM. After the class is loaded, the JVM creates an instance of the class. This can occur immediately after loading, or it can occur when the first request is made.

•Initialization: The JSP page object is initialized. If you need to execute code during initialization, you can add a method to the page that will be called during initialization.

•Request processing: The page object responds to requests. Note that a single object instance will process all requests. After performing its processing, a response is returned to the client. The response consists solely of HTML tags or other data; none of the Java source code is sent to the client.

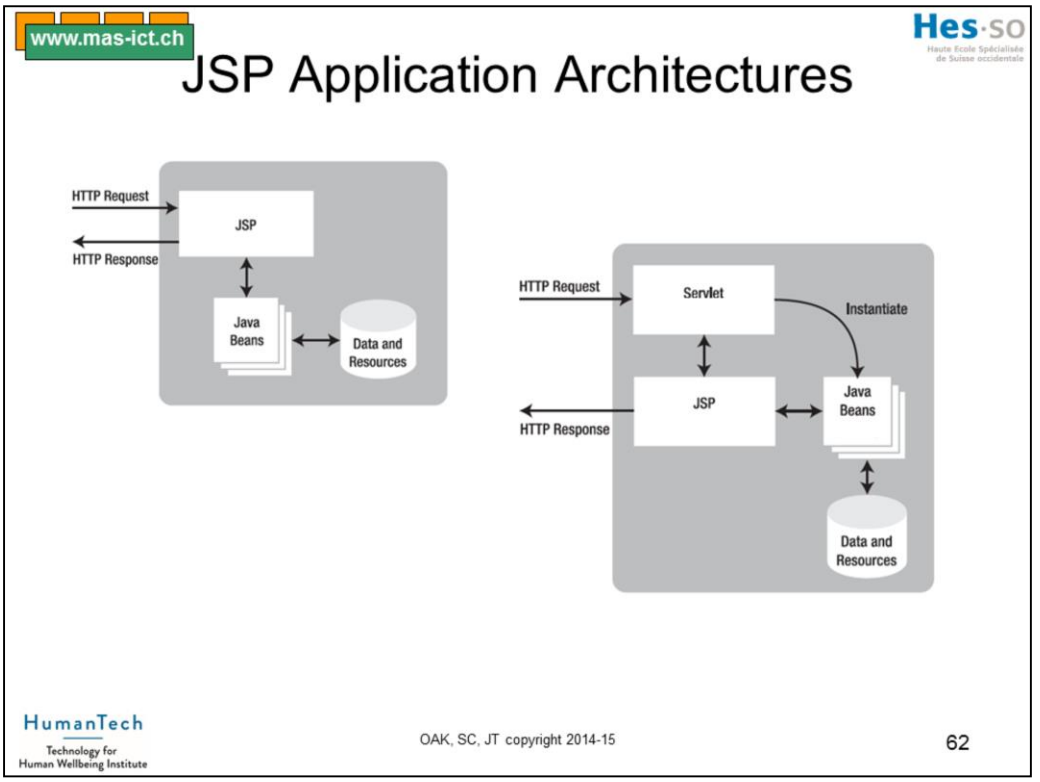
•End of life: The server stops sending requests to the JSP. After all current requests are finished processing, any instances of the class are released. This usually occurs when the server is being shut down, but can also occur at other times, such as when the server needs to conserve resources, when it detects an updated JSP source file, or when it needs to terminate the instance for other reasons. If you need code to execute and perform any cleanup actions, you can implement a method that will be called before the class instance is released.

Uses of JSP Constructs

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)

- Beans
- Custom tags

- Servlet/JSP (MVC), with beans and possibly custom tags

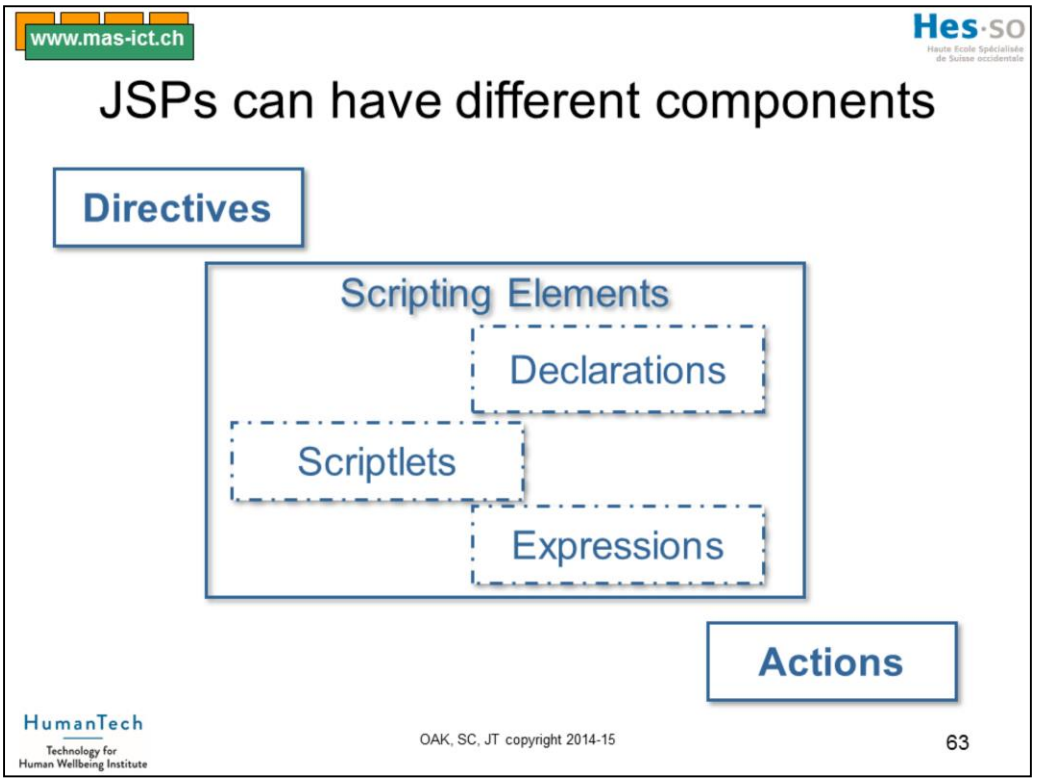


The key problem with mixing Java and HTML, as in “Hello World!,” is that the application logic and the way the information is presented in the browser are mixed. In general, the business application designers and the web page designers are different people with complementary and only partly overlapping skills. While application designers are experts in complex algorithms and databases, web designers focus on page composition and graphics. The architecture of your JSP-based applications should reflect this distinction.

The Model 1 Architecture: The first solution to this problem that developers found was to define the JSP Model 1 architecture, in which the application logic is implemented in Java classes (i.e., Java beans), which you can then use within JSP. Model 1 is acceptable for applications containing up to a few thousand lines of code, and especially for programmers, but the JSP pages still have to handle the HTTP requests, and this can cause headaches for the page designers.

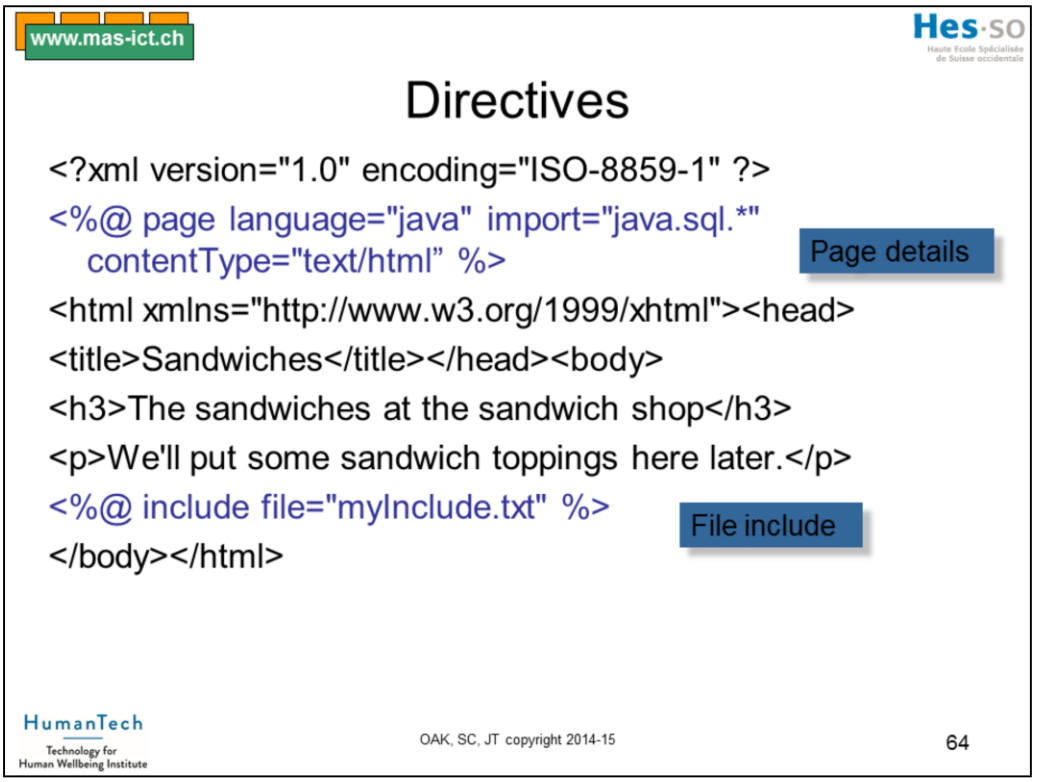
The Model 2 Architecture

A better solution, also suitable for larger applications, is to separate application logic and page presentation. This solution comes in the form of the JSP Model 2 architecture, also known as the model-view-controller (MVC) design pattern. With this model, a servlet processes the request, handles the application logic, and instantiates the Java beans. JSP obtains data from the beans and can format the response without having to know anything about what’s going on behind the scenes.



