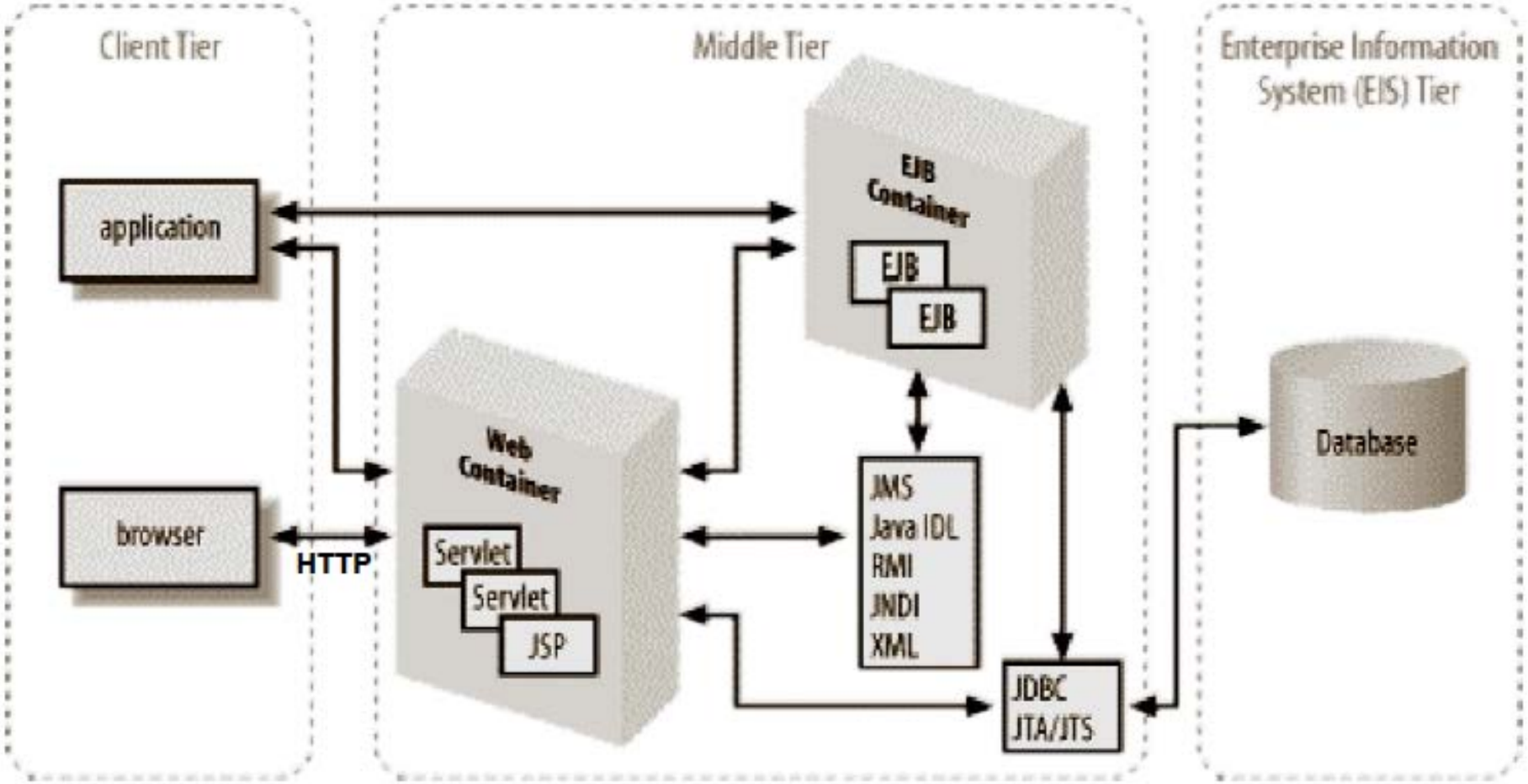
CT5106 SOFTWARE ENGINEERING II

2 SERVLETS

Java Enterprise Edition Architecture



Typical process in a Servlet

- Regardless of the application, servlets usually carry out the following routine:

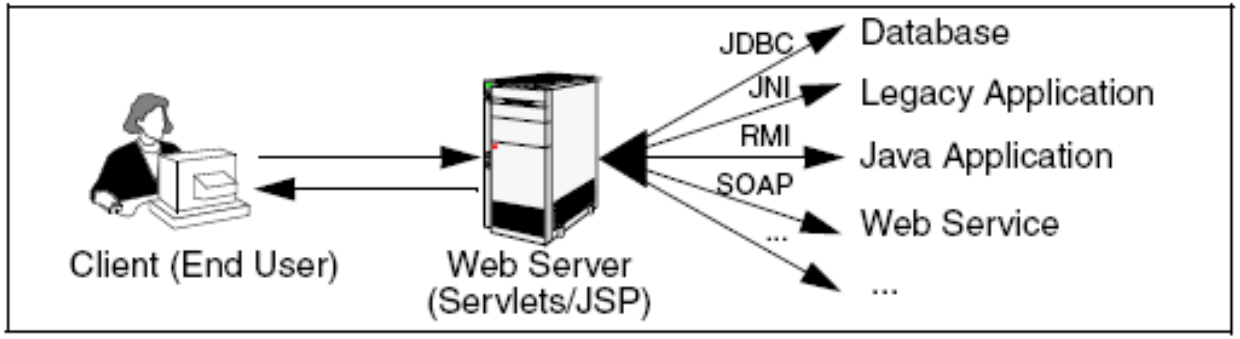
1) Read any data sent by the user
   - Capture data submitted by an HTML form.
2) Look up any HTTP information
   - Determine the browser version, host name of client, cookies, etc.
3) Process the submitted data & Generate the Results
   - Connect to databases, connect to legacy applications, etc.
   - E.g. LoginServlet might get username and password from a form, check the data against the username/password in the database, and return result or forward the user to the next page

Life of a Servlet (cont.)

4) Format the Results
   - Generate HTML on the fly
5) Set the Appropriate HTTP headers
   - Tell the browser the type of document being returned or set any cookies.
6) Send the document back to the client

What can you build with Servlets?

- Search Engines
- Personalization Systems
- E-Commerce Applications
- Shopping Carts
- Product Catalogs
- Intranet Applications
- Groupware Applications: bulletin boards, file sharing, etc.

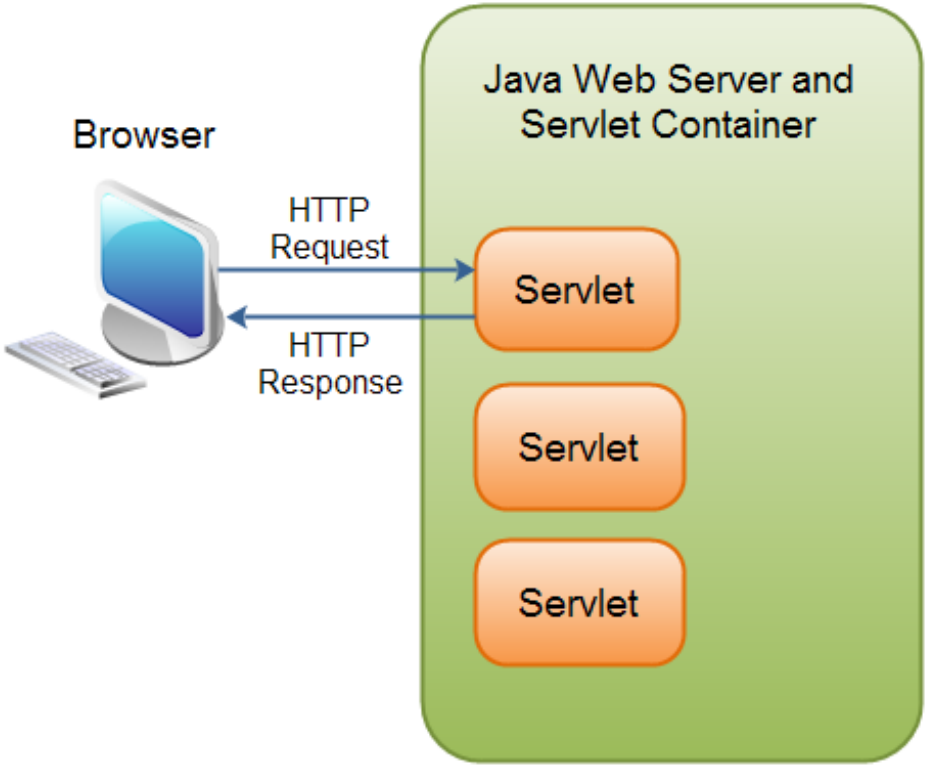


Servlet

- Receive request from client (normally a Get or Post request)
- Read the data sent by the client
- Process data and generate results
- Compose response
- Send response (explicit and implicit) to client

