TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

HTTP and Java Web Application Programming

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

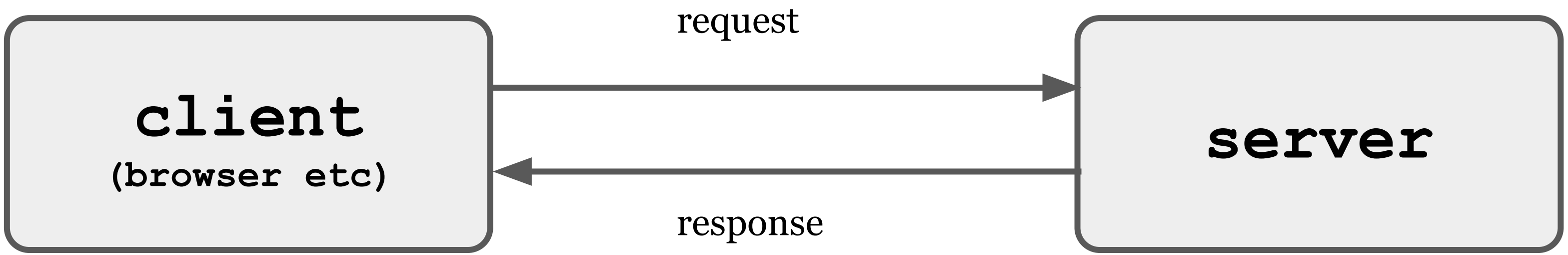
HTTP Basics

The **Hypertext Transfer Protocol (HTTP)** is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which **can be used for many tasks beyond its use for hypertext**.

https://tools.ietf.org/html/rfc2616

Disclaimer: we only discuss some very basic features of HTTP1.1 as needed for this course, full coverage is provided in Network Engineering courses.

HTTP Request and Response



- a client makes a **request**
- a server returns a **response**
- consequences (limitations!):
    - HTTP is **stateless**, i.e. no association between multiple request/response pairs
    - information is **pulled**, not pushed

Uniform resource Locators (URLs) and Identifiers (URI)

- requests refer to a resource that is being identified by URLs
- basic structure: protocol://hostname:port/path
- example: https://www.victoria.ac.nz/about (port is optional, for http it defaults to 80)
- standard: https://tools.ietf.org/html/rfc1738 (URL) , https://tools.ietf.org/html/rfc3986 (URI)
- URLs are unique: domains are administered by ICANN, organisations are responsible for the path part
- uniform resource identifiers (URI) are generalisations of URLs, URIs are unique resource names, URLs also provide information to access the resource

