Modern Web Applications

Devdatta Kulkarni

Fall 2016

Agenda

- Introductions
- Course goals and contents
- Prerequisites and Books
- Assignments
- Examinations
- Expectations from you
- Schedule
- Questions

Introductions

- Teaching staff:
  - Me:
    - Name: Devdatta Kulkarni
    - Email: devdatta@cs.utexas.edu
    - Office: GDC 4.812
  - Teaching Assistant:
    - Eddy Hudson
- You:

Web Applications

- Applications vs. Systems
  - Systems provide basic abstractions that are used for building applications
    - Examples:
      - Operating System and bash shell
      - File system and file browser
- Web Applications
  - Applications that are accessible over the Internet (“Web”)
  - Built using abstractions provided by Web systems
    - Web Servers

Characteristics of Web Applications

- Dynamic content
- Persistent storage
- Scalable
- Work across different platforms
- Secure
- Support concurrent requests

Characteristics cont.

- Robust and reliable
- Responsive
- API access
- Implementing current authentication and authorization standards
- Dependent on other such applications

‘Modern’ Web Applications

- Service-oriented Architecture
  - RESTful APIs (*we will learn)*
- Asynchronous
- Cloud-based
  - Use elastic ‘Cloud’ resources
- Built using modern programming tools, techniques and frameworks

Course Goal

- Learn about fundamental concepts in building modern web applications using some of the ‘modern’ tools and technologies
- Once you understand the concepts, changes in technologies won’t be difficult to handle for building next generations of such applications

Course contents

- Topics that we will learn
  - Hypertext transfer protocol (HTTP), Servlets, Session management, Logging, Dependency Injection, Spring framework, Unit testing, Functional testing, REST, XML and Json parsing, JavaScript, Cloud computing paradigms, OpenStack
- What we won't cover
  - Mobile development (Android/iPhone programming)

Prerequisites

- Principles of Computer Systems (CS 439)
  - Understanding of synchronization, race condition, etc.
- Programming experience in Java/C# (CS 312, CS 314)

Books

- Class Notes
- Reference Books
  - Professional Java for Web Applications Featuring WebSockets, Spring Framework, JPA Hibernate, and Spring Security
    - Author: Nicholas S. Williams
  - RESTful java with JAX-RS 2.0 Designing and Developing Distributed Web Services
    - Author: Bill Burke
  - Online tutorials

Course components

- Assignments
- Home works
- Examinations
  - Midterm
  - Final

Assignments

- 6 programming assignments
  - 5 (Java), 1 (Java + JavaScript)
- Submission
  - We will use Bitbucket for assignment submissions
- Points and late policy
  - Each assignment will be out of 100 points
  - You loose 5 points for each late day
- To be done individually
- Assignment grading
  - May require demonstration to TA

Home work

- Home works will be assigned for self-study purpose
- Important to do them, as some of the questions in midterm and final may be based on home works

Examinations

- Midterm
  - Date: TBD
- Final
  - Examination week
- Exam format
  - Open book vs. closed book
    - Will announce later

Grade distribution

- 6 assignments: 12% each
- Midterm: 13 %
- Final: 15 %

Class communication tools

- Canvas
  - Will be used for
    - Announcements
    - Publishing class notes and assignments
    - Grades
- Github examples
  - https://github.com/devdattakulkarni/ModernWebApps
- Piazza
  - Class discussions
    - Will post sign-up link on Canvas

Grade cutoffs

- > 95%: A
- 90 - 95: A-
- 85 - 90: B+
- 80 - 85: B
- 75 - 80: B-
- 70 - 75: C+
- 65 - 70: C
- 60 - 65: C-
- 55 - 60: D+
- 50 - 55: D
- 45 - 50: D-
- < 45: F

Office hours

- Devdatta Kulkarni:
  - After class on Thursday (starting September 1)
    - 7.15pm – 8.30pm in GDC 6.202
- TA office hours:
  - TBD

Schedule

Dates	Tentative topics
Week 1	Class introduction, HTTP
Week 2	HTTP
Week 3	Servlets, Session management, XML Parsing
Week 4	Unit testing, Logging
Week 5	Dependency injection
Week 6	Spring framework
Week 7	Functional testing
Week 8	REST

Schedule

Dates	Tentative topics
Week 9	Midterm
Week 10	Databases, JDBC
Week 11	JDBC, ORM
Week 12	ORM
Week 13	JavaScript
Week 14	JavaScript, Cloud computing
Week 15	Cloud computing
Week 16	Cloud computing, review
Week 17 (Finals week)	Final

Schedule Implications

- Two to three lectures per topic
  - We won’t be able to cover every aspect of the topic
    - Self-study will be required
    - Most of the learning will happen while implementing assignments
- How to succeed in the course?
  - Start early on assignments
  - Don’t loose patience with the code
  - Embrace debugging
  - Write unit and functional tests (we will learn how to do this)

Questions?

Hypertext Transfer Protocol (HTTP)

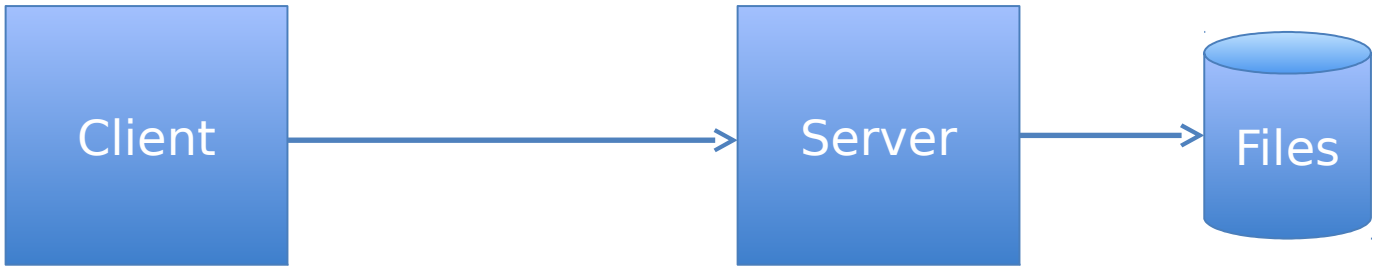
Devdatta Kulkarni

HTTP

- **H**yper**T**ext
  - Text and Binary data (images, doc files, pdf files, etc.)
- **T**ransfer
  - Act of moving the hypertext between client and server
- **P**rotocol
  - Set of agreed upon actions
- HTTP:
  - A protocol that defines how to move hypertext between client and server *over Internet*
- Over Internet – What does this mean?
  - IP address of the server is publicly accessible
  - The server process is running on port 80 (typically)

Problem

- What aspects do we need to consider when building a protocol that allows files of hypertext stored on a server to be requested by Client?



5 minute exercise

- Aspects to consider:
  - Standard way of identifying yourself
  - Integrity availability confidentiality of hypertext
  - How to support hypertext of different sizes?
  - Reliable communication between client and server
  - Uniform and standard way to represent hypertext
  - Identify specific resources and do actions on them
  - How to manage state?
  - How to handle failures?

Protocol design considerations

- Clients should be able to unambiguously request one file from another
- There should be a mechanism to differentiate good Client request from a bad request from a Client
- Files to serve may be large
- Clients should be able to request transfer of files only if they don’t already have them

Relevant HTTP mechanisms

- **URLs** to uniquely reference resources (files) on a server
- **Status codes** to distinguish good requests from bad requests
- **Chunking mechanism** to transfer large resources from the server to the client
- **Caching mechanisms** to conditionally request resources

HTTP

- Request-response protocol
- Where is it implemented?
  - Within browsers, web servers, proxy servers, load balancers, web application containers
- Version 1.0 published in 1996
  - http://www.rfc-base.org/txt/rfc-1945.txt
- Version 1.1 published in 1999
  - https://www.ietf.org/rfc/rfc2616.txt

HTTP

Client and Server interact with each other using HTTP

HTTP: Main concepts

- Resources
- Methods
- Protocol Operation
  - Request
  - Response

Resources

- A resource is an entity maintained by the web application running on the server
- Resources are organized in a hierarchy
  - Similar to file system hierarchy
- Examples:
  - "index.html" file when we visit a web page:
    - http://www.cs.utexas.edu/~devdatta/index.html
  - A repository stored on github.com
- A resource is identified by Uniform Resource Locator (URL)

HTTP URLs

http_url = "http:" "//”host[":”port]
[abs_path]

host: A legal Internet host domain name or IP address

port: 0 or more digits
if port is empty or not given, port 80 is assumed

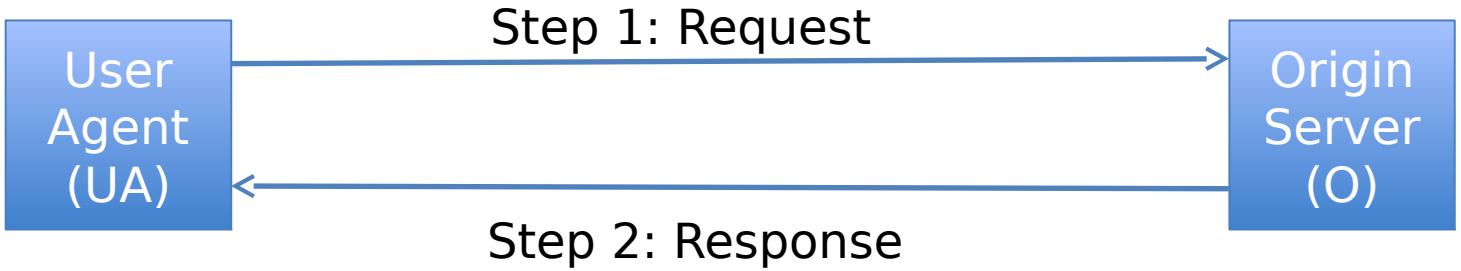
abs_path: path of resource on the server

Methods

- A method indicates what to do with a resource
  - Retrieve information
  - Update/delete information
  - Delete the resource
  - Create a new resource (by making a call to its parent resource)
- HTTP defines several methods which have pre-defined meaning. Example:
  - GET
    - Get information about a resource
  - POST
    - Create a resource
  - PUT
    - Modify information about a resource
  - DELETE
    - Delete a resource

Protocol Operation

Protocol Operation



Client/UA initiates a socket connection and
sends the request to the server

Server sends the response and closes the socket connection

Tools to interact with Web applications

- curl
- Chrome Apps and Extensions
  - Advanced Rest Client
  - Postman
- Firefox add-on
  - Poster
- Firefox built-in tools
  - Tools → Web Developer → Web Console → Network
- Chrome built-in tools
  - Tools → Developer tools → Network
- Java programs
  - TestClientSocket from https://github.com/devdattakulkarni/ModernWebApps/tree/master/HTTP

Examples

- curl -i http://www.cs.utexas.edu/~devdatta/test-file.txt

- curl -i http://www.cs.utexas.edu/~devdatta/test-file.txt \

-H "Host: www.cs.utexas.edu" \

-H "User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:44.0) Gecko/20100101 Firefox/44.0" \

-H "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8" \

-H "Accept-Language: en-US,en;q=0.5" \

-H "Accept-Encoding: gzip, deflate" \

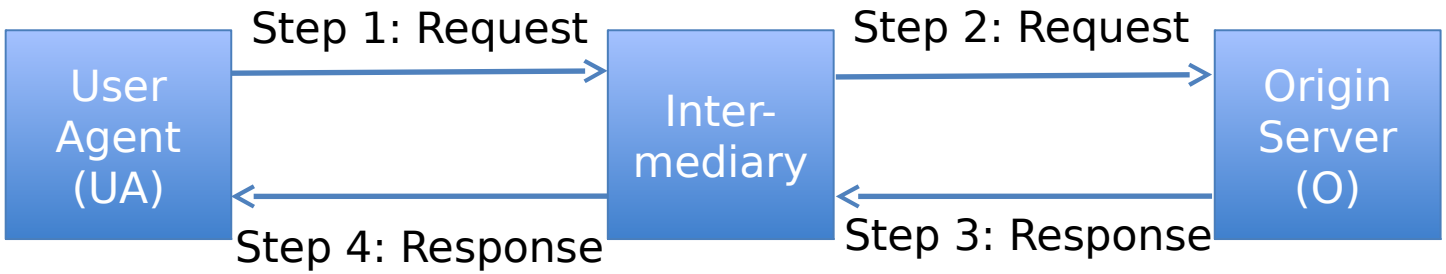
-H "Cookie: _ga=GA1.2.699921686.1471968877" \

-H "Connection: keep-alive" \

-H "If-Modified-Since: Tue, 30 Aug 2016 01:42:06 GMT" \

-H "If-None-Match: "17a1e7f-2-53b4016909fa7"" --compressed

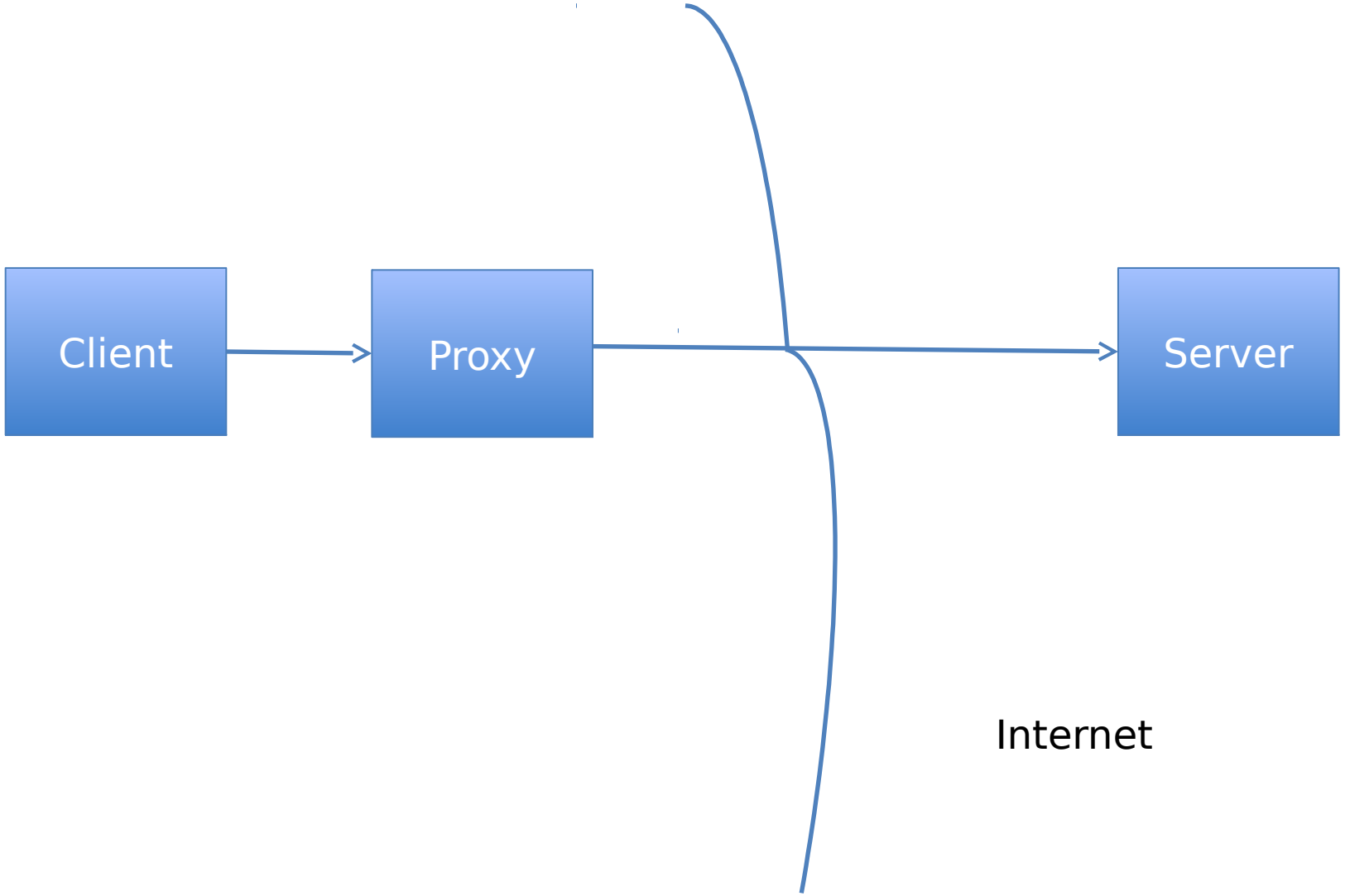
Protocol Operation



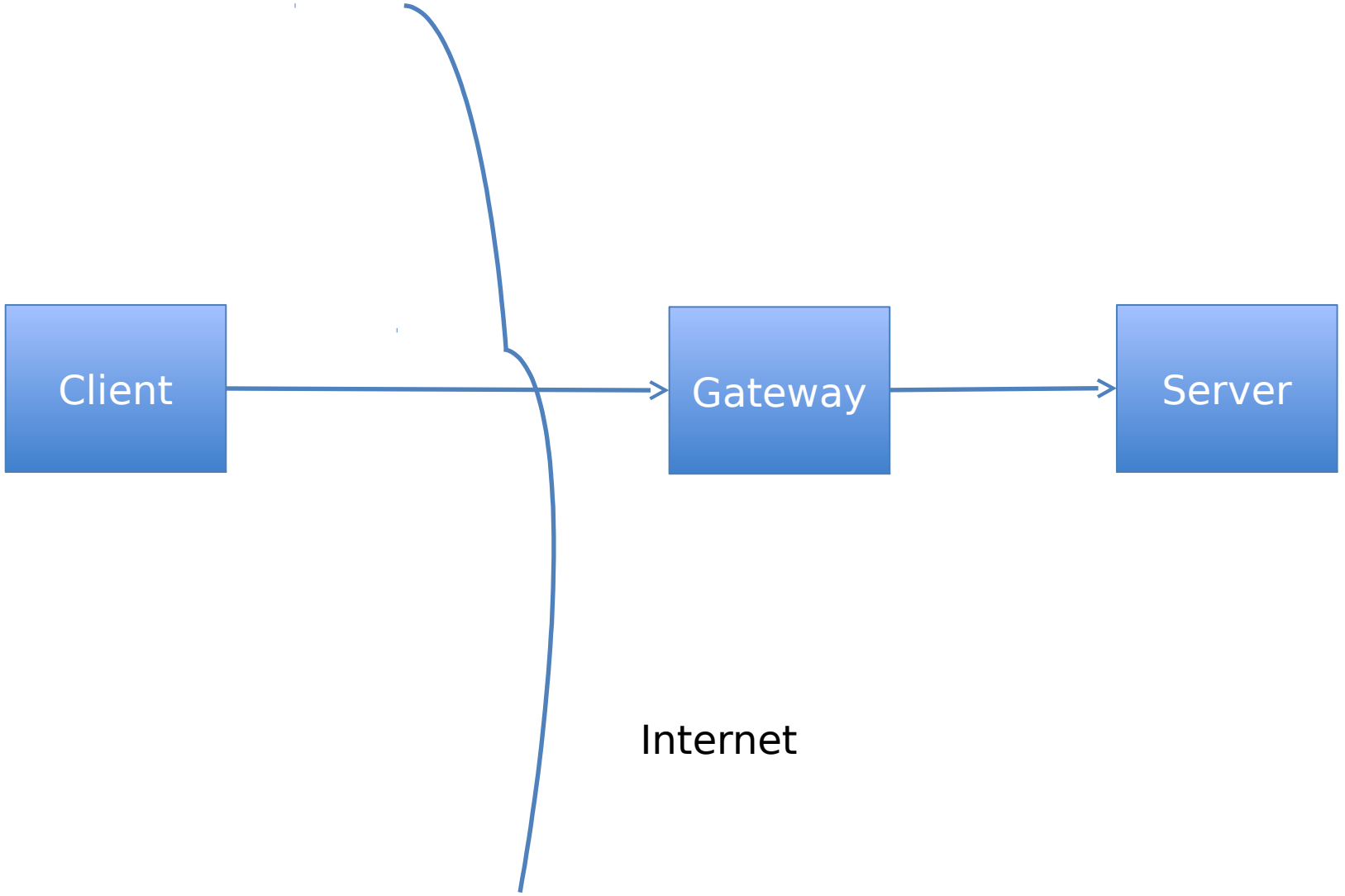
Intermediary: Proxy or Gateway

A Proxy or Gateway may cache responses

Proxy



Gateway / Reverse Proxy



Protocol communication

- HTTP needs reliable transport mechanism
  - TCP: Okay
  - SMTP: Okay
  - UDP: Not okay
- HTTP communication typically happens over port 80
- HTTPS (Secure HTTP)
  - Communication happens on port 443

HTTP Request

HTTP Request

A request consists of four components:

- Request-line
- Request Headers
- CRLF (Newline)
- Request Body

Request-Line

Method <space> URI <space> HTTP-Version CRLF

CRLF: Newline

E.g:
GET /~devdatta/index.html HTTP/1.1\n
Host: www.cs.utexas.edu

URI vs. URL

- URI: Uniform Resource Identifier
  - Identifier/Name for a resource
    - Could be path of the resource (such as file path)
- URL: Uniform Resource Locator
  - Network location of resource
    - Essentially, specifies how to access the resource

HTTP Request Headers

- Request headers allow the client to pass additional information about the request or the client itself to the server
- Some HTTP 1.0 Request Headers
  - Authorization
  - User-Agent
  - If-Modified-Since

Authorization Header

- A user agent can authenticate itself with the server by including the “Authorization” header

- HTTP supports
  - Basic Authentication
  - Digest Authentication

Basic Authentication

- “Basic” authentication scheme is based on the model that the user agent must authenticate itself with a user-ID and a password
- Client sends Base64 encoded string of userID:password
- Example:
  - Suppose username=‘Aladdin’ password=‘open sesame’
  - Client sends
    - Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
    - Example: java Base64EncodeDecode

Basic Authentication

- Not secure
  - username:password are ‘encoded’ not ‘encrypted’
- Works under the assumption that the transport layer between the client and the server is trusted

User-Agent Header

- The User-Agent request header contains information about the user agent originating the request

- Server can use this value for tailoring the response to be sent to the client
  - Sending images of different resolution to desktop browser vs. Android browser

HTTP Response

HTTP Response

A Response consists of four components:

- Status-Line
- Response headers
- CRLF (essentially a newline)
- Response Body

Response Status-Line

Status-Line =
- HTTP-Version <space> Status-Code <space> Reason-Phrase CRLF

- Status codes:
  - 3 digit integer
  - First digit defines the class of response
    - 1xx – Informational
    - 2xx – Success
    - 3xx – Redirection (further action must be taken)
    - 4xx – Client Error (bad syntax or cannot be fulfilled)
    - 5xx – Server Error

Example Status Codes

- http://eavesdrop.openstack.org/
- 200 – OK
  - java TestClientSocket eavesdrop.openstack.org / html
- 301 – Moved Permanently
  - java TestClientSocket eavesdrop.openstack.org /irclogs html
- 400 – Bad Request
  - java TestClientSocket eavesdrop.openstack.org index.html html
- 404 – Not found
  - java TestClientSocket eavesdrop.openstack.org /irclogs1 html
- 401 – Unauthorized
- 500 – Internal Server Error

Things that client care

- Timestamp
  - When the data was last modified
- Till when the data is valid per the timestamp
- Data format
- How large the response is?
- If the client has the authorization
  - Determines if the Authorization header needs to be included

HTTP 1.0 Response Headers

- Date
  - Indicates the date/time when the message originated
- Last-Modified
  - Date/time when the resource was last modified
    - For files, it may be just the file system last-modified time.
      - How to deal with a group of resources?
- Invariant:
  - *Date value is always greater than value of Last-Modified header*
- Content-Length
  - Indicates the size of the Entity-Body (response that contains entity)
  - The value is decimal
    - Example: java TestClientSocket www.cs.utexas.edu /~devdatta/test-file.txt txt
      - Show Content-Length
      - Show serveroutput.txt
- Content-Type
  - Indicates the media type of the Entity-Body

HTTP 1.1: Chunked transfer encoding

- Transfer Coding:
  - Encoding transformation that has been applied to an entity-body
  - Chunked transfer coding
    - transfer-coding: chunked
    - Chunked-body = *chunk
      last-chunk
      trailer
      CRLF

chunk = chunk-size CRLF chunk-data CRLF
chunk-size = string of HEX digits
last-chunk has chunk-size “0”

HTTP 1.1: Chunked transfer encoding

- Example:
  - Step 1: java TestURLConnection http://eavesdrop.openstack.org/irclogs >& output.txt
  - Step 2: Remove headers from output.txt
  - Step 3: java TestClientSocket eavesdrop.openstack.org /irclogs/ html >& output1.txt
  - Step 4: http://www.binaryhexconverter.com/hex-to-decimal-converter
  - Step 5: Check the results

HTTP Caching

HTTP Caching

- Goal
  - From client side:
    - Eliminate the need to send requests to reduce the number of ‘round-trips’ required for many operations
  - From server side:
    - Eliminate the need to send full responses to reduce the ‘network bandwidth’ requirements

HTTP Caching

- Both HTTP/1.0 and HTTP/1.1 support caching
- Caching is managed via request and response headers

Caching: Overall Operation

- User Agent (UA) requests a resource
- Server sends the resource along with some caching related headers
- An intermediate Cache/Proxy_server saves the resource and the headers
- Cache/Proxy_server applies a "caching algorithm" to:
  - Satisfy *subsequent requests* from the UA

HTTP 1.0: Caching Headers

- Expires (Response header)
  - Date/time after which the resource should be considered stale
  - Applications/Cache/Proxies must not cache the entity beyond the given date
  - If the value is equal to or earlier than the value of the ‘Date’ header then the entity should not be cached
- If the Expires header is absent then the response should not be cached
- If-Modified-Since (Request header)
  - Used with the GET method to make it conditional
    - If the requested resource has not been modified since the time specified in the field, a copy of the resource will not be returned from the server; instead a ‘304’ status will be returned

If-Modified-Since Header

- Which date to use in the If-Modified-Since header?
  - When the resource was first requested, server should have sent either the “Date” header or the “Last-Modified” header in the response
  - That date should be used with the “If-Modified-Since” header
  - If both are present, use the Last-Modified header

- Example:
  - Open http://www.openstack.org in
    - Chrome->View->Developer->Developer Tools
  - Show request headers for “clear.png”

HTTP 1.0: Caching Headers

- Pragma: no-cache (Request header)
  - An intermediary should forward the request toward the origin server even if it has a cached copy of what is being requested
  - Client's way to tell an intermediary to not serve the response from the cache