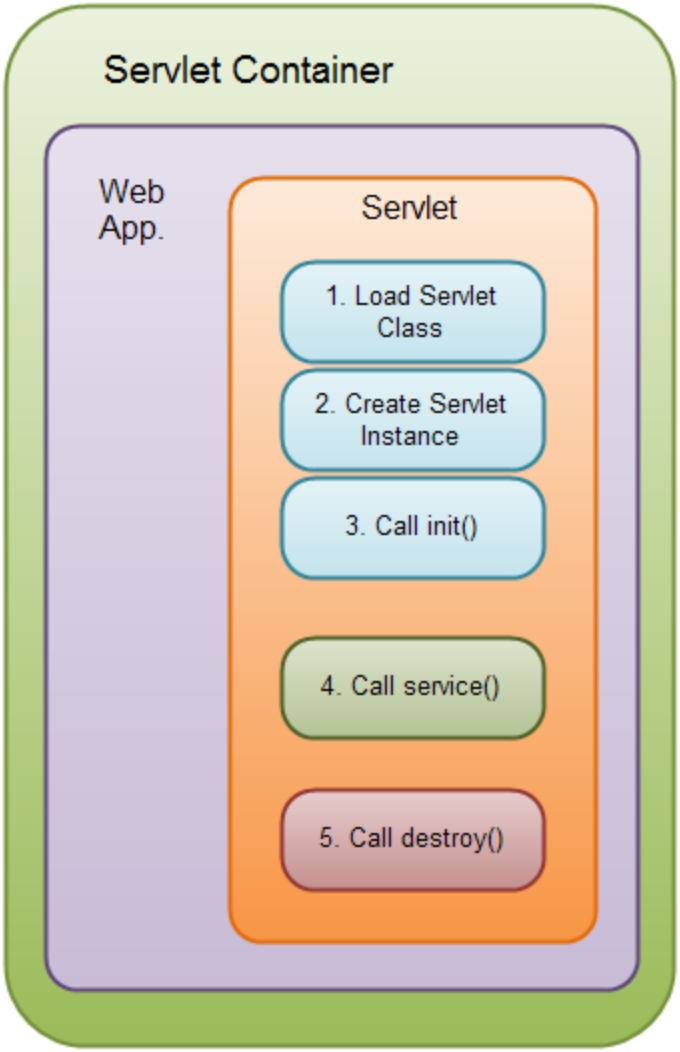
Servlets - Introduction

- A Java Servlet is a Java object that responds to HTTP requests. It runs inside a Servlet container
- The servlet container initialises the servlet, from when it is available for processing requests (GET, POST)
- Can be used to dynamically generate HTML to return to browser
- Simple building block of Java web applications



Servlet lifecycle

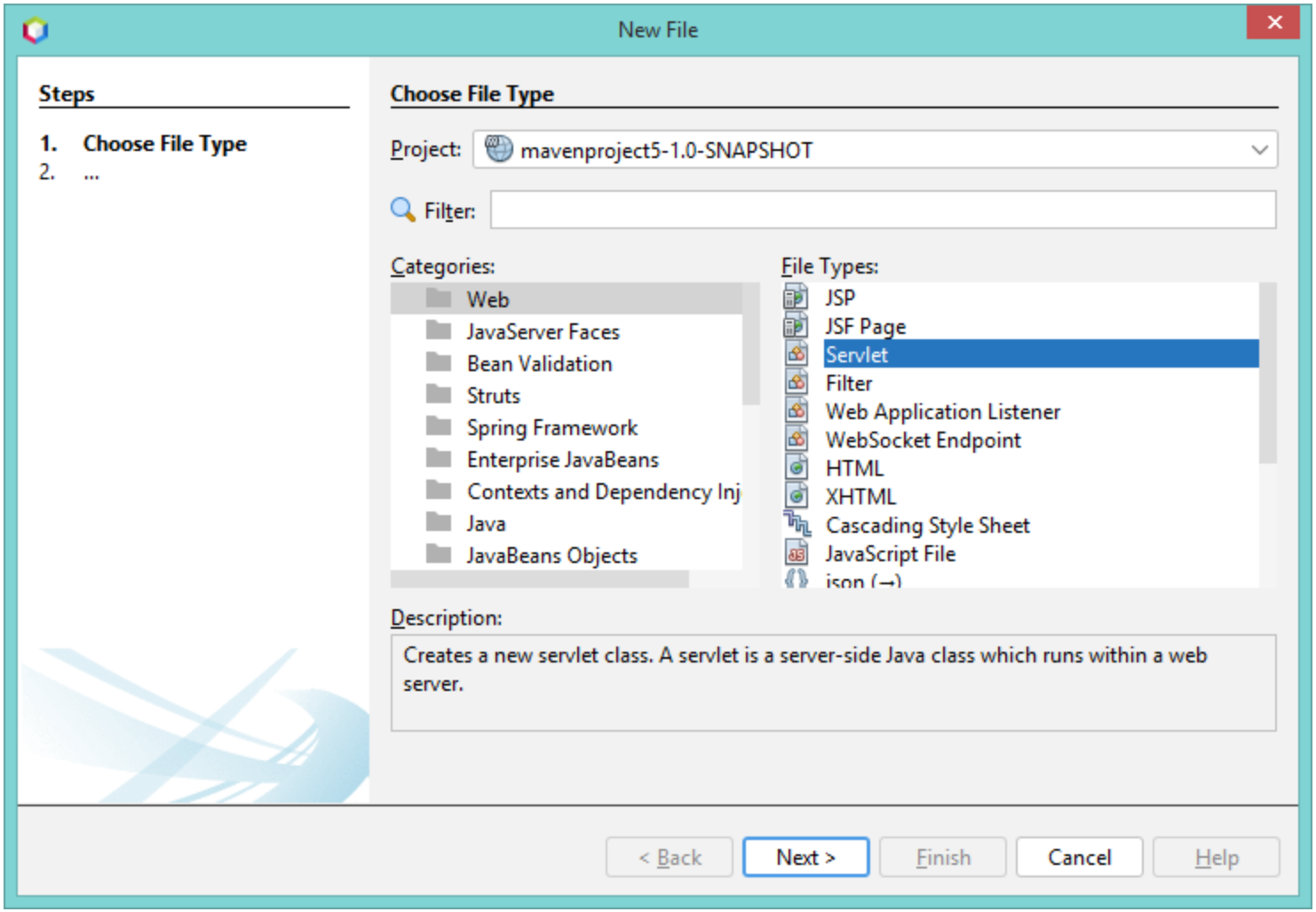
- The servlet life cycle is managed by the servlet container. The steps are:
  1. Load Servlet Class.
  2. Create Instance of Servlet.
  3. Call the servlets init() method.
  4. Call the servlets service() method.
  5. Call the servlets destroy() method.
  - Step 1, 2 and 3 are executed only once, when the servlet is initially loaded.
  - By default the servlet is not loaded until the first request is received for it.
  - Step 4 is executed multiple times - once for every HTTP request to the servlet.
  - Step 5 is executed when the servlet container unloads the servlet.



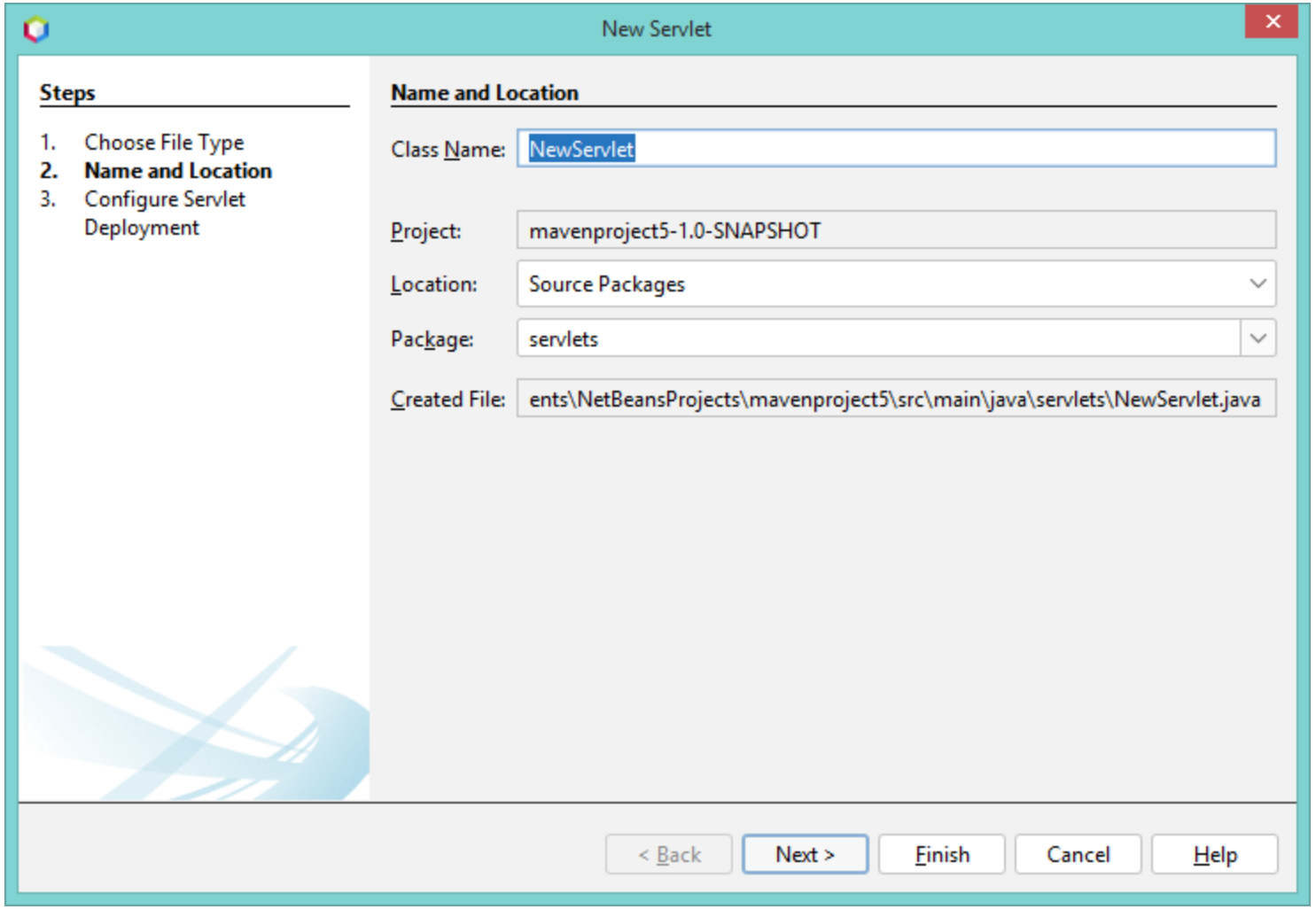
Useful interfaces used in servlets

Interface	Description
HttpSession	Allows state to be stored for a user across one or more HTTP requests
Cookie	Object used to store small amounts of information on the client browser
ServletContext	Provides methods to communicate with the servlet container
Filter	Provides means to intercept and pre-process / post-process requests and responses

