UNIVERSITÀ DEGLI STUDI DI PADOVA

NOTES OF

Web Applications

SUBTITLE

(Version 22/05/2021)

Edited by:
Stefano Ivancich

CONTENT

1.Introduction to Webapps

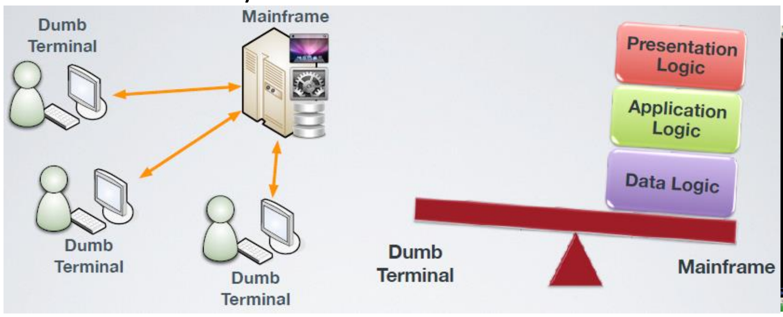
Distributed Applications

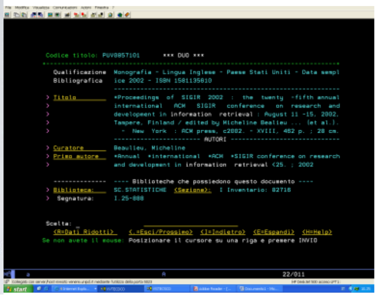
Application Layers:

- **Presentation/Interface/User Logic:** manages the interaction with the user, defines the format and visualization of information, accepts and validates user input.
- **Application/Business Logic:** defines and controls the flow of operations in, the application defines the basic operations on the data and their constraints, often called business rules.
- **Data Logic:** manages the persistent storage of the data, searches and retrieves data, ensures consistency of the data.

Balancing of the Application Load

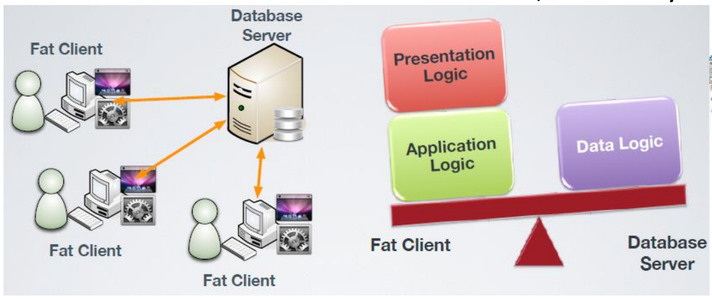
- **Single-tier architectures**: Terminal/Mainframe. (Uni clusters, Banks…)
    - **Pros**: easy to implement, no management of the terminals
    - **Cons**: computational load fully on the mainframe (single point-of-failure), scalability.

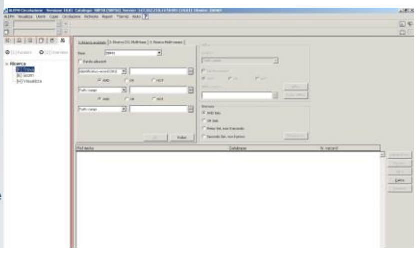




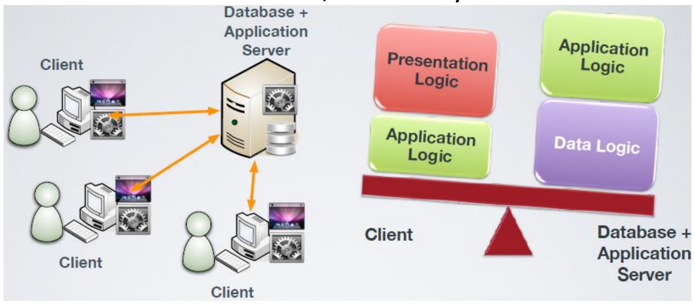
```
Codice titolo: PUV0857101      *** DUO ***
------------------------------------------------------------------------
  Qualificazione  Monografia - Lingua Inglese - Paese Stati Uniti - Data sempl
  Bibliografica   ice 2002 - ISBN 1581135610
                  ------------------------------------------------------
> Titolo          *Proceedings of SIGIR 2002 : the twenty -fifth annual
>                 international ACM SIGIR conference on research and
>                 development in information retrieval : August 11 -15, 2002,
>                 Tampere, Finland / edited by Micheline Beaulieu ... [et al.].
>                   - New York : ACM press, c2002. - XVIII, 462 p. ; 28 cm.
                  ---------------------- AUTORI ----------------------------
> Curatore        Beaulieu, Micheline
> Primo autore    *Annual *international *ACM *SIGIR conference on research
>                 and development in information retrieval <25. ; 2002

  --------------  ---- Biblioteche che possiedono questo documento ----
> Biblioteca:     SC.STATISTICHE <Sezione>: I Inventario: 82716
>  Segnatura:     I.25-888


Scelta: _
  <R=Dati Ridotti>  <.=Esci/Prossimo> <I=Indietro> <E=Espandi> <H=Help>
Se non avete il mouse: Posizionare il cursore su una riga e premere INVIO
```

- **Two-tiers architectures:**
    - Fat Client/Server:
        - **Pros**: easy to implement, possibility to balance load
        - **Cons**: client maintenance, scalability

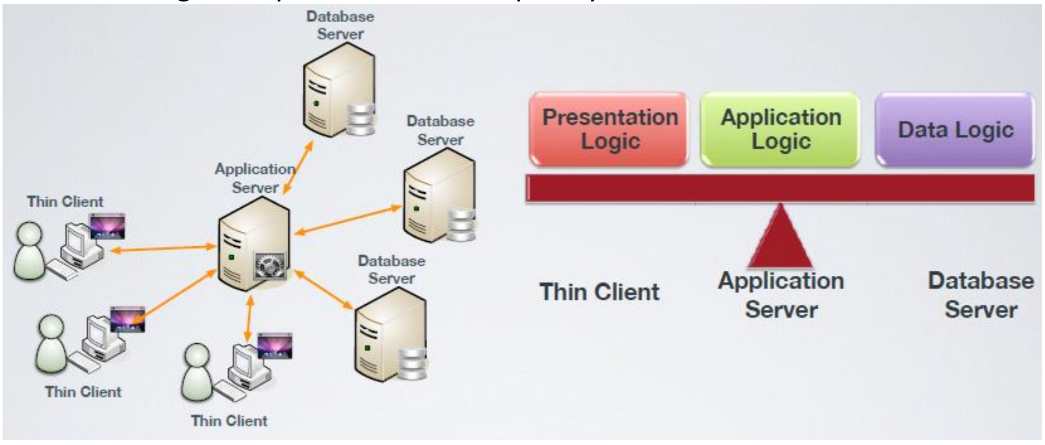




- **Client/Fat Server**
    - **Pros**: easy to implement, possibility to balance load
    - **Cons**: client maintenance, scalability

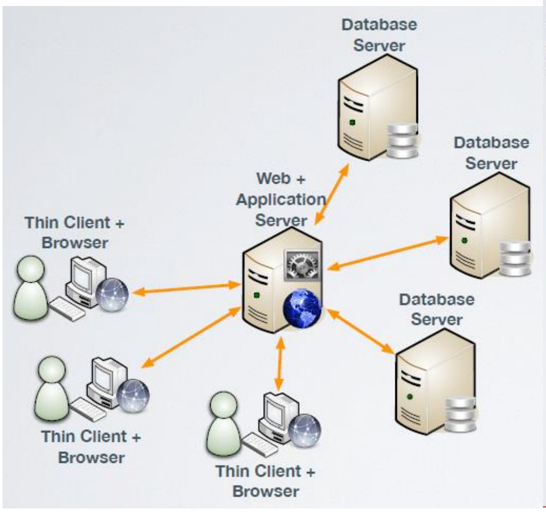


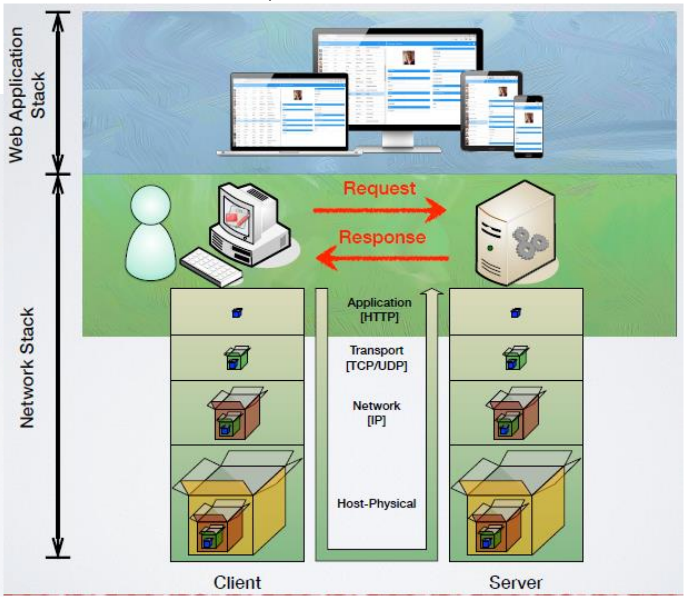
- **Three-tiers architectures**: Client/Middleware/Server
    - **Pros**: easy client maintenance, possibility to balance load, high scalability.
    - **Cons**: higher implementation complexity



Web Applications:

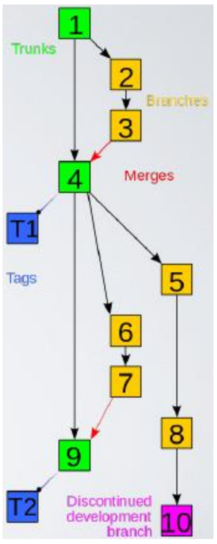
- are an instance of a **three-tiers architecture**.
- are based on standard and ubiquitous technologies (server-side), typically already part of an organization IT infrastructure.
- Do not require any management of the clients (browser)
- Clients are ubiquitous and allow access from any kind of devices (desktop and mobile)
- Users are already comfortable with the basic interaction patterns and interface elements.





2. Git

Introduction to Git

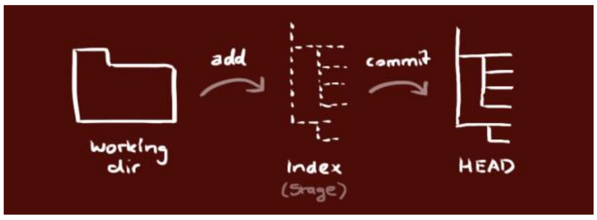


- A version control system manages the versions, called **revisions**, of files and directories.
- Manages conflicts, e.g. when the same file is edited concurrently, and their resolution (**merge**)
- Centralized approach (cvs, svn):
  - a single central repository manages all the versions of files and directories, allowing us to keep track of all the changes over time
  - the client uses a local copy of the files and keeps the synchronization with the central repository
- **Distributed approach (git):**
  - the local copy of every client is a complete repository
  - the synchronization happens exchanging patches among peers
- Code development is modelled ad a directed graph where, from the main development line (**master**), alternative development lines (**branch**) and/or stable versions (**tag**) can depart and join.

Create a new repository: `git init`
Clone: `git clone username@host:/path/to/repos`

Workflow:

- **Working directory:** keeps the actual files and directories, which may or may not be unversioned
- **Index**: is a staging area
- **HEAD**: represents the last commit made



Add files/directories to the Index: `git add <filename>`
Confirm updates and add the to the HEAD: `git commit -m "Description"`
Send committed updates to a remote server: `git push origin master`

- `master` (or any other name) is the repository branch to send to the remote server
- `origin` indicates the default remote repository, e.g. the one from which we cloned the repository

Create a new branch: `git checkout -b <branch-name>`
Get back to the master branch (or any other branch name): `git checkout master`
Send a branch to the remote repository: `git push origin <branch-name>`

Update a local repository: `git pull origin <branch-name>`

Merge a branch into the current branch: `git merge <branch-name>`

Pull Request: functionality given by platforms, is a mechanism for a developer to notify team members that they have completed a feature.
This lets everybody involved know that they need to review and discuss the code and, eventually, merge it into the master branch.

`.gitignore` file has to be put in the root folder of your development tree.
It specifies intentionally untracked files that Git should Ignore.
Each line specifies a pattern to be matched to decide whether to exclude files and/or directories.

`README` file has to be put in the root folder of your development tree.
Provides overall information about your project

3. Maven

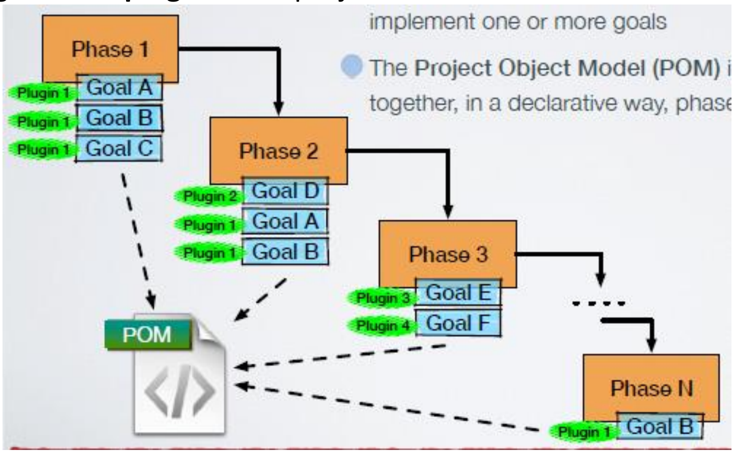
Maven is a tool for managing **Java** software projects and supporting developers in keeping track of the status of a project: build, dependency management, deployment and packaging, collaboration and documentation.

Advantages

- **coherence**: standardization of the management of Java project, increased transparency and reduced time to get an understanding of the different projects of an organization.
- **reuse**: similar projects can reuse and extend the setup of previous projects.
- **simplicity**: simplification of the creation and integration of new components as well as of the sharing of packages and executables. Moreover, the learning curve for each project is reduced.
- **maintenance**: reduced effort and resources to keep building scripts as well as development and deployment environments.

Main Concepts:

- Software development happens according to a **life cycle** made up of **phases.**
- Zero or more **goals** are associated to each phase and they are the operations actually carried out in that phase.
- Goals are implemented by means of plugins and each plugin may implement one or more goals.
- The **Project Object Model (POM)** is a single XML file which puts together, in a declarative way, phases, goals and **plugins** for a project.

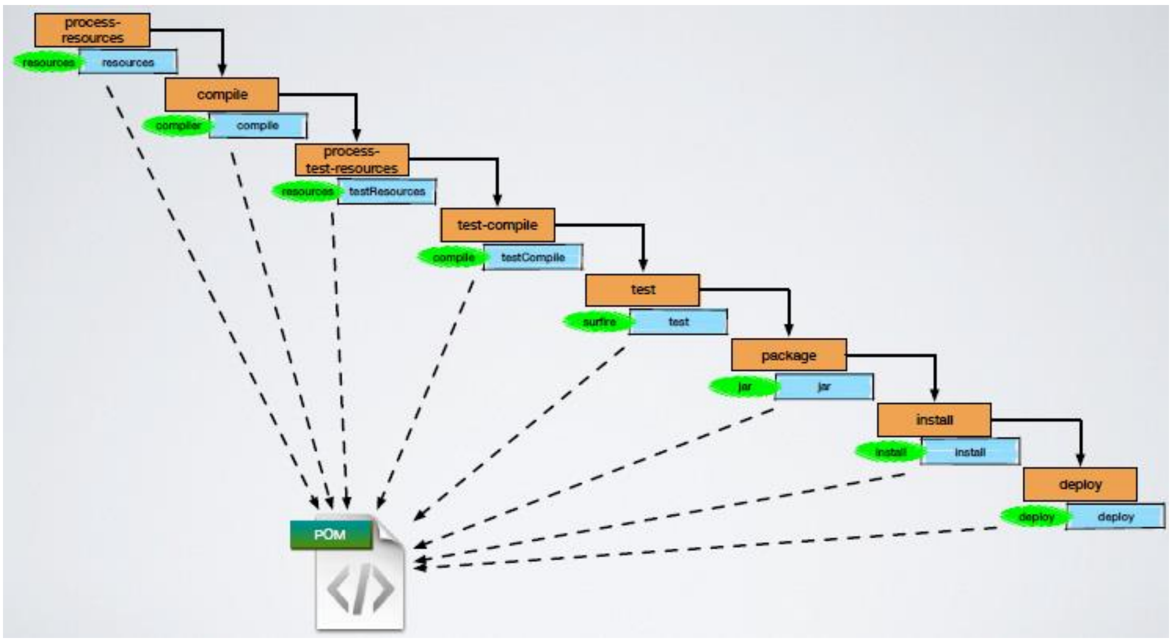


Build Lifecycle:

- A build lifecycle is needed to create, compile, integrate, test, and distribute a software project.
- The phases of a lifecycle are executed in sequence to complete that Lifecycle.
  - if you invoke an intermediate phase of a lifecycles, all the phases up to that phase will be execute.
  - if you invoke the last phase, all the phases will be executed.
- Predefined build lifecycles:
  - **clean**: manages the cleaning of the project, i.e. it deletes all the files generated by a build.
  - **default**: manages the whole development of the project.
  - **site**: manages the creation of a project site and of the documentation.

Default Build LifeCycle:

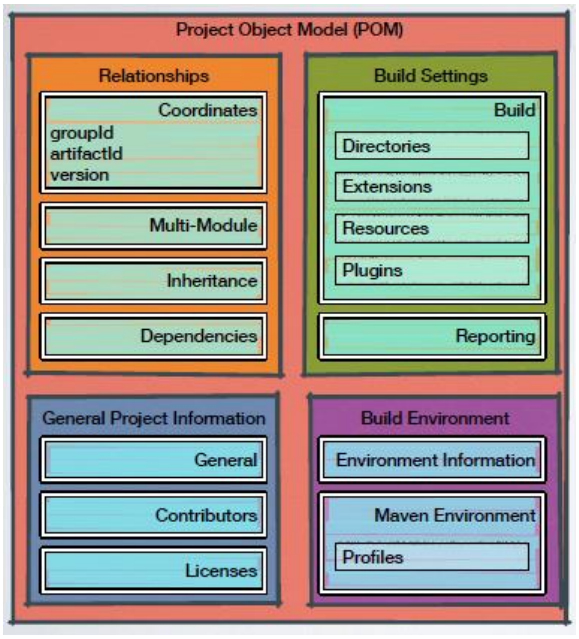
- **Setup of the project**
  - **validate:** validates the project is correct and all necessary information is available.
  - **initialize**: initialize build state, e.g. set properties or create directories.
- **Source processing**
  - **generate-sources:** generates any source code for inclusion in compilation
  - **process-sources:** processes the source code, for example to filter any values
  - **generate-resources:** generates resources for inclusion in the package
  - **process-resources:** copies and processes the resources into the destination directory, ready for packaging
  - **compile**: compiles the source code of the project
  - **process-classes:** post-processes the generated files from compilation, for example to do bytecode enhancement on Java classes.
- **Testing:**
  - **generate-test-sources:** generates any test source code for inclusion in compilation.
  - **process-test-sources:** processes the test source code, for example to filter any values
  - **generate-test-resources:** creates resources for testing.
  - **process-test-resources:** copies and processes the resources into the test destination directory.
  - **test-compile**: compiles the test source code into the test destination directory.
  - **process-test-classes:** post-processes the generated files from test compilation, for example to do bytecode enhancement on Java classes.
  - **test:** runs tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
- **Packaging**
  - **prepare-package:** performs any operations necessary to prepare a package before the actual packaging.
  - **package**: takes the compiled code and package it in its distributable format, such as a JAR.
- **Integration**
  - **pre-integration-test:** performs actions required before integration tests are executed. This may involve things such as setting up the required environment.
  - **integration-test:** process and deploy the package if necessary into an environment where integration tests can be run.
  - **post-integration-test:** performs actions required after integration tests have been executed. This may including cleaning up the environment
- **Deployment**
  - **verify:** runs any checks to verify the package is valid and meets quality criteria
  - **install**: installs the package into the local repository, for use as a dependency in other projects locally
  - **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and project.



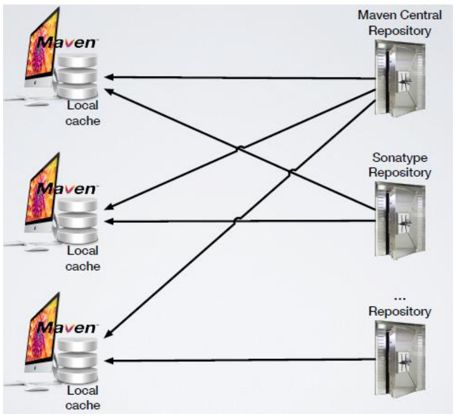
Default Build Lifecycle for JAR Packages

Project Object Model (POM):

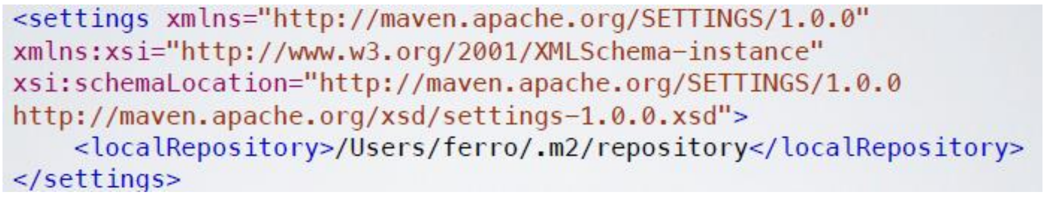
- **Relationships:** defines the structure of the project (coordinates and modules), its relationships with other projects (inheritance), dependencies on other projects and libraries
- **General project information:** maintains general information about the project, such as, project name, project Web site, organization developing the project, developers, licences
- **Build settings:** customizes the default build lifecycle by adding goals and plugins to the different phases as well as information about source, test, and resources
- **Build environment:** defines profiles corresponding to different environments and/or operating systems.



Maven Repositories:



Setting up Maven: the `settings.xml` file
Contains the overall configuration for Maven where to store the local cache for libraries and plugins, the configuration about a local repository, if any, and credentials to access it.
Has to be saved in the .m2 folder in each own home. If it does not already exist, you need to create it.



```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>/Users/ferro/.m2/repository</localRepository>
</settings>
```

Running Maven: `mvn [options] [<goal(s)>] [<phase(s)>]`

- **phase**: one or more phase names according to the available build lifecycles. Remember that all the phases up to the selected one(s) will be executed.
- **goal**: one or more goal names to be executed goal names have the following format:
  `<plugin-name>:<goal-name>`

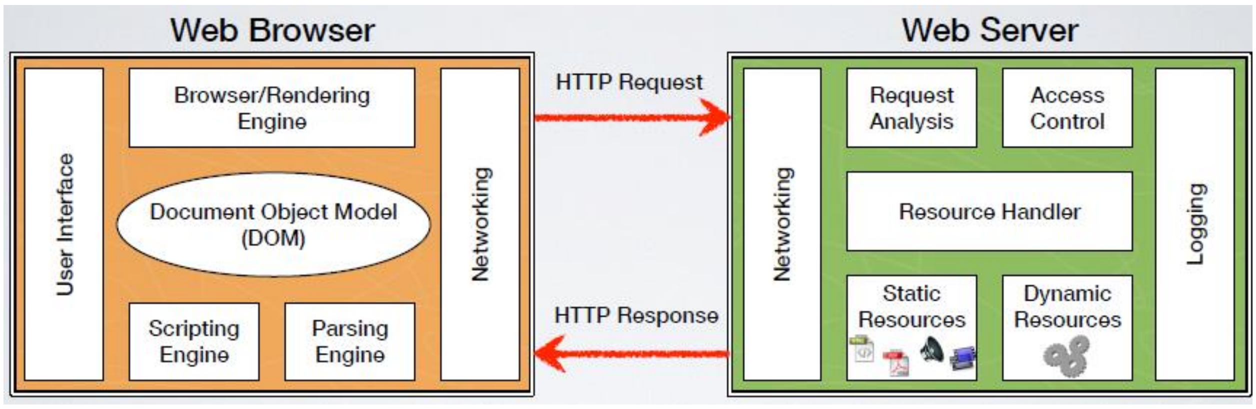
Example: `mvn clean deploy checkstyle:check`

- invokes the `clean` phase of the clean build lifecycle
- invokes the deploy `phase` (and all the phases before it) of the default build lifecycle
- invokes the check goal of the `checkstyle` plugin.

4.Java Servlet

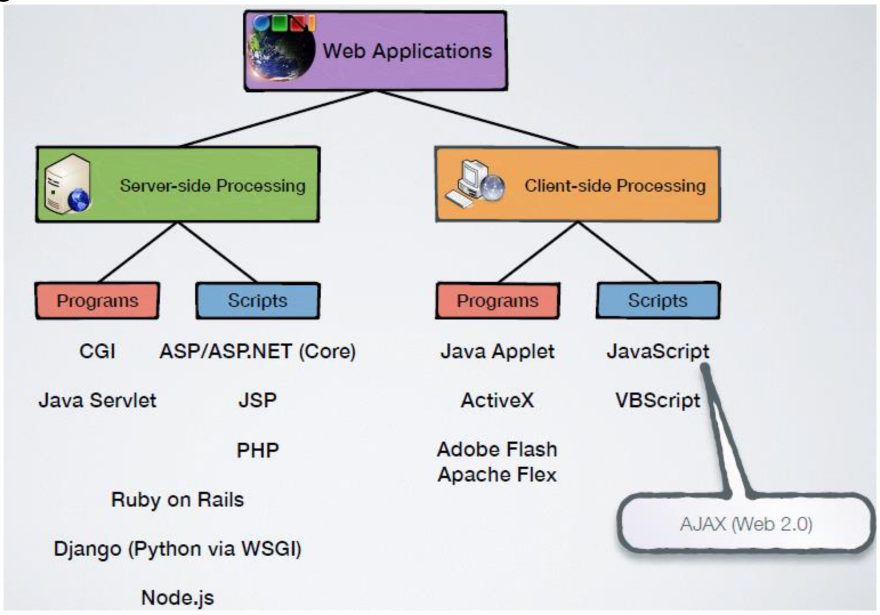
4.1. Technologies for Web applications

Browser and Server Architecture



Static resources: files
Dynamic resources: created on the fly
Logging: logs request done do the webserver, managed by the webserver

Some technologies:



CGI: common gateway interfaces, external to the process of the web server
Programs: compiled and executed
Script: interpreted
Fronted: programs are dead. Now we use scripts.

4.2. Java Servlet

Java Enterprise Edition (Java EE) platform aims at:

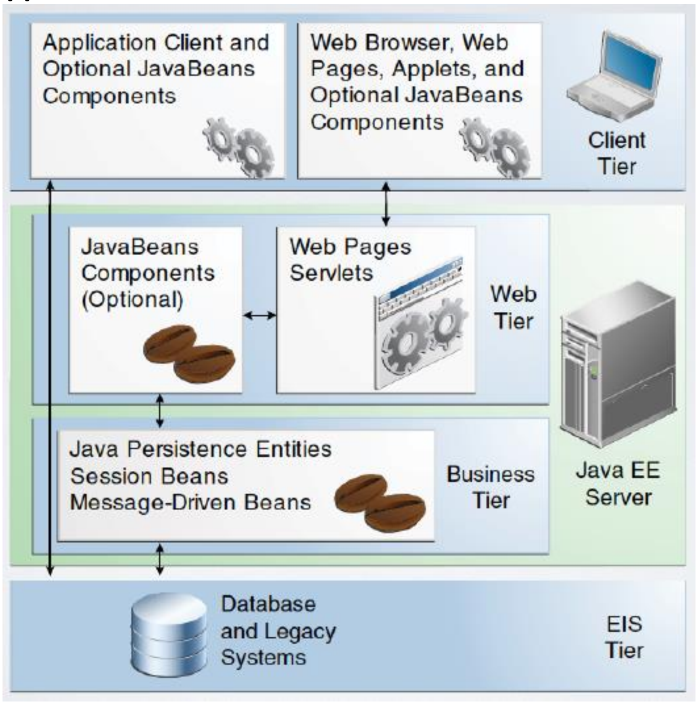
- standardize and reduce the complexity of developing multi-tiered enterprise applications.
- provide specific API for Web development, e.g. servlet, REST

Web container: implements the API defined by Java EE and allows for executing applications.
Web component: is a part of a Web application (servlet, JSP, …) hosted and executed by the Web container.

Jakarta EE: opensource version of Java EE under the Eclipse Foundation.

- Jakarta EE 8 is the same as Java EE 8, just renaming Java into Jakarta
- Jakarta EE 9, currently under development, will mainly target package name changes from `javax.*` into `jakarta.*`
- Jakarta originally was an Apache Software Foundation sub-project, retired in 2011, aimed at developing open source Java solutions, e.g. Maven, Tomcat, Lucene, …

Java EE: Multi-tiered Applications


Application Client and Optional JavaBeans Components
Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
Web Pages Servlets
Web Tier
Java Persistence Entities Session Beans Message-Driven Beans
Business Tier
Java EE Server
Database and Legacy Systems
EIS Tier

Java Servlet: is a Java technology-based Web component, managed by a container, that generates dynamic content.

- Servlets are platform-independent Java **classes** that are compiled to platform-neutral byte code that can be **loaded dynamically** into and run by a Java technology-enabled Web server.
- A servlet container may send concurrent requests to a servlet. To handle the requests, the developer must make adequate provisions for concurrent processing with multiple threads
- Servlets are **not thread-safe**.
- Servlets are part of JavaEE and they are contained in the `javax.servlet` and `javax.servlet.http` packages.

`Javax.servlet` Main Classes

- `Servlet`: defines methods that all servlets must implement.
- `ServletRequest`: defines an object to provide client request information to a servlet.
- `ServletResponse`: Defines an object to assist a servlet in sending a response to the client.
- `ServletConfig`: a servlet configuration object used by a servlet container to pass information to a servlet during initialization.
- `ServletContext`: defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

`Javax.servlet.http` Main Classes

- `HttpServlet`: Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. All your servlets will extend this calls.
- `HttpServletRequest`: extends the `ServletRequest` interface to provide request information for HTTP servlets.
- `HttpServletResponse`: extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response.
- Cookie: creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server
- `HttpSession`: provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

Servlet LifeCycle:

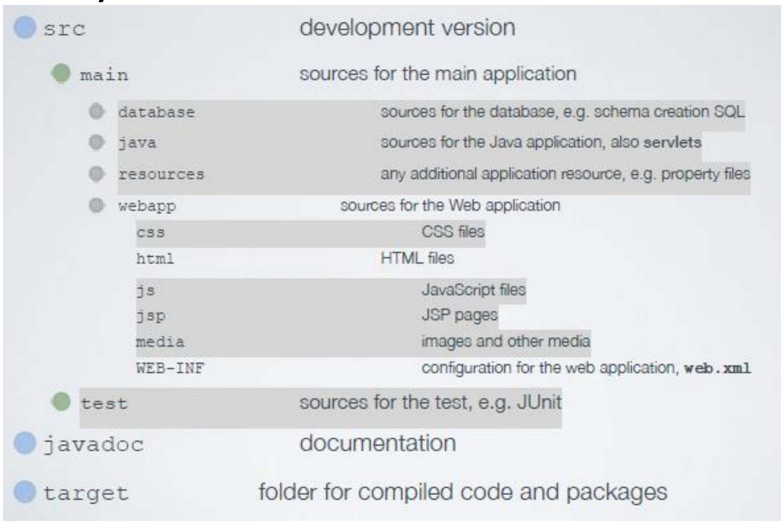
- `init(ServletConfig)` : called by the servlet container to indicate to a servlet that the servlet is being placed into service. The servlet container calls the `init` method exactly once after instantiating the servlet. The `init` method must complete successfully before the servlet can receive any requests.
  - the `ServletConfig` object gives also access to a `ServletContext` object which defines a servlet's view of the Web application within which the servlet is running and allows for servlet-container communication.
- `service(ServletRequest, ServletResponse)` : called by the servlet container to allow the servlet to respond to a request.
  - Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently. Developers must be aware to synchronize access to any shared resources such as files, network connections, and as well as the servlet's class and instance variables.
  - In the case of an `HttpServlet` this method is specialized by methods for each HTTP request
    - `doGet(HttpServletRequest, HttpServletResponse)`: called by the server (via the service method) to allow a servlet to handle a GET request.
    - `doPost(HttpServletRequest, HttpServletResponse)`: called by the server (via the service method) to allow a servlet to handle a POST request.
    - `doPUT(HttpServletRequest, HttpServletResponse)`: called by the server (via the service method) to allow a servlet to handle a PUT request.

  - `doDelete(HttpServletRequest, HttpServletResponse)`: called by the server (via the service method) to allow a servlet to handle a DELETE request.
- `destroy()`: called by the servlet container to indicate to a servlet that the servlet is being taken out of service. This method is only called once all threads within the servlet's service method have exited or after a timeout period has passed. After the servlet container calls this method, it will not call the service method again on this servlet. This method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the servlet's current state in memory.

4.3. Apache Tomcat

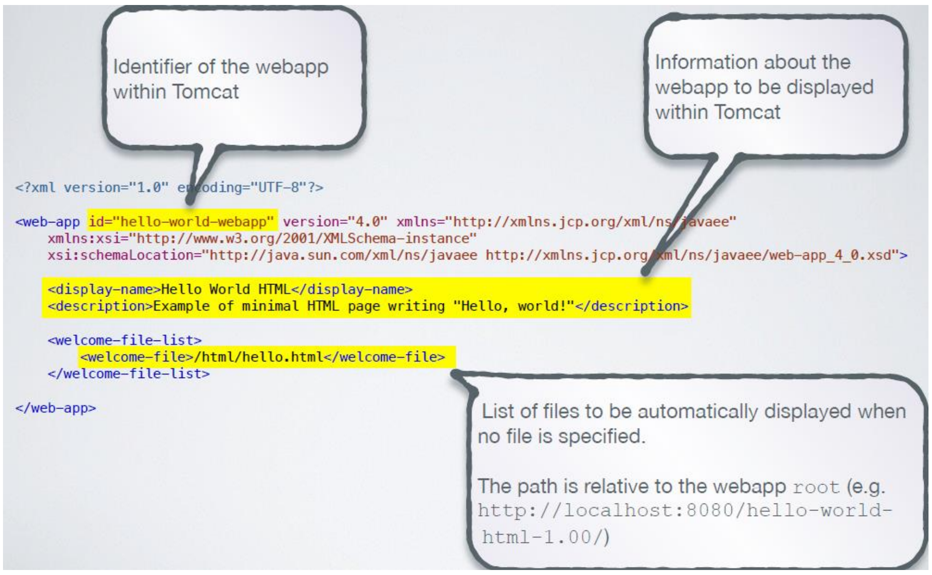
Is an open source implementation of the Java Servlet.
Tomcat 10 implements Jakarta EE, so everything is in the `jakarta.*` package
Tomcat 9 implements Java EE, so everything is in the `java.*` package.
Don't need installation. Just unzip.
`Start tomcat`: Go in bin folder, run startup shell.
`Mvn clean package`: to generate the war file
Select war file of the application to run.
Localhost:8080

Setup the Project Directory Structure



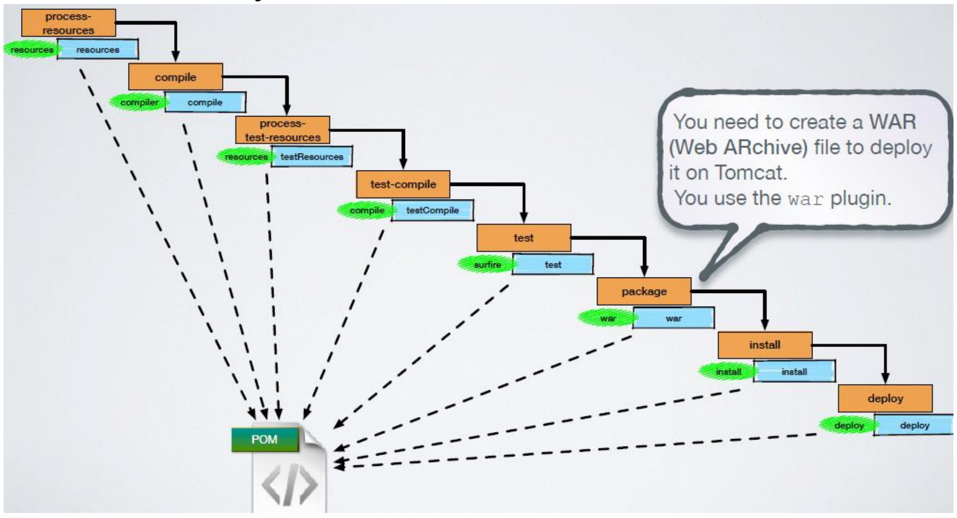
src	development version
main	sources for the main application
database	sources for the database, e.g. schema creation SQL
java	sources for the Java application, also **servlets**
resources	any additional application resource, e.g. property files
webapp	sources for the Web application
css	CSS files
html	HTML files
js	JavaScript files
jsp	JSP pages
media	images and other media
WEB-INF	configuration for the web application, **web.xml**
test	sources for the test, e.g. JUnit
javadoc	documentation
target	folder for compiled code and packages

The web.xml Configuration File:


Identifier of the webapp within Tomcat
Information about the webapp to be displayed within Tomcat
List of files to be automatically displayed when no file is specified.
The path is relative to the webapp root (e.g. http://localhost:8080/hello-world-html-1.00/)

```xml
<?xml version="1.0" encoding="UTF-8"?>

<web-app id="hello-world-webapp" version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <display-name>Hello World HTML</display-name>
    <description>Example of minimal HTML page writing "Hello, world!"</description>

    <welcome-file-list>
        <welcome-file>/html/hello.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Configuration of the Maven Project



4.4. SQL Injection

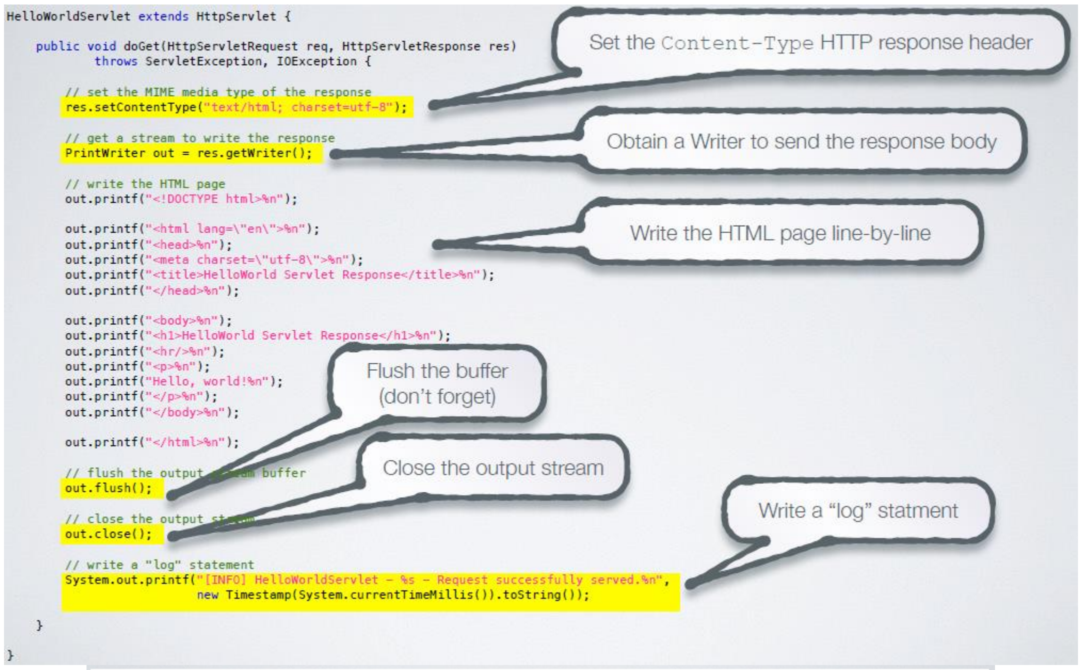
SQL Injection (SQLI) is a code injection technique, used to attack data-driven applications, in which nefarious SQL statements are inserted into an entry field for execution

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

SQL injection is the first among the OWASP “The Ten Most Critical Web Application Security Risks” in both 2013 and 2017

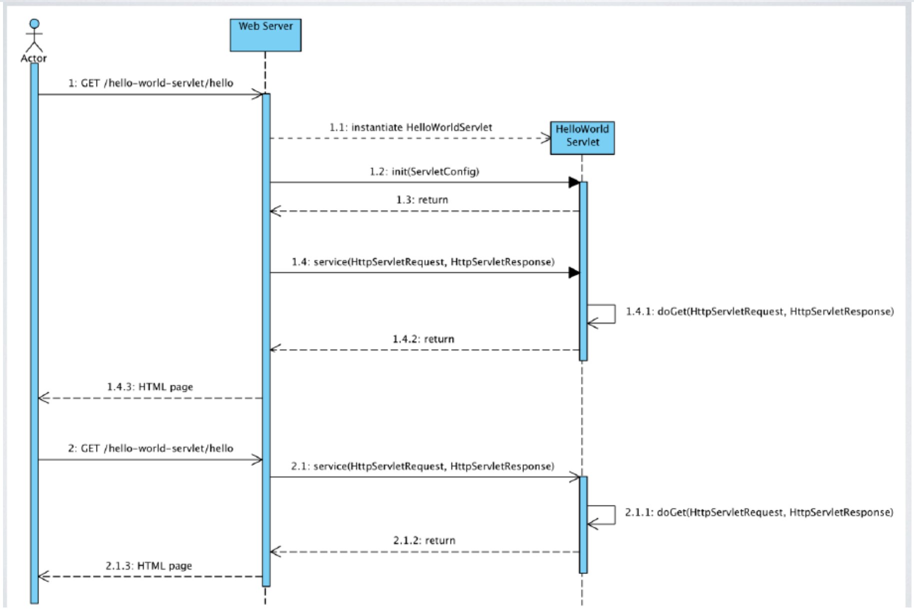
4.5. Examples

4.5.1. HelloWorld Servlet

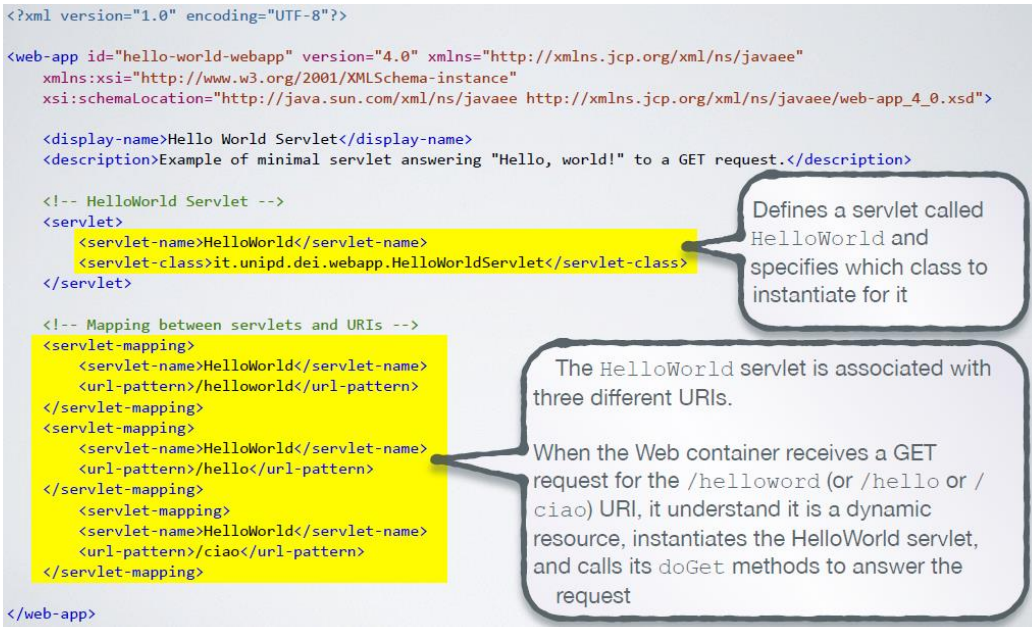