The JSP specification defines HTML-like or XML tags that enclose the code in the JSP. Those tags come in three categories:

- Directive elements
- Scripting elements
- Action elements



Directive Elements

Directive elements provide information to the JSP container about the page. Three directives are available: **page**, **include**, and **taglib**.

Page Directives

The page directive is used to specify page attributes. The page directive's JSP-style form is: <%@ page *attributes* %>

The page directive's XML-style form is:

<jsp:directive.page *attributes* />

Include Directives

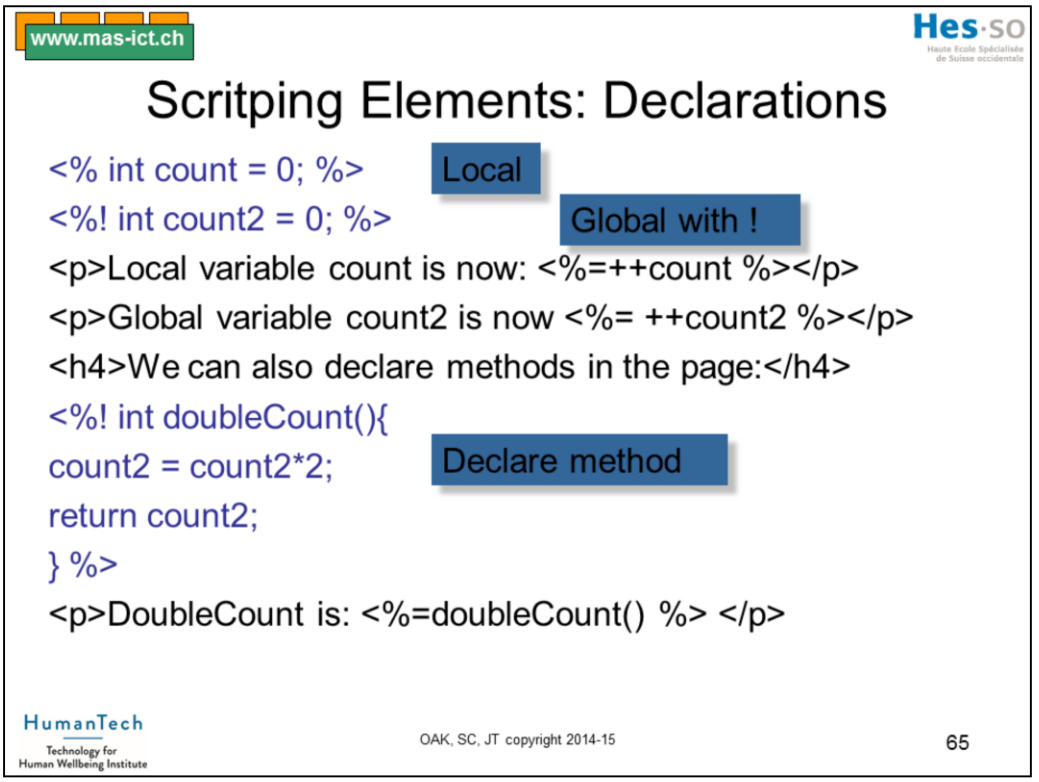
The include directive is used to include another page within the current page. The include directive's JSP-style form is:

<%@ include *attributes* %>

And its XML-style form is: <jsp:directive.include *attributes* />

This directive has a single attribute named file. The file attribute specifies the name of the file to be included at the current position in the file.

Directives affect pages, includes and a number of aspects related to Tags. Here (figure) the page directive at the top specifies the language, the imports, and content type. The file include specifies which file should be included into the body of the jsp BEFORE it is compiled.



Scripting Elements

The scripting elements are the elements in the page that include the Java code. There are three subforms of this element: declarations, scriptlets, and expressions.

Declarations

A declaration is used to **declare, and optionally define, a Java variable or a method**. It works just like any declaration within a Java source code file. The declaration element's JSP-style form is:

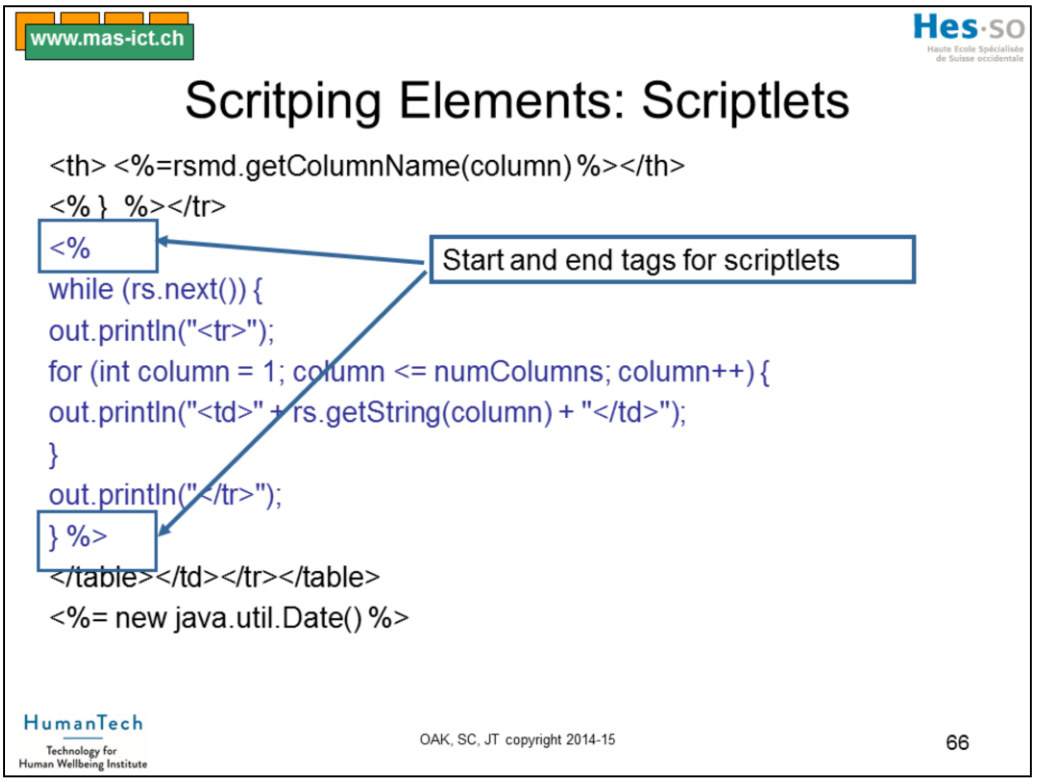
<%! *declaration* %>

And its XML-style form is:

<jsp:declaration>*declaration*</jsp:declaration>

The declaration appears only within the translated JSP page, but not in the output to the client.

Declarations of variables can be local, i.e., each time the page is requested it is set, or global, so that it is set once when the page is initialised by the container.



Scriptlets

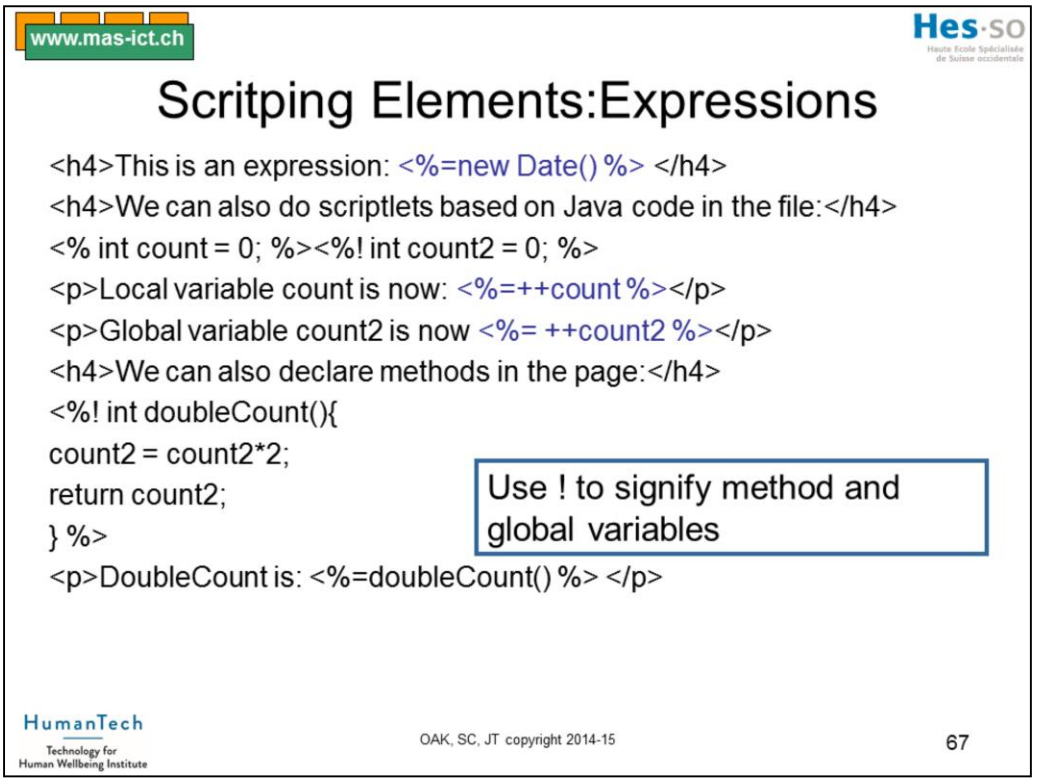
Scriptlets contain Java code statements. The code in the scriptlet appears in the translated JSP, but not in the output to the client. The scriptlet element's JSP-style form is:

<% *scriptlet code %>*

And its XML-style form is:

<jsp:scriptlet>*code fragment</jsp:scriptlet>*

A code fragment valid in the language used. Notice that it can be interspersed with HTML.



