Web Services | HTTP | Servlets

Web Services

- a service is, essentially, software that provides services to other components over a network
- a web service is the above, where the network is the internet
- follows service-oriented architecture (SOA)
    1. logically represents a business activity with a specified outcome
    2. self-contained
    3. black box for consumers (they don't know the details of their interaction with the software)
    4. may consist of underlying services (doesn't have to)
- two main types
    - SOAP: simple object access protocol. older, more rigid design, transfers XML documents, any protocol
    - REST: representational state transfer. newer, more flexible, transfers any type of document, protocol must be HTTP

REST

- REpresentational State Transfer
- a web service architecture designed by Roy Fielding
- transfer the state of a representation of an object to a database
- a RESTful web service must meet the following constraints:
    1. client-server architecture: the front end and back end of the application are two separate applications (i.e. Java back end, Angular/React/etc. front end)
    2. statelessness: the server does not store client context between requests, all necessary information is sent to the server in every request from the front end
    3. cacheability: particular responses from the back end are able to be cached for easy reuse, especially if they are used often
    4. layered system: the user cannot tell whether there are any intermediaries between them and the database
    5. code on demand: OPTIONAL. where necessary, the back end can send scripts to offer additional functionality to the front end as a response.
    6. uniform interface: requests/responses use the following patterns:
        - URI model: requests are sent in a particular format: /collection/identifier
        - resources are manipulated through representations, i.e. when changes are made, they are made to the representations being sent, rather than in the back end after certain directions were given.
        - self-descriptive messages: a GET request to "/cat" should get the cats.
        - HATEOAS: hypermedia as the engine of application state. when a resource has a lot of information, it can include URIs so that the client can send requests to those URIs for further information.

HTTP

- hypertext transfer protocol: a client-server protocol
    - client to server = front-end to back-end
    - client initiates the communication
- follows a request-response format, sort of like sending a letter then receiving one back.
- http requests consist of:
    - verb: what HTTP method is being used
    - URI: the endpoint of the requested resource
    - HTTP version
    - request header: metadata
    - request body: message content
- http responses consist of:
    - status code
    - http version
    - response header: metadata
    - response body: message content
- http methods (verbs)
    - GET: used for retrieving a resource or set of resources. request body is empty.
    - POST: sends an entity for the server to accept, usually used for creating/adding resources (a POST request in a RESTful application should be sent to the URI of a collection, e.g. "/collection" with no identifier), often also used for logging in
    - PUT: sends an entity for the server to accept, usually used for updating resources (a PUT request in a RESTful application should be sent to the URI of the specific entity being changed, e.g. "/collection/identifier")
    - DELETE: specifies an entity to be deleted (URI pattern should be same as PUT), often also used for logging out
    - OPTIONS: sent by the browser, checks which verbs are allowed on the specified endpoint before actual request is sent there
- idempotence: if a function is idempotent, you can repeat it over and over with exactly the same results
    - GET, PUT, and DELETE are idempotent: the results after performing the method will be the same if you repeat it
    - POST is not idempotent
- status codes
    - 100s: informational
    - 200s: success
        - 201: created
    - 300s: redirect
    - 400s: client error
        - 400: bad request
        - 401: unauthorized (must be logged in)
        - 403: forbidden (cannot access even if logged in)
        - 404: not found
        - 405: method not allowed
        - 418: i'm a teapot (cannot brew coffee)
    - 500s: server error
        - 500: internal server error

        - 503: service unavailable (usually for maintenance)

Servlets

- a servlet is an object that handles requests and responses
- servlet hierarchy: Servlet (interface) GenericServlet (AC) FacesServlet (C) HTTPServlet (AC)
- servlets are not part of Java standard library
    - javax.servlet (javax = java extensions) can be included from the Maven repository
    - OR tomcat catalina can be used, which includes that package and acts as our servlet/web container
- web container
    - tomcat is our web container
    - manages servlet environment and lifecycle
    - reads deployment descriptor (web.xml)
    - calls the servlet lifecycle methods:
        - init: initializes the servlet, called once
        - service: is called every time a request is received
        - destroy: called when app stops running, once
- deployment descriptor (web.xml)
    - file that describes how an application should be deployed by the server
    - right click on project that has been set up to be packaged as a war file > Java EE Tools > Generate Deployment Descriptor Stub
    - this is where we define and map our servlets so that the web container knows where to find them and what requests to send to them
    - `<servlet-name>Name</servlet-name>`
    - `<servlet-class>com.revature.servlets.Name</servlet-class>`
    - `</servlet>`
    - `<servlet-mapping>`
    - `<servlet-name>Name</servlet-name>`
    - `<url-pattern>/name</url-pattern>`
    - `<servlet-mapping>```
    -
- each servlet will get a class, and in that class you can set up overridden methods for each HTTP method
    - protected void doGet (HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException { // this handles GET requests. }
    - protected void doPost (HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException { // this handles POST requests. }
    - etc.
- PrintWriter
    - the printwriter is an object that you get from the response object
        - resp.getWriter();
    - this allows you to write directly to the response body using .write()
    - you can write plain text, JSON, HTML, XML, etc. whatever is relevant to the particular response and application in general