```java
HelloWorldServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        // set the MIME media type of the response
        res.setContentType("text/html; charset=utf-8");

        // get a stream to write the response
        PrintWriter out = res.getWriter();

        // write the HTML page
        out.printf("<!DOCTYPE html>%n");

        out.printf("<html lang=\"en\">%n");
        out.printf("<head>%n");
        out.printf("<meta charset=\"utf-8\">%n");
        out.printf("<title>HelloWorld Servlet Response</title>%n");
        out.printf("</head>%n");

        out.printf("<body>%n");
        out.printf("<h1>HelloWorld Servlet Response</h1>%n");
        out.printf("<hr/>%n");
        out.printf("<p>%n");
        out.printf("Hello, world!%n");
        out.printf("</p>%n");
        out.printf("</body>%n");

        out.printf("</html>%n");

        // flush the output stream buffer
        out.flush();

        // close the output stream
        out.close();

        // write a "log" statement
        System.out.printf("[INFO] HelloWorldServlet - %s - Request successfully served.%n",
                new Timestamp(System.currentTimeMillis()).toString());

    }

}
```

Set the Content-Type HTTP response header
Obtain a Writer to send the response body
Write the HTML page line-by-line
Flush the buffer (don't forget)
Close the output stream
Write a "log" statment


Actor
Web Server
1: GET /hello-world-servlet/hello
1.1: instantiate HelloWorldServlet
HelloWorld Servlet
1.2: init(ServletConfig)
1.3: return
1.4: service(HttpServletRequest, HttpServletResponse)
1.4.1: doGet(HttpServletRequest, HttpServletResponse)
1.4.2: return
1.4.3: HTML page
2: GET /hello-world-servlet/hello
2.1: service(HttpServletRequest, HttpServletResponse)
2.1.1: doGet(HttpServletRequest, HttpServletResponse)
2.1.2: return
2.1.3: HTML page

The web.xml Configuration File:



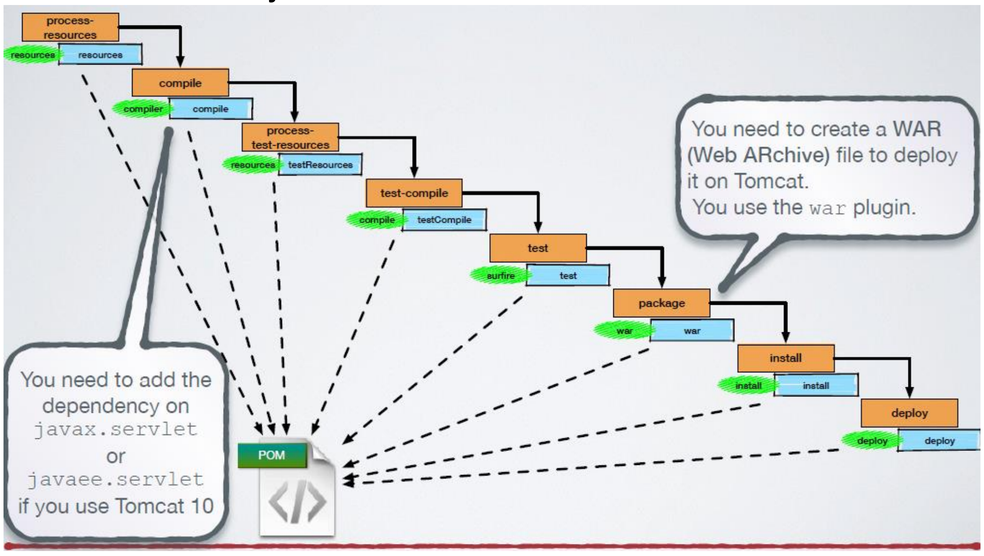
```xml
<?xml version="1.0" encoding="UTF-8"?>

<web-app id="hello-world-webapp" version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <display-name>Hello World Servlet</display-name>
    <description>Example of minimal servlet answering "Hello, world!" to a GET request.</description>

    <!-- HelloWorld Servlet -->
    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>it.unipd.dei.webapp.HelloWorldServlet</servlet-class>
    </servlet>

    <!-- Mapping between servlets and URIs -->
    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/helloworld</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
        <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/ciao</url-pattern>
    </servlet-mapping>

</web-app>
```

Defines a servlet called `HelloWorld` and specifies which class to instantiate for it

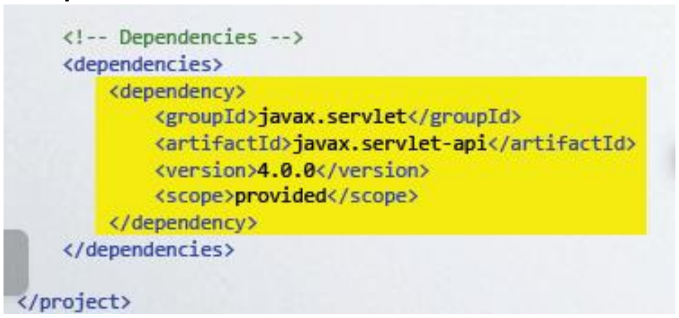
The `HelloWorld` servlet is associated with three different URIs.

When the Web container receives a GET request for the `/helloword` (or `/hello` or `/ciao`) URI, it understand it is a dynamic resource, instantiates the HelloWorld servlet, and calls its `doGet` methods to answer the request

Configuration of the Maven Project:



Project Object Model (POM):

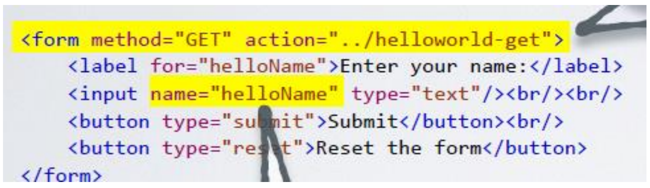


```xml
<!-- Dependencies -->
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

</project>
```

You need to add the dependency on the servlet API.
However, the Web container (Tomcat) already has this API installed. Therefore, you set scope to provided to indicate that the libraries are needed at compilation time on the local machine but they will be available in the deployment environment and so you do not need to package them in the war file.

4.5.2. Examples – Get and Post

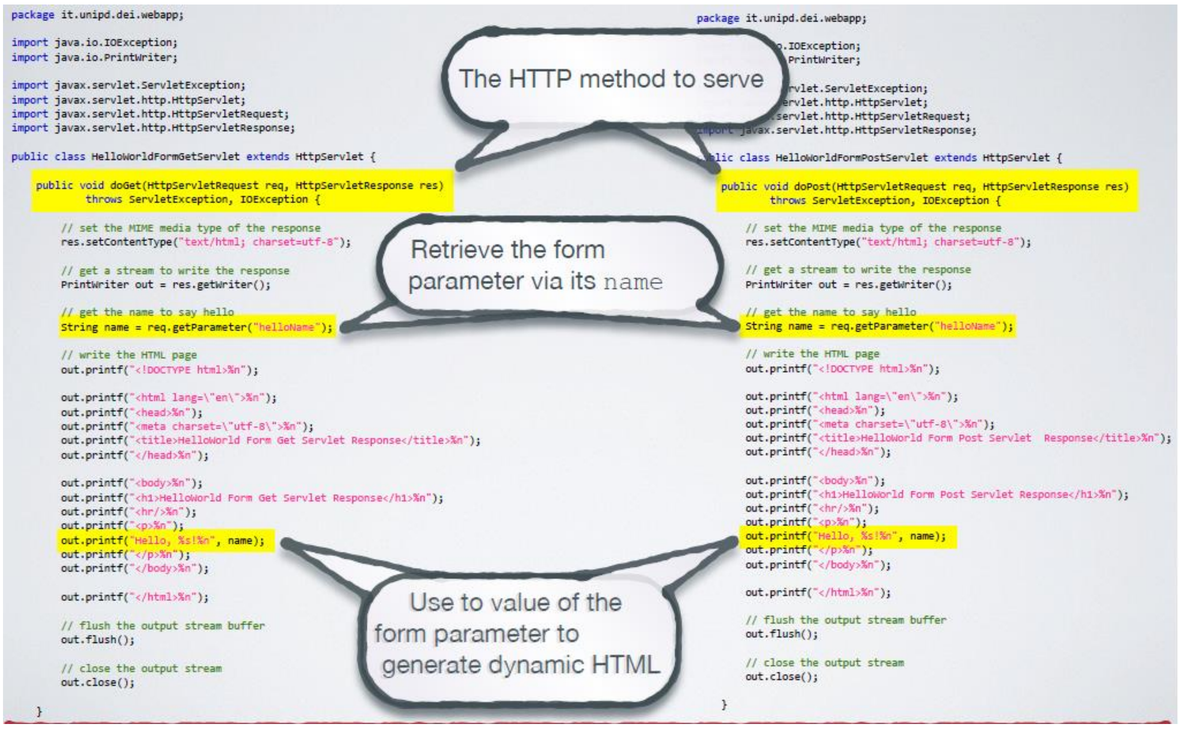
Html file:



```html
<form method="GET" action="../helloworld-get">
    <label for="helloName">Enter your name:</label>
    <input name="helloName" type="text"/><br/><br/>
    <button type="submit">Submit</button><br/>
    <button type="reset">Reset the form</button>
</form>
```

The value of the name attribute (helloName) will be used by the servlets to access the submitted form parameter.

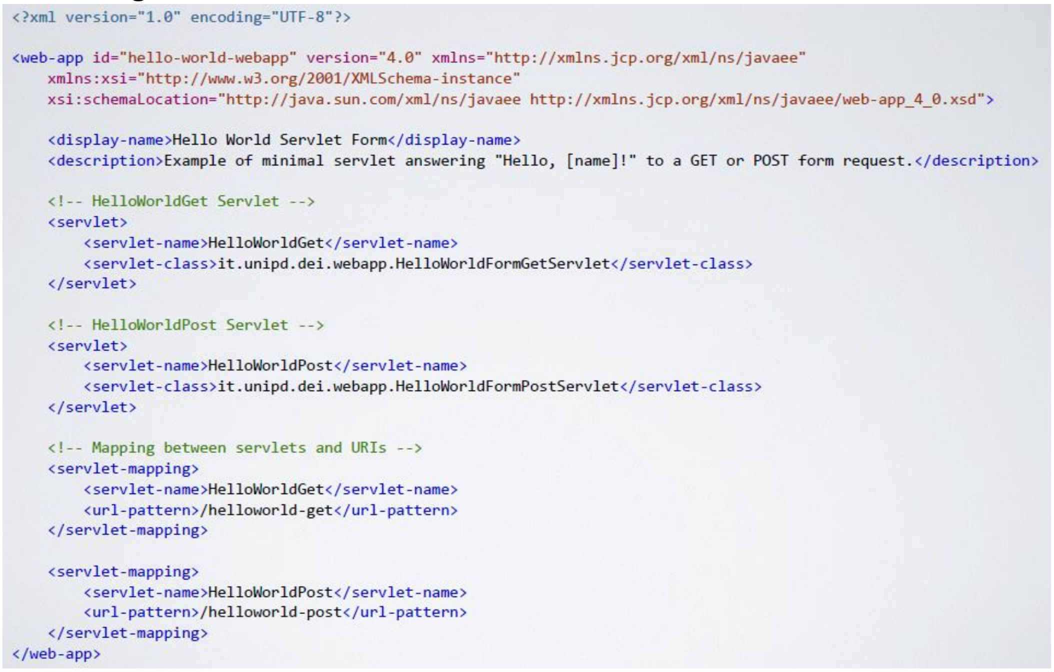
Servlets:


The HTTP method to serve
Retrieve the form parameter via its name
Use to value of the form parameter to generate dynamic HTML

```java
package it.unipd.dei.webapp;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldFormGetServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        // set the MIME media type of the response
        res.setContentType("text/html; charset=utf-8");

        // get a stream to write the response
        PrintWriter out = res.getWriter();

        // get the name to say hello
        String name = req.getParameter("helloName");

        // write the HTML page
        out.printf("<!DOCTYPE html>%n");

        out.printf("<html lang=\"en\">%n");
        out.printf("<head>%n");
        out.printf("<meta charset=\"utf-8\">%n");
        out.printf("<title>HelloWorld Form Get Servlet Response</title>%n");
        out.printf("</head>%n");

        out.printf("<body>%n");
        out.printf("<h1>HelloWorld Form Get Servlet Response</h1>%n");
        out.printf("<hr/>%n");
        out.printf("<p>%n");
        out.printf("Hello, %s!%n", name);
        out.printf("</p>%n");
        out.printf("</body>%n");

        out.printf("</html>%n");

        // flush the output stream buffer
        out.flush();

        // close the output stream
        out.close();

    }
```

```java
package it.unipd.dei.webapp;

                o.IOException;
                 PrintWriter;

               rvlet.ServletException;
              ervlet.http.HttpServlet;
             .servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

  lic class HelloWorldFormPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        // set the MIME media type of the response
        res.setContentType("text/html; charset=utf-8");

        // get a stream to write the response
        PrintWriter out = res.getWriter();

        // get the name to say hello
        String name = req.getParameter("helloName");

        // write the HTML page
        out.printf("<!DOCTYPE html>%n");

        out.printf("<html lang=\"en\">%n");
        out.printf("<head>%n");
        out.printf("<meta charset=\"utf-8\">%n");
        out.printf("<title>HelloWorld Form Post Servlet  Response</title>%n");
        out.printf("</head>%n");

        out.printf("<body>%n");
        out.printf("<h1>HelloWorld Form Post Servlet Response</h1>%n");
        out.printf("<hr/>%n");
        out.printf("<p>%n");
        out.printf("Hello, %s!%n", name);
        out.printf("</p>%n");
        out.printf("</body>%n");

        out.printf("</html>%n");

        // flush the output stream buffer
        out.flush();

        // close the output stream
        out.close();

    }
```

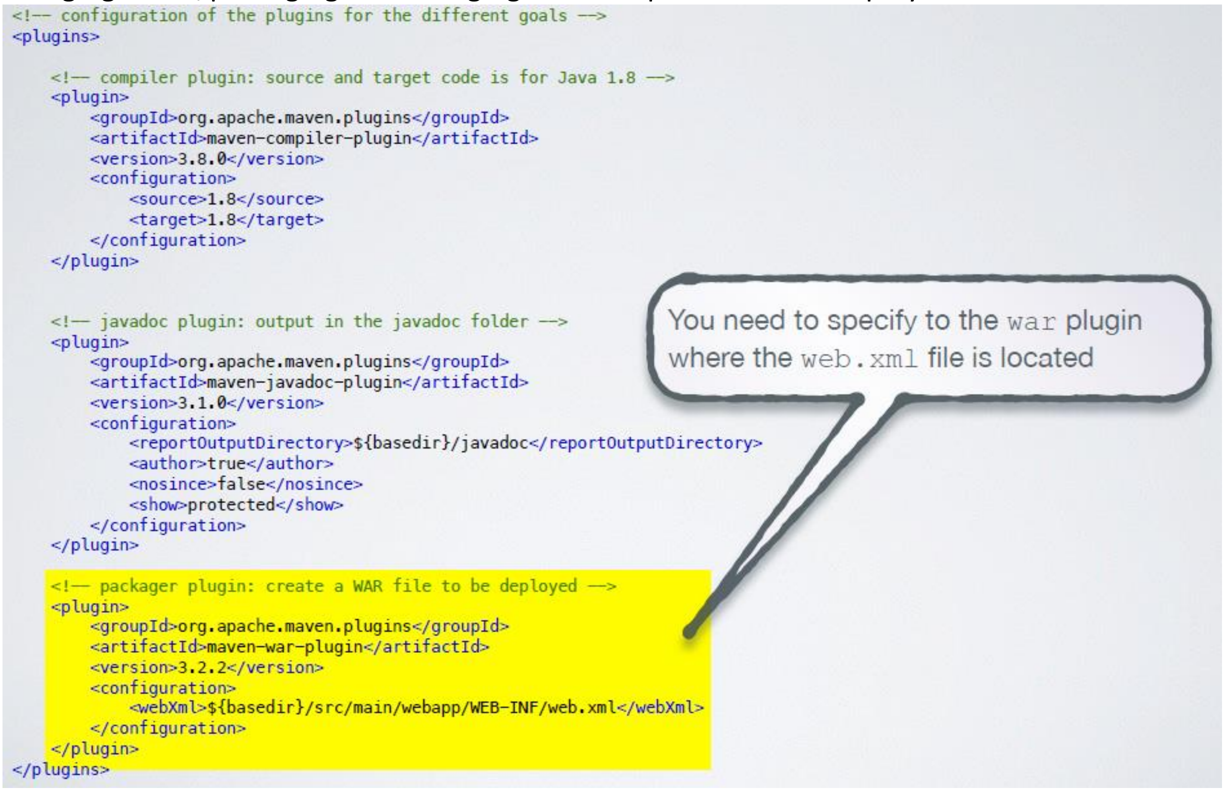
The web.xml Configuration File



```xml
<?xml version="1.0" encoding="UTF-8"?>

<web-app id="hello-world-webapp" version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <display-name>Hello World Servlet Form</display-name>
    <description>Example of minimal servlet answering "Hello, [name]!" to a GET or POST form request.</description>

    <!-- HelloWorldGet Servlet -->
    <servlet>
        <servlet-name>HelloWorldGet</servlet-name>
        <servlet-class>it.unipd.dei.webapp.HelloWorldFormGetServlet</servlet-class>
    </servlet>

    <!-- HelloWorldPost Servlet -->
    <servlet>
        <servlet-name>HelloWorldPost</servlet-name>
        <servlet-class>it.unipd.dei.webapp.HelloWorldFormPostServlet</servlet-class>
    </servlet>

    <!-- Mapping between servlets and URIs -->
    <servlet-mapping>
        <servlet-name>HelloWorldGet</servlet-name>
        <url-pattern>/helloworld-get</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>HelloWorldPost</servlet-name>
        <url-pattern>/helloworld-post</url-pattern>
    </servlet-mapping>
</web-app>
```

Project Object Model (POM):

<packaging>war</packaging>   Packaging is war to produce a file deployable on Tomcat



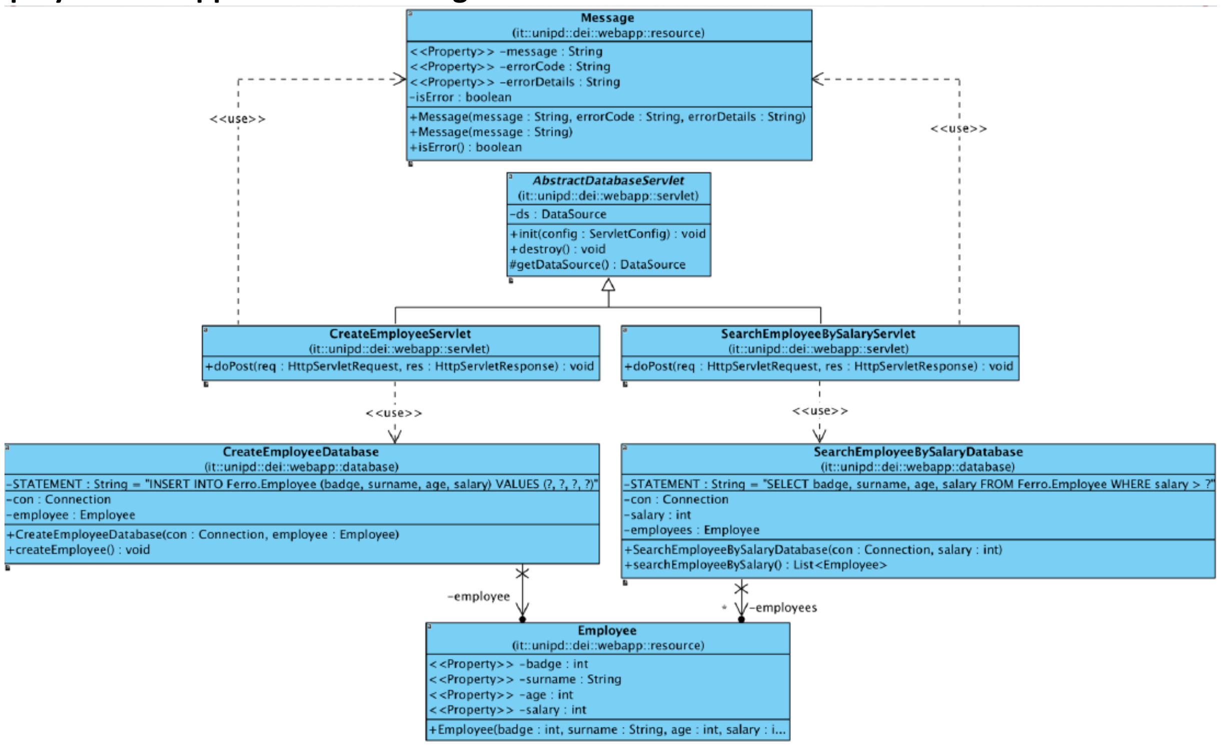
```xml
<!-- configuration of the plugins for the different goals -->
<plugins>

    <!-- compiler plugin: source and target code is for Java 1.8 -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
        </configuration>
    </plugin>


    <!-- javadoc plugin: output in the javadoc folder -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.1.0</version>
        <configuration>
            <reportOutputDirectory>${basedir}/javadoc</reportOutputDirectory>
            <author>true</author>
            <nosince>false</nosince>
            <show>protected</show>
        </configuration>
    </plugin>

    <!-- packager plugin: create a WAR file to be deployed -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
            <webXml>${basedir}/src/main/webapp/WEB-INF/web.xml</webXml>
        </configuration>
    </plugin>
</plugins>
```

You need to specify to the `war` plugin where the `web.xml` file is located

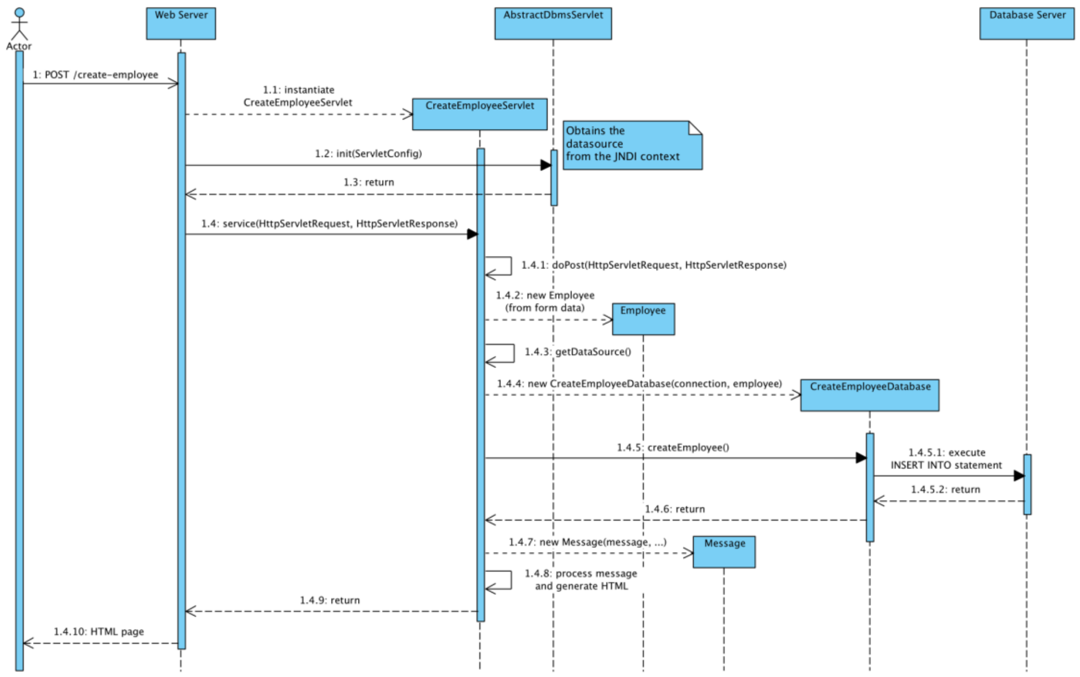
4.6. Java Servlets and Access to the Database

Overall architecture of a full-stack application

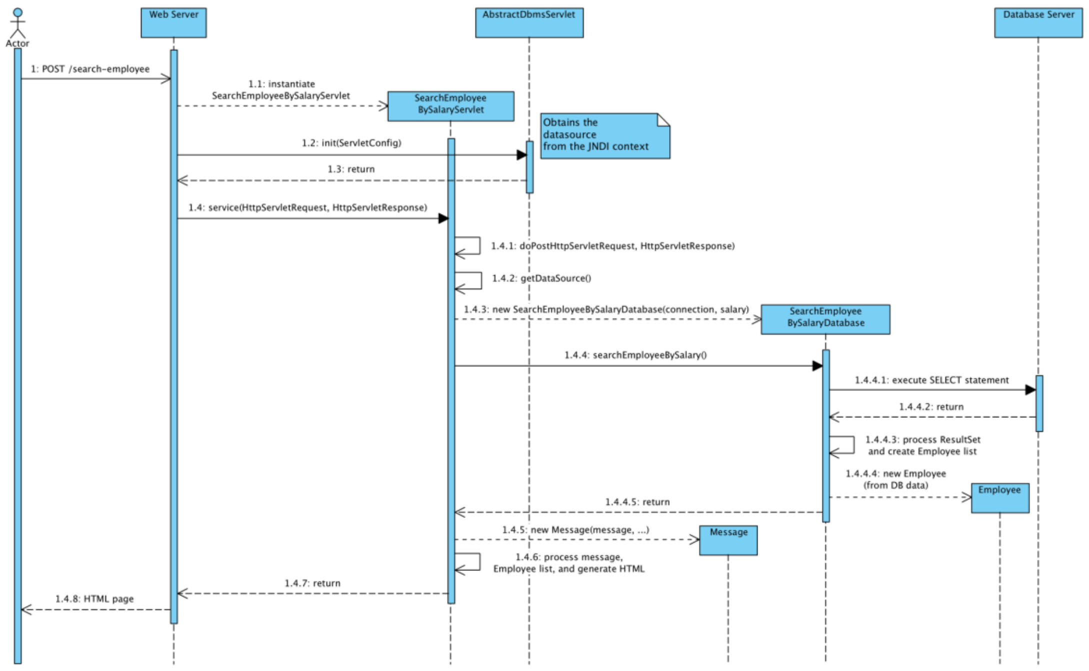
Employee Web Application Class Diagram:


Message
(it::unipd::dei::webapp::resource)
<<Property>> -message : String
<<Property>> -errorCode : String
<<Property>> -errorDetails : String
-isError : boolean
+Message(message : String, errorCode : String, errorDetails : String)
+Message(message : String)
+isError() : boolean
<<use>>
<<use>>
AbstractDatabaseServlet
(it::unipd::dei::webapp::servlet)
-ds : DataSource
+init(config : ServletConfig) : void
+destroy() : void
#getDataSource() : DataSource
CreateEmployeeServlet
(it::unipd::dei::webapp::servlet)
+doPost(req : HttpServletRequest, res : HttpServletResponse) : void
SearchEmployeeBySalaryServlet
(it::unipd::dei::webapp::servlet)
+doPost(req : HttpServletRequest, res : HttpServletResponse) : void
<<use>>
<<use>>
CreateEmployeeDatabase
(it::unipd::dei::webapp::database)
-STATEMENT : String = "INSERT INTO Ferro.Employee (badge, surname, age, salary) VALUES (?, ?, ?, ?)"
-con : Connection
-employee : Employee
+CreateEmployeeDatabase(con : Connection, employee : Employee)
+createEmployee() : void
SearchEmployeeBySalaryDatabase
(it::unipd::dei::webapp::database)
-STATEMENT : String = "SELECT badge, surname, age, salary FROM Ferro.Employee WHERE salary > ?"
-con : Connection
-salary : int
-employees : Employee
+SearchEmployeeBySalaryDatabase(con : Connection, salary : int)
+searchEmployeeBySalary() : List<Employee>
-employee
*
-employees
Employee
(it::unipd::dei::webapp::resource)
<<Property>> -badge : int
<<Property>> -surname : String
<<Property>> -age : int
<<Property>> -salary : int
+Employee(badge : int, surname : String, age : int, salary : i...

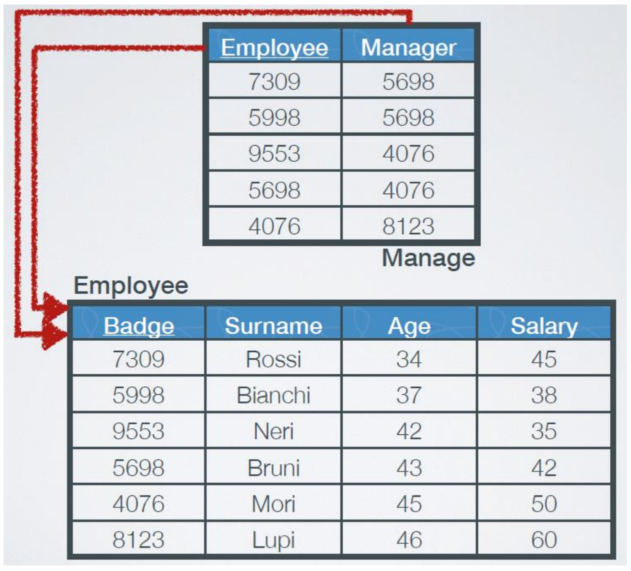
Create Employee: Sequence Diagram:


Actor
Web Server
AbstractDbmsServlet
Database Server
1: POST /create-employee
1.1: instantiate CreateEmployeeServlet
CreateEmployeeServlet
Obtains the datasource from the JNDI context
1.2: init(ServletConfig)
1.3: return
1.4: service(HttpServletRequest, HttpServletResponse)
1.4.1: doPost(HttpServletRequest, HttpServletResponse)
1.4.2: new Employee (from form data)
Employee
1.4.3: getDataSource()
1.4.4: new CreateEmployeeDatabase(connection, employee)
CreateEmployeeDatabase
1.4.5: createEmployee()
1.4.5.1: execute INSERT INTO statement
1.4.5.2: return
1.4.6: return
1.4.7: new Message(message, ...)
Message
1.4.8: process message and generate HTML
1.4.9: return
1.4.10: HTML page

Search Employee: Sequence Diagram


Actor
Web Server
AbstractDbmsServlet
Database Server
1: POST /search-employee
1.1: instantiate SearchEmployeeBySalaryServlet
SearchEmployee BySalaryServlet
Obtains the datasource from the JNDI context
1.2: init(ServletConfig)
1.3: return
1.4: service(HttpServletRequest, HttpServletResponse)
1.4.1: doPostHttpServletRequest, HttpServletResponse)
1.4.2: getDataSource()
1.4.3: new SearchEmployeeBySalaryDatabase(connection, salary)
SearchEmployee BySalaryDatabase
1.4.4: searchEmployeeBySalary()
1.4.4.1: execute SELECT statement
1.4.4.2: return
1.4.4.3: process ResultSet and create Employee list
1.4.4.4: new Employee (from DB data)
Employee
1.4.4.5: return
1.4.5: new Message(message, ...)
Message
1.4.6: process message, Employee list, and generate HTML
1.4.7: return
1.4.8: HTML page

Database:



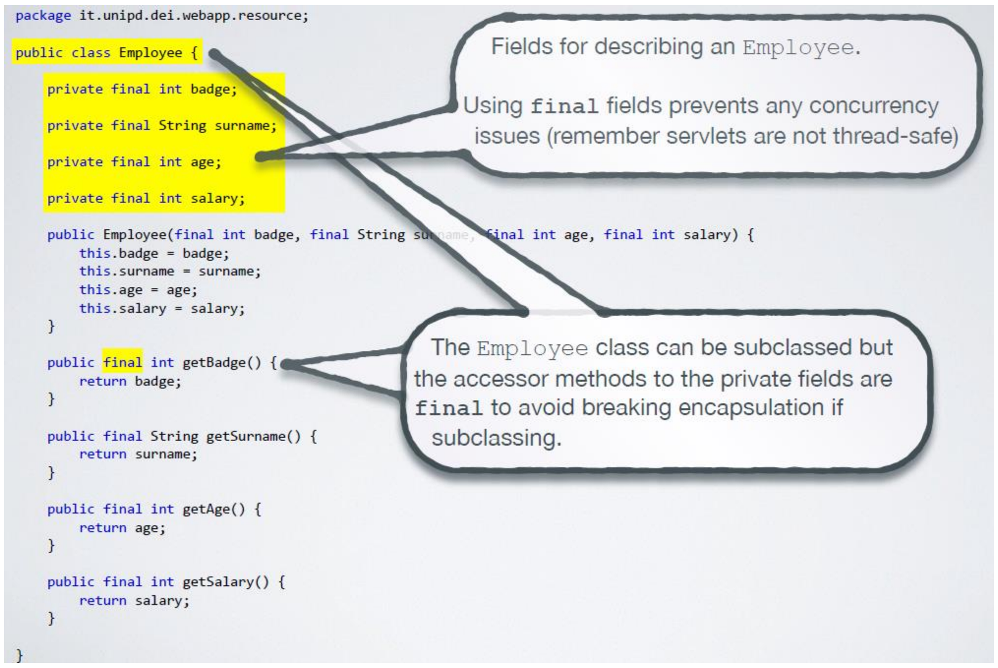
Employee	Manager
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

Manage

Employee

Badge	Surname	Age	Salary
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

The Employee Class:



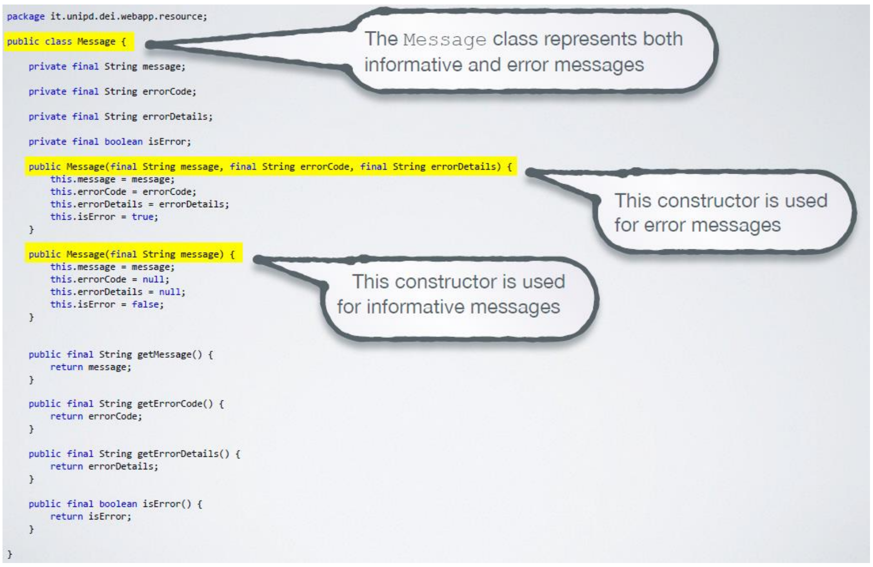
```java
package it.unipd.dei.webapp.resource;

public class Employee {

    private final int badge;

    private final String surname;

    private final int age;

    private final int salary;

    public Employee(final int badge, final String surname, final int age, final int salary) {
        this.badge = badge;
        this.surname = surname;
        this.age = age;
        this.salary = salary;
    }

    public final int getBadge() {
        return badge;
    }

    public final String getSurname() {
        return surname;
    }

    public final int getAge() {
        return age;
    }

    public final int getSalary() {
        return salary;
    }

}
```

Fields for describing an `Employee`.

Using **`final`** fields prevents any concurrency issues (remember servlets are not thread-safe)

The `Employee` class can be subclassed but the accessor methods to the private fields are **`final`** to avoid breaking encapsulation if subclassing.

The Message Class:



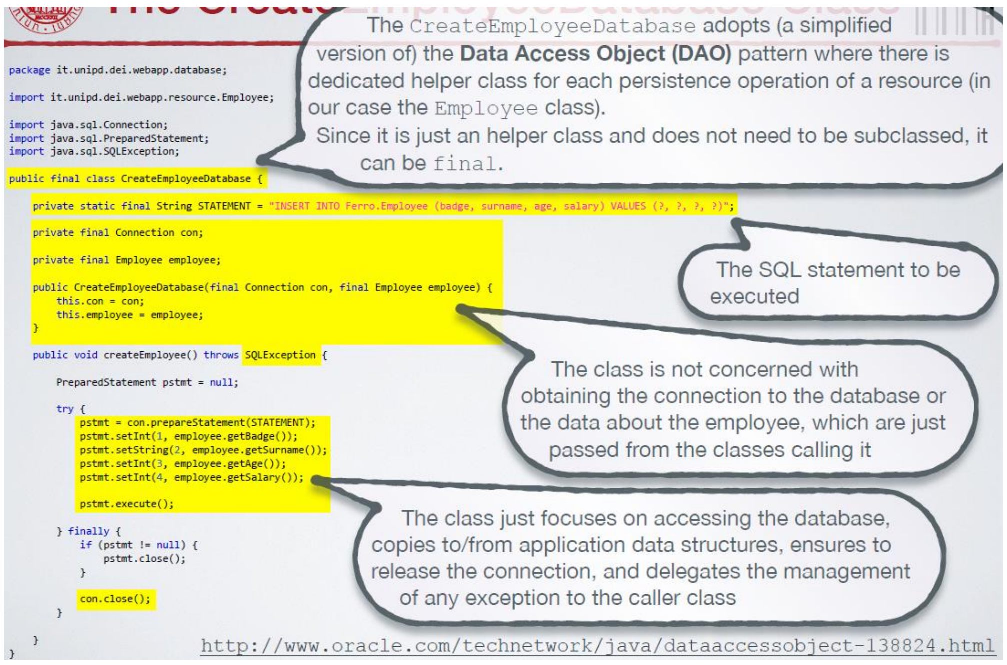
```java
package it.unipd.dei.webapp.resource;

public class Message {

    private final String message;

    private final String errorCode;

    private final String errorDetails;

    private final boolean isError;

    public Message(final String message, final String errorCode, final String errorDetails) {
        this.message = message;
        this.errorCode = errorCode;
        this.errorDetails = errorDetails;
        this.isError = true;
    }

    public Message(final String message) {
        this.message = message;
        this.errorCode = null;
        this.errorDetails = null;
        this.isError = false;
    }


    public final String getMessage() {
        return message;
    }

    public final String getErrorCode() {
        return errorCode;
    }

    public final String getErrorDetails() {
        return errorDetails;
    }

    public final boolean isError() {
        return isError;
    }

}
```

The `Message` class represents both informative and error messages

This constructor is used for error messages

This constructor is used for informative messages

The CreateEmployeeDatabase Class:



```java
package it.unipd.dei.webapp.database;

import it.unipd.dei.webapp.resource.Employee;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public final class CreateEmployeeDatabase {

    private static final String STATEMENT = "INSERT INTO Ferro.Employee (badge, surname, age, salary) VALUES (?, ?, ?, ?)";

    private final Connection con;

    private final Employee employee;

    public CreateEmployeeDatabase(final Connection con, final Employee employee) {
        this.con = con;
        this.employee = employee;
    }

    public void createEmployee() throws SQLException {

        PreparedStatement pstmt = null;

        try {
            pstmt = con.prepareStatement(STATEMENT);
            pstmt.setInt(1, employee.getBadge());
            pstmt.setString(2, employee.getSurname());
            pstmt.setInt(3, employee.getAge());
            pstmt.setInt(4, employee.getSalary());

            pstmt.execute();