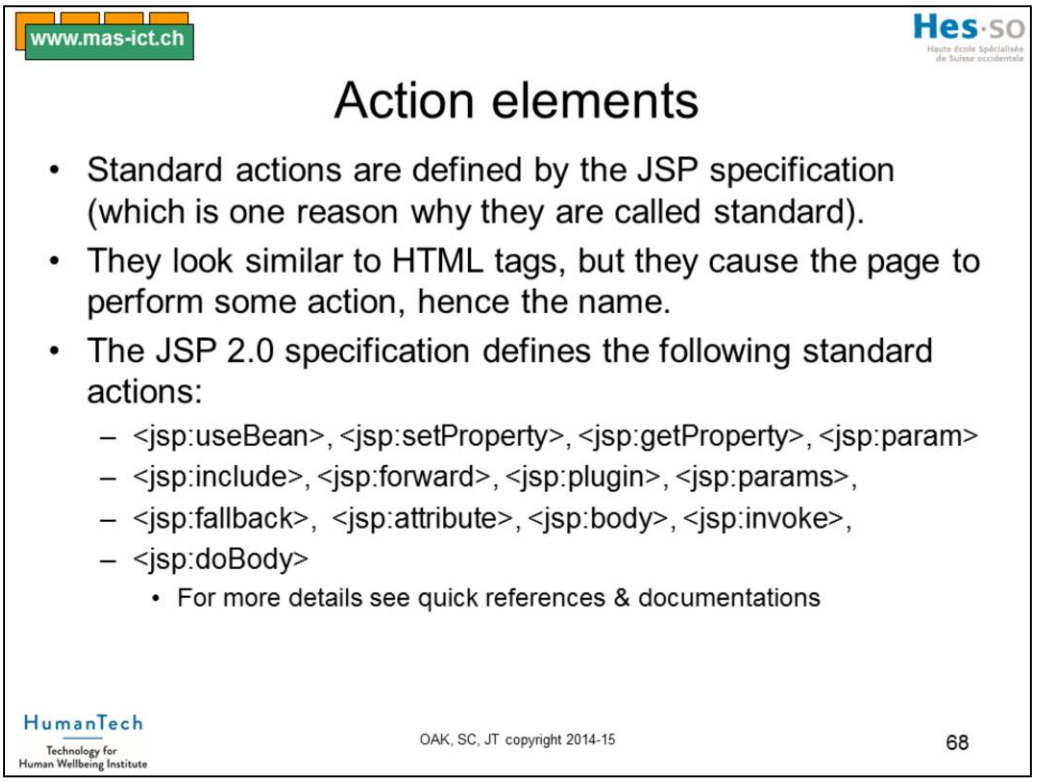
Expressions

Expressions are used to output the value of a Java expression to the client. The expression element's JSP-style form is: <%= *expression %>*

And its XML-style form is:

<jsp:expression>*expression*</jsp:expression>

Expressions must evaluate to a value which is then converted into a String for display in the page.



The purpose of JSP actions is to specify activities to be performed when a page is requested. Actions can operate on objects and have an effect on each response. They normally take this form:

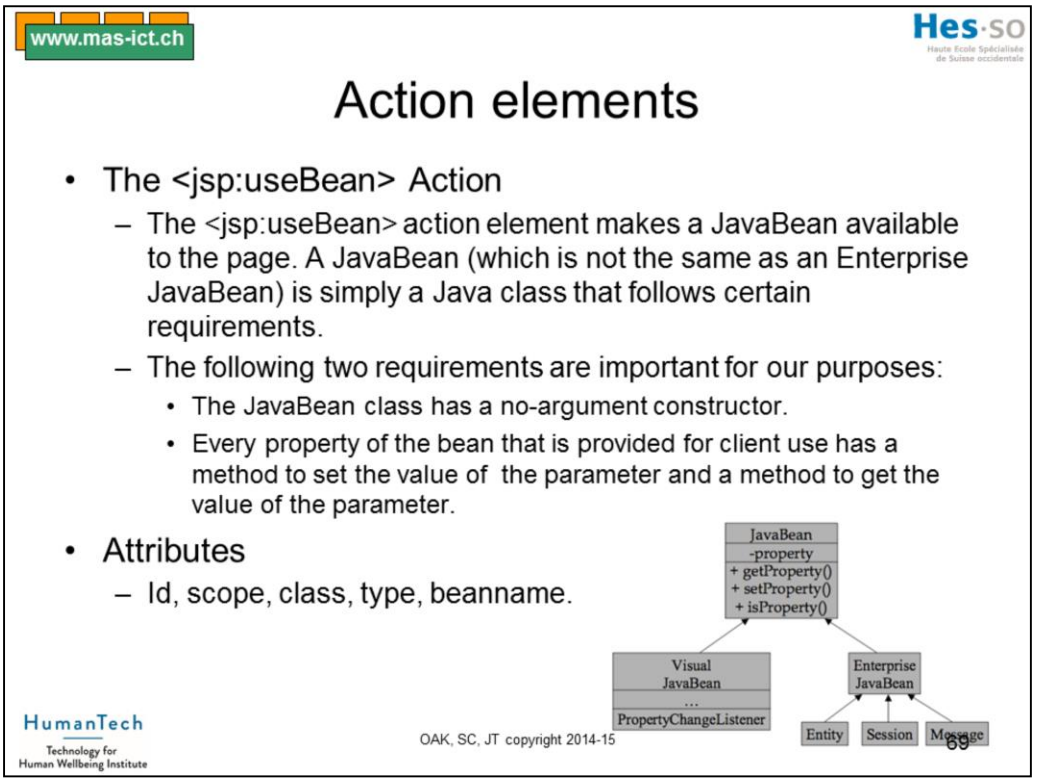
<jsp:action-name action-attribute-list/>

action-attribute-list is a sequence of one or more *attribute-name="attribute-value"* pairs. However, actions can also have a body, like in the following example:

<jsp:action-name attribute-list>

<jsp:subaction-name subaction-attribute-list/>

</jsp:action-name>



Attributes of the useBean Tag:

• id : The name used to access the bean in the rest of the page. It must be unique. It is essentially the variable name that references the bean instance. When a <jsp:useBean> action is used in a scriptless page, or in the body of an action marked as scriptless, no Java scripting variables are created; instead, an Expression Language variable is created.

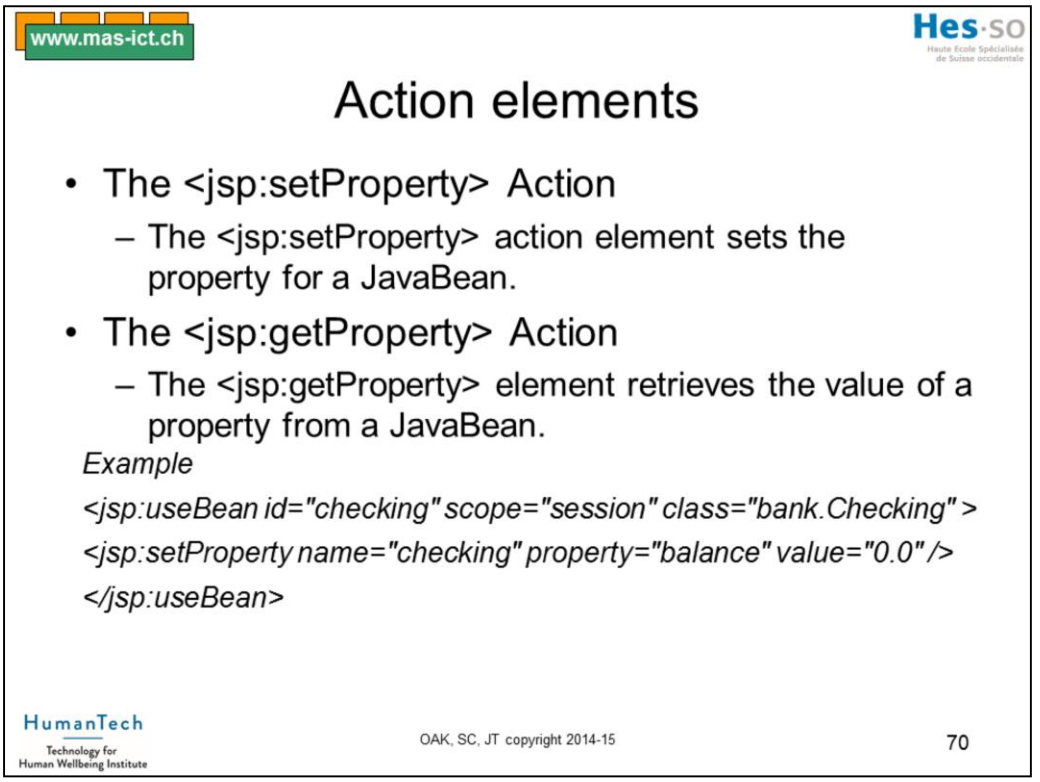
• scope The scope of the bean. Valid values are page, request, session, or application. The default is page.

• class The fully qualified class name of the bean class.

• beanname The name of a bean, as expected by the instantiate() method of the java.beans.Beans class.

• type The type to be used for the variable that references the bean. This follows Java rules, so it can be the class of the bean, any parent class of the bean, or any interface implemented by the bean or by a parent class.

The <jsp:useBean> element causes the container to try to find an existing instance of the object in the specified scope and with the specified id. If no object with the specified id is found in that scope, and a class or bean name is specified, the container will try to create a new instance of the object.



Attributes of the setProperty Tag

Attribute Description

•name: The id of the bean as defined by the useBean action.

•Property: The name of the property whose value will be set. The property attribute can explicitly name a property of the bean; in which case, the set*XXX() method* for the property will be called. The value can also be "*"; in which case, the JSP will read all the parameters that were sent by the browser with the client's request and set the properties in the bean that have the same names as the

parameters in the request.

•param: The parameter name in the browser request whose value will be used to set the property. Allows the JSP to match properties and parameters with different names.

•value: The value to assign to the property.