Problems with HTTP 1.0 Caching

- Too tied with absolute dates and times
  - May not work appropriately if the clocks on the server and client are not synchronized
- Not fine-grained enough
  - Cannot work if a common cache is present for multiple different clients
    - “Shared” cache

Problems with HTTP 1.0 Caching

- Too tied with absolute dates and times
  - May not work appropriately if the clocks on the server and client are not synchronized
- Not fine-grained enough
  - Cannot work if a common cache is present for multiple different clients
    - “Shared” cache

HTTP 1.1: Caching Headers

- Cache-control
- Entity-tag cache validators

HTTP 1.1: Cache-Control Headers

- The Cache-Control header allows a client or server to transmit a variety of directives in either requests or responses.
  - These directives typically override the default caching algorithms
  - Must be obeyed by all caching mechanisms along the request/response chain
- Cache-Control Headers:
  - max-age: Servers specify explicit expiration times using either the Expires header or the ‘max-age’ directive
    - If both are present then the ‘max-age’ directive takes the precedence
  - s-maxage: For “shared” cache the specified maximum age overrides max-age and Expires header
- Cache-Control: max-age=0
  - Client’s way to force a recheck with the origin server by an intermediate cache
  - This is same as the ‘Pragma: no-cache’ header of HTTP 1.0

HTTP 1.1: Entity Tag Cache Validators

- What is an Entity Tag?
  - Think of an Entity Tag as a “signature” of the resource
- Its purpose is to validate that the cached entity responses are good enough to be used
- Two kinds of validators
  - Strong validators and Weak validators
  - Strong validators
    - A server changes the validator for every change in entity
  - Weak validator
    - A server changes the validator only for semantically significant changes to an entity
- Example
  - Web application for support tickets; Changing ‘ticket name’ from lower case to mix case may not trigger this; but addition of new email address to ticket’s support account attribute will

- It is up to the server when to calculate the entity tag
  - Strong validator corresponds to calculating the entity tag / signature on every change of the resource
  - Weak validator corresponds to calculating the entity tag / signature on “major” changes to an entity

Entity Tag Cache Validators

- ETag (Response header):
  - The ETag response-header field provides the current value of the entity tag for the requested resource

- How can client use the value read from the ETag header?
  - Client sends the value back to the server and may want the server to perform an action only if:
    - The current value of ETag *does not match* one in the request header
    - The current value of Etag *does match* in the request header

- If-None-Match (Request header):
  - Perform the action only if the value specified in the header *does not match* that of the Etag of that resource on the server
    - Show example with GET on
      - http://www.openstack.org/themes/openstack/images/auto-banner.jpg
      - Use Chrome Developer Tool and Firefox RESTClient to see difference in behavior

Entity Tag Cache Validator

- If-None-Match (Request header):
  - Works in tandem with If-Modified-Since header
  - If the Entity tag has not been modified on the server, it may still perform the specified action if the resource’s modification date fails to match that specified in the If-Modified-Since header

Entity Tag Cache Validators

- If-Match (Request header):
  - Perform the action only if the value specified in the header *matches* that of the Etag of that resource on the server
  - Used as a ‘precondition’ mechanism for performing a resource modification action
  - Server returns ‘412’ precondition failed

Persistent Connections

Persistent Connections

- HTTP is a request response protocol
  - Client establishes the connection; server sends the response and closes the connection
  - For next request a new connection is opened
- Problem
  - Opening up a new socket connection for each request is a slow process
- Solution is to use “persistent connections”

- Both HTTP 1.0 and HTTP 1.1 support persistent connections
- HTTP 1.0
  - Persistent connections need to be explicitly negotiated
    - Connection: Keep-Alive
- HTTP 1.1
  - Persistent connections by default
    - Server will maintain a persistent connection with the client
      - Unless ‘Connection: close’ header is sent in the request
    - A single user client SHOULD NOT maintain more than 2 connections with a server

Persistent Connections

- Advantages:
  - Lower CPU and memory consumption on the client and the server
  - Lower latency to subsequent requests
- Disadvantages:
  - Keeping connection open may exhaust server socket resources
    - If appropriate timeouts are not used this can become a cause denial-of-service attack on the server

Summary

HTTP 1.0 vs. HTTP 1.1

- Caching
  - HTTP 1.0
    - Has limited number of cache control headers
      - Date, Last-Modified, If-Modified-Since, no-cache
  - HTTP 1.1
    - Additionally has
      - Etag, If-None-Match, If-Match, max-age
- Persistent connections
  - HTTP 1.0
    - Need to explicitly negotiated using “Connection: Keep-alive” header
  - HTTP 1.1
    - By default
- *Host* request header
  - HTTP 1.0 – not required
  - HTTP 1.1 - required

HTTP Request Examples

- HTTP 1.0

  GET /irclogs/ HTTP/1.0

- HTTP 1.1

  GET /irclogs/ HTTP/1.1

  Host: <host-name>

  Host header required for HTTP 1.1

Reference

- HTTP 1.0
  - http://www.rfc-base.org/txt/rfc-1945.txt
- HTTP 1.1
  - https://www.ietf.org/rfc/rfc2068.txt
  - https://www.ietf.org/rfc/rfc2616.txt
- Differences between HTTP 1.0 and HTTP 1.1
  - http://www8.org/w8-papers/5c-protocols/key/key.html

Additional Material

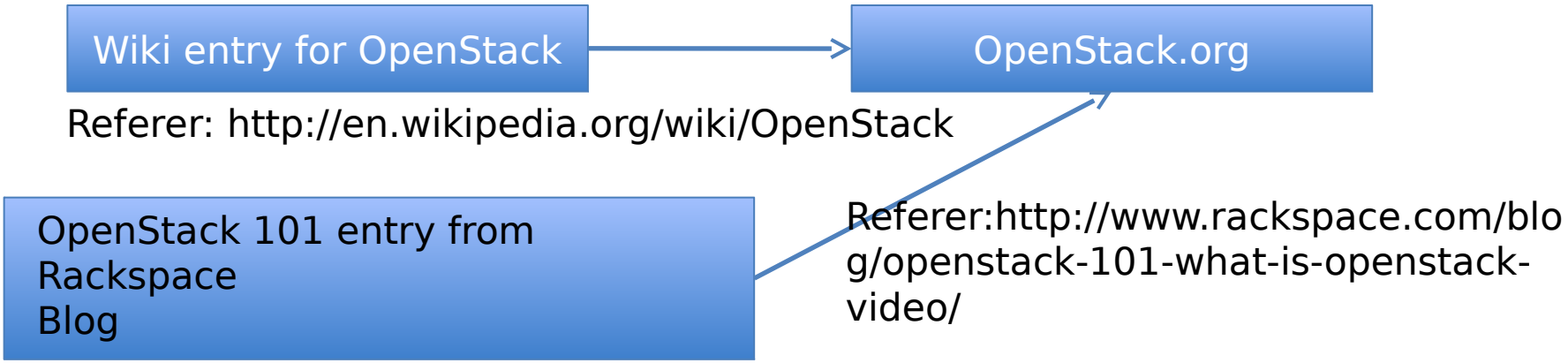
Authentication vs. Authorization

- Authentication:
  - Server is trying to answer the question, “Do I know this client?”

- Authorization:
  - Server is trying to answer the question, “Does this client have permission to perform this method on this resource?”
  - Authentication information can be used to perform authorization
    - If a client can correctly authenticate then it can access the specified resource

Referer Header

- Referer:
  - The address (URI) of the resource from which the Request-URI was obtained
    - Show example:
      - Open www.openstack.org and show in Chrome Developer tool
    - Useful for building the ‘back pointer’ graph



Servlets

Devdatta Kulkarni

What about server side?

- What concerns need to be handled on the HTTP server?
  - 3 minute exercise

Server side concerns

- Being available to accept connections
- Responding to legitimate requests in timely manner
- Handle concurrent client requests
- Handle large content
- Authentication and authorization
- Caching
  - Handling different kinds of headers

Server side concerns

- Track resources and permissions
- Correctly respond to HTTP methods
  - GET, HEAD, POST, PUT
- Correctly create the required response headers for each request
- Manage and maintain resources
- Request logging

Approaches

- Approach 1: Build as part of your application
  - Example:
    - java TestServerSocket
    - java TestHttPClientSocket localhost /
  - Each web application implements the server functionality
    - Lot of work
    - May get repetitive after implementing few applications

Approaches

- Approach 2:
  - Build a framework that handles common things:
    - Request handling
    - Logging
    - Authentication
  - Let the application handle:
    - Defining resources
    - Defining access control policies
    - Adding header values
  - *Servlets*

Servlets

- A servlet is a Java-based *Web component*, managed by a *container*, that generates *dynamic content*
  - Web component
    - Understands HTTP
  - Container
    - Software entity which manages Servlets through their life-cycle
  - Dynamic content
    - Content that depends on input data
- Specification:
  - http://download.oracle.com/otndocs/jcp/servlet-3_1-fr-eval-spec/index.html

Servlet Container

- Servlet container is a part of a Web server or application server that
  - contains and manages servlets through their lifecycle
  - provides network services over which requests and responses are sent

Servlet Container Functions

- *Must* implement HTTP/1.0 and HTTP/1.1
- *May* support additional request/response-based protocols such as HTTPS
- Container may perform caching subject to RFC 2616:
  - May respond to requests without delivering them to the servlet
    - E.g.: Send static resources from Servlet container cache
    - Tomcat has the “cachingAllowed” attribute which controls this
      - http://tomcat.apache.org/tomcat-7.0-doc/config/context.html
  - May modify requests from the clients before delivering them to the servlet
    - E.g.: Decoding URL encoded values in the request
      - Example: query-parameters
  - May modify responses from the servlets before sending them to the clients
    - E.g.: Adding the ‘Date’ header

Running Servlet examples

- Setting up Eclipse Neon
- Setting up Tomcat
  - Tomcat is a servlet container that we will use
- Creating a Servlet project
  - Dynamic Web project
    - Select “Target runtime”
      - Apache Tomcat v8.0
- hello-world
  - HelloServlet

Setting up Eclipse and Tomcat

- Eclipse Java EE IDE for Web Developers.
  - Version: Neon Release (4.6.0)
  - Build id: 20160613-1800
- Tomcat
  - Download Apache Tomcat 9.0
- Setup Eclipse to use Tomcat
  - Eclipse -> Preferences -> Server -> Runtime Environments -> Add
  - Check Page 35 of ``Java for Web Applications'' book

Creating Servlet Project

- Option 1: Create project from scratch
- Option 2: Import existing project

Creating project from scratch

- In Eclipse, Servlet-based projects are called ‘Dynamic Web Project’
  - File -> New -> Other -> Web -> Dynamic Web Project
    - Give ‘Project Name’ -> ‘Finish’
    - Select “Target runtime”
      - Apache Tomcat v9.0

- Dynamic Web Project structure

  /WebContent

  /META-INF

  /WEB-INF

- Setting up the Dynamic Web Project
  - Add web.xml to WEB-INF
    - See web.xml in ‘hello-world’ for example
    - Add HelloServlet class
      - Eclipse may add the class inside a package; update ‘servlet-class’ attribute in web.xml to include <packagename>.<classname>

Running Servlet

- Right click on the project
- Select ‘Run As’ -> ‘Run on Server’
- Choose the Tomcat v8.0 Server that we configured -> ‘Next’
- Servlet will be now accessible at:

  http://localhost:8080/<context_root>/<url-pattern>

  - Context Root
    - Right click on the project
    - Web Project settings
  - url-pattern
    - Specified in web.xml

Importing existing project

- File -> Import -> Maven -> Existing Maven Projects
- Browse to the directory where you have cloned the repo and select the hello-world project
- In Project Explorer / Package Explorer menu, right click the project and then Maven -> Update Project
- Once Maven has finished downloading the dependencies, right click on the project and choose Run As -> Run on Server
- Run on the configured Tomcat instance

- http://localhost:8080/<context-root>/<url-pattern>

Maven

- Tool that allows us to specify application's dependencies in a *declarative manner*

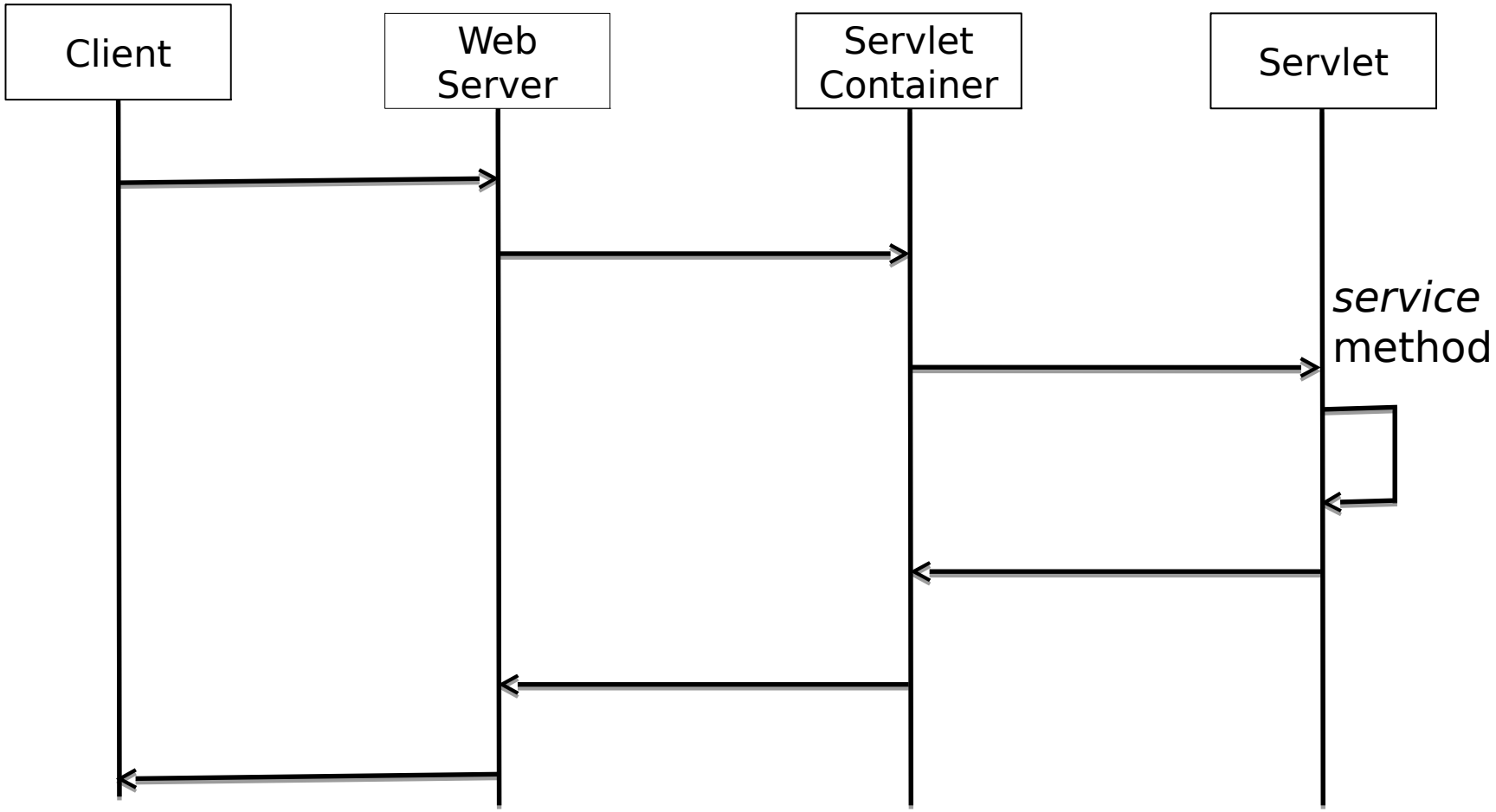
- Dependencies are specified in *pom.xml*

Context Root

- A context root identifies a web application in a Java EE server
  - Think of this as a pointer/reference that allows the web container to identify a web application

- http://docs.oracle.com/javaee/5/tutorial/doc/bnadx.html

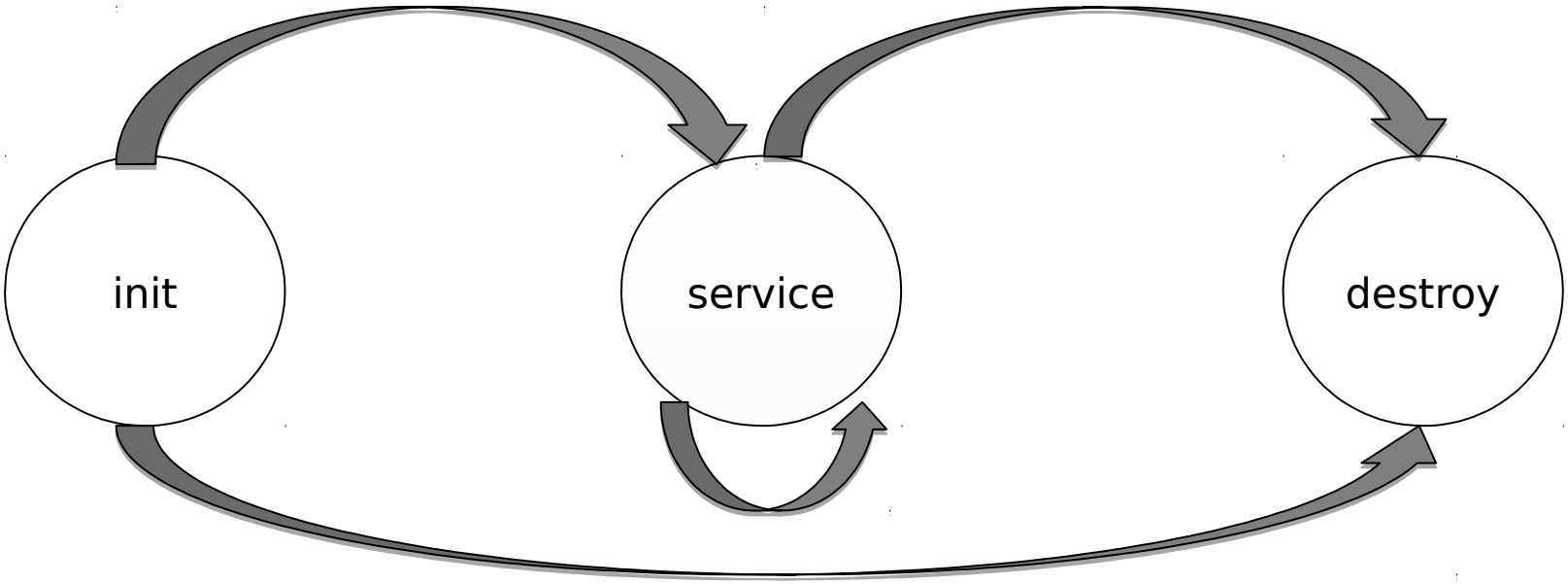
Interaction



Servlet Life-cycle

- Loading and Initialization
  - Servlet container is responsible for loading and instantiating servlets
    - When the container is started or when the servlet is needed
  - Servlet container calls servlet's 'init' method
  - Servlet container calls servlet's 'service' method
- Destroy
  - Servlet container calls servlet's 'destroy' method

Servlet Life-cycle



Web.xml

- How does the Servlet container know where to route incoming requests?
  - Solution: web.xml
- web.xml is the ‘Deployment descriptor’
- Application developer should use it to define
  - Servlet’s name
  - Servlet mapping
  - Servlet’s parameters
  - When to start a Servlet
    - <load-on-startup>
      - Example: Use test-load-on-startup

Servlet Interface

javax.servlet

```
public interface Servlet {
    void init(ServletConfig config)
    void service(ServletRequest req,   ServletResponse res)
    void destroy()
}
```

http://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html

HTTPServlet

public abstract class HttpServlet {

protected void doGet(HttpServletRequest req, HttpServletResponse res)

protected void doPost(HttpServletRequest req, HttpServletResponse res)

}

http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html

Number of Instances

- In a non-distributed environment
  - the servlet container uses *only one instance per servlet declaration*
- In a distributed environment
  - the servlet container creates *one instance per servlet* declaration *per Java Virtual Machine (JVM)*

Servlets and threading

- Servlet container creates a single instance of a Servlet
- Multiple threads can be active in a Servlet's 'service' method
- Servlet container *does not* synchronize access
- It is up to the application developer to synchronize the access

- Example:
  - Run test-thread-servlet
  - Run test-servlet.sh
    - ./test-servlet.sh

Reading

- Read chapters 1, 2, and 3 from the ``Java for Web Applications'' book

Servlets
Query Parameters, Sessions

Devdatta Kulkarni

Passing parameters to Servlets

Passing Parameters to Servlet

- Query Parameters
- Request Headers
- Request Body

Query Parameters

- Parameters that are passed with the resource URL as a query string
  - Query String: Starts with ‘?’
  - Parameters separated by: &

- Example: query-parameters

Parsing different parts of the resource

- getRequestURL
  - Returns the entire URL
- getRequestURI
  - Returns the context root
- getServletPath
  - Returns the url-pattern

- Example:
  - Use ‘query-parameters’

Request Headers

- We can set arbitrary headers
- Convention is to start the header with “X-”
  - http://stackoverflow.com/questions/3561381/custom-http-headers-naming-conventions

Request Body

- Parameters can be sent in as request body

- Example:
  - Use ‘query-parameters’ doPost
  - Use RESTClient Firefox add-on

When to use which type of parameter?

- Query parameters
  - When the parameters are concerned with the resource itself
    - Search queries that will provide subset of resources in response
- Request Headers
  - When the parameters are concerned with the request/response interaction
    - E.g.: Cookies
- Request Body
  - When the parameters are related to the resource's content
    - E.g.: Form submission

Sessions

What is a Session

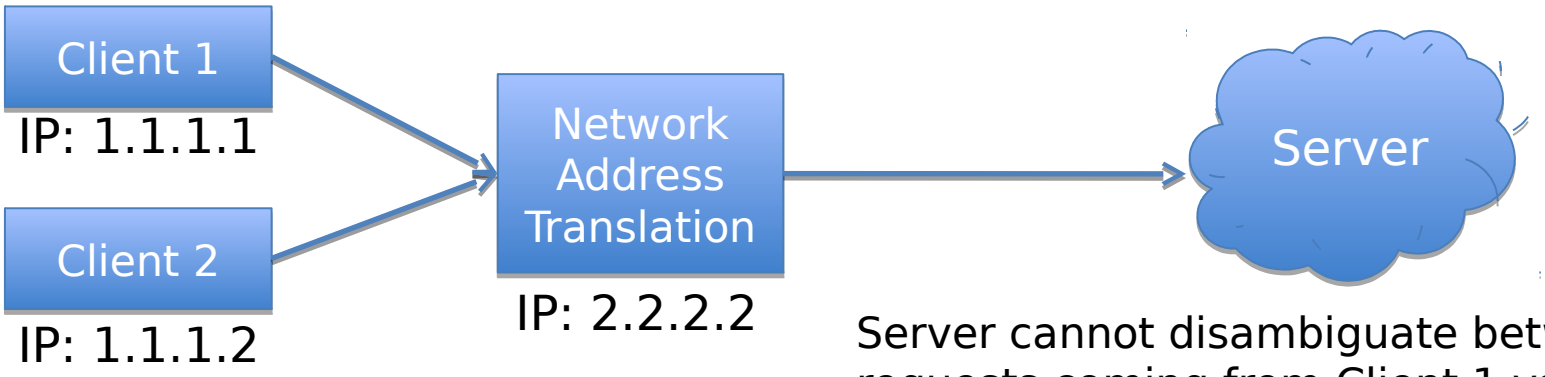
- Application-defined set of logical actions that have some application-specific semantic meaning
  - Example 1: Shopping cart
    - A session is actions between user adding something to a cart till the payment is confirmed on the payment screen
  - Example 2: Email systems
    - A session is actions between a user logging in and user logging out
      - All the activity during that time belongs to that session

Why are sessions needed?

- So that the server will know that *consecutive requests* are part of the *same user-level action*
  - Why is that needed?
    - To support high-level user actions such as doing online purchases
    - To customize and personalize displayed information
      - Recommendations on Netflix

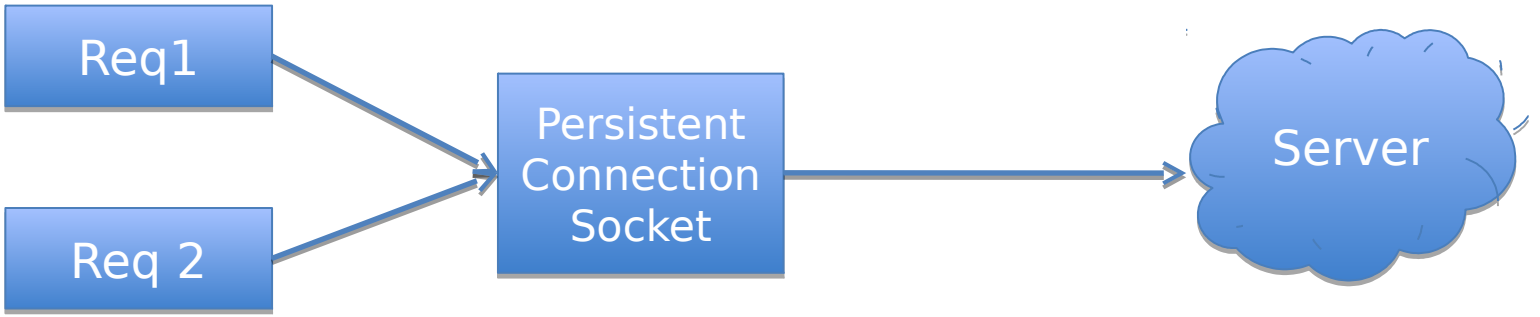
Approaches to track requests?

- Server needs something unique to identify a client
  - Could use Client IP + Browser combination
    - Problematic due to things such as Network Address Translation



Approaches to track requests?