        } finally {
            if (pstmt != null) {
                pstmt.close();
            }

            con.close();
        }

    }
}
```

The CreateEmployeeDatabase adopts (a simplified version of) the **Data Access Object (DAO)** pattern where there is dedicated helper class for each persistence operation of a resource (in our case the Employee class).
Since it is just an helper class and does not need to be subclassed, it can be final.

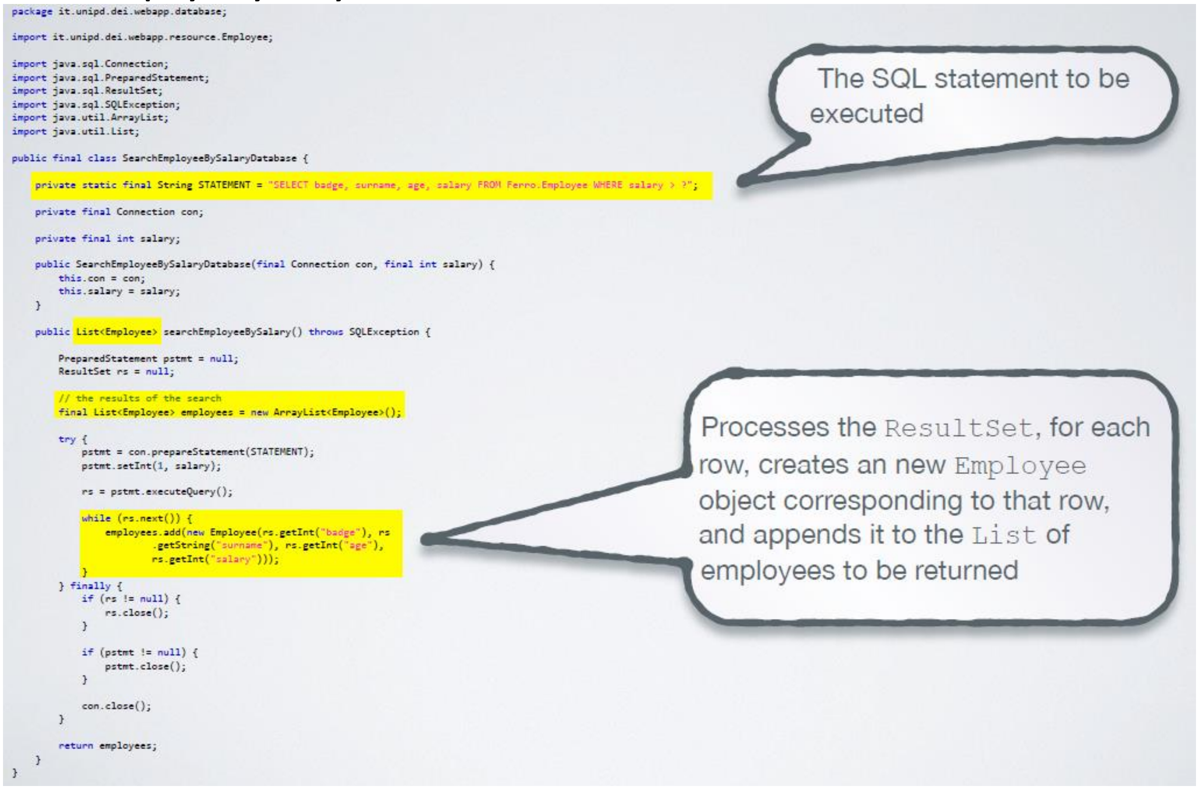
The SQL statement to be executed

The class is not concerned with obtaining the connection to the database or the data about the employee, which are just passed from the classes calling it

The class just focuses on accessing the database, copies to/from application data structures, ensures to release the connection, and delegates the management of any exception to the caller class

http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

The SearchEmployeeBySalaryDatabase Class:

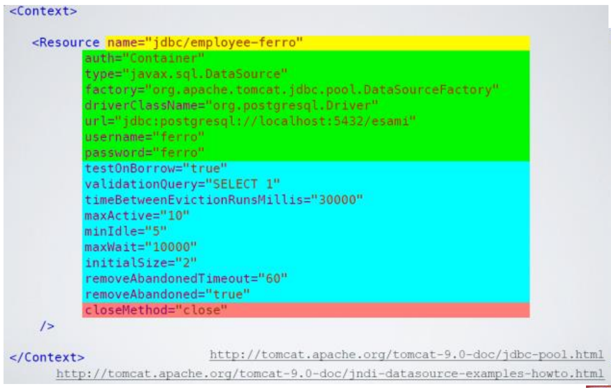


```java
package it.unipd.dei.webapp.database;

import it.unipd.dei.webapp.resource.Employee;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public final class SearchEmployeeBySalaryDatabase {

    private static final String STATEMENT = "SELECT badge, surname, age, salary FROM Ferro.Employee WHERE salary > ?";

    private final Connection con;

    private final int salary;

    public SearchEmployeeBySalaryDatabase(final Connection con, final int salary) {
        this.con = con;
        this.salary = salary;
    }

    public List<Employee> searchEmployeeBySalary() throws SQLException {

        PreparedStatement pstmt = null;
        ResultSet rs = null;

        // the results of the search
        final List<Employee> employees = new ArrayList<Employee>();

        try {
            pstmt = con.prepareStatement(STATEMENT);
            pstmt.setInt(1, salary);

            rs = pstmt.executeQuery();

            while (rs.next()) {
                employees.add(new Employee(rs.getInt("badge"), rs
                        .getString("surname"), rs.getInt("age"),
                        rs.getInt("salary")));
            }
        } finally {
            if (rs != null) {
                rs.close();
            }

            if (pstmt != null) {
                pstmt.close();
            }

            con.close();
        }

        return employees;
    }
}
```

The SQL statement to be executed

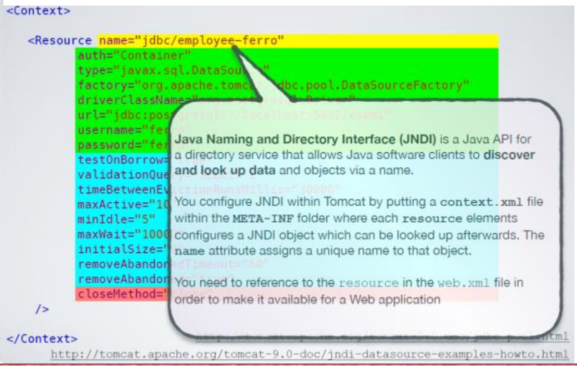
Processes the ResultSet, for each row, creates an new Employee object corresponding to that row, and appends it to the List of employees to be returned

Pool of Database Connections via Tomcat: context.xml



```
<Context>

    <Resource name="jdbc/employee-ferro"
              auth="Container"
              type="javax.sql.DataSource"
              factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
              driverClassName="org.postgresql.Driver"
              url="jdbc:postgresql://localhost:5432/esami"
              username="ferro"
              password="ferro"
              testOnBorrow="true"
              validationQuery="SELECT 1"
              timeBetweenEvictionRunsMillis="30000"
              maxActive="10"
              minIdle="5"
              maxWait="10000"
              initialSize="2"
              removeAbandonedTimeout="60"
              removeAbandoned="true"
              closeMethod="close"
     />

</Context>
```

http://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html
http://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html

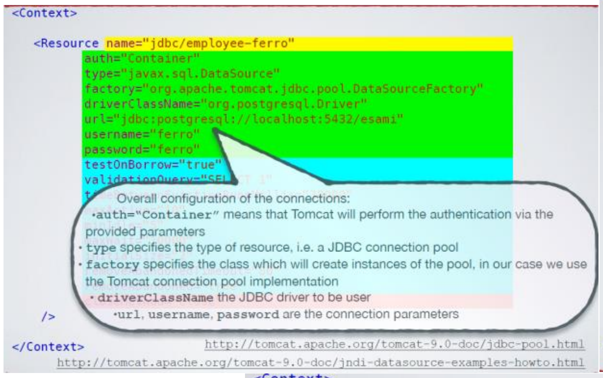


Java Naming and Directory Interface (JNDI) is a Java API for a directory service that allows Java software clients to **discover and look up data** and objects via a name.

You configure JNDI within Tomcat by putting a `context.xml` file within the `META-INF` folder where each `resource` elements configures a JNDI object which can be looked up afterwards. The `name` attribute assigns a unique name to that object.

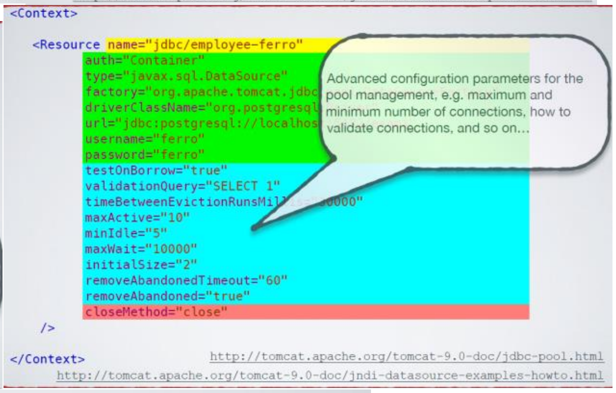
You need to reference to the `resource` in the `web.xml` file in order to make it available for a Web application

http://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html



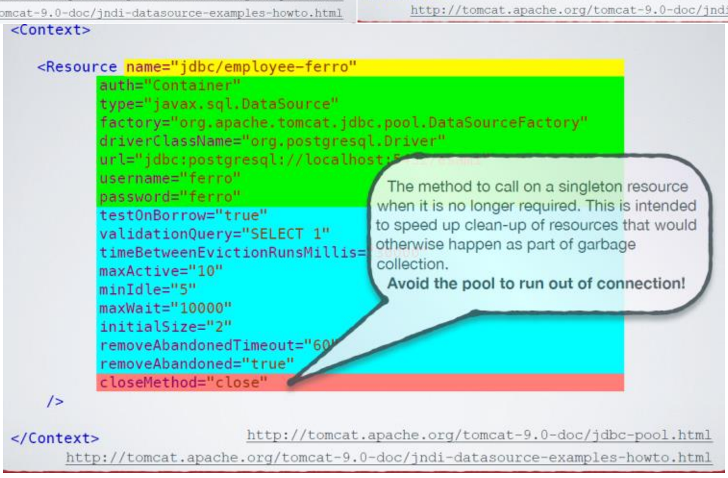
Overall configuration of the connections:
- `auth="Container"` means that Tomcat will perform the authentication via the provided parameters
- `type` specifies the type of resource, i.e. a JDBC connection pool
- `factory` specifies the class which will create instances of the pool, in our case we use the Tomcat connection pool implementation
- `driverClassName` the JDBC driver to be user
- `url`, `username`, `password` are the connection parameters

http://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html
http://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html



Advanced configuration parameters for the pool management, e.g. maximum and minimum number of connections, how to validate connections, and so on…

http://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html
http://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html

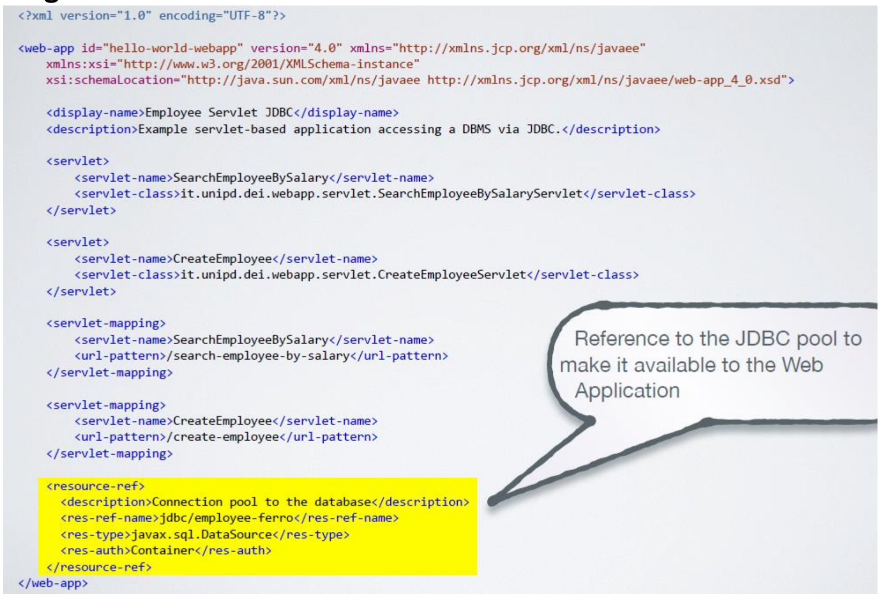


The method to call on a singleton resource when it is no longer required. This is intended to speed up clean-up of resources that would otherwise happen as part of garbage collection.

Avoid the pool to run out of connection!

http://tomcat.apache.org/tomcat-9.0-doc/jdbc-pool.html
http://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html

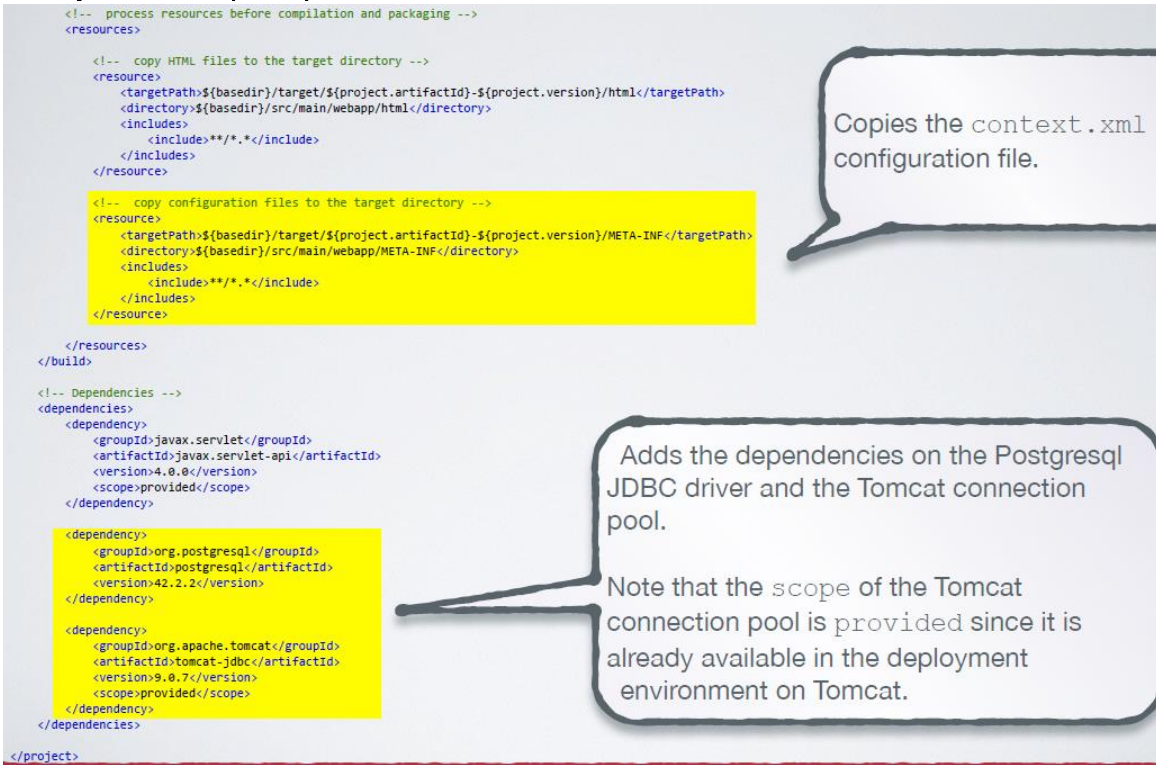
The web.xml Configuration File



```
<?xml version="1.0" encoding="UTF-8"?>

<web-app id="hello-world-webapp" version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <display-name>Employee Servlet JDBC</display-name>
    <description>Example servlet-based application accessing a DBMS via JDBC.</description>

    <servlet>
        <servlet-name>SearchEmployeeBySalary</servlet-name>
        <servlet-class>it.unipd.dei.webapp.servlet.SearchEmployeeBySalaryServlet</servlet-class>
    </servlet>

    <servlet>
        <servlet-name>CreateEmployee</servlet-name>
        <servlet-class>it.unipd.dei.webapp.servlet.CreateEmployeeServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>SearchEmployeeBySalary</servlet-name>
        <url-pattern>/search-employee-by-salary</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>CreateEmployee</servlet-name>
        <url-pattern>/create-employee</url-pattern>
    </servlet-mapping>

    <resource-ref>
      <description>Connection pool to the database</description>
      <res-ref-name>jdbc/employee-ferro</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Reference to the JDBC pool to make it available to the Web Application

Project Object Model (POM)



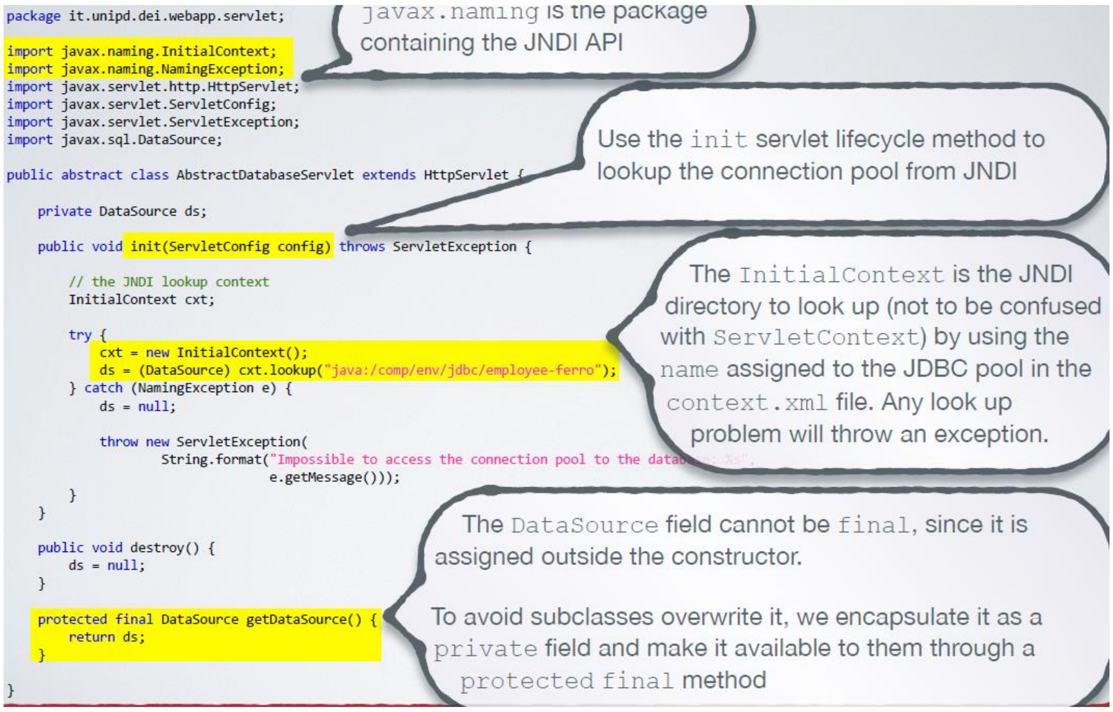
```
    <!--  process resources before compilation and packaging -->
    <resources>

        <!--  copy HTML files to the target directory -->
        <resource>
            <targetPath>${basedir}/target/${project.artifactId}-${project.version}/html</targetPath>
            <directory>${basedir}/src/main/webapp/html</directory>
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>

        <!--  copy configuration files to the target directory -->
        <resource>
            <targetPath>${basedir}/target/${project.artifactId}-${project.version}/META-INF</targetPath>
            <directory>${basedir}/src/main/webapp/META-INF</directory>
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>

    </resources>
</build>

<!-- Dependencies -->
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.2.2</version>
    </dependency>

    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-jdbc</artifactId>
        <version>9.0.7</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

</project>
```

Copies the `context.xml` configuration file.

Adds the dependencies on the Postgresql JDBC driver and the Tomcat connection pool.

Note that the `scope` of the Tomcat connection pool is `provided` since it is already available in the deployment environment on Tomcat.

The AbstractDatabaseServlet Class:



```java
package it.unipd.dei.webapp.servlet;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.sql.DataSource;

public abstract class AbstractDatabaseServlet extends HttpServlet {

    private DataSource ds;

    public void init(ServletConfig config) throws ServletException {

        // the JNDI lookup context
        InitialContext cxt;

        try {
            cxt = new InitialContext();
            ds = (DataSource) cxt.lookup("java:/comp/env/jdbc/employee-ferro");
        } catch (NamingException e) {
            ds = null;

            throw new ServletException(
                    String.format("Impossible to access the connection pool to the database: %s",
                                  e.getMessage()));
        }
    }

    public void destroy() {
        ds = null;
    }

    protected final DataSource getDataSource() {
        return ds;
    }

}
```

`javax.naming` is the package containing the JNDI API

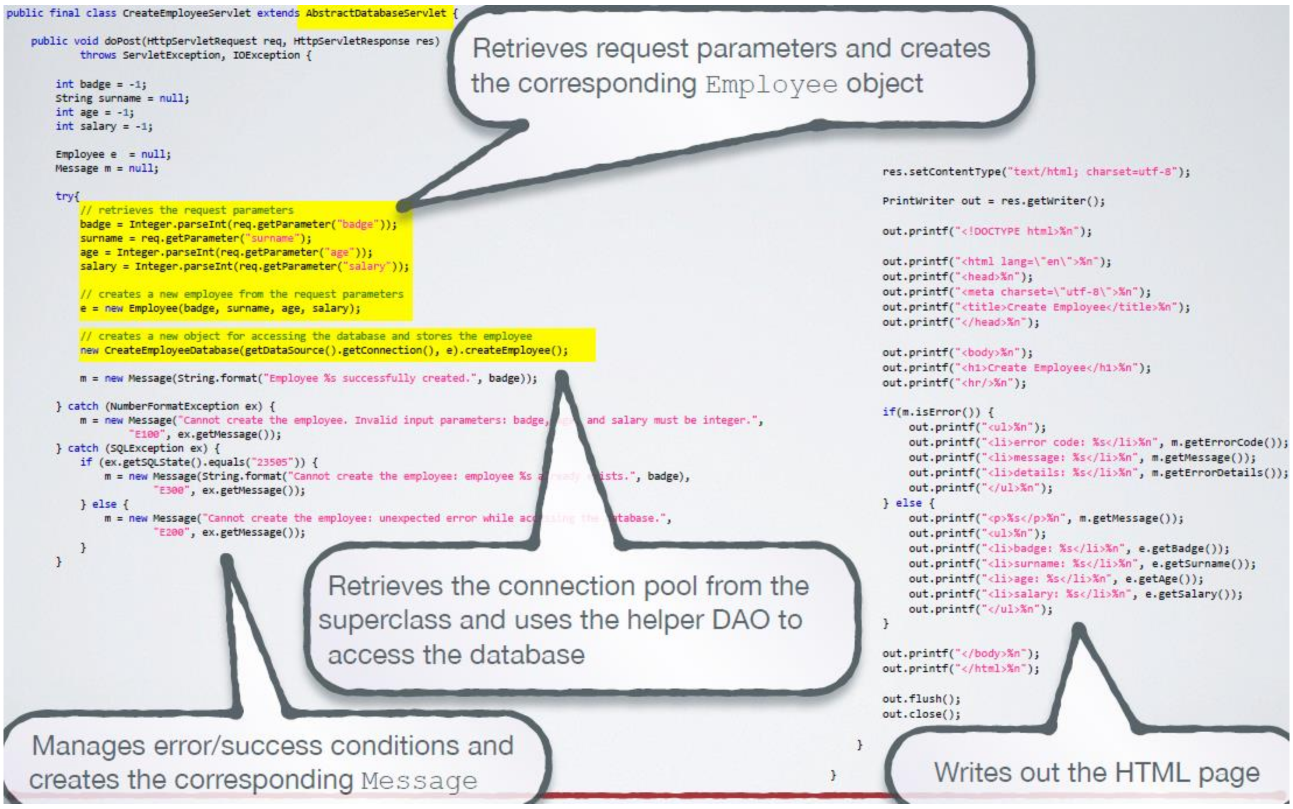
Use the `init` servlet lifecycle method to lookup the connection pool from JNDI

The `InitialContext` is the JNDI directory to look up (not to be confused with `ServletContext`) by using the `name` assigned to the JDBC pool in the `context.xml` file. Any look up problem will throw an exception.

The `DataSource` field cannot be `final`, since it is assigned outside the constructor.

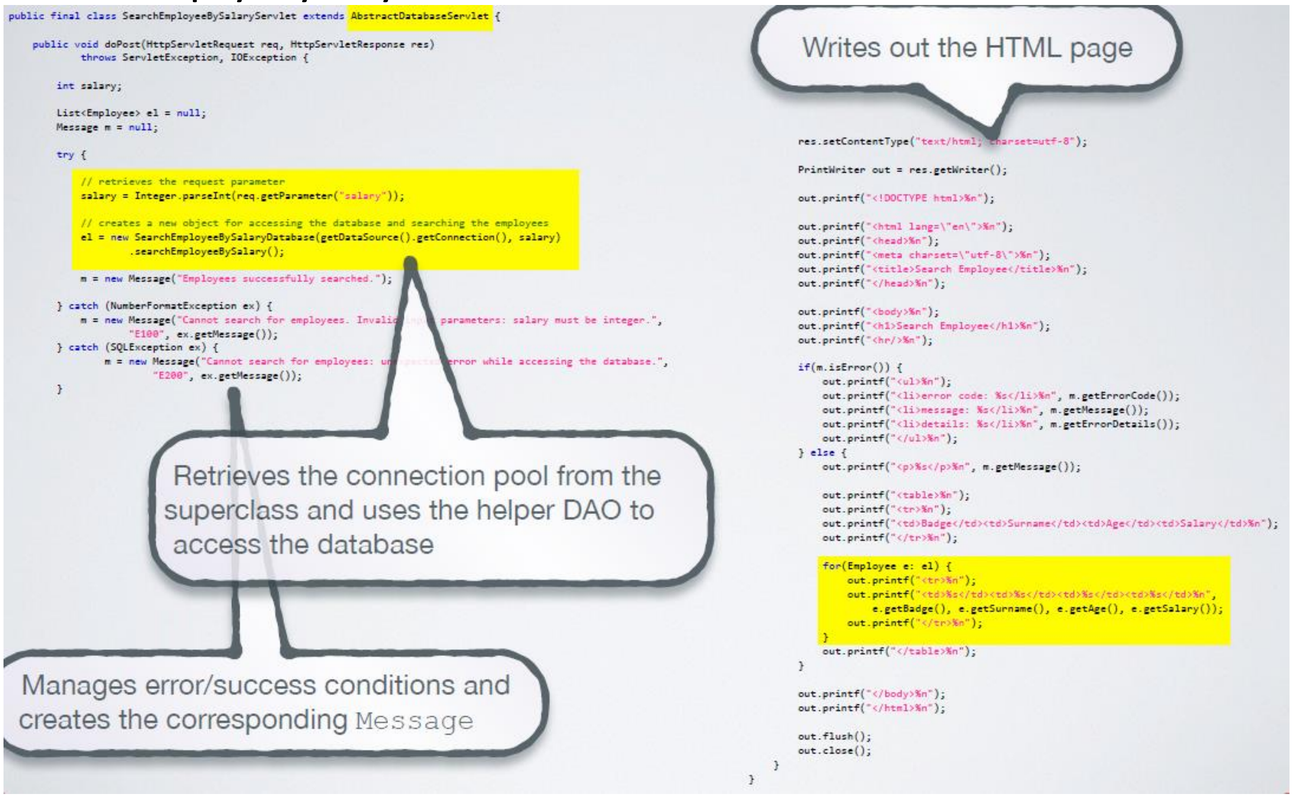
To avoid subclasses overwrite it, we encapsulate it as a `private` field and make it available to them through a `protected final` method

The CreateEmployeeServlet Class:


Retrieves request parameters and creates the corresponding Employee object
Retrieves the connection pool from the superclass and uses the helper DAO to access the database
Manages error/success conditions and creates the corresponding Message
Writes out the HTML page

```java
public final class CreateEmployeeServlet extends AbstractDatabaseServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        int badge = -1;
        String surname = null;
        int age = -1;
        int salary = -1;

        Employee e  = null;
        Message m = null;

        try{
            // retrieves the request parameters
            badge = Integer.parseInt(req.getParameter("badge"));
            surname = req.getParameter("surname");
            age = Integer.parseInt(req.getParameter("age"));
            salary = Integer.parseInt(req.getParameter("salary"));

            // creates a new employee from the request parameters
            e = new Employee(badge, surname, age, salary);

            // creates a new object for accessing the database and stores the employee
            new CreateEmployeeDatabase(getDataSource().getConnection(), e).createEmployee();

            m = new Message(String.format("Employee %s successfully created.", badge));

        } catch (NumberFormatException ex) {
            m = new Message("Cannot create the employee. Invalid input parameters: badge, age, and salary must be integer.",
                    "E100", ex.getMessage());
        } catch (SQLException ex) {
            if (ex.getSQLState().equals("23505")) {
                m = new Message(String.format("Cannot create the employee: employee %s already exists.", badge),
                        "E300", ex.getMessage());
            } else {
                m = new Message("Cannot create the employee: unexpected error while accessing the database.",
                        "E200", ex.getMessage());
            }
        }

        res.setContentType("text/html; charset=utf-8");

        PrintWriter out = res.getWriter();

        out.printf("<!DOCTYPE html>%n");

        out.printf("<html lang=\"en\">%n");
        out.printf("<head>%n");
        out.printf("<meta charset=\"utf-8\">%n");
        out.printf("<title>Create Employee</title>%n");
        out.printf("</head>%n");

        out.printf("<body>%n");
        out.printf("<h1>Create Employee</h1>%n");
        out.printf("<hr/>%n");

        if(m.isError()) {
            out.printf("<ul>%n");
            out.printf("<li>error code: %s</li>%n", m.getErrorCode());
            out.printf("<li>message: %s</li>%n", m.getMessage());
            out.printf("<li>details: %s</li>%n", m.getErrorDetails());
            out.printf("</ul>%n");
        } else {
            out.printf("<p>%s</p>%n", m.getMessage());
            out.printf("<ul>%n");
            out.printf("<li>badge: %s</li>%n", e.getBadge());
            out.printf("<li>surname: %s</li>%n", e.getSurname());
            out.printf("<li>age: %s</li>%n", e.getAge());
            out.printf("<li>salary: %s</li>%n", e.getSalary());
            out.printf("</ul>%n");
        }

        out.printf("</body>%n");
        out.printf("</html>%n");

        out.flush();
        out.close();
    }
}
```

The SearchEmployeeBySalaryServlet Class


Writes out the HTML page
Retrieves the connection pool from the superclass and uses the helper DAO to access the database
Manages error/success conditions and creates the corresponding Message

```java
public final class SearchEmployeeBySalaryServlet extends AbstractDatabaseServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        int salary;

        List<Employee> el = null;
        Message m = null;

        try {

            // retrieves the request parameter
            salary = Integer.parseInt(req.getParameter("salary"));

            // creates a new object for accessing the database and searching the employees
            el = new SearchEmployeeBySalaryDatabase(getDataSource().getConnection(), salary)
                    .searchEmployeeBySalary();

            m = new Message("Employees successfully searched.");

        } catch (NumberFormatException ex) {
            m = new Message("Cannot search for employees. Invalid input parameters: salary must be integer.",
                    "E100", ex.getMessage());
        } catch (SQLException ex) {
                m = new Message("Cannot search for employees: unexpected error while accessing the database.",
                        "E200", ex.getMessage());
        }

        res.setContentType("text/html; charset=utf-8");

        PrintWriter out = res.getWriter();

        out.printf("<!DOCTYPE html>%n");

        out.printf("<html lang=\"en\">%n");
        out.printf("<head>%n");
        out.printf("<meta charset=\"utf-8\">%n");
        out.printf("<title>Search Employee</title>%n");
        out.printf("</head>%n");

        out.printf("<body>%n");
        out.printf("<h1>Search Employee</h1>%n");
        out.printf("<hr/>%n");

        if(m.isError()) {
            out.printf("<ul>%n");
            out.printf("<li>error code: %s</li>%n", m.getErrorCode());
            out.printf("<li>message: %s</li>%n", m.getMessage());
            out.printf("<li>details: %s</li>%n", m.getErrorDetails());
            out.printf("</ul>%n");
        } else {
            out.printf("<p>%s</p>%n", m.getMessage());

            out.printf("<table>%n");
            out.printf("<tr>%n");
            out.printf("<td>Badge</td><td>Surname</td><td>Age</td><td>Salary</td>%n");
            out.printf("</tr>%n");

            for(Employee e: el) {
                out.printf("<tr>%n");
                out.printf("<td>%s</td><td>%s</td><td>%s</td><td>%s</td>%n",
                    e.getBadge(), e.getSurname(), e.getAge(), e.getSalary());
                out.printf("</tr>%n");
            }
            out.printf("</table>%n");
        }

        out.printf("</body>%n");
        out.printf("</html>%n");

        out.flush();
        out.close();
    }
}
```

5.Java Server Pages (JSP)

5.1. JavaServer Pages

Creating HTML (CSS, JS) directly from servlets is a cumbersome process

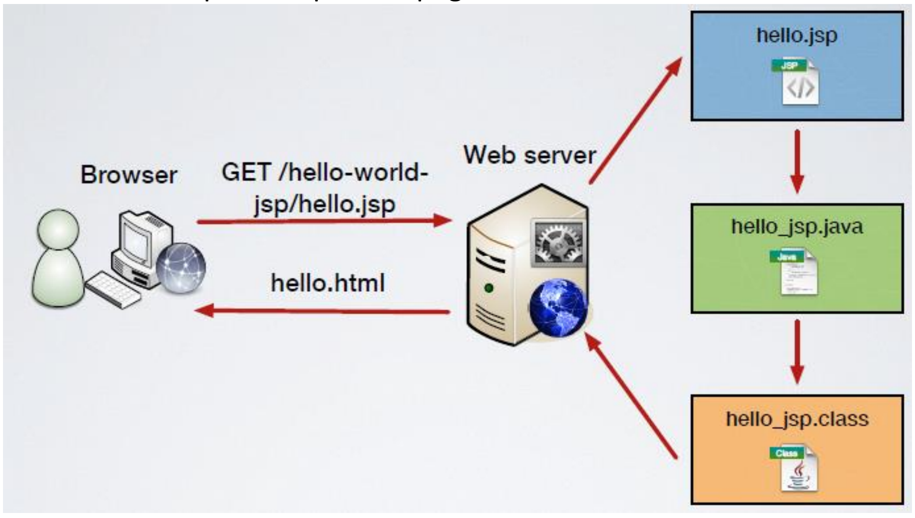
- no support to code in HTML (CSS, JS) since they are just Java strings
- ease of errors
- difficult maintenance and upgrade of the code

JavaServer Pages (JSP) technology provides the means for textual specification of the creation of a dynamic response to a request.

The technology builds on the following concepts:

- **Template Data:** a substantial portion of most dynamic content is fixed or template content. Text or XML fragments are typical template data. JSP technology supports natural manipulation of template data
- **Addition of Dynamic Data:** JSP technology provides a simple, yet powerful, way to add dynamic data to template data
- **Encapsulation of Functionality:** JSP technology provides two related mechanisms for the encapsulation of functionality: **JavaBeans** component architecture, and **tag libraries** delivering custom actions, functions, listener classes, and validation.

Execution of a JSP Page: On first invocation, JSP pages are turned into the "corresponding servlet" and compiled to a Java class. Subsequent invocation will directly refer to the compiled class. You can ask the Web container to pre-compile JSP pages before use.



Components of a JSP page:

- **template text:** it is the static HTML (CSS, …) text
- **directives:** provide global information that is conceptually valid independent of any specific request received by the JSP page
  - `<%@ page … %>`: defines pages dependent attributes
  - `<%@ include … %>`: includes a file
  - `<%@ taglib … %>`: declares a tag library
- **actions:** perform a given operation. The use standard XML syntax <prefix:action>, e.g <jsp:param>

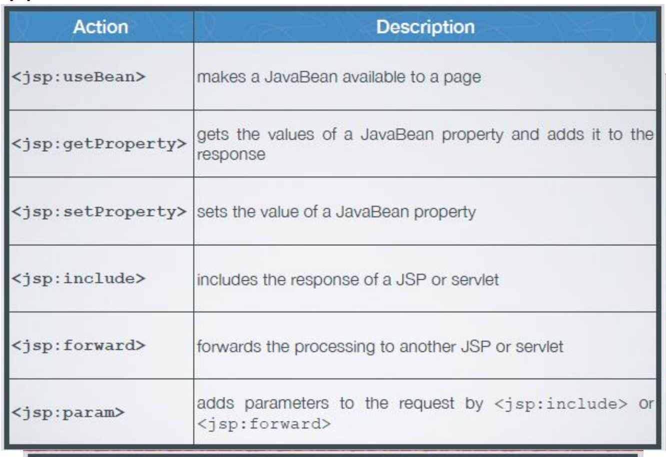
- **standard action:** a set of base action defined in the JSP specification.
- **custom action:** personalized actions to support specific task and collected into tag libraries. The JSTL (JSP Standard Tag Library) is one of such extensions, it is standardized and supports all the typical needs of an applications (conditional instructions, formatting, internationalization, …)

- **scripting:** it is raw Java code to add further flexibility (to be avoided as much as possible)
  - `<% … %>` scriptlet: a fragment of Java code
  - `<%= … %>` expression: embeds the results of a Java expression
  - `<%! … %>` declaration: allows for declaring variables and methods which will be used in the JSP page
- **expression language (EL):** is it a simple language to access data and variable made available from the application.
  ${…}: it contains the expression to be evaluated and executed

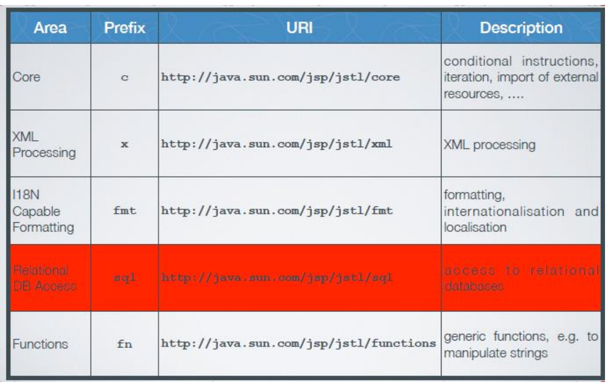
JavaBean is a Java class, providing a reusable software component which follows a specific naming conventions and can thus be manipulated in an applicative framework. For example, component of a GUI framework.
JavaBean conventions:

- it must have a no-argument constructor, to facilitate its instantiation.
- its fields must be exposed through accessor methods which are called: getXXX and setXXX for a generic field name XXX, and isXXX for a boolean field XXX

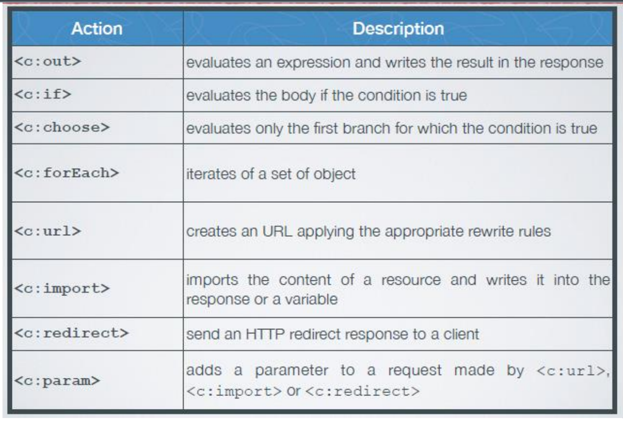
JSP relies on JavaBeans to exchange information among the different components of the application.