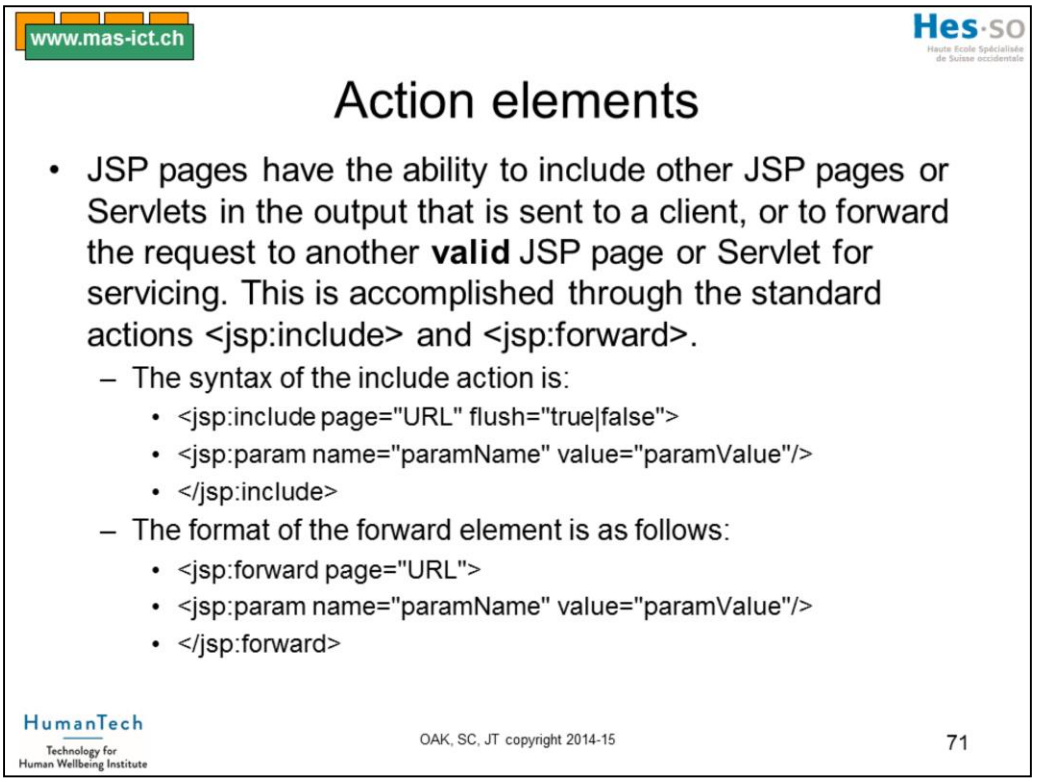
Attributes of the getProperty Tag

Attribute Description

•name: The id of the bean

•property: The name of the property to get

-More infos and examples at http://java.sun.com/products/jsp/tags/11/syntaxref11.fm14.html



include Action vs include Directive

Recall that an include directive can be used in either of the following two formats, anywhere within the JSP:

<%@ include file="/WEB-INF/footer.jspf">

<jsp:directive.include file="/WEB-INF/footer.jspf"/>

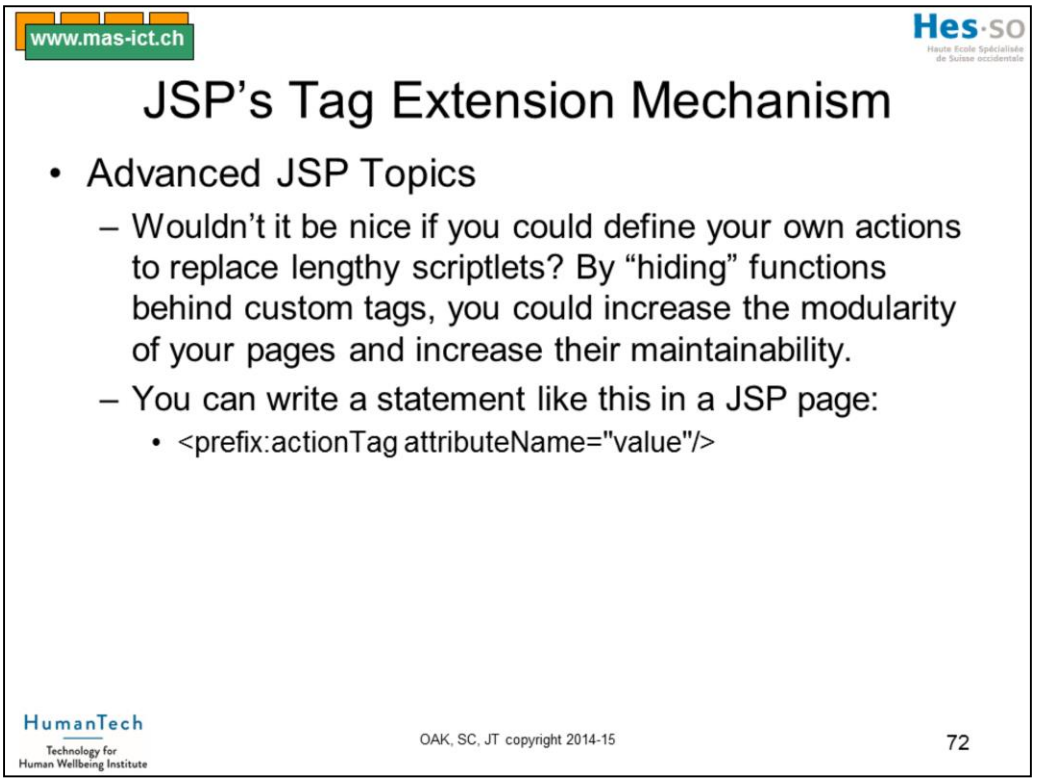
When the JSP container translates the page, this directive causes the indicated file to be included in that place in the page and become part of the Java source file that is compiled into the JSP page implementation class; that is, it is included at translation time. Using the include directive, the included file does not need to be a complete and valid JSP page.

With the include standard action, the JSP file stops processing the current request and passes the request to the included file. The included file passes its output to the response. Then control of the response returns to the calling JSP, which finishes processing the response. The output of the included page or Servlet is included at request time. Components that are included via the include action must be valid JSP pages or Servlets.

For the include element, the page attribute is required, and its value is the URL of the page whose output is included in the response. If the JSP needs to pass parameters to the included file, it does so with the <jsp:param> element.

forward Action

With the forward action, the current page stops processing the request and forwards the request to another web component. This other component completes the response. Execution never returns to the calling page. Unlike the include action, which can occur at any time during a response, the forward action must occur prior to writing any output to the OutputStream. In other words, the forward action must occur prior to any HTML template data in the JSP, and prior to any scriptlets or expressions that write data to the OutputStream.



you need to follow these steps:

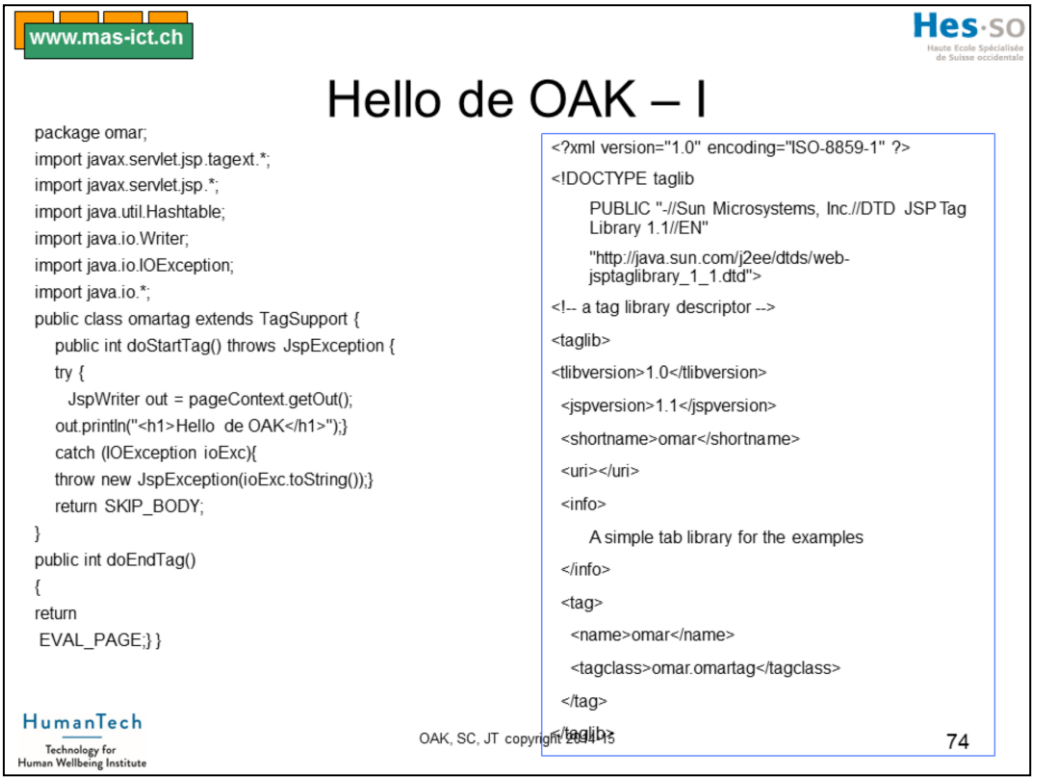
1. Define Java classes that provide the functionality of the actions you're defining, including the definition of their attributes (e.g., attributeName). These classes are called tag handlers.

2. Provide a formalized description of your action elements, so that Tomcat knows how to handle them. For example, you need to specify which actions can have a body and which attributes can be omitted. Such a description is called a tag library descriptor (TLD).

3. In the JSP pages, tell to the server that the pages need your tag library and specify the prefix that you want to identify those custom tags with.

JSP's Tag Extension Mechanism

- 4 Steps
  - Creation of the file describing the taglib (TLD)
  - Creation of the Java class
  - Taglib definition inside the file *web.xml*
  - Taglib declaration and utilization inside the JSP file



Mandatory Subelements of <taglib>

•tlib-version: The version number of the library

•short-name: A simple default name, which may be used as the preferred prefix value in taglib directives

•Tag: Information about a tag handler

Subelements of <tag>

•name: The name of the tag handler (mandatory).