- `<load-on-startup>1</load-on-startup>`
    - in web.xml, allows you to set the web container to call the "init" lifecycle method for the servlet right when the application starts rather than waiting until it receives its first request
- because the web container (tomcat) gains control of our program, there is NO main method

Servlets continued

config vs. context

- in the web.xml (deployment descriptor), you can specify servlet config and context
- config: parameters for a particular servlet
    - `<param-name>paramName</param-name>`
    - `<param-value>paramValue</param-value>`
    - `</init-param>`
    -
    - inside of the servlet tag for that servlet
- context: parameters for all of the servlets in the application
    - `<param-name></param-name>`
    - `<param-value></param-value>`
    - `</context-param>`
    -
    - at the same level as the servlet tags

redirect vs. forward

- forwards happen in the web container; the URL still reflects the request made when it gets back to the client but the response came from a different one (the one that we forward the request to)
    - req.getRequestDispatcher("/newURI").forward(req, resp);
- redirects send a response that triggers a new request to be made to the new URI that it is redirected to; the URL will be replaced with the new one
    - resp.sendRedirect("newURI");

Sessions

- a session represents an object that persists between requests from the same client (identified by an ID from the browser)
- there are three ways to associate a client with a session (i.e. maintain session identity), these are managed on the front end
    1. cookies
    2. URL rewriting (putting information in the URL)
    3. hidden form fields
    - you don't need to worry about these too much as you will not be managing them in your API

- HttpSession object, retrieved from the request
- req.getSession()
- session.setAttribute("name", value)
- session.getAttribute("name")
- session.invalidate()

Design Patterns

- MVC: Model View Controller
    - controller handles request flow
    - model handles data access/CRUD
    - view handles presentation rendering (for server-side rendering)
    - model & view NEVER interact, only accessed by the controller.
- Front Controller
    - front controller sends requests to handler/dispatcher
    - handler/dispatcher decides which delegate to send request to
    - front controller then sends the request to the appropriate delegate
    - delegate ("logic class") handles CRUD, presentation, etc.
    - can go hand-in-hand with MVC

Servlet Error/Exception Handling

- you can set up a particular error/exception handling servlet in the web.xml
- `<error-page>` is the outer tag
    - you can either have `<error-code>` with a status code or `<exception-type>` with a Java class for any error or exception (Throwable)
    - you then have a `<location>` to specify the url mapping to send the request to
        - this can match the url-mapping tag of the servlet you are using for the error/exception handling, or it could direct to a static page
- you can have many of these for different error codes and exception types, directing to the same servlet or different servlets that handle it differently

Query & Path Parameters

- when data is sent to the back end through an HTML form, it is sent as query parameters by default
- query parameters can also be defined directly in the request URI, like so:
    - localhost:8080/user/login?user=sierra&pass=p4ss
- path parameters are simply parameters passed in as part of the actual path (they do not use the ?=& syntax)
    - localhost:8080/user/5
    - when defining endpoints, path parameters are usually written like so: /user/:id
- query parameters are best for filtering purposes, or when parameters need to be passed that may not be necessary or useful to pass through the request body
    - for example, if you are logging in, you wouldn't probably be passing in an entire person object, so it's not really an ideal case to put in the request body (a username

and password alone are not going to be easily mapped unless you make a special bean for that, and...why would you?)
    - therefore, you can just pass in the pieces you need as query parameters. you don't have to worry about them being passed into the address bar if you are sending the request internally (always the case with single-page applications and/or pages that use AJAX, which is usually the case with a RESTful application)
- path parameters are best when you are specifying a particular resource from a collection using a unique identifier (not necessarily the primary key)
    - usually this would be ID, but it could also be something like a name if that is unique