- What about requests coming on a particular ``persistent connection''



Could be used; but having a persistent connection is not a necessary condition

Persistent Connections vs. Sessions

- Persistency is the property of the transport mechanism
  - A persistent connection is essentially a ``long-lived'' Socket
- A Session is the property of a group of HTTP requests
  - A request is either part of a Session or it is not
- Requests belonging to a Session *may or may not* use the same persistent connection

Approaches to track requests

- Use a *unique session_id* to identify a client
  - Server generates and sends it to the Client
  - Client then sends the *session_id* in each subsequent request
  - Server considers all the requests that include the *session_id* as belonging to one session

Session

- How to send *sessionid* to the client?
  - What mechanisms does the server have to send back data to the Client?
    - Approach 1: Response Headers
    - Approach 2: Resource URL itself
- Approach 1: By embedding it in a Response Header (Cookies)
  - Example
    - Use: session-example
- Approach 2: By embedding it within resource URL itself
  - URL Rewriting

Session Cookie

- Approach 1: By embedding it in a Response Header (Cookies)

- What is a Cookie?
  - Small piece of information sent by the server in a response header to the client for tracking purposes

Session Cookie

- Server uses *Set-Cookie* response header to send the *sessionid* to the client

- Client uses *Cookie* request header to send the *sessionid* in subsequent requests
  - RFC 2109
    - https://www.ietf.org/rfc/rfc2109.txt

Session Cookie

- Cookie attributes
  - Cookie name
  - Maximum age
  - Secure flag
    - The client will send the cookie only if the server is operating with a secure protocol like https
  - Http-only flag
    - The cookie will be accessible only to browsers and not to technologies such as JavaScript

Session Cookie

- Cookie name, cookie value:
  - ASCII character set
    - http://stackoverflow.com/questions/1969232/allowed-characters-in-cookies
  - Maximum age
    - Specified in seconds
    - Negative value means that the cookie won't be persistently stored

Session Cookie

- Cookie attributes
  - Domain name
    - Cookie set by parent domain is available to all its sub-domains
      - Example:
        - » Cookie set by cs.utexas.edu will be sent when requesting the page for www.cs.utexas.edu
        - » But it won't be sent for utexas.edu
  - Path
    - Resource path

Session: Server side

- How to send session_id to the Client
  - By embedding it within resource URL itself
    - URL Rewriting
  - By embedding it in a Response Header as a ``Cookie''
    - Use Set-Cookie Response header
      - RFC 2109
        - » https://www.ietf.org/rfc/rfc2109.txt
    - What is a ``Cookie''?
      - Small piece of information used for tracking purposes

Session: Client side

- How to send session_id to the Server
  - URL_Rewriting method used
    - Nothing to do; the session_id is available in the URL itself
  - Set-Cookie response header used
    - Send the session_id within the ``Cookie'' request header

Examples

- Cookie storage in Firefox
  - Show how to access cookies.sqlite
    - http://stackoverflow.com/questions/7610896/how-to-use-sqlite-to-read-data-from-the-firefox-cookies-file
- SQLite Database Browser
  - http://sourceforge.net/projects/sqlitebrowser/

Checking Cookies in Firefox

- Firefox version 35.0.1
  - Preferences -> Privacy -> remove individual cookies -> localhost

- Find the cookies file (on MacOS)
  - Finder -> CMD+Shift+G
    - ~/Library/Application Support/Firefox/Profiles
    - cookies.sqlite
    - sqlitebrowser

Cookie handling on User Agent

- Latest Http state management RFC
  - http://tools.ietf.org/html/rfc6265
- Cookie header handling algorithm (Section 5.4)
  - Cookies with longer paths are listed before cookies with shorter paths.
  - Among cookies that have equal-length path fields, cookies with earlier creation-times are listed before cookies with later creation-times.
  - Update the last-access-time of all cookies to the current date and time.

URL Rewriting

URL Rewriting

- Server will generate a unique session ID
- Within application code, we ``rewrite'' URL by ``encoding'' it with the generated session ID
  - Encoding step appends the session ID to the URL
    - JSESSIONID=<sessionID>
- When the encoded URL is clicked the request is considered to be part of that session

URL Rewriting

- Example:
  - Use url-rewriting:
  - Key steps:
    - HttpSession session = request.getSession(true);
    - String encodedURL = response.encodeURL(url);
  - request.getSession():
    - Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session
  - response.encodeURL():
    - Encodes the specified URL by including the session ID in it

URL Rewriting: Issues

- Unsafe
  - Session id visible within the URLs
- All the URLs that need to be considered as part of a particular session workflow need to be encoded
- Can we use it to control access to different url patterns?
  - Yes;
    - String url = request.getRequestURL().toString(); → is specific to a URL
- URL Rewriting is useful when browser does not support cookies

- How to expire the session when it is tracked using URL rewriting?
  - One approach: remove the session id from all the URLs

Cookies: Security Issues

- Copy and Paste Mistake
  - User sends the link with session ID embedded in it to others
    - Show using url-rewriting
  - How to avoid
    - Disable embedding sessionIDs in URLs
      - So no URL rewriting method for session tracking
- Session Fixation
  - Attacker sends the links to the victims with a session ID embedded in it
  - When users click on it, the attacker gets the control

Cookies: Security Issues

- Cross-Site Scripting and Session Hijacking
  - Attacker injects attack JavaScript into vulnerable site
  - Attack JavaScript copies the session ID cookie using ‘document.cookie’ property
    - How to prevent?
      - Set ``HttpOnly’’ attribute of the cookie
- Insecure Cookies
  - Man-in-the-middle attack
    - Attacker observing network traffic between Client and Server
    - How to prevent?
      - Use the ``Secure’’ attribute
        - » Indicates that the cookie should only be transferred over HTTPS
          - Cookie is transmitted as encrypted
      - Drawback:
        - » Site needs to be behind HTTPS

Readings

- Chapters 1, 2, 3, 5 of ``Java for Web Applications'' book

Deploying web apps to Tomcat

Devdatta Kulkarni

Approaches

- Manually deploy
- Deploy via Eclipse

Tomcat

- Web server
- Important directories
  - bin: contains scripts to start and shutdown tomcat
  - conf: contains configuration files
  - logs: contains logs (relevant file is catalina.out)
  - webapps: contains web applications deployed on tomcat
- Important files and variables
  - conf/server.xml has Connector for HTTP/1.1 which controls what port Tomcat receives HTTP/1.1 requests

Deploying a web app directly to Tomcat

- Assumptions:
  - Project is managed using maven (i.e. there is a pom.xml in the project's folder)
  - Maven is installed on the machine
- Go to project directory
- Run “mvn war:war”
  - This will generate a war file in the “target” directory
  - The default name of the war file has following pattern
    - <artificatId>-<version>, where artifiactId and version are as defined in the pom.xml
  - If you want a specific name for the war file, add the “finalName” attribute in the “build” section of pom.xml
- Context Root: This is essentially the name of the war file.
- Copy the war file to tomcat's webapps directory
- Start Tomcat by running bin/startup.sh
- Once Tomcat has started, it will expand the war file into a directory. You can observe this by running “ls -l” inside the webapps directory
- If there are no errors, your web application will be available at:
  - http://localhost:<connector port>/<context-root>/<url-pattern>
- Check the logs
  - more catalina.out, less catalina.out, tail -f catalina.out

Deploying web app through Eclipse

- Setup Eclipse to use Tomcat as the web server
  - Eclipse → Window → Preferences → Servers → Runtime Environment → Add
    - Provide a name to the “New Server Runtime environment”
    - On the next screen, provide path of Tomcat Installation Directory (the directory where you downloaded and unzipped Tomcat)
- Running a Servlet project
  - Right click on the project
  - Select ‘Run As’ -> ‘Run on Server’
  - Choose the Tomcat you configured
  - Eclipse will start Tomcat and deploy your web application / Servlet to it
  - You can change the port on which Tomcat runs by clicking “Servers” tab and double clicking the Server that you are using
- Servlet will be now accessible at:
  - http://localhost:8080/<context_root>/<url-pattern>
- Context Root
  - Right click on the project
  - Web Project settings
  - Note that the context-root is not affected by presence of pom.xml
- url-pattern
  - specified in web.xml

Servlet related details

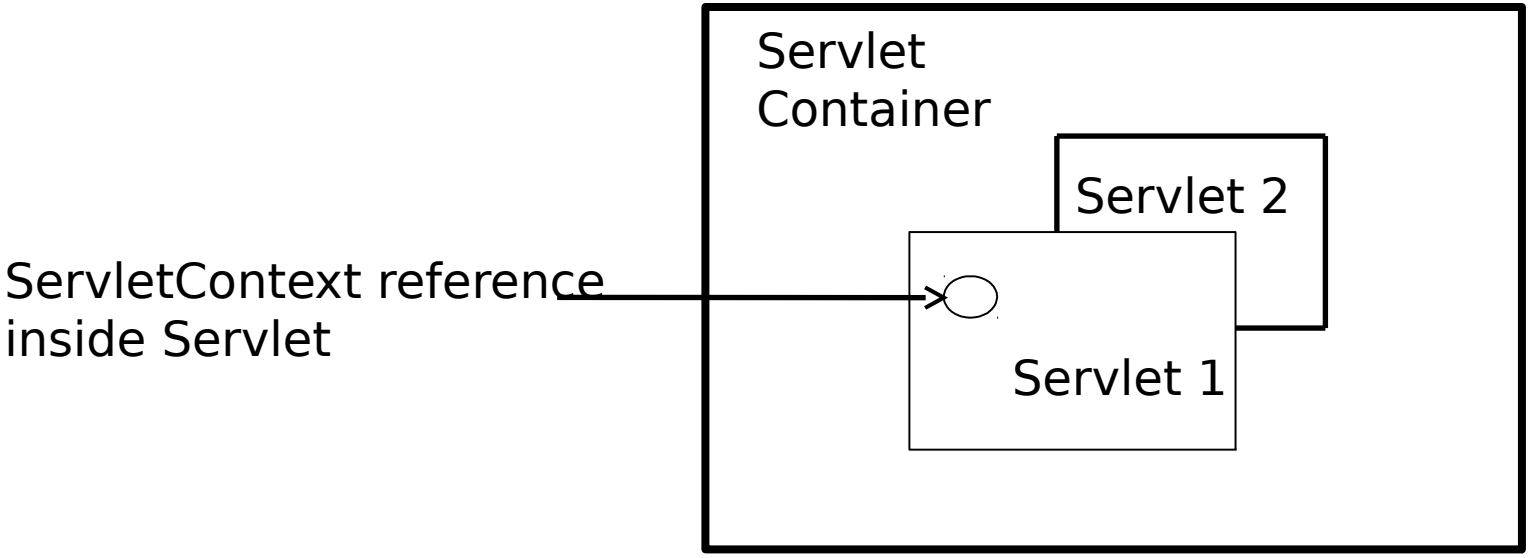
Devdatta Kulkarni

Topics

- Servlet Context
- Servlet Config
- Servlet Loading
- Spring’s ApplicationContext
- Spring’s WebApplicationContext
- Spring’s DispatcherServlet

ServletContext

- What is Servlet's Context?
  - The context within which the Servlet is running
    - The Servlet container!!!



ServletConfig

- Used by Servlet container to pass configuration information to a Servlet during initialization
- ServletContext can be obtained from ServletConfig
- Save it in the init method if required to be used inside doGet/doPost method

Servlet Loading: Load on startup

- Servlet can be configured to be loaded on:
  - Servlet container starts up
  - When the first request hits the Servlet

- Starting up on container start up makes the Servlet ready when requests arrive; delaying it till the first request causes the first request to incur the start up latency

Starting up when Servlet container starts

- <load-on-startup>
- The Servlet starts up in the logical order determined by the value of the ``load-on-startup'' tag
- If the values are same then the initialization happens in the order in which the Servlets are defined in web.xml
- Examples:
  - Use test-load-on-startup
    - Show use of different values
    - Show use of same value for different Servlets

Servlet Initialization Parameters

- Parameters that are passed to a Servlet at the Servlet initialization time
  - Servlet config params
    - These apply to only that Servlet for which they are defined
    - Available via *ServletConfig* object
      - Example: test-load-on-startup
        - » printServletConfigParams()
  - Context init params
    - These apply to all Servlets defined in a deployment descriptor
    - Available via *ServletContext* object
      - Example: test-load-on-startup
        - » printServletContextParams()

ServletConfig vs. ServletContext

- ServletConfig Servlet's Configuration
  - Servlet specific parameters

- ServletContext -> Servlet's *Context (a.k.a: ServletContainer)*
  - *ServletContext* object pertains to parameters for *all* the Servlets

- Possible to obtain the *ServletContext* object from the *ServletConfig* object

Spring ApplicationContext

- Central interface to provide configuration for an application

- Details:
  - Bean factory methods to access application components
  - Inherits from parent’s application context if one is defined

Spring's WebApplicationContext

- Interface to provide configuration of a web application
- Contains getServletContext() method to get the ServletContext

- How to get access to ApplicationContext within our Controller bean?
  - By implementing the 'ApplicationContextAware' interface

DispatcherServlet

- There can be more than one
  - One for UI-based web resources (forms)
  - One for REST API of your application
- Each DispatcherServlet has its own ApplicationContext
- All DispatcherServlets inherit the Root ApplicationContext via their respective ApplicationContexts

XML/HTML Parsing

Devdatta Kulkarni

What is XML? What is HTML?

- XML:
  - A “generic” markup language
  - Data is defined within matching tags/nodes

```
<cs378>
    <assignments>
        <assignment1>Servlets</assignment1>
    </assignments
</cs378>
```

  - XML document needs to be “well-formed” (every opening tag should have a closing tag)
- HTML:
  - A markup language for representing data/pages for browser display
  - Pre-defined set of tags (such as <html>, <title>, <head>, <a>, etc.)
  - HTML document *need not be well-formed*

Problem

- Given an XML/HTML document, output specific nodes from it that match a *Query* criteria

Example:

Input:

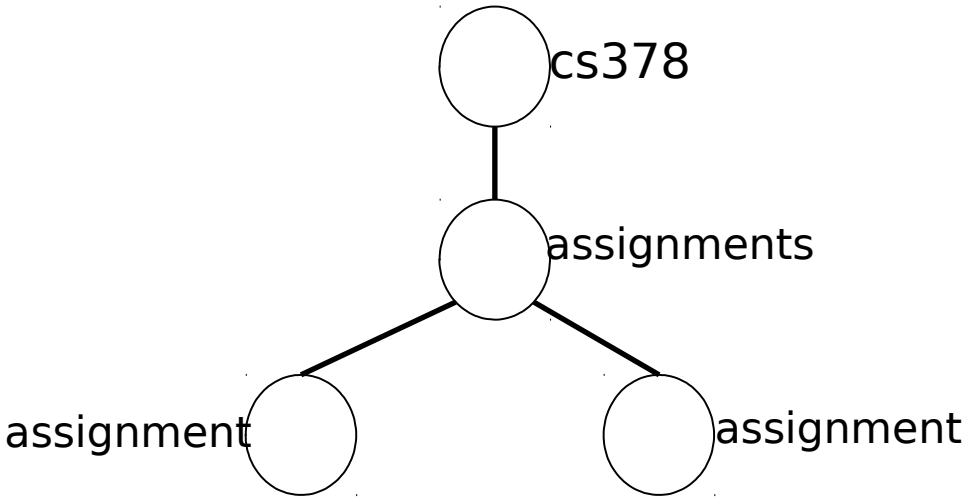
```
<cs378><assignments>
      <assignment>Caching Proxy</assignment>
      <assignment>Servlets</assignment>
</assignment></cs378>
```

Query Criteria: ‘assignment’ node

Output:

```
<assignment>Caching Proxy</assignment>
<assignment>Servlets</assignment>
```

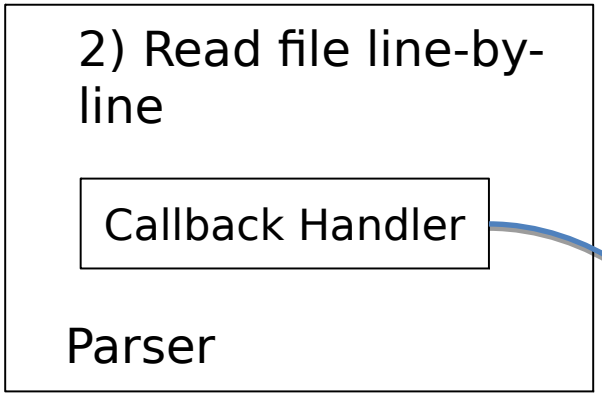
XML Parsing: Option 1: Tree parsing




```
List assignmentNodes = new
ArrayList();
Node head = cs378;
Queue.enqueue(head);
while(!Queue.isEmpty()) {
  Node n = Queue.dequeue();
  if (n.getNodeName().equals
    (“assignment”)) {
     assignmentNodes.add(n);
  } // end of if
 Queue.enqueue(n.getChildren());
}// end of while
return assignmentNodes;
```

This approach is called *DOM* parsing
DOM stands for Document Object Model (DOM)

XML Parsing: Option 2: Parsing using callbacks



1) Register callback handler with the Parser

3) Invoke Callback Handler

```
List assignmentNodes = new
ArrayList();

public nodeHandler(Node n) {
    if (n.getNodeName().equals
        (“assignment”)) {
        assignmentNodes.add(n);
  }
}
```

This approach is called *SAX* parsing
SAX stands for Simple API for XML

DOM vs SAX Comparison

- DOM
  - Advantage:
    - Provides fine-grained control over parsing
  - Disadvantage:
    - Entire tree is built in memory before parsing can begin
      - Memory intensive
- SAX
  - Advantage:
    - Does not build entire tree; so memory is not an issue
  - Disadvantage:
    - State between callback invocations needs to be maintained by the program

Parsing XML: Current way

- XPath
  - Declarative model for querying XML documents
    - ``Queries'' are specified using ``path expressions''
      - Example Query: /cs378/assignments/assignment
      - Read:
        - » http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/package-summary.html

- Example: XPathParser.java

Examples

- Examples
  - DOMParser
  - SAXParser
- What about parsing HTML documents?
  - Can we use XML parsing techniques?
    - Use DOMParser and SAXParser with cs378.html
    - Use DOMParser and SAXParser with cs378.not_well_formed.html

Parsing HTML

- Parsing using Java regular expressions
  - Example: RegexParser
- Parsing using a library such as jsoup
  - http://jsoup.org/
  - Example: JSoupParser

- Parsing HTML disadvantages:
  - Parsing presentation logic instead of working with the domain objects
  - Very brittle; will break if the HTML page is changed
  - No formal contract defined; so cannot validate the HTML document

Reading

- XML Parsing
  - http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/package-summary.html
  - http://docs.oracle.com/javase/tutorial/essential/regex/test_harness.html
  - http://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html
  - http://jsoup.org/

Spring Framework

Devdatta Kulkarni

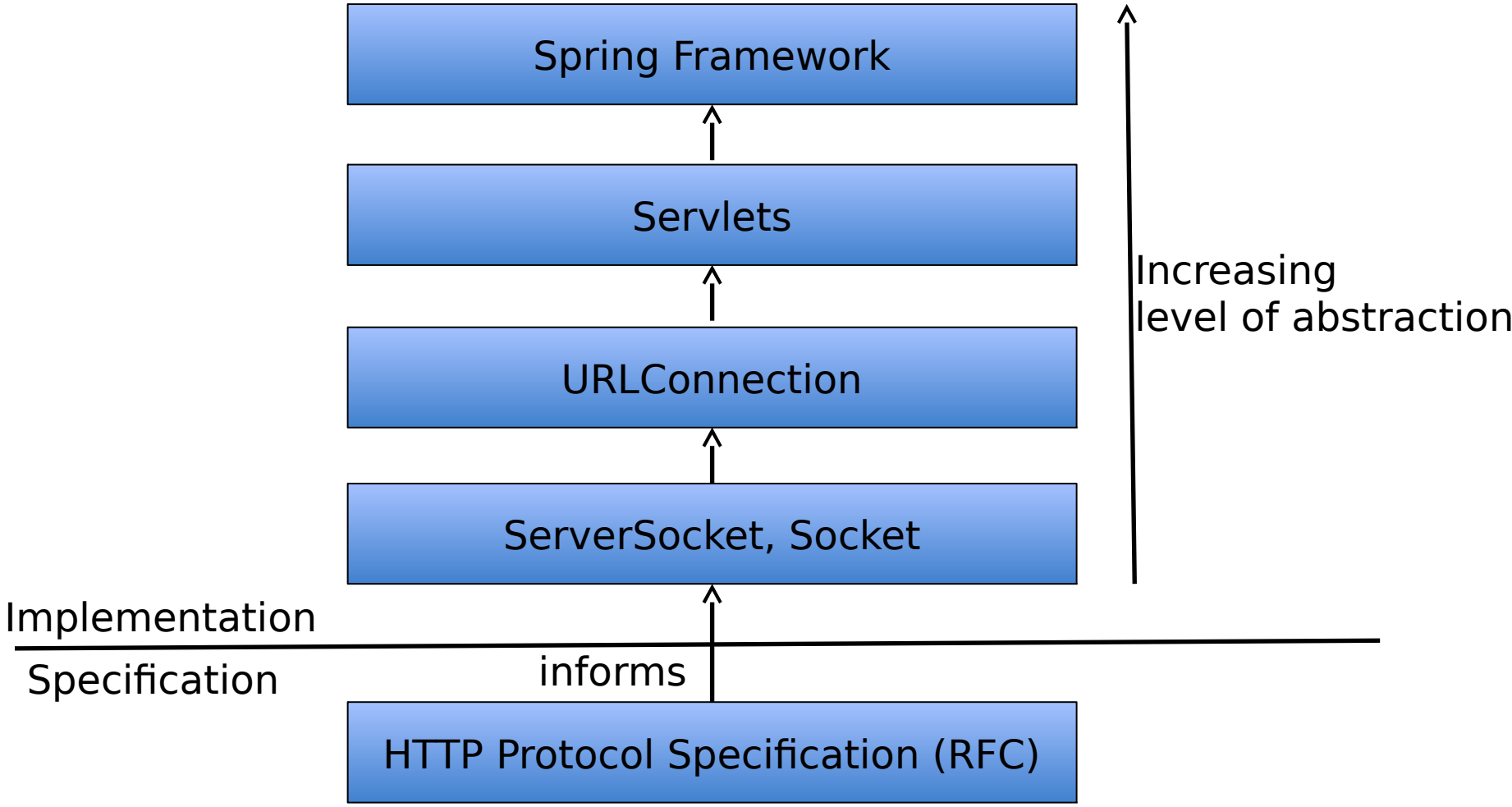
Issue with Servlets

- Interface is oriented towards underlying low-level concepts (e.g.: HTTP), rather than application-level concepts
  - *doGet*, *doPost*, instead, for example, *queryAustinSocialMedia*

Spring Framework:

- What is it?
  - Its an *application container* for Java that provides many useful features, such as
    - *Dependency Injection*
    - Transaction management
    - Support for developing aspect oriented programs
- Why is it useful?
  - One level more abstracted from Servlets
    - Request/Response are abstracted
  - Provides method-level mapping for resources
    - Logical code groupings
      - Map Servlet actions to methods

Levels of Abstraction



Dependency Injection

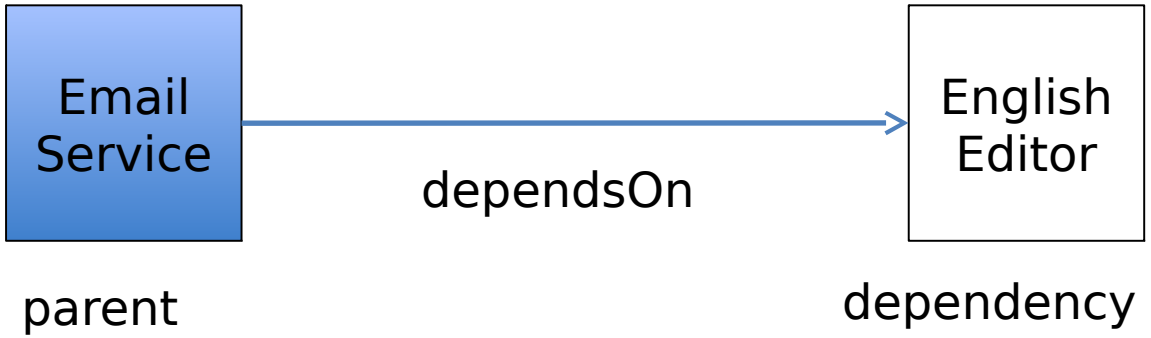
- What is it?
  - Separating creation of dependencies from their consumption
- Why dependency injection matters?
  - Allows flexibility in configuring a class
    - E.g.: Easy to provide Test class as a dependency for testing vs. real dependency
- How to achieve it?
  - Developing against an Interface instead of concrete class
  - Let the application container like Spring to construct the *parent -> dependency* object graph

Need of dependency injection

- To demonstrate this issues lets look at an another example
- Suppose we want to build a *Email Service* that supports composing emails in *English or Spanish*
- Suppose that the ability to compose emails in English is provided by an object of type *EnglishEditor*
- To compose emails in English the Email Service needs an instance of the *EnglishEditor* object

Email Service

- Build a Email Service that supports creation of emails in *English*



Reference: Example referenced from book *"Dependency Injection"* by Dhanji R. Prasanna

Email Service: Initial design

```
public class EmailService {
    private class EnglishEditor editor;

    public EmailService() {
        this.editor = EnglishEditor();
    }

    public void sendEmail() {
        this.editor.compose();
    }
}
```

Create the dependency in the constructor

Problems with this approach?

- Cannot easily modify EmailService to support emails in Spanish

- Cannot easily test EmailService's functionality
  - For instance, we would like to test that:
    - When *sendEmail* method is called
    - Verify that the *compose* method of the editor is called

Addressing concrete binding issue

- Develop against an *Interface* instead of using *concrete* implementation classes
  - Instead of using *EnglishEditor* within EmailService, define and use *Editor* interface
    - Interface vs. implementation distinction example
      - Map is an interface
      - HashMap, Hashtable, TreeMap, LinkedHashMap are implementations of the Map interface
- How is this useful?
  - The Editor class is free from any one particular implementation

Addressing testability issue

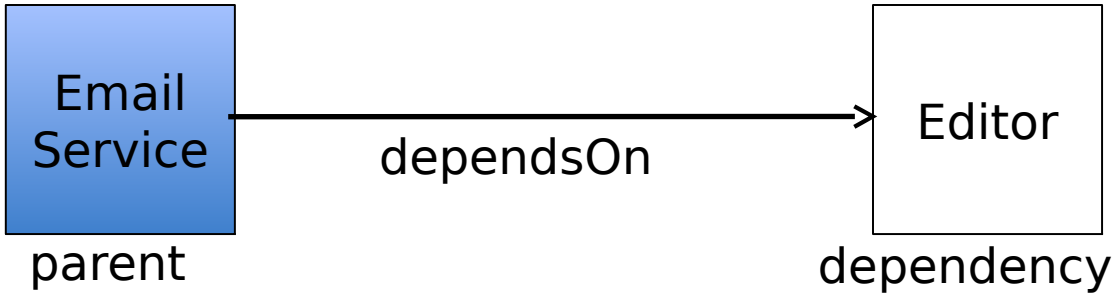
- *Externalize* dependency creation
  - Don’t create the dependency in the constructor, but create it elsewhere and pass it to the parent
  - Verify that required methods were called on the dependency

Testing EmailService

```
public class EmailService {
    private class Editor editor;

public EmailService(Editor editor) {
        this.editor = editor;
 }

    public void sendEmail() {
        this.editor.compose();
    }
}
```

```
public class TestEmailService {
    private EmailService emailService;

  public void testEditorCompose() {
     Editor editor = mock(Editor.class);
     emailService = new
EmailService(editor)
     emailService.sendEmail();
     verify(editor.called(compose));
  }
}
```

Dependency Injection



EmailService *depends* on Editor

Spring Framework performs dependency injection by:

- creating a Editor object and passing it to the EmailService

Dependency Injection is also called *Inversion of Control (IoC)*

Why Inversion of Control?
Because instead of the parent object creating and instantiating the dependency, control of
creating the dependency is given to the container.