•tag-class: The fully qualified class name of the tag handler class (mandatory).

•body-content Whether the body of the tag can have content.

•valid values are tagdependent, scriptless, or empty. The default is scriptless. If the value is empty, the tag is not allowed to have a body.

•variable: Defines the scripting variables created by this tag handler and made available to the rest of the page. This element must contain one of two subelements: name-given or name-from-attribute. If name-given is used, the value of this element defines the name that other JSP elements can use to access the created scripting variable. If name-from-attribute is used, the value of the attribute with the name given by this element defines the name of thescripting variable.

•Attribute: Defines attributes for the tag. This element has three subelements: name, required, and rtexprvalue. The value of the name element will be the name of the attribute. The element named required is optional, and must be one of true, false, yes, or no. This indicates whether the attribute is required or optional. The default value is false (meaning the attribute is optional). The rtexprvalue element is optional, and must be one of true, false, yes, or no. The default value is false, which means that the attribute can be set only by using a static value known at compile time. If the element contains true or yes, the attribute can be set using a runtime expression.

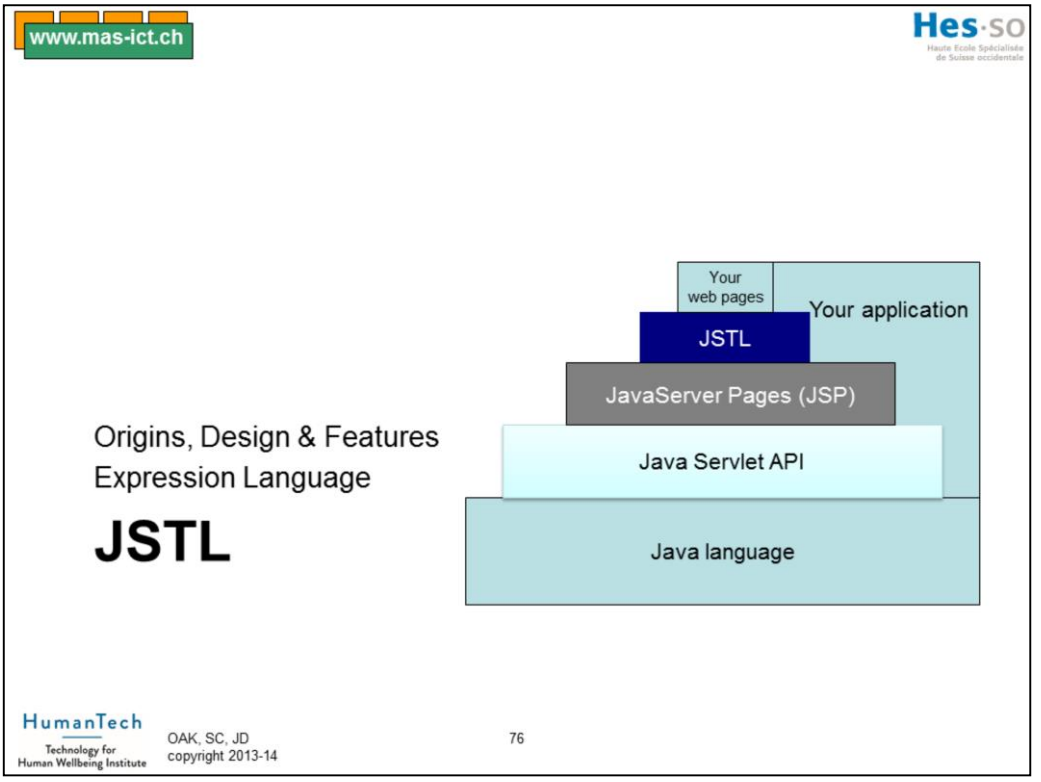
Hello de OAK – II

- In web.xml you have to add:

```
<taglib>
  <taglib-uri>
    http://java.apache.org/tomcat/omar-taglib
  </taglib-uri>
  <taglib-location>
    /WEB-INF/jsp/omar.tld
  </taglib-location>
</taglib>
```

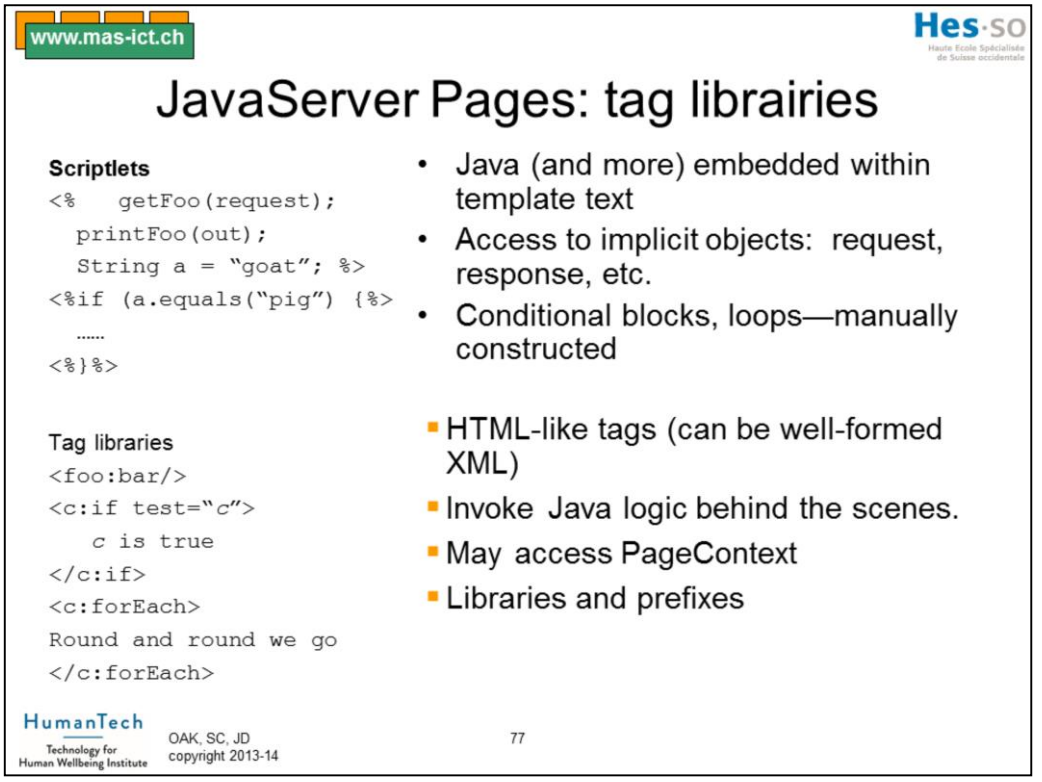
- My file maytag.jsp:

```
<%@ taglib
uri=http://java.apache.org/tomcat/omar-taglib
prefix="mytag" %>
<HTML><HEAD><TITLE>tag</TITLE>
  </HEAD>
  <BODY>
  <mytag:omar/>
  </BODY>
  </HTML>
```

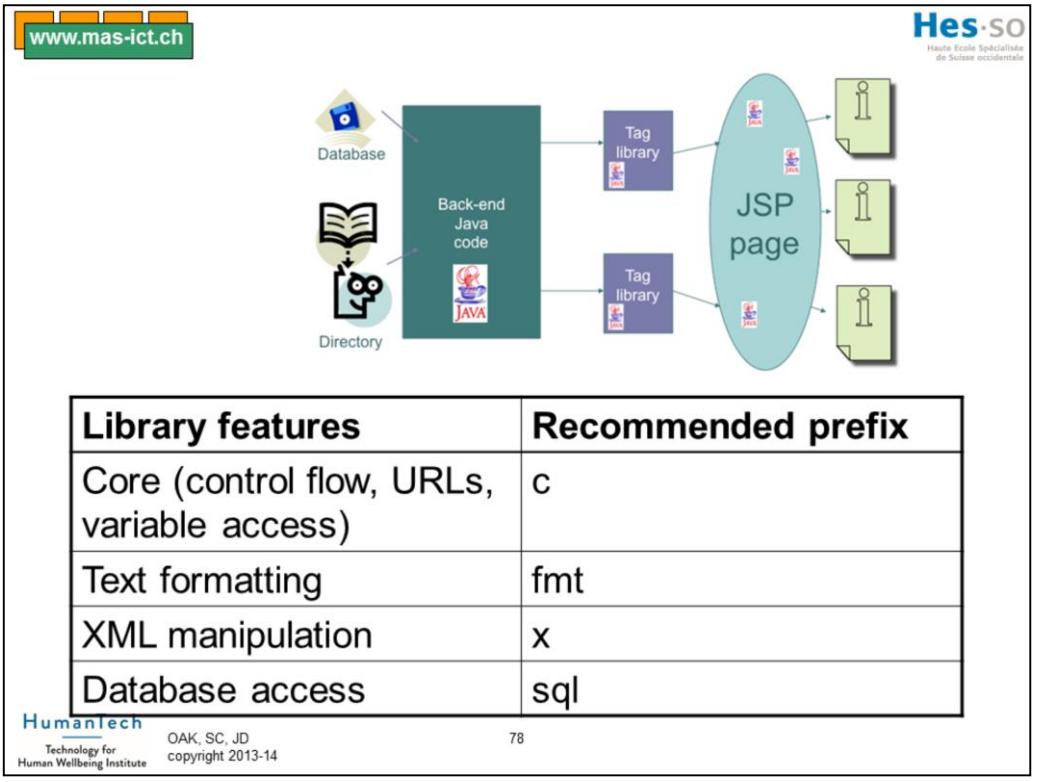


JSP Standard Tag Library (JSTL). What are the benefits of the newer standards and technologies?

- Easier development
- Easier debugging
- Easier maintenance
- Easier reuse



The JSTL is a collection of custom tags that provide important utilities.