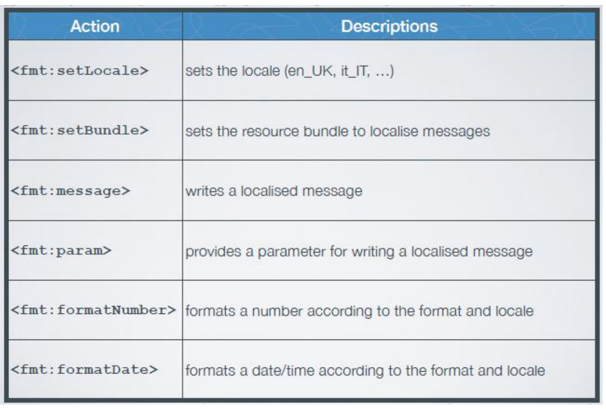
Action	Description
`<jsp:useBean>`	makes a JavaBean available to a page
`<jsp:getProperty>`	gets the values of a JavaBean property and adds it to the response
`<jsp:setProperty>`	sets the value of a JavaBean property
`<jsp:include>`	includes the response of a JSP or servlet
`<jsp:forward>`	forwards the processing to another JSP or servlet
`<jsp:param>`	adds parameters to the request by `<jsp:include>` or `<jsp:forward>`



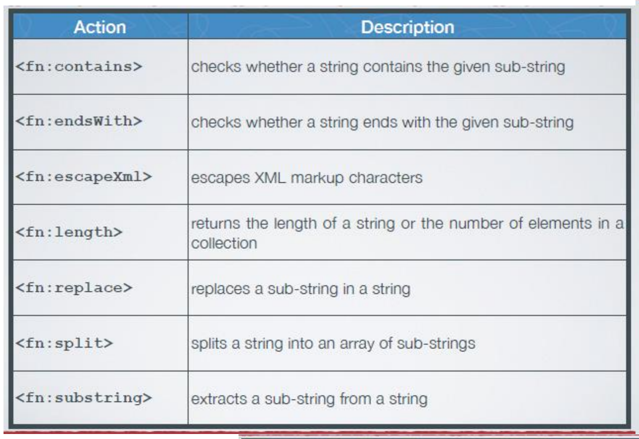
Area	Prefix	URI	Description
Core	c	http://java.sun.com/jsp/jstl/core	conditional instructions, iteration, import of external resources, ….
XML Processing	x	http://java.sun.com/jsp/jstl/xml	XML processing
I18N Capable Formatting	fmt	http://java.sun.com/jsp/jstl/fmt	formatting, internationalisation and localisation
Relational DB Access	sql	http://java.sun.com/jsp/jstl/sql	access to relational databases
Functions	fn	http://java.sun.com/jsp/jstl/functions	generic functions, e.g. to manipulate strings



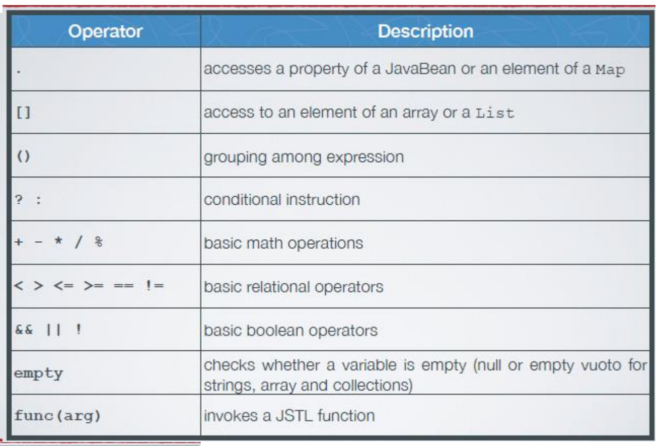
Action	Description
`<c:out>`	evaluates an expression and writes the result in the response
`<c:if>`	evaluates the body if the condition is true
`<c:choose>`	evaluates only the first branch for which the condition is true
`<c:forEach>`	iterates of a set of object
`<c:url>`	creates an URL applying the appropriate rewrite rules
`<c:import>`	imports the content of a resource and writes it into the response or a variable
`<c:redirect>`	send an HTTP redirect response to a client
`<c:param>`	adds a parameter to a request made by `<c:url>`, `<c:import>` or `<c:redirect>`



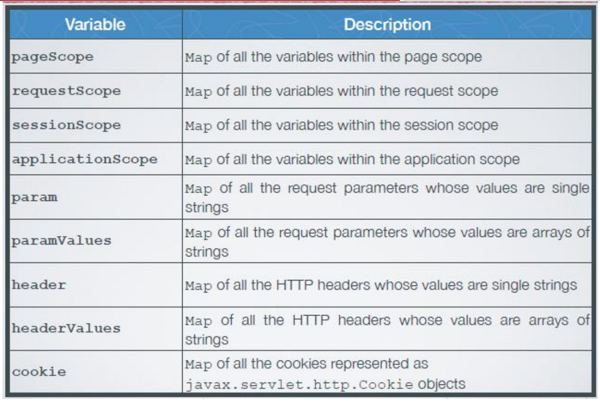
Action	Descriptions
`<fmt:setLocale>`	sets the locale (en_UK, it_IT, …)
`<fmt:setBundle>`	sets the resource bundle to localise messages
`<fmt:message>`	writes a localised message
`<fmt:param>`	provides a parameter for writing a localised message
`<fmt:formatNumber>`	formats a number according to the format and locale
`<fmt:formatDate>`	formats a date/time according to the format and locale



Action	Description
`<fn:contains>`	checks whether a string contains the given sub-string
`<fn:endsWith>`	checks whether a string ends with the given sub-string
`<fn:escapeXml>`	escapes XML markup characters
`<fn:length>`	returns the length of a string or the number of elements in a collection
`<fn:replace>`	replaces a sub-string in a string
`<fn:split>`	splits a string into an array of sub-strings
`<fn:substring>`	extracts a sub-string from a string



Operator	Description
`.`	accesses a property of a JavaBean or an element of a `Map`
`[]`	access to an element of an array or a `List`
`()`	grouping among expression
`? :`	conditional instruction
`+ - * / %`	basic math operations
`< > <= >= == !=`	basic relational operators
`&& || !`	basic boolean operators
`empty`	checks whether a variable is empty (null or empty vuoto for strings, array and collections)
`func(arg)`	invokes a JSTL function

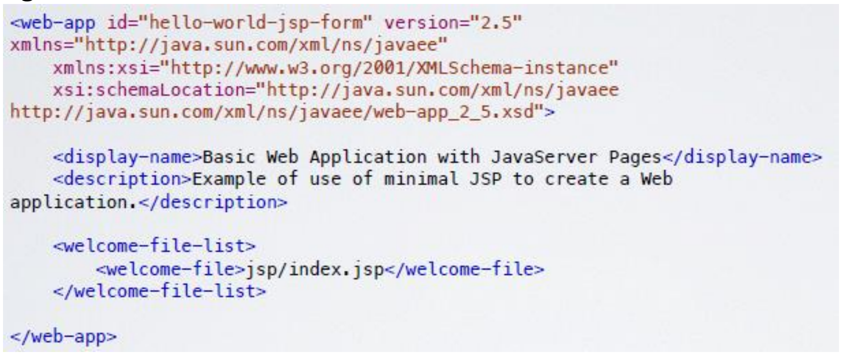


Variable	Description
`pageScope`	`Map` of all the variables within the page scope
`requestScope`	`Map` of all the variables within the request scope
`sessionScope`	`Map` of all the variables within the session scope
`applicationScope`	`Map` of all the variables within the application scope
`param`	`Map` of all the request parameters whose values are single strings
`paramValues`	`Map` of all the request parameters whose values are arrays of strings
`header`	`Map` of all the HTTP headers whose values are single strings
`headerValues`	`Map` of all the HTTP headers whose values are arrays of strings
`cookie`	`Map` of all the cookies represented as `javax.servlet.http.Cookie` objects

EL variables ${…}

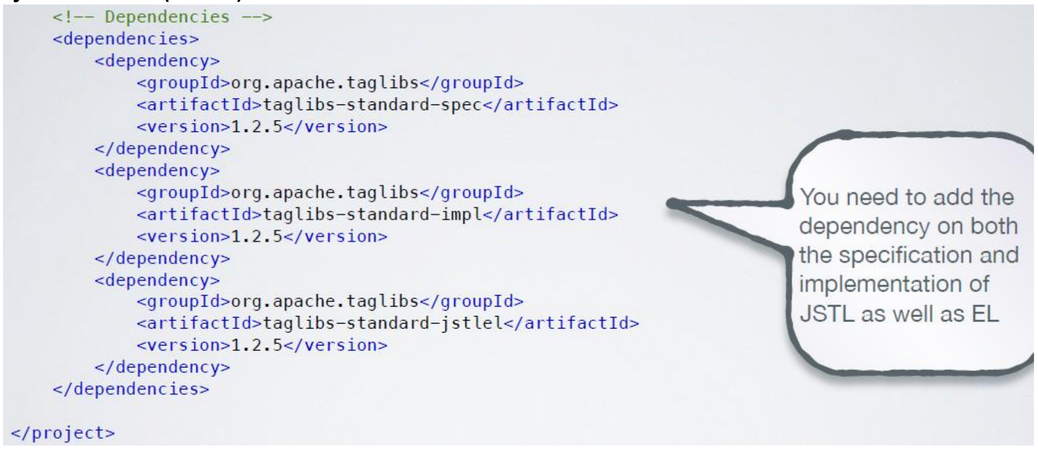
Example: pdf 06.

The web.xml Configuration File:



```xml
<web-app id="hello-world-jsp-form" version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <display-name>Basic Web Application with JavaServer Pages</display-name>
    <description>Example of use of minimal JSP to create a Web
application.</description>

    <welcome-file-list>
        <welcome-file>jsp/index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

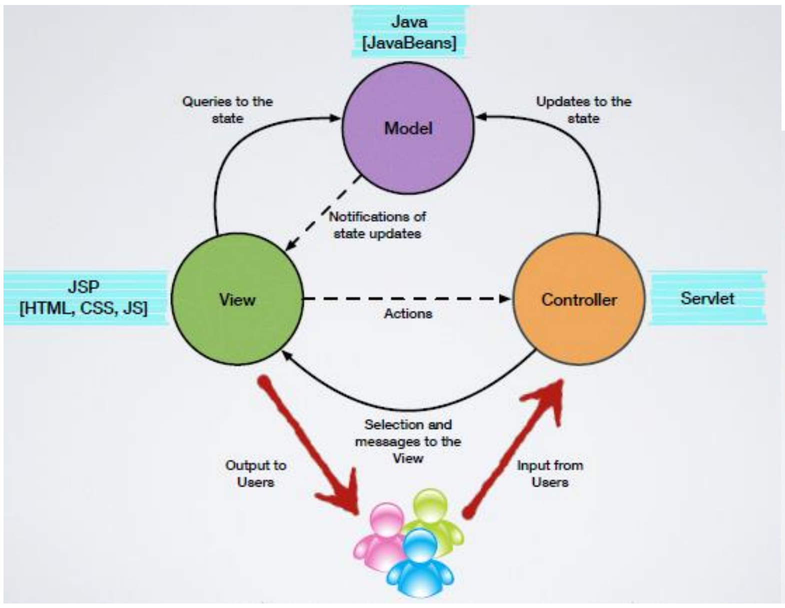
Project Object Model (POM):

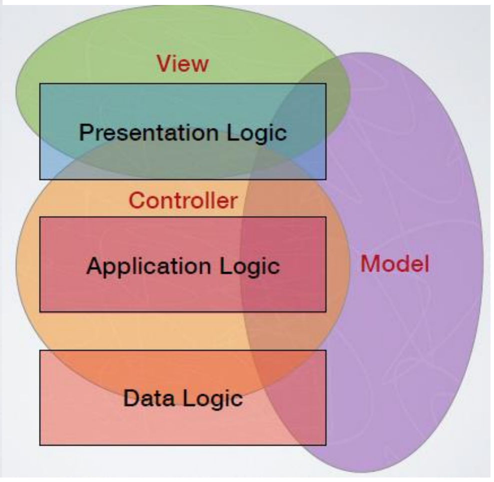


```xml
    <!-- Dependencies -->
    <dependencies>
        <dependency>
            <groupId>org.apache.taglibs</groupId>
            <artifactId>taglibs-standard-spec</artifactId>
            <version>1.2.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.taglibs</groupId>
            <artifactId>taglibs-standard-impl</artifactId>
            <version>1.2.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.taglibs</groupId>
            <artifactId>taglibs-standard-jstlel</artifactId>
            <version>1.2.5</version>
        </dependency>
    </dependencies>

</project>
```

You need to add the dependency on both the specification and implementation of JSTL as well as EL

5.2. Model-View-Controller





6.HTTP and REST

Basic Web Technology

- **HyperText Markup Language (HTML):** the markup language to write Web pages
- **HyperText Transfer Protocol (HTTP):** the application layer protocol which rules the communication between Web clients and Web servers
- **Multipurpose Internet Mail Extensions (MIME):** the media type and the encoding of the exchanged information
- **Uniform Resource Locator (URL):** the way to identify and locate resources on the Web.

6.1. URL

Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.

- **Uniform**: it allows different types of resource identifiers to be used in the same context, even when the mechanisms used to access those resources may differ
  - uniform semantic interpretation of common syntactic conventions
  - consistent introduction of new types of resource identifiers
- **Resource**: is used in a general sense for whatever might be identified, e.g. an electronic document, an image
  - a resource is not necessarily accessible via the Internet; e.g., human beings or books in a library
  - abstract concepts can be resources
- **Identifier**: embodies the information required to distinguish what is being identified from all other things within its scope of identification.

URI are a generic and abstract identification mechanism
Uniform Resource Locator (URL) refer to the subset of URI that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). Example: https://www.rfc-editor.org/rfc/rfc1738.txt
Uniform Resource Name (URN) refer to the subset of using the "urn" scheme (see later on) and with the properties of a "name". the syntactical correctness of a name starting with "urn:" is not sufficient to make it a URN. In order for the name to be a valid URN, the namespace identifier needs to be registered in accordance with the well-defined rules and the remaining parts of the assigned-name portion of the URN need to be generated in accordance with the rules for the registered URN namespace. Example: urn:isbn:978-951-0-18435-6
Internationalized Resource Identifier (IRI) is an extension of the URI syntax to allow for Unicode Characters. Example: https://en.wiktionary.org/wiki/Ῥόδος

URI Syntax

- **scheme**: a name that refers to a specification for assigning identifiers within that scheme; examples include http(s), ftp, mailto, and file
- **two slashes (//)**: required by some schemes and not required by some others
- an **authority** part, comprising:
  - an optional **authentication section** of a user name and password, separated by a colon, followed by an at symbol (@)

    - a "**host**", consisting of either a registered domain name or an IP address
    - an optional **port number**, separated from the hostname by a colon
- a **path**, which contains data, usually organized in hierarchical form, that appears as a sequence of segments separated by slashes
- an optional **query**, separated from the preceding part by a question mark (?), typically consisting of a sequence of attribute=value pairs separated by a delimiter (&)
- an optional **fragment**, separated from the preceding part by a hash (#). The fragment contains a fragment identifier providing direction to a secondary resource, such as a section heading in an article identified by the remainder of the URI
- **Percent-Encoding:** an octet encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing that octet's numeric value. It is used for escaping both reserved characters and non-ASCII characters. E.g. %20 is the percent-encoding for space, %3F for ?, %26 for &, %23 for #, %2F for /, %E0 for à

ASCII Character Encoding

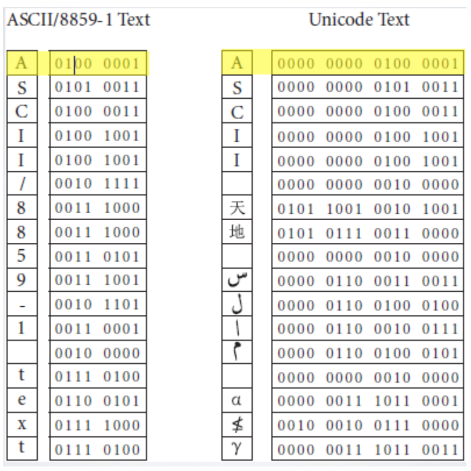
- **ASCII** (American Standard Code for Information Interchange) is a character encoding scheme introduced in 1963 by the American Standards Association
- It uses **7 bits** to represents **128 characters** — control characters, latin alphabet letters (lower and upper cases), numbers, punctuation, some symbols.
- It has been then standardized by ISO in 1972. Since ASCII did not provide a number of characters needed in languages other than English, a number of national variants were made that substituted a few less-used characters with needed ones, leading to incompatibilities.

Extended ASCII:

- include also non-English symbols
- uses 8 bits to encode 256 characters.
- the first 128 characters are the same as in ASCII at 7 bits
- the additional (upper) 128 characters are used to define a set of alternative code tables, e.g. for different European and non-European languages, leading to several compatibility issues
- Extended ASCII is standardized in the ISO 8859 sets of recommendations since 1987.

The Unicode Standard:

- In 1991 the Unicode Consortium (https://home.unicode.org/) developed a new standard to address the compatibility issues among the different ASCII encodings and to develop a single set of characters suitable for all the different alphabets and symbols
- The first versions of Unicode used **16 bits** to represent **65,536 characters** while the more recent versions use **32 bits** to represent up to **4,294,967,296** possible characters. The first 256 characters are in common with the ISO 8859-1 standard
- To save memory, alternative encoding schemes have developed for "packing" Unicode symbols, called **Unicode Transformation Format (UTF)**
    - **UTF-8** is among the most adopted: it uses 8 bits for the characters which are in common with extended ASCII, 16 bits for the new characters added by the first Unicode versions, and 32 bits only when needed to represents the newest characters.
- It has been 30et30Attribut by ISO in 1993 as **Universal Character Set (UCS)**



ASCII/8859-1 Text		Unicode Text	
A	0100 0001	A	0000 0000 0100 0001
S	0101 0011	S	0000 0000 0101 0011
C	0100 0011	C	0000 0000 0100 0011
I	0100 1001	I	0000 0000 0100 1001
I	0100 1001	I	0000 0000 0100 1001
/	0010 1111		0000 0000 0010 0000
8	0011 1000	天	0101 1001 0010 1001
8	0011 1000	地	0101 0111 0011 0000
5	0011 0101		0000 0000 0010 0000
9	0011 1001	س	0000 0110 0011 0011
-	0010 1101	ل	0000 0110 0100 0100
1	0011 0001	ا	0000 0110 0010 0111
	0010 0000	م	0000 0110 0100 0101
t	0111 0100		0000 0000 0010 0000
e	0110 0101	α	0000 0011 1011 0001
x	0111 1000	≴	0010 0010 0111 0000
t	0111 0100	γ	0000 0011 1011 0011

Example of Extended ASCII and Unicode

6.2. MIME

Multipurpose Internet Mail Extensions (MIME) is a standard supporting the encoding of information for e-mail and the Web.

- MIME defines several media types — e.g. text, image, audio — and subtypes — e.g. plain, html, xml for text.
- MIME media types are registered by **IANA (Internet Assigned Numbers Authority)** https://www.iana.org/assignments/media-types/media-types.xhtml
- For each type and subtype it is possible to specify additional information, when needed, such as the charset of a text type.
- MIME defines a set of headers which are used by protocols like SMTP for email and HTTP for the Web to specify the media type, format and encoding of the exchanged content.

(Some) MIME Headers:

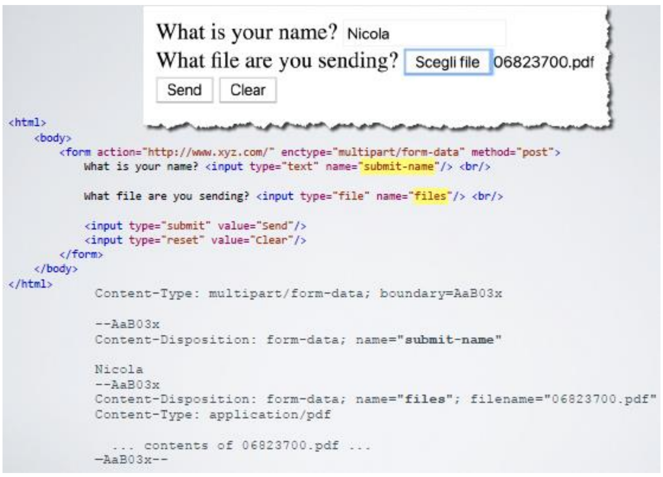
- **MIME-Version:** defines the version of MIME used. Example, `MIME-Version: 1.0`
- **Content-Type:** specifies the nature of the data in the body of an entity by giving media type and subtype identifiers, and by providing auxiliary information that may be required for certain media types. Example, `Content-Type: text/plain; charset=ISO-8859-1`
- **Content-Transfer-Encoding:** defines a set of methods for representing binary data in formats other than ASCII text format. Some possible values are 7bit, 8bit, base64. Example
```
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64
```
- **Content-Disposition:** defines how the content body has to be represented/displayed on the client side. A body part should be marked "inline" if it is intended to be displayed automatically upon display of the message; it can be designated "attachment" to indicate that it is separate from the main body of the mail message, and that their display should not be automatic, but contingent upon some further action of the user. Additional fields like, filename, modification-date, size and so on are available to provide further Information. Example
```
Content-Type: image/jpeg
Content-Disposition:    attachment;    filename=genome.jpeg;
modification-date="Wed, 12 Feb 2020
16:29:51 -0500"; size=9028
```

The `multipart` **media type** represents one or more different sets of data combined in a single body. The body must then contain one or more body parts, each preceded by a boundary delimiter line, and the last one followed by a closing boundary delimiter line.
The `mixed` **subtype** is intended for use when the body parts are independent and need to be bundled in a particular order, e.g. content of an email and attachments.
The `alternative` **subtype** is syntactically identical to multipart/mixed but the semantics is different. In particular, each of the body parts is an "alternative" version of the same information, e.g. content of an email in plain text and html version.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=frontier

This is a message with multiple parts in MIME format.
--frontier
Content-Type: text/plain

This is the body of the message.
--frontier
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGh0bWw+CiAgPGhlYWQ+CiAgPC9oZWFkPgogIDxib2R5PgogICAgPHA+VghpcyBpcyB0aGUg
Ym9keSBvZiB0aGUgbWVzc2FnZS48L3A+CiAgPC9ib2R5Pgo8L2h0bWw+Cg==
--frontier-
```

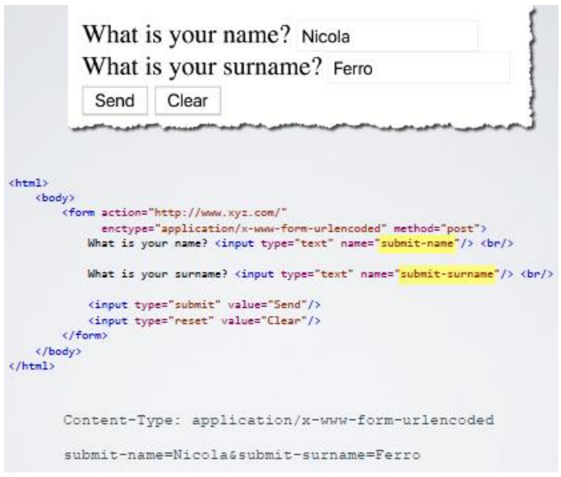
Sending Web forms and Uploading Files

- The `multipart/form-data` media type allows for uploading files and sending fields from Web forms. More effective encoding of (large) binary files, with the same mechanisms as the `multipart` type in general, but too much header overhead to just send a few form fields.
- The `application/x-www-form-urlencoded` media types allows for sending field from Web forms.
    - all the `name=value` pairs are concatenated into a single string, separated by `&`, and the string is then percent-encoded. They can also be appended as a `query` part in a URI, instead of being sent as content body
    - not suitable for sending (large) binary files, due to huge encoding overhead, but effective for a few form fields.


What is your name? Nicola
What file are you sending? Scegli file 06823700.pdf
Send
Clear

```
<html>
    <body>
        <form action="http://www.xyz.com/" enctype="multipart/form-data" method="post">
            What is your name? <input type="text" name="submit-name"/> <br/>

            What file are you sending? <input type="file" name="files"/> <br/>

            <input type="submit" value="Send"/>
            <input type="reset" value="Clear"/>
        </form>
    </body>
</html>
```

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
Content-Disposition: form-data; name="submit-name"

Nicola
--AaB03x
Content-Disposition: form-data; name="files"; filename="06823700.pdf"
Content-Type: application/pdf

  ... contents of 06823700.pdf ...
—AaB03x--
```



```
<html>
    <body>
        <form action="http://www.xyz.com/"
              enctype="application/x-www-form-urlencoded" method="post">
            What is your name? <input type="text" name="submit-name"/> <br/>

            What is your surname? <input type="text" name="submit-surname"/> <br/>

            <input type="submit" value="Send"/>
            <input type="reset" value="Clear"/>
        </form>
    </body>
</html>
```

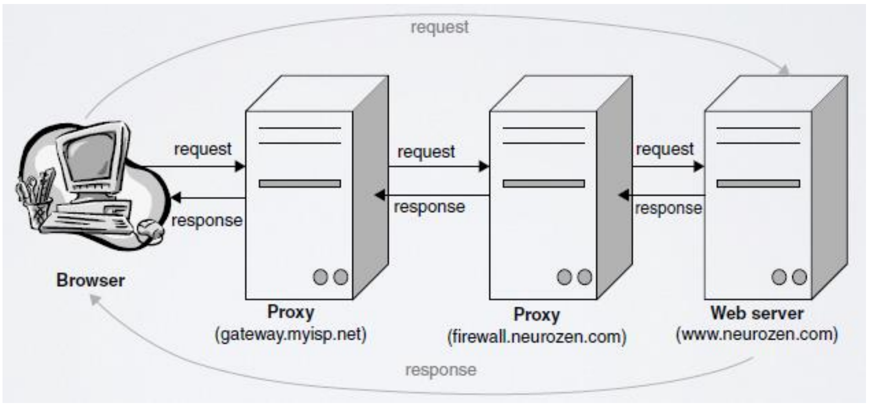
```
Content-Type: application/x-www-form-urlencoded

submit-name=Nicola&submit-surname=Ferro
```

multipart/form-data Example. Application/x-www-form-urlencoded Example

6.3. HTTP 1.1

Hypertext Transfer Protocol (HTTP) is a textual request-response protocol where clients and servers exchange messages constituted by an header and an optional body.

- is a stateless protocol, i.e. each request-response is independent and neither the client nor the server has to keep trace of the exchanged messages. This simplifies the implementation of the protocol and makes it more scalable.
- s designed to favour the use of intermediaries or proxies, typically for caching or security purposes.



HTTP Request:

```
METHOD /path-to-resource HTTP/version-number
Header-Name-1: value
Header-Name-2: value
[ optional request body ]
```

- **Request line:**
  - **METHOD:** one of the HTTP methods, e.g. `GET` or `POST`
  - `/path-to-resource`: the path part of an URI, including query and fragment if available
  - `HTTP/version-number`: the version of HTTP used by the clients
- **Headers**: a set of headers, separated by the body of the request by a blank line, often called `<CR><LF>`
- **Body:** an optional content to be sent to the server, followed by a blank line

```
GET /sj/index.html HTTP/1.1
Host: www.mywebsite.com
```

HTTP Response:

```
HTTP/version-number status-code message
Header-Name-1: value
Header-Name-2: value
[ response body ]
```

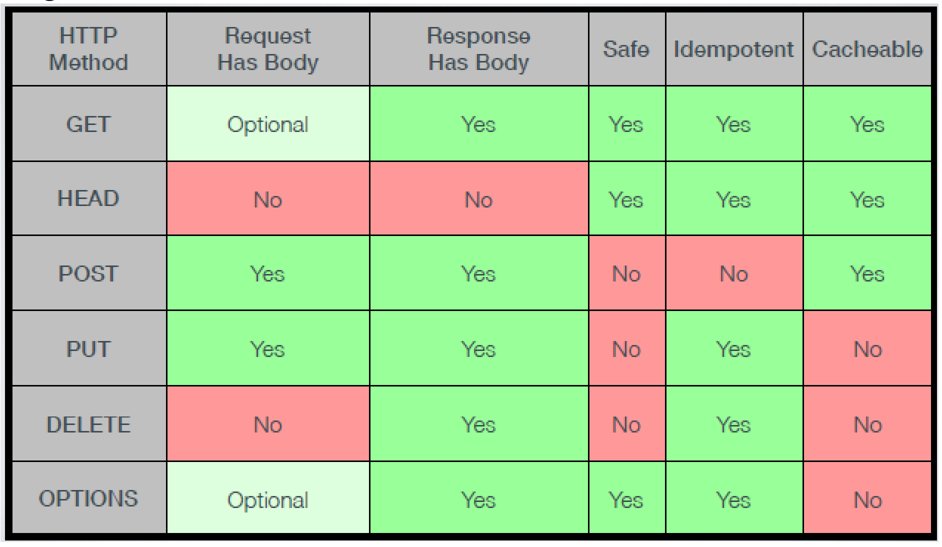
- **Status line:**
  - `HTTP/version-number`: the version of HTTP used by the clients
  - `status-code`: 3-digit integer result code of the attempt to understand and satisfy the request.
  - `message`: is intended to give a short textual description of the status code
- **Headers**: a set of headers, separated by the body of the request by a blank line
- **Body**: an optional content sent by the server, followed by a blank line

HTTP Request Methods:

- **GET**: means retrieve whatever information is identified by the request URI
- **POST**: is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the request URI
- **PUT**: requests that the enclosed entity be stored under the supplied request URI. If the request URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server. If the request URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI
- **DELETE**: requests that the origin server deletes the resource identified by the request URI
- **HEAD**: is identical to GET except that the server must not return a message-body in the response. The meta-information contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request
- **OPTIONS**: represents a request for information about the communication options available on the request/response chain identified by the Request-URI.

Properties of HTTP Methods:

- **Safe methods:** if their defined semantics is essentially read-only; i.e., the client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource; in other words, they should not have side effects
    - The purpose of distinguishing between safe and unsafe methods is to allow automated retrieval processes (spiders) and cache performance optimization (pre-fetching) to work without fear of causing harm.
    - safe: **GET**, HEAD, OPTIONS; not safe: DELETE, POST, PUT
- **Idempotent methods:** if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request.
    - Idempotent methods are distinguished because the request can be repeated automatically if a communication failure occurs before the client is able to read the server's response.
    - idempotent: **GET**, HEAD, OPTION (safe methods), DELETE, PUT; not idempotent: POST
- **Cacheable methods**: indicate that responses to them are allowed to be stored for future reuse; in general, safe methods are defined as cacheable.



HTTP Method	Request Has Body	Response Has Body	Safe	Idempotent	Cacheable
GET	Optional	Yes	Yes	Yes	Yes
HEAD	No	No	Yes	Yes	Yes
POST	Yes	Yes	No	No	Yes
PUT	Yes	Yes	No	Yes	No
DELETE	No	Yes	No	Yes	No
OPTIONS	Optional	Yes	Yes	Yes	No

HTTP Response Status Codes: The first digit of the Status-Code defines the class of response.

- 1xx: **Informational** – Request received, continuing process.
  `HTTP/1.1 101 Switching Protocols`
- 2xx: **Success** – The action was successfully received, understood, and accepted.
  `HTTP/1.1 200 OK`
- 3xx: **Redirection** – Further action must be taken in order to complete the request
  ```
  HTTP/1.1 301 Moved Permanently
  Location: http://www.dei.unipd.it/
  ```
- 4xx: **Client Error** – The request contains bad syntax or cannot be fulfilled.
  `HTTP/1.1 404 Not Found`
- 5xx: **Server Error** – The server failed to fulfill an apparently valid request.
  `HTTP/1.1 500 Internal Server Error`

(Some) HTTP Request Headers:

- **Accept:** specifies response media types that are acceptable
  `Accept: text/plain, text/plain, image/*`
- **Accept-Charset:** indicates what charsets are acceptable in textual response content
  `Accept-Charset: iso-8859-5, UTF-8`
- **Accept-Encoding:** indicates what response content-codings are acceptable in the response. An "identity" token is used as a synonym for "no encoding" in order to communicate when no encoding is preferred.
  `Accept-Encoding: compress, gzip`
- **Accept-Language:** indicates the set of natural languages that are preferred in the response
  `Accept-Language: it, da, en-gb`
  for language codes see, e.g., ISO 639-1 (2002). Codes for the representation of names of languages – Part 1: Alpha-2 code. Recommendation ISO 639-1:2002.
- **Referer:** allows the user agent to specify a URI reference for the resource from which the target URI was obtained, i.e. the "referrer", though the field name is misspelled. The 35et35Attr header field allows servers to generate backlinks to other resources for simple analytics, logging, optimized caching, etc.
  `Referer: http://www.example.org/hypertext/Overview.html`
- **User-Agent:** contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor responses to avoid particular user agent limitations, and for analytics regarding browser or operating system use
  `User-Agent: CERN-LineMode/2.15 libwww/2.17b3`

(Some) HTTP Response Headers:

- **Content-Type:** indicates the MIME media type of the associated representation
  `Content-Type: text/html; charset=ISO-8859-4`
- **Content-Encoding:** indicates what content codings have been applied to the representation, typically compression
  `Content-Encoding: gzip`
- **Content-Language:** describes the natural language(s) of the intended audience for the representation
  `Content-Language: it, en`
- **Content-Length:** provides the anticipated size, as a decimal number of octets, for a potential payload body
  `Content-Length: 8092`
- **Allow**: lists the set of methods advertised as supported by the target resource

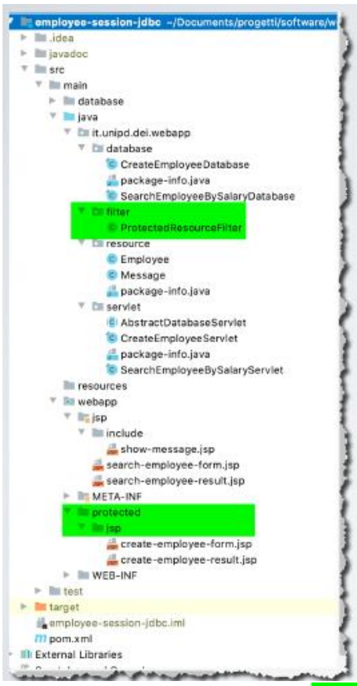
```
Allow: GET, HEAD, PUT
```

- **Server**: contains information about the software used by the origin server to handle the request, which is often used by clients to help identify the scope of reported interoperability problems, to work around or tailor requests to avoid particular server limitations, and for analytics regarding server or operating system use
  ```
  Server: CERN/3.0 libwww/2.17
  ```
- **Date**: represents the date and time at which the message was originated
  ```
  Date: Tue, 15 Nov 1994 08:12:31 GMT
  ```
- **Last-Modified:** provides a timestamp indicating the date and time at which the origin server believes the selected representation was last modified
  ```
  Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
  ```

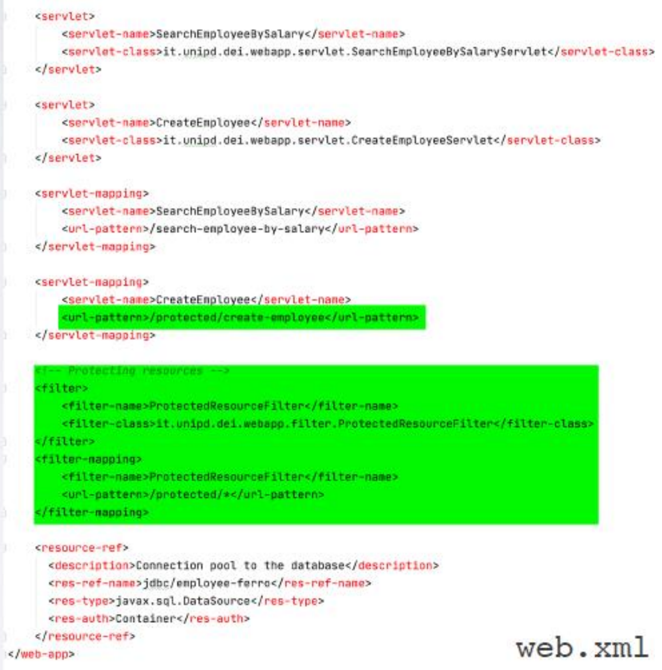
Authentication and Authorization:

- To access secured resources, the client has to send authentication information by using the `Authorization` header, which supports various authentication mechanisms
- The simplest authentication mechanism is the Basic one where user name and password are concatenated with a colon (😊 and encoded base64
  - example for a user with user name `36et36At` and password `ferro` — the authentication string `36et36At:ferro` becomes
    ```
    bmljb2xhOmZlcnJv in base64
    GET /secured-resource/pippo.jpg
    Authorization: Basic bmljb2xhOmZlcnJv
    ```
  - not that with basic authentication, user credentials are just encoded but not encrypted. So, this mechanism does not guarantee confidentiality, if not used together with some other technique such as https
- If the client tries to access secured resources without providing authentication credentials (or providing the wrong ones), the server replies with an authentication challenge returning the status code 401 Unauthorized and setting the WWW-Authenticate header to specify the expected authentication mechanism.
  - example of authentication challenge
    ```
    HTTP/1.1 401 Unauthorized
    WWW-Authenticate: Basic realm="Webapp"
    ```
  - the `realm` allows the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database.
  - Web browsers reply to an authentication challenge by showing a username/password windows to enter user credentials. Authentication is required only the first time a real is accessed because, after the first successful authentication, Web browser automatically add the `Authorization` header to all subsequent request under the same realm.

Session JDBC: filter class check if there is the authentication. ProtectedResourceFilter.

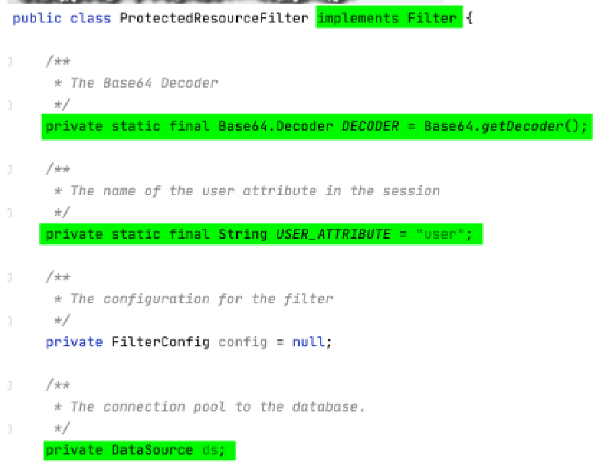


```
employee-session-jdbc ~/Documents/progetti/software/w
  .idea
  javadoc
  src
    main
      database
      java
        it.unipd.dei.webapp
          database
            CreateEmployeeDatabase
            package-info.java
            SearchEmployeeBySalaryDatabase
          filter
            ProtectedResourceFilter
          resource
            Employee
            Message
            package-info.java
          servlet
            AbstractDatabaseServlet
            CreateEmployeeServlet
            package-info.java
            SearchEmployeeBySalaryServlet
      resources
      webapp
        jsp
          include
            show-message.jsp
          search-employee-form.jsp
          search-employee-result.jsp
        META-INF
        protected
          jsp
            create-employee-form.jsp
            create-employee-result.jsp
        WEB-INF
    test
  target
  employee-session-jdbc.iml
  pom.xml
External Libraries
```

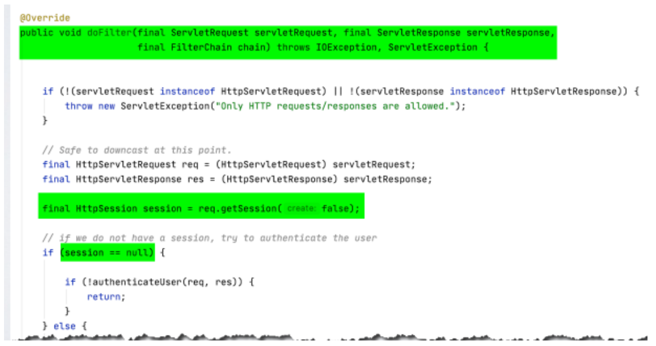


```xml
<servlet>
    <servlet-name>SearchEmployeeBySalary</servlet-name>
    <servlet-class>it.unipd.dei.webapp.servlet.SearchEmployeeBySalaryServlet</servlet-class>
</servlet>

<servlet>
    <servlet-name>CreateEmployee</servlet-name>
    <servlet-class>it.unipd.dei.webapp.servlet.CreateEmployeeServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>SearchEmployeeBySalary</servlet-name>
    <url-pattern>/search-employee-by-salary</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>CreateEmployee</servlet-name>
    <url-pattern>/protected/create-employee</url-pattern>
</servlet-mapping>

<!-- Protecting resources -->
<filter>
    <filter-name>ProtectedResourceFilter</filter-name>
    <filter-class>it.unipd.dei.webapp.filter.ProtectedResourceFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ProtectedResourceFilter</filter-name>
    <url-pattern>/protected/*</url-pattern>
</filter-mapping>

  <resource-ref>
   <description>Connection pool to the database</description>
   <res-ref-name>jdbc/employee-ferro</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

web.xml



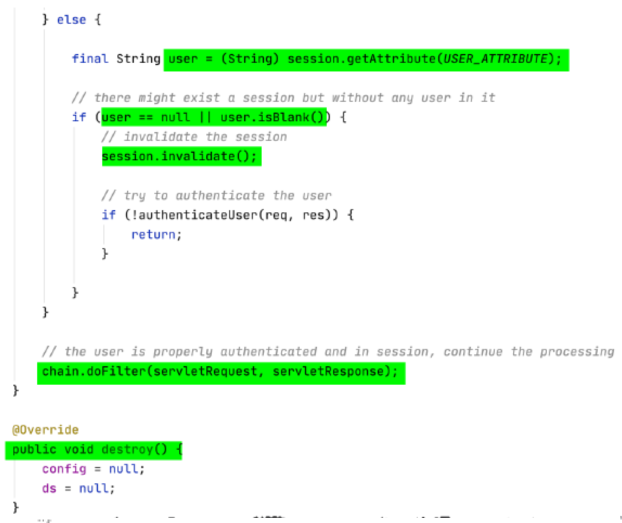
```java
public class ProtectedResourceFilter implements Filter {

    /**
     * The Base64 Decoder
     */
    private static final Base64.Decoder DECODER = Base64.getDecoder();

    /**
     * The name of the user attribute in the session
     */
    private static final String USER_ATTRIBUTE = "user";

    /**
     * The configuration for the filter
     */
    private FilterConfig config = null;

    /**
     * The connection pool to the database.
     */
    private DataSource ds;
```



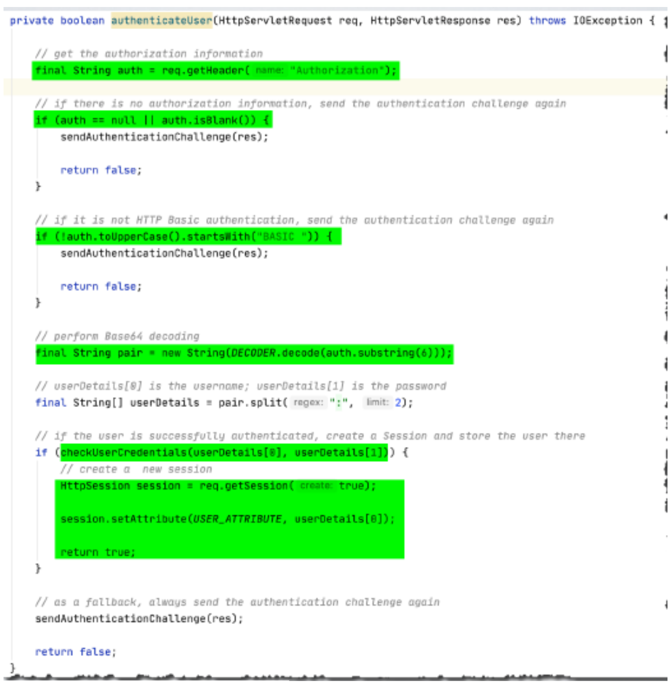
```java
@Override
public void doFilter(final ServletRequest servletRequest, final ServletResponse servletResponse,
        final FilterChain chain) throws IOException, ServletException {

    if (!(servletRequest instanceof HttpServletRequest) || !(servletResponse instanceof HttpServletResponse)) {
        throw new ServletException("Only HTTP requests/responses are allowed.");
    }

    // Safe to downcast at this point.
    final HttpServletRequest req = (HttpServletRequest) servletRequest;
    final HttpServletResponse res = (HttpServletResponse) servletResponse;

    final HttpSession session = req.getSession( create: false);

    // if we do not have a session, try to authenticate the user
    if (session == null) {

        if (!authenticateUser(req, res)) {
            return;
        }
    } else {
```

```java
public class ProtectedResourceFilter implements Filter {

    /**
     * The Base64 Decoder
     */
    private static final Base64.Decoder DECODER = Base64.getDecoder();

    /**
     * The name of the user attribute in the session
     */
    private static final String USER_ATTRIBUTE = "user";

    /**
     * The configuration for the filter
     */
    private FilterConfig config = null;

    /**
     * The connection pool to the database.
     */
    private DataSource ds;
```