Container creates the dependency and injects it into the parent object

Spring Framework Basics

- Bootstrapping and configuration
- Dependency Injection
- Java Beans
- Dispatcher Servlet
- Annotations

Bootstrapping and Configuration

- Bootstrapping refers to starting up of the Spring framework
- Requires an instance of the *DispatcherServlet* given configuration file in the form of a *contextConfigLocation* init parameter and instructed to load on startup
- Example:
  - spring-email-service

contextConfigLocation: servletContext.xml

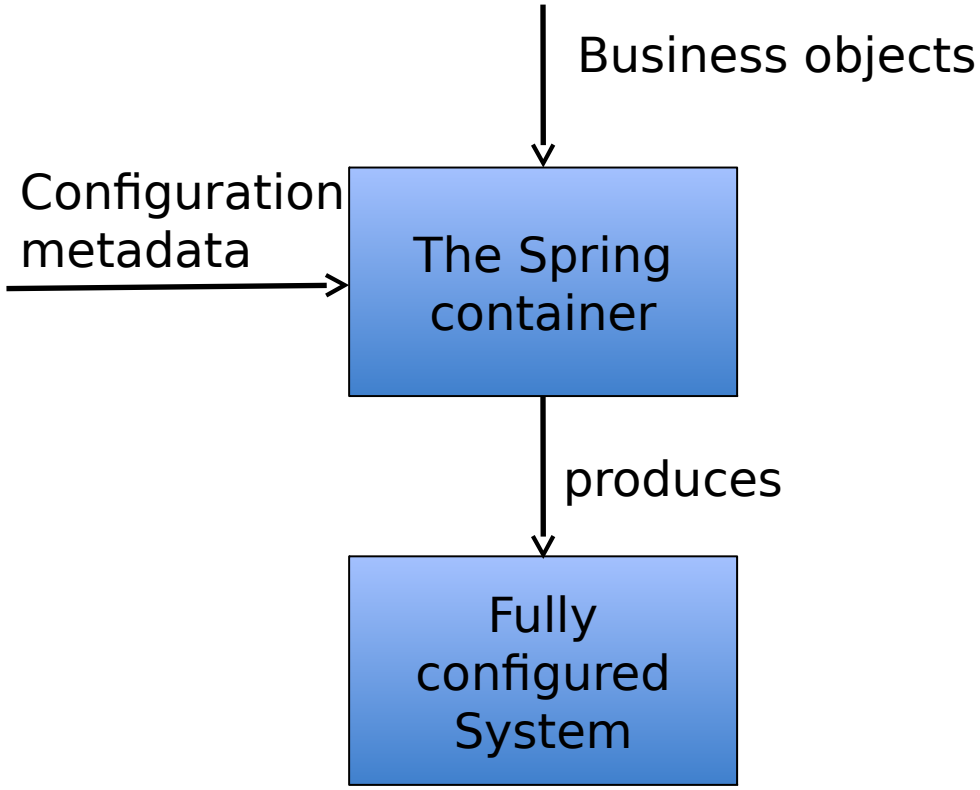
- Defines the configuration metadata about your application in XML
- Configuration metadata allows you to express the objects that compose your application and the rich interdependencies between such objects
- The objects that compose your application are called *beans*

contextConfigLocation: servletContext.xml

- A bean is an object that is instantiated, assembled, and managed by a Spring IoC container
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading this configuration metadata

http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-introduction

Spring IoC



http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-introduction

Java Beans details

- Java Beans:
  - What is a Java Bean?
    - Java class that is built based on following convention:
      - Properties are accessible through getter/setter methods
        - » get/set + property_name_with_first_letter_capital
  - EmailController
    - Setter method for a property

Dispatcher Servlet

- Central dispatcher for HTTP request handlers/controllers
- Dispatches to registered handlers for processing a web request
- http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html

Spring Example Details

- web.xml
  - Servlet name: springDispatcher
  - Init-param: contextConfigLocation
  - load-on-startup
- servletContext.xml
  - Beans:
    - emailController
    - editorServiceImpl

Dependency Injection

- Setter injection
  - setter methods need to be defined for each dependent variable
    - Example: EmailController
- Constructor injection
  - constructor parameters passed in for each dependent variable

Setter Injection

- In bean configuration file:

```
<bean name="emailController" class="EmailController”>
    <property name="editorService"
ref="editorServiceImpl" />
</bean>
```

- In code:

```
public void setEditorService(EditorService editorService)
{
     this.editorService = editorService;
}
```

Constructor Injection

- In bean configuration file:

```
<bean name="emailController" class="EmailController”>
  <constructor-arg
ref="englishEditorServiceImpl"></constructor-arg>
</bean>
```

Typical Issues in getting Spring working

- Class not found: DispatcherServlet
- Resolution:
  - Add Maven Dependencies to “Deployment Assembly”
    - In Eclipse,
      - Right click project -> Build Path -> Configure Build Path -> Deployment Assembly -> Add -> Java Build Path Entries -> Maven Dependencies
  - Add “build” and “WEB-INF/lib” folders to Java Build Path
    - In Eclipse,
      - Right click project -> Build Path -> Configure Build Path -> Java Build Path -> Source -> Add folder

Reading

- Chapter 12 from Java for Web Applications book

References

http://www.martinfowler.com/articles/injection.html

Application Context

- Represented by org.springframework.context.ApplicationContext interface
  - It represents the Spring container itself and is responsible for instantiating, configuring, and assembling the Java classes that make up your application
- A Spring application always has at least one application context

Unit Testing

Devdatta Kulkarni

Unit Testing

- How to ensure that a ``unit of code'' is working as expected?
- In your assignment1, how did you know that you had accounted for all the specified and non-specified combinations of the query parameters?

Unit Testing

- Concepts
  - Code under test
  - Dependencies
- Question that we ask?
  - When the dependencies behave as specified, does the code under test behave as we expect it to behave?

Unit Testing: Example 1

- What does testing the ‘doGet’ method mean?
  - It means verifying that:
    a) When the method is called with instances of HttpServletRequest and HttpServletResponse objects as input parameters, the HttpServletResponse’s Printwriter’s println() method is passed the “Hello world.” string
    b) HttpServletResponse’s getWriter() method is called

Unit Testing: Example 2

- What does testing the ‘doGet2’ method mean?
  - It means verifying that:

  a) When “username” query parameter is passed in, the response is of the form “Hello <name>”

  b) When “username” is null or empty

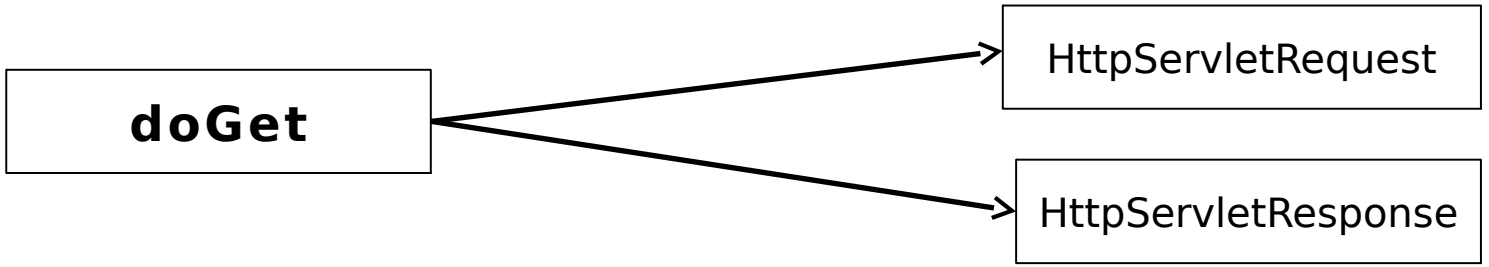
  These are two separate test cases

Unit Testing: Stages

1) Decompose your application into small units
2) Control the values that are generated by any dependencies
   - This is called setting up ``expectations'' for the code under test
3) Invoke the code under test
4) Verify that the result is what you are expecting

Unit Testing: Expectation setting and verification

- So, we need to set expectations on the dependencies
  - How to do it?



How do we make the HttpServletRequest object return the values that we control?

Enter
Mocks

Mocks

- Mock
  - A ``mock'' is an object that can be used for testing purposes in place of the real dependency object
  - A mock object provides mechanisms to:
    - set the values that would be returned when a method is called on the object
    - verify that a method was called on an object
    - verify that a method was called given number of times on an object

Unit Testing Library

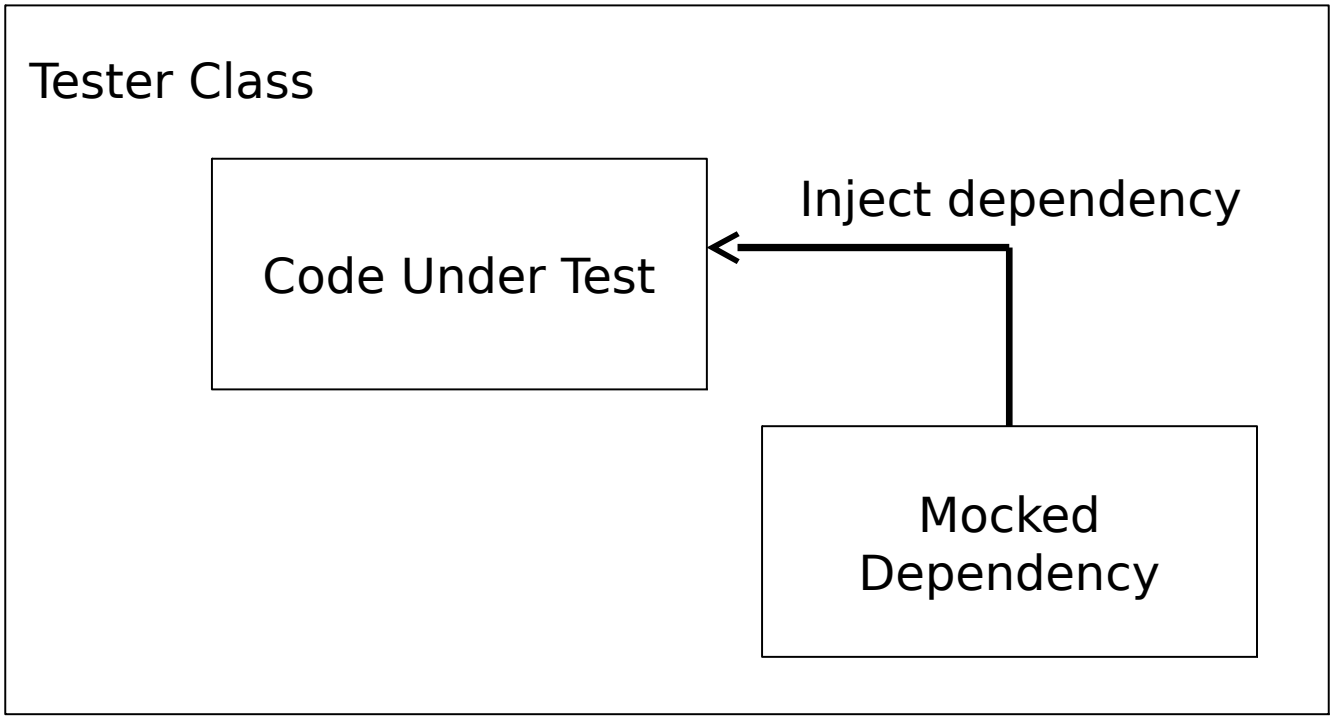
- Junit (http://junit.org/)

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
</dependency>
```

- Mockito (https://code.google.com/p/mockito/)

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
<version>1.9.5</version>
</dependency>
```

Unit Testing



Unit testing examples

- https://github.com/devdattakulkarni/ModernWebApps.git
  - Servlets/servlet-unit-test
  - Servlets/mockito-example
- Running tests:
  - Right click on src/test/java in the "Project Explorer" view
  - Select "Run As" -> Junit test

Code organization for Unit testing

- Typical code structure:

  src/main/java/*.java

  src/test/java/*.java

- Example:

  src/main/java/HelloServlet.java

  src/test/java/TestHelloServlet.java

JUnit Annotations

- @Before
  - This annotation identifies the method that runs *before* running any test method
    - Conventionally this method is named as *setUp*
- @Test
  - This annotation identifies a test method.

Unit testing steps

- Step 1: Identify *code under test*
- Step 2: Identify dependencies in code under test
- Step 3: Create mock dependencies
- Step 4: Set up expectations
- Step 5: Invoke *code under test*
- Step 6: Assertions and verifications
  - Assert state
  - Verify behavior

Step 1: Identify code under test

Let us test the following method in doGet method in HelloServlet:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.getWriter().println("Hello world.");
}
```

Step 2: Identify dependencies in code under test

- What are the dependencies of *doGet()* method?

- HttpServletRequest and HttpServletResponse objects

Step 3: Create mock dependencies

HttpServletRequest request = *mock(HttpServletRequest.class);*

Structure:

<Type> mockedObject = *mock(*Type.class)

Step 4: Set up expectations

PrintWriter writer = *mock(PrintWriter.class);*

when(response.getWriter()).thenReturn(writer);

Structure:

*when(*mockedObject.method()).*thenReturn(*value under our control*)*

Step 5: Call code under test

```
helloServlet.doGet(request, response);
```

Step 6: Verify interaction

verify(writer).println("Hello world.");
verify(response).getWriter();

Structure:
*verify(*mockedObject).methodName();

Step 6: Verify state

Pattern:

assertEquals(expected, actual)

Code under test and dependent objects

How to add dependency objects in the code under test?

- If we create dependencies on-the-fly, we won't be able to mock them and control their behavior
- But we need temporary objects all the time, which we will end up creating at runtime (on-the-fly)

So then what to do?

Code under test and dependent objects

Thumb rule:

- Don’t create application-level domain objects such as controller classes, service classes, and so on as part of a method’s execution
- Create them outside of any method
  - Create them inside constructor, or init method (for Servlets), or let the *container create them and inject into your class*

Points to remember

- Mockito does not allow mocking of *Final* classes
  - Cannot Mock URL class
- Cannot call private methods from the test class
  - Need to make methods either public or protected

References

- Static imports:
  - https://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html
  - http://monkeyisland.pl/2008/04/26/asking-and-telling/
- Types of test objects:
  - http://www.javaworld.com/article/2074508/core-java/mocks-and-stubs---understanding-test-doubles-with-mockito.html

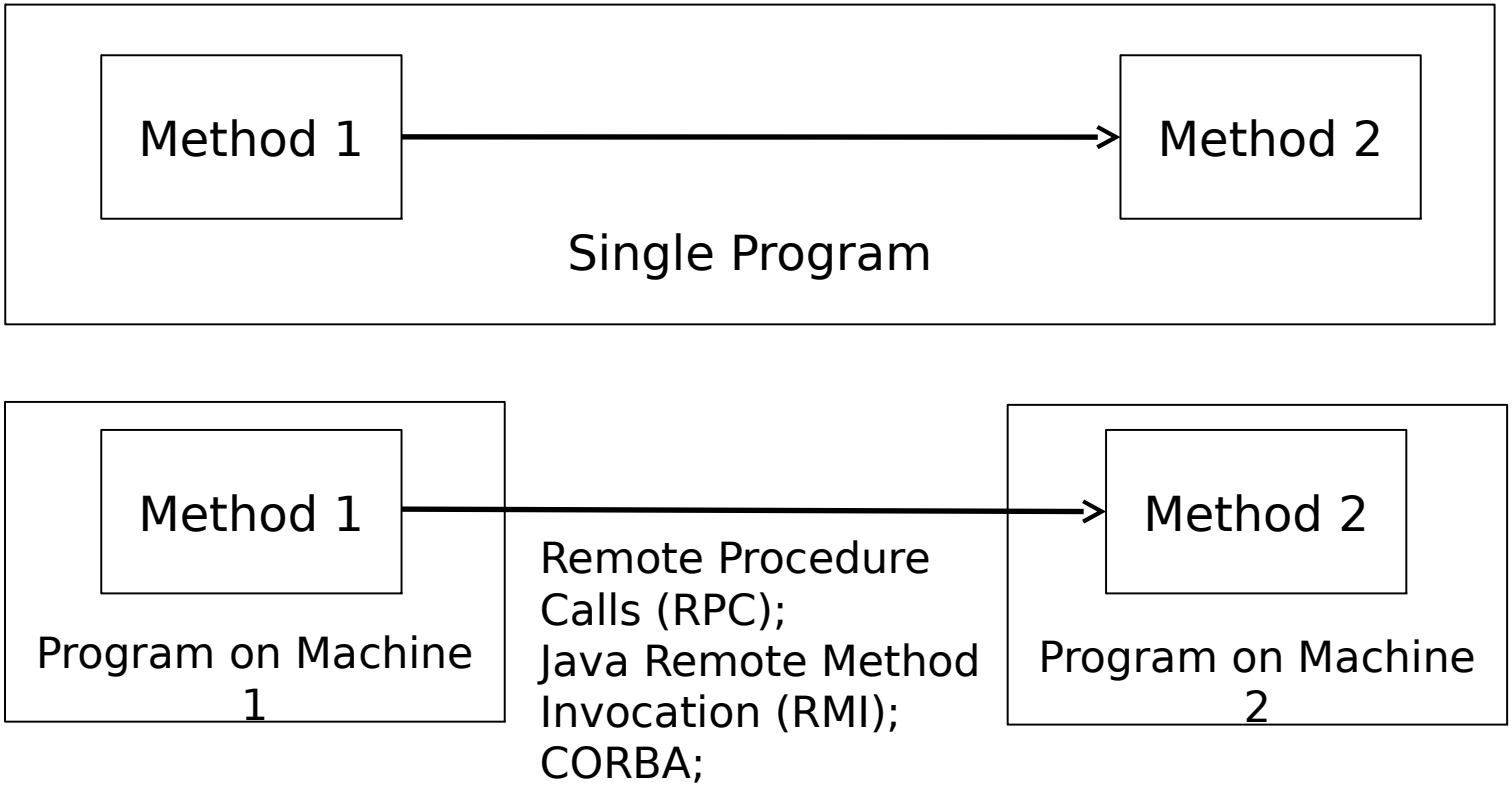
REST

Devdatta Kulkarni

REST

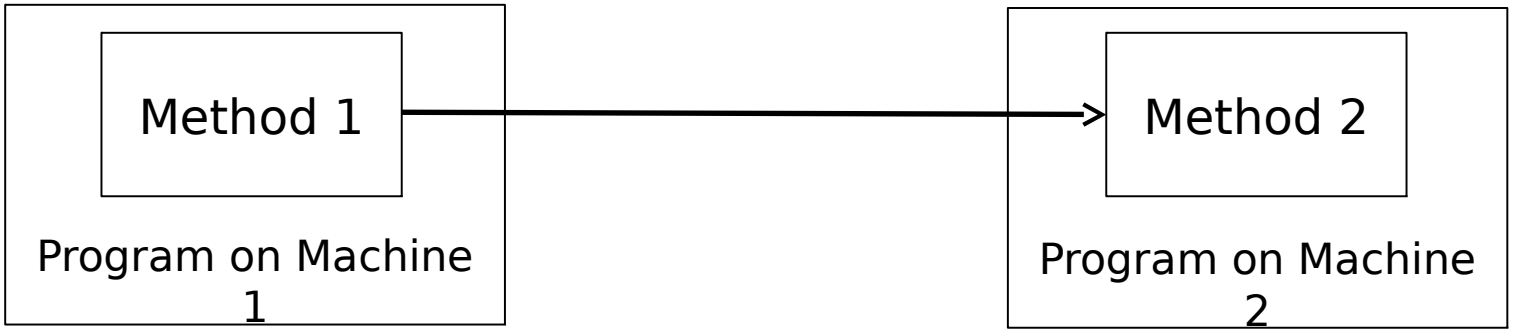
- What is REST?
  - Representational state transfer
  - A methodology for managing and manipulating web resources through their state representations
    - Web resources
      - Resources that are accessible over HTTP; typically available on the web server
    - State representations
      - Values for a resources' attributes
    - Managing and Manipulation
      - Change the resource through its state representation
    - Methodology
      - HTTP methods to achieve above goal
        - » GET, PUT, POST, HEAD, PATCH

Emergence of REST



A Classic Paper:
Implementing remote procedure calls
http://research.cs.wisc.edu/areas/os/Qual/papers/rpc.pdf
Citations: 2455

Emergence of REST



Java RMI; CORBA over WEB (SOAP)

REST:

- Keep transport mechanism HTTP (i.e. keep the WEB part of SOAP)
- Define a “uniform” interface instead of arbitrary method signatures

SOAP: Simple Object Access Protocol

REST example

- Obtaining data from the eavesdrop site:
  - We resorted to opening connection and parsing the HTML using Jsoup
  - But we would not have had to do that if the eavesdrop site had provided a REST API

REST Architectural Principles

- Addressability
- Uniform Constrained Interface
- Representation Oriented
- Communicate Statelessly
- Hypermedia As The Engine of Application State (HATEOS)

Addressability

- Every object and resource in your system is reachable through a *unique identifier*

- Example:
  - Resource: *Course 378*
    - /departments/cs/courses/cs378
  - Resource: *Assignments of CS 378*
    - /../cs/courses/cs378/assignments

Uniform Constrained Interface

- Use finite set of actions
- For HTTP these are:
  - GET
  - POST
  - PUT
  - DELETE
  - PATCH

Why Uniform Constrained Interface?

- Familiarity
  - Each resource is accessible with only the familiar set of actions
    - No need to look up method signatures
      - Great advantage over protocol such as SOAP
- Interoperability
  - HTTP is universal; Language libraries available for HTTP; no need for special client libraries

REST Action semantics

- POST (Also called as ‘create’ (C))
  - Create a new resource
- GET (Also called as ‘read’ (R))
  - Get information about a specific resource
- PUT (Also called as ‘update’ (U))
  - Update an existing resource
- DELETE (D)
  - Delete an existing resource
- Above actions are also called as CRUD operations

REST Action semantics

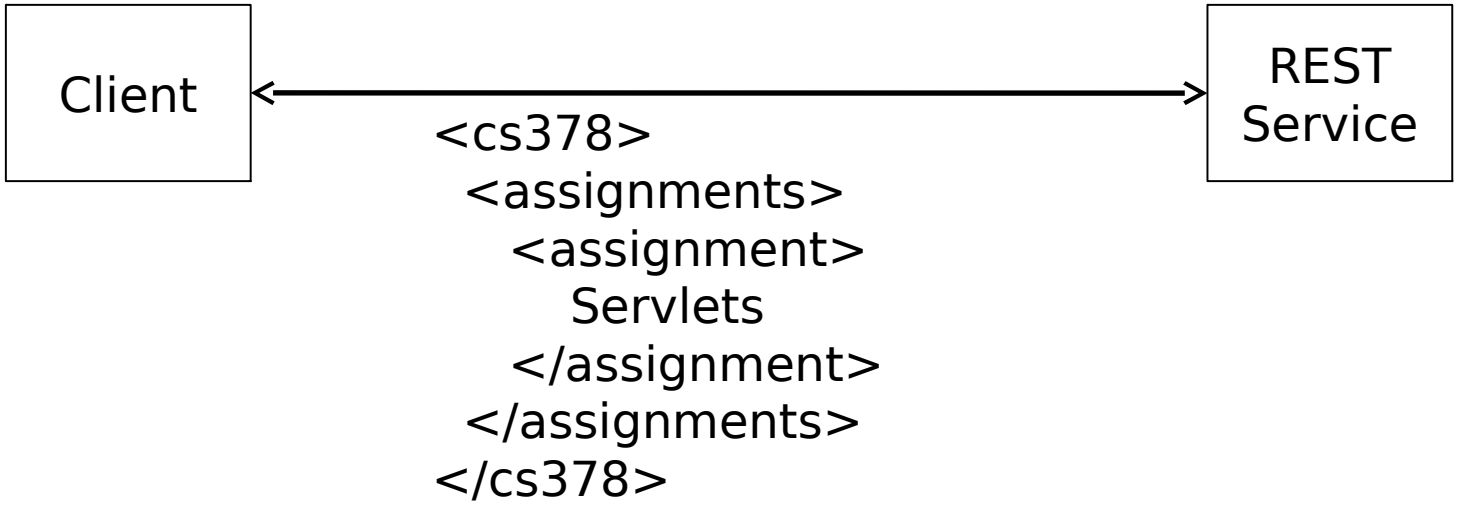
- Idempotent actions
  - Side-effect of N > 0 identical requests on the web application/REST service is the same as for a single request.
  - GET, HEAD, PUT, DELETE are idempotent
    - http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
- GET – Getting a resource multiple times does not change the state of the web app/REST service
- PUT – Executing the PUT operation with the same data multiple times does not change the state of that resource
- DELETE – Deleting a resource once is same as deleting it multiple times (responses will be different for first and other deletes but the state of web app will be same in both cases)
- What about POST?

REST Action semantics (cont.)

- Idempotent sequences
  - A sequence is idempotent if a single execution of the entire sequence always yields a result that is not changed by a re-execution of all, or part, of that sequence.
    - Idempotent sequence: <GET, GET, GET>, <PUT>, <DELETE>, <DELETE, DELETE>
    - Non-idempotent sequences: <PUT, PUT> (where data is different for each PUT action)
- Are action semantics dependent on returned status?
  - After initial DELETE, subsequent DELETE invocations will result in 404 (Not Found)
  - Does the returned status code determine the semantics of the action?
    - No. Action semantics are only dependent on the state of web application system and not on the returned status

Representation Oriented

- REST service is addressable through a specific URI and representations are exchanged between the client and service



Stateless communication

- No client session data is stored on server
- Server only records and manages the state of the resources it exposes
- If any session data is required, it has to be maintained by the client and sent with each request
  - We already know techniques for this
    - Cookies
    - URLRewriting
    - Request body parameters

HATEOAS

- Hypermedia As The Engine Of Application State (HATEOAS)
  - Discoverability
    - One resource points to other resources