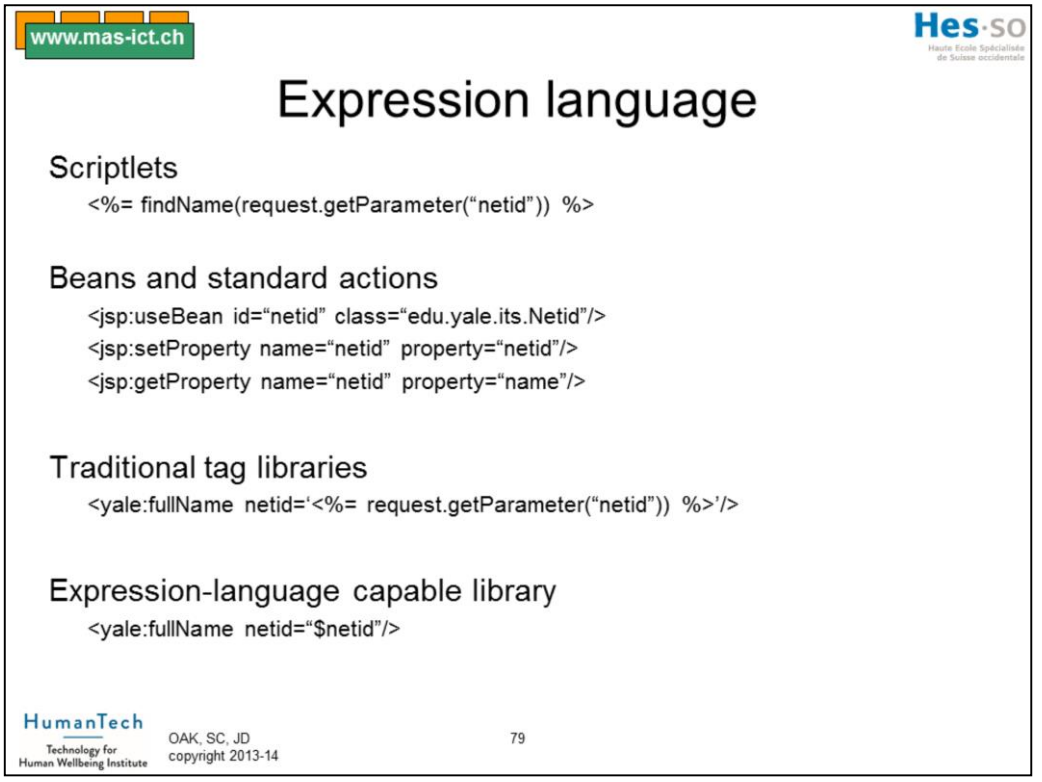
Advantages of tag libraries:

-Abstraction, abstraction, abstraction.

-Abstraction: 1) Separation of logic from presentation and content; 2) Simple, familiar interface for page authors; 3) Implicit (versus explicit) arguments.

Features:

1) Conditional logic
2) Iteration
3) Text retrieval (URL, RequestDispatcher)
4) Text formatting and parsing (i18n)
5) XML manipulation (XPath, XSLT)
6) Database access
7) **Last but certainly not least: Expression language**



Expression Language

The JSTL expression language (EL) uses ${ .. } to indicate an expression. The expression language includes standard operators.

The param object is predefined in EL to provide data submitted with an HTTP request:

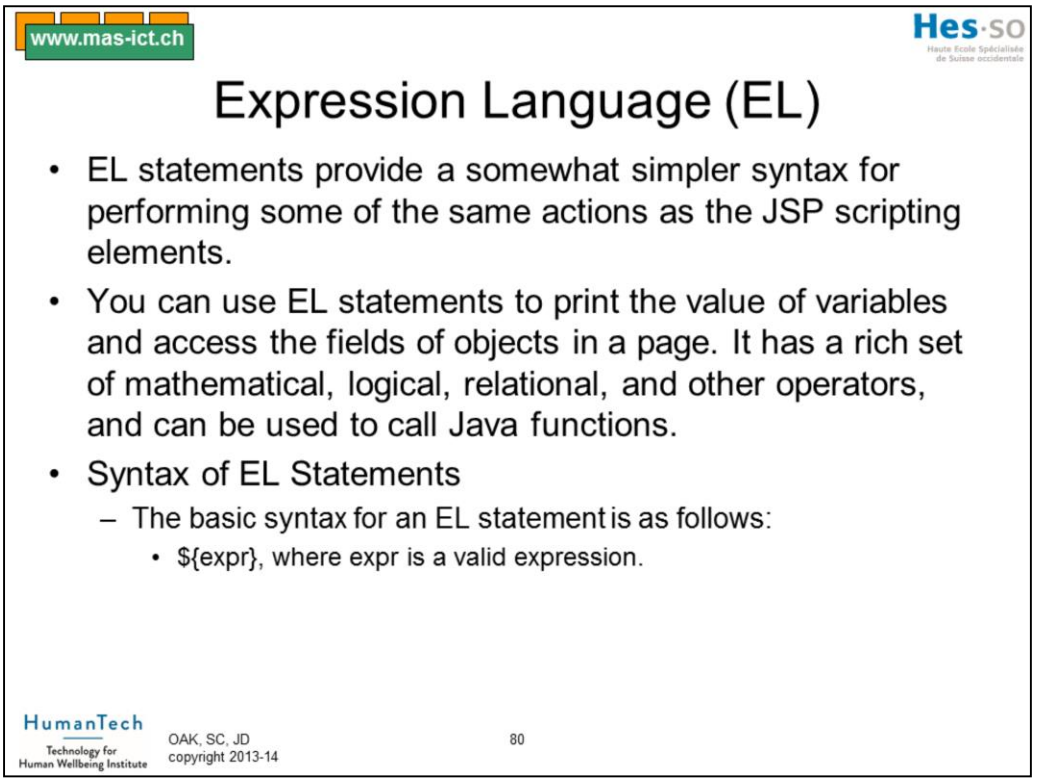
> ${param.name} gets the value associated with name
>
> ${param['fancy name']} gets the value if the name is not a proper identifier

The JSP Expression Language (EL): Key syntax

-Expressions appear between ${ and }. Note that ${ and } may contain whole expressions, not just variable names, as in the Bourne shell (and its dozen derivatives.) E.g., ${myExpression + 2}

-Expressions' default targets are *scoped attributes* (page, request, session, application): ${user} ≡ pageContext.findAttribute("user")

-The . and [] operators refer to JavaBean-style properties and Map elements: ${user.getAddress} can resolve to **((User) pageContext.getAttribute("user")).getAddress()**



Expression Language (EL):

The Java Unified Expression Language is a special purpose programming language mostly used in Java web applications for embedding expressions into web pages. The expression language started out as part of the JavaServer Pages Standard Tag Library (JSTL) and was originally called SPEL (Simplest Possible Expression Language). It offered a simple way to access data objects. Over the years, the expression language has evolved to include more advanced functionality and it was included in the JSP 2.0 specification, because of the popularity and success in the community.

EL is an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans).

Enabling EL Statements

EL statements are enabled or disabled through two different techniques:

• You can enable EL statements for particular pages through the page directive.

• You can enable EL statements for whole sets of pages using the deployment descriptor.

The page directive for enabling or disabling EL statements looks like this: <%@ page isELIgnored="true|false" %>

You can also specify EL configuration information in the deployment descriptor. EL statements are evaluated based on the value of the <el-ignored> element of the <jsp-property-group> element:

<jsp-config>

<jsp-property-group>

<url-pattern>*.jsp</url-pattern>

<el-ignored>true</el-ignored>

```
</jsp-property-group>
</jsp-config>
```

Expression Language

- Inspirations: JavaScript, XPath
  - But it's much simpler than even these basic expression languages.
- Expression
  - Literals
  - Operators
  - Variables
    - The web container evaluates a variable that appears in an expression by looking up its value
    - For example, when evaluating the expression ${product}, the container will look for the name "product" in the page, request, session, and application scopes and will return its value.

EL Expression	Result
${1 <= (1/2)}	false
${5.0 > 3}	true
${100.0 == 100}	true
${'a' < 'b'}	true
${'fluke' gt 'flute'}	false
${1.5E2 + 1.5}	151.5
${1 div 2}	0.5
${12 mod 5}	2

EL Operators

Arithmetic: +, -, *, /, %, div, mod

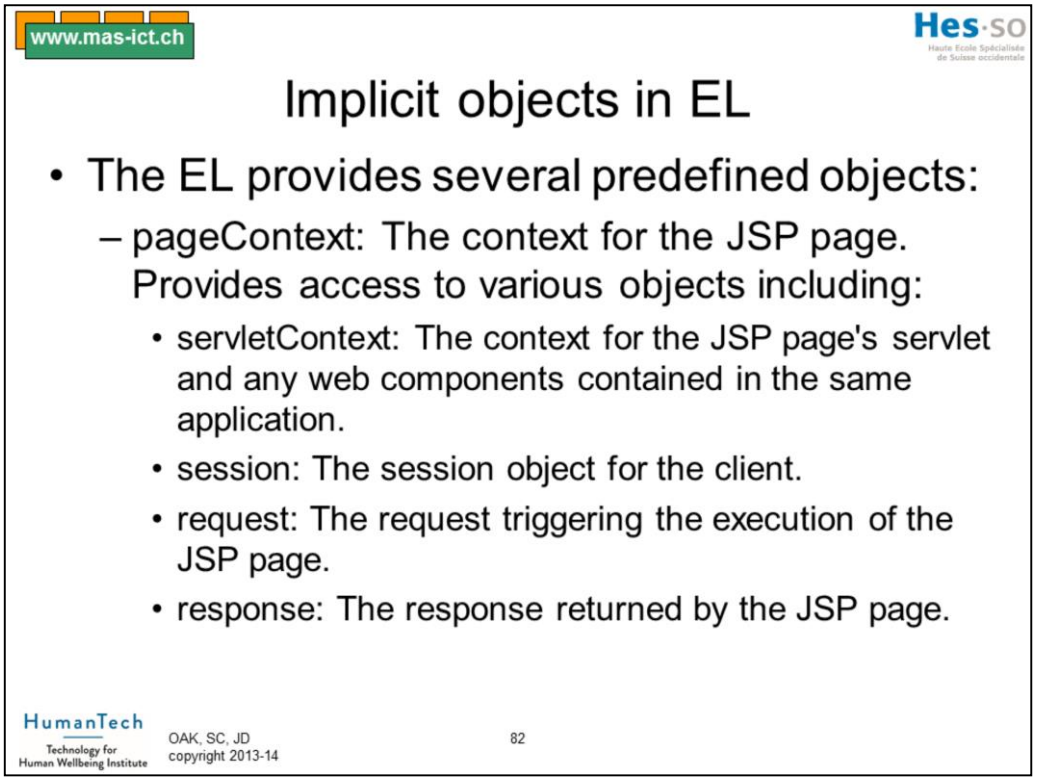
Logical: &&, ||, !, and, or, not

Relational: ==, !=, <, >, <=, >=, eq, ne, lt, gt, le, ge

Conditional: ? :