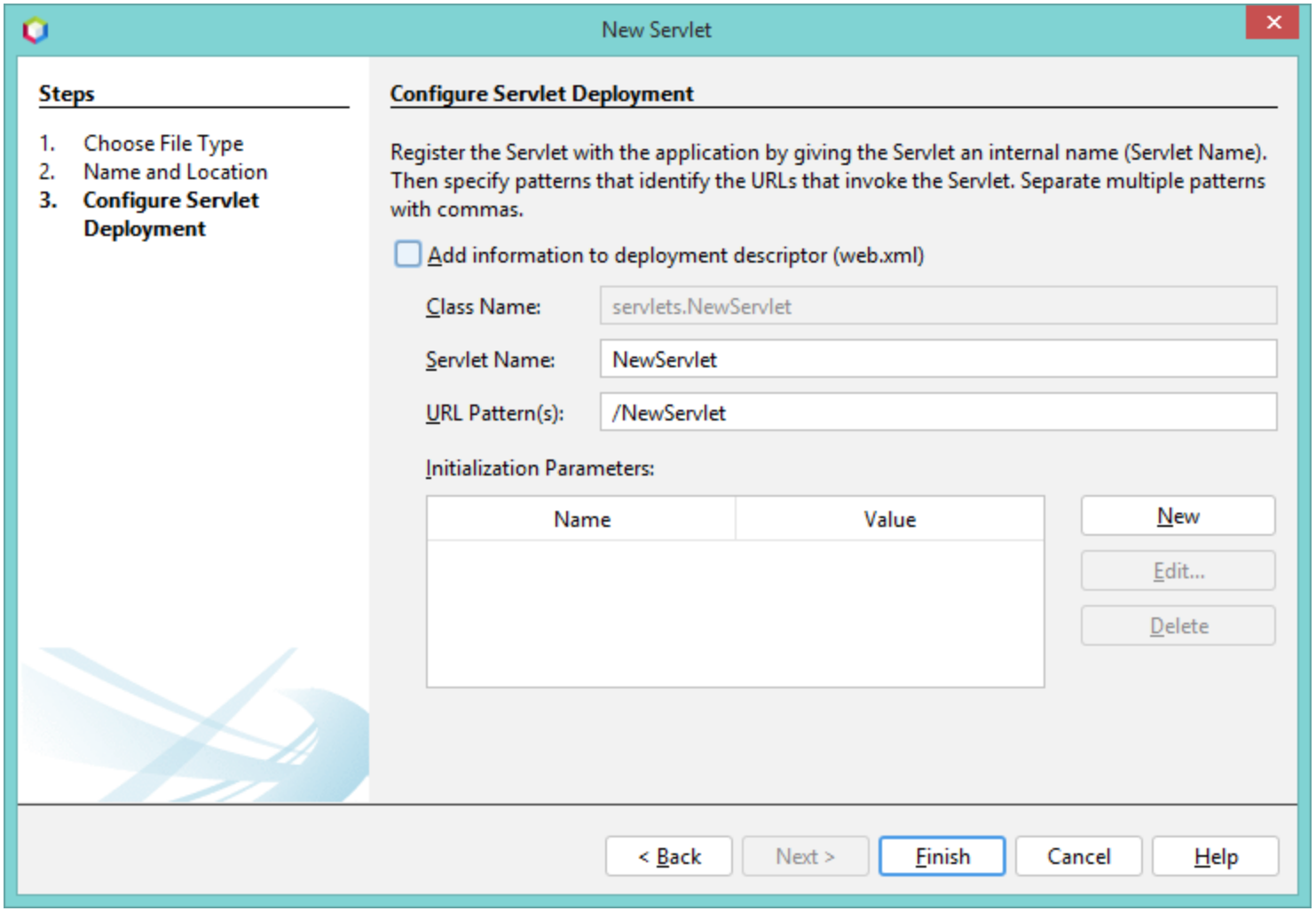
Simple first servlet in NetBeans



Specify name for servlet and package to put it in



Don’t need to use web.xml since JEE7



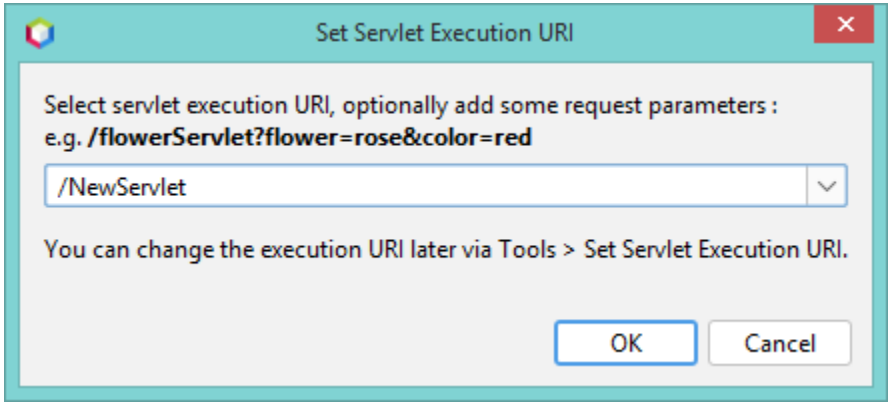
Done!

- You've now created a Java class which implements the servlet methods, and which is mapped to the URL pattern you specified

```
@WebServlet(name = "NewServlet", urlPatterns = {"/NewServlet"})
public class NewServlet extends HttpServlet
{
```

Run servlet – right click and “Run File”

- Just click OK here



Browser displays the response from the servlet

- The servlet creates a default html page which is sent back to the browser
- You should try changing it to print something else

Basics

- Servlets typically extend HttpServlet and override doGet or doPost, depending on whether the data is being sent by GET or by POST.
- If you want a servlet to take the same action for both GET and POST requests, simply have doGet call doPost, or vice versa.
- In NetBeans, this is done for us – both doGet and doPost are redirected to processRequest by default (you can of course change this if you want)
  - Expand doGet and doPost code to see the redirection

Request / Response

- Both doGet and doPost take two arguments: an HttpServletRequest and an HttpServletResponse.
  - The HttpServletRequest lets you get at all of the *incoming data; the class has methods by which you can find out about information* such as form (query) data, HTTP request headers, and the client's hostname.
  - The HttpServletResponse lets you specify *outgoing information such as HTTP status* codes (200, 404, etc.) and response headers (Content-Type, Set-Cookie, etc.).
- Most importantly, HttpServletResponse lets you obtain a PrintWriter that you use to send document content back to the client. For simple servlets, most of the effort is spent in println statements that generate the desired page.

The HttpRequest object

- Provides methods to access different parts of the request, e.g.
  - Request URI
  - Parameters
    - Passed from client to server
  - Attributes
    - Can be added to request by server for passing on to next object that processes this request
  - Session
  - Plus headers, request body via getInputStream() , info on remote host etc.

Servlet That Generates HTML

- Most servlets generate HTML. To generate HTML, you add three steps to the process just shown:
  1. Tell the browser that you're sending it HTML.
  2. Modify the println statements to build a legal Web page.
- You accomplish the first step by setting the HTTP Content-Type response header to text/html.
- The way to designate HTML is with a type of text/html, so the code would look like this:

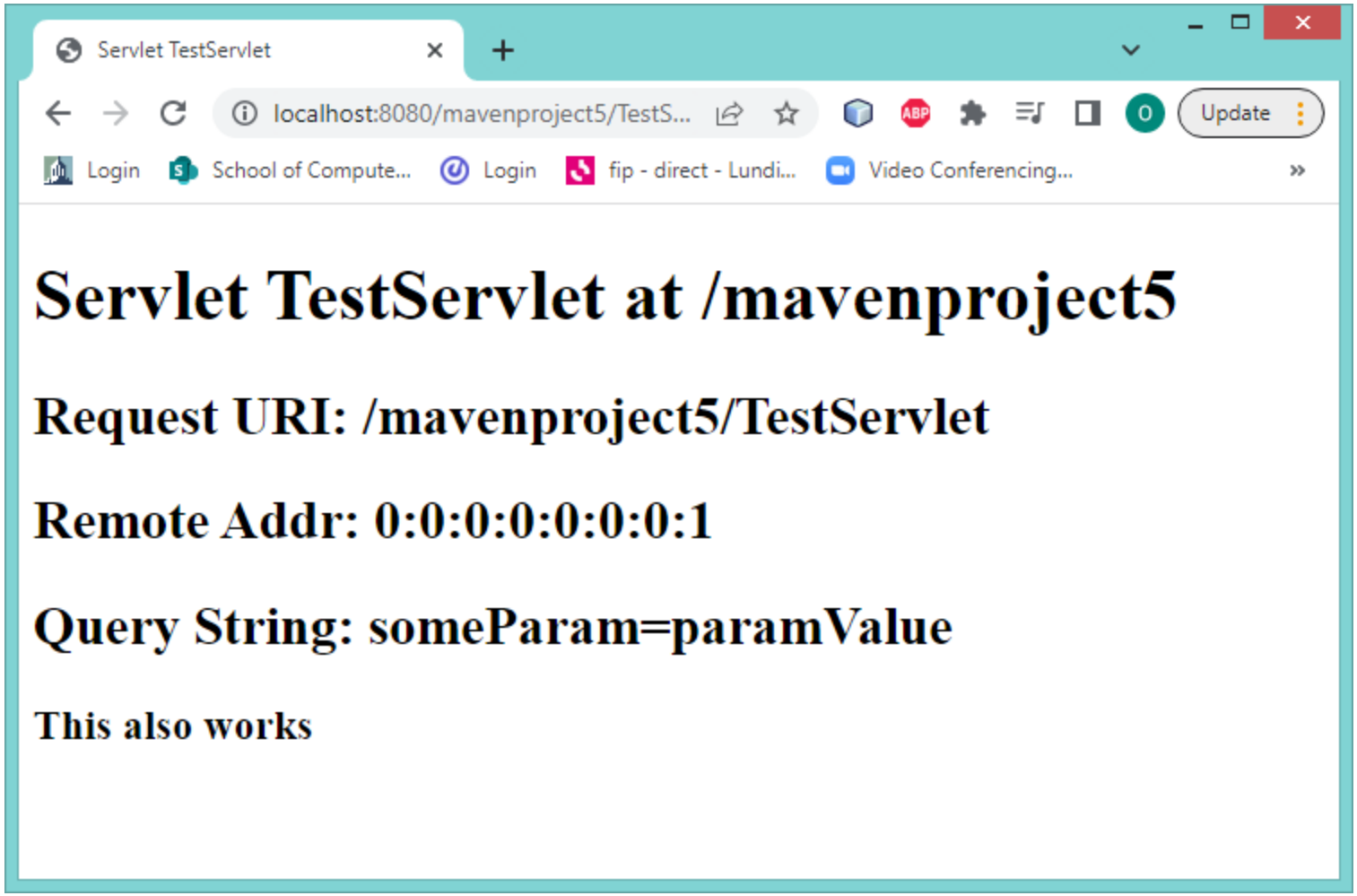
```
response.setContentType("text/html");
```

Dynamic HTML output

- Just building the response in HTML
- Can also append to the response using getWriter().append()

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");

    try ( PrintWriter out = response.getWriter()) {
        /* TODO output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet TestServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet TestServlet at " + request.getContextPath() + "</h1>");
        out.println("<h2>Request URI: " + request.getRequestURI() + "</h2>");
        out.println("<h2> Remote Addr: " + request.getRemoteAddr() + "</h2>");
        out.println("<h2> Query String: " + request.getQueryString() + "</h2>");
        response.getWriter().append("<h3>This also works</h3>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Sample output



GET vs POST

The GET Method

Note that query strings (name/value pairs) is sent in the URL of a GET request:

`/test/demo_form.asp?name1=value1&name2=value2`

Some other notes on GET requests:

GET requests can be cached
GET requests remain in the browser history
GET requests can be bookmarked
GET requests should never be used when dealing with sensitive data
GET requests have length restrictions
GET requests should be used only to retrieve data

GET vs POST

The POST Method

Note that query strings (name/value pairs) is sent in the HTTP message body of a POST request:

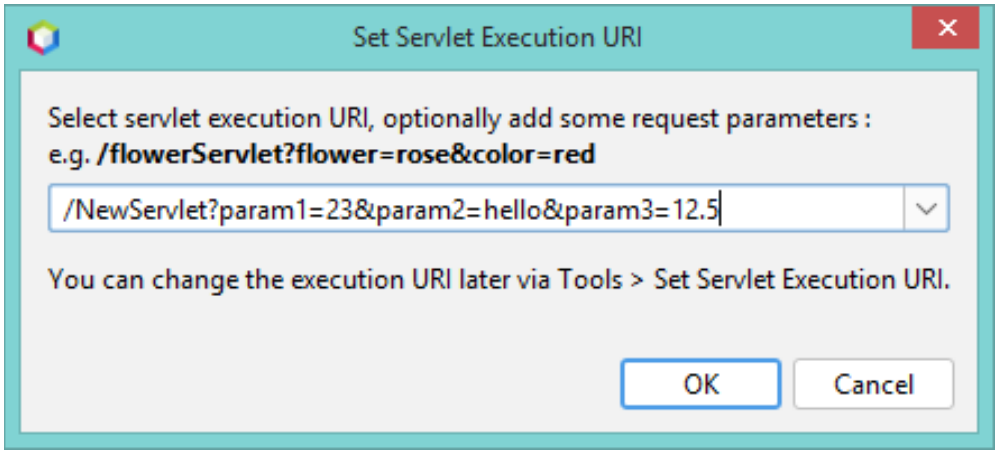
```
POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

Some other notes on POST requests:

POST requests are never cached
POST requests do not remain in the browser history
POST requests cannot be bookmarked
POST requests have no restrictions on data length

Using query string to send parameters

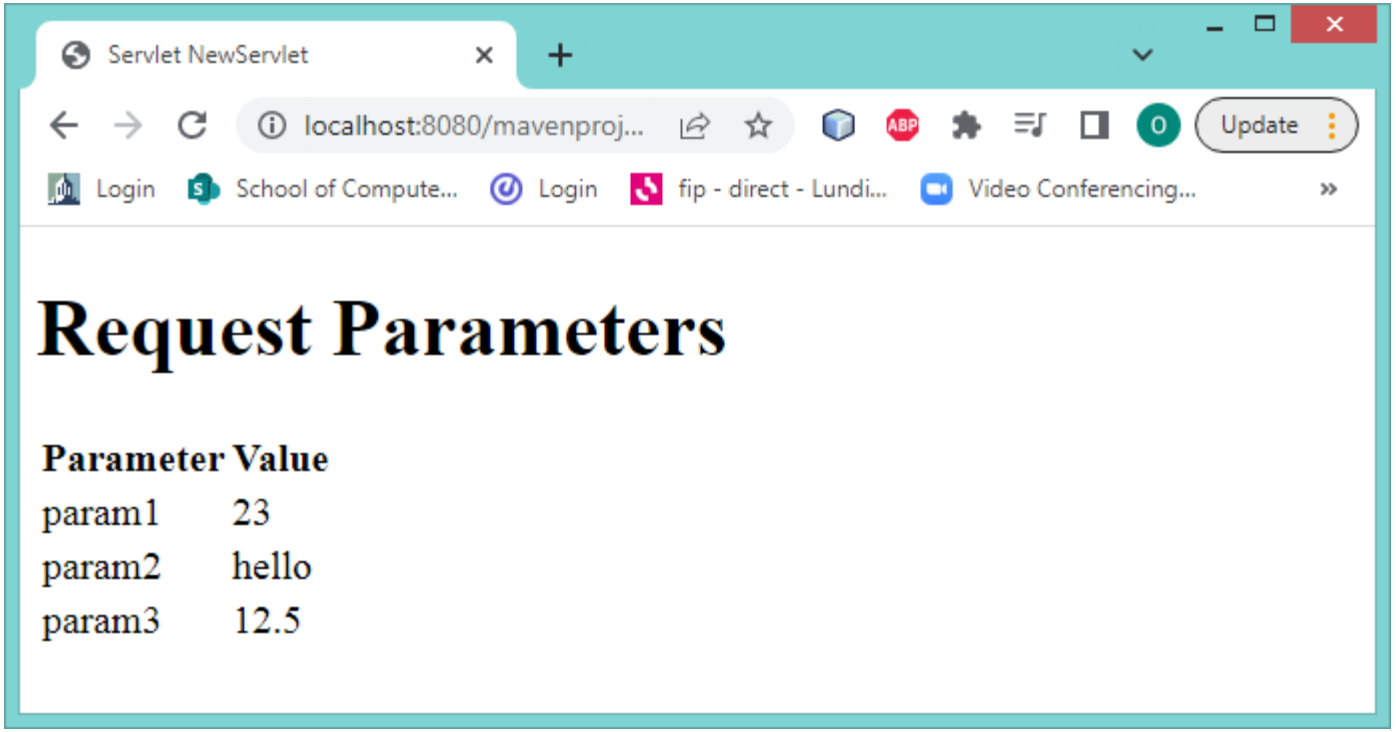
- You can send request parameters (e.g. from HTML form) in the URL (GET) or in the body (POST)
- To use the GET method, right click in the body of the servlet code
- You can then add parameters in the URL, e.g.



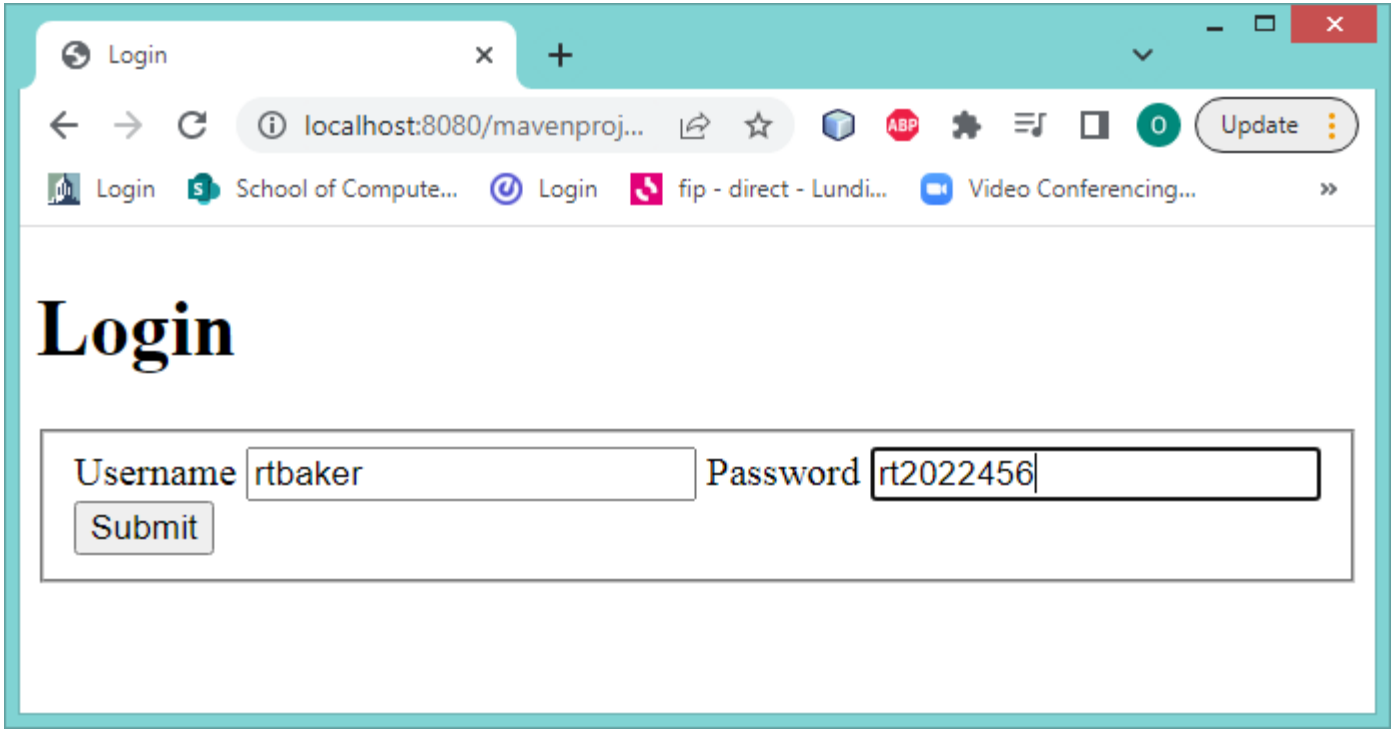
Retrieving request parameters