HTTP Request Example

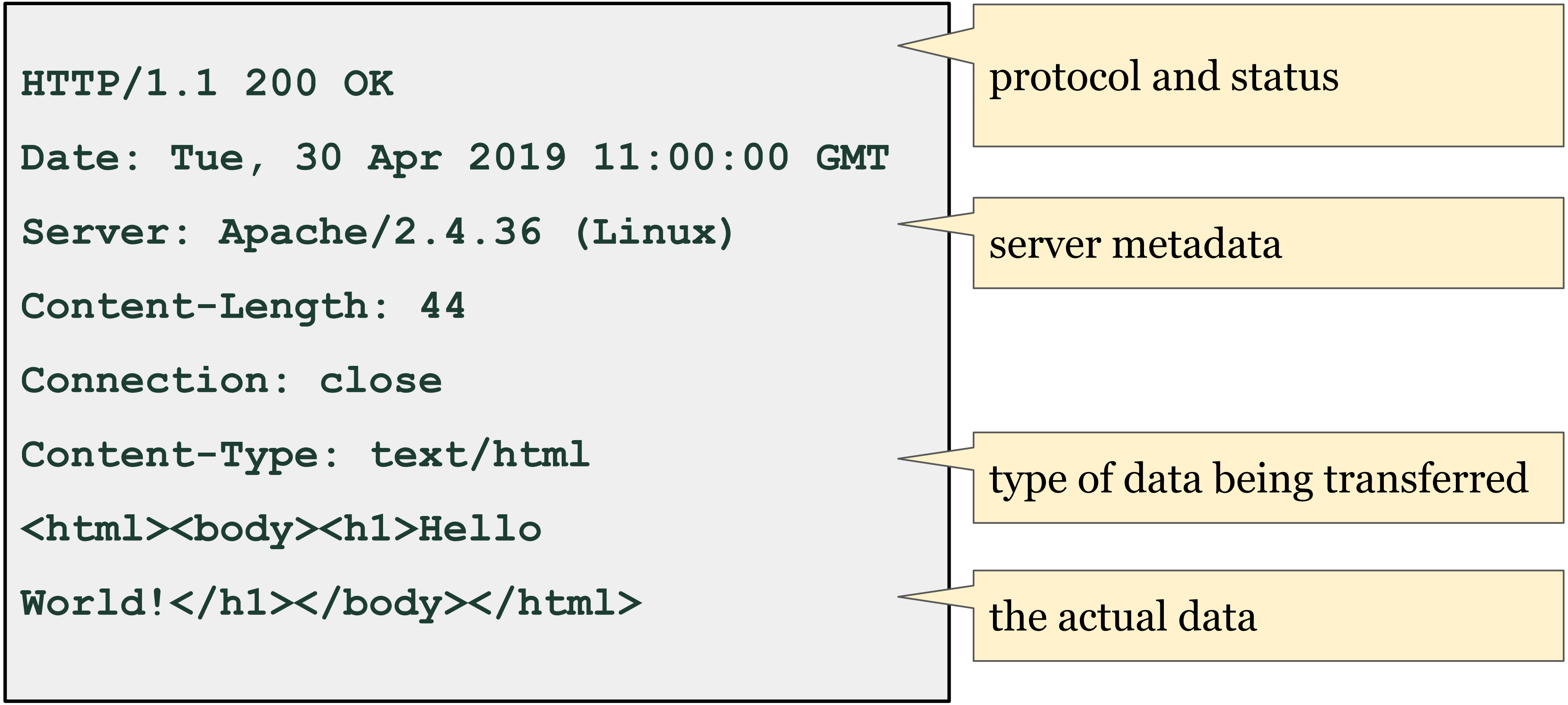
```
GET /about HTTP/1.1
Host: https://www.victoria.ac.nz
Accept: image/gif, image/jpeg
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
\n
```

get resource about using protocol HTTP/1.1

from server **https://www.victoria.ac.nz**

request headers: meta data that can be used by the server to optimise request handling: e.g. client capabilities

HTTP Response Example



```
HTTP/1.1 200 OK
Date: Tue, 30 Apr 2019 11:00:00 GMT
Server: Apache/2.4.36 (Linux)
Content-Length: 44
Connection: close
Content-Type: text/html
<html><body><h1>Hello
World!</h1></body></html>
```

protocol and status

server metadata

type of data being transferred

the actual data

HTTP Methods

- methods (aka verbes) indicate the action to be performed on the resource. The most widely used four methods loosely correspond to CRUD database actions:
- **GET** -- fetch (a representation) of a resource -- widely used by browsers
- **POST** -- send data to the server to create or update a resource, the server will create a new URI -- used by browsers to submit forms
- **PATCH** -- update a resource
- **PUT** -- send data to the server to be accessible under a URI supplied by client, replace/update resource for this URI if it already exists
- **DELETE** - request a resource to be deleted

HTTP Response Codes

Server responses have a response code and additional message. Response codes are 3-digit numbers grouped as follows:

- informational 1XX
- success: 2XX   (example: “200 OK”)
- redirection 3XX (example: “301 Moved Permanently”)
- client error 4XX (examples: “404 Not Found” and “405 Method Not Allowed”)
- server error 5XX (example: “500 Internal Server Error”)
- full list: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

418 I'm a teapot

Easter egg in HTTP

The HTTP 418 I'm a teapot client error response code indicates that the server refuses to brew coffee because it is, permanently, a teapot. A combined coffee/tea pot that is temporarily out of coffee should instead return 503. This error is a reference to Hyper Text Coffee Pot Control Protocol defined in April Fools' jokes in 1998 and 2014.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418

What the server is telling the client ..