Empty: check whether a value is null or empty

Other: [], ., ()



In addition, several implicit objects are available that allow easy access to the following objects:

param: Maps a request parameter name to a single value

paramValues: Maps a request parameter name to an array of values

header: Maps a request header name to a single value

headerValues: Maps a request header name to an array of values

cookie: Maps a cookie name to a single cookie

initParam: Maps a context initialization parameter name to a single value

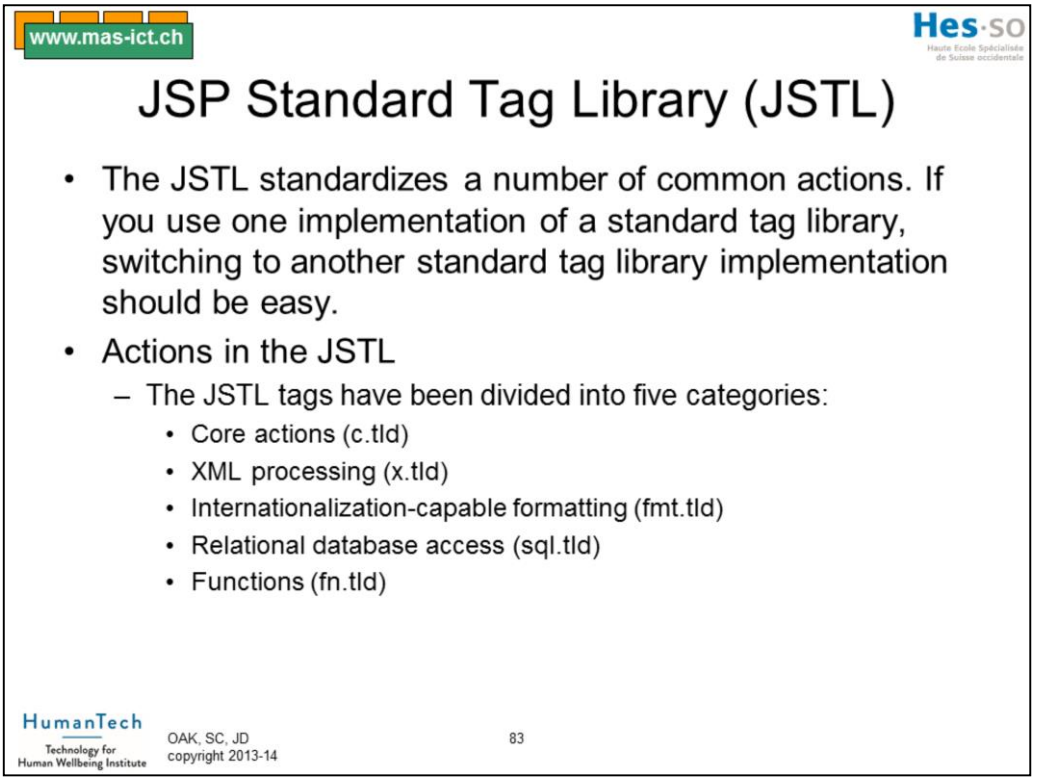
Finally, there are objects that allow access to the various scoped variables

pageScope: Maps page-scoped variable names to their values

requestScope: Maps request-scoped variable names to their values

sessionScope: Maps session-scoped variable names to their values

applicationScope: Maps application-scoped variable names to their values



One place to learn about JSTL is to start with The Java EE 5 Tutorial. Chapter 7 provides an excellent introduction to JSTL (http://download.oracle.com/javaee/5/tutorial/doc/bnakc.html)

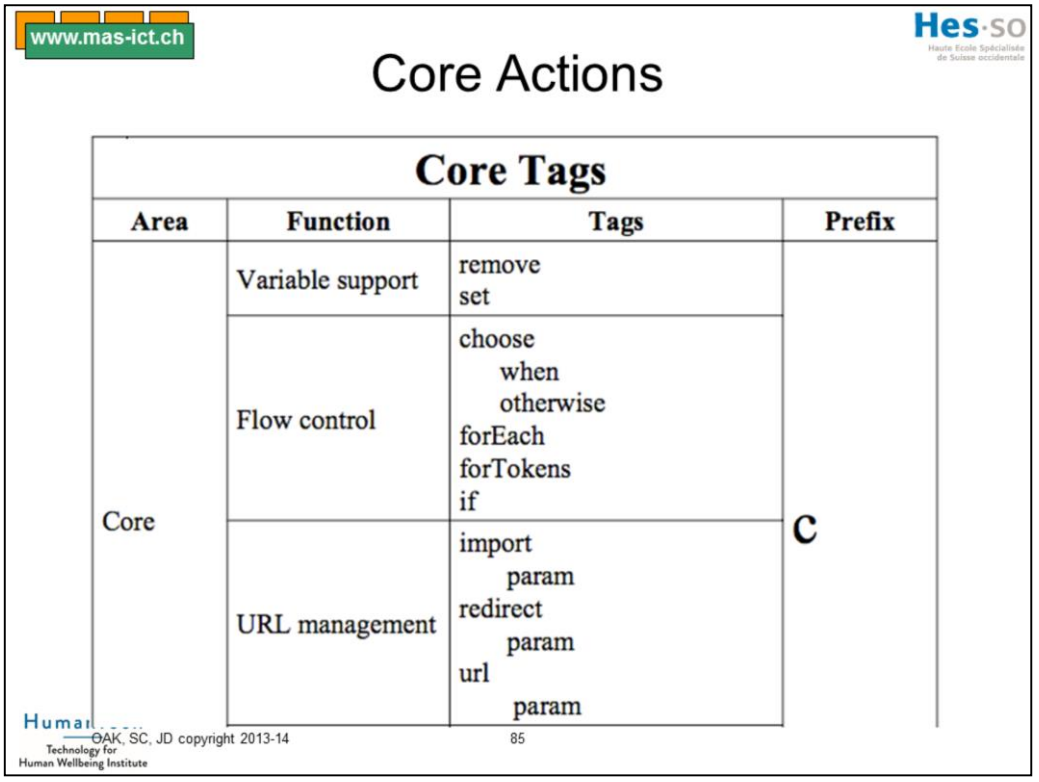
JSP Standard Tag Library (JSTL)

before

```
<!--Set global information for the page
    - ->
<%@ page language="java" %>
<!--Declare the variable -->
<%!int count = 0; %>
 <!--Scriptlet-Java code -->
<%
for (int i = 0; i < 8; i++)
{
count = count+1; %>
<BR> The counter value is: <%=
    count %>
<% } %>
```

after

```
<%@ tagliburi="/WEB-INF/c.tld"
    prefix="c" %>
<c:set var="count" value="0"/>
<c:forEachvar="i" begin="0" end="7">
<c:setvar="count" value="${count + 1}"/>
    JSTL Output: The counter value is:
<c:out value="${count}"/><br/>
</c:forEach>
```



General-Purpose Actions in the JSTL Core Category

<c:out value="" default=""> : Sends the value to the response stream. You can specify an optional default value so that if the value attribute is set with an EL expression, and the expression is null, the default value will be output.

<c:set var="" value="">: Sets the JSP-scoped variable identified by var to the given value.

<c:set target="" property="" value="">: Sets the property of the given JavaBean or Map object to the given value.

<c:remove var="" scope="">: Removes the object identified by var from the given scope. The scope attribute is optional. If the scope is not given, each scope will be searched in the order page, request, session, application, until the object is found or all scopes are searched. If scope is given, the object is removed only if it is in the given scope. If the object is not found, an exception will be thrown.

<c:catch var="">: Encloses a block of code that might throw an exception. If the exception occurs, the block terminates but the exception is not propagated. The thrown exception can be referenced by the variable named by var.

Conditional Actions in the JSTL Core Category

<c:if test="" var=""> Used like a standard Java if block. The var attribute is optional; if present, the result of the test is assigned to the variable identified by var. If the test expression evaluates to true, the tag isevaluated; if false, it is not.