```
response.setContentType("text/html;charset=UTF-8");

String param1 = request.getParameter("param1");
String param2 = request.getParameter("param2");
String param3 = request.getParameter("param3");

try ( PrintWriter out = response.getWriter())
{
    /* TODO output your page here. You may use following sample code. */
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet NewServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Request Parameters</h1>");
    out.println("<table><tr><th>Parameter</th><th>Value</th></tr>");
    out.println("<tr><td>param1</td><td>" + param1 + "</td></tr>");
    out.println("<tr><td>param2</td><td>" + param2 + "</td></tr>");
    out.println("<tr><td>param3</td><td>" + param3 + "</td></tr>");
    out.println("<table>");
    out.println("</body>");
    out.println("</html>");
}
```

Browser displays response from the servlet



Sending request from a form



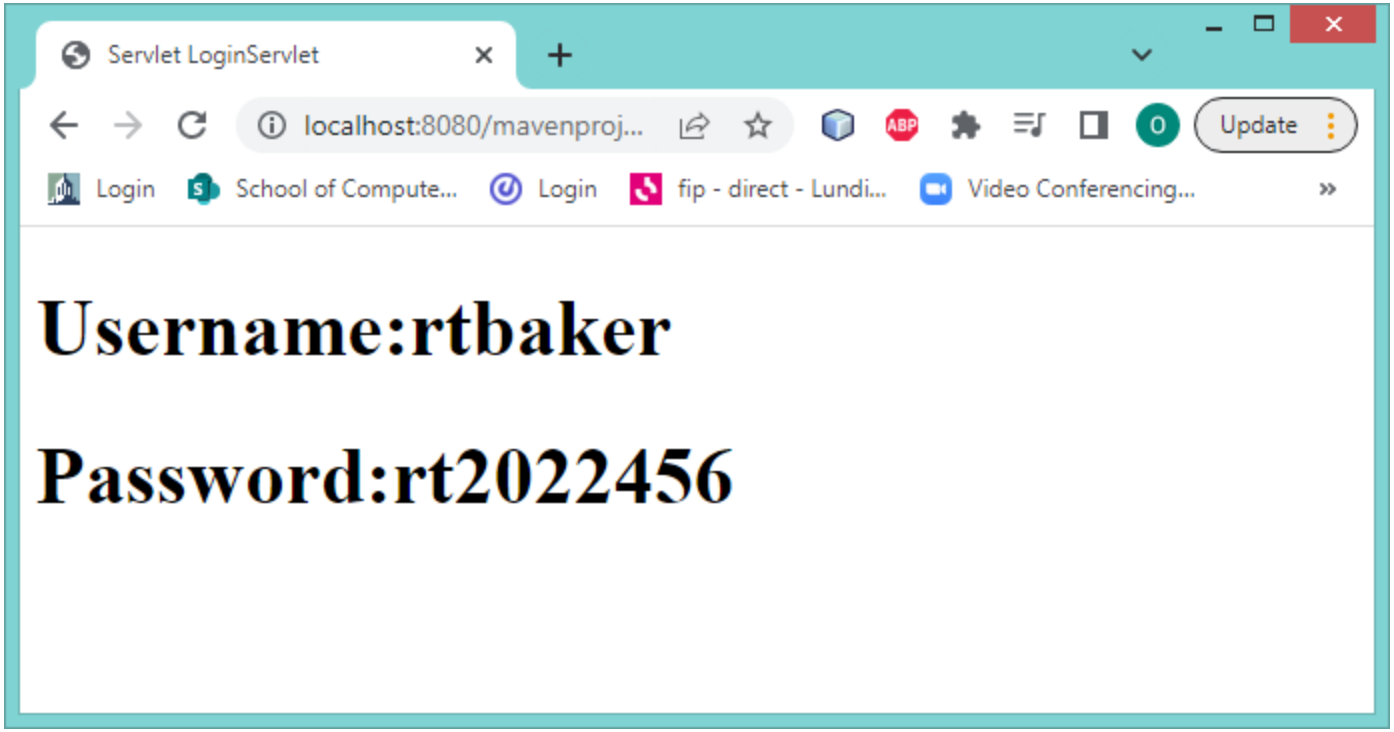
HTML form

```
<body>
    <h1>Login</h1>
    <form action="LoginServlet" method="post">
        <fieldset>
            <label>Username</label>
            <input type="text" name="username">
            <label>Password</label>
            <input type="text" name="password">
            <input type="submit" value="Submit">
        </fieldset>
    </form>
</body>
```

The servlet code

```
response.setContentType("text/html;charset=UTF-8");

String username = request.getParameter("username");
String pwd = request.getParameter("password");

try ( PrintWriter out = response.getWriter())
{
    /* TODO output your page here. You may use following sample code. */
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet LoginServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Username:" + username + "</h1>");
    out.println("<h1>Password:" + pwd + "</h1>");
    out.println("</body>");
    out.println("</html>");
}
```

Result


Servlet LoginServlet
localhost:8080/mavenproj...
Update
Login
School of Compute...
Login
fip - direct - Lundi...
Video Conferencing...
Username:rtbaker
Password:rt2022456

Dynamic => need changing data

- Normally we build the web pages dynamically using data, typically just simple POJO's / Java Bean classes which carry data back from the persistence layer to our View layer
- Servlets often used as the routers, forwarding requests to the appropriate business logic (session beans), and to the view layer (e.g. Java Server Pages)

e.g. add new user object

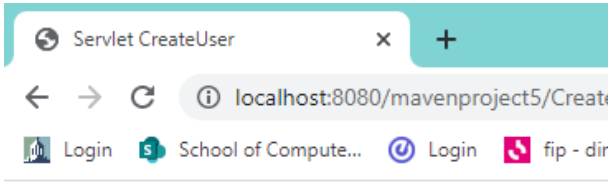
- Starting with form which calls a servlet to create a new user

```
<form action="CreateUser" method="post">
    <fieldset>
        <label>email</label>
        <input type="text" name="email">
        <label>name</label>
        <input type="text" name="name">
        <label>password</label>
        <input type="text" name="pwd">
        <input type="submit" value="Submit">
    </fieldset>
</form>
```

```
String email = request.getParameter("email");
String name = request.getParameter("name");
String pwd = request.getParameter("pwd");

User newUser = new User(email, name, pwd);

// assume we have an object that contains a list of current users
// we could add our new user to this list
Users myusers = new Users();