- informational 1* -- **hold on**
- success: 2*   -- **here you go**
- redirection 3* -- **go away**
- client error 4* -- **you messed up**
- server error 5* -- **I messed up**

Content Types

- the server describes the type of content with a **`content-type`** header
- **`Content-Type: text/html`**
- this is an instruction to the client what to do with it (render as html, image, pdf, ..)
- the client can tell the server what kind of content it can handle with the **`Accept`** header

Content Types (ctd)

- list of standard content types: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types
- popular:
  - **`text/html, text/javascript, text/plain`**
  - **`image/gif, image/jpg, image/png`**
  - **`application/json, application/xml, application/pdf`**
  - **`audio/mpeg, video/mpeg`**

HTTP Clients

- the obvious ones: web browsers
- CLI clients -- try: **`curl http://google.com`**
- standalone apps for web development, like PostMan
- client libraries in various PLs that can be embedded into programs, used in applications, other libraries and mobile apps
- examples -- Java:
    - Apache HTTP client library for Java
    - the HTTP client in Java 11
    - OkHttp
    - **`java.net.HttpURLConnection`** in older Java versions

Example: HTTP Client in Java (using Apache Client)

```
// 1. build Google query URL
URIBuilder builder = new URIBuilder();

builder.setScheme("http").setHost("www.google.com").setPath("/search")
       .setParameter("q", "the answer to life the universe and everything");
URI uri = builder.build();

// 2. create and execute the request
HttpGet request = new HttpGet(uri);
HttpClient httpClient = HttpClientBuilder.create().build();
HttpResponse response = httpClient.execute(request);

// 3. do something with the response
String content = EntityUtils.toString(response.getEntity());
assertTrue(content.indexOf(" 42") > -1);
```

https://github.com/jensdietrich/se-teaching/tree/main/httpclient

Server Programming

- an http server creates responses for incoming HTTP requests
- support for http servers is part of the **J**ava **E**nterprise **E**dition (JEE), more specifically, the web profile (part of JEE)
- the core are **servlets**, a simple API to create web applications
- (parts of) JEE is implemented by various servers, including several excellent open source servers:
  - Apache Tomcat -- http://tomcat.apache.org/
  - Jetty -- https://www.eclipse.org/jetty/

Accessing and Running the Server Examples

- check out https://github.com/jensdietrich/se-teaching/tree/main/servlets
- the project uses the Maven jetty plugin for easy deployment
- build project and start server with **mvn jetty:run**
- point browser to http://localhost:8080/webapp/

Core Servlet Packages Overview

package	description
`javax.servlet`	core servlet interfaces / classes such as request and response
`javax.servlet.http`	core servlet interfaces / classes such as request and response with additional methods to add HTTP support
`javax.servlet.jsp`	Java Server Pages, a technology build on top of servlets

The Basic Structure of a Servlet

- a servlet must implement a **service method** that has two parameters representing the request and the response
- the service method depends on the HTTP method used
- example: **`doGet`**, **`doPost`** , ..