```
<cs378>
  <assignments>
    <assignment>
      <name>Caching Proxy Server</name
      <link>cachingproxy</link>
    </assignment>
  </assignments>
</cs378>
```

How to design REST APIs?

- How to go about designing a REST API?

- A recipe

Recipe for designing REST APIs

- From the problem statement:
  - Step 1: Identify potential *resources* in the system
  - Step 2: Map *actions* to the identified resources
  - Step 3: Identify actions that cannot be mapped to resources
  - Step 4: Refine resource definitions by adding new resources in the resource model if required
  - Step 5: Introduce resource representations
  - Step 6: Re-map actions by establishing resource hierarchies
  - Step 7: Revisit above steps if required

REST API for UT course listings

- Queries
  - List all departments
  - List all courses within a department
  - Find information about a particular course, such as the instructor, pre-requisites, meeting time

Designing REST API

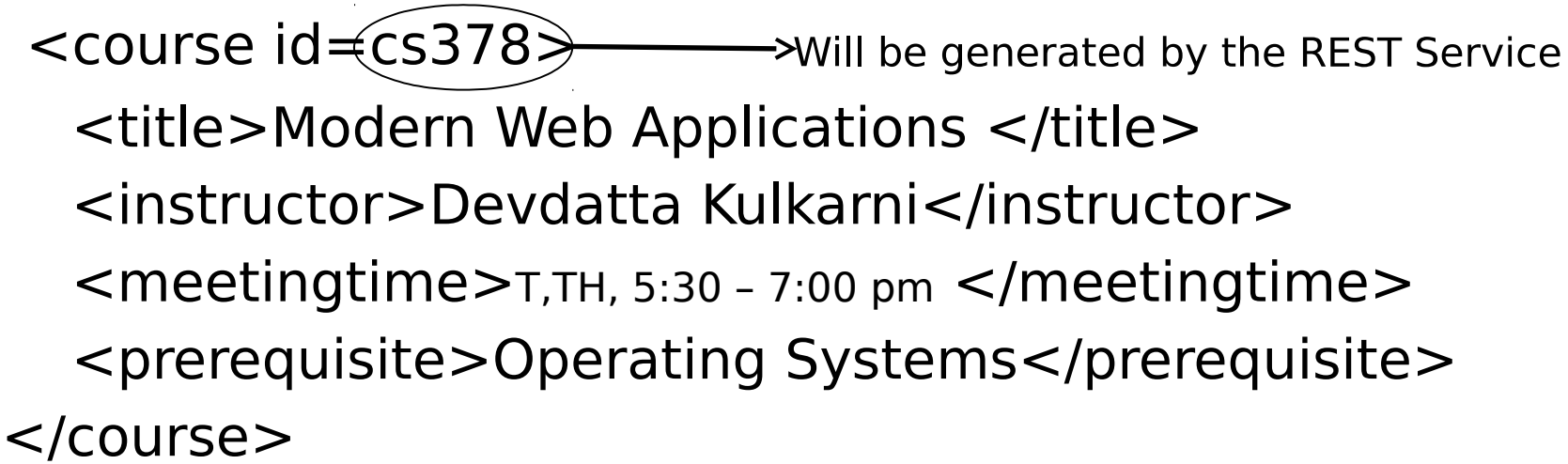
- Step 1: Identify the *resources*
  - Hint: Look for nouns
    - course, department, instructor, prerequisite, meeting time
- Steps 2 and 3: Map actions to resources
  - Action that can be mapped to above resources
    - Find information about a course
      - GET /course/{course_id}
        - {course_id} is a placeholder parameter
  - Action that cannot be mapped
    - List all courses within a department
    - Why?
      - Above identified resources are useful to get information about a specific resource
      - No resource available which would be appropriate to be used in a GET call to obtain listing of *all* courses

Designing REST API

- Step 4: Refine resource model
  - Introduce following resources
    - *courses* (plural form of course), *departments* (plural form of department)
- What about following entities?
  - instructor, pre-requisites, meeting time
  - Should we model these as separate resources or should we model them as *attributes* of the course resource?
  - Depends on how we intend to use them
    - If we are planning to support CRUD operations on them then they should be modeled as a separate resource
    - If not, we can keep them as attributes on the *course* resource

Designing REST API

- Step 5: Introduce resource representations



Designing REST API

- Step 6: Re-map actions to resources
  - List all departments
    - GET /departments
  - List all courses within a department
    - GET /departments/{department_id}/courses
      - E.g.: GET /departments/cs/courses
  - Find information about a particular course
    - GET /departments/{department_id}/courses/{course_id}
      - E.g.: GET /departments/cs/courses/cs378

REST API to REST Service

- What is a REST Service?
  - A service implemented to support REST-based access to resources maintained by a web application

Java REST Specification

- The Java API for RESTful Web Services (JAX-RS)
  - https://jsr311.java.net/nonav/releases/1.1/spec/spec.html
- JAX-RS implementations
  - Jersey (https://jersey.java.net/)
    - Reference implementation by SUN
  - RESTEasy (http://resteasy.jboss.org/)
    - From Redhat Jboss
      - Bill Burke, author of RESTful Java book is the project lead
  - Restlet (http://restlet.com/)
  - Spring's in-built support

JAX-RS Concepts

- Resources
- Resource methods
- Annotations
  - @Path
  - @Consumes
  - @Produces
  - :
  - :
- Application Class

Resources

- Using JAX-RS, a Web resource is implemented as a *resource class* and requests are handled by *resource methods*
- Resource class
  - It is a Java class that uses *JAX-RS annotations* to implement a corresponding Web resource
  - A resource class needs to have at least one method annotated with @Path or a request method designator (@GET, @POST, etc.)

Resource Life-cycle

- By default a new *resource class instance* is created for each request to the resource
  - Per-request model
- Life-cycle
  - First the constructor is called
  - Then, any requested dependencies are injected
  - Then, the appropriate method is invoked
  - Finally, the object is made available for garbage collection

Resource Constructors

- Root resource classes are instantiated by the JAX-RS runtime and MUST have a public constructor for which JAX-RS runtime can provide all parameter values
  - Zero argument constructor is permissible under this rule
- Non-root resource classes are instantiated by application and do not require such a public constructor
- A public constructor MAY include parameters annotated with one of the following:
  - @Context, @Header-Param, @CookieParam, @MatrixParam, @QueryParam, @PathParam
    - However, per-request information may not make sense in a constructor

Resource Methods

- These are methods of a resource class annotated with a *request method designator*
- Examples of request method designators defined by JAX-RS
  - @GET, @POST, @PUT, @DELETE, @HEAD
- Visibility
  - Only *public* methods may be exposed as resource methods

Resource Methods

- Parameters
  - Annotated parameters
    - Method parameters can be annotated to allow injecting of values from the request
    - It is possible to specify a *default value* for a parameter
  - Non-annotated parameters
    - These are called *entity parameters*
    - The value of an entity parameter is mapped from the request entity body
  - Resource methods can have *at most one* non-annotated parameter

Return Type

- Resource methods may return following types:
  - void
    - Results in empty entity body with a 204 status code
  - Response
    - Entity body of the result is mapped from *entity* property of the Response
    - Status code of the result is mapped from *status* property of the Response
  - A Java type
    - Entity body of the result is mapped from the Java class
  - GenericEntity
    - Represents a response entity of a generic type

Declaring Media Capabilities

- Application classes can declare supported request and response media types using the @Consumes and @Produces annotations

- These annotations can be applied to a resource method or a resource class

Application Class

- This class tells our application server which JAX-RS components we want to register with the JAX-RS runtime

- In our code we need to implement a class that extends:

  javax.ws.rs.core.Application class

Application Class

- Has two methods
  - public Set<Class<?>> getClasses()
    - Get a set of root resource *classes*
    - Default life-cycle for resource class instances is ‘per-request’
  - public Set<Object> getSingletons()
    - Get a set of *objects* that are *singletons* within our application
      - These objects are shared across different requests
- Difference between getClasses() and getSingletons()
  - http://stackoverflow.com/questions/18254555/jax-rs-getclasses-vs-getsingletons

JAX-RS Applications

- A JAX-RS application is packaged as a Servlet in a .war file
  - The *Application* subclass and resource classes are packaged in WEB-INF/classes
  - The required libraries are packaged in WEB-INF/lib
  - When using a JAX-RS aware servlet container, the *servlet-class* element of web.xml should name the *application-supplied* subclass of Application class

JAX-RS Applications

- When using a non-JAX-RS aware servlet container, the *servlet-class* element of web.xml should name the *JAX-RS implementation-supplied Servlet class*.

- The application-supplied subclass of Application is identified using an *context-param* with a param name of *javax.ws.rs.Application*

Examples

- RESTful Java with JAX-RS 2.0
  - Example code available at:
    - https://github.com/oreillymedia/restful_java_jax-rs_2_0
    - Once you clone it and unzip the file, the code examples are available in:
      - /examples/oreilly-jaxrs-2.0-workbook/

Running Examples

- Run from command line using Maven
  - Check out chapters 17 and 18 of the “RESTful Java” book
  - Note: Chapter 18 has good discussion about “pom.xml”

Additional Example

- https://github.com/devdattakulkarni/ModernWebApps/tree/master/REST

- Setting up and running the project in Eclipse.

  1) Import project as follows:
  - In "Package Explorer" view: - Import -> Existing Projects into Workspace

  2) Build: - Project -> Build Project

  3) Run on Server

  4) Try out following urls:
  - http://localhost:8080/assignment4/ut/courses
  - http://localhost:8080/assignment4/ut/helloworld

Running RESTEasy examples in Eclipse

- Precautions to take:
  - Check that the "Java Build Path" for your project has Maven Dependencies set in the Deployment Assembly.
  - How to check (or set it)?
  - Right click on project in the Package Explorer view
  - Choose *Build Path*
  - Choose *Configure Build Path*
  - Choose *Deployment Assembly*
  - Choose *Add*
  - Choose *Java Build Path Entries*
  - Choose *Maven Dependencies*

References

- http://www.infoq.com/articles/springmvc_jsx-rs
- http://stackoverflow.com/questions/80799/jax-rs-frameworks
- http://karanbalkar.com/2013/09/tutorial-54-getting-started-with-resteasy-using-eclipse/
- http://docs.jboss.org/resteasy/docs/3.0.1.Final/userguide/html/Maven_and_RESTEasy.html

References

- http://www.javacodegeeks.com/2011/01/restful-web-services-with-resteasy-jax.html
- http://docs.jboss.org/resteasy/docs/3.0.5.Final/userguide/html_single/#javax.ws.rs.core.Application
- http://howtodoinjava.com/2013/07/30/jaxb-exmaple-marshalling-and-unmarshalling-list-or-set-of-objects/
- https://access.redhat.com/solutions/55793

Annotations

Spring vs. JAX-RS

- Spring Annotations
  - @RequestMapping, @Controller, @Component, @PathVariable, @RequestHeader, @RequestBody, @ResponseBody, etc.
- JAX-RS Annotations
  - @Path, @PathParam, @MatrixParam, @HeaderParam, @FormParam, @CookieParam, etc.

Spring vs. JAX-RS

- Why different annotations?
  - Different use cases
    - Spring started with supporting ‘user-in-the-loop’ model-view-controller web applications
    - JAX-RS focuses on REST API based web services
- Spring Annotations are tied to spring framework
  - Lock-in to Spring
- JAX-RS Annotations are part of a specification
  - In theory, application code is portable across different JAX-RS implementations

Parameters

Scope of Path Parameters

- If a named URI path parameter is repeated by different @Path expressions, the @PathParam annotation will always reference the final path parameter

```
@Path(“/customers/{id}”)
    public class CustomerResource {
        @Path(“/address/{id}”)
        @GET
        public String getAddress(@PathParam(“id”) String addressId)
    }
```

If we do *GET /customers/123/address/456*, the addressId is bound to 456

@MatrixParam

```
@Path(“/{make}”)
public class CarResource {
    @GET
    @Path(“/{model}/{year}”)
    @public Jpeg getPicture
        (@PathParam(“make”) String make,
         @PathParam(“model”) String model,
         @MatrixParam(“color”) String color)
}
```

GET /mercedes/e55;color=black/2006

@MatrixParam

- Example
  - Ex05_1 (O'reilly book)
    - Change Junit version in pom.xml from 4.1 to 4.4

@FormParam

```
@POST
public void createCustomer
   (@FormParam(“firstname”) String first,
    @FormParam(“lastname”)
   ) {
}
```

Example: ex05_02

Programmatic URI Information

```
public interface UriInfo {
   public String getPath();
   public List<PathSegment> getPathSegments();
   public MultivaluedMap<String, String> getPathParameters();
   :
}
@Path(“/cars/{make}”)
public class CarResource {
    @GET
    @Path(“/{model}/{year}”)
    public Jpeg getPicture(@Context UriInfo info)
}
```

Spring Beans vs @BeanParam

- Injecting Spring Beans and RESTEasy
  - http://docs.jboss.org/resteasy/docs/1.1.GA/userguide/html/RESTEasy_Spring_Integration.html
  - http://www.mkyong.com/webservices/jax-rs/resteasy-spring-integration-example/

@BeanParam

```
public class CustomerInput {
    @FormParam(“first”)
    String firstName;

    @FormParam(“last”)
    String lastName;

    @HeaderParam(“Content-Type”)
    String contentType;
}

@POST
public void createCustomer(@BeanParam CustomerInput newCust)
```

@BeanParam

- The JAX-RS runtime will introspect the @BeanParam parameter’s type for injection annotations and then set them as appropriate

- Using Beans is a great way to aggregate information instead of having a long list of method parameters

Type Conversion

Type Conversion

- How to convert from String representation within an HTTP request into a specific Java type?

- JAX-RS can convert String data into any Java type, provided that it matches one of the following criteria:
  - It is a primitive type (int, short, float, double, byte, char, and boolean)
  - It is a Java class that has a constructor with a single String parameter
  - It is a Java class that has a static method named *valueOf()* that takes a single String argument and returns an instance of the class
  - It is a java.util.List<T>, java.util.Set<T>, java.util.SortedSet<T>

Custom parameter conversion

- Custom conversion of HTTP parameters to Java objects
  - JAX-RS provides two interfaces
    - *ParamConverter*
    - *ParamConverterProvider*

```
public interface ParamConverter<T> {
    public T fromString(String value)
    public String toString(T value)
}
```

Custom parameter conversion

```
public interface ParamConverterProvider {
    public <T> ParamConverter<T> getConverter(
        Class<T> rawType,
        Type genericType,
        Annotation annotations[]
    )
}
```

The class that implements this interface needs to be registered with your *Application* deployment class

Content Handling

- Unmarshalling
- Marshalling

- Unmarshalling
  - Converting from the over-the-wire representation of data into an in-memory representation
- Marshalling
  - Converting from the in-memory representation to a representation that is suitable for over-the-wire transmission

JAXB

- JAXB is an annotation framework that maps Java classes to XML and XML schema

- Example: ex06_1

Custom Marshalling

- JAX-RS allows writing your own handlers for performing marshalling/unmarshalling actions

```
public interface MessageBodyWriter<T> {
    boolean isWriteable()
    long getSize()
    void writeTo()
}

public interface MessageBodyReader<T> {
    boolean isReadable()
    T readFrom()
}
```

Exception Handling

- Application code is allowed to throw any Exception
- Thrown exceptions are handled by the JAX-RS runtime if you have registered an exception mapper
- An exception mapper can convert an exception to an HTTP response
- If the thrown exception is not handled by a mapper, it is propagated and handled by the container

Exception Handling

- JAX-RS provides the javax.ws.rs.WebApplicationException
- This can be thrown by application code and automatically processed by JAX-RS without having to write an explicit mapper
- When JAX-RS sees that a WebApplicationException has been thrown, it catches the exception and calls its getResponse() method

Exception Handling

- Example: ex07_1
  - Exception Mapper is annotated with @Provider

HATEOAS

- Data format provides extra information on how to change state of your application
- Atom Links:

```
<link rel="next"
      href=
http://example.com/customers?start=2&size=2
      type="application/xml" />
```

- Example: ex10_1

HATEOAS

- Advantages
  - Location transparency
    - Server can change the links without clients having to know about them
  - Logical name to a state transition

    href="http://example.com/customers?start=2&size=2"

Caching

- Conditional GETs
  - Server sends ‘max-age’ and ‘Last-Modified’ headers
  - Client sends ‘If-Modified-Since’ header
  - Server sends either
    - 304 - ‘Not-Modified’
    - 200 – new representation

Caching

- Conditional GETs
  - Server sends ‘Etag’
  - Client sends Etag value in ‘If-None-Match’ header
  - Server sends either
    - 304 - ‘Not-Modified’
    - 200 – new representation

Caching

```
public interface Request {
....
ResponseBuilder
evaluatePreconditions(EntityTag eTag)

}
```

Reading

- Chapters
  - 1, 2, 3, 4, 5, 6, 7, 10, 11
- RESTful Java with JAX-RS 2.0
  - Chapters 1 and 2
- Java for Web Applications
  - Chapter 13

References

- http://www.hascode.com/2011/09/rest-assured-vs-jersey-test-framework-testing-your-restful-web-services/
- http://www.baeldung.com/2011/10/13/integration-testing-a-rest-api/
- http://www.javacodegeeks.com/2011/10/java-restful-api-integration-testing.html
- http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

REST Service Example

- REST API for BitBucket
  - http://restbrowser.bitbucket.org/
- Resources
  - User, Privileges, Follows, Repositories, Issues, Milestones

Field and Bean Properties

- When a resource class is instantiated, the values of field and bean properties annotated with one of the following annotations are set according to the semantics of the annotation:
  - @QueryParam: Extracts the value of a URI query parameter (http://localhost:8080/classes?name=cs378)
  - @PathParam: Extracts the value of a URI template parameter (http://localhost:8080/classes/cs378/)
  - @CookieParam: Extracts the value of a cookie
  - @HeaderParam: Extracts the value of a header
  - @MatrixParam: Extracts the value of a URI matrix parameter (http://localhost/ut/dept;name=cs/classes;name=cs378/)
  - @Context: Injects an instance of a supported resource
    - UriInfo, HttpHeader, Request

Field and Bean injection

- Injection of properties occurs at object creation time
- Therefore, use of annotations on resource class fields and bean properties is only supported for the default per-request resource class lifecycle
- A JAX-RS implementation is required to set these properties only for root resources
  - It is the responsibility of the application to set the properties for sub resources

Environment

- The container-managed resources available to a JAX-RS root resource class depend on the environment in which it is deployed.

- Resources that are available for JAX-RS apps deployed within a Servlet container
  - ServletConfig, ServletContext, HttpServletRequest, HttpServletResponse

Context

- JAX-RS provides facilities for obtaining and processing information about the application deployment context and the context of the individual requests

- Available contexts for resource classes
  - UriInfo, HttpHeaders, Request, SecurityContext

Context Parameter: UriInfo

```
@GET
@Produces(“text/plain”)
public String listQueryParams(@Context UriInfo info)
{
    StringBuilder buf = new StringBuilder();
    for(String param:
info.getQueryParameters().keySet()) {
      buf.append();
   }
   return buf.toString();
}
```

Context Parameter: HttpHeaders

- public String listHeaderNames(@Context HttpHeaders headers)

Context Parameter: Request

```
@PUT
public Response updateAccount(@Context Request request,
Account accnt) {
	EntityTag tag = getCurrentTag();
	 ResponseBuilder responseBuilder =
request.evaluatePreconditions(tag);
	if (responseBuilder != null) {
		return responseBuilder.build();
	}
	else {
		return updateAccount(accnt);
	}
}
```

Functional Testing

Devdatta Kulkarni

Functional Testing

- Testing from end user's perspective whether the REST Service/API is working as expected
- Steps:
  - Design and implement your REST Service
  - Create tests that programmatically exercise the API
  - For each test:
    - Save the response
    - Write assertions

Differences between unit and functional testing

- Unit testing is concerned with testing individual layers of your application
- Functional testing is concerned with testing whether the entire application is working as expected or not
- Unit testing involves mocking out the dependencies of the code under test
- Functional tests do not involve mocking

Differences between unit and functional testing

- Changing application's internal architecture should not affect functional tests if they are written properly

- Unit tests are *supposed to* get affected by changes to application's architecture

Similarities between unit testing and functional testing

- Both involve writing assertions on the output

Functional testing tools

- What does such a tool need?
  - Ability to make REST calls
  - Ability to capture response data
  - Ability to parse response data using different parsing criteria
  - Ability to write and verify assertions
- SOAPUI
  - http://www.soapui.org/
- Spock
  - https://code.google.com/p/spock/
- Straight Java libraries

SOAP UI

- UI-based test tool
- Supports using of XPath for parsing response data via language Groovy
- Supports sharing test code between different test plans
- We will *not* use it
  - Latest version seems unstable (at least on Mac)
  - Need to learn Groovy

Spock

- https://code.google.com/p/spock/
- Java and Scala based functional testing tool
- Requires understanding of Java and Scala
- We will not use it

Approach that we will use

- Use Java code
- Example:
  - ex03_1
- Conceptual steps:
  - Build a Client
  - Make a request
  - Read the response
  - Assert

Running Functional Test

- Orielly Book example: ex03_1
- Steps:
  - New -> Java Project
  - Uncheck “Use default location”
  - Browse to the folder containing code for ex03_1
  - Change Junit pom dependency to use version 4.12
  - Next->Finish
  - Configure -> Convert to Maven Project
  - Maven -> Update Project
  - Change Context Root from ex03_1 to “/”
    - Right click on project
    - Properties
    - Web Project Settings
  - Deploy the application by running the server
  - Run “CustomerResourceTest”
  - The test should pass

Persistent Storage using Databases

Devdatta kulkarni

What is Persistent Storage?

- Storage medium that supports storage of data independent of whether an application program is running or not
  - Examples of persistent storage
    - Files written to disk
    - Records written to a database (topic of today's lecture)
- Opposite of persistent storage
  - In-memory storage (Main memory, random access memory, memory)
  - Examples of Non-persistent storage
    - Objects in our Java applications
      - The object "exists" only while the program is running

What is a database?

- Persistent storage in which data is in one or more *tables*
- Data is managed by database server
- A table has one or more *columns*

Retrieving data from a database

- How to retrieve data from a database table?
  - Using Structured Query Language (SQL)

- What is SQL?
  - SQL is a *declarative language* that supports writing queries over a set of tables to retrieve records that match the specified query criteria

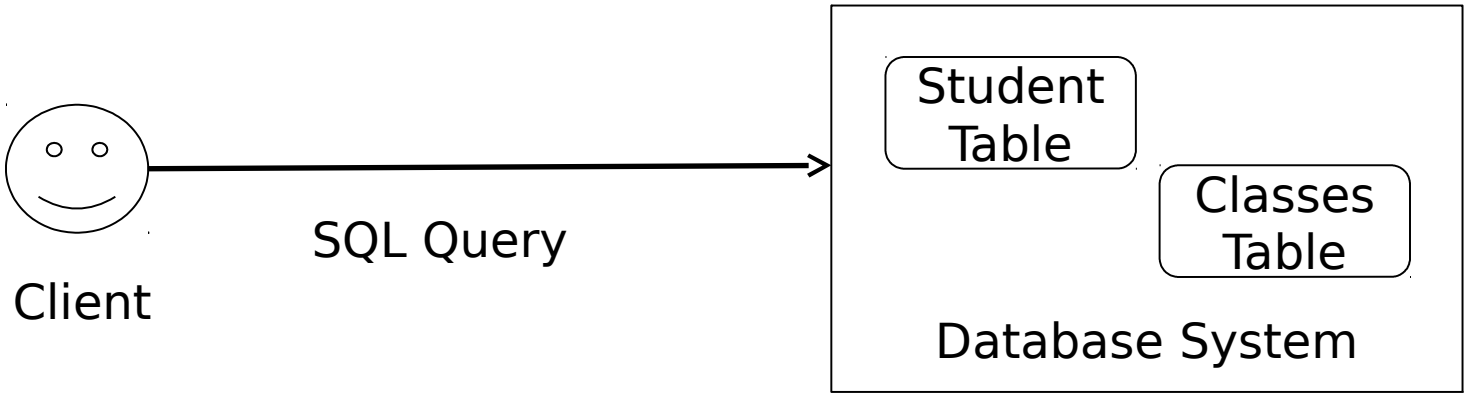
SQL

- What is a declarative language?
  - It is a language in which you write code identifying “what” the answer should look like instead of writing code identifying “how” to produce the answer

- Suppose we wanted to retrieve records of all the students living in Austin from the Student table:
  - In the declarative model, we will write a query of the following form and send it to the database system which manages all the database tables

  “Retrieve all records from the Student table for which City column is equal to Austin”

Querying a Database system



SQL Statements and CRUD Operations

- SQL Insert == Create (REST POST)
- SQL Update == Update (REST PUT)
- SQL Delete == Delete (REST DELETE)
- SQL Select == Read (REST GET)

SQL Examples

- http://www.w3schools.com/sql/default.asp

- Create
  - http://www.w3schools.com/sql/sql_insert.asp
- Read
  - http://www.w3schools.com/sql/sql_select.asp
- Update
  - http://www.w3schools.com/sql/sql_update.asp
- Delete
  - http://www.w3schools.com/sql/sql_delete.asp

SQL

- Some of the most important SQL commands
  - http://www.w3schools.com/sql/sql_syntax.asp
    - Create database
    - Create table
    - Select
    - Update
    - Delete
    - Insert Into
    - Alter database
    - Alter table
    - Drop table

Installing MySQL

- MySQL Community Server
  - http://dev.mysql.com/downloads/
    - On Mac, download the .dmg file
- Starting MySQL Server (on Mac)
  - In Spotlight search for MySQL
  - MySQL window should open up with “Start MySQL Server” button on it
  - MySQL Server Instance should be in “stopped” status
  - Hit the “Start MySQL Server” button; you may be prompted for a password
  - Enter the password; MySQL server should be up
- MySQL Clients
  - Command line
    - /usr/local/mysql/bin/mysql – if you install the .dmg from above link
  - MySQL Workbench
  - SQuirrel SQL
    - http://squirrel-sql.sourceforge.net/
- Local MySQL Server
  - 127.0.0.1
  - Port number 3306
  - Usename: root

Setting up the Database

- Run mysql client
  - /usr/local/mysql/bin/mysql -u root
- Create Database
  - create database student_courses;
- Create User
  - Example: create user 'devdatta'@'localhost’;
- Grant Privileges
  - GRANT ALL ON student_courses.* TO ‘devdatta’@’localhost’;
- Logout
  - On mysql prompt: “quit”
- Login as user;
  - /usr/local/mysql/bin/mysql –u devdatta –h localhost
- Create Table
  - use student_courses;

Problem

- Design a database schema to represent the following:
  - Courses in the CS department
  - Students in the CS department
  - A student can take at most one course
  - A course can be taken by zero or more students
- We want to support following queries:
  - Find all the students who are taking a particular course
  - Find all students and the courses that they are taking

Designing DB

- A *course* table to represent courses
- A *students* table to represent students
- A *relationship* from the *students* table to *courses* table to indicate that a student is taking a particular course

Courses table

A row in the table represents one course

Table columns:

- Name of the course
- Course number
- A unique id

Constraints:

- Name: Should not be NULL
- Course number: Should not be NULL
- Unique id: This will be the PRIMARY KEY of the table

Primary key

- What is a primary key?
  - Any attribute that can uniquely identify a row in the table
- If the resource has a unique attribute, can we use that as the primary key (natural key)?
  - Examples:
    - SSN number to represent a person
    - UTEID to represent a student
- Or, should we generate unique ids and use those (surrogate key)?

Pros/Cons Analysis

- Using Natural keys (resource attributes)
  - Pro
    - Easier to write queries
      - http://stackoverflow.com/questions/63090/surrogate-vs-natural-business-keys
  - Con
    - Problematic if the attribute name is changed in the future
- Using surrogate keys
  - Pro
    - Not affected by changes to a resource’s attributes
  - Con
    - Does not have meaning outside of the service

Courses table

create table courses(name varchar(255) NOT NULL, course_num varchar(20) NOT NULL, course_id int NOT NULL AUTO_INCREMENT, PRIMARY KEY(course_id));

course_id is the *surrogate key*

AUTO_INCREMENT:

- Let the database generate the values for course_id

Student table

A row in the table represents one student

A student can take at most one course

Table columns:

- Name of the student
- A unique id
- A column to capture the relationship between a student and a course

Constraints:

- name: Should not be NULL
- Unique id: This will be the PRIMARY KEY of the table
- course_id: A FOREIGN KEY to courses table

Foreign Key

- What is a Foreign key?
  - A column in a table that refers to primary_key column in another table

Student table

create table students(name varchar(255) NOT NULL, student_id int AUTO_INCREMENT, course_id int, PRIMARY KEY(student_id), FOREIGN KEY(course_id) REFERENCES courses(course_id));

student_id is the *surrogate key*

AUTO_INCREMENT:

- Let the database generate the values for student_id

Populate the tables

- insert into courses(name, course_num) values("Data Management", "CS347");
- insert into students(name) values("Student 2");
- insert into students(name, course_id) values("Student 4", (select course_id from courses where course_num="CS378"));

Queries

- Queries:
  - Find all the students who are taking a particular course
  - Find all students and the courses that they are taking

Join

- A *join* is a mechanism that allows writing of queries over several tables

- A join of two tables selects all the rows from both the tables as long as there is a match between the specified columns of the two tables

Types of Joins

- Inner Join
  - If we imagine tables being sets, then inner join can be thought of as *intersection* of the two sets with the intersection criteria being the matching column values on the join column
- Left outer join
  - Inner join + rows from the “left” table which may not have matched on the join column
- Right outer join
  - Inner join + rows from the “right” table which may not have matched on the join column
- Full outer Join
  - Left outer join + Right outer join – duplicates
    - MySQL does not support it

Join

- Find all students who are taking course CS378