ArrayList<User> userlist = (ArrayList<User>) myusers.getUsers();

userlist.add(newUser);

try ( PrintWriter out = response.getWriter())
{
    /* TODO output your page here. You may use following sample code. */
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet CreateUser</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Users</h1>");
    out.println("<table><tr><th>email</th><th>Name</th></tr>");

    for (User u : userlist)
    {
        out.println("<tr><td>" + u.getEmail() + "</td><td>" + u.getName() + "</td></tr>");
    }

    out.println("<table>");

    out.println("</body>");
    out.println("</html>");
}
```

Output from servlet

- Simple example, but shows that we can build fairly useful web app using simple building blocks
- Also shows the need for Persistence (data layer), session management (keep track of current state of user session / application), validation (e.g. of inputs(), redirect (if user creation fails, then redirect to error page or back to user creation page



Users

email	Name
jm@ebay.uk	J. Murphy
arit@disc.com	Ari
s19@peqx.ie	Max
lib12@gmail.com	libra

Request Dispatcher

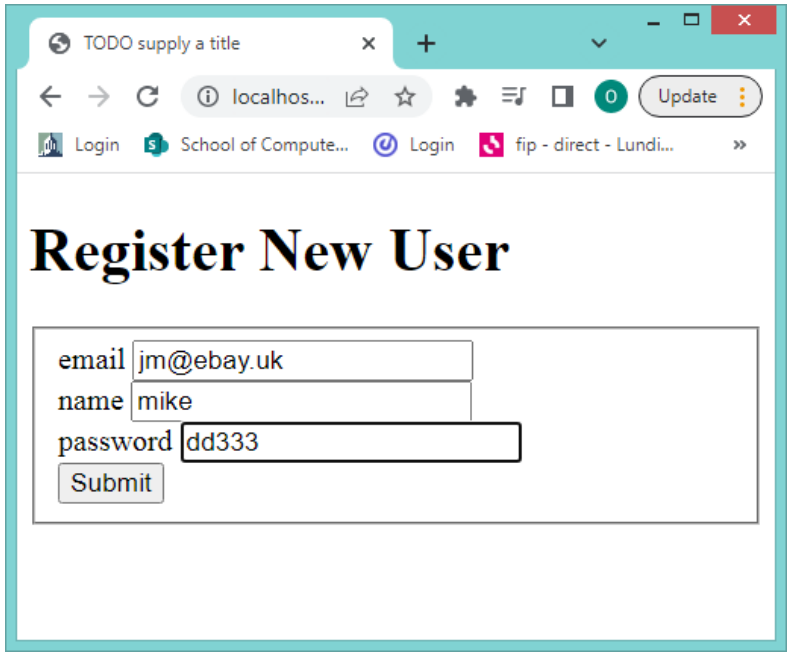
- The RequestDispatcher class enables your servlet to "call" another servlet from inside another servlet. The other servlet is called as if an HTTP request was sent to it by a browser.

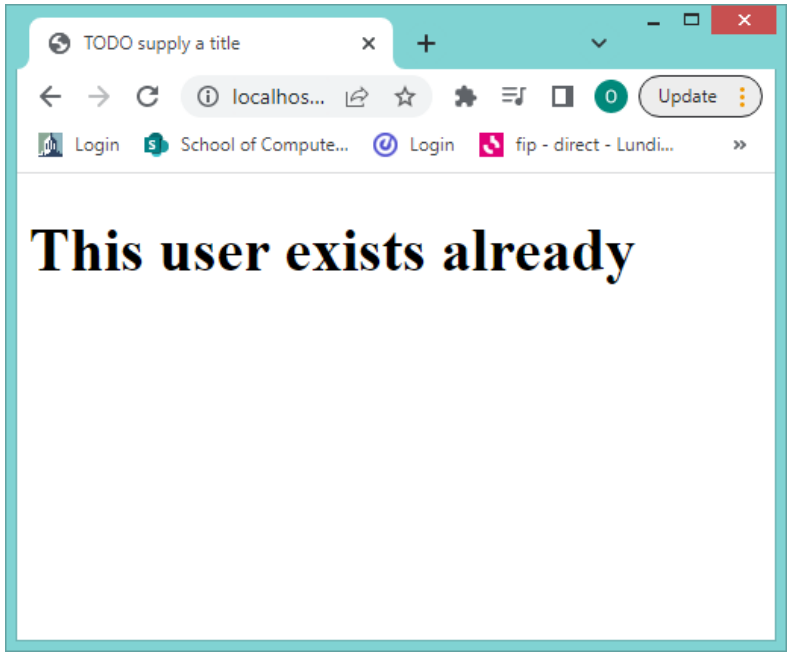
```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException

{
        RequestDispatcher requestDispatcher =
request.getRequestDispatcher("/anotherURL.simple");

// You can call the RequestDispatcher using either its include()
// or forward() method:

requestDispatcher.forward(request, response);


}
```

requestDispatcher example

- Check if user exists before proceeding, and redirect if yes