The Basic Structure of a Servlet (ctd)

servlet code usually has the following simple structure:

1. set the **content type** and other headers
2. acquire a **stream** (= writer) from the response to write object
3. use this writer to write the response
4. **close** the stream

advanced:

- set custom HTTP status code
- **`HttpServletResponse.setStatus(int sc)`**
- **`HttpServletResponse`** also defined constants for status codes, like **`HttpServletResponse.SC_OK`**

Example: ServerDateServlet

```
public class ServerDateServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp) {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>Shows the Current Server Time</h1>");
        out.println(new java.util.Date());
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

dynamic part: will change when page is reloaded !

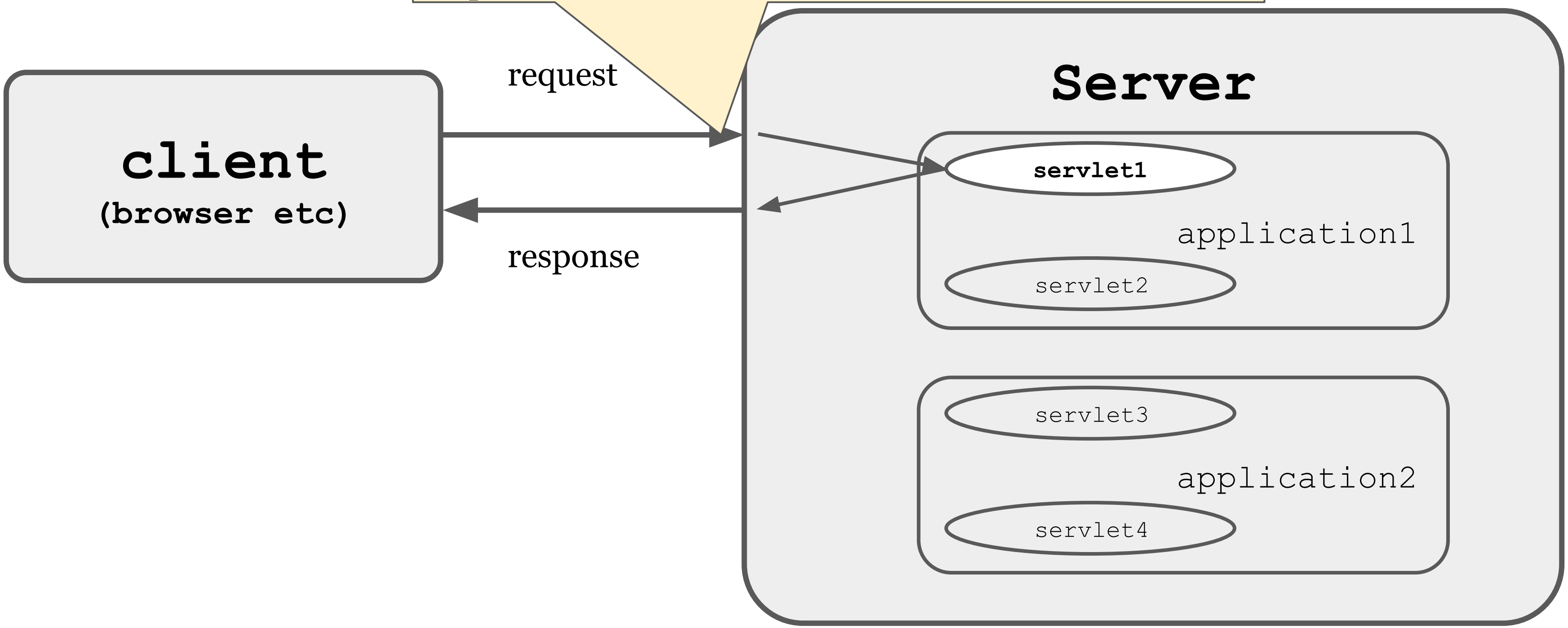
https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/ServerDateServlet.java

Routing – Mapping URLs to Code

- to access the servlet, it needs to be mapped to a URL
- this can be done using on of the following approaches:
- configure the servlet in `WEB-INF/web.xml`
- use web annotations
- the server then uses the info to **route** requests to the servlet
- routing: mapping URLs and method to code
- *the examples use web.xml -- annotations are in code but commented out as they were not consistently supported by the Maven jetty plugin at the time of writing this*

Mapping Code to URLs (ctd)

based on the URL of an incoming request, the server selects a webapplication and a servlet to process the request



Mapping URLs to Servlets: web.xml

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

    <servlet>
        <servlet-name>ServerDate</servlet-name>
        <servlet-class>
            nz.ac.vuw.jenz.servlets.ServerDateServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServerDate</servlet-name>
        <url-pattern>/ServerDate</url-pattern>
    </servlet-mapping>

</web-app>
```