```
select * from students join courses on students.course_id = courses.course_id where courses.course_num="CS378";
```

Join

- Find all students and the courses that they are taking
- Find the students who are taking at least one course and find the information of the course that they are taking

select * from students join courses on students.course_id = courses.course_id;

Left outer join

- Find all the students irrespective of whether they are taking any course; for those who are taking at least one course, find the information about the course

```
select * from students left outer join courses on students.course_id = courses.course_id;
```

Right outer join

- Find all the courses irrespective of whether a course is being taken by any student. For the course that is being taken by at least one student, find the information about all those students

```
select * from (students right outer join courses on students.course_id = courses.course_id);
```

Revisiting Problem

- We want to add support for the following:
  - A student can take more than one course
- And we want to support following query:
  - Find all courses that a student is taking

How to store data in tables?

- How would we represent a Student who is taking several classes?
  - Issues
    - We would need to repeat student information in each such record
    - If we have to change any attribute of a student then we will have to update all such records
- Ideally we would like to:
  - Avoid repetition
  - Store data in one place
- The concept of *Normalization* is defined for this purpose

Normalization

- Store data in one place
  - Single row in a single table
- Student – course example
  - 3 tables
    - student
    - courses
    - student_courses_xref
- Flip-side to normalization
  - In order to retrieve data, we need to query it using ``joins''

Schema change

- Our current schema cannot support the new requirement
  - Why?
    - Because it is not possible to represent a student with multiple courses
      - Adding multiple rows per student in the *students* table is not an option because each row in that table represents one and only one student
- Solution:
  - 3 tables
    - students
    - courses
    - student_courses_xref

The cross reference table

- Each row represents <student_id, course_id> pairs
  - E.g.:

<student_1, course_1>

<student_1, course_2>

<student_2, course_1>

Join

- Find all students and the courses that they are taking

```
select s.name, c.name from student s join student_courses_xref scx on s.uteid=scx.uteid join courses c on c.course_id=scx.course_id;
```

```
select s.name "Student", c.name "Course" from student s, course c, student_courses_xref scx where s.student_id=scx.student_id and c.course_id=scx.course_id;
```

Left outer join

- select s.name as 'student_name', c.name as 'course_name' from (students s left join student_courses_xref scx on s.student_id = scx.student_id left join courses c on c.course_id = scx.course_id) order by course_name;

Right outer join

- select s.name as 'student_name', c.name as 'course_name' from (students s right join student_courses_xref scx on s.student_id = scx.student_id right join courses c on c.course_id = scx.course_id) order by course_name;

References

- SQL Tutorial
  - http://www.w3schools.com/sql/
- Primary key vs. surrogate key
  - http://stackoverflow.com/questions/2186260/when-to-use-an-auto-incremented-primary-key-and-when-not-to
  - http://stackoverflow.com/questions/63090/surrogate-vs-natural-business-keys

JDBC

Devdatta Kulkarni

Databases and Java

- JDBC
  - Java Database Connectivity
  - MySQL Connectors
    - http://dev.mysql.com/downloads/connector/
- Example
  - JDBC

JDBC topics

- Maven dependency
  - commons-dbcp
  - mysql-connector-java
- Main Concepts/Classes
  - Data source
  - Connection
  - PreparedStatement
  - ResultSet
  - Transactions

Data Source

- A *data source* is where the data is stored by your application (or is accessed from)
  - E.g.: DBMS, a legacy file system
- A JDBC application connects to a target data source using one of the following classes
  - DriverManager
    - Automatically loads any JDBC 4.0 drivers available on the class path
  - DataSource
    - Abstracts the details of the underlying data source from your application

DriverManager

- Database connection is established using the DriverManager’s “getConnection” method
  - dbURL = "jdbc:mysql://localhost:3306/student_courses";
  - conn = DriverManager.*getConnection(dbURL,*"*devdatta*", "");

DataSource

```
// Setup data source
BasicDataSource ds = new BasicDataSource();
ds.setUsername(this.dbUsername);
ds.setPassword(this.dbPassword);
ds.setUrl(this.dbURL);
ds.setDriverClassName("com.mysql.jdbc.Driver");

// Open connection
Connection conn = ds.getConnection();
```

Connection class

- Connection
  - Represents a database connection
    - Typical required parameters: Database URL, username, password
- Connection object is *not* thread safe

- How to support application's multiple threads?
  - Option 1: Accessing Connection through synchronized methods
    - Not a good strategy
      - Different threads may be accessing different data items; such threads may end up blocking each other unnecessarily
  - Option 2: Use a separate connection per thread
    - Not a good strategy either
      - Each thread typically represents a user request; your application will have lots of users
      - It cannot open new Connections for all (it will run out of resources)

Connection Pool

- Connection Pooling
  - Keep a pool of open connections
  - Each thread will get its own connection
  - Threads will block if there are no connections available in the pool
  - A connection is given back to the pool once a thread is done using it
  - Apache DBCP
    - Library that implements connection pooling
    - http://commons.apache.org/proper/commons-dbcp/

PreparedStatement

- Represents a *parameterized SQL query*
- Parameter values are set based on data provided at runtime
- Use “executeQuery” to execute a statement that returns some value
- Use “executeUpdate” to execute a statement that does not return a value

PreparedStatement

```
String query = "select * from courses where
course_id=?";

Connection conn = ds.getConnection();

PreparedStatement s =
conn.prepareStatement(query);

s.setString(1, String.valueOf(courseId));

ResultSet r = s.executeQuery();
```

PreparedStatement

- “Parameter binding provides a means of separating executable code, such as SQL, from content, transparently handling content encoding and escaping.“ [1]

[1] http://martinfowler.com/articles/web-security-basics.html#BindParametersForDatabaseQueries

PreparedStatement – Getting keys back

```
String insert = "INSERT INTO courses(name,
course_num) VALUES(?, ?)";

PreparedStatement stmt =
conn.prepareStatement(insert,
Statement.RETURN_GENERATED_KEYS);

stmt.setString(1, c.getName());
stmt.setString(2, c.getCourseNum());

int affectedRows = stmt.executeUpdate();
```

ResultSet

- Represents a set of records from the table that satisfy the specified query criteria
- The records are accessed through a *cursor*
- The cursor is a pointer that points to one row of data in the ResultSet
- Initially the cursor is positioned before the first row
- The method ResultSet.next moves the cursor to the next row

ResultSet – Returning a single row

```
ResultSet r = s.executeQuery();

        if (!r.next()) {
            return null;
        }

NewCourse c = new NewCourse();
c.setCourseNum(r.getString("course_num"));
c.setName(r.getString("name"));
c.setCourseId(r.getInt("course_id"));
```

ResultSet – Returning multiple rows

```
ResultSet r = s.executeQuery();

List<NewCourse> courseList = new
ArrayList<NewCourse>();

while (!r.next()) {
    NewCourse c = new NewCourse();
    c.setCourseNum(r.getString("course_num"));
    c.setName(r.getString("name"));
    c.setCourseId(r.getInt("course_id"));
    courseList.add(c);
}
return courseList;
```

ResultSet Types

- Determines how does the *cursor* moves
  - TYPE_FORWARD_ONLY
    - Cursor only moves forward
  - TYPE_SCROLL_INSENSITIVE
    - The result can be scrolled; the result set is insensitive to changes made to the underlying data source while it is open
  - TYPE_SCROLL_SENSITIVE
    - The result can be scrolled; the result set reflects the changes made to the underlying data source while it is open

Transactions

- Problem:
  - How to ensure that more than one database statements are executed as a unit?
- Solution:
  - Transactions

Example where a transaction is needed

- Suppose we want to create a new project and add a meeting as part of project creation:

```
POST /myeavesdrop/projects/
<project>
   <name>NewSolum</name>
   <description>NewSolum</description>
   <meetings>
     <meeting>
        <name>M1</name>
        <year>2016</year>
        <month>March</month>
        <day>26</day>
     </meeting>
   </meetings>
</project>
```

Transactions

- We need to define a transaction that includes the following two statements:
  - Insert the new project's name and description into the "projects" table
  - Insert the project's meeting information into the "meetings" table

  - We need to execute both these within a single transaction
    - Either both are committed successfully or both are not
      - The execution of the two statements needs to be *'atomic'*

Transactions

- A transaction is a set of statements that are executed as a *unit (also called the 'atomicity' property of a transaction)*
  - Either all statements in a transaction are completed successfully or none are
  - The effect of executing a statement is reflected within a database only after it is *committed*

- A *Connection* is in auto-commit mode by default
  - Each individual statement is treated as a transaction
  - If we want to group two or more statements into a transaction then we have to *disable* the auto-commit mode
    - conn.setAutoCommit(false);
  - Commit/rollback of the transaction is our responsibility
    - conn.commit();
    - conn.rollback();

- https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

Transactions

- A simple approach for the database system to guarantee atomicity (*all or nothing property*) is to perform only one transaction at a time
- But this approach would limit performance severely
- So we want transactions to be executed concurrently to gain performance benefits
- However, then we need to solve for *isolation*

Isolation

- What is isolation?
  - At high-level it means preserving data integrity by ensuring that concurrently executing transactions do not interfere with one another
  - What does this mean?
    - Prevent transactions from interfering with one another
    - For example, if two transactions are modifying the same row then the *isolation* property guarantees that none of the transactions would see an *intermediate* state of the row

How to provide isolation?

- We know how to prevent concurrent updates to shared state
  - Use locks
- At a high-level that is exactly what the database does
  - It *locks* the rows that are part of the transaction
  - After a lock is set, it remains in effect till either the transaction is committed or rolled back
  - No other transaction can access the rows that are locked

Transaction Isolation

- Instead of all-or-nothing isolation, can we do better to improve performance?

- Define different types of locks (isolation levels) that provide different concurrency guarantees
  - Read_uncommitted
  - Read_committed
  - Repeatable_read
  - Seriazable
    - Guarantees that there will no dirty reads, no non-repeatable reads, and no phantom reads

Concurrent Transactions: Issues

- Read uncommited level leads to *Dirty reads*
  - A transaction reads a row from a database table containing uncommitted changes from another transaction.
- Read commited leads to *Non-repeatable reads*
  - A transaction reads a row from a database table, a second transaction changes the same row and the first transaction re-reads the row and gets a different value.
- Repeatable reads leads to *Phantom reads*
  - A transaction re-executes a query, returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.
- http://www.onjava.com/pub/a/onjava/2001/05/23/j2ee.html?page=2

Transaction Isolation Levels

- Read Uncommitted
  - Read data that is not yet committed (dirty reads)
- Read Committed
  - No dirty reads; but non-repeatable reads
- Repeatable Read
  - No dirty reads
  - No non-repeatable reads
- Serializable
  - No dirty reads
  - No non-repeatable reads
  - No phantom reads

  - Example:
    - findTransactionIsolationLevel
    - testRepeatableRead
    - testSerializableRead

Transaction Isolation Levels

- Serializable
  - http://sqlperformance.com/2014/04/t-sql-queries/the-serializable-isolation-level
- Repeatable read
  - http://sqlperformance.com/2014/04/t-sql-queries/the-repeatable-read-isolation-level
- Why use read-uncommitted?
  - http://stackoverflow.com/questions/2471055/why-use-a-read-uncommitted-isolation-level

JDBC Reference

- https://github.com/devdattakulkarni/ModernWebApps/tree/master/JDBC
- http://stackoverflow.com/questions/4246646/mysql-java-get-id-of-the-last-inserted-value-jdbc
- http://stackoverflow.com/questions/42648/best-way-to-get-identity-of-inserted-row
- http://stackoverflow.com/questions/8146793/no-suitable-driver-found-for-jdbcmysql-localhost3306-mysql

JDBC Reference

- http://stackoverflow.com/questions/9428573/is-it-safe-to-use-a-static-java-sql-connection-instance-in-a-multithreaded-syste
- http://stackoverflow.com/questions/7592056/am-i-using-jdbc-connection-pooling
- Apache DBCP2:
  - http://svn.apache.org/viewvc/commons/proper/dbcp/trunk/doc/BasicDataSourceExample.java?view=markup

JDBC References

- Auto increment id
  - http://stackoverflow.com/questions/1915166/how-to-get-the-insert-id-in-jdbc
- Foreign Key
  - http://stackoverflow.com/questions/25920251/how-to-automatically-insert-foreign-key-references-in-tables-in-mysql-or-jdbc
- Triggers
  - http://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html

Logging

Devdatta Kulkarni

Logging

- Why do we need logging?
  - Log ``interesting'' events in an application
    - Application configuration messages
    - Informational debug messages
    - Diagnostic messages
    - Warnings
    - Exceptions

Logging approaches

- Use System.out.println()

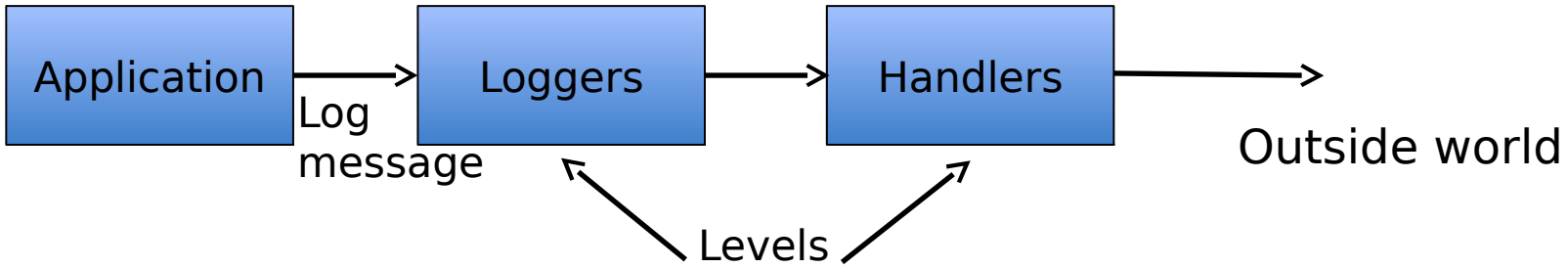
- What are the issues?
  - Not fine-grained enough
    - Cannot selectively control which messages are printed
  - Not flexible enough
    - Cannot send/redirect the messages to different “syncs”
      - Only sync is the standard output stream

Logging requirements

- Fine-grain control
  - Should be able to support different categories of log messages
- Flexibility
  - When to log?
  - Where to log?

Java Logging Architecture

- Loggers
  - Generate log records
- Handlers
  - Send log records to different destinations



http://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html

https://jcp.org/aboutJava/communityprocess/review/jsr047/spec.pdf

What is a logging level?

- A logging level defines relative *order* for log messages
- Logging Levels in Java's default Logging package (java.util.logging):
  - SEVERE
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST

Descending order of level

Logging Levels

- Levels are associated with
  - Messages
  - Loggers
    - If set to null, level is inherited from the parent
  - Handlers

- Logging algorithm:
  - A log message is sent to the Logger
  - If message’s level >= logger’s level then the logger passes the message to the handler
  - If message’s level >= handler’s level then the handler sends the message to appenders for output

Logging Levels

- Levels are associated with:
  - Messages
  - Loggers
    - If set to null, level is inherited from the parent
  - Handlers
    - StreamHandler
    - ConsoleHandler
    - FileHandler
    - SocketHandler
    - MemoryHandler

Logging Algorithm details

- Applications make logging calls on Logger
- Logger allocates LogRecord objects which are passed to Handler for publication
- The LogManager class keeps track of a set of global Handlers
- By default all Loggers send their output to this standard set of global Handlers
- Loggers may also be configured to ignore the global Handler list and/or to send output to specific target Handlers.

Ref section 2.1 of https://jcp.org/aboutJava/communityprocess/review/jsr047/spec.pdf

Distinction between Loggers and Handlers

- Loggers
  - Create LogRecord
  - Sends LogRecords to the default set of global handlers
  - If Logging is enabled and message passes the level check, Logger is careful to minimize execution costs before passing LogRecord into the Handler
- Handlers
  - Localization and formatting of LogRecord (relatively expensive task)
  - Send LogRecords to output

Loggers and Handlers

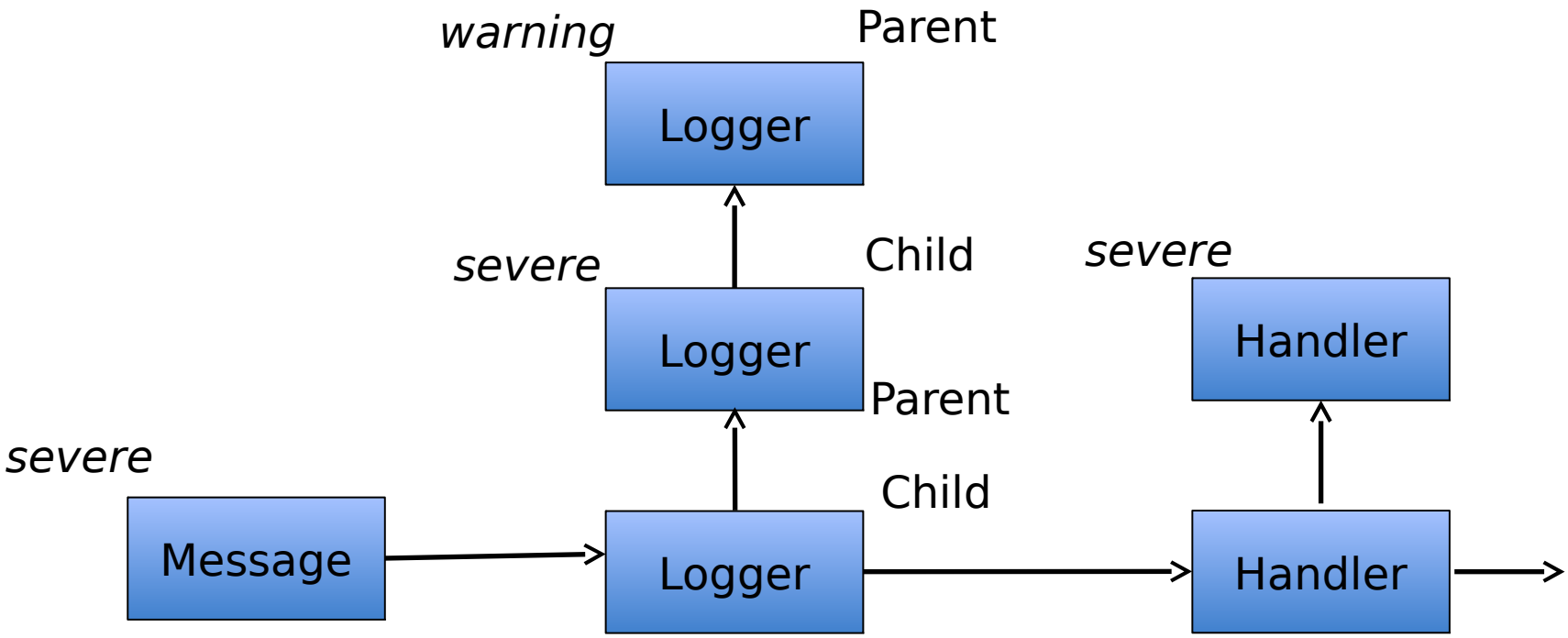
Why perform level checking and filtering in both, loggers and handlers?

- For performance reason
  - If no level checking and filtering is done at Logger, then all messages may need to be sent to all the handlers as there might be more than one handlers each with different destinations
  - If no level checking and filtering in Handler, then all the messages would need to be formatted for output which is an expensive operation

Logging Hierarchy

- Loggers are arranged in a hierarchy
  - The hierarchy is defined based on the names of the loggers
    - The naming hierarchy follows the ``dotted'' notation
      - *cs378.examples.TestLogging* is child of *cs378.examples*
    - If a Class does not have a Logger defined for it, the Logger for its parent is inherited by the Class
      - Root system logger is parent of all loggers
    - A logger will publish the log message to its parent's handlers

Logging example



Logging: Config.properties

Mac

- /Library/Java/JavaVirtualMachines/<JDK Home>/Contents/Home/jre/lib/logging.properties

Ubuntu

- /usr/lib/jvm/java-8-oracle/jre/lib/logging.properties

Specify custom logging.properties

- Use “java.util.logging.config.file” flag

-D java.util.logging.config.file= /Users/devdatta.kulkarni/Documents/UT/ModernWebApps/logging-example/logging.properties

Logging Examples

- Example 1: getLoggerAndHandlerLevels()
- Example 2: testPrintInfoMessage()
- Example 3: testPrintConfigMessageFailure()
- Example 4: testPrintConfigMessageLoggerAndHandlerLevelSet()

Logging Framework

- Logging API vs Implementation
  - Swap out implementation without having to change your code

Popular Logging APIs

- Apache commons
  - http://commons.apache.org/proper/commons-logging/
- Simple Logging Façade for Java
  - http://www.slf4j.org/

Logging APIs and Logging Frameworks

- The logging APIs provide ``adapters'' to convert the logging API events to the logging framework
  - Commons logging provides adapters for:
    - Avalon, java.util.logging, Log4j 1.2, and LogKit
  - SLF4J provides adapters for:
    - Commons logging, java.util.logging, and Log4j 1.2

Log4j 2

- http://logging.apache.org/log4j/2.x/
- Provides both API and implementation

Logging Appenders

- Where are logs written?
  - Console
  - Flat Files
  - Sockets
  - SMTP and SMS
  - Databases

References

- https://jcp.org/aboutJava/communityprocess/review/jsr047/spec.pdf
- http://www.onjava.com/pub/a/onjava/2002/06/19/log.html?page=2

Object/Relational Mapping

Devdatta Kulkarni

Issues with JDBC

- Abstraction mismatch between Java Domain Objects and Relational tables
  - In our code we work with Java objects, but for storage and retrieval from the database, we need to work with *ResultSet*
  - Wouldn’t it be nice to not have to worry about low level classes, such as *PreparedStatement* and *ResultSet*?
- SQL is relatively low level interface for our needs
  - Wouldn’t it be nice to write queries referencing Java objects rather than tables and columns of a database?

Issues with JDBC

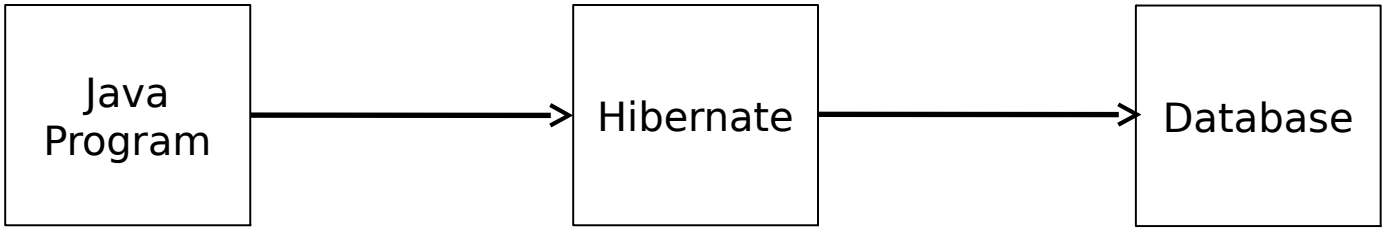
- *PreparedStatement* does not help with parameter types
  - As application developer you need to know the position and type of each parameter in a *PreparedStatement*

Object Relational Mapping

- Technology that maps Java objects to database tables and vice versa

- Specification
  - Java Persistence Architecture (JPA)
    - An API for Java O/RM
  - JPA implementations provided by:
    - Hibernate
    - iBatis
    - MyBatis
    - EclipseLink

Hibernate Main Concepts



- Configuration
  - Specified through “hibernate.cfg.xml” file
- Entities
  - Support JPA annotations
- Sessions

Configuration File

- Name
  - hibernate.cfg.xml
- Preferred location
  - src/main/resources/hibernate.cfg.xml
- Important fields
  - Connection fields
  - Cache provider
    - Session level cache (first level cache)
      - http://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/
    - SessionFactory level cache (second level cache)
      - http://howtodoinjava.com/2013/07/02/how-hibernate-second-level-cache-works/
  - hbm2ddl.auto
    - Field which controls schema creation

Configuration File

- Important Parameters
  - hibernate.connection.driver_class
  - hibernate.connection.url
  - hibernate.connection.username
  - hibernate.connection.password
  - hibernate.connection.pool_size

Hibernate Entities

- A database table is represented using a Java class marked with the @Entity annotation
- The table that the entity maps to is marked with @Table annotation
  - Specifies the primary table for an annotated entity
  - The name of the table can be set by specifying the “name” attribute value
- Each entity instance corresponds to a row in that table
- The table columns are represented as *fields* in the entity class

Entities

@Entity @Table(name = “assignments”)

public class Assignment { ... }

Entities

@Entity @Table(name = “assignments")

public class Assignment { ... }

- Each field gets translated into a table column
- A field that is to be used as the primary_key should be marked with @Id annotation

Requirements for Entity Classes

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance be passed by value as a detached object, the class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Clients must access the entity's state through *accesssor or business methods*.

Annotations within an Entity

- The persistent state of an entity can be accessed either through the entity's instance variables or through JavaBeans-style properties.
- Entities may either use *persistent fields* or *persistent properties*.
- When the mapping annotations are applied to the entity's *instance variables:*
  - It means that the entity uses *persistent fields*
- When the mapping annotations are applied to the entity's *getter methods* for JavaBeans-style properties
  - It means that the entity uses *persistent properties*
- You cannot apply mapping annotations to both fields and properties in a single entity.

Primary Keys in Entities

- Each entity has a unique object identifier
  - The primary key
- An entity may have either a simple or a composite primary key
- Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field

Primary key – Id generation

- Option 1: Use JPA strategies

  @GeneratedValue(strategy=GenerationType.IDENTITY)
- JPA strategies
  - AUTO
  - IDENTITY
  - SEQUENCE
  - TABLE
- Option 2: Use Hibernate’s generator

  @GeneratedValue(generator="increment")

  @GenericGenerator(name="increment", strategy = "increment")

Hibernate Session

- Main interface between a Java application and Hibernate
- Life cycle of a session is bounded by beginning and end of a transaction
- The function of a session is to offer create, read and delete operations for instances of mapped entity classes

Session

- org.hibernate.SessionFactory create and pool JDBC connections
- Open a new Session
  - Session session = sessionFactory.openSession();

Session details

- Session object is not thread-safe. Instead each thread/transaction should obtain its own instance from a SessionFactory.
- If the Session throws an exception, the transaction must be rolled back and the session discarded:
  - The internal state of the Session might not be consistent with the database after the exception
- https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Session.html

Session pattern