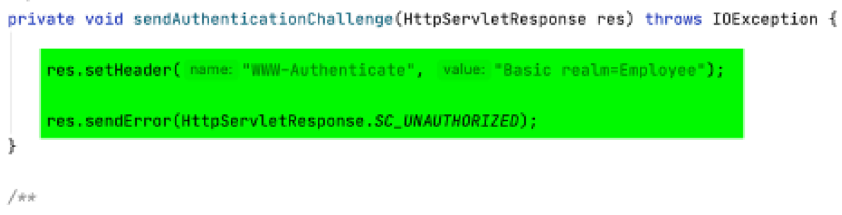
```java
@Override
public void doFilter(final ServletRequest servletRequest, final ServletResponse servletResponse,
        final FilterChain chain) throws IOException, ServletException {

    if (!(servletRequest instanceof HttpServletRequest) || !(servletResponse instanceof HttpServletResponse)) {
        throw new ServletException("Only HTTP requests/responses are allowed.");
    }

    // Safe to downcast at this point.
    final HttpServletRequest req = (HttpServletRequest) servletRequest;
    final HttpServletResponse res = (HttpServletResponse) servletResponse;

    final HttpSession session = req.getSession( create: false);

    // if we do not have a session, try to authenticate the user
    if (session == null) {

        if (!authenticateUser(req, res)) {
            return;
        }
    } else {
```



```
    } else {

        final String user = (String) session.getAttribute(USER_ATTRIBUTE);

        // there might exist a session but without any user in it
        if (user == null || user.isBlank()) {
            // invalidate the session
            session.invalidate();

            // try to authenticate the user
            if (!authenticateUser(req, res)) {
                return;
            }

        }
    }

    // the user is properly authenticated and in session, continue the processing
    chain.doFilter(servletRequest, servletResponse);
}

@Override
public void destroy() {
    config = null;
    ds = null;
}
```



```
private boolean authenticateUser(HttpServletRequest req, HttpServletResponse res) throws IOException {

    // get the authorization information
    final String auth = req.getHeader( name: "Authorization");

    // if there is no authorization information, send the authentication challenge again
    if (auth == null || auth.isBlank()) {
        sendAuthenticationChallenge(res);

        return false;
    }

    // if it is not HTTP Basic authentication, send the authentication challenge again
    if (!auth.toUpperCase().startsWith("BASIC ")) {
        sendAuthenticationChallenge(res);

        return false;
    }

    // perform Base64 decoding
    final String pair = new String(DECODER.decode(auth.substring(6)));

    // userDetails[0] is the username; userDetails[1] is the password
    final String[] userDetails = pair.split( regex: ":",  limit: 2);

    // if the user is successfully authenticated, create a Session and store the user there
    if (checkUserCredentials(userDetails[0], userDetails[1])) {
        // create a  new session
        HttpSession session = req.getSession( create: true);

        session.setAttribute(USER_ATTRIBUTE, userDetails[0]);

        return true;
    }

    // as a fallback, always send the authentication challenge again
    sendAuthenticationChallenge(res);

    return false;
}
```



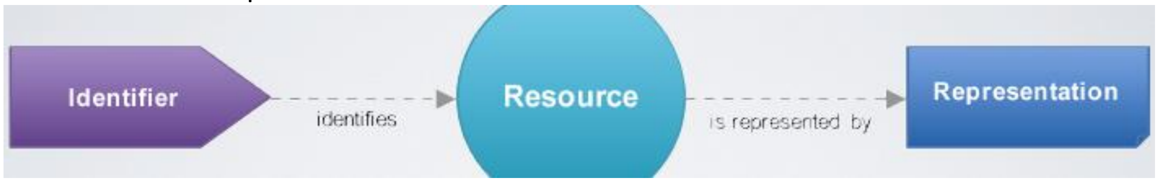
```
private void sendAuthenticationChallenge(HttpServletResponse res) throws IOException {

    res.setHeader( name: "WWW-Authenticate",  value: "Basic realm=Employee");

    res.sendError(HttpServletResponse.SC_UNAUTHORIZED);
}

/**
```

6.4. The REST architectural paradigm

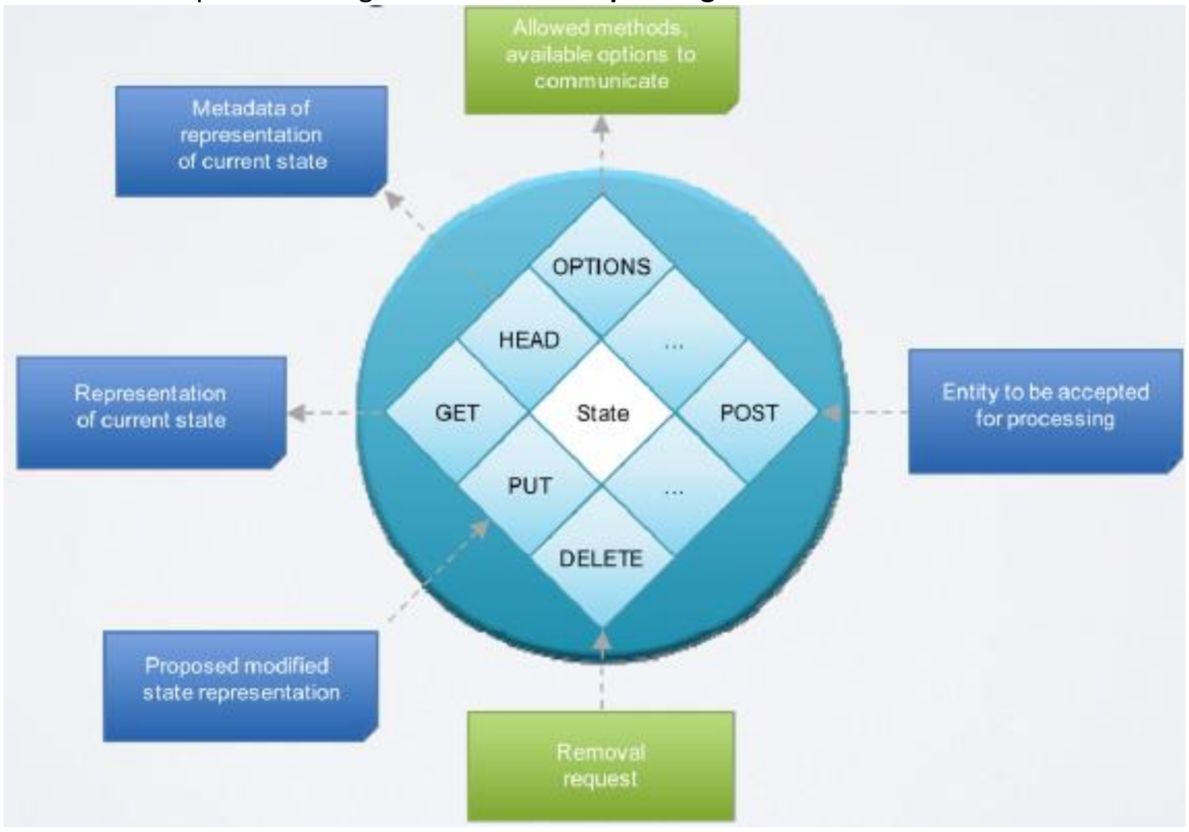
REST is an architectural paradigm which applies the architectural principles of the Web to Web services.
REST relies on a network of Web resources where users proceed in the application by following links (state transitions) which provide the representation of the next resource (new state) to them.
Features: Simplicity, state-less, scalability

Resource :whatever has identity

- have a **state** which can change over time.
- have an **identifier (URI)** which is unique and global.
- can transfer a representation of their state.



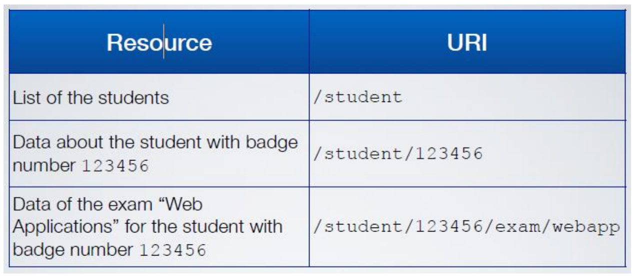
Representation: upon request, a resource may transfer a representation of its state to a client resources expose a uniform interface for their management.
Stateless: each request between client and server must contain all the information needed to understand the request. Messages must be **self-explaining.**


Allowed methods, available options to communicate
Metadata of representation of current state
OPTIONS
HEAD
...
Representation of current state
GET
State
POST
Entity to be accepted for processing
PUT
...
DELETE
Proposed modified state representation
Removal request

HTTP is a stateless protocol

- provides an uniform interface to access resources, i.e. the HTTP methods which have a well defined semantics: GET, POST, PUT, DELETE
- requests/response rely on headers/bodies which are self-explaining

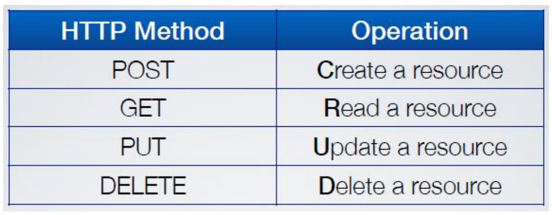
REST: Example of Resources and URIs



Resource	URI
List of the students	`/student`
Data about the student with badge number `123456`	`/student/123456`
Data of the exam “Web Applications” for the student with badge number `123456`	`/student/123456/exam/webapp`

- Each resource has a unique **identifier**, i.e. an URI, which has to be descriptive Enough.
- REST relies on URI templates to specify how resources are identified. `/student/{badge}/exam/{id}`

REST and HTTP Request Methods

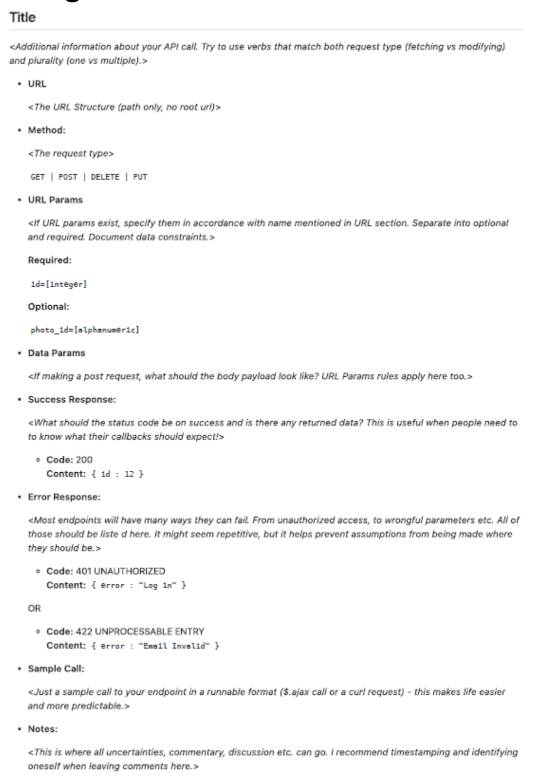


HTTP Method	Operation
POST	Create a resource
GET	Read a resource
PUT	Update a resource
DELETE	Delete a resource

REST: Design Principles

- Identify all the resources which have to be exposed
- Create a URI for each resource, preferably using nouns and verbs
- Determine which HTTP request methods are needed for each resource
- Link resource and “unveil” information by following links
- Specify the format of representation of a resource, possibly using a schema
- Accurately document all the services

Documenting a REST API:



Title

<Additional information about your API call. Try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple).>

- **URL**

  <The URL Structure (path only, no root url)>

- **Method:**

  <The request type>

  `GET` | `POST` | `DELETE` | `PUT`

- **URL Params**

  <If URL params exist, specify them in accordance with name mentioned in URL section. Separate into optional and required. Document data constraints.>

  Required:

  `id=[integer]`

  Optional:

  `photo_id=[alphanumeric]`

- **Data Params**

  <If making a post request, what should the body payload look like? URL Params rules apply here too.>

- **Success Response:**

  <What should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect!>

  - **Code:** 200
    Content: `{ id : 12 }`

- **Error Response:**

  <Most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be liste d here. It might seem repetitive, but it helps prevent assumptions from being made where they should be.>

  - **Code:** 401 UNAUTHORIZED
    Content: `{ error : "Log in" }`

  OR

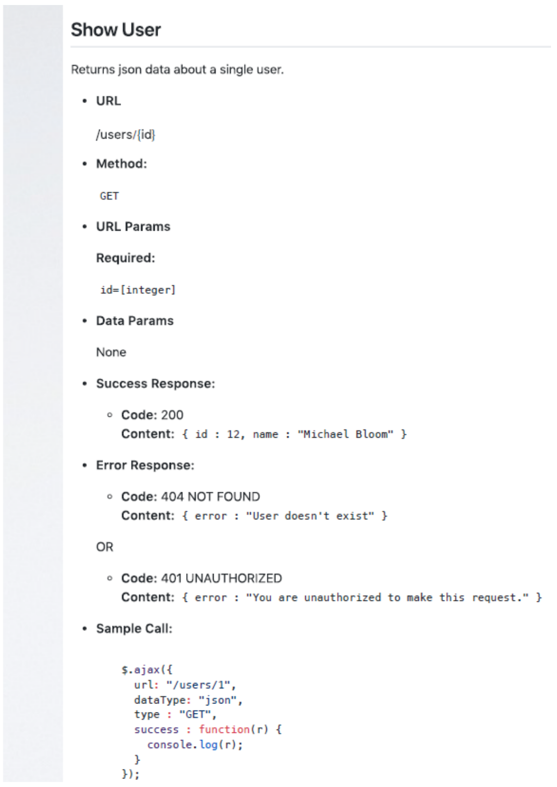
  - **Code:** 422 UNPROCESSABLE ENTRY
    Content: `{ error : "Email Invalid" }`

- **Sample Call:**

  <Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) - this makes life easier and more predictable.>

- **Notes:**

  <This is where all uncertainties, commentary, discussion etc. can go. I recommend timestamping and identifying oneself when leaving comments here.>



Show User

Returns json data about a single user.

- **URL**

  /users/:id

- **Method:**

  `GET`

- **URL Params**

  Required:

  `id=[integer]`

- **Data Params**

  None

- **Success Response:**

  - **Code:** 200
    Content: `{ id : 12, name : "Michael Bloom" }`

- **Error Response:**

  - **Code:** 404 NOT FOUND
    Content: `{ error : "User doesn't exist" }`

  OR

  - **Code:** 401 UNAUTHORIZED
    Content: `{ error : "You are unauthorized to make this request." }`

- **Sample Call:**

```
$.ajax({
  url: "/users/1",
  dataType: "json",
  type : "GET",
  success : function(r) {
    console.log(r);
  }
});
```

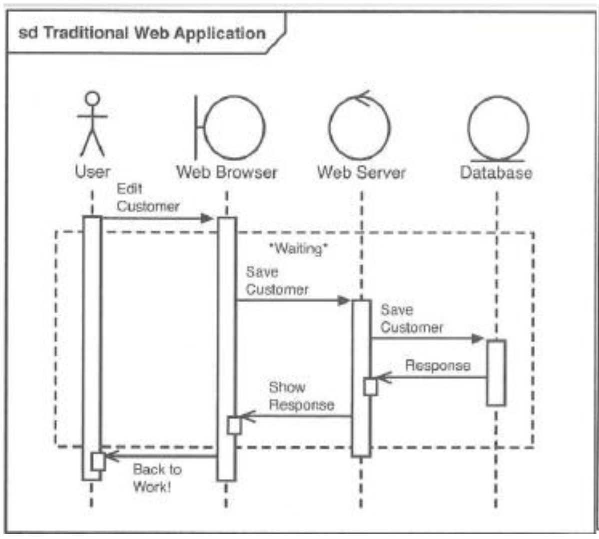
6.5. AJAX

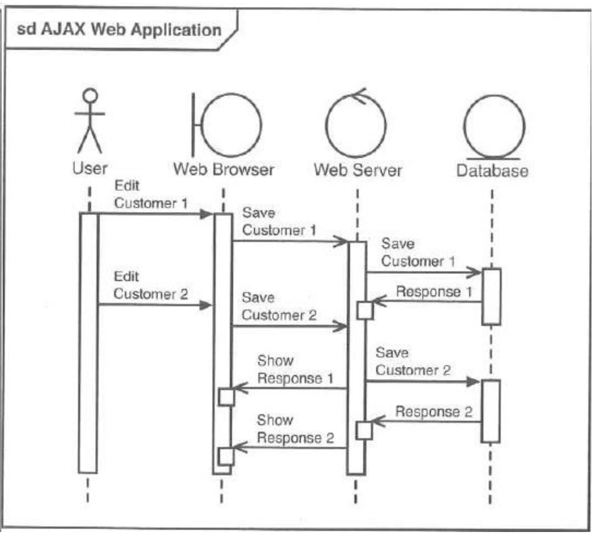
Traditional (Web 1.0) Applications:

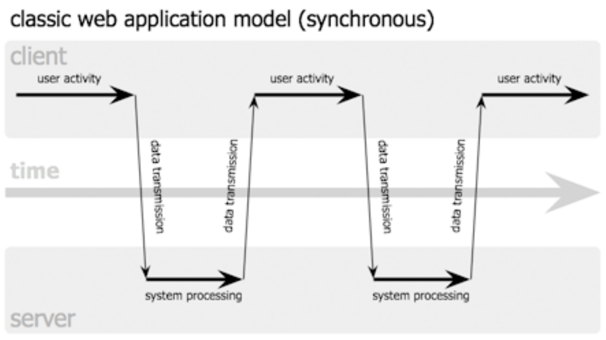
- the user interacts with the browser interface
- the interaction generates requests from the client to the server
- the server processes the requests and returns a new HTML pages
- the browser renders the page to the user

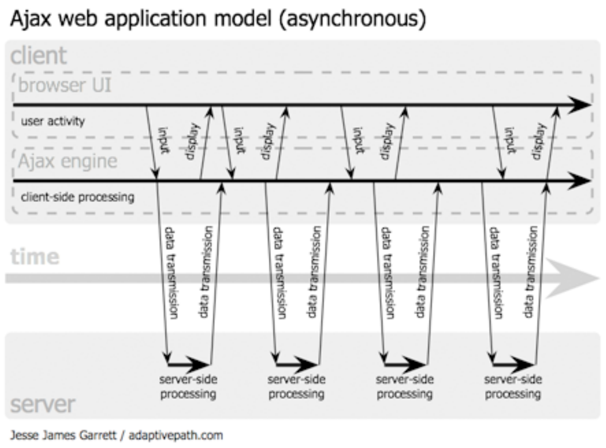
Web 2.0 Applications (AJAX):

- the user interacts with the browser interface
- one or more requests are sent (asynchronously) to the server to react to the user interaction
- the Web server processes the requests and returns XML/JSON/HTML
- the browser updates the corresponding part of the page with the information received back from the server as it arrives









7. Markup Languages

7.1. Markup languages

The Markup is part of our writing system
The markup is not part of the text or content of the expression, but tells us something about it
The digital format allows us to develop markup technologies, beyond those of traditional writing systems, geared towards the automatic processing of information.

Types of Markup:

- **No markup:** typical of ancient writing systems, e.g. scriptio continua or Boustrophedon.
- **Punctuational**: consists of the use of a closed set of marks to provide primarily syntactic information about written utterances
- **Presentational**: deals with the actual laying out contents on a page. It includes horizontal and vertical spacing, folios, page breaks, enumeration of lists and notes, and a host of ad hoc symbols and devices
- **Procedural**: consists of commands indicating how text should be formatted
- **Descriptive**: defines the type or class of the content, e.g. to indicate its intended use
- **Referential**: refers to entities external to the document and is replaced by those entities during processing
- **Meta-markup:** provides the means for controlling the interpretation of markup and for extending the vocabulary of descriptive markup languages.

Standard Generalized Markup Language (SGML)

Features

- it is a descriptive and referential markup language
- it is a meta-markup language and the languages derived from SGML are called
- applications
- introduces the concept of document type and Document Type Definition (DTD), i.e. the mechanism by which the tags and structure of derived languages are defined.

History

- it derives from GML (Generalized Markup Language) developed by Charles Goldfarb, Edward Mosher e Raymond Lorie at IBM in 1974
- it becomes an ANSI (American National Standard Institute) standard in 1983
- it becomes and ISO (International Organization for Standardization) standard in 1986

7.2. HTML

HTML4:

- HTML is a markup language to create hypertextual Web pages
- It defines the **structure/content** and the **presentation** of a Web page
- It is an application of SGML
- is Procedural, descriptive, referential

Problems of HTML4:

- **Loose code parsing:** parsing heuristics for missing tags, swapped tags, … which cause inconsistencies and incompatibilities among browsers
- Lack of separation between content and presentation
    - difficulty to reuse content in different contexts (desktop, mobile) or for different purposes
    - abuse of the semantics of the tags just to obtain a presentation effect, e.g. <h1> or <table>
    - Cascading Style Sheets (CSS) have introduced to overcome these issues
- Lack of support for semantic description of content
- HTML 4.0 it is not a meta-markup language

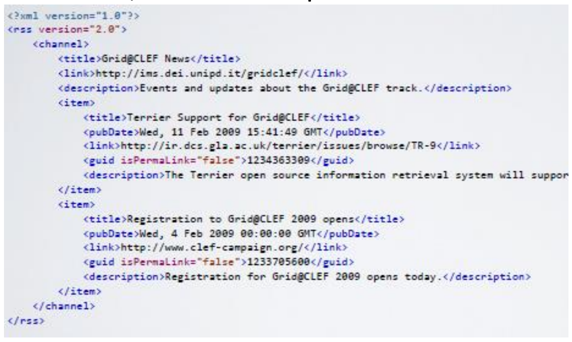
HTML5: It is a re-design of the HTML4 language to clearly separate the structure/content from the presentation of a Web page. Some new features:

- tighter integration with CSS (in charge of the presentation) and JavaScript (in charge of the interaction)
- tags that describe parts of a document, e.g. navigation elements, articles, sections, headers, and footers
- new form elements, e.g. several new versions of the <input> element, allowing users to pick colors, numbers, e-mail addresses, and dates
- native support for audio and video streams
- improved graphics support with Scalable Vector Graphics (SVG) and/or canvas
- a new local storage mechanism, the Web Store, which improves over cookies
- is procedural, descriptive, referential

7.3. XML

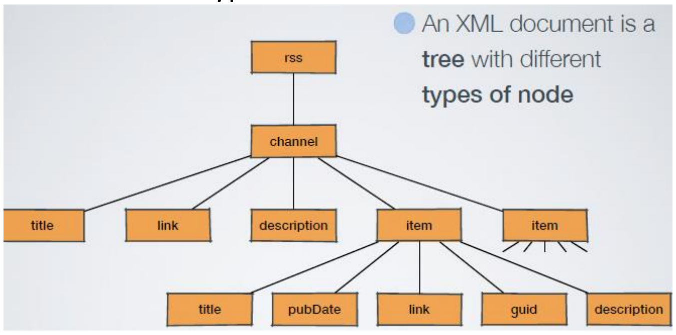
eXtensible Markup Language (XML): It is a markup language for representing and exchanging information, geared towards interoperability among distributed systems

- data and information represented in XML is considered semi-structured, i.e. inbetween the rigid structured approach of databases and the unstructured approach of full text documents
- It is an application of SGML
- It is a typed language, offering two alternatives to define document types**: DTD (Document Type Definition)** borrowed from SGML or **Xschema (XML schema)** based on an XML syntax.
- XML is Descriptive, referential, meta-markup



```
<?xml version="1.0"?>
<rss version="2.0">
    <channel>
        <title>Grid@CLEF News</title>
        <link>http://ims.dei.unipd.it/gridclef/</link>
        <description>Events and updates about the Grid@CLEF track.</description>
        <item>
            <title>Terrier Support for Grid@CLEF</title>
            <pubDate>Wed, 11 Feb 2009 15:41:49 GMT</pubDate>
            <link>http://ir.dcs.gla.ac.uk/terrier/issues/browse/TR-9</link>
            <guid isPermaLink="false">1234363309</guid>
            <description>The Terrier open source information retrieval system will suppor
        </item>
        <item>
            <title>Registration to Grid@CLEF 2009 opens</title>
            <pubDate>Wed, 4 Feb 2009 00:00:00 GMT</pubDate>
            <link>http://www.clef-campaign.org/</link>
            <guid isPermaLink="false">1233705600</guid>
            <description>Registration for Grid@CLEF 2009 opens today.</description>
        </item>
    </channel>
</rss>
```

XML document is a **tree** with different types of node.



Types of Node in XML

- **Text**: fragment of (unstructured) information represented in the XML document
- **Element**: contains others nodes and it is logical grouping of the information represented by its descendants
- **Attribute**: associated to an element represents its additional properties
- **Comment**: a textual content ignored in the processing
- **Processing instruction**: meta-directive for the XML processor
- **Root**: the whole XML tree, implicit

Textual Representation of XML Nodes

- **Text**: represented exactly as the contained text
- **Element**: defined by opening and closing tags enclosed by angular brackets < >
    - o `<elements> ... </elements>`
    - o `<empty-element></empty-element> or <empty-element/>`
- **Attribute**: only in the opening tag of an elements and cannot be duplicated `attribute-name = "value"`
- **Comment**: everywhere in the document (but not within other markup) enclosed by <!--... -->. `<!-comment` →

- **Processing instruction:** enclosed by angular brackets and question marks, it defines the application which has to process it, called target, and the information to pass to the application. `<?target value?>`
- **Root**: implicit

Properties of an XML Document

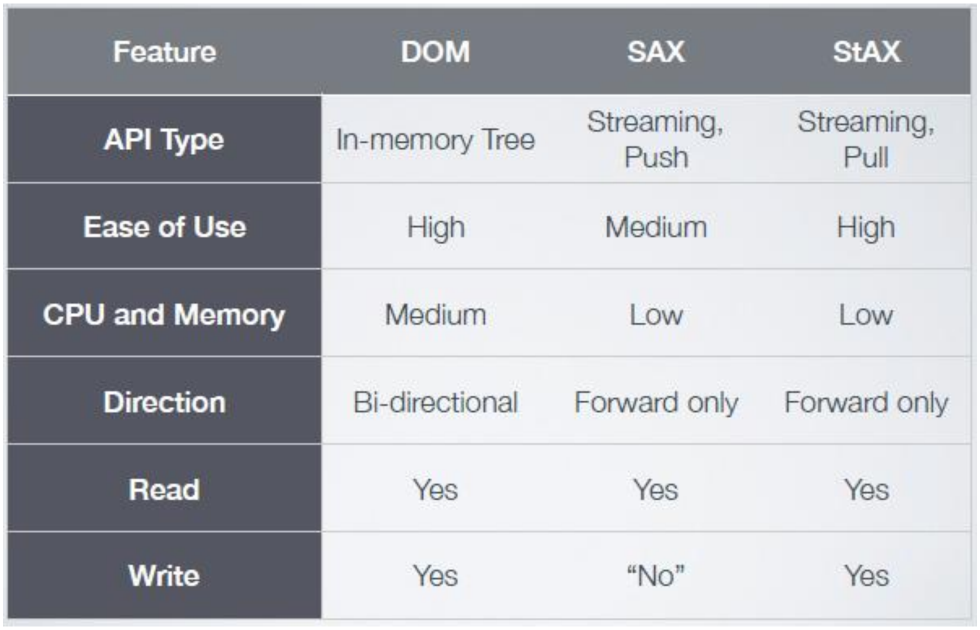
- XML document must be **well-formed**
  - opening and closing tags must match and be properly nested
  - attribute values must be enclosed by quotes
  - there must be a unique root node element
- If a document type is specified (with either DTD or XML schema) they may be **valid**, if the document complies with the constraint expressed in the document type.

DOM (Document Object Model) is a model and an API independent from the platform and language which allow us to process HTML and XML documents, creating an in-memory representation of them

- The DOM API is expressed by using the **Interface Definition Language (IDL)**, a generic syntax to define object-oriented interfaces.
- A language can implement DOM by providing a binding from the abstract interface to a concrete one in the language. Bindings towards many languages, among which JavaScript, Java, PHP
- DOM is a set of W3C recommendations, being updated to better match with HTML5.

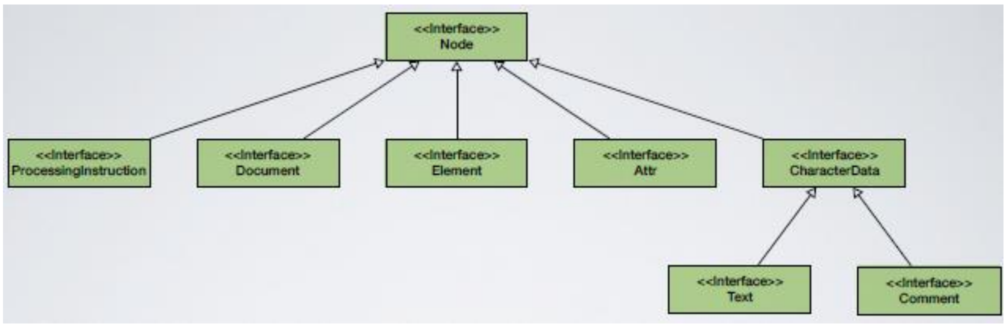
Approaches to parse XML:

- **DOM (Document Object Model):** is an API based on creating an in-memory representation of the XML used also in browsers to parse, represent, and render HTML documents.
- **SAX (Simple API for XML):** is a push streaming API which notifies the applications of events (start tag, end tag, …) by means of call-backs.
- **StAX (Streaming API for XML):** is a pull streaming API which returns events to the applications when asked.



Feature	DOM	SAX	StAX
API Type	In-memory Tree	Streaming, Push	Streaming, Pull
Ease of Use	High	Medium	High
CPU and Memory	Medium	Low	Low
Direction	Bi-directional	Forward only	Forward only
Read	Yes	Yes	Yes
Write	Yes	"No"	Yes

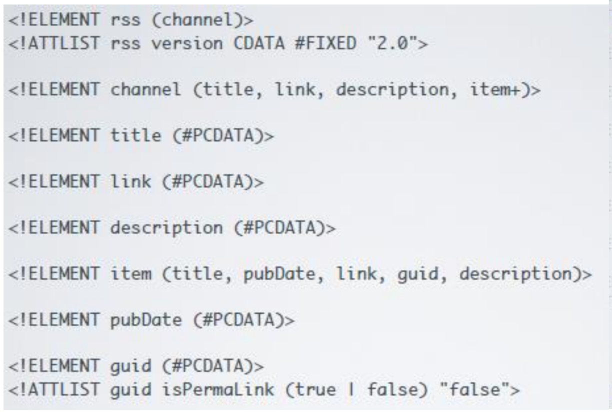
DOM: Main Interfaces



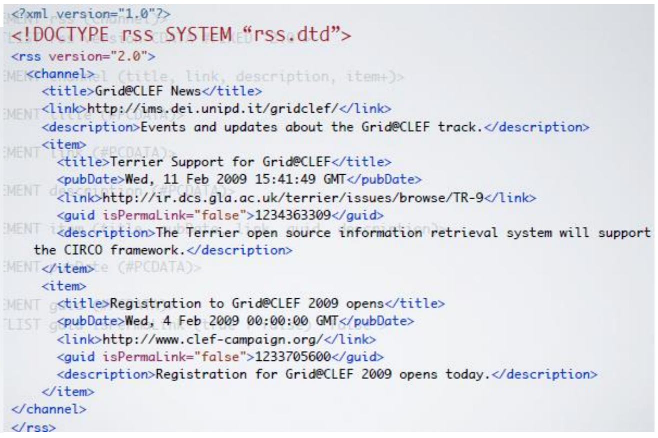
Document Type Definition (DTD): It is a way to express the structure (document type) of an XML Document. Same mechanism used in SGML.

- It allows us to define element, attributes, and textual data
- It uses a syntax inspired by regular expressions
    - o `<!ELEMENT name model>`: defines an element with the given name and model
    - o `<!ATTLIST element-name attribute-name type default>`: defines an attribute of an element
    - o `<!DOCTYPE root SYSTEM "URI">`: declares which DTD to used in an XML document and which elements is to be considered the root of the Document.

Example:



```
<!ELEMENT rss (channel)>
<!ATTLIST rss version CDATA #FIXED "2.0">

<!ELEMENT channel (title, link, description, item+)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT link (#PCDATA)>

<!ELEMENT description (#PCDATA)>

<!ELEMENT item (title, pubDate, link, guid, description)>

<!ELEMENT pubDate (#PCDATA)>

<!ELEMENT guid (#PCDATA)>
<!ATTLIST guid isPermaLink (true | false) "false">
```



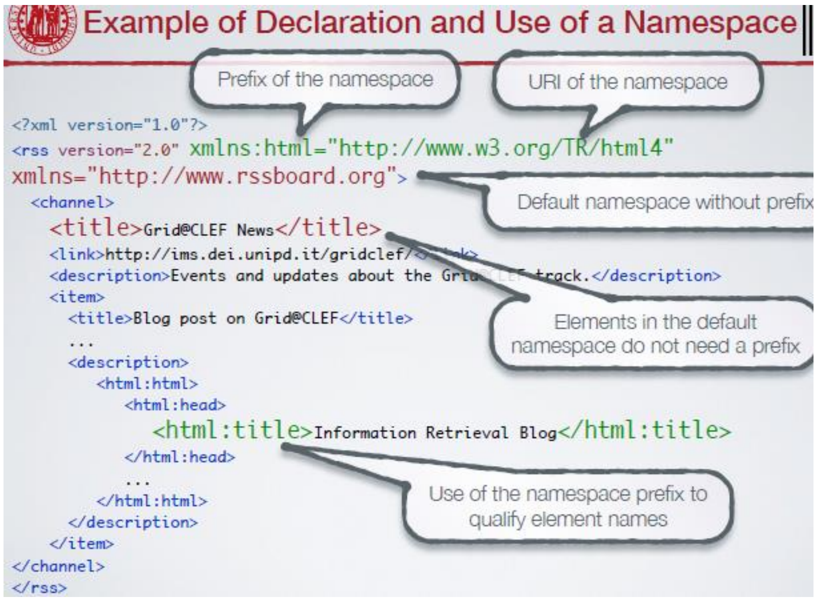
```
<?xml version="1.0"?>
<!DOCTYPE rss SYSTEM "rss.dtd">
<rss version="2.0">
  <channel>
    <title>Grid@CLEF News</title>
    <link>http://ims.dei.unipd.it/gridclef/</link>
    <description>Events and updates about the Grid@CLEF track.</description>
    <item>
      <title>Terrier Support for Grid@CLEF</title>
      <pubDate>Wed, 11 Feb 2009 15:41:49 GMT</pubDate>
      <link>http://ir.dcs.gla.ac.uk/terrier/issues/browse/TR-9</link>
      <guid isPermaLink="false">1234363309</guid>
      <description>The Terrier open source information retrieval system will support
   the CIRCO framework.</description>
    </item>
    <item>
      <title>Registration to Grid@CLEF 2009 opens</title>
      <pubDate>Wed, 4 Feb 2009 00:00:00 GMT</pubDate>
      <link>http://www.clef-campaign.org/</link>
      <guid isPermaLink="false">1233705600</guid>
      <description>Registration for Grid@CLEF 2009 opens today.</description>
    </item>
</channel>
</rss>
```

Limitations of DTD:

- Lacks a mechanism to support data types in elements and attributes
- Declaration of elements and attributes is independent from the context e.g., you cannot define the value of an attributes on the basis of the values of another element.
- It does not use the XML syntax
- It does not support namespaces (see later on)
- It does not support auto-documentation

XML Namespace

- Programming languages address clashes in the names defined in different modules, classes, etc. by using namespaces which fully qualify a name e.g. Java packages
- XML uses **Uniform Resource Identifiers (URI)** to uniquely identify a namespace.
- Since URIs are typically very long strings, it is possible to associate a **prefix** to a URI to be used locally in a XML document in place of the full URI.



Example of Declaration and Use of a Namespace

Prefix of the namespace
URI of the namespace
Default namespace without prefix
Elements in the default namespace do not need a prefix
Use of the namespace prefix to qualify element names

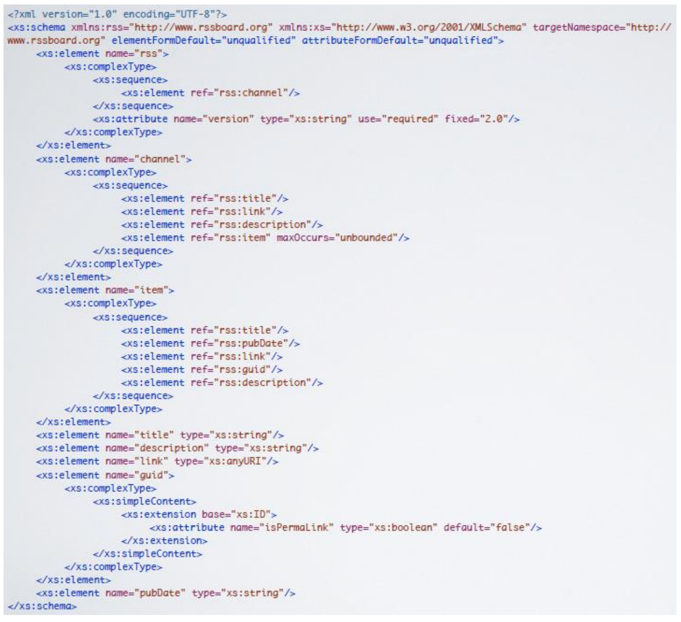
```
<?xml version="1.0"?>
<rss version="2.0" xmlns:html="http://www.w3.org/TR/html4"
xmlns="http://www.rssboard.org">
  <channel>
    <title>Grid@CLEF News</title>
    <link>http://ims.dei.unipd.it/gridclef/</link>
    <description>Events and updates about the Grid@CLEF track.</description>
    <item>
      <title>Blog post on Grid@CLEF</title>
      ...
      <description>
         <html:html>
            <html:head>
               <html:title>Information Retrieval Blog</html:title>
            </html:head>
            ...
         </html:html>
      </description>
    </item>
</channel>
</rss>
```

xmlns attribute:

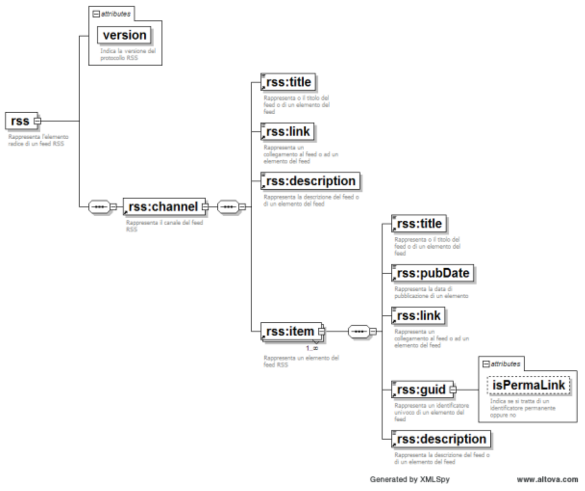
- When using prefixes in XML, a namespace for the prefix must be defined
- The namespace can be defined by an `xmlns` attribute in the start tag of an element
- The syntax is `xmlns:name_of_the_prefix="URI"`
- Namespaces can also be declared in the XML root element
- The namespace URI is not used by the parser to look up information, but only to give the namespace a unique name (companies, however, often use the namespace as a pointer to a web page containing information about the namespace)
- The default namespace for an element allows us from using prefixes in all the child elements, the syntax is `xmlns="default_namespace_URI"`

XML Schema: It is a way to express the structure (document type) of an XML document by using the XML syntax.

- It allows us to define: simple and complex (data) types, elements with either a simple or a complex type, attributes with simple type.
- An XML schema is linked to an XML document in its root element.