Alternative Mapping of URLs to Servlets: Web Annotations

```
import javax.servlet.annotation.WebServlet;
@WebServlet(name = "ServerDate", urlPatterns = "/ServerDate")
public class ServerDateServlet extends HttpServlet {..}
```

Building Deployable Web Applications

- example: can be run with Maven jetty plugin directly
- general: build a web application archive (war file)
- war files are zip files containing compiled classes, libraries, resources (static web pages, images) and configuration files (web.xml) in a canonical structure
- war files are build with Maven using **`mvn package`**
- war files can be deployed on any JEE-compliant server in different ways, including:
  - upload war from a management console
  - simply copy files into server **`webapps`** folder
  - Example: build war (mvn package), copy war into Tomcat **`webapps`**

Servlet Miscellaneous

- servlets are usually stateless, i.e. they do not have instance variables
- a server may only instantiate a servlet class once, this sole instance then serves all requests for the respective URL
- incoming requests are executed in separate threads (usually reusing threads with a thread pool), so if servlets had state, concurrency is an issue
- the servlet API has methods to customise the lifecycle of objects (init / destroy)

Analysing Request Parameters

- the **`HttpResponse`** type has methods to access request parameters, and query the names of parameters available
- **`String value = request.getParameter(“foo”);`**
- **`Enumeration params = request.getParameterNames();`**
- values are always strings, may need to be converted
- use case: form processing, works for GET and POST (forms encoded in the URL or in the request body)
- example: https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/FormAnalyzerServlet.java

Analysing Headers

- the **`HttpResponse`** type has methods to access header values, and query the names of headers available
- **`String value = request.getHeader(header);`**
- **`Enumeration headers = request.getHeaderNames();`**
- values are always strings, may need to be converted
- example: https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/HeaderAnalyzerServlet.java

Creating Non-HTML Content: Text-Based

- for text-based formats (JSON, XML, ..) the approach is similar
- the correct content type should be set
- options:
  - directly output data using writer **`out.println(..)`** -- fast but error-prone
  - use a (JSON, XML, ..) library to build document in memory, then flush to **`out`** -- some overhead, but uses library verification to ensure format is correct
  - use a template, bind and output (see JSP section later)

Creating Non-HTML Content: Binary

- must use binary stream instead of writer
- building resource in memory is recommended
- use case: create visualisation of real-time data on the fly
- idea: draw image in memory using Java 2D drawing API, the flush to (binary) stream (**`out`**)
- example: https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/ImageServlet.java

Adding State

- use case: shopping carts in eCommerce applications
- servlet API:
    - get a session from the request: HttpSession session = request.getSession();
    - the session acts like a map Map<String,Object>
    - add information: session.setAttribute(“shoppingcart”, new List<Product>());
    - retrieve information: session.getAttribute(“shoppingcart”);
- subsequent requests will see same session
- sessions are managed by the server, relies on session ids sent to and returned by the client

Adding State (ctd)

- by default cookies are used
- URL rewriting is supported as well in case cookies are not supported, this requires the manipulation of URLs returned to the server
- the server also manages session lifecycle:
    - sessions can be explicitly destroyed (e.g., when client logs out)
    - otherwise the server will timeout sessions (timeout can be set
    - lifecycle listeners can be added (via web.xml) to customise this process further, e.g. to close database connections
    - this avoids memory leaks and contributes to security !
- examples:
  https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/ShoppingCartServlet.java (requires cookies)
  https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/ShoppingCartServlet2.java (does not rely on cookies)