```
Session session = factory.openSession();
 Transaction tx;
 try {
     tx = session.beginTransaction();
     //do some work
     tx.commit();
 }
 catch (Exception e) {
     if (tx!=null) tx.rollback();
     throw e;
 }
// continue using the session

 finally {
     session.close();
 }
```

Hibernate Object states

- Objects may exist in one of three states:
  - *transient:* just created, has never been made persistent, not associated with any Session
  - *persistent:* associated with a unique Session
  - *detached:* previously persistent; not associated with a Session anymore

Object state: Transient

- An object is transient if it has just been instantiated using the *new* operator, and it is not associated with a Hibernate Session.

Examples

```
public Long addAssignment(String title) throws Exception {
   Session session = sessionFactory.openSession();
   Transaction tx = null;
   Long assignmentId = null;
   try {
   tx = session.beginTransaction();
(transient) Assignment newAssignment = new Assignment(title, new
Date(), new Long(1));
(persistent)   session.save(newAssignment);
                      tx.commit();
               } catch (Exception e) {
                   tx.rollback();
               }
}
```

Object state: Persistent

A persistent instance has a representation in the database and an identifier value.

It might just have been saved or loaded.

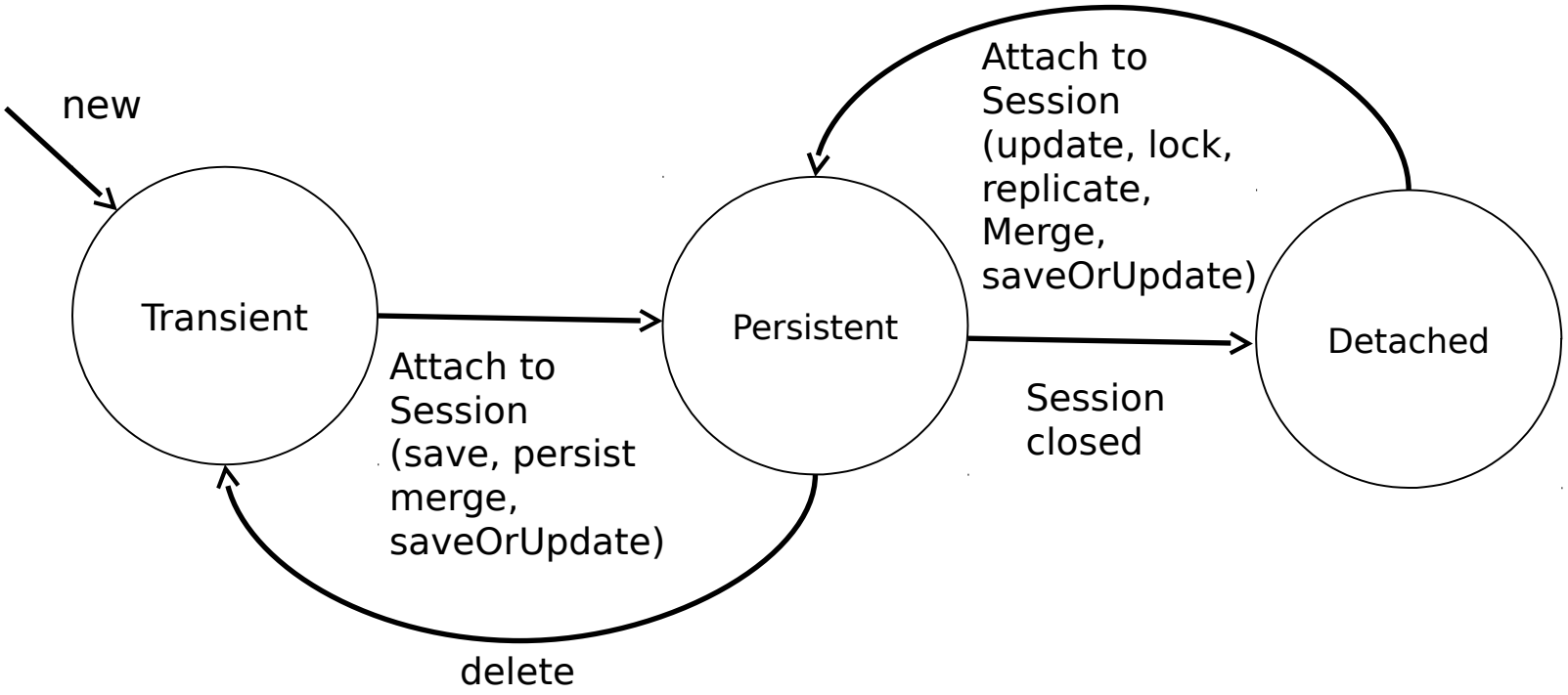
It is by definition in the scope of a Session.

Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes.

Detached state and state transitions

- Object state: Detached
  - Not associated with any Session
  - Was previously persistent
- State transitions
  - Transient instances may be made persistent by calling save(), persist(), merge(), saveOrUpdate()
  - Persistent instances may be made transient by calling delete()
  - Detached instances may be made persistent by calling update(), saveOrUpdate(), lock(), merge() or replicate()

State transitions



https://docs.jboss.org/hibernate/core/3.3/reference/en/html/objectstate.html

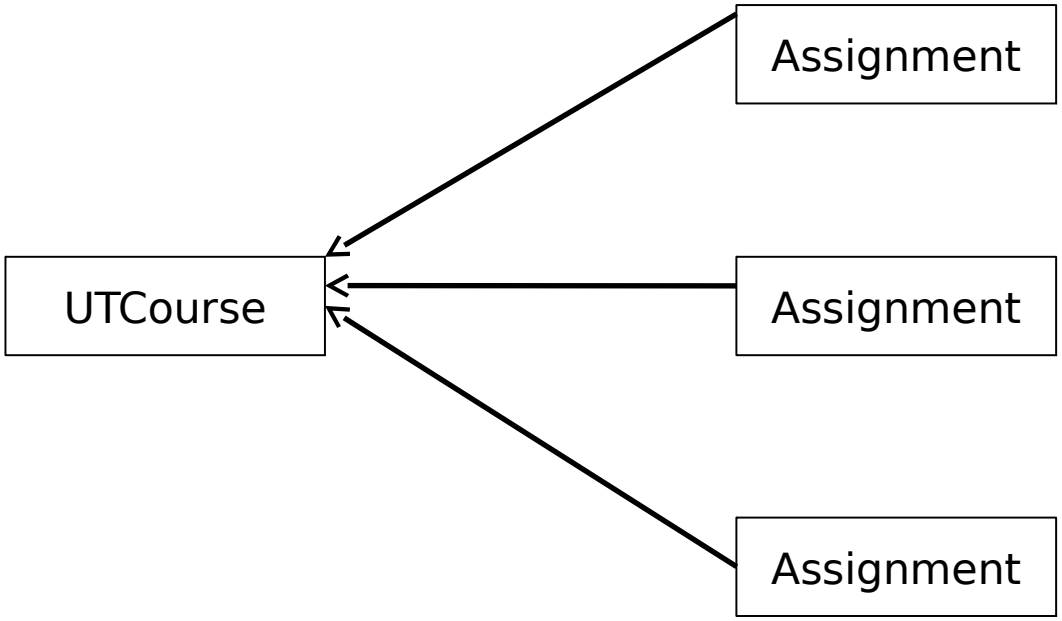
Hibernate and SQL correspondence

Hibernate API methods	Corresponding SQL actions
save(), persist()	insert
delete()	delete
update(), merge()	update
saveOrUpdate()	insert or update

JPA Entity Relationships

Example

- A UTCourse can have one or more assignments



JPA Entity Relationships

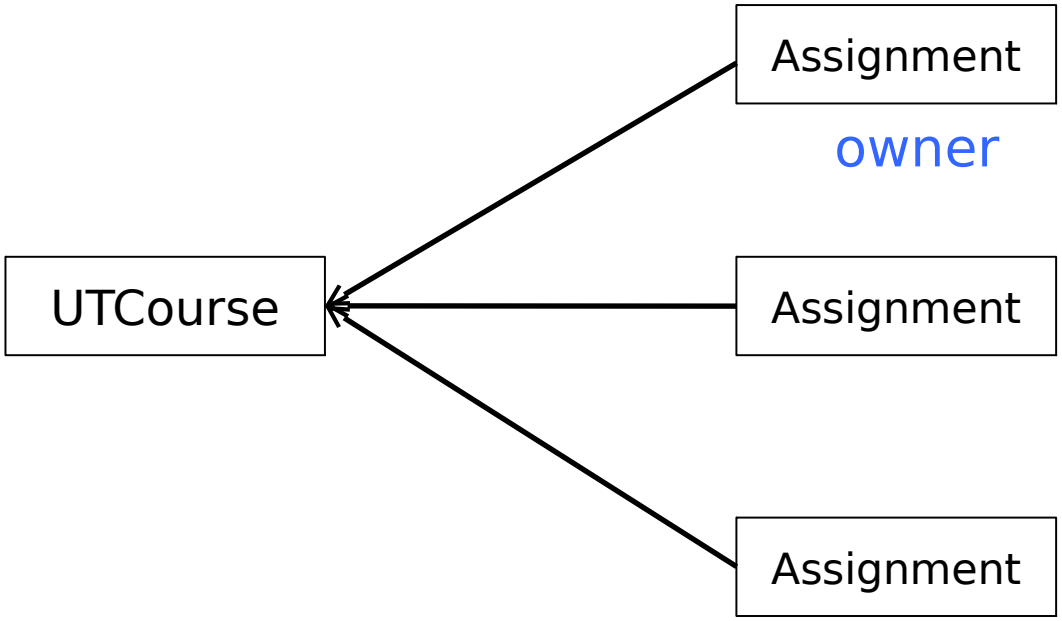
- **One-to-one**: Each entity instance is related to a single instance of another entity.
- **One-to-many**: An entity instance can be related to multiple instances of the other entities.
- **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity.
- **Many-to-many**: The entity instances can be related to multiple instances of each other.

Direction in Entity Relationships

- Unidirectional
  - Has only an owning side
- Bidirectional
  - Has an owning side and an inverse side
- The owning side of the relationship is the entity that has reference to the other entity
- The owning side determines how Hibernate makes updates to the relationship in the database.

Hibernate Entity Association

- A UTCourse can have one or more assignments



Hibernate Entity Relationship Annotations

- Annotations
  - @ManyToOne
    - Added on the “owning” side of a relationship
    - Typically combined with @JoinColumn annotation
  - @OneToMany
    - Added on the “inverse” side of the relationship
  - Show example

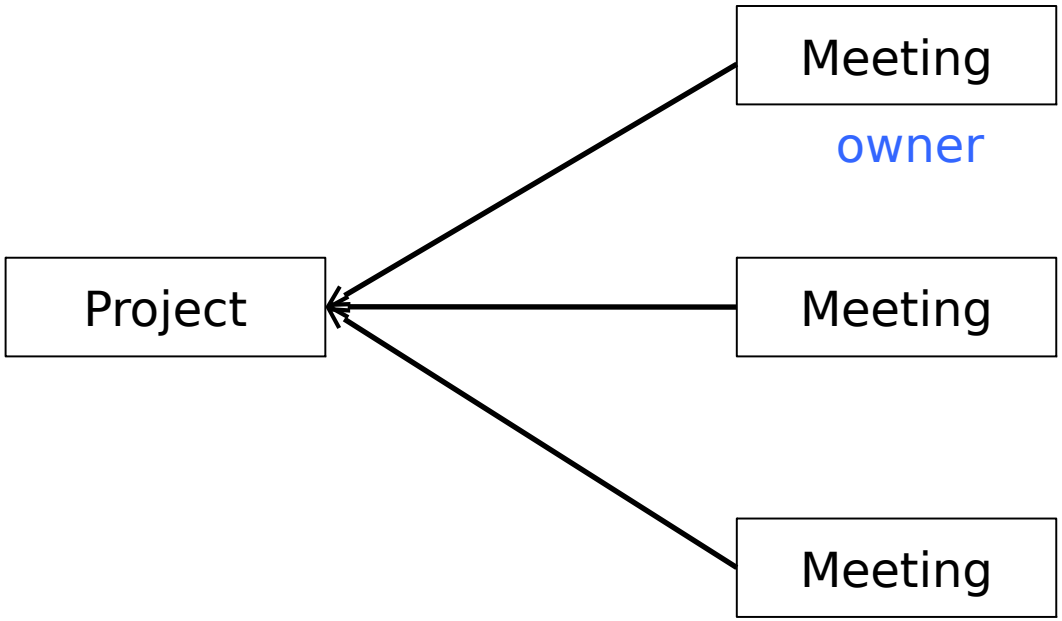
Entity relationships

```
 // In the “owner” side – Assignments class
@ManyToOne
@JoinColumn(name="course_id")
 public UTCourse getCourse123() {
        return this.utcourse;
 }

// In the reverse side  -- UTCourse class
@OneToMany(mappedBy="course123")
public Set<Assignment> getAssignments() {
     return this.assignments;
}
```

Exercise: 10 minutes

- Model a project that can have one or more meetings:
  - Define the data model
    - Entities
    - Fields/columns
    - Relationships

Hibernate Entity Association



Fields/Columns

- Project
- Meetings

Entity relationships

Hibernate Operations

Hibernate Operations

- Insert (REST: Create/POST)
- Select (REST: Read/GET)
  - From a single table without any criteria
  - From a single table with a selection criteria
  - From two tables with join condition
- Delete (REST: Delete/DELETE)
- Update (REST: Update/PUT)

Hibernate: Insert

```
Session session = sessionFactory.openSession();
session.beginTransaction();

Assignment newAssignment = new
Assignment( title, new Date() );

session.save(newAssignment );

session.getTransaction().commit();
session.close();
```

Hibernate Querying

- HQL
  - Hibernate Query Language
    - Similar to SQL, but closer to the Java world
    - Supported via the *Criteria* API
- Native SQL

HQL: Criteria API

Criteria criteria =
session.createCriteria(Assignment.class).
add(Restrictions.*eq("title", "ETL"));*

Query condition

Session.m1(param).m2(param2)

Method chaining

Selection Query using Native SQL

```
List<Assignment> assignments =
session.createQuery("from Assignment
where course=1").list();
```

“Assignment” → Entity class name

Parameterized variables

```
String query = "from Assignment a where a.title = :title”;