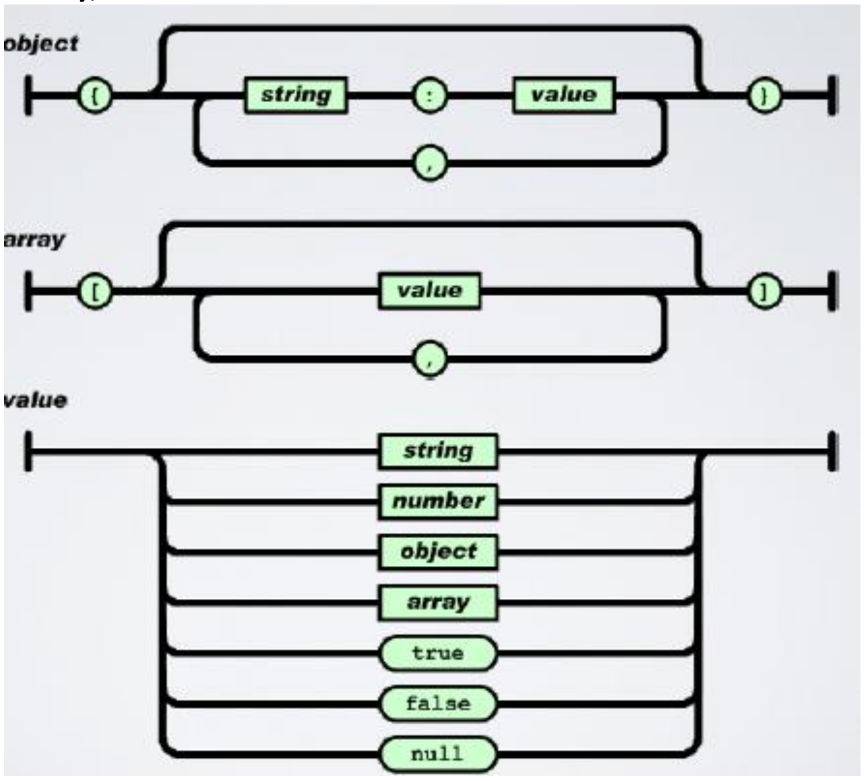
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:rss="http://www.rssboard.org" xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://
www.rssboard.org" elementFormDefault="unqualified" attributeFormDefault="unqualified">
    <xs:element name="rss">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="rss:channel"/>
            </xs:sequence>
            <xs:attribute name="version" type="xs:string" use="required" fixed="2.0"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="channel">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="rss:title"/>
                <xs:element ref="rss:link"/>
                <xs:element ref="rss:description"/>
                <xs:element ref="rss:item" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="item">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="rss:title"/>
                <xs:element ref="rss:pubDate"/>
                <xs:element ref="rss:link"/>
                <xs:element ref="rss:guid"/>
                <xs:element ref="rss:description"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="link" type="xs:anyURI"/>
    <xs:element name="guid">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:ID">
                    <xs:attribute name="isPermaLink" type="xs:boolean" default="false"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
    <xs:element name="pubDate" type="xs:string"/>
</xs:schema>
```



7.4. JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format.

- it is based on a subset of the JavaScript Programming Language and browsers automatically parse JSON into JavaScript objects
- JSON is a text format that is completely language independent.
- JSON is built on two **structures**:
    - an **object**, i.e. a collection of name/value pairs.
    - an **array**, i.e. ordered list of values



JSON Schema is a JSON media type for defining the structure of JSON data
Is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data.

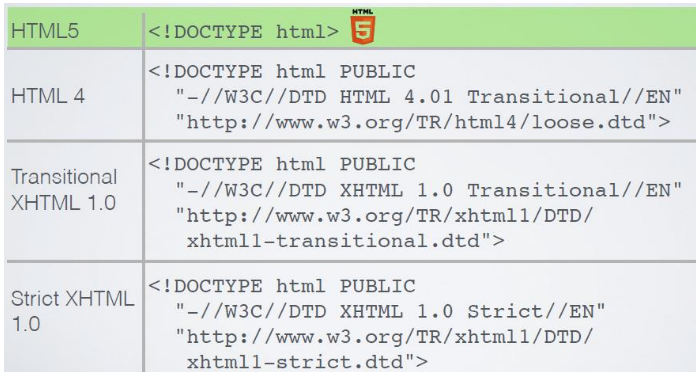
Processing JSON in Java: is usually parsed using a pull streaming API, similar to StAX for XML

- Jackson Project: https://github.com/FasterXML/48et48Att
- Java API for JSON Processing Specification (JSON-P) under Java EE 8: https://jcp.org/en/jsr/detail?id=374

8.HTML

8.1. Main Elements

Each web page should begin with a DOCTYPE declaration to tell a browser which version of HTML the page is using.

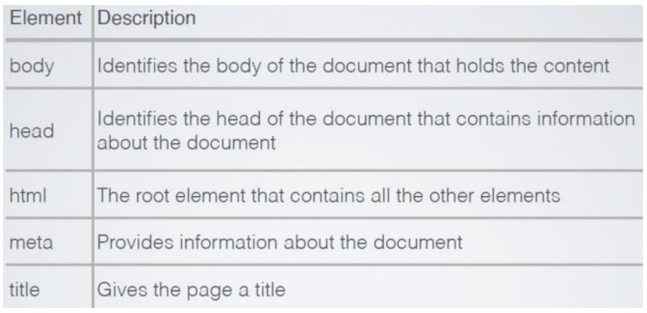


HTML5	`<!DOCTYPE html>`
HTML 4	`<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">`
Transitional XHTML 1.0	`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`
Strict XHTML 1.0	`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">`

Meta Elements

- The `<meta>` elements provide information about the document itself, they can be used to provide all sorts of information.
- The `<meta>` element is an empty element so it does not have a closing tag.
- The meta elements are not displayed by the browser, but are machine parsable and are usually placed within the head element.
- Typically used to specify character set, page description,
- keywords, author of the document, and viewport settings (the
- viewport is the user's visible area of a web page, and it varies with
- the device)
- Metadata is used by browsers (to understand how to display the content), search engines (by using the specified keywords), and other web services.

Document Structure:



Element	Description
body	Identifies the body of the document that holds the content
head	Identifies the head of the document that contains information about the document
html	The root element that contains all the other elements
meta	Provides information about the document
title	Gives the page a title

8.1.1. Text

Semantic markup: are text elements that are not intended to affect the structure of your web pages, but they do add extra information to the pages.

- Examples: `<h1>` indicates the most important heading at the beginning of the document, `<em>` indicates where emphasis should be placed, and `<blockquote>` indicates that a block of text is a quotation.
- Browsers often display the contents of these elements in a different way.

- They should not be used to change the way that the text **looks**; their purpose is to **describe** the content of a web page more accurately. i.e., you should not use `<h1>` (or any other HTML tag) because it "looks good" for your purpose. You should use it because the text contained in the tag has the importance of a title. For the appearance of that text, we will use CSS.

Headings: HTML has six "levels" of headings: `<h1>` is used for main headings, `<h2>` is used for subheadings, and so on

- Browsers display the contents of headings at different sizes. The contents of an `<h1>` element is the largest, and the contents of an `<h6>` element is the smallest.
- Browsers also automatically add some white space (a margin) before and after a heading (more about margins later in CSS)
- The CSS allows you to control the size of text, its color, and the fonts used.
- Search engines use headings to index the structure and content of your web pages, use them with good reason if you want to be correctly retrieved.
- Use heading to make headings only, don't use them to make the text big or bold.

Paragraphs: usually a block of text, and always starts on a new line (it is an example of block element, more on this later).

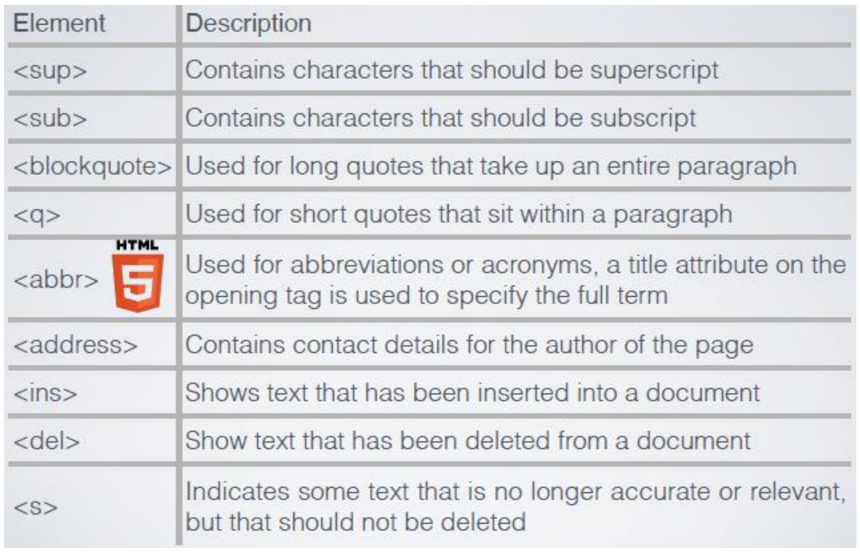
- Defined by the `<p>` tag.
- Browsers automatically add a margin before and after a paragraph.
- With HTML, you cannot be sure, or known in advance, how your HTML will be displayed. Large and small screens and resized windows will produce different results.
- In HTML you cannot change how the text (or other elements) are displayed by adding extra spaces or extra lines in the HTML code: the browser will automatically remove any extra spaces and lines when rendering the page. We will need to use CSS.
- A paragraph consists of one or more sentences that form a self-contained unit of discourse. The start of a paragraph is indicated by a new line.
- Text is easier to understand when it is split up into units of text. For example, a book may have chapters. Chapters can have subheadings. Under each heading there will be one or more paragraphs.
- Paragraphs may contain text, images, and other inline elements, but they may **not contain headings**, lists, sectioning elements, or any element that typically displays as a block by default.

Bold and Italic: `<b>, <i>.` They are old, try not use them. Use strong ang Emphasis.
Strong and Emphasis: `<strong>, <em>`

Line Breaks and Horizontal Rules:

- To add a line break inside the middle of a paragraph you can use the line break tag `<br />`.
- To create a break between themes — such as a change of topic in a book or a new scene in a play — you can add a paragraph-level thematic break (horizontal rule) between sections using the `<hr />` tag.
- **empty elements:** do not have any words between an opening and closing tag.
- An empty element usually has only one tag: before the closing angled bracket of an empty element there will often be a space and a forward slash character.

More Text Elements:



Element	Description
<sup>	Contains characters that should be superscript
<sub>	Contains characters that should be subscript
<blockquote>	Used for long quotes that take up an entire paragraph
<q>	Used for short quotes that sit within a paragraph
<abbr>	Used for abbreviations or acronyms, a title attribute on the opening tag is used to specify the full term
<address>	Contains contact details for the author of the page
<ins>	Shows text that has been inserted into a document
<del>	Show text that has been deleted from a document
<s>	Indicates some text that is no longer accurate or relevant, but that should not be deleted

8.1.2. Lists

Type of Lists:

- **Ordered lists**: are lists in which the sequence of the items is important, each item in the list is numbered;
- **Unordered lists**: collections of items that appear in no particular order, begin with a bullet point, rather than characters that indicate order;
- **Description lists**: lists that consist of name and value pairs, including but not limited to terms and definitions.

All list elements (the lists themselves and its items) are displayed as block elements by default, which means that they start on a new line and have some space above and below.

Ordered List: created with the `<ol>` element.
Each item in the list is placed between an opening `<li>` (list item) tag and a closing `</li>` tag.
The CSS list-style-type property can be used to change the bullets and numbers for lists.

Unordered list: is created with the `<ul>` element.
Each item in the list is placed between an opening `<li>` (list item) tag and a closing `</li>` tag.
The CSS list-style-type property can be used to change the the type of bullet points (circles, squares, diamonds and so on).

Description list is created with the `<dl>` element. Inside the <dl>
element you will usually see pairs of `<dt>` and `<dd>` elements.

- `<dt>` is used to contain the definition term.
- `<dd>` is used to contain the actual definition.

Nested Lists: You can put a second list inside an <li> element to create a sub-list or nested list.

8.1.3. Links

Anchor Syntax: Links are the defining feature of the web because they allow you to move from one web page to another, enabling the very idea of browsing or surfing.

- Links are created using the anchor element: `<a>`
- Users can click on anything between the opening `<a>` tag and the closing `</a>` tag. You specify which page you want to link to using the `href` attribute.

Link Text: Is the text between the opening <a> tag and closing </a> tag.

- Where possible, your link text should explain where visitors will be taken if they click on it (rather than just saying “click here”).
- Nearly all graphical browsers display linked text as blue and underlined by default. Visited links are generally displayed in purple.
- If you choose to change your link colors, keep them consistent throughout your site so as not to confuse your users.
- In HTML5, you can put any element in an <a> element, even block elements.

The href Attribute (hypertext reference) provides the address of the page or resource (its URL) to the browser.

- The URL can point to other HTML documents or to other web resources, such as images, audio, and video files.
- There are two ways to specify the URL:
    - **Absolute URLs** provide the full URL for the document, including the protocol (http://), the domain name, and the pathname as necessary. Absolute URLs are used when pointing to a document on the Web (i.e., not on your own server);
    - **Relative URLs** describe the pathname to a file relative to the current document. Relative URLs can be used when you are linking to another document on your own site (i.e., on the same server).

Email and Telephone Links:

- `href=”mailto:jon@example.org”`
- `href=”tel:+18005551212”`

Opening Links in a New Window: `target=”_blank”`

Linking to a Page Fragment: Useful to provide shortcuts to information at the bottom of a long, scrolling page or for getting back to the top of a page with just one click or tap.
Two part process:

- **Identifying the destination:** use the **id attribute** (can be used on every HTML element), the value of the id attribute should start with a letter or an underscore (not a number or any other character) and it is has to be unique: it has to appear only once in the document;
- **Linking to the destination**: use the **<a>** element, but the value of the href attribute starts with the **#** symbol, followed by the value of the id attribute of the element you want to link to.

```
<h1 id=”top”>Film-Making Terms</h1>
…
<p><a href=”#top”>Top</a></p>
```

Linking to a Fragment in Another Page:

```
<a href=“http://www.htmlandcssbook.com/index.html#bottom”>
```

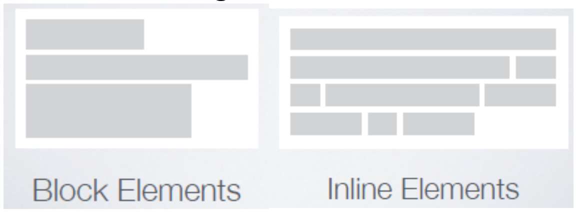
8.1.4. Images

Adding Images: `<img>`

- `src`: (source) tells the browser where it can find the image file. This will usually be a relative URL pointing to an image on your own site.
- `alt`: (alternate text) provides a text description of the image which describes the image if you cannot see it.
- `title`: provides additional information about the image. Most browsers will display the content of this attribute in a tootip when the user hovers over the image.
- `height`: this specifies the height of the image in pixels;
- `width`: this specifies the width of the image in pixels.

Block and Inline Elements: Browsers show HTML elements in one of two ways:

- **Block elements** always appear on a new line. Examples of block elements: <h1>, <p>, <ul>, <li>.
- **Inline elements** sit within a block level element and do not start on a new line. Examples of inline elements: <a>, <em>, and <img> elements.



Placing Images: <img> is an inline element, when the browser window is resized, the line of images reflows to fill the new width.

- Where an image is placed in the code will affect how it is displayed: Before a paragraph, Inside the start of a paragraph, In the middle of a paragraph
- New websites should use CSS to control the alignment of images (instead of the align attribute).

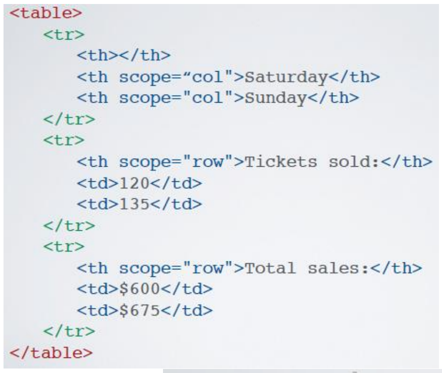
Figure and Figure Caption: `<figure>` and `<figcaption>`

- You can have more than one image inside the `<figure>` element as long as they all share the same caption.
- Before these elements were created there was no way to associate an `<img>` element with its caption.

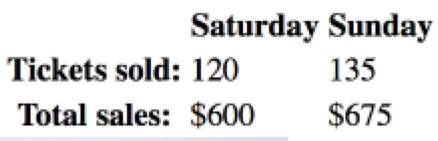
8.1.5. Tables

Basic Table Structure

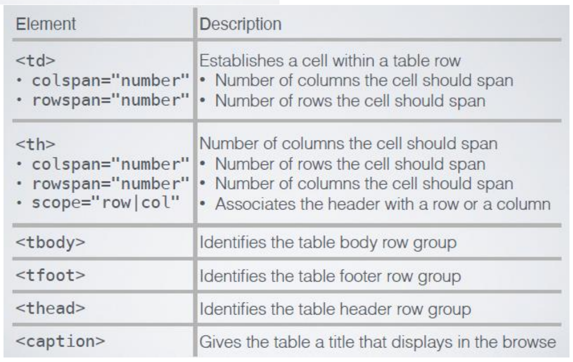
- `<table>` element is used to create a table.
- `<tr>` (table row) tag indicates the start of each row, and </tr> indicates the end of the row.
- Each cell of a table is represented using <td> and </td> tags (table data).
- `<th>` (table header) element is used just like the <td> element but its purpose is to represent the heading for either a column or a row.
- The `scope` attribute on the <th> element indicates whether it is a heading for a column (value equals to col) or a row (value equals to row).



```
<table>
   <tr>
      <th></th>
      <th scope="col">Saturday</th>
      <th scope="col">Sunday</th>
   </tr>
   <tr>
      <th scope="row">Tickets sold:</th>
      <td>120</td>
      <td>135</td>
   </tr>
   <tr>
      <th scope="row">Total sales:</th>
      <td>$600</td>
      <td>$675</td>
   </tr>
</table>
```



	Saturday	Sunday
Tickets sold:	120	135
Total sales:	$600	$675



Element	Description
`<td>` • `colspan="number"` • `rowspan="number"`	Establishes a cell within a table row • Number of columns the cell should span • Number of rows the cell should span
`<th>` • `colspan="number"` • `rowspan="number"` • `scope="row|col"`	Number of columns the cell should span • Number of rows the cell should span • Number of columns the cell should span • Associates the header with a row or a column
`<tbody>`	Identifies the table body row group
`<tfoot>`	Identifies the table footer row group
`<thead>`	Identifies the table header row group
`<caption>`	Gives the table a title that displays in the browse

8.1.6. Forms

Form Controls:

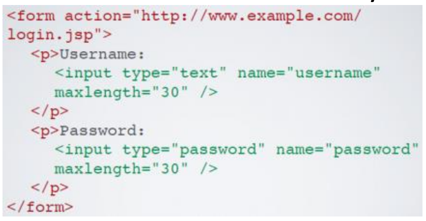
- **Adding Text:** Text input (single line), Password input, Text area
- **Making Choices:** Radio buttons, Checkboxes, Drop-down boxes
- **Submitting Forms:** Submitting Buttons, Image Buttons
- **Uploading File:** File upload

The <form> element carries:

- **action** (mandatory): its value is the URL for the page on the server that will receive the information in the form when it is submitted.
- **method**: forms can be sent using GET or POST (default: GET).
- **id**: its value is used to identify the form distinctly from other elements on the page.

Text Input: `<input>`

- The value of the **type attribute** determines what kind of input they will be creating:
  - **type="text"**: creates a single-line text input.
  - **type="password"**: creates a text box that acts just like a single-line text input, except the characters are blocked out.
- **name**: the value of this attribute identifies the form control and is sent along with the information they enter to the server (to differentiate between various pieces of inputted data, information is sent from the browser to the server using name/value pairs).
- **maxlength**: limits the number of characters a user may enter into the text field.



```
<form action="http://www.example.com/
login.jsp">
  <p>Username:
    <input type="text" name="username"
    maxlength="30" />
  </p>
  <p>Password:
    <input type="password" name="password"
    maxlength="30" />
  </p>
</form>
```

Difference Between Id and Name Attributes

Id Attribute:

- Every HTML element can carry the id attribute.
- The id value must be unique.
- It is used to uniquely identify that element from other elements on the page.
- Useful with CSS and javascript.

Name Attribute:

- The name attribute provides the variable name for the control.
- The name value do not need to be unique.
- When a user enters a comment in a control field, it would be passed to the server as a name/value pair.
- All form control elements must include a name attribute (except submit) so the form-processing application can sort the information.
- The web application that processes the data is programmed to look for specific variable names.

Text Area: `<textarea>`

- used to create a mutli-line text input.
- Unlike other input elements this is not an empty element. Any text that appears between the opening <textarea> and closing </textarea> tags will appear in the text box when the page loads, if the user does not delete it, this message will be sent to the server along with whatever the user has typed.

Radio Buttons and Checkboxes, Use the `<input>` element with:

- `type="radio"`: radio buttons allow users to pick just one of a number of options.
- `type="checkbox"`: checkboxes allow users to select (and unselect) one or more options in answer to a question.
- `name`: the value of the name attribute should be the same for all of the radio buttons or checkboxes used to answer a question.
- `value`: the value sent to the server for the selected option, the value of each of the buttons in a group should be different.
- `checked`: the checked attribute can be used to indicate which value (if any) should be selected when the page loads. The value of this attribute is checked.

Drop Down List

- The `<select>` element is used to create a drop down list box (select box).
- It contains two or more `<option>` elements.
- The words between the opening <option> and closing </ option> tags will be shown to the user in the drop down box.
- The <option> element uses the `value` attribute to indicate the value that is sent to the server along with the `name`.
- The `selected` attribute can be used to indicate the option that should be selected when the page loads, otherwise the first option will be shown.

```
<form action="http://www.example.com/profile.jsp">
    <p>What device do you listen to music on?</p>
    <select name="devices">
        <option value="ipod">iPod</option>
        <option value="radio">Radio</option>
```

```
                <option value=”computer”>Computer</option>
            </select>
        </form>
```

File Input Box and Submit Button: `<input>` element with

- `type=“file”`: allows users to upload files. It creates a box that looks like a text input followed by a browse button, that allows the user to select a file from their computer to be uploaded.
- `type=“submit”`: creates a submit button. The value attribute is used to control the text that appears on a button.
- `type=“image”`: uses an image for the submit button.

```
        <form action=”http://www.example.com/upload.jsp” method=”post”>
            <p>Upload your song in MP3 format:</p>
            <input type=”file” name=”user-song” />
            <br />
            <input type=”submit” value=”Upload” />
        </form>
```

HTML5 `<input>`: supports form validation, the browser can give users messages if the form control has not been filled in correctly. Traditionally, form validation has been performed using JavaScript. Introduces new elements:

- `type=“date”`: date input control
- `type=“range”`: slider input
- `type=“email”`
- `type=“url”`
- `type=“search”`
- `type=“color”`: color selector

DataList `<datalist>` element allows the author to provide a drop-down menu of suggested values for any type of text input.
It gives the user some shortcuts to select from, but if none are selected, the user can still type in her own text.
The list attribute in the input element to associate it with the id of its respective datalist.

Comments: `<!-comment goes here` →

Class Attribute `class=”…”`: identifies several elements as being different from the other elements on the page, its value should describe the class it belongs to. The class attribute on any element can share the same value.

Groups in a Block and Inline

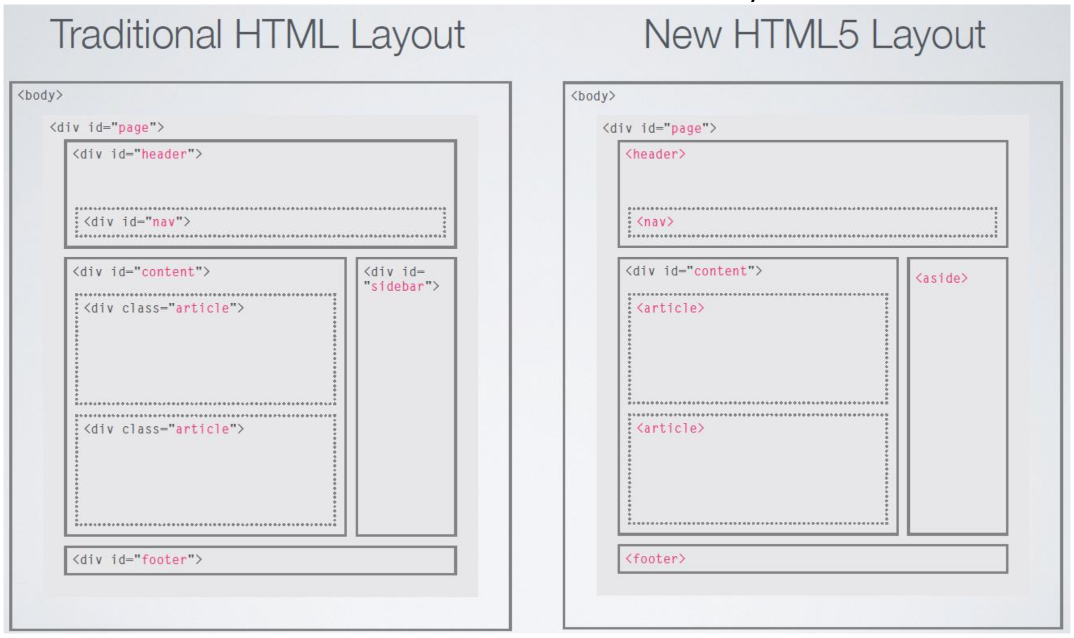
- The `<div>` element allows you to group a set of elements together in one block-level box.
- The `<span>` element acts like an inline equivalent of the <div> element.
- Using an `id` or `class` attribute on the <div> or <span> elements means that you can create CSS style rules to change the appearance of all the elements contained within them.

8.2. HTML5 New Elements

`article, aside, audio, bdi, canvas, command, datalist, details, embed, figcaption, Figure, footer, header, hgroup, keygen, mark, meter, nav, output, progress, rp, rt, ruby, section, source, summary, time, track, video, wbr`

Page Layout

- HTML 4: web page authors used <div> elements to group together related elements on the page (such as the elements that form a header, an article, footer or sidebar), and used class or id attributes to indicate the role of the <div> element in the structure of the page.
- HTML5: introduces a new set of elements that allow you to divide up the parts of a page. The names of these elements indicate the kind of content you will find in them.



Headers and Footers

- The `<header>` and `<footer>` elements can be used for:
    - The main header or footer that appears at the top or bottom of every page on the site.
    - A header or footer for an individual `<article>` or `<section>` within the page.
- The `<nav>` element is used to contain the major navigational blocks on the site such as the primary site navigation.

Article, Section, and Aside

- The `<article>` element acts as a container for any section of a page that could stand alone (a blog entry, a comment or forum post).
- The `<section>` element groups related content together, and typically each section would have its own heading. It may contain several distinct `<article>` elements that have a common theme or purpose.
- The `<aside>` element has two purposes:
    - When used inside an `<article>` element, it should contain information that is related to the article but not essential to its overall meaning.
    - When used outside of an `<article>` element, it acts as a container for content that is related to the entire page.

Linking Block Elements: HTML5 allows web page authors to place an <a> element around a block level element that contains child elements. This allows you to turn an entire block into a link.

HTML 5 API: HTML5 introduces many APIs (Application Programming Interfaces) for the creation of web applications. APIs standardize tasks that traditionally required proprietary plug-ins or custom programming. The following APIs are part of the W3C HTML5 specification:

- Media API, for playback of video and audio files;
- Session History API, for exposing the browser history;
- Offline Web Applications API, which allows web resources to be used while offline;
- Editing API, to create in-browser text editors;
- Drag and Drop API;
- Canvas API, for two dimensional drawing;
- Web Storage API, allows data to be stored in the browser's cache;
- Geolocation API, lets users share longitude and latitude information;
- Web Workers API, that allows scripts to run in the background;
- Web Sockets API, that maintains an open connection between the client and the server.

Video

- The `<video>` element embeds a video file in the web page.
- The video resource can be provided with the `src` attribute or by one or more `<source>` elements inside the video element to provide several video format options.
- There is still debate regarding the supported video formats for the video element. No file format is supported by all browsers.
- width and height (pixel measurement): size of the box the embedded media player takes up on the screen.
- `poster`: provides the location (url) of a still image to use in place of the video before it plays.
- `controls`: prompts the browser to display its built-in media controls, (play/pause button, "seeker", volume).
- `autoplay`: makes the video start playing automatically once it has downloaded enough of the media file (to be avoided).

```
<video src="highlight_reel.mp4" width="640" height="480"
poster="highlight_still.jpg" controls autoplay> </video>
```

Audio

- The `<audio>` element uses the same attributes as the video element, with the exception of width, height, and poster (because there is nothing to display!).
- `preload`: suggests the browser whether the audio data should be fetched or not:
    - `preload="auto"`: the video should be fetched as soon as the page loads.
    - `preload="none"`: wait until the user presses the play button and then fetch the video.
    - `preload="metadata"`: loads information about the media file, but not the media itself.

```
<audio id="soundtrack" controls preload="auto">
<source src="soundtrack.mp3" type="audio/mp3">
<source src="soundtrack.ogg" type="audio/ogg">
<source src="soundtrack.webm" type="audio/webm">
</audio>
```

Canvas

- The `<canvas>` element creates an area on a web page that you can draw on using a set of JavaScript functions for creating lines, shapes, fills, text, animations, and so on.
- Everything on the canvas is generated with scripting, that means it is dynamic and can draw things on the fly and respond to user input.
- You add a canvas space to the page with the canvas element and specify the dimensions with the width and height attributes.

```
<canvas width="600" height="400" id="my_first_canvas">Your browser
does not support HTML5 canvas. Try using Chrome, Firefox, Safari or
Internet Explorer 9. </canvas>
```

9. CSS

9.1. Introduction to CSS

Cascading Style Sheets (CSS) is the W3C standard for defining the presentation
of documents written in HTML. Presentation, refers to the way the document is displayed or delivered to the user.
CSS allows you to create rules that specify how the content of an element should appear.
CSS treats each HTML element as if it appears in a box.

The Benefits of CSS:

- **Precise type and layout controls.** You can achieve printlike precision using CSS.
- **Less work.** You can change the appearance of an entire site by editing one style sheet.
- **Reliable browser support**. Every browser in current use supports CSS.

CSS History

- CSS was first proposed by Håkon Wium Lie on October 10, 1994. At the time, Lie was working with Tim Berners-Lee at CERN.
- The CSS 1 specification was completed in 1996. Browser CSS support was typically incomplete and had many bugs that prevented CSS from being usefully adopted.
- CSS level 2 specification was developed by the W3C and published as a recommendation in May 1998.
- The earliest CSS 3 drafts were published in June 1999.

Attaching the Style to the HTML Document:

- **External style sheets** is a separate, text-only document that contains a number of style rules. It must be named with the .css suffix. The .css document is then linked to or imported into one or more HTML documents.
  ```
  <link href=”css/styles.css” type="text/css” rel=”stylesheet” />
  ```
- **Embedded style sheets** is placed in a document using the style element, and its rules apply only to that document. The style element must be placed in the head of the document.
  ```
  <style type=”text/css”>…..</style>
  ```
- **Inline styles,** to apply properties and values to a single element using the style attribute in the element itself.
  ```
  <h1 style=”color: red; margin-top:2em”>Introduction</h1>
  ```

Multiple Style Sheets:

- Your HTML page can link to one style sheet and that stylesheet can use the `@import` rule to import other style sheets.
  ```
  @import url("tables.css”);
  ```
- In the HTML you can use a separate <link> element for each style sheet.

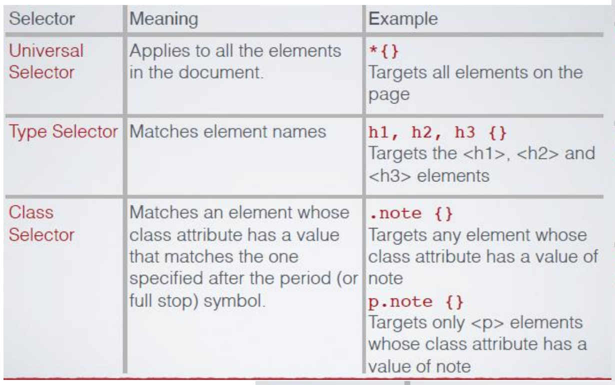
A style sheet is made up of one or more style instructions, called **rules** or rule sets.
They describe how an element or group of elements should be displayed.
Each rule **selects** an element and **declares** how it should look.

CSS works by associating rules with HTML elements.
A CSS rule contains two parts: a selector and a declaration:

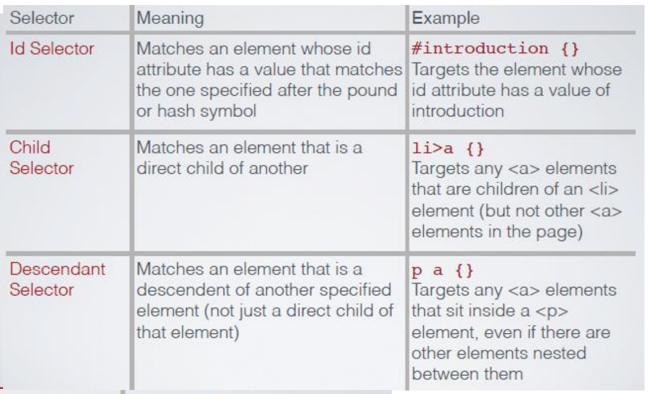
- **Selectors** indicate which element the rule applies to.
- **Declarations** indicate how the elements referred to in the selector should be styled. Declarations are split into two parts (a property and a value), and are separated by a colon. There can be more than one declaration in a single rule.

```
      Selector {property1:value1; …}
```

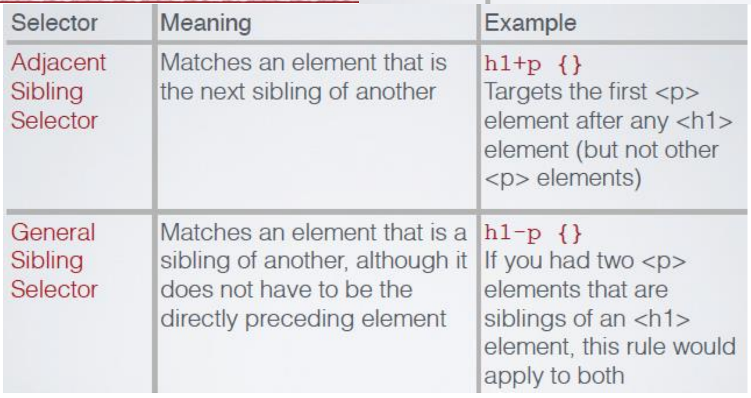
CSS Selectors Types:



Selector	Meaning	Example
Universal Selector	Applies to all the elements in the document.	`*{}` Targets all elements on the page
Type Selector	Matches element names	`h1, h2, h3 {}` Targets the <h1>, <h2> and <h3> elements
Class Selector	Matches an element whose class attribute has a value that matches the one specified after the period (or full stop) symbol.	`.note {}` Targets any element whose class attribute has a value of note `p.note {}` Targets only <p> elements whose class attribute has a value of note



Selector	Meaning	Example
Id Selector	Matches an element whose id attribute has a value that matches the one specified after the pound or hash symbol	`#introduction {}` Targets the element whose id attribute has a value of introduction
Child Selector	Matches an element that is a direct child of another	`li>a {}` Targets any <a> elements that are children of an <li> element (but not other <a> elements in the page)
Descendant Selector	Matches an element that is a descendent of another specified element (not just a direct child of that element)	`p a {}` Targets any <a> elements that sit inside a <p> element, even if there are other elements nested between them



Selector	Meaning	Example
Adjacent Sibling Selector	Matches an element that is the next sibling of another	`h1+p {}` Targets the first <p> element after any <h1> element (but not other <p> elements)
General Sibling Selector	Matches an element that is a sibling of another, although it does not have to be the directly preceding element	`h1~p {}` If you had two <p> elements that are siblings of an <h1> element, this rule would apply to both

Pseudo Class Selector

- The browser keeps track of:
  - Whether a link was already clicked (the color changes);
  - Whether the cursor is over an element (hover state);
  - Whether a form element has been checked or disabled;
  - …
- Pseudo-class selectors are used to apply styles to elements in these states.
- Pseudo-class selectors are indicated by the colon (😊 character. They typically go immediately after an element name, for example: `li:first-child`.

Link Pseudo-Classes

- `:link`, applies a style to unclicked (unvisited) links
- `:visited`, applies a style to links that have already been clicked

```
                    a:link { color: maroon; }
```

User Action Pseudo-Classes:

- `:focus`, applies when the element is selected and ready for input;
- `:hover`, applies when the mouse pointer is over the element;
- `:active`, applies when the element (such a link or button) is in the process of being clicked or tapped.

Group Selectors: `h1, h2, p, em, img {…}`

How CSS Rules Cascade

- If there are two or more rules that apply to the same element, it is important to understand which will take precedence.
- **Specificity**: if one selector is more specific than the others, the more specific rule will take precedence over more general ones.
- **Last rule**: if the two selectors are identical, the latter of the two will take precedence.
- You can add **`!important`** after any property value to indicate that it should be considered more important than other rules that apply to the same element.

Inheritance:

- You can force a lot of properties to inherit values from their parent elements by using **`inherit`** for the value of the properties.

Style Sheet Hierarchy: Style information can come from various sources, listed here from general to specific. Items lower in the list will override items above them:

- Browser default settings
- User style settings (set in a browser as a “reader style sheet”)
- Linked external style sheet (added with the link element)
- Imported style sheets (added with the @import function)
- Embedded style sheets (added with the style element)
- Inline style information (added with the style attribute in an opening tag)
- Any style rule marked !important by the author
- Any style rule marked !important by the reader (user)

9.2. Color Property

Foreground Color: `color`
RGB Values, HEX Codes, Color Names, HSLA

Background Color: `background-color`
Default is transparent.

RGB: rgb(red, green, blue)

Contrast: Text is harder to read when there is low contrast between background and foreground colors

CSS3 Opacity: `opacity: 0.5;`

- The value is a number between 0.0 and 1.0.
- `rgba(0,0,0,0.5)`: rgb + opacity

HEX: `#rrggbb`

- values vary between 00 and ff (same as decimal 0-255)

HSL Colors: `hsl(0,0%,78%)`

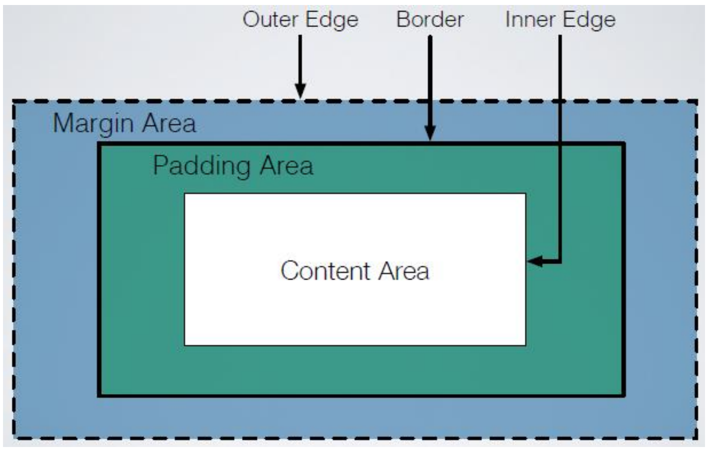
- **Hue** represents the color, is often represented as a color circle where the angle represents the color.
- **Saturation** is the amount of gray in a color, is represented as a percentage.
- **Lightness** is the amount of white (lightness) or black (darkness) in a color, is represented as a percentage.

HSLA: HLS + opacity

9.3. The box Model

Every element in a document generates a box to which different properties can be applied:

- Control the dimensions of your boxes
- Create borders around boxes
- Set margins and padding for boxes
- Show and hide boxes



The Box Components

- **Content area:** the core of the element box
- **Inner edge:** the edges of the content area. In real pages, the edge of the content area would be invisible.
- **Padding:** the padding is the area held between the content area and an optional border. Padding is optional.
- **Border:** is a line (or stylized line) that surrounds the element and its padding. Borders are also optional.
- **Margin:** is an optional amount of space added on the outside of the border.
  Margins are always transparent, allowing the background of the parent element to show through.
- **Outer edge:** the outside edges of the margin area. This is the total area the element takes up on the page, and it includes the width of the content area plus the total amount of padding, border, and margins applied to the element.

Sizing the Content Box: `width`, `height` properties

- **Pixels**: they allow designers to accurately control the box size.

- **Percentages**: the size of the box is relative to the size of the browser window or, if the box is encased within another box, it is a percentage of the size of the containing box.
- **Em**: the size of the box is based on the size of text within it.

Specifying Height

- It is less common to specify the height of elements, since it is better to keep the height calculated automatically allowing the element box to change based on the font size, user settings, or other factors.
- If you do specify a height for an element containing text, be sure to also consider what happens should the content not fit with the overflow property.
- Values of the overflow property:
    - `visible` (default) allows the content to hang out over the element box so that it all can be seen.
    - `hidden`: the content that does not fit does not appear beyond the edges of the element's content area.
    - `scroll`: when scroll is specified, scrollbars are added to the element box to let users scroll through the content. Be aware that when you set the value to scroll, the scrollbars will always be there, even if the content fits in the specified height just fine.
    - `auto`: allows the browser to decide how to handle overflow. In most cases, scrollbars are added only when the content doesn't fit and they are needed.

Padding: allows you to specify how much space should appear between the content of an element and its border. Is **not inherited** by child elements.

- `padding-top`
- `padding-right`
- `padding-bottom`
- `padding-left`
- `padding: 10px 5px 3px 1px;`

Borders: According to the CSS specification, if there is no border style specified, the border does not exist. In other words, you must always declare the style of the border, or the other border properties will be ignored.

- `border-width` (thickness)
- `border-style`
- `border-color`
- `border` (shorthand)

`border-width` property is used to control the width of a border. The value of this property can either be given in pixels or using one of the following values:

- thin
- medium
- thick

You can control the individual size of borders using four separate properties:

- border-top-width
- border-right-width
- border-bottom-width
- border-left-width

You can also specify different widths for the four border values in one property (clockwise order: **top, right, bottom, left.):** border-width: 2px 1px 1px 2px;

`border-style`

- `solid`: a single solid line
- `dotted`: a series of square dots
- `dashed`: a series of short lines
- `double`: two solid lines
- `groove`: appears to be carved into the page
- `ridge`: appears to stick out from the page
- `inset`: appears embedded into the page
- `outset`: looks like it is coming out of the screen
- `hidden / none`: no border is shown

individually change the styles of different borders using: `border-topstyle, border-left-style, border-right-style, border-bottomstyle`

`border-color`

- `border-top-color`
- `border-right-color`
- `border-bottom-color`
- `border-left-color`
- clockwise order: top, right, bottom, left

border: style, width, and color values in one declaration

Circular borders: border-radius: 5px 20px 40px 60px;
Elliptical borders: (horizontal radius and vertical radius): border-top-right-radius: 100px 50px;

Other CSS 3 Borders Properties
`border-image:` the border-image property applies an image to the border of any box. It takes a background image and slices it into nine pieces. This property requires three pieces of information:

- The URL of the image;
- Where to slice the image;
- What to do with the straight edges: (stretch, repeat, round).

`Box-shadow:` the box-shadow property allows you to add a drop shadow around a box. It must use at least the first of these two values as well as a color:

- Horizontal offset: negative values position the shadow to the left of the box.
- Vertical offset: negative values position the shadow to the top of the box.
- Blur distance: if omitted, the shadow is a solid line like a border.
- Spread of shadow: if used, a positive value will cause the shadow to expand in all
- directions, and a negative value will make it contract.

Margin: is an optional amount of space that you can add on the outside of the border. Margins keep elements from bumping into one another.
The value of the margin property **is not inherited** by child elements.

`Display` property defines the type of element box an element generates in the layout.

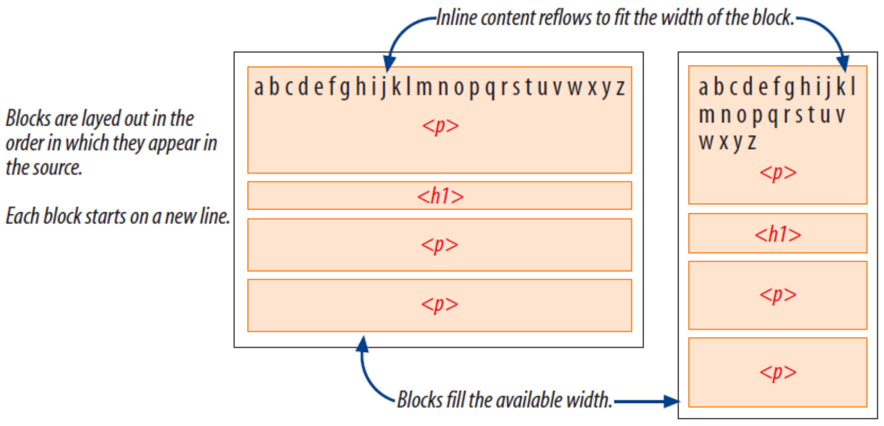
In addition to the familiar inline and block display roles, you can also make elements display as list items or the various parts of a table.

- `none`, which removes the content from the normal flow entirely

9.4. Floating and Positioning

Normal Flow

- Text elements are laid out from top to bottom in the order in which they appear in the source, and from left to right.
- Block elements stack up on top of one another and fill the available width of the browser window or other containing element.
- Inline elements and text characters line up next to one another to fill the block elements.
- When the window or containing element is resized, the block elements expand or contract to the new width, and the inline content reflows to fit.
- The normal flow is the default way in which browsers treat HTML elements, thus there is no need of a CSS property to indicate that elements should appear in normal flow.
- Anyway, the syntax would be: `position: static;`


Inline content reflows to fit the width of the block.
Blocks are layed out in the order in which they appear in the source.
Each block starts on a new line.
a b c d e f g h i j k l m n o p q r s t u v w x y z
<p>
<h1>
<p>
<p>
a b c d e f g h i j k l m n o p q r s t u v w x y z
<p>
<h1>
<p>
<p>
Blocks fill the available width.

Relative positioning moves an element in relation to where it would have

- been in normal flow: shifting it to the top, right, bottom, or left of where it would have been placed.
- This does not affect the position of surrounding elements; they stay in the position they would be in in normal flow.
- The space it would have occupied is preserved and continues to influence the layout of surrounding content.
- The element can potentially overlap other elements.