Application-Level Modularity

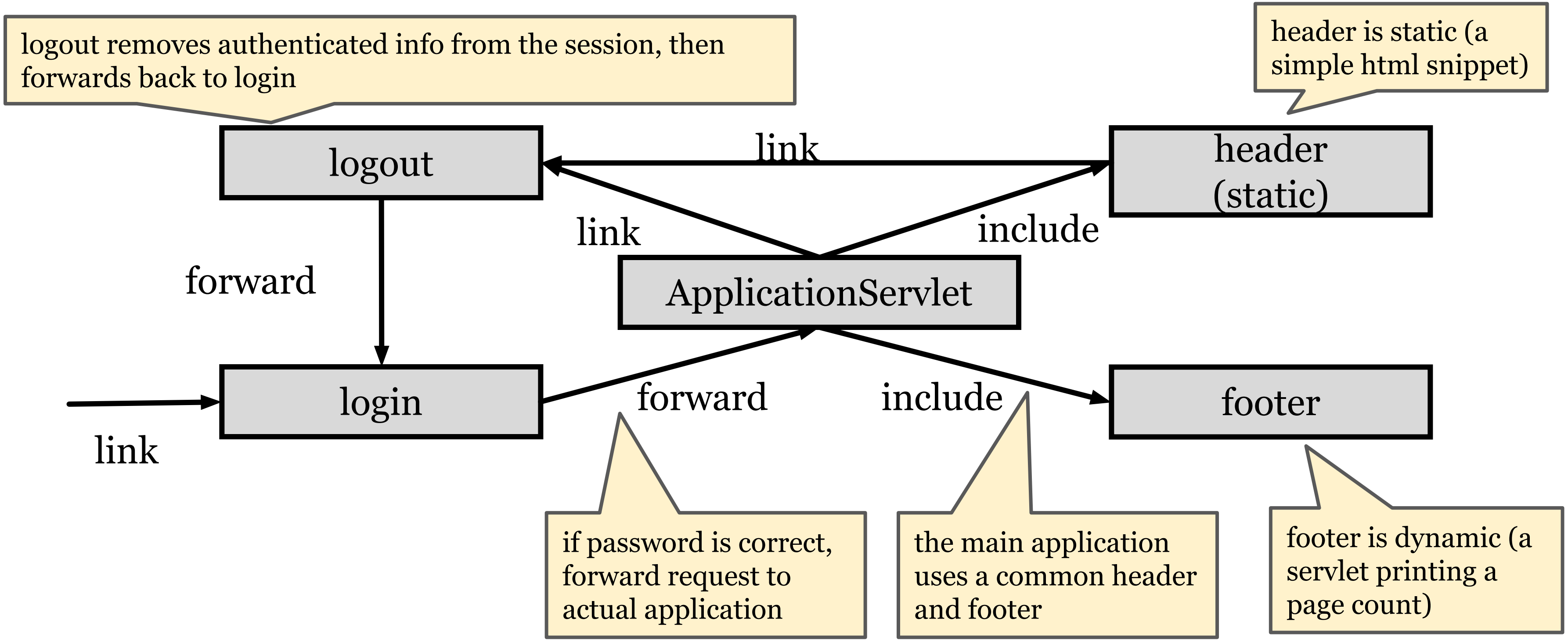
- idea: divide applications into parts that communicate with HTTP
- can also be used to communicate with resources managed by other servers
- servlets can access request dispatcher to do this:
- **`getServletContext().getRequestDispatcher("<url>")`**

Application-Level Modularity (ctd)