```
for (User u : userlist)
{
    if (u.getEmail().equals(newUser.getEmail()))
    {
        RequestDispatcher requestDispatcher = request.getRequestDispatcher("userCreationFailed.html");
        requestDispatcher.forward(request, response);
    }
}
```

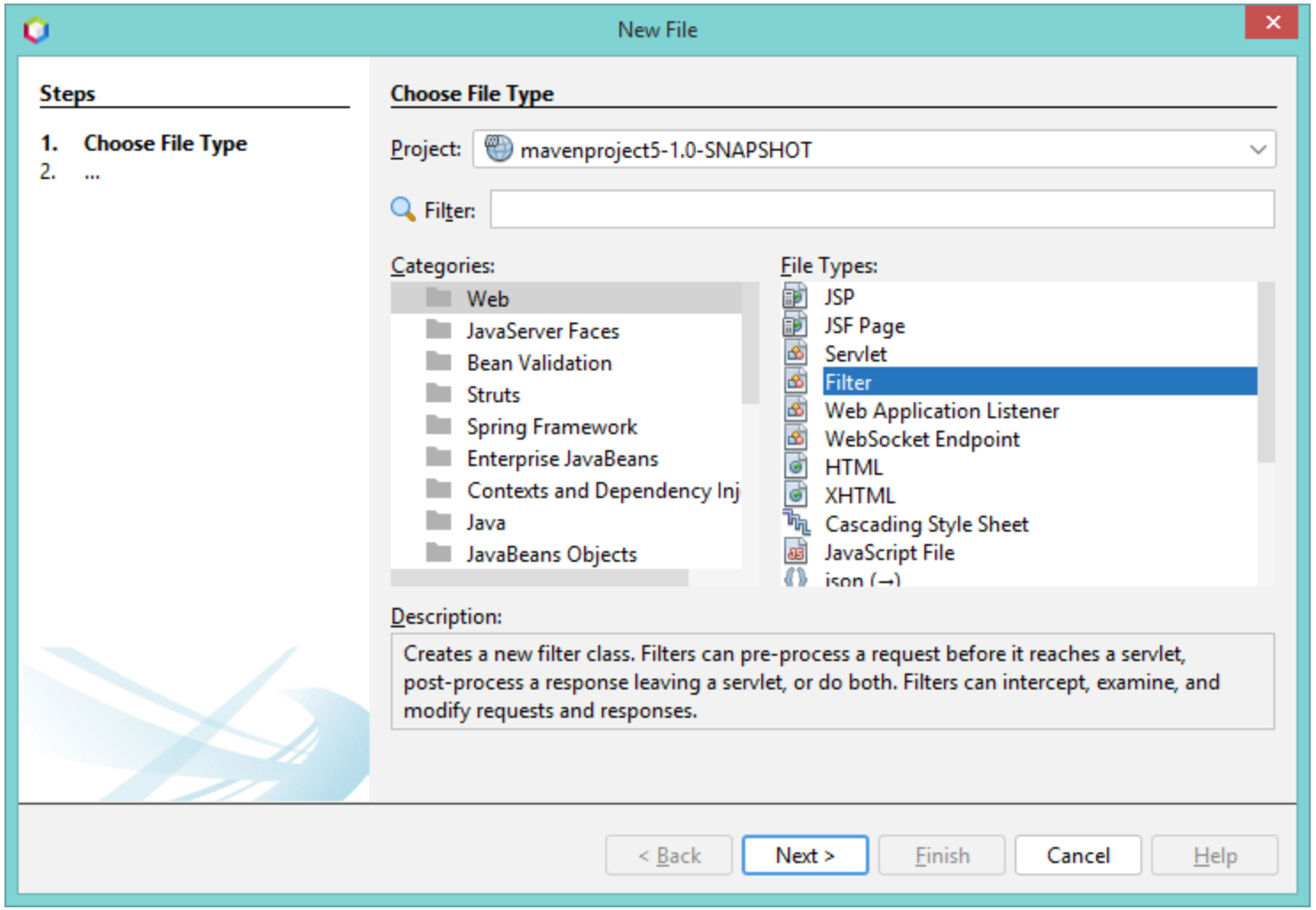

TODO supply a title
localhos...
Update
Login
School of Compute...
Login
fip - direct - Lundi...
Register New User
email jm@ebay.uk
name mike
password dd333
Submit


TODO supply a title
localhos...
Update
Login
School of Compute...
Login
fip - direct - Lundi...
This user exists already

Filters

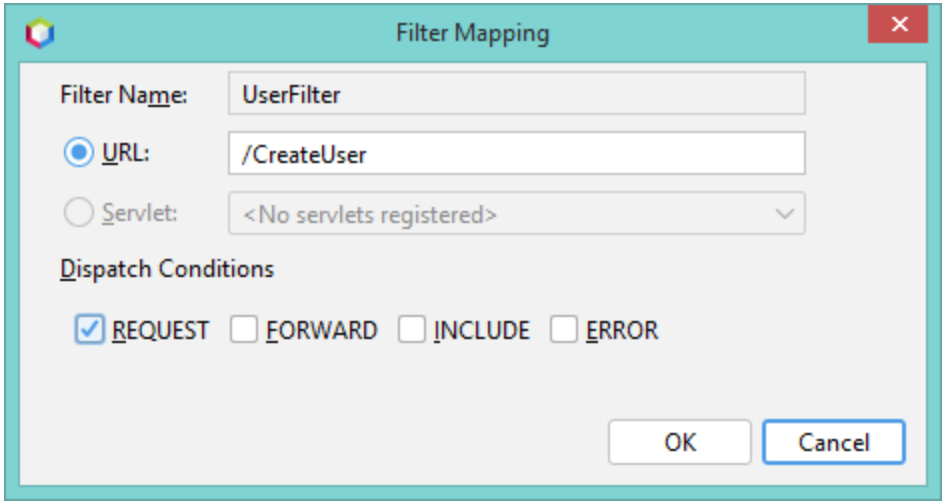
- Would be useful to be able to apply some filtering to URLs and requests / responses before sending them on their way
- For example, there could be checks to perform on headers, body, request parameters etc. before passing on to any servlets
- Or in this following simple example, we can use to check the email address for the presence of the '@' symbol

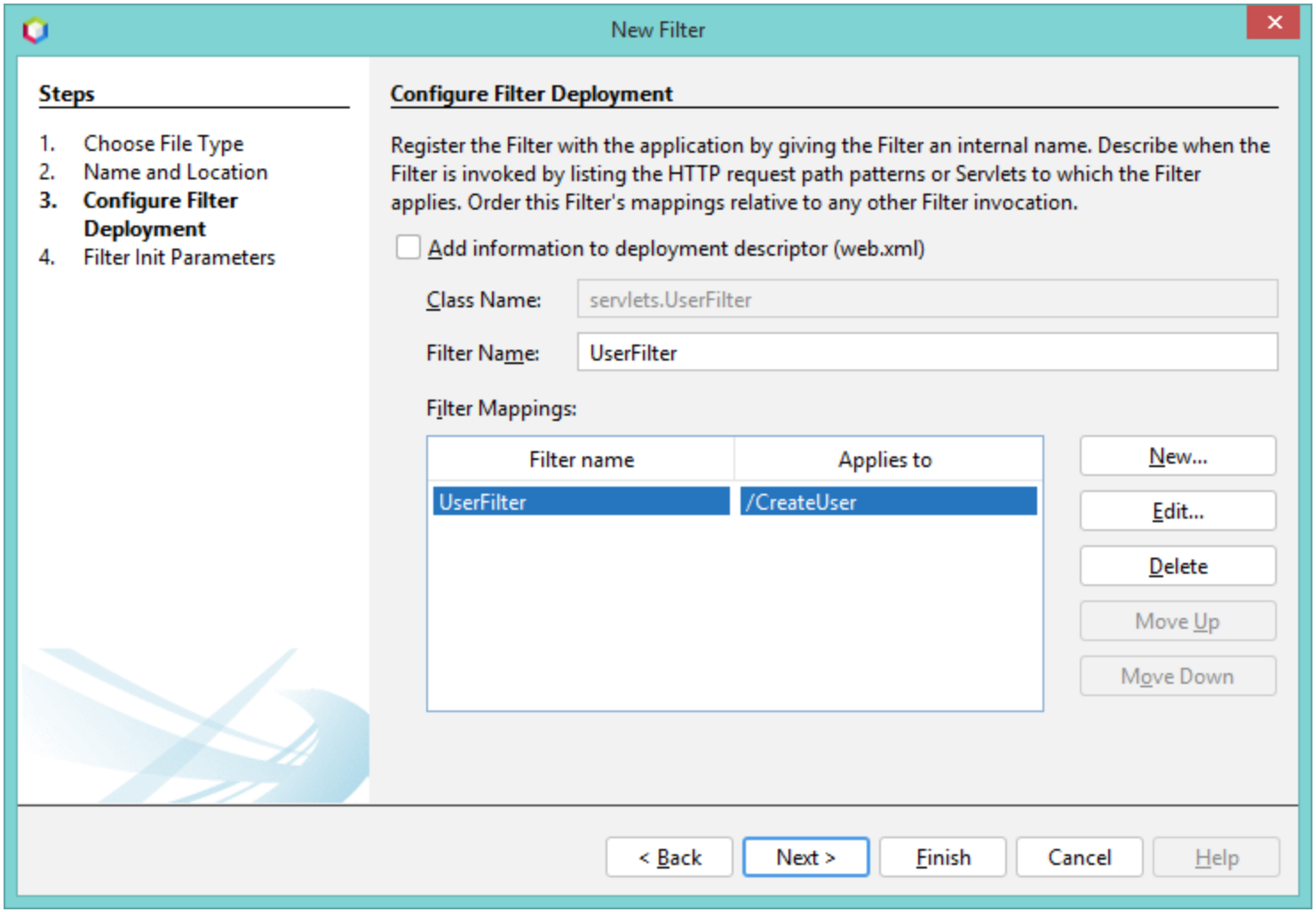
Create new class will implement the Filter interface

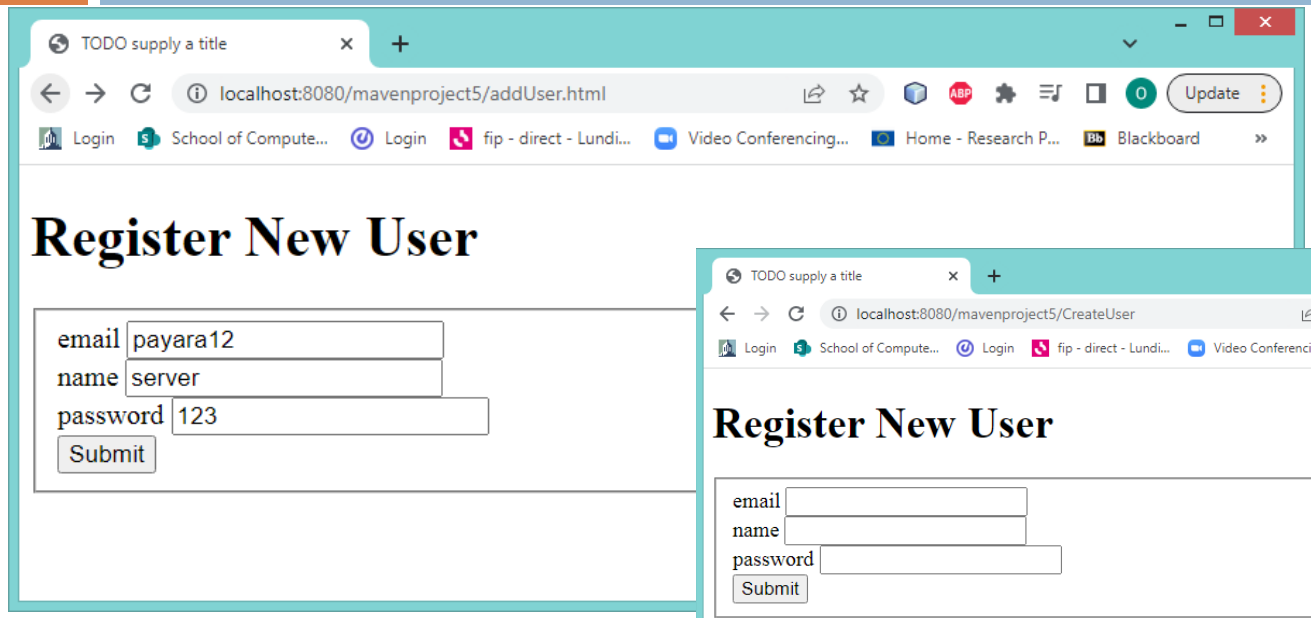


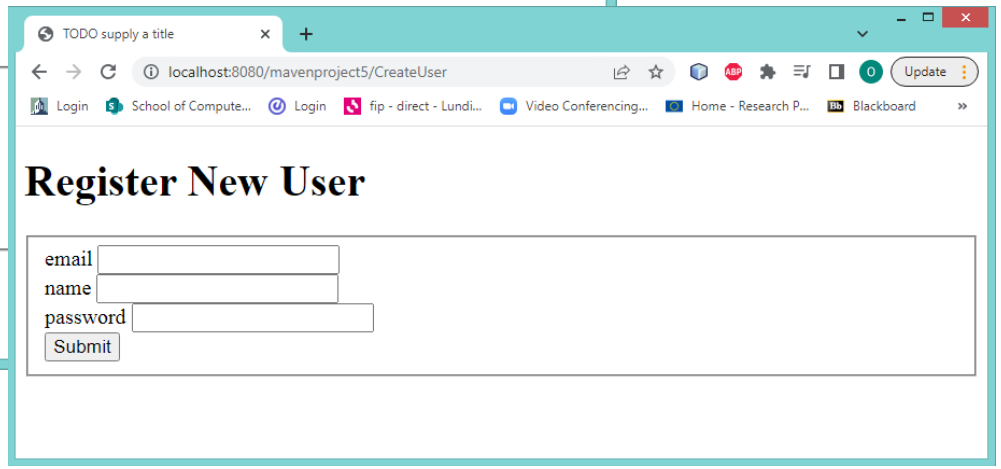
Specify the URL pattern

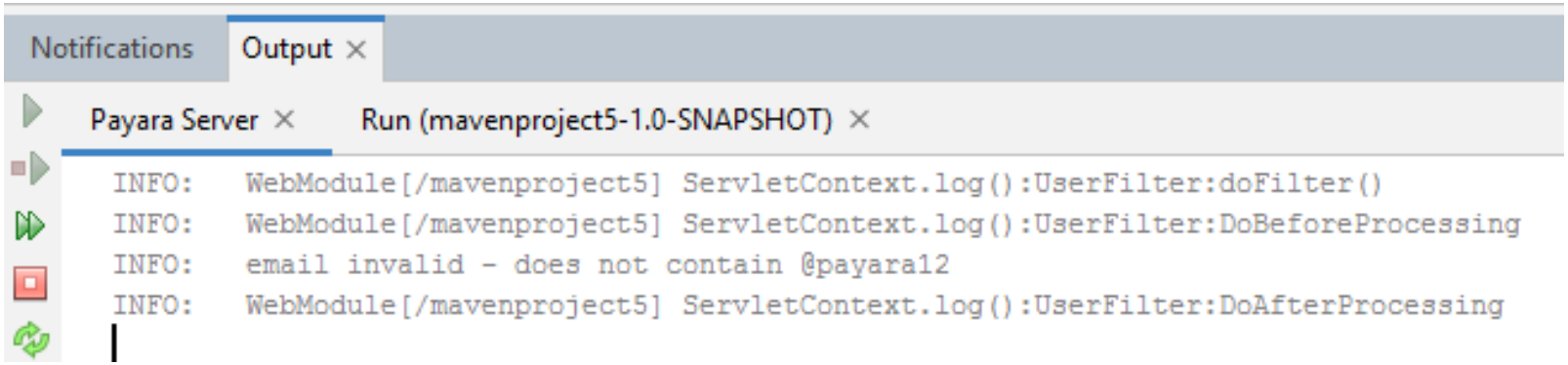
- Which URL patterns to apply the filter to
- /* would apply to all
- Here we only apply to calls to a specific servlet




New Filter
Steps
1. Choose File Type
2. Name and Location
3. Configure Filter Deployment
4. Filter Init Parameters
Configure Filter Deployment
Register the Filter with the application by giving the Filter an internal name. Describe when the Filter is invoked by listing the HTTP request path patterns or Servlets to which the Filter applies. Order this Filter's mappings relative to any other Filter invocation.
Add information to deployment descriptor (web.xml)
Class Name:
servlets.UserFilter
Filter Name:
UserFilter
Filter Mappings:
Filter name
Applies to
UserFilter
/CreateUser
New...
Edit...
Delete
Move Up
Move Down
< Back
Next >
Finish
Cancel
Help


TODO supply a title
localhost:8080/mavenproject5/addUser.html
Update
Login
School of Compute...
Login
fip - direct - Lundi...
Video Conferencing...
Home - Research P...
Blackboard
Register New User
email payara12
name server
password 123
Submit


TODO supply a title
localhost:8080/mavenproject5/CreateUser
Update
Login
School of Compute...
Login
fip - direct - Lundi...
Video Conferencing...
Home - Research P...
Blackboard
Register New User
email
name
password
Submit


Notifications
Output
Payara Server
Run (mavenproject5-1.0-SNAPSHOT)
INFO: WebModule[/mavenproject5] ServletContext.log():UserFilter:doFilter()
INFO: WebModule[/mavenproject5] ServletContext.log():UserFilter:DoBeforeProcessing
INFO: email invalid - does not contain @payara12
INFO: WebModule[/mavenproject5] ServletContext.log():UserFilter:DoAfterProcessing

Filter code

- Here we are interrupting processing of the filter chain (there could be multiple filters applied to some URL patterns) and redirecting back to the addUser.html page

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

    if (debug)
    {
        log("UserFilter:doFilter()");
    }

    doBeforeProcessing(request, response);

    Throwable problem = null;
    try
    {
        String email = request.getParameter("email");
        if (!email.contains("@"))
        {
            System.out.println("email invalid - does not contain @" + email);
            request.getRequestDispatcher("addUser.html").forward(request, response);
        } else
        {
            chain.doFilter(request, response);
        }
```