<c:choose>, <c:when test="">, <c:otherwise>: The analog to a Java if...elseif...else block. The <c:choose> action starts and ends the block. The test in each <c:when test=""> tag is evaluated;

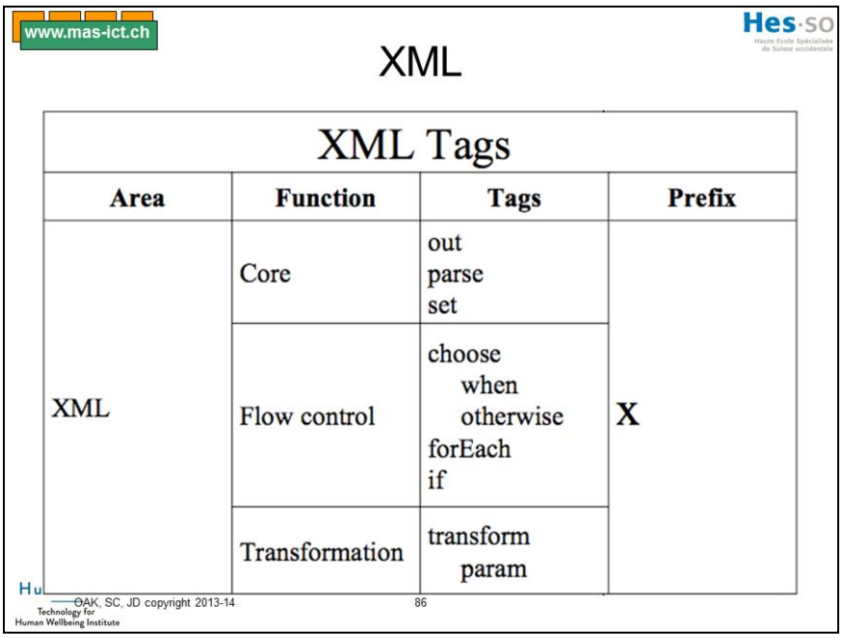
the first test that evaluates to true causes that tag to be evaluated. If no <c:when> action evaluates to true, the <c:otherwise> tag is evaluated.

Iterator Actions in the JSTL Core Category

<c:forEach var="" items=""> Iterates over each item in the collection identified by items. Each item can be referenced by var. When items is a Map, the value of the item is referenced by var.value.

<c:forEach var="" begin="" end="" step=""> The tag for a for loop. The step attribute is optional.

<c:forTokens items="" delims=""> Iterates over the tokens in the items string.



JSTL contains several tags for working with XML and XSLT. JSTL has support for navigating XML with XPath and to do XSLT processing

<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %> is used to specify that the XML part is to be used

JSTL XML – parsing

A XML structure can be parsed into a XML document that can be navigated

<x:parse var="a"> <a> <c> foo </c> <d> bar </d> </x:parse>

<x:out select="$a/a/d"/>

The XML is parsed into variable a. The value of the element d is selected

JSTL XSL Transformation

<c:set var="xml"> <a>header! </c:set>

<c:set var="xsl">

<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="text()"> <h1>

<xsl:value-of select="."/></h1>

</xsl:template>

</xsl:stylesheet>

</c:set>

<x:transform xml="${xml}" xslt="${xsl}"/>

1. Combine the Core import with XSLT to read the XSL file from disk
2. Get XML output from a Custom tag (or Java Bean)
3. Do XSL Transformation

<c:set var="booklist_xslt">

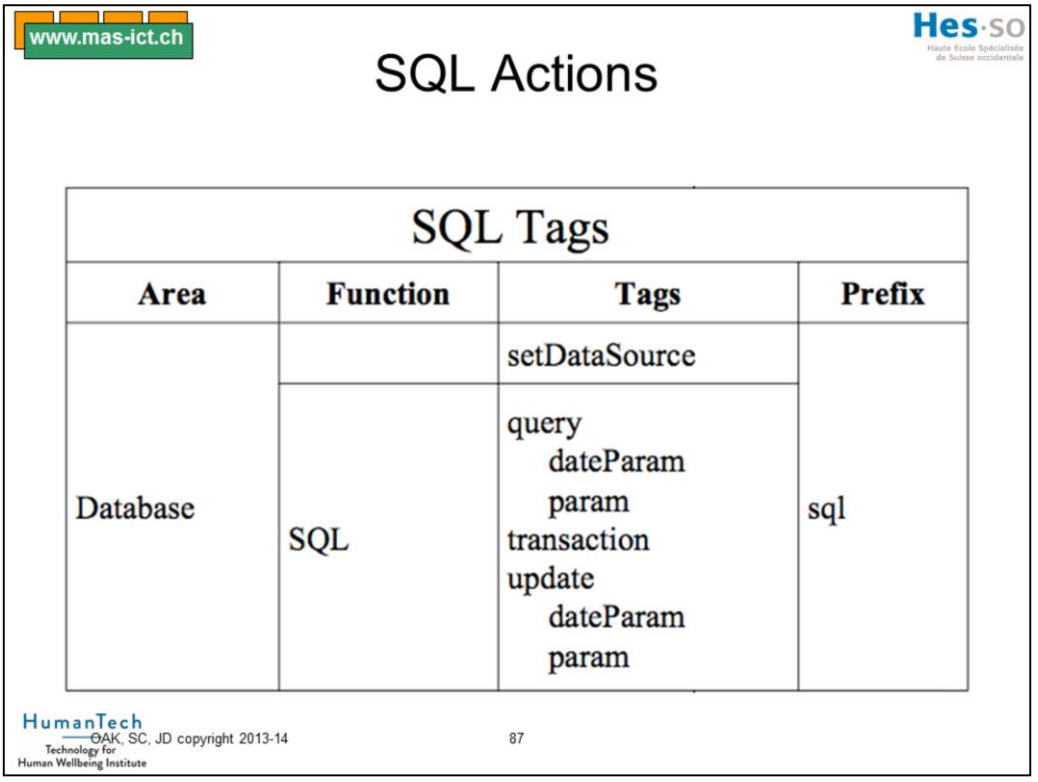
<c:import url="booklist_xslt.xsl"/>

</c:set>

<x:transform xslt="${booklist_xslt}">

<jsp:getProperty name="bookList" property="xml"/>

</x:transform>



SQL Actions

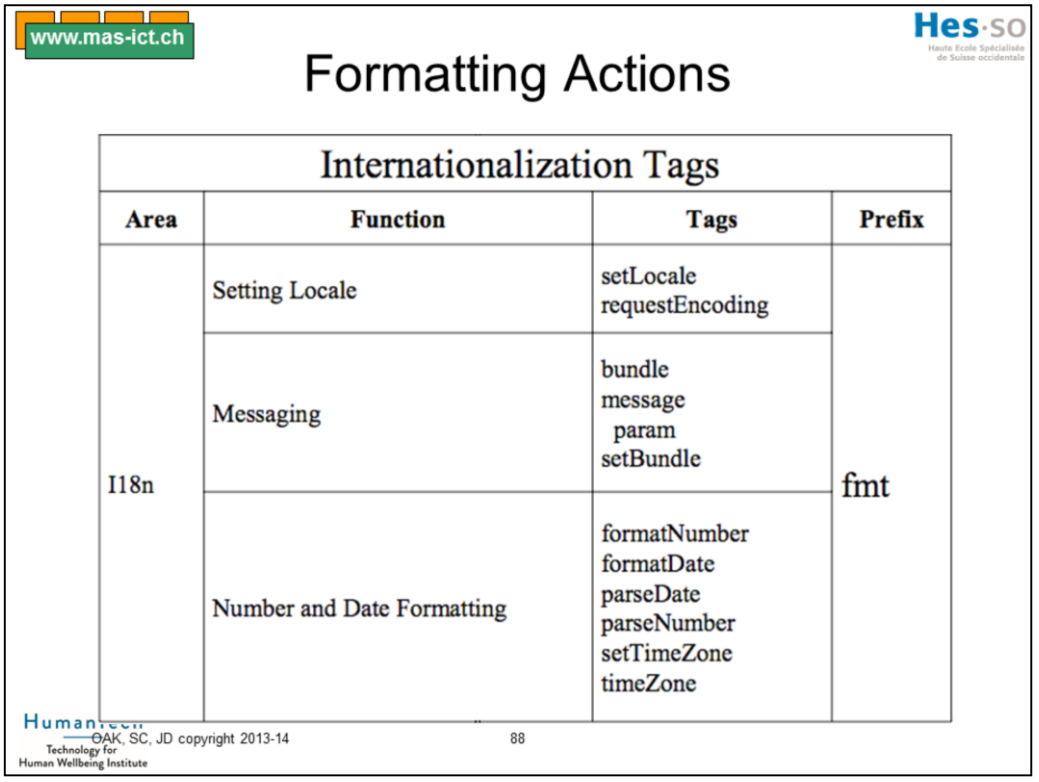
The JSTL SQL actions allow page authors to perform database queries, access query results, and perform inserts, updates, and deletes. One of the many SQL actions is <sql:query>:

<sql:query var="" dataSource=""> SQL Command </sql:query>

This action queries the database given by the dataSource attribute. The query that is performed is given in the body of the tag. The results of the query can be accessed by var.rows. You can use the <c:forEach> tag to iterate over the collection of rows. The dataSource attribute can identify the database in two ways: use the JDBC URL to access the database or use the Java Naming and Directory Interface (JNDI) data source name to look up the database.

Example:

<c:set var="bid" value="${param.Add}"/>
<sql:query var="books" >
select * from PUBLIC.books where id = ?
<sql:param value="${bid}" />
</sql:query>



Formatting actions are part of the internationalization, or I18N, library. As you might guess, they provide support for formatting output. Among the actions for setting locales and time zones, are actions for formatting numbers.

The following formatting action is for dates:

<fmt:formatDate value="date" [type="{time|date|both}"]

[dateStyle="{default|short|medium|long|full}"]

[timeStyle="{default|short|medium|long|full}"]

[pattern="customPattern"] [timeZone="timeZone"] [var="varName"]

[scope="{page|request|session|application}"]/>

Functions

Area	Function	Tags	Prefix
Functions	Collection length	`length`	`fn`

	String manipulation	`toUpperCase, toLowerCase`
		`substring, substringAfter, substringBefore`
		`trim`
		`replace`
		`indexOf, startsWith, endsWith, contains, containsIgnoreCase`
		`split, join`
		`escapeXml`	

- toUpperCase, toLowerCase: Changes the capitalization of a string
- substring, substringBefore, substringAfter: Gets a subset of a string
- trim: Trims whitespace from a string
- replace: Replaces characters in a string
- indexOf, startsWith, endsWith, contains, containsIgnoreCase: Checks whether a string contains another string
- split: Splits a string into an array
- join: Joins a collection into a string
- escapeXml: Escapes XML characters in a string

Conclusion

- To be done with class