- request dispatch can either delegate the entire request handling to another resource (`forward(..)`)
- use case: split workload based on request parameters, forward only if state indicates that user has logged in
- or use another resource to handle the request partially (i.e., produce parts of the response) (`include(..)`)
- use case: different pages use same resources to create header and footer of a web page
- example: https://github.com/jensdietrich/se-teaching/tree/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/modular

deprecated

Example: Application-Level Modularity (ctd)



logout removes authenticated info from the session, then forwards back to login

header is static (a simple html snippet)

logout

header (static)

link

link

include

forward

ApplicationServlet

login

forward

include

footer

link

if password is correct, forward request to actual application

the main application uses a common header and footer

footer is dynamic (a servlet printing a page count)

deprecated

Server-Side MVC

- web applications quickly become complex, and it is desirable to use modular designs
- often, it is effective to explicitly create a **model** domain object that the web applications displays or modifies (example: collection used as shopping card in session example(s))
- then there are dedicated servlets to analyse requests, and interact with this model -- so-called **controllers**
- once this is done, controllers forward requests to dedicated **views** (servlets, or often JSPs, see below) to generate the new display pages
- this is the idea behind server-side **MVC** (model-view-controller) frameworks like Struts, Spring MVC etc

Reflection on Architecture: Container vs Application

- servlets are running in applications managed by the server (aka **container**)
- web applications comply to **contracts** to interact with the server:
  - classes extend / implement a certain superclass
  - configuration files use a certain format
  - web applications use a defined file and folder structure
- in many cases, this is transparent (happens in the background, without the application being aware of it)

Container-Provided Services

- **naming**: locate methods by name (URL)
- **multithreading**: servlet code is executed in (multiple) threads
- **clustering**: the ability to run on a cluster of computers to boost performance
- **lifecycle**: the container culls inactive objects (like sessions)
- **security**: users can define authentication constraints for certain URLs using server config files or management consoles (Tomcat: **`config/tomcat-users.xml`**)

Container-Provided Services (ctd)

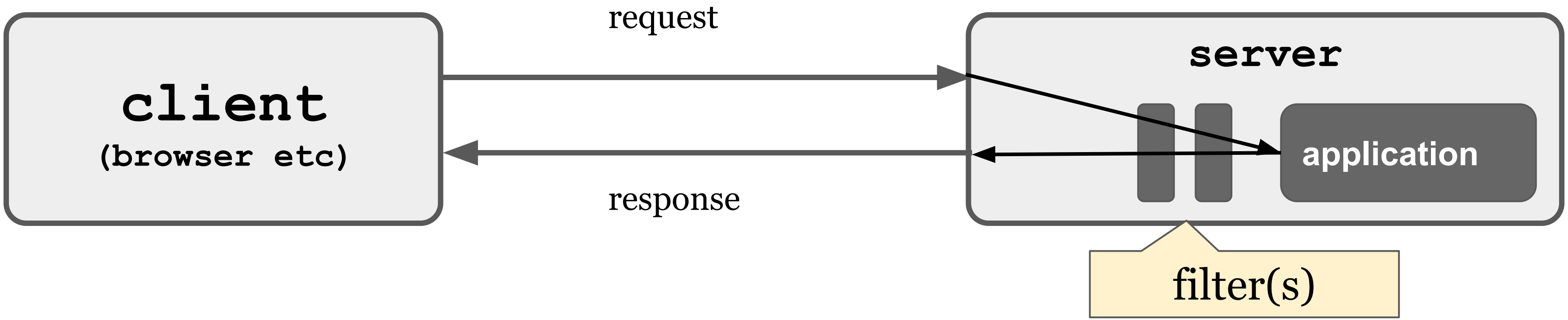
- **encryption**: servers support https, apply this automatically based on protocol used
- **compression**: based on headers (= client capabilities), servers may transparently compress content
- **monitoring**: monitor traffic, creates stats
- **logging** (to be explained)

Logging via a Container-Provided Service

- using **`System.out.println(..)`** makes little sense for a server-side application:
- it is not clear what the console is, and there might be multiple applications running on the same server
- use a standard logging framework is possible
- Servers offer a logging service that can be invoked as follows (from a servlet):
  `getServletContext().log(message)` and
  `getServletContext().log(message,exception)`