Session management

- A Session is a conversation between client and server – multiple requests and responses
- We need a way to identifying which session (client) each request belongs to
- There are a number of ways of doing this
  - URL rewriting
    - Attaching a session identifier with every request and response
    - Servlets support doing this in case cookies are disabled
  - Cookies – storing small pieces of information on client (sent back via data in the response header)
  - Session Management API
    - Server just stores a single piece of information on the client (jsessionid) as a cookie and uses it to associate the client with it's own session object which is held on the server

Session Tracking

- Cookies?
  - You can use cookies to store an ID for a shopping session; with each subsequent connection, you can look up the current session ID and then use that ID to extract information about that session from a lookup table on the server machine. So, there would really be two tables: one that associates session IDs with user tables, and the user tables themselves that store user-specific data.
- URL Rewriting
  - With this approach, the client appends some extra data on the end of each URL. That data identifies the session, and the server associates that identifier with user-specific data it has stored. For example, with http://host/path/file.html;jsessionid=a1234, the session identifier is attached as jsessionid=a1234, so a1234 is the ID that uniquely identifies the table of data associated with that user.

Sending and Receiving Cookies

- To send cookies to the client, a servlet should use the Cookie constructor to create one or more cookies with designated names and values, set any optional attributes with cookie.set*Xxx* *(readable later by cookie.getXxx), and insert the cookies* into the HTTP response headers with response.addCookie.
- To read incoming cookies, a servlet should call request.getCookies, which returns an array of Cookie objects corresponding to the cookies the browser has associated with your site (null if there are no cookies in the request). In most cases, the servlet should then loop down this array calling getName on each cookie until it finds the one whose name matches the name it was searching for, then call getValue on that Cookie to see the value associated with the name.

```
Cookie userCookie = new Cookie("user", "uid1234");
userCookie.setMaxAge(60*60*24*365); // Store cookie for 1 year
response.addCookie(userCookie);
```

Reading Cookies

```
String cookieName = "userID";
Cookie[] cookies = request.getCookies();
if (cookies != null) {
  for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookieName.equals(cookie.getName())) {


      doSomethingWith(cookie.getValue());
    }
  }
}
```

Using Servlet HttpSession API

- **Accessing the session object associated with the current request.**
  - Call request.getSession to get an HttpSession object, which is a simple hash table for storing user-specific data.
- **Looking up information associated with a session.**
  - Call getAttribute on the HttpSession object, cast the return value to the appropriate type, and check whether the result is null.
- **Storing information in a session.**
  - Use setAttribute with a key and a value.
- **Discarding session data.**
  - Call removeAttribute to discard a specific value. Call invalidate to discard an entire session. Call logout to log the client out of the Web server and invalidate all sessions associated with that user.

Accessing the session object

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException

{

        HttpSession session = request.getSession();

}
```

get / set attribute values in the session object

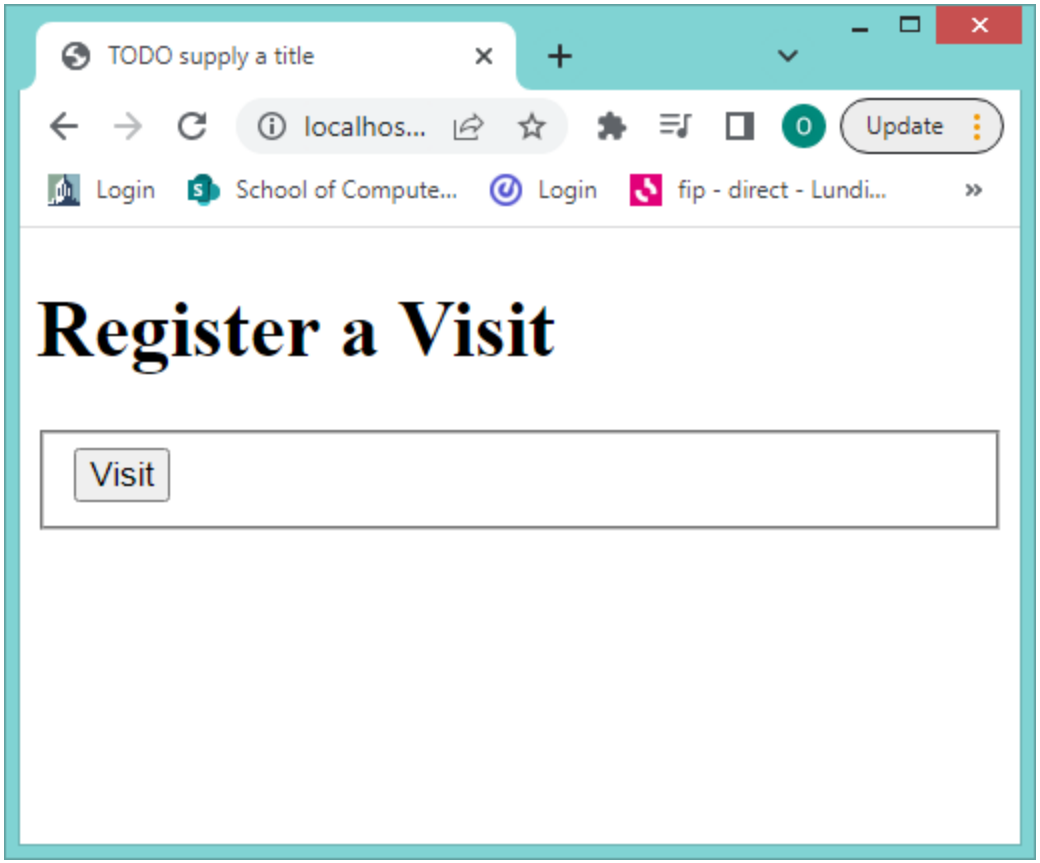
```
session.setAttribute("userName", "John123");
```

```
String userName = (String) session.getAttribute("userName")
```

- We keep *Attributes* on the session, whereas *Parameters* are what are passed in on the request (e.g. from a *Form*)
- We will see further on that we can also associate attributes to both the request and the application scope, as well as the session scope
  - E.g. we can set attributes that just live for the scope of the request, or also ones that can be shared across the application with all clients

Simple example

- Use session to keep track of number of visits to a page



Servlet code

```
HttpSession session = request.getSession();

Integer numvisits = (Integer) session.getAttribute("numvisits");
if (numvisits == null)
{
    numvisits = 1;
} else
{
    numvisits++;
}

System.out.println("Number of Visits = " + numvisits);

session.setAttribute("numvisits", numvisits);

request.getRequestDispatcher("visit.html").forward(request, response);
```

- If session doesn't already exist for this client, then a new one is created and returned by getSession()

- System output