```
B {
   position: relative;
   top: 30px;
   left: 60px;
   background-color: fuchsia;
}
```

Absolute positioning places the element in relation to its containing element. It is taken out of normal flow, meaning that it does not affect the position of any surrounding elements (as they simply ignore the space it would have taken up). The element is positioned relative to its nearest containing block.

```
B {
    position: absolute;
    top: 30px;
    left: 60px;
```

```
        background-color: fuchsia;
    }
```

Fixed Positioning: is a form of absolute positioning that positions the element in relation to the browser window (viewport), as opposed to the containing element.
Elements with fixed positioning do not affect the position of surrounding elements.
Fixed elements are often used for menus that stay in the same place at the top, bottom, or side of a screen so they are always available, even when the content scrolls.

```
Position: fixed;
```

Floating: The `float` property moves an element as far as possible to the left or right, allowing the following content to wrap around it.
Floats are one of the primary tools of modern CSS based web design, used to create multicolumn layouts, navigation toolbars, and table-like alignment without tables.

```
Img { float: right; }
```

Key Behaviors of Floating Elements

- A floated element is like an island in a stream: they are not in the flow, but the stream has to flow around them. This behavior is unique to floated elements.
- Floats stay in the content area of the containing element: it is also important to note that the floated element is placed within the content area (the inner edges) of the element that contains it. It does not extend into the padding area.
- Margins are maintained: in addition, margins are held on all sides of the floated element. In other words, the entire element box, from outer edge to outer edge, is floated.

Floating Block Elements

- You must provide a width for floated block elements: If you do not provide a width value, the width of the floated block will be set to auto, which fills the available width of the browser window or other containing element.
- Elements do not float higher than their reference in the source: a floated block will float to the left or right relative to where it occurs in the source, allowing the following elements in the flow to wrap around it. It will stay below any block elements that precede it in the flow (in effect, it is “blocked” by them).

Side by Side Element: The float property is commonly used to place boxes next to each other.

Clearing Floated Elements: Applying the `clear` property to an element prevents it from appearing next to a floated element and forces it to start against the next available “clear” space below the float.

- `left`: the left-hand side of the box should not touch any other elements appearing in the same containing element.
- `right`: the right-hand side of the box will not touch elements appearing in the same containing element.
- `both`: neither the left nor right-hand sides of the box will touch elements appearing in the same containing element.
- `none`: elements can touch either side.

```
.clear { clear: left;}
```

Parent of Floated Elements: If a containing element only contains floated elements, some browsers will treat it as if it is zero pixels tall. The CSS solution adds two CSS rules to the containing element:

- The `overflow` property is given a value auto.
- The `width` property is set to 100%.

Fixed width layout designs do not change size as the user increases or decreases the size of their browser window. Measurements tend to be given in pixels.

- **Advantages**: Pixel values are accurate at controlling size and positioning of elements. Great control over the appearance and position of items.
- **Disadvantages**: If the user's screen is a much higher resolution than the designer's screen, the page can look smaller and text can be harder to read. You can end up with big gaps around the edge of a page.

Liquid layout designs stretch and contract as the user increases or decreases the size of their browser window. They tend to use percentages.

- Advantages: Pages expand to fill the entire browser window so there are no spaces around the page on a large screen. If the user has a small window, the page can contract to fit it. The design is tolerant of users setting.
- Disadvantages: If you do not control the width of sections of the page then the design can look very different than you intended. If the user has a wide window, lines of text can become very long. If the user has a very narrow window, you can end up with few words on each line.

9.5. Responsive Web Design

History of Web Design

- Up until the last few years, websites were designed so they would t well on the most common sizes of desktop and laptop screens.
- Early 2000s, ideas of fluid design and liquid layout: these techniques used percentage-based widths to allow a web page's design to flow to fit the width of the screen, so it could take advantage of the available space on wider screens.
- When mobile phones with Internet access first became available the easiest solution was to simply make separate mobile websites with a fixed page width that would fit on small screen.
- As more and more device sizes arrived on the market, it was no longer sustainable to create separate websites for every possible screen size.

Introduction of Media Queries

- Without having to create separate sites, how can a website be displayed with different layouts both on narrow and wide screens? Media queries.
- The CSS `@media` rule allows you to display different CSS styles based on device qualities without affecting the HTML.
- CSS3 proposed a detailed specification for media queries which includes precise queries based on media (device) features, such as width, height, and color capability.
- Media queries can rearrange your layout, but responsive design wouldn't work without a flexibility: this means that every horizontal measurement on your site needs to be in flexible units (ems or percentage) rather than inflexible pixels.

Why Responsive Design?

- Getting the right design on every device, you don't run the risk that users will be viewing the mobile version of a site on their desktop monitors, or vice versa.
- Less work, you only have to create one website, one design, one set of code, and one set of content.
- Optimized for search, a separate mobile site, with a separate set of URLs, can create issues with your site's placement in search results.

Viewport

- The `viewport` meta element is the key to making a responsive site work.
- Viewport: area on the computer or device screen where you are viewing a web page.
- Desktop viewport: browser window without the menus, toolbars, scrollbar, and everything else that's part of the browser itself.
- Mobile viewport: the viewport width is the same as the screen width.
- The viewport is different from the screen size.

```
<meta name="viewport" content="width=devicewidth, initial-scale=1">
```

- `width` attribute tells the browser how to scale the web page, for a responsive site, the value `width=device-width` tells the browser to render the page at full size, whatever the size may be.
- The `initial-scale` attribute tells the browser how to scale the web page when it's first loaded on the screen (the zoom factor).
- Using the value `initial-scale=1` means that the page will be rendered at the size determined by the width attribute, and will not be zoomed in or out.

Media Queries: allow you to apply different style declarations based on qualities of the device the website is being viewed on, most commonly the width of the viewport. Sort of if/then statement.

```
@media not|only mediatype and (mediafeature and|or|not mediafeature) {
    CSS code;
}
```

Following the `@media` are one or more expressions:

- `only`: cause those older browsers to ignore the whole query.
- `screen`: is the media type.
- `(min-width: 40em)`: media feature expression.
- curly braces surround all the CSS that will be applied if the entire media query is true.

There are three possibility to **use media queries**:

- Writing media queries inside the stylesheets
- Tell the browser that the entire stylesheet should only be applied if a media query is true, and ignored if the media query is not true.
- Include a media query as an attribute to the <style> element in the <head> of a page.

Media Features

- Viewport width and height (width, height);
- Screen width and height (device-width, device-height);
- Orientation, landscape or portrait (orientation);
- Ratio of the viewport (aspect-ratio);
- Ratio of the device screen (device-aspect-ratio);
- Resolution of the device screen (resolution).

Breakpoint: is the point at which you use a media query to change the design. It breaks your design into two (or more) variations.

- A design range is the range of screen sizes. Each design range gets a different variation of the design.
- The design needs to look good at any width, not just at certain points.

Designing Responsively

- **Progressive enhancement** is the idea that you start with the basics, and add on from there for browsers and devices that can handle more.
- **Designing with grids:** the design is made up of multiple columns of equal widths, with equal gutters (margins) between them, and everything on the page is based around those columns.
- **Design for small screen first**, it is much easier to create a layout and then make it bigger than it is to make a layout smaller.

10. Javascript

10.1. Introduction to JavaScript

JavaScript is the programming language of the Web, it adds interactivity and custom behaviors to a Web page.
JavaScript is a **high-level, dynamically typed, interpreted** programming language that is well-suited to object-oriented and functional programming styles.
It is traditionally a client-side scripting language, which means it runs on the user’s machine and not on the server.
Nowadays JavaScript is more and more also a server side language (Node.js).

How JavaScript make pages more interactive:

- Access the content: select any element, attribute or text from an HTML page;
- Modify content: add or remove any element, attribute or text in a HTML page;
- React to events: specify that a script should run when a specific event has occurred.

What JavaScript can do:

- Form validation: altering the contents of the page and blocking the form submission;
- Slideshow: display different images within the same space on a given page;
- Reload part of a page: request content and information from the server and inject it into the current document as needed, without reloading the entire page;
- Filtering data: help users to find the information they need by providing filters;
- Test for browsers’ features and capabilities: test for the device type and add more user-friendly styles and interaction methods based on the device type

What JavaScript Can’t Do: For security reasons browsers impose restrictions on the use of certain JS features that they do support:

- A JavaScript program can open new browser windows, but, to prevent **pop-up** abuse by advertisers, most browsers restrict this feature so that it can happen only in response to a user-initiated event, such as a mouse click.
- A JavaScript program can **close browser windows** that it opened itself, but it is not allowed to close other windows without user confirmation.
- A script **cannot read or modify** the content of documents loaded from **other tabs** or **windows**. Similarly, a script cannot register event listeners on pages on different tabs or windows.

Adding JavaScript to a Page:

- **Embedded script:** `<script> ... JavaScript code goes here </script>`
- **External scripts:** `<script src=”my_script.js”></script>`

Advantages of External Scripts

- **It simplifies the HTML files**: it helps keep content and behavior separate.
- When multiple web pages share the same JavaScript code, using the src attribute allows you to maintain only a **single copy** of that code.

- If a file of JavaScript code is shared by more than one page, it only needs to be downloaded once.
- Because the src attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ **code exported** by other web servers.

Execution of JavaScript Program:

- JavaScript program consists of all the JavaScript code in a web page (embedded and external scripts), which see the same Document object and they share the same set of global functions and variables.
  If a script defines a new global variable or function, that variable or function will be visible to any JavaScript code that runs after the script does.
- JavaScript programs are **loaded and executed in the same order as they appear in the document.**
- If a JavaScript program registers an event handler, i.e. a function, this is invoked and executed when the event occurs.
  Examples of events are: document loaded, user interactions (clicks, submission of a button form, …)

10.2. Core JavaScript

Case Sensitivity

- JavaScript is a case-sensitive language: this means that language keywords, variables, function names, and other identifiers must always be typed with a consistent capitalization of letters.
- Note that HTML is not case-sensitive.
- Many client-side JavaScript objects and properties have the same names as the HTML tags and attributes they represent. While these tags and attribute names can be typed in any case in HTML, in JavaScript they typically must be all lowercase.

Comments: // or /**/
Optional Semicolons.

JavaScript Data Types

- primitive types: numbers, strings of text, and Boolean;
- object types (e.g. array, function, …).
- Special JavaScript values `null` and `undefined` are primitive values, but they are not numbers, strings, or 72et72Attr.
- Any JavaScript value that is not a number, a string, a 72et72Att, or null or undefined is an object.
- The JavaScript interpreter performs automatic garbage collection for memory management.

Variables: `var i=3;`
If you don't specify an initial value for a variable with the var statement, the variable is declared, but its value is `undefined` until your code stores a value into it.

Null and Undefined

- `null` is a language keyword that is usually used to indicate the **absence of a value**.

- `undefined` value is the value of variables that have **not been initialized** and the value you get when you query the value of an object property or array element that does not exist.
- null and undefined both indicate an absence of value and can often be used interchangeably.

Numbers: all defined has float
String: + operator to concatenate strings
Booleans: `true` or `false`
&&, ||, !

JavaScript Objects: are associative arrays. Key:value

- `object.property`
  `object["property"]`
- In strongly typed languages, an object can have only a fixed number of properties, and the names of these properties must be defined in advance. In JavaScript this rule does not apply: a program can create any number of properties in any object.
- Creation: `var o={name:value,….}`
- `new o(par1,par2,…)`
- a function, the `this` keyword refers to the "owner" of the function
- Deleting Properties: `delete objectname.name`
  delete expression evaluates to true if the delete succeeded or if the delete had no effect.

Arrays:

- no fixed size
- access: a[0]
- a.length
- a.forEach(function(par1,par2,…) { ….});

Functions

- defined with the `function` keyword

The `window` Object: to manipulate the parts of the browser window
Dialog Boxes: The Window object provides three methods for displaying simple dialog boxes to the user:

- `alert()` displays a message to the user and waits for the user to dismiss the dialog.
- `confirm()` displays a message, waits for the user to click an OK or Cancel button and returns a 73et73Att value.
- `prompt()` displays a message, waits for the user to enter a string, and returns that string.

Timers: setTimeout() and setInterval() allow you to register a function to be invoked once or repeatedly after a specified amount of time has elapsed.

- `setTimeout()`: method schedules a function to run after a specified number of milliseconds elapses.
- `setInterval()`: is like setTimeout() except that the specified function is invoked repeatedly at intervals of the specified number of milliseconds.

They both take as input arguments the called function and the interval of time, expressed in milliseconds.

Scripts sometimes need to obtain information about the web browser in which they are running or the desktop on which the browser appears.
The `navigator` property of a Window object refers to a Navigator object that contains browser vendor and version number information. The Navigator object has four properties:

- `appName`: full name of the web browser;
- `appVersion`: browser vendor and version information;
- `userAgent`: string that the browser sends in its User-Agent HTTP header; platform: the operating system.

The `screen` property of a Window object refers to a Screen object that provides information about the **size** of the user's display and the number of **colors** available on it.

The `console` object provides access to the browser's debugging console. The specifics of how it works varies from browser to browser, but there is a de facto set of features that are typically provided

- `log(), trace(), debug(), info(), warn(), error()`: output log messages with increasing level of severity, together with additional information, e.g. the stack trace in the case of `trace()`
- `time(), timeEnd(), timeLog()`: starts and stops a timer and logs the time passed
- `assert()` : logs a message and stack trace to console if the first argument is `false`
- `dir()` logs a JavaScript representation of the specified object. If the object being logged is an HTML element, then the properties of its DOM representation are printed,
- `table()`: logs an array of objects as a table

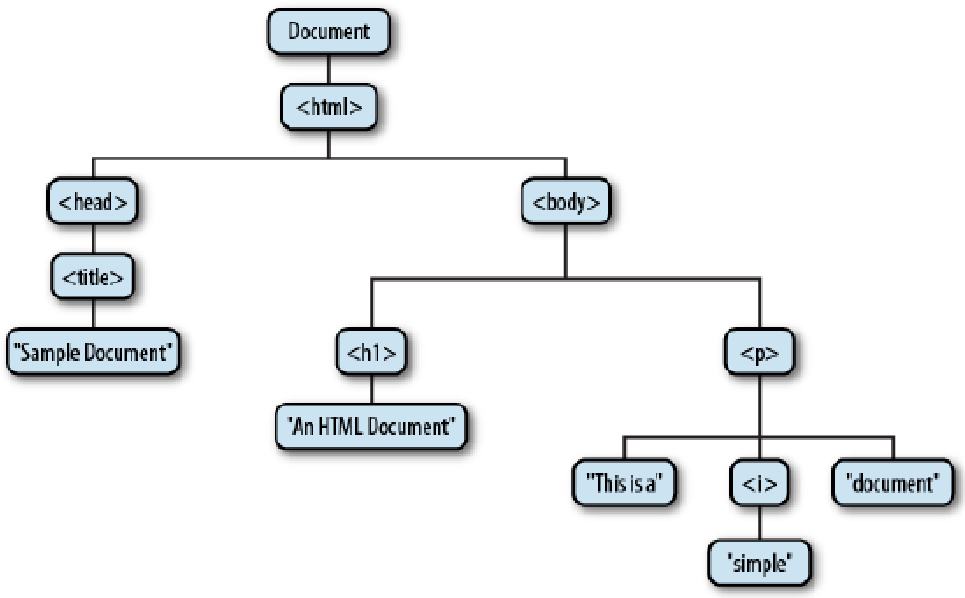
10.3. The Document Object Model

The DOM

- Every `Window` object has a document property that refers to a `Document` object, which represents the content of the window.
- The `Document` object is part of the Document Object Model, or DOM, which is the fundamental API for representing and manipulating the content of HTML.
- Recall: tree representation of an HTML document contains nodes representing HTML elements.

DOM Representation of a Document

- The DOM represents the HTML document as a tree.
- The root of the tree is the Document node that represents the entire document.
- The nodes that represent HTML elements are Element nodes.
- The nodes that represent text are Text nodes.
- Document, Element, and Text are subclasses of Node.



Document Elements

- with a specified `id` attribute;
- with a specified `name` attribute;
- with the specified `tag` name;
- with the specified `CSS class` or classes;
- matching the specified `CSS selector`

Selecting Elements by Id

- Recall: any HTML element can have an id attribute and its value must be unique within the document.
- You can select an element based on this unique id with the `getElementById()` method of the Document object.
  `Var section1 = `**`document.getElementById`**`("section1");`
- This is the simplest and most commonly used way to select elements.

Selecting Elements by Name

- Recall: the HTML name attribute is intended to assign names to form elements, and the value of this attribute is used when form data is submitted to a server. Unlike id, however, the value of a name attribute does not have to be unique: multiple elements may have the same name (radio buttons and checkboxes).
- To select HTML elements based on the value of their name attributes, you can use the getElementsByName() method of the Document object:
  ```
  var radiobuttons = document.getElementsByName("favorite_color");
  ```
- It returns a NodeList object that behaves like a read-only array of Element objects.

Selecting Elements by Type

- You can select all HTML elements of a specified type (or tag name) using the `getElementsByTagName()` method of the Document object.
  ```
  Var spans = document.getElementsByTagName("span");
  ```
- returns a NodeList object.

Selecting Elements by CSS Class and Selectors

- You can select all HTML elements of a specified class using the `getElementsByClassName()` method
  ```
  var warnings = log.getElementsByClassName("warning");
  ```
- `querySelectorAll()` allows you to access nodes of the DOM based on a CSS-style selector.
  ```
  Var sidebarPara = document.querySelectorAll(".sidebar p");
  var textInput = document.querySelectorAll("input[type='text']");
  ```

Node Object: Properties

- `parentNode`
- `childNodes`
- `firstChild, lastChild`
- `nextSibling, previousSibling`
- `nodeType`: the kind of node this is. Document nodes have the value 9. Element nodes have the value 1. Text nodes have the value 3. Comments nodes are 8 and Document-Fragment nodes are 11.
- `nodeValue`: the textual content of a Text or Comment node.
- `nodeName`: the tag name of an Element, converted to uppercase.

Document as a Tree of Element: When you are primarily interested in the Elements of a document instead of the text within them, you can treat a document as a tree of Element objects, ignoring Text and Comment nodes. Element properties are:
children, returns only Element objects.

- `firstElementChild, lastElementChild`
- `nextElementSibling, previousElementSibling`
- `childElementCount`: the number of element children.

Attributes as Element Properties

- The HTMLElement objects define read/write **properties** that mirror the HTML attributes of the **elements**.

- Example: to query the URL of an image, you can use the src property of the HTMLElement that represents the <img> element:
  ```
  var image = document.getElementById("myimage");
  var imgurl = image.src;
  ```
- The Element type also defines `getAttribute()` methods that you can use to query HTML attributes:
  ```
  var image = document.getElementById("myimage");
  var imgurl = image.getAttribute("src");
  ```
- Attribute values are all treated as strings, this means that `getAttribute()` never returns a number, 77et77Att, or object.
- `setAttribute()` methods that you can use to set HTML attributes.
- `77et77Attribute()` checks for the presence of a named attribute.
- `removeAttribute()` removes an attribute entirely.

Manipulating Nodes: It is possible to alter a document at the level of individual nodes. The Document type defines methods for creating Element and Text objects, and the Node type defines methods for inserting, deleting, and replacing nodes in the tree.

Creating Nodes: createElement()

- `var newDiv = document.`**`createElement`**`("div");`
- `var ourText = document.`**`createTextNode`**`("Put text here.");`
- Once you create an element in this way, that new element remains "floating" until you add it to the document.

Inserting Nodes: Once you have a new node, you can insert it into the document.

- `appendChild()` is invoked on the Element node that you want to insert into, and it inserts the specified node so that it becomes the last Child of that node.
- `insertBefore()` is like appendChild(), but it takes two arguments: the first argument is the node to be inserted, the second argument is the node before which that node is to be inserted. This method is invoked on the node that will be the parent of the new node, and the second argument must be a child of that parent node.

Removing and Replacing Nodes

- The `removeChild()` method removes a node from the document tree. Invoke the method on the parent node (not the node that you want to remove) and pass the child node that is to be removed as the method argument.
- The `replaceChild()` method removes one child node and replaces it with a new one. Invoke this method on the parent node, passing the new node as the first argument and the node to be replaced as the second argument.

10.4. Handling events

JavaScript Timeline

- The web browser creates a Document object and begins parsing the web page.
- When the HTML parser encounters <script> elements, it adds those elements to the document and then executes the script. These scripts are executed synchronously, and the parser pauses while the script downloads (if necessary) and runs.
- The document is completely parsed at this point, but the browser may still be waiting for additional content, such as images, to load. When all such content finishes loading, and when all scripts have loaded and executed, the document.readyState property changes to complete and the web browser fires a load event on the Window object.
- From this point on, event handlers are invoked asynchronously in response to user input events, network events, timer expirations, and so on.

Events are occurrences that a web browser will notify your JavaScript program about
If a JavaScript application cares about a particular type of event, it can register one or more functions to be invoked when events of that type occur.
Event type is a string that specifies what kind of event occurred, examples are: mousemove, keydown, road, …
Event target is the object on which the event occurred or with which the event is associated.
When we speak of an event, we must specify both the type and the target: for example a load event on a Window, or a click event on a <button> Element.
Window, Document, and Element objects are the most common event targets in client-side JavaScript applications.
Event Handler or event listener is a function that handles or responds to an event.

- Applications register their event handler functions with the web browser, specifying an event type and an event target.
- When an event of the specified type occurs on the specified target, the browser invokes the handler.
- When event handlers are invoked for an object, we sometimes say that the browser has “fired”, “triggered”, or “dispatched” the event.

Event object is an object that is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function.
All event objects have a type property that specifies the event type and a target property that specifies the event target.
Each event type defines a set of properties for its associated event object, for example the object associated with a mouse event includes the coordinates of the mouse pointer.

Types of Events:

- **Mouse events are** generated when the user moves or clicks the mouse over a document.
  - The **mousemove** event is triggered any time the user moves or drags the mouse.
  - The **mousedown** and **mouseup** events are triggered when the user presses and releases a mouse button.
  - The click event it is triggered on any document element, not just form elements, when a click occurs.
  - The second click event will be followed by a **dblclick** event.

  - When the user moves the mouse so that it goes over a new element, the browser fires a mouseover event on that element. When the mouse moves so that it is no longer over an element, the browser fires a **mouseout** event on that element.
  - When the user rotates the mouse wheel, browsers trigger a **mousewheel** event.
- **Keyboard events** are triggered on whatever document element has keyboard focus, and they bubble up to the document and window.
  - The **keydown** and **keyup** events are low-level keyboard events: they are triggered whenever a key is pressed or released.
  - When a **keydown** event generates a printable character, an additional keypress event is triggered after the **keydown** but before the **keyup**.
- **Form Events**
  - Form elements typically fire a click or change event when the user interacts with them, and you can handle these events by defining an **onclick** or **onchange** event handler.
  - In general, form elements that are buttons fire a click event when activated (even when this activation happens through the keyboard rather than via an actual mouse click).
  - Other form elements fire a change event when the user changes the value represented by the element. This happens when the user enters text in a text field or selects an option from a drop-down list. Note that this event is not fired every time the user types a key in a text field. It is fired only when the user changes the value of an element and then moves the input focus to some other form element.
  - Radio buttons and checkboxes are buttons that have a state, and they fire both click and change events; the change event is the more useful of the two.
  - Form elements also fire a **focus** event when they receive keyboard focus and a blur event when they lose it.
  - Each Form element has an **onsubmit** event handler to detect form submission and an **onreset** event handler to detect form resets.
  - **Form validation:** the **onsubmit** handler is triggered just before the form is submitted by a click on a submit button; it can cancel the submission by returning **false**.
  - The **onreset** event handler is invoked just before the form is reset, and it can prevent the form elements from being reset by returning **false**, it is used to make the user confirm the reset.
- **Window events** represent occurrences related to the browser window itself.
  - The **load** event is fired when a document and all of its external resources (such as images, style sheets, or scripts) are fully loaded and displayed to the user.
  - The **unload** event is the opposite of load: it is triggered when the user is navigating away from a document. An unload event handler might be used to save the user's state.
  - The **beforeunload** event is similar to unload but gives you the opportunity to ask the user to confirm that they really want to navigate away from your web page.
  - The **resize** and **scroll** events are fired on a window when the user resizes or scrolls the browser window.

Registering Event Handlers: 2 ways

- set a **property** on the object or document element that is the event target:
  - You can set an **event handler property** in JavaScript code;

```
window.onload = function() {
        var 80et = document.getElementById("address");
        80et.onsubmit = function() { return validate(this); }
    }
```

- For document elements, you can set the corresponding **attribute** directly in HTML.

  ```
  <button onclick="alert('Thank you');">Click Here</button>
  ```

- pass the handler to a **method** of the object or element, for handler registration:
  - Any object that can be an event target (includes the Window, Document and all Elements objects), defines a method named `addEventListener()`; takes two mandatory arguments:
    - the event **type** (string without the on prefix) for which the handler is being registered;
    - the **function** that should be invoked when the specified type of event occurs

    ```
    <button id="mybutton">Click me</button>
    <script>
        var b = document.getElementById("mybutton");
        b.onclick = function() { alert("Thanks for clicking me!"); };
        b.addEventListener("click", function() { alert("Thanks again!"); });
    </script>
    ```

    It allows **adding more than a single handler** for an event. This is particularly useful for AJAX libraries, JavaScript modules, or any other kind of code that needs to work well with other libraries/extensions.

    It gives you finer-grained control of the phase when the listener is activated (capturing vs. bubbling).

    It works on **any DOM element**, not just HTML elements.
  - a different method, named `attachEvent()`, is paired with a `removeEventListener()` method which removes an event handler function from an object. It is useful to temporarily register an event handler and then remove it soon afterward.

    ```
    Document.removeEventListener("mousemove", handleMouseMove);
    document.removeEventListener("mouseup", handleMouseUp);
    ```

10.5. Form Validation

Form validation: when you enter data in a Web page, the Web application checks it to see that the data is correct. If correct, the application allows the data to be submitted to the server and (usually) saved in a database; if not, it gives you an error message explaining what corrections need to be made.

There are three main reasons to validate forms:

- To get the right data, in the right format: Web applications won't work properly if the user's data is stored in the incorrect format, if they don't enter the correct information, or omit information altogether.
- To protect the users' accounts by forcing them to enter secure passwords.
- To protect ourselves, there are many ways that malicious users can misuse unprotected forms to damage the application they are part of.

There are **two different types of form validation**:

- **Client-side validation** occurs in the browser, before the data has been submitted to the server. This is more user-friendly than server-side validation as it gives an instant response. This can be further subdivided:
  - **JavaScript validation** is coded using JavaScript. It is completely customizable.
  - Built-in form validation using **HTML5 form validation** features. This has better performance, but it is not as customizable as JavaScript.
- **Server-side validation** occurs on the server, after the data has been submitted. It is used to validate the data before it is saved into the database. If the data fails authentication, a response is sent back to the client to tell the user what corrections to make. Server-side validation is not as user-friendly as client-side validation, as it does not provide errors until the entire form has been submitted. However, server-side validation is the application's last line of defense against incorrect or even malicious data.

Developers use a combination of client-side and server-side validation.

HTML 5 Validation: One of the features of HTML5 is the ability to validate most user data without relying on scripts. This is done by using validation attributes on form elements, which allow you to specify rules for a form input. If the entered data follows all those rules, it is considered valid; if not, it is considered invalid.

When an element is valid:

- The element matches the :valid CSS pseudo-class; this will let you apply a specific style to valid elements.
- If the user tries to send the data, the browser will submit the form, provided there is nothing else stopping it from doing so (e.g., JavaScript).

When an element is invalid:

- The element matches the :invalid CSS pseudo-class; this will let you apply a specific style to invalid elements.
- If the user tries to send the data, the browser will block the form and display an error message.

HTML5 provides **the constraint validation API** to check and customize the state of a form element. Among other things, it's possible to change the text of the error message with the `setCustomValidity()` method.

```
Var email = document.getElementById(“provide_email”);
email.addEventListener(“input”, function (event) {
    if (email.validity.typeMismatch) {
        email.setCustomValidity(“Please insert an email address!”);
    } else {
        email.setCustomValidity(“”);
    }
});
```

Validating Forms without a Built-in API. To validate a form, you have to ask yourself a few questions:

- What **kind of validation** should I perform? You need to determine how to validate your data: string operations, type conversion, regular expressions, etc. Remember that form data is always text and is always provided to your script as strings.
- What should I do if the form **does not validate?** You have to decide how the form will behave: should you highlight the fields which are in error? Should you display error messages?
- How can I **help the user to correct** invalid data? In order to reduce the user’s frustration, it’s very important to provide as much helpful information as possible in order to guide them in correcting their inputs. You should offer up-front suggestions so they know what’s expected, as well as clear error messages.

10.6. AJAX – Scripted HTTP

Synchronous vs Asynchronous

- When a browser comes across a `<script>` tag, it will typically stop processing the rest of the page until it has loaded and processed it. This is an example of **synchronous processing model**
- This can need time, e.g. if for example the script requires data from the server. Then you need to further wait the answer from the server
- **AJAX** instead uses an **asynchronous (non-blocking) processing model**, i.e. the user can do other things while the web browser is waiting for the data to load, speeding up the user experience.
- When the server responds with the data, and event is fired, which can call a function that processes the data. This function can update only one element of the page, instead of the whole page.

AJAX:

- Historically, **AJAX** stands for **Asynchronous JavaScript And XML**, an acronym containing the technologies used at the time (JavaScript and XML). AJAX now indicates a group of technologies that offer asynchronous functionality in the browser. The key feature of an Ajax application is that it uses scripted HTTP to initiate data exchange with a web server without causing pages to reload.
- AJAX uses the **XMLHttpRequest** object to communicate with servers. It can send and receive information in various formats, including JSON, XML, HTML, and text files. AJAX's most appealing characteristic is its "asynchronous" nature, which means it can communicate with the server, exchange data, and update the page without having to refresh the page.
- The ability to **avoid page reloads** results in responsive web applications.
- A web application might use Ajax technologies to log user interaction data to the server or to improve its start-up time by displaying only a simple page at first and then downloading additional data and page components on an as-needed basis.

Using XMLHttpRequest:

- Each instance of this class represents a single request/response pair, and the properties and methods of the object allow you to specify request details and extract response data.
```
Var request = new XMLHttpRequest();
```
- An HTTP request consists of four parts: the HTTP request method; the URL being requested; an optional set of request headers, which may include authentication information; an optional request body.
- The HTTP response sent by a server has three parts: a numeric and textual status code that indicates the success or failure of the request; a set of response headers; the response body

Specifying the Request

- After creating an XMLHttpRequest object, the next step in making an HTTP request is to call the **`open()`** method of your XMLHttpRequest object:
```
request.open('GET', 'http://www.example.org/some.file');
```
- The first parameter of the `open()` method is the **HTTP request method**. Keep the method all-capitals as per the HTTP standard, otherwise some browsers might not process the request.

- The second parameter is the **URL** that is the subject of the request. This is relative to the URL of the document that contains the script that is calling `open()`. As a security feature, you cannot call URLs on third-party domains. Be sure to use the exact domain name on all of your pages or you will get a “permission denied” error.
- To set the request headers, if any, you can use the setRequestHeader() method. POST requests, for example, need a “Content-Type” header to specify the MIME type of the request body:

```
request.setRequestHeader(“Content-Type”, “text/plain”);
```

- If you call setRequestHeader() multiple times for the same header, the new value does not replace the previously specified value: instead, the HTTP request will include multiple copies of the header or the header will specify multiple values.
- The final step in making an HTTP request with XMLHttpRequest is to send it off to the server, with the send() method:

```
request.send();
```

Encoding the Request Body

- Recall: HTTP POST requests include a request body that contains data the client is passing to the server.
- **Form-encoded requests:** URI encoding (replacing special characters with hexadecimal escape codes) on the name and value of each form element, separate the encoded name and value with an equals sign, and separate these name/value pairs with ampersands.

```
Find=pizza&zipcode=02134&radius=1km
```

  form data encoding format has a formal MIME type

```
application/x-www-form-urlencoded
```

- **Json-encoded requests:**

```
request.setRequestHeader(“Content-Type”, “application/json”);
```

Cross Origin Resource Sharing (CORS)

- The XMLHttpRequest object can normally issue HTTP requests only to the server from which the document that uses it was downloaded, (same-origin security policy). That is, browsers do not load AJAX responses from other domains. There are different workarounds, among those we present CORS.
- Cross-Origin Resource Sharing, (CORS): is a mechanism that uses additional HTTP headers to let a user agent gain permission to access selected resources from a server on a different origin (domain) than the site currently in use. A user agent makes a cross-origin HTTP request when it requests a resource from a different domain, protocol, or port than the one from which the current document originated.
- Example: A HTML page served from http://domain-a.com makes an <img> src request for http://domain-b.com/image.jpg.
- The Cross-Origin Resource Sharing standard works by adding **new HTTP headers** that allow servers to describe the set of origins that are permitted to read that information using a web browser.
- Cross-origin requests do not normally include any user credentials: username and password, cookies, HTTP authentication tokens, …

Retrieving the Response

- The same object that sent the request deals with the answer. When you send the request, you should provide the name of a JavaScript function to handle the response:

```
request.onload = nameOfTheFunction;
```

- The function needs to check the request's state. If the state has the value of XMLHttpRequest.DONE, that means that the full server response was received and it's OK to continue processing it.
- The full list of the `readyState` values is as follows:
    - 0 (uninitialized) or (request not initialized), open() has not been called yet;
    - 1 (loading) or (server connection established), open() has been called;
    - 2 (loaded) or (request received), headers have been received;
    - 3 (interactive) or (processing request), the response body is being received;
    - 4 (complete) or (request finished and response is ready), the response is complete.
- Next, check the response code of the HTTP response (successful = 200).

```
Request.status == 200
```

- After checking the state of the request and the HTTP status code of the response, you have two options to access that data:
- `request.responseText` – returns the server response as a string of text;
- `request.responseXML` – returns the response as an XMLDocument object you can traverse with JavaScript DOM functions.

Types of receivable data

- **HTML**
    - Pros: easy to write, request and display; The data sent from the server can go straight into the page, no need to process it (e.g. through JavaScript).
    - Cons: the server must produce the HTML in a format that is ready for use on our page; it is not wellsuited for use in applications other than web browsers, i.e. no good data portability.
- **XML**
    - Pros: it is a flexible data format, and can represent complex structures. It works well with different platforms and applications. It is processed using the same HTML DOM methods.
    - Cons: it is considered a verbose language, the tags add a lot of extra characters to the file being sent; it can require a lot of code to be processed.
- **JSON**
    - Pros: it can be called from any domain (CORS); more concise than the other twos; commonly used with JavaScript (it has gained wide use across web applications).
    - Cons: the syntax is very strict (unlike HTML), i.e. a missed quote, comma or colon can "break" the file; since it is still JavaScript, it can contain malicious content, therefore only use JSON that has been produced by trusted sources

Loading JSON with AJAX

- The server sends JSON data to a web browser as a **string**
- When it reaches the browser, a script must convert the string into a KavaScript object — **deserialization of the object** — through the `parse()` method of the JSON object. It is a global object, so you do not need to instantiate it.
- Once the string has been parsed, the script can access the data in the object, and use it to create HTML.
- The HTML is added to the page using the `innerHTML` property, thus only use it when you are confident that it does not contain malicious code.
- The method `JSON.stringify()` converts objects into a string using JSON notation, thus to send the object from the browser back to the server, a.k.a. **serialization of the object**.

10.7. jQuery

10.7.1. Introduction to jQuery

jQuery is a fast, small, and featurerich **JavaScript library** (http:// jquery.com/).
Offers a simple way to achieve a variety of common JavaScript tasks quickly and consistently across all major browsers and without any fallback code needed.
It allows to:

- Select elements in a simpler and more powerful way with CSS-style selectors;
- Manipulate the DOM tree;
- Attach event listeners without any fallback code.

jQuery is a lightweight, "write less, do more", JavaScript library. Its aim is to make easier to use JS on a website.
It allows to perform many tasks, that otherwise would have required many lines of JavaScript code, in single lines of code.
There are many other JS libraries available. jQuery is one of the most popular and extendable.
Some of jQuery's features are: HTML/DOM manipulation, CSS manipulation, HTML event methods, Effect and animations, Ajax, Utilities.

jQuery Basics

- The jQuery library defines a single global function named `jQuery()`, with the symbol $ as a shortcut for it.
  ```
  Var divs = $("div");
  ```
- The value returned by this function represents a set of zero or more DOM elements and is known as a jQuery object.
- jQuery objects define many methods for operating on the sets of elements they represent.
  ```
  $("p.details").css("background-color", "yellow").show("fast");
  ```

jQuery objects are array-like and they have the following properties:

- The **length** property;
- The **selector** property is the selector string (if any) that was used when the jQuery object was created.
- The **context** property is the context object that was passed as the second argument to $(), or the Document object otherwise.
- The **jquery** property: testing for the existence of this property is a simple way to distinguish jQuery objects from other arraylike objects.

Queries and Query Results

- When you pass a CSS selector string to `$()`, it returns a jQuery object that represents the set of matched elements.
- jQuery objects are array-like: they have a length property and you can access the contents of the jQuery object using standard squarebracket array notation:
  ```
  $("body").length
  $("body")[0]
  ```
- If you prefer not to use array notation with jQuery objects, you can use the `size()` method instead of the length property and the `get()` method instead of indexing with square brackets. If you need to convert a jQuery object to a true array, call the `toArray()` method.

Creating DOM Elements: If a string is passed as the parameter to `$()`, jQuery examines the string to see if it looks like HTML (i.e., it starts with `<tag ... >`). If not, the string is interpreted as a selector expression. But if the string is a HTML snippet, jQuery attempts to create new DOM elements, then a jQuery object is created and returned.

```
Var img = $("<img/>",
    { src: url,
      css: {borderWidth:5},
      click: handleClick
    });
```

Each() Method

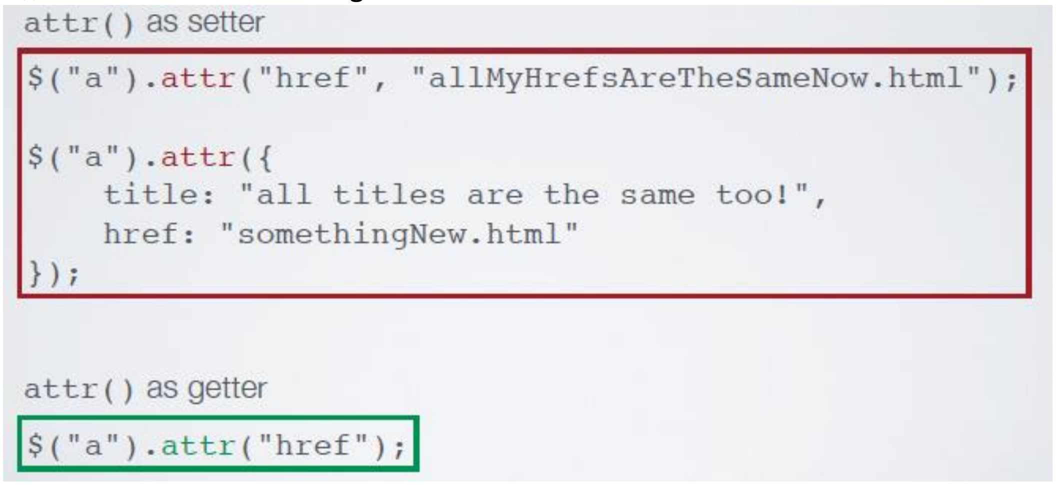
- If you want to loop over all elements in a jQuery object, you can call the `each()` method instead of writing a for loop. The `each()` method is similar to the `forEach()` array method.
- It expects a callback function as its sole argument, and it invokes that callback function once for each element in the jQuery object.
- Despite the power of the `each()` method, it is not very commonly used, since jQuery methods usually iterate implicitly over the set of matched elements and operate on them all.

10.7.2. jQuery Getters and Setters

jQuery objects allow you to get or set the value of HTML attributes, CSS styles or element content:

- jQuery uses a **single method** as both getter and setter. If you pass a new value to the method, it sets that value; if you don't specify a value, it returns the current value.
- When used as setters, these methods set values **on every element** in the jQuery object, and then **return** the **jQuery object** to allow method chaining.
- When used as getters, these methods query **only the first element** of the set of elements and **return** a **single value**, therefore they can only appear at the end of a method chain.
- When used as setters, these methods often **accept object** arguments. In this case, each property of the object specifies a name and a value to be set.
- When used as setters, these methods often **accept functions** as values. In this case, the function is invoked to compute the value to be set.

The `attr()` method acts as both a getter and a setter.



attr() as setter

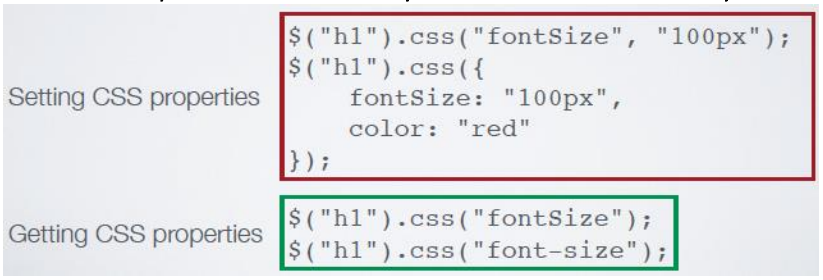
```
$("a").attr("href", "allMyHrefsAreTheSameNow.html");

$("a").attr({
    title: "all titles are the same too!",
    href: "somethingNew.html"
});
```

attr() as getter

```
$("a").attr("href");
```

Getting and Setting CSS Attributes

- The `css()` method is similar to the `attr()` method, but it works with the CSS styles of an element.
- When querying style values, `css()` returns the current (or computed) style of the element: the returned value may come from the style attribute or from a stylesheet.



Setting CSS properties

```
$("h1").css("fontSize", "100px");
$("h1").css({
    fontSize: "100px",
    color: "red"
});
```

Getting CSS properties

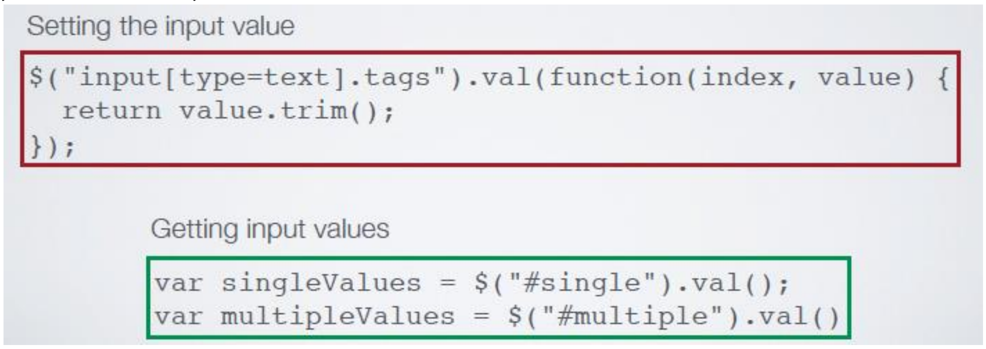
```
$("h1").css("fontSize");
$("h1").css("font-size");
```

Getting and Setting CSS Classes

- jQuery defines `addClass()` and `removeClass()` to add and remove classes from the selected elements.
- `toggleClass()` adds classes to elements that don't already have them and removes classes from those that do.
- `hasClass()` tests for the presence of a specified class.