Adding Custom Services via Filters

- filters can be provided to intercept incoming requests and outgoing responses
- the offer transparent (non-invasive) pre- and postprocessing, similar to aspect-oriented programming (AOP)
- filters can be chained
- filters are deployed by URL (pattern) using **`web.xml`** or annotations



A Profiling Filter

purpose: measure time it takes to process a request, and log it

```
public void doFilter(ServletRequest request, ServletResponse response,
     FilterChain chain) {

     // BEFORE
     long before = System.currentTimeMillis();

     // ACTUAL INVOKE
     chain.doFilter(request, response);

     // AFTER
     long after = System.currentTimeMillis();
     String msg = String.format("Time to process was %d ms",after-before);
     filterConfig.getServletContext().log(msg);
}
```

https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/java/nz/ac/vuw/jenz/servlets/ProfilingFilter.java

Deploying the Profiling Filter (web.xml)

```
<filter>
    <filter-name>ProfilingFilter</filter-name>
    <filter-class>nz.ac.vuw.jenz.servlets.ProfilingFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ProfilingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

filter is applied to requests for all URLs within this application

deprecated

Java Server Pages

- document-centred way to write servlets
- based on templating
- focus is on writing a document (HTML or other), with embedded scripting sections that are evaluated at runtime
- similar to PHP and ASP technologies

deprecated

Processing JSPs (by the Container)

- JSPs are compiled by a page compiler (like Jasper) into Servlet source code
- the generated servlet is then compiled (with javac)
- the compiled class is loaded, and the server associated this with a URL derived from the name of the page
- the URL can be customised in **`web.xml`**

deprecated

JSP Example

directive for the JSP compiler: this is a JSP page

```
<%@page contentType="text/html"%>
<html>
<head><title>Server Date - JSP Version</title></head>
<body>
<h1>Server Date - JSP Version</h1>

The current server date is <%= new java.util.Date() %>
</body>
</html>
```

dynamic section, this will be replaced at request time by the result of evaluating this Java code

https://github.com/jensdietrich/se-teaching/blob/main/servlets/src/main/webapp/ServerDate.jsp

deprecated

Generated Servlet

```
public final class ServerDate_jsp extends org.apache.jasper.runtime.HttpJspBase .. {
public void _jspService(final javax.servlet.http.HttpServletRequest request, final
javax.servlet.http.HttpServletResponse response)
     ..
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                    null, true, 8192, true);
      out = pageContext.getOut();

      out.write("<html>\n");
      ..
      out.write("The current server date is ");
      out.print( new java.util.Date() );
```

the jetty Maven plugin generated servlets in **`target/tmp`** when the JSP is first invoked

deprecated

More JSP Features

- scriptlet sections
- predefined variables code snippets can reference: out, session, request, application, ..
- a builtint expression language
- declarative exception handling
- special support for Java Beans
- custom tags: build HTML-like tags with custom semantics
- …

some more examples:
https://github.com/jensdietrich/se-teaching/tree/main/servlets/src/main/webapp

Maven (JEE) Web Projects

- example: https://github.com/jensdietrich/se-teaching/tree/main/servlets
- **`pom.xml`** :
  - **`<packaging>war</packaging>`** – set maven output format
  - needs dependency to:

    `<groupId>javax.servlet</groupId>`

    `<artifactId>javax.servlet-api</artifactId>`

    (this dependency has the servlet API and is not part of the standard Java class library)
  - nice to have: jetty plugin to conveniently start / stop server (see servlet example for details)
- static web sites (html, jpg, css, ..) and jsps go into **`src/main/webapp/`**
- **`web.xml`** configuration files goes here: **`src/main/webapp/WEB-INF/web.xml`**
- an archetype (template) can also be used to create projects, see https://maven.apache.org/archetypes/maven-archetype-webapp/