```
Var h1 = $("h1");
h1.addClass("big");
h1.removeClass("big");
h1.toggleClass("big");
if (h1.hasClass("big")) {
    ...
}
```

Getting and Setting HTML Form Values: `val()` is a method for setting and querying the value attribute of HTML form elements and also for querying and setting the selection state of checkboxes, radio buttons, and `<select>` elements.



Setting the input value

```
$("input[type=text].tags").val(function(index, value) {
  return value.trim();
});
```

Getting input values

```
var singleValues = $("#single").val();
var multipleValues = $("#multiple").val()
```

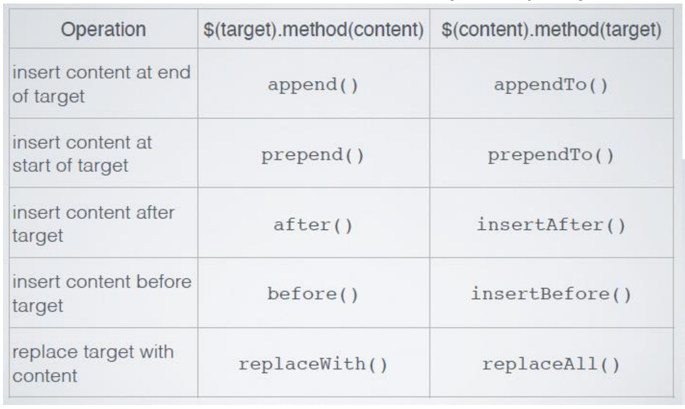
Getting and Setting Element Content

- The `text()` and `html()` methods query and set the plain-text or HTML content of an element or elements.
- When invoked with no arguments, `text()` returns the plain-text content of all descendant text nodes of all matched elements.
- If you invoke the `html()` method with no arguments, it returns the HTML content of just the first matched element.
- If you pass a string to `text()` or `html()`, that string will be used for the plain-text or HTML-formatted text content of the element, and it will replace all existing content.

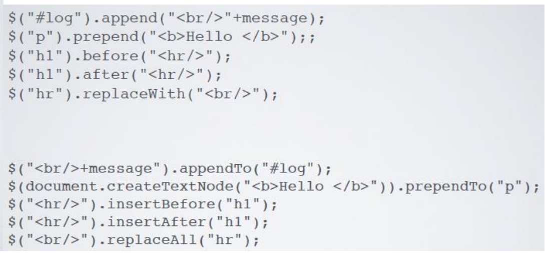
10.7.3. Altering the DOM Structure

Inserting and Replacing Elements

- Each of the following methods takes an argument that specifies the content that is to be inserted into the document. This can be a string of plain text or of HTML to specify new content, or it can be a jQuery object or an Element or text Node.
- The insertion is made into or before or after or in place of (depending on the method) each of the selected elements.
- If the content to be inserted is an element that already exists in the document, it is moved from its current location. If it is to be inserted more than once, the element is cloned as necessary.
- These methods all return the jQuery object on which they are called.



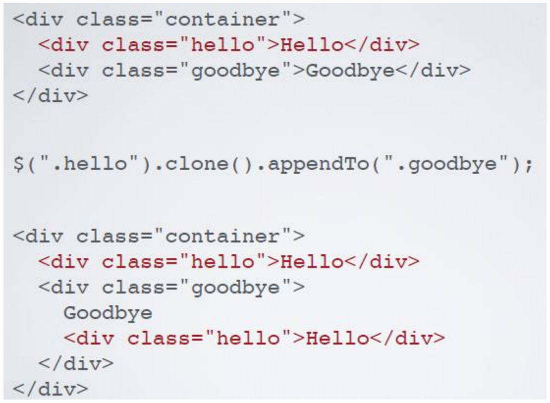
Operation	$(target).method(content)	$(content).method(target)
insert content at end of target	`append()`	`appendTo()`
insert content at start of target	`prepend()`	`prependTo()`
insert content after target	`after()`	`insertAfter()`
insert content before target	`before()`	`insertBefore()`
replace target with content	`replaceWith()`	`replaceAll()`



```
$("#log").append("<br/>"+message);
$("p").prepend("<b>Hello </b>");;
$("h1").before("<hr/>");
$("h1").after("<hr/>");
$("hr").replaceWith("<br/>");


$("<br/>+message").appendTo("#log");
$(document.createTextNode("<b>Hello </b>")).prependTo("p");
$("<hr/>").insertBefore("h1");
$("<hr/>").insertAfter("h1");
$("<br/>").replaceAll("hr");
```

Copying Elements

- If you insert elements that are already part of the document, those elements will simply be moved, not copied, to their new location.
- If you are inserting the elements in more than one place, jQuery will make copies as needed.
- If you want to copy elements to a new location instead of moving them, you must first make a copy with the **clone()** method. Clone() makes and returns a copy (jQuery object) of each selected element (and of all of the descendants of those elements).



```
<div class="container">
  <div class="hello">Hello</div>
  <div class="goodbye">Goodbye</div>
</div>


$(".hello").clone().appendTo(".goodbye");


<div class="container">
  <div class="hello">Hello</div>
  <div class="goodbye">
    Goodbye
    <div class="hello">Hello</div>
  </div>
</div>
```

Wrapping Elements: jQuery defines three wrapping functions.

- wrap() wraps each of the selected elements.
- wrapInner() wraps the contents of each selected element.
- wrapAll() wraps the selected elements as a group.

These methods are usually passed a newly created wrapper element or a string of HTML used to create a wrapper.

```
$(“h1”).wrap(document.createElement(“I”));
$(“.inner”).wrapInner(“<div class=’new’></div>”);
$(“.inner”).wrapAll(“<div class=‘new’></div>”);
```

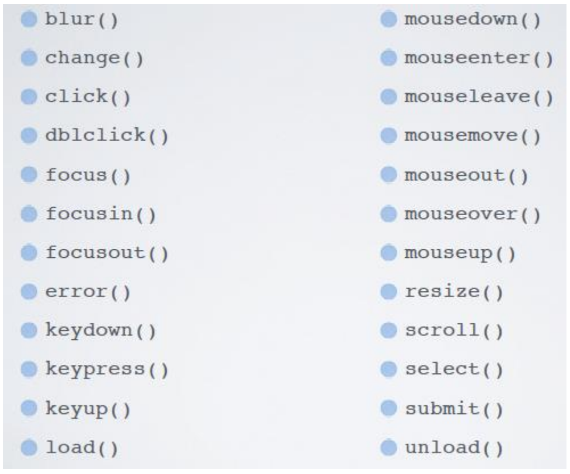
Deleting Elements: jQuery defines several methods for deleting elements.

- **empty()** removes all children of each of the selected elements.
- **remove()** removes the selected elements (together with their event handlers and data) from the document. If you pass an argument, that argument is treated as a selector, and only elements of the jQuery object that also match the selector are removed.
- **detach()** method works like remove() but does not remove event handlers and data. Detach() may be more useful when you want to temporarily remove elements from the document for later reinsertion.
- **unwrap()** method performs element removal in a way that is the opposite of the wrap() or wrapAll() method: it removes the parent element of each selected element without affecting the selected elements or their siblings. That is, for each selected element, it replaces the parent of that element with its children.

10.7.4. Handling Events with jQuery

Simple Event Handler Registration

- jQuery defines simple event-registration methods for each of the commonly used and universally implemented browser events.
- To register an event handler for click events, for example, just call the click() method:
```
$("p").click(function() { $(this).css("background-color", "gray"); });
```
- Calling a jQuery event-registration method registers your handler on all of the selected elements. This is typically much easier than one-at-a-time event handler registration with `addEventListener()`.



● blur()	● mousedown()
● change()	● mouseenter()
● click()	● mouseleave()
● dblclick()	● mousemove()
● focus()	● mouseout()
● focusin()	● mouseover()
● focusout()	● mouseup()
● error()	● resize()
● keydown()	● scroll()
● keypress()	● select()
● keyup()	● submit()
● load()	● unload()

jQuery Event Handler

- The method `bind()` binds a handler for a named event type to each of the elements in the jQuery object. Using `bind()` allows you to use more advanced event registration features.
- `bind()` expects an event type string as its first argument and an event handler function as its second.
```
$("p").click(f);      $("p").bind("click", f);
```
- If the first argument is a space separated list of event types, then the handler function will be registered for each of the named event types.
```
$("a").hover(f);
$("a").bind("mouseenter mouseleave", f);
```

Deregistering Event Handlers

- After registering an event handler with `bind()` (or with any of the simpler event registration methods), you can deregister it with `unbind()`.
- `unbind()` only deregisters event handlers registered with `bind()` and related jQuery methods (not with `addEventListener()`).
- With no arguments, `unbind()` deregisters all event handlers (each event for each element):

```
$("*").unbind()
```

- With string arguments, all handlers for the named event type are unbound from all elements in the jQuery object:

```
$("a").unbind("mouseover mouseout");
```

10.7.5. AJAX with jQuery

jQuery provides several methods for AJAX functionality. With these, it is possible to request text, HTML, XML, or JSON from remote servers using both HTTP GET and POST.
Writing regular AJAX code can be tricky, because different browsers have different syntax for AJAX implementation. Thus, it may be necessary to write extra code to test for different browsers. jQuery takes care of this.

AJAX Function

- The `jQuery.ajax()` function performs asynchronous HTTP requests. It underlies all Ajax requests sent by jQuery. It is often unnecessary to directly call this function, as several higher-level alternatives are available.
- `ajax()` accepts a single argument: an **options object** whose properties specify the details about how the AJAX request is to be performed.
- By **default**, data passed in to the data option as an object will be processed and transformed into a query string, fitting to the default content-type "`application/xwww- form-urlencoded`".

```
$.ajax({
  method: "POST",
  url: "some.jsp",
  data: { name: "John", location: "Boston" }
})
  .done(function( msg ) {
    alert( "Data Saved: " + msg );
  });
```

AJAX Utility Functions – get()

- `jQuery.get()`, load data from the server using a HTTP GET request.
- GET is basically used for getting data from a server. It may also return cached data.

```
$.get(URL, callback);
```

- The required URL parameter specifies the URL we wish to request.
- The optional callback parameter is the name of a function to be executed if the request succeeds. The callback has two parameters: the content of the page requested, and the status of the request

```
$.get("test.jsp", { name: "John", time: "2pm" } )
  .done(function(data) {
    alert("Data Loaded: " + data);
  });
```

AJAX Utility Functions – post()

- `jQuery.post()`, load data from the server using a HTTP POST request. URL specifies the URL we wish to request

```
                        $.post(URL, data, callback)
```

- The optional data parameters specifies some data to send along with the request.
- The optional callback parameter is the name of a function to be executed if the request succeeds.

```
      $.post("test.jsp", { name: "John", time: "2pm" })
        .done(function(data) {
          alert("Data Loaded:" + data);
        });
```

AJAX Utility Functions – getScript(): `jQuery.getScript()`, load a JavaScript file from the server using a GET HTTP request, then execute it.

```
$.getScript("ajax/test.js", function( data, textStatus, jqxhr) {
  console.log(data); // Data returned
  console.log(textStatus); // Success
  console.log(jqxhr.status); // 200
  console.log("Load was performed.");
});
```

The load() Method

- `load()` is a simple but powerful AJAX method. It loads data from a server and puts it into a selected element. Its syntax:

```
                  $(selector).load(URL,data,callback);
```

- The `URL` parameter specifies the URL you want to load
- The `selector` specifies the elements where the returned data will be loaded
- The optional `data` parameter specifies a set of querystring key/value pairs to send along with the request.
- The optional `callback` parameter is the name of the function to be executed after the `load()` method is completed and the data are returned.
- The `load()` method with an URL as argument will asynchronously load the content of that URL and then insert that content into each of the selected elements, **replacing** any content that is already there.

```
               $("#result").load("ajax/test.html");
```

- The `load()` method, allows you to specify a fragment of the document to be inserted.

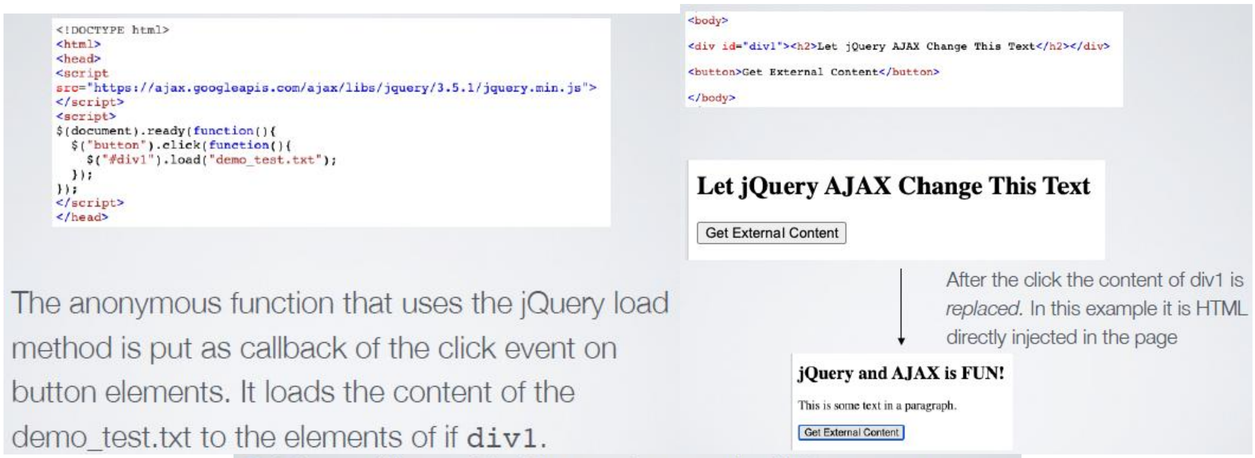
```
          $("#result").load("ajax/test.html #container");
```

- The POST method is used if data is provided as an object; otherwise, GET is assumed.

```
$("#address").load("address.jsp", { zipcode:"02134", country:"IT" });
```

- An optional argument to load() is a **callback** function that will be invoked when the AJAX request completes successfully or unsuccessfully.

Load text into a div



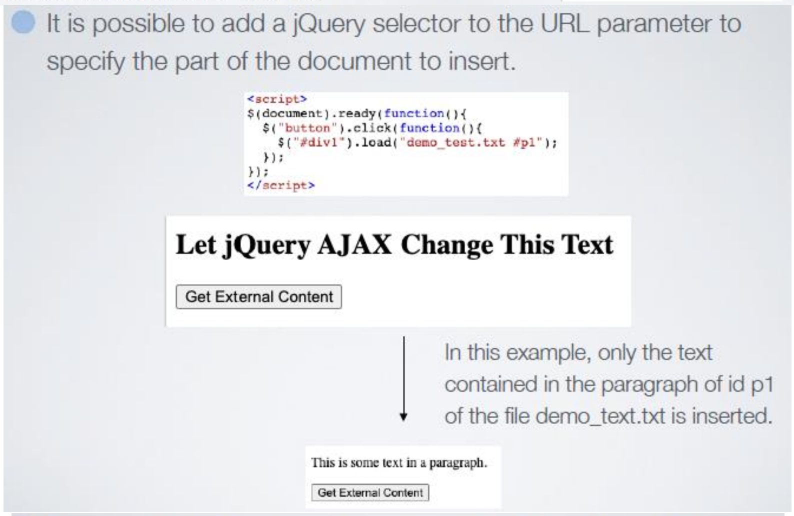
```
<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js">
</script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").load("demo_test.txt");
  });
});
</script>
</head>
```

```
<body>

<div id="div1"><h2>Let jQuery AJAX Change This Text</h2></div>

<button>Get External Content</button>

</body>
```

The anonymous function that uses the jQuery load method is put as callback of the click event on button elements. It loads the content of the demo_test.txt to the elements of if `div1`.

After the click the content of div1 is *replaced*. In this example it is HTML directly injected in the page

jQuery and AJAX is FUN!
This is some text in a paragraph.
Get External Content



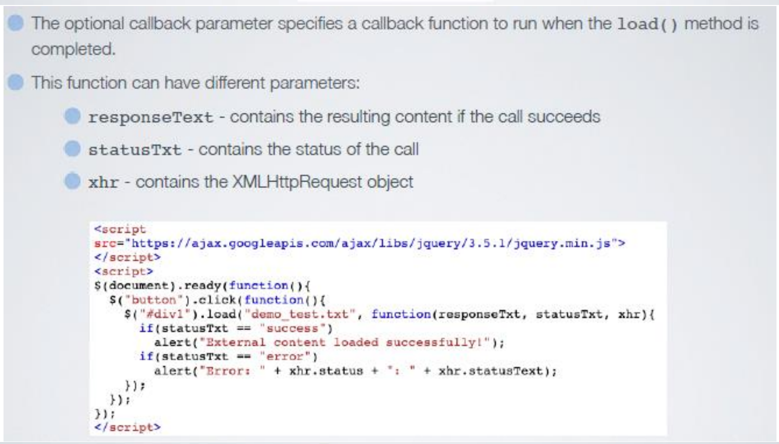
- It is possible to add a jQuery selector to the URL parameter to specify the part of the document to insert.

```
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").load("demo_test.txt #p1");
  });
});
</script>
```

Let jQuery AJAX Change This Text
Get External Content

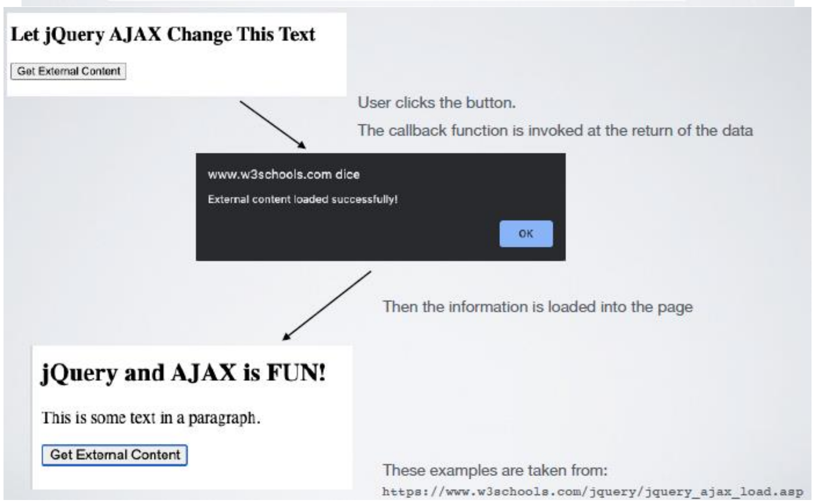
In this example, only the text contained in the paragraph of id p1 of the file demo_text.txt is inserted.

This is some text in a paragraph.
Get External Content



- The optional callback parameter specifies a callback function to run when the `load()` method is completed.
- This function can have different parameters:
  - `responseText` - contains the resulting content if the call succeeds
  - `statusTxt` - contains the status of the call
  - `xhr` - contains the XMLHttpRequest object

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js">
</script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("#div1").load("demo_test.txt", function(responseTxt, statusTxt, xhr){
      if(statusTxt == "success")
        alert("External content loaded successfully!");
      if(statusTxt == "error")
        alert("Error: " + xhr.status + ": " + xhr.statusText);
    });
  });
});
</script>
```



Let jQuery AJAX Change This Text
Get External Content

User clicks the button.

The callback function is invoked at the return of the data

www.w3schools.com dice
External content loaded successfully!
OK

Then the information is loaded into the page

jQuery and AJAX is FUN!
This is some text in a paragraph.
Get External Content

These examples are taken from:
`https://www.w3schools.com/jquery/jquery_ajax_load.asp`

11. Bootstrap and Font Awesome

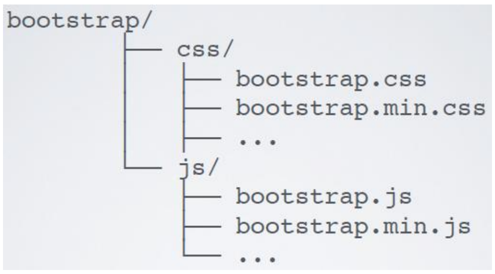
11.1. Bootstrap

Bootstrap is an open source toolkit for developing websites, which includes: HTML, CSS, Javascript (JS); Font styles; Icons; …

Bootstrap History

- Bootstrap is an open source product from Mark Otto and Jacob Thornton (employees at Twitter).
- There was a need to standardize the frontend toolsets of engineers across the company.
- "In the earlier days of Twitter, engineers used almost any library they were familiar with to meet front-end requirements. Inconsistencies among the individual applications made it difficult to scale and maintain them. Bootstrap began as an answer to these challenges and quickly accelerated during Twitter's first Hackweek. By the end of Hackweek, we had reached a stable version that engineers could use across the company."

Bootstrap File Structure:



```
bootstrap/
    ├── css/
    │   ├── bootstrap.css
    │   ├── bootstrap.min.css
    │   ├── ...
    └── js/
        ├── bootstrap.js
        ├── bootstrap.min.js
        └── ...
```

Minified Version of a File

- Minification is the process of removing all unnecessary characters from source code without changing its functionality. These unnecessary characters usually include: white space characters; new line characters; comments.
- Minified source code is especially useful for interpreted languages deployed and transmitted on the Internet (such as JavaScript), because it reduces the amount of data that needs to be transferred.
- Minification can be distinguished from data compression in that the minified source can be interpreted immediately without the need for an uncompression step.

Responsive Web Design

- The term responsive web design was coined by Ethan Marcotte.
- Responsive web design comprises three techniques: flexible grid layout; flexible images and media; media queries.
- Viewport: viewable area of the device.
- Screen size: physical display area of a device.

Containers are the most basic layout element in Bootstrap and are required when using the default grid system.

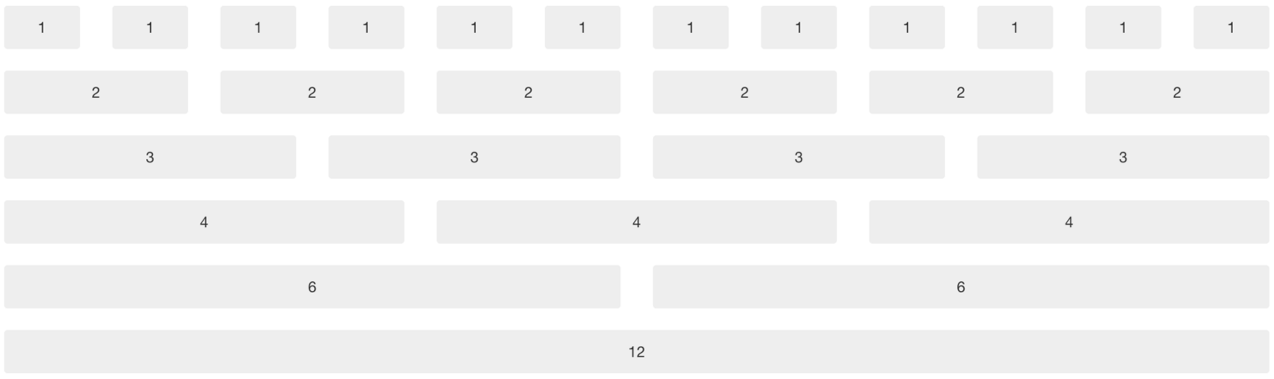
- `<div class="container">...</div>` add a fixedwidth, centered layout meaning its `max-width` changes at each **breakpoint**.

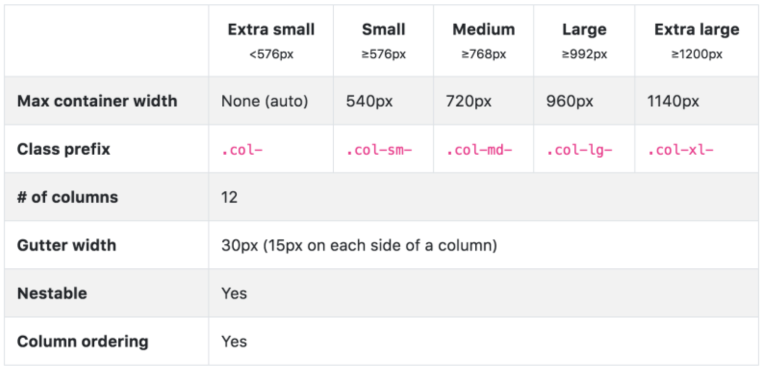
- `<div class="container-fluid">...</ div>` to use a fluid layout (100% wide all the time).

The Grid System: Grids set consistent proportions and spaces between items which helps to create a professional looking design. A grid helps in:

- Creating a continuity between different pages which may use different designs;
- Helps users predict where to find information on various pages;
- Makes it easier to add new content to the site in a consistent way;
- Helps people collaborate on the design of a site in a consistent way.

The default Bootstrap grid system divides the page with rows (row) and utilizes **12 columns** (col).



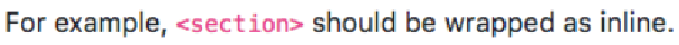


	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
Max container width	None (auto)	540px	720px	960px	1140px
Class prefix	`.col-`	`.col-sm-`	`.col-md-`	`.col-lg-`	`.col-xl-`
# of columns	12				
Gutter width	30px (15px on each side of a column)				
Nestable	Yes				
Column ordering	Yes				

Bootstrap Components: Typography; Code; Table; Figures; Forms; Buttons; Navbar; Dropdown Menu; Icons; …

Code

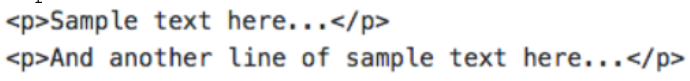
- Wrap inline snippets of code with `<code>`. Be sure to escape HTML angle brackets.
  For example, `<code>&lt;section&gt;</code>` should be wrapped as inline.



For example, `<section>` should be wrapped as inline.

- Use `<pre>`s for multiple lines of code. The `<pre>` tag defines preformatted text. Text in a `<pre>` element is displayed in a fixed-width font (usually Courier), and it preserves both spaces and line breaks.

```
<pre><code>
&lt;p&gt;Sample text here...&lt;/p&gt;
&lt;p&gt;And another line of sample text
here...&lt;/p&gt;
</code></pre>
```



```
<p>Sample text here...</p>
<p>And another line of sample text here...</p>
```

Tables: Add the base class .table to any `<table>` to use Bootstrap table style.

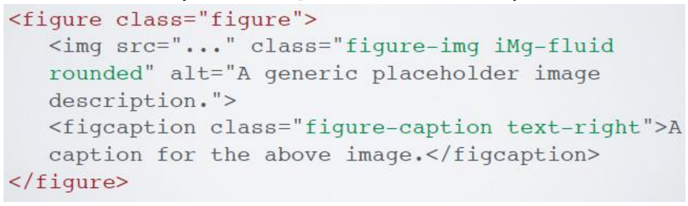
```
<table class="table">
```

Other Table Styles

- Use `.table-striped` to add zebra-striping to any table row within the `<tbody>`.
- Add `.table-bordered` for borders on all sides of the table and cells.
- Add `.table-hover` to enable a hover state on table rows within a `<tbody>`.
- Add `.table-responsive` for horizontally scrolling tables.

Figures

- Use the included `.figure`, `.figure-img` and `.figure-caption` classes to provide some baseline styles for the HTML5 `<figure>` and `<figcaption>` elements.
- Add the `.img-fluid` class to your `<img>` to make it responsive.

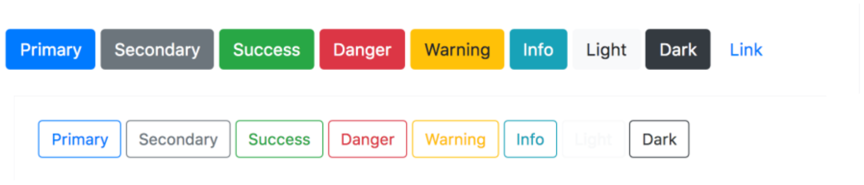


```
<figure class="figure">
   <img src="..." class="figure-img iMg-fluid
   rounded" alt="A generic placeholder image
   description.">
   <figcaption class="figure-caption text-right">A
   caption for the above image.</figcaption>
</figure>
```

Buttons

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

Replace the default modifier classes with the `.btnoutline-*` ones to remove all background images and colors on any button.

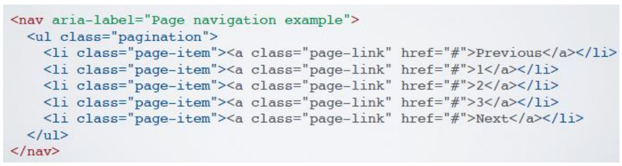


Navs: Navigation in Bootstrap share general markup and styles, from the base `.nav` class. You can add classes to switch between different styles:

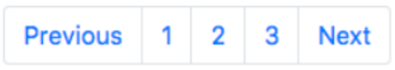
- `.justify-content-center`
- `.flex-column`
- `.nav-tabs`
- `.nav-pills`

```
<nav class="nav">
```

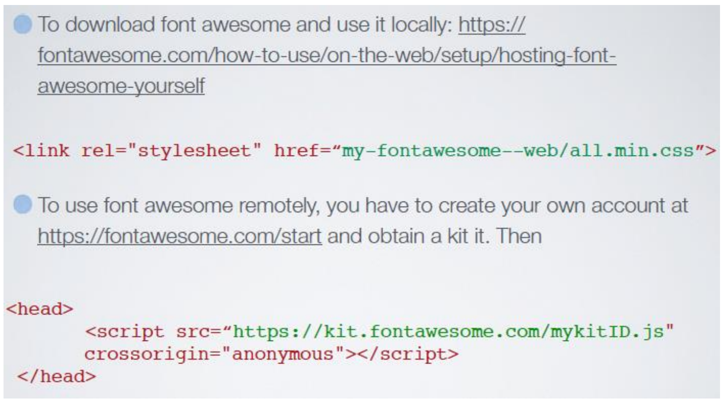
Pagination: is built with `<nav>` and `<list>` HTML elements. Since pages likely have more than one navigation section, it's advisable to provide a descriptive aria-label for the `<nav>` to reflect its purpose.



```
<nav aria-label="Page navigation example">
  <ul class="pagination">
    <li class="page-item"><a class="page-link" href="#">Previous</a></li>
    <li class="page-item"><a class="page-link" href="#">1</a></li>
    <li class="page-item"><a class="page-link" href="#">2</a></li>
    <li class="page-item"><a class="page-link" href="#">3</a></li>
    <li class="page-item"><a class="page-link" href="#">Next</a></li>
  </ul>
</nav>
```



11.2. Font Awesome



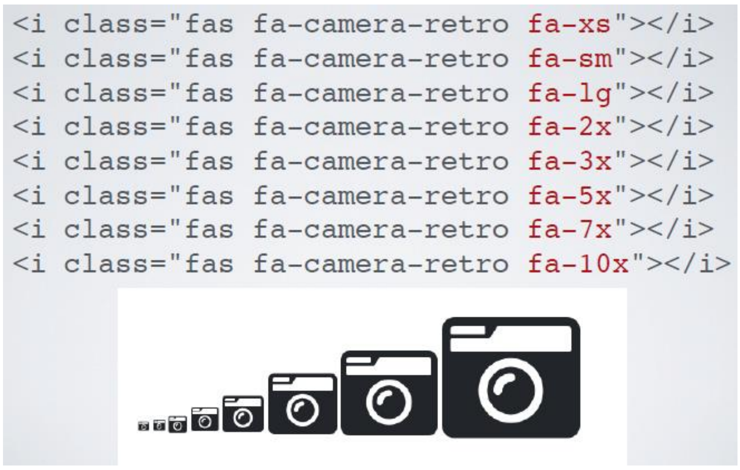
- To download font awesome and use it locally: https://fontawesome.com/how-to-use/on-the-web/setup/hosting-font-awesome-yourself

```
<link rel="stylesheet" href="my-fontawesome--web/all.min.css">
```

- To use font awesome remotely, you have to create your own account at https://fontawesome.com/start and obtain a kit it. Then

```
<head>
       <script src="https://kit.fontawesome.com/mykitID.js"
       crossorigin="anonymous"></script>
 </head>
```

Using Font Awesome: `<i class="fas fa-camera-retro fa-sm"></i>`



```
<i class="fas fa-camera-retro fa-xs"></i>
<i class="fas fa-camera-retro fa-sm"></i>
<i class="fas fa-camera-retro fa-lg"></i>
<i class="fas fa-camera-retro fa-2x"></i>
<i class="fas fa-camera-retro fa-3x"></i>
<i class="fas fa-camera-retro fa-5x"></i>
<i class="fas fa-camera-retro fa-7x"></i>
<i class="fas fa-camera-retro fa-10x"></i>
```

12. Semantic Web

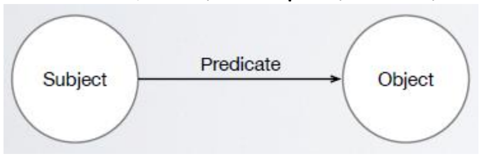
Evolution of the Web

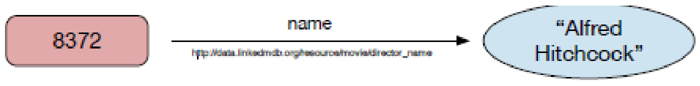
- Initially, the Web was an hypertext made of resources (HTML documents, images, ...) so that human beings can browse, surf and access them
- The current vision of the Web 3.0 encompasses other kinds of resources, namely the data (genoma and proteins, clinical trials, scientific and statistical data, Internet-of-Thing streaming data), which need to be connected by means of typed links so that their structure and semantics is explicit and they can be accessed by both human beings and machines

Web of Documents
Web of Data

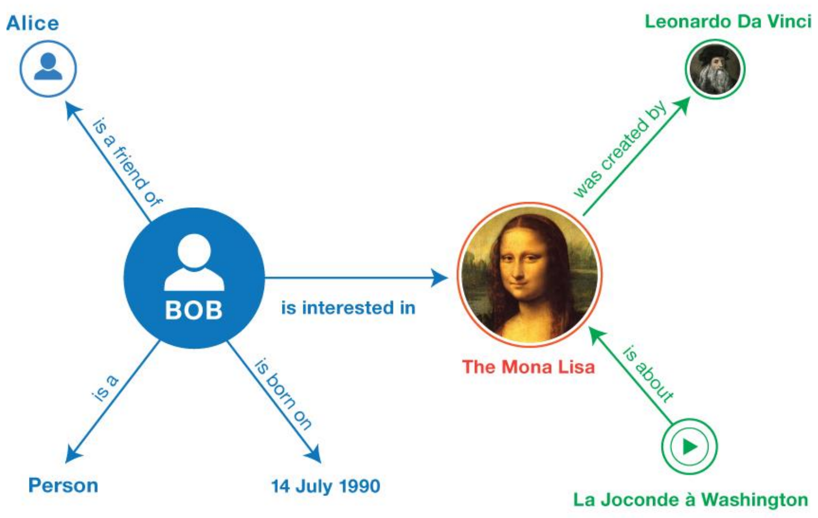
Resource Description Framework (RDF)

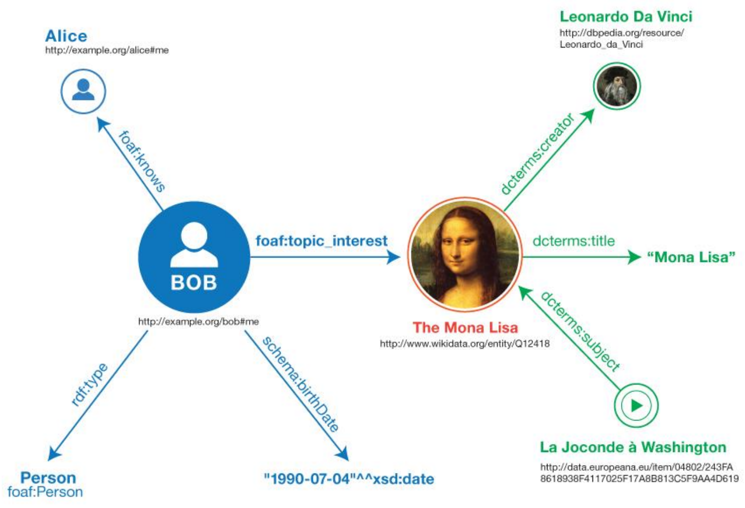
- RDF is a framework for representing information in the Web
- The core structure of the data model is a set of RDF triples, each consisting of a subject (URI), a predicate (URI) and an object (URI or literal). It mimics the basic sentence of the human language (subject - predicate - object).
- Asserting an RDF triple says that some relationship, indicated by the predicate, holds between the resources denoted by the subject and object. This statement corresponding to an RDF triple is known as an RDF statement
- A set of such triples is called an RDF graph. An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link
- An RDF document is a document that encodes an RDF graph in a concrete RDF syntax, such as RDF/XML, N-Triples, Turtle, JSON-LD, RDFa, or TriG





Example of RDF




Alice
http://example.org/alice#me
foaf:knows
BOB
http://example.org/bob#me
foaf:topic_interest
rdf:type
Person
foaf:Person
schema:birthDate
"1990-07-04"^^xsd:date
Leonardo Da Vinci
http://dbpedia.org/resource/Leonardo_da_Vinci
dcterms:creator
The Mona Lisa
http://www.wikidata.org/entity/Q12418
dcterms:title
"Mona Lisa"
dcterms:subject
La Joconde à Washington
http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619

Different data formats

- The basic idea is that an RDF triple is a combination of a subject, a predicate, and an object.
- Subjects, predicates and objects can be annotated with IRIs (an IRI is, fundamentally, an URL that can also contain UNICODE characters).
- Objects can also be annotated with a Literal value. A Literal can be a string, an integer, a double, a date, or another datatype.
- An RDF graph/RDF database is "simply" a set of triples.
- More recently, a fourth field has been added, making it a quadruple. It is the context, and it can be used to group triples in subgraphs, i.e. each triple may have a fourth value declaring the name of the subgraph to which it belongs. If no value is specified, the triple belongs to the default graph.

Literal Datatypes

- Datatypes represent values, such as strings, numbers, and dates. The abstraction used in RDF is compatible with XML Schema (https://www.w3.org/TR/xmlschema11-2/).
- A datatype consists of a lexical space, a space value, and a lexical-tovalue mapping.
- So, for example, the `xsd:boolean` datatype is composed in this way:

Lexical space:
{"`true`", "`false`", "`1`", "`0`"}
Value space:
{***true***, ***false***}
Lexical-to-value mapping
{ <"`true`", ***true***>, <"`false`", ***false***>, <"`1`", ***true***>, <"`0`", ***false***>, }

A language to query RDF: SPARQL

- SPARQL is the most famous language used today to interrogate RDF databases.
- It is based on providing a basic pattern structure P, which is matched to the underlying database (graph homomorphism).
- With SPARQL it is possible to interrogate an RDF database and manipulate it (CRUD operations).
- SPARQL is a set of specifications that provide languages and protocols to query and manipulate RDF graph content on the Web or in an RDF store.

Characteristics of RDF and SPARQL

- To avoid to write each time very long IRIs, SPARQL, such as XMLS, uses prefixes, corresponding to namespaces. Prefixes identify namespaces. For example:
- Here dc: is the prefix of the URL http://purl.org/dc/elements/1.1/. Thus, the string dc:title is to be read as http://purl.org/dc/elements/1.1/title.
- The second prefix, :, is the base prefix. It is intended for the URL that is used more frequently in the database
- The prefixes are defined at the beginning of an RDF file or a query.

SPARQL query forms

- SQL has only one query form: it takes a relational database as an input, and returns a relational database (a single table) as output. In certain cases (such as aggregation queries) this output relation may be composed by only one value, but it is still a relation.
- SPARQL, on the other hand, has 4 different query forms. It takes in input one RDF database, and it can return 3 different types of results, depending on its form.
- This results are: a relation (SELECT), a subgraph (CONSTRUCT), a boolean value (ASK), a subgraph (DESCRIBE).

…