Assignment a = (Assignment)session.createQuery(query).setParameter("title", title).list().get(0);
```

Join Query

Use names from Java side when creating the query

- Entity class name
- Property name

String query = "from Assignment a join a.course123 c where c.courseName = :cname";

Assignment: Entity class name

course123, courseName: Property names

Hibernate Delete

- Deleting a row in a table that *is not* part of any relationship(s)
- Deleting a row in a table that *is* part of a relationship
  - leads to ConstraintViolationException if no cascaded delete specified
- Cascading delete
  - Add @Cascade({CascadeType.DELETE}) on the inverse side of relationship
- Parameterized delete

Hibernate Criteria API

- http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/querycriteria.html#querycriteria-associations
- http://levelup.lishman.com/hibernate/hql/joins.php

Advanced Hibernate concepts

- Hibernate caching
  - Session cache
  - SessionFactory cache
  - Query cache
  - http://www.tutorialspoint.com/hibernate/hibernate_caching.htm
- N+1 query issue
  - http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue

H2

- In memory database
- Useful for testing and debugging

References

- ORM/JPA
  - http://martinfowler.com/bliki/OrmHate.html
  - http://hibernate.org/orm/
  - https://docs.jboss.org/author/display/AS7/JPA+Reference+Guide
  - http://hibernate.org/orm/what-is-an-orm/
  - http://www.h2database.com/html/tutorial.html
  - http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html

References

- http://viralpatel.net/blogs/hibernate-one-to-many-annotation-tutorial/
- http://hibernate.org/orm/what-is-an-orm/
- http://hibernate.org/orm/documentation/getting-started/

Hibernate Operations

- Hibernate operations are performed as part of a session
- A session is
  - Main interface between a Java application and Hibernate
  - Life cycle of a session is bounded by beginning and end of a transaction
  - The function of a session is to offer create, read, and delete operations for the entities
  - Pattern
    - Begin a transaction
    - Add entities to be “saved/modified” in the session
    - Perform commit action
    - Close the transaction
    - On error, roll-back the transaction

Will Hibernate add an Id?

- *No*

- Hibernate expects that each entity has some field with the Id annotation
  - hibernate.AnnotationException is raised if such a field is not present

Schema updates and data migrations

- Hibernate supports schema changes
  - <property name="hibernate.hbm2ddl.auto" value="create-drop" />
  - <property name="hibernate.hbm2ddl.auto" value="update" />
  - http://docs.jboss.org/hibernate/core/3.3/reference/en/html/session-configuration.html#configuration-optional
- Hibernate does not support data migration
  - Typically, separate tool is used
    - https://flywaydb.org/
    - http://www.liquibase.org/
    - http://www.mybatis.org/migrations/

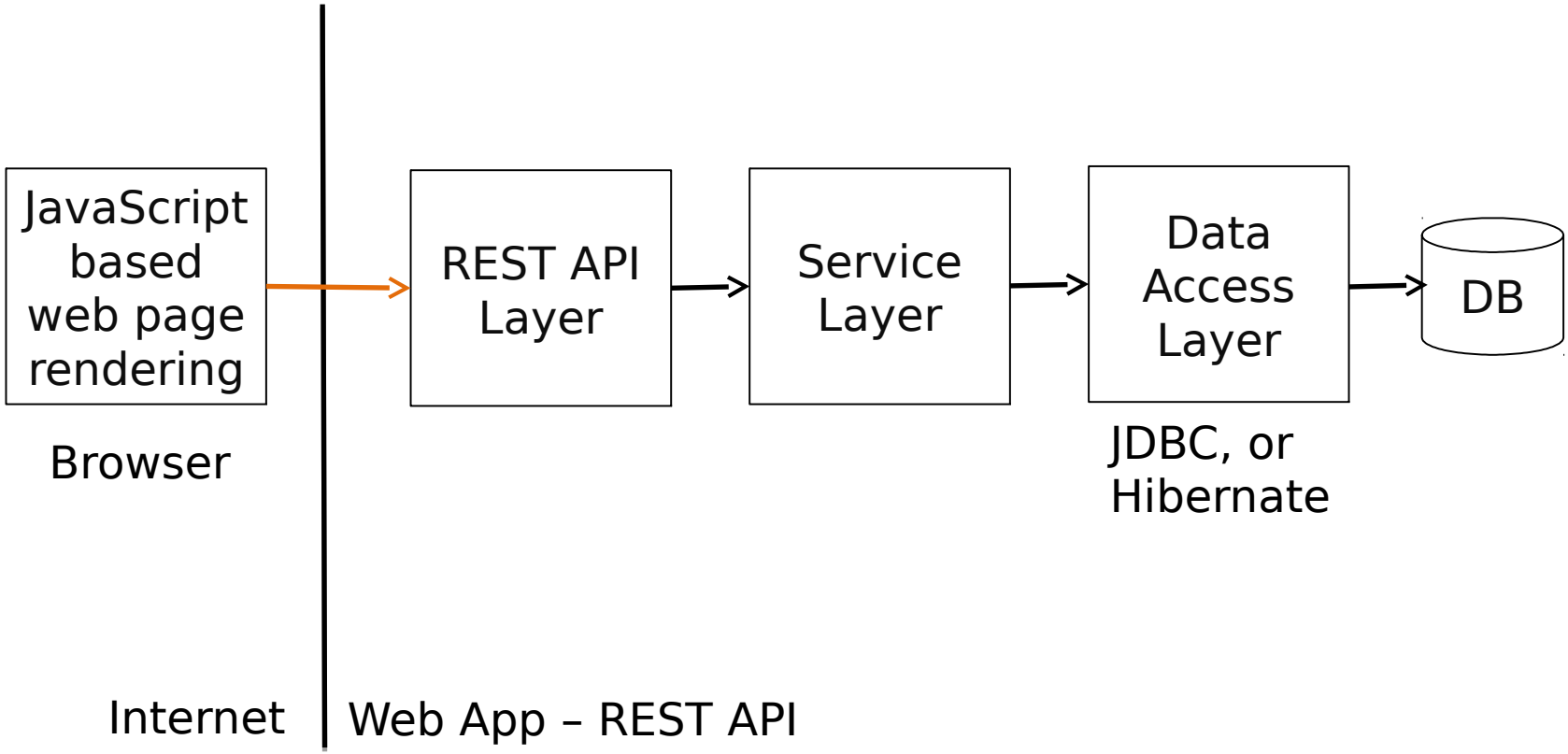
JavaScript

Devdatta Kulkarni

JavaScript

- What is it?
  - Scripting language that controls how web browsers render web pages
- Purpose
  - Make web pages *dynamic*
    - Client side error checking of forms
    - Taking different actions based on *events* happening on the browser
    - Dynamically updating content on the browser
- Where is the script specified?
  - Within <script></script> elements which is included within
    - the <head></head> element
    - the <body></body> element

Modern Web Applications



JavaScript

- Tutorial to follow:
  - http://www.w3schools.com/js/
- Where to add the scripts?
  - src/main/webapp/*.html
  - src/main/webapp/*.js
  - Show Example

JavaScript Example

- https://github.com/devdattakulkarni/ModernWebApps/tree/master/JavaScript-Example
- http://localhost:8080/js-example/test.html
- http://localhost:8080/js-example/ajaxexample.html

JavaScript main concepts

- Document Object Model (DOM) associated with the web page
- In HTML, JavaScript statements are "instructions" to be "executed" by the web browser.
- Event handling model
  - Event bubbling (inner most to outer most)
  - Event capturing (outer most to inner most)
- Asynchronous JavaScript (AJAX)
  - To interact with REST API
  - Cross-origin resource sharing (CORS)

JavaScript

- Script
- Comments
  - //
  - /* */
- Case Sensitive

  lastName = "Doe";
  lastname = "Peterson”;

  lastName and lastname are different variables
- Hyphens are not allowed in JavaScript. Hyphen is reserved for subtractions.

DOM

- The “document” object
  - This object represents the web page
  - http://www.w3schools.com/js/js_htmldom.asp
- Accessing an element in the DOM
  - Use getElementById method
    - document.getElementById(“<some_id>”)
      - <some_id> is the value of the id attribute of some element in the HTML page
- Accessing contents of an element from the DOM
  - Use the “innerHTML” property

Finding HTML Elements

- document.getElementById(*id*)
  - Find an element by element id
- document.getElementsByTagName(*tagName*)
  - Return a list of elements by tag name
- document.getElementsByClassName(*className*)
  - Return a list of nodes by class name

Changing HTML Elements

- *element*.innerHTML
  - Change the inner HTML of an element

- *element.attribute*
  - Change the attribute of an HTML element

Adding and Deleting Elements

- document.createElement(*elementName*)
  - Create an HTML element
- *parentNode.*appendChild(*childNode*)
  - Appends the *childNode* as the last child to the *parentNode*
- *parentNode.*removeChild(*childNode*)
  - Removes the *childNode* from the *parentNode*
- *parentNode.*replaceChild(*new, current*)
  - Replace *current* node with *new* node

HTML DOM Events

- A JavaScript can be executed when an event occurs on the web page
- Examples of HTML events:
  - When a user clicks the mouse
  - When a web page has loaded
  - When an image has been loaded
  - When the mouse moves over an element
  - When an input field is changed
  - When an HTML form is submitted

HTML DOM EventListener

- Adding an event listener
  - document.getElementById("myBtn").addEventListener("click", displayDate);
  - document.getElementById("myBtn").onclick = displayDate;
- The addEventListener() method attaches an event handler to the specified element.
- It does not overwrite existing event handlers.
- Many event handlers can be attached to one element

HTML DOM EventListeners

- *element*.addEventListener(*event, function, useCapture*);
  - The first parameter is the type of the event (like "click" or "mousedown").
  - The second parameter is the function we want to call when the event occurs.
  - The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.
    - Default is ‘false’, which means the event bubbling model will be used

Event Propagation Model

- Event Propagation Model
  - Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a element, and the user clicks on the <p> element, which element's ``click'' event should be handled first?
- Event Bubbling
  - In *bubbling,* the inner most element's event is handled first and then the outer element's
- Event Capturing
  - In *capturing,* the outer most element's event is handled first and then the inner element's

BOM

- Browser Object Model
  - http://www.w3schools.com/js/js_window.asp
- The “window” object represents the browser’s window
  - All global JavaScript objects, functions, and variables automatically become members of the window object.
  - Global variables are properties of the window object.
  - Global functions are methods of the window object.
  - The document object (of the HTML DOM) is a property of the window object

window.document.getElementById("header"); and document.getElementById("header"); are same

AJAX

- AJAX
  - Asynchronous JavaScript and XML.
- http://www.w3schools.com/xml/ajax_intro.asp
- The XMLHttpRequest Object
  - All modern browsers support the XMLHttpRequest object (IE5 and IE6 use an ActiveXObject).
  - The XMLHttpRequest object is used to exchange data with a server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

AJAX - Details

- Send a Request to Server
  - open(*method,url,async*)
    - *method*: the type of request: GET or POST
      url: the location of the file on the server
      async: true (asynchronous) or false (synchronous)
- Server Response
  - responseText get the response data as a string
    responseXML get the response data as XML data
- The onreadystatechange event
  - http://www.w3schools.com/ajax/ajax_xmlhttprequest_onreadystatechange.asp

AJAX - Details

- XMLHTTPRequest readyState
  - State Description
  - 0 The request is not initialized
  - 1 The request has been set up
  - 2 The request has been sent
  - 3 The request is in process
  - 4 The request is complete

Same Origin Policy

- http://tools.ietf.org/html/rfc6454

- A JavaScript running on a web browser is able to interact with web resources arising from the *same origin as that of the script*
- Same origin:
  - Two URIs are part of the same origin (i.e., represent the same security principal) if they have the same *scheme*, *host*, and *port*
  - Scheme: http/https

Same Origin Policy

- Following have same origin
  - http://example.com/
  - http://example.com:80/
  - http://example.com/path/file
- Different origin from each other
  - http://example.com/
  - http://example.com:8080/
  - http://www.example.com/
  - https://example.com:80/
  - https://example.com/
  - http://example.org/
  - http://ietf.org/

Same origin policy

- Applies only to AJAX requests
- Does not apply to loading scripts or images
-
- http://stackoverflow.com/questions/5707363/same-origin-policy-and-external-scripts

Frameworks and libraries

- Google's AngularJS
  - MVC framework for JavaScript
  - http://www.w3schools.com/angular/angular_intro.asp
- Facebook's ReactJS
  - Only the "view" layer
  - Has the notion of a virtual DOM; allows partial DOM updates
- Twitter's Bootstrap
  - Framework for web UI development
  - http://www.w3schools.com/bootstrap/
- JQuery
  - Library for building JavaScript based applications

References

- Browser security
  - https://code.google.com/p/browsersec/wiki/Main
- How Ad Servers work?
  - http://www.adopsinsider.com/ad-serving/how-does-ad-serving-work/

Cross-site scripting

- https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

Cloud Computing

Devdatta Kulkarni

What is Cloud Computing?

- On-demand provisioning and management of resources

- *Resources*
  - Compute servers, database instances, object stores, load balancers
    - Compute servers can be virtual machines and/or dedicated hardware
- *Provisioning and management*
  - creation, scaling, deletion
- *On-demand*
  - When needed by application developer
  - When needed by running application
    - Automatic scale up/scale down

How does one consume the Cloud?

- If you are an application developer:
  - You can provision a virtual machine (VM)
    - This is called -- “Host” or “VM”
  - Login
  - Install Tomcat on it
  - Deploy your web application to Tomcat
    - Lets say your web app is available at the context root “myeavesdrop” and tomcat is running on port 8080
  - Adjust firewall rules on the Host to allow traffic to the Host’s IP address and port 8080
  - Your web app will be available at
    - http://<ip-address:8080/myeavesdrop/

Cloud computing categories

- Two basic categories
  - Infrastructure-as-a-Service (IaaS)
  - Platform-as-a-Service (PaaS)

Infrastructure-as-a-Service (IaaS)

- In this model you provision a VM, ssh into it, install application container such as Tomcat, deploy your code
  - Advantage:
    - As an application developer you have complete control over how to setup your application's environment
  - Disadvantage:
    - You have to deal with issues outside of core application development
      - Setting up Tomcat
      - Modifying firewall rules
  - Examples:
    - Commercial
      - Amazon AWS, Google Compute Engine (GCE), Microsoft Azure, IBM Softlayer, Rackspace cloud servers
    - Open source
      - OpenStack, CloudStack

Platform-as-a-Service (PaaS)

- In this model you develop your code locally and push it to the “platform”. The platform takes care of deploying it to an appropriate application container
  - Advantage:
    - You just need to worry about developing your application’s code
    - Appealing model from developers' point of view
  - Disadvantage:
    - Hard to troubleshoot since no access to the VM or the application container
      - Typically, the platforms make logs available via an API
  - Examples
    - Commercial
      - Heroku, Amazon Elastic Beanstalk, Google App Engine (GAE), Redhat OpenShift, Pivotal CloudFoundry, IBM BlueMix
    - Open source
      - OpenStack Solum, Redhat OpenShift, Pivotal CloudFoundry

IaaS Example: AWS EC2

- Deploying web application on Amazon EC2 instance
  - Check Webapp-deployments-to-public-clouds.pdf on Canvas

PaaS Example: Heroku

- Deploying web application on Heroku
  - Check Webapp-deployments-to-public-clouds.pdf on Canvas

OpenStack

What is OpenStack?

- Open source software for creating private and public clouds (http://www.openstack.org/)
  - For instance, using OpenStack it is possible for any organization to provide cloud computing capabilities to its users
    - CS department can install OpenStack and all those who have CS accounts can spin up virtual machines as and when needed -- similar to what you get currently from commercial vendors (Amazon, Azure, Rackspace, etc.)
- Founded in 2010 by Rackspace and NASA
  - Many more companies are involved now
    - http://www.openstack.org/foundation/companies/

OpenStack

- Current release
  - http://www.openstack.org/software/ocata/
- Integrated projects (Main projects)
  - Compute (Nova)
  - Object storage (Swift)
  - Image Service (Glance)
  - Database Service (Trove)
  - Orchestration (Heat)
  - Identity Service (Keystone)
  - Networking (Neutron)
  - Dashboard (Horizon)
  - Block storage (Cinder)
  - Application PaaS (aPaaS) (Solum)

OpenStack: High-level details

- Code
  - written in Python
  - available on github
- Projects are available in the “openstack” organization on Github
    - https://github.com/openstack

OpenStack: Development steps

- Sign the CLA
- Setup code submission and code review hooks on your development machine
  - Gerrit is used for code submission and code reviews
- Clone the project that you want to work on
- Work on your patch
- Test the patch using tox
- Submit the patch for review

OpenStack: Development Resources

- Devstack
  - http://docs.openstack.org/developer/devstack/
- Vagrant
  - https://www.vagrantup.com/
- VirtualBox
  - https://www.virtualbox.org/
- IRC channels
  - https://wiki.openstack.org/wiki/IRC

Contributing to OpenStack

- http://www.slideshare.net/devkulkarni/contributing-to-openstack-59590072

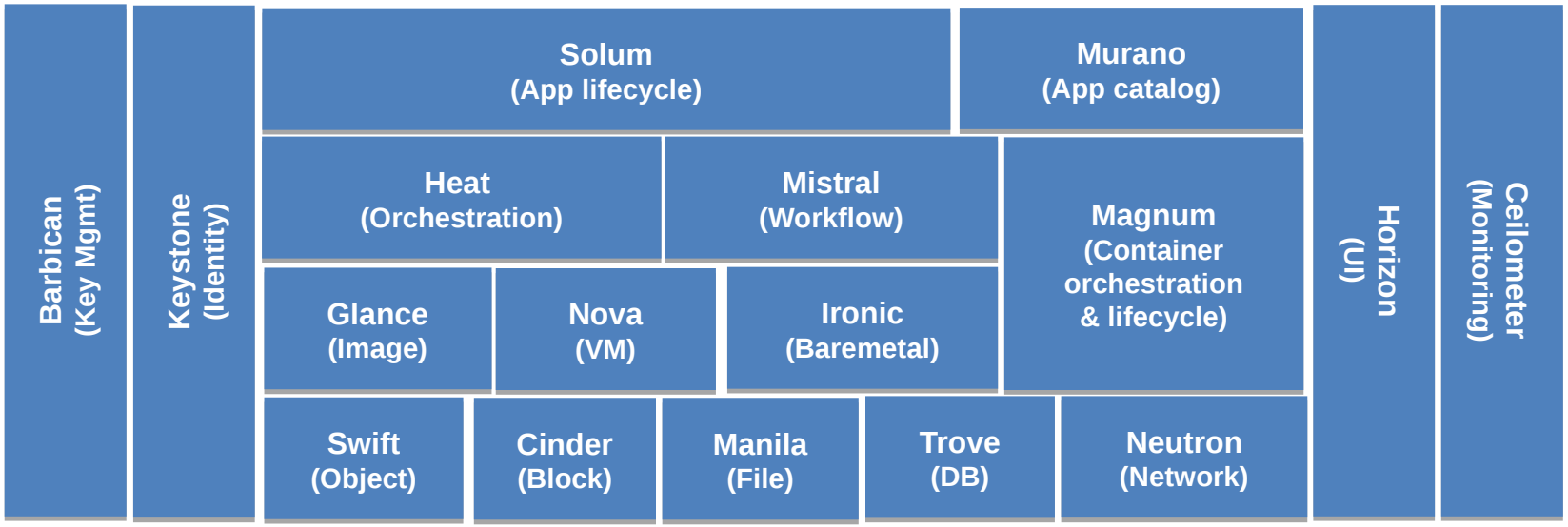
PaaS Example: Solum

- System that supports testing, building, deploying of applications starting from the source code to OpenStack clouds

- Main concepts
  - App
  - Language Pack
  - Workflows

Solum in context with other OpenStack services

Solum - System for building and deploying applications to OpenStack Clouds



Solum: App

- Specifies data associated with an application
  - Github repository of the code
  - Languagepack to be used to build the application
  - Workflow to be associated with the application
    - Unit testing
    - Unit testing and build
    - Unit testing, build, and deploy
  - Commands
    - To run/start the application
    - To unit test the application

Solum: Languagepack

- It is the environment for running/testing the application
- It includes
  - Compilers (e.g.: javac)
  - Runtime libraries (e.g.: jre, jars)
  - Unit testing libraries (e.g.: tox)
- Built as a Docker (https://www.docker.com/) image

Solum Resources

- Solum Overview
  - https://drive.google.com/file/d/0BxmgNoI8UMz5elk4ZXdhXzZiRWM/view
- Solum Demo
  - https://drive.google.com/file/d/0BxmgNoI8UMz5NXpaTFQzYXYwUzA/view

https://wiki.openstack.org/wiki/Solum#Resources

http://docs.openstack.org/developer/solum/getting_started/

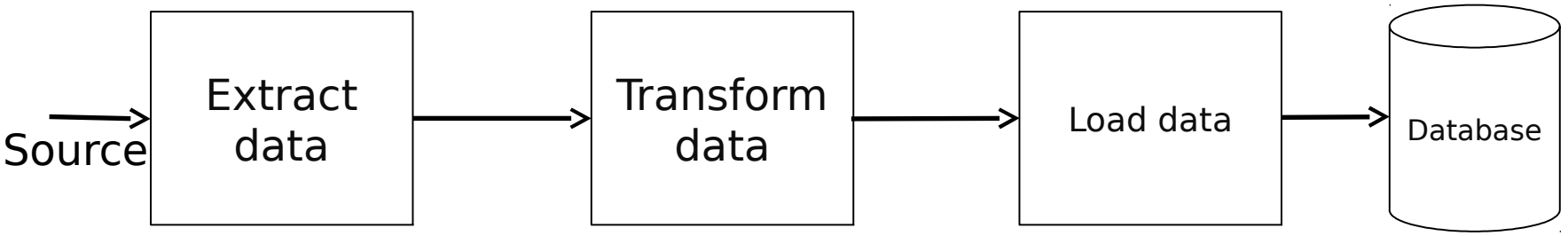
ETL (Extract-Transfer-Load)

Devdatta kulkarni

ETL

- ETL (Extract, Transform, Load)
  - Extract
    - Read the data from one or more sources
  - Transform
    - Transform the data
      - There could be series of transformations
  - Load
    - Load the data into the target system
      - Database
      - Workflow system

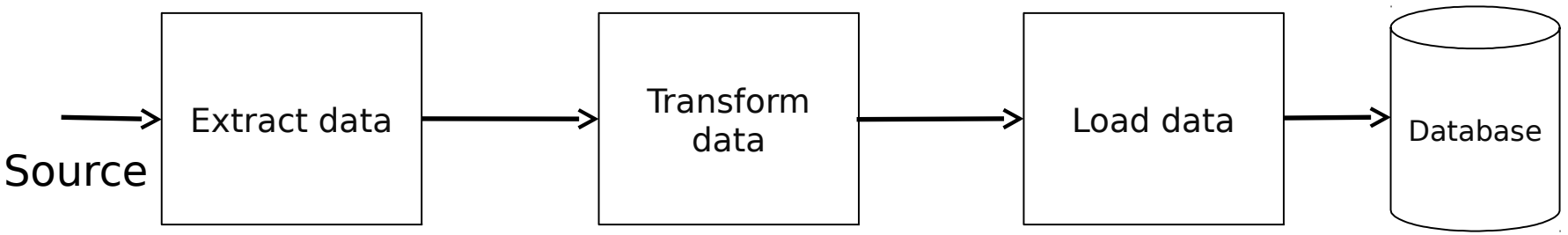
ETL system: Conceptual architecture



(XML Feeds (Atom, RSS))
REST APIs
Databases
Websites

Example

Search emails from particular user on openstack-dev mailing list

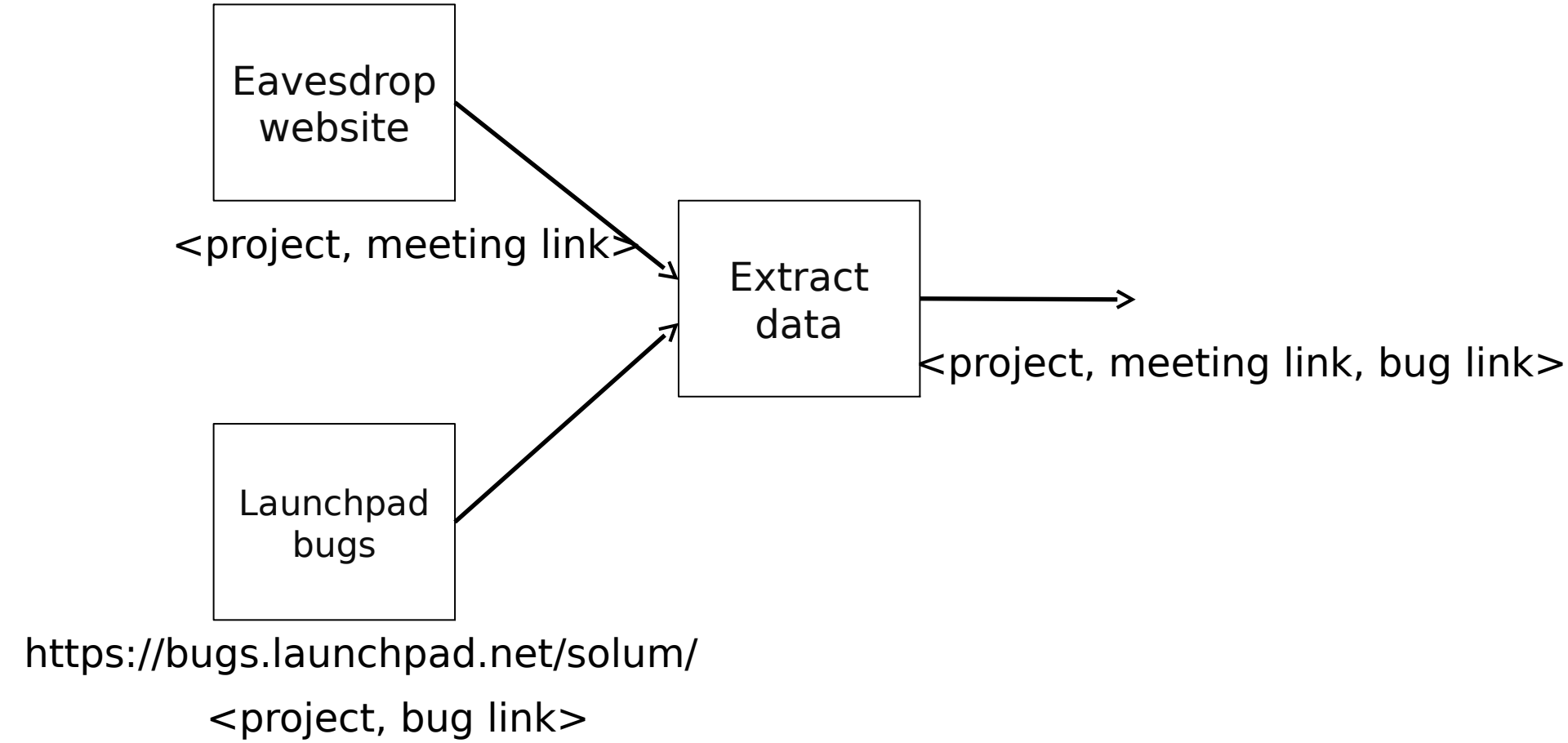


Source

openstack-dev mailing list archieve

Loading from different sources

http://eavesdrop.openstack.org/meetings/solum_team_meeting/



Issues in designing ETL systems

- Continually updating source
  - Need to periodically poll; maintain “last-queried-pointer”
- Load from different sources
- Merge data
  - Concurrency
    - Upsert
      - Insert or Update
- Target data representation
- Transformations
- High throughput
  - Using JDBC vs Hibernate (Object/Relational Mapping)

Target data representation

- Single row, multiple columns

Project	Meeting link	Bug link
Solum	<meeting link>	<bug link>

- Multiple rows, multiple columns

Project	Meeting link	Bug link
Solum	<meeting link>	-
Solum	-	<bug link>

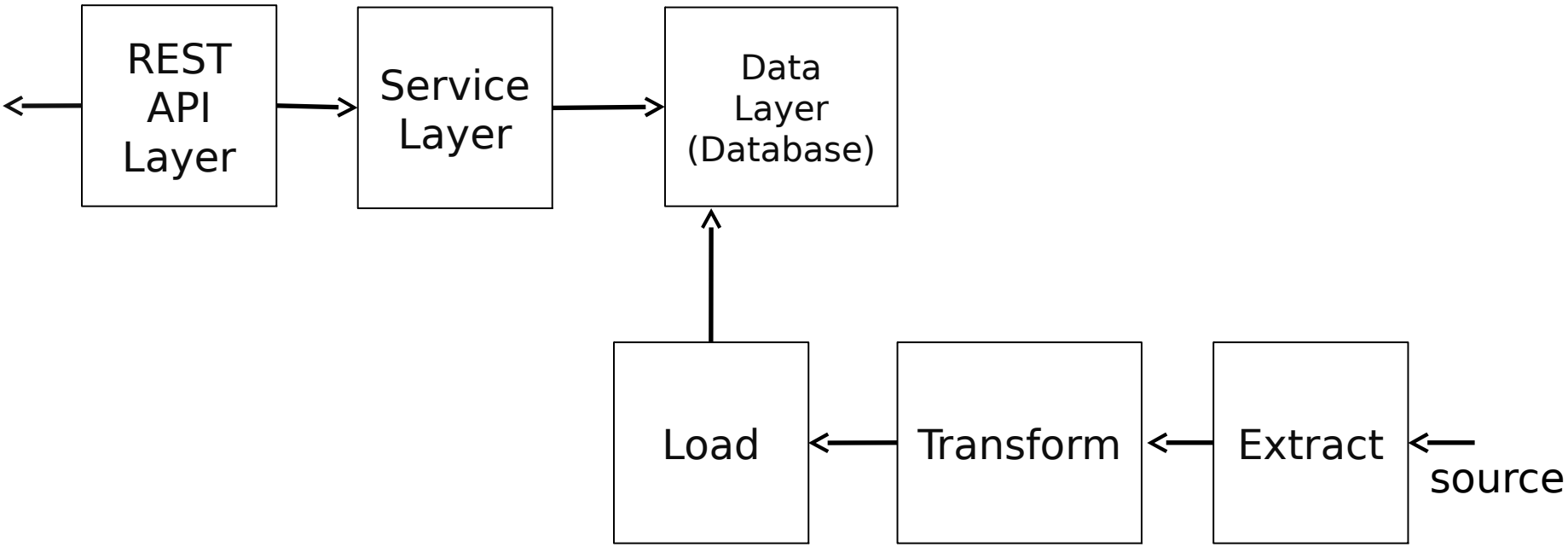
Target data representation

- Single row, multiple columns
  - Pro:
    - Logic on the retrieval side is easier
      - E.g.: GET /projects/solum needs to query only single row from the target table
    - Data storage requirements proportional to number of entities (projects) in the system
  - Con:
    - Logic on the insert side is complex
      - Need to use ‘Upserts’ (Insert or Update)
        - » Why?
      - Prone to race condition issues

Target data representation

- Multiple rows, multiple columns
  - Pro:
    - Logic on the insert side is easier
      - Every external representation of an entity is *inserted* as a separate row into the table
  - Con:
    - Logic on the retrieval side is complex
      - Need to write complex joins
      - Storage is proportional to product of number of entities and number of updates to them
        - » Could be huge

ETL + REST


REST
API
Layer
Service
Layer
Data
Layer
(Database)
Load
Transform
Extract
source

CS 378: Modern Web Applications
University of Texas at Austin
Instructor: Devdatta Kulkarni

Deploying web app on AWS EC2 (IaaS approach)

1) Create AWS account by signing up at https://aws.amazon.com/console/
- You will be required to provide credit card details. You will be charged only if usage of AWS resources goes beyond specified limits for the free tier

2) Create an EC2 instance using Ubuntu 14.04 image.

3) Create a security group for the EC2 instance to allow traffic to port 8080

4) Download the SSH private key to your local machine

5) Get the Public DNS of the EC2 instance and login to it using command of the following form:
ssh -i <Private-key-file.pem> ubuntu@<Public-DNS-of-EC2-Instance>

6) Once logged in, install required software packages on the instance:
- sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install git maven
- wget http://www-eu.apache.org/dist/tomcat/tomcat-8/v8.5.8/bin/apache-tomcat-8.5.8.tar.gz
- gunzip apache-tomcat-8.5.8.tar.gz
- tar -xvf apache-tomcat-8.5.8.tar
- ./apache-tomcat-8.5.8/bin/startup.sh
- sudo apt-add-repository ppa:webupd8team/java
- sudo apt-get update
- sudo apt-get install oracle-java8-installer

7) Clone ModernWebApps repo
- https://github.com/devdattakulkarni/ModernWebApps.git

8) Build js-example
- cd ModernWebApps/JavaScript-Example/js-example
- mvn clean compile
- mvn package war:war

9) Deploy js-example
- cp target/js-example.war ~/apache-tomcat-8.5.8/webapps/.

10) Access the application at:
http://<Public-IP-Address-of-EC2-Instance>/js-example/ajaxexample.html
http://<Public-IP-Address-of-EC2-Instance>/js-example/test.html

Deploying web app on Heroku (PaaS approach)

1. Signup for a Heroku Account
- https://www.heroku.com/

2. Install Heroku CLI
- https://devcenter.heroku.com/articles/heroku-command-line

3. Copy Procfile, pom.xml, system.properties, and ajaxexample.html from Heroku folder inside JavaScript-Example to js-example/
- Copy Procfile, pom.xml, and system.properties in js-example/
- Copy ajaxexample.html in js-example/

4. Copy js-example to a temporary folder
- cp js-example /tmp/.

5. Switch to the js-example folder in the temporary location
- cd /tmp/js-example

6. Setup up application on Heroku
- heroku create <unique-app-name>

7. Initialize the application with source code
- git init
- git add .
- git commit -m “Deploying app to Heroku”

8. Deploy to Heroku
- git push heroku master

9. Test the application
- https://<unique-app-name>.herokuapp.com/test.html
- https://<unique-app-name>.herokuapp.com/ajaxexample.html

References:
- https://devcenter.heroku.com/articles/deploy-a-java-web-application-that-launches-with-jetty-runner

Deploying web app on AWS Elastic Beanstalk (PaaS approach)

1) Create AWS account by signing up at https://aws.amazon.com/console/
- You will be required to provide credit card details. You will be charged only if usage of AWS resources goes beyond specified limits for the free tier

2) Go to the folder of js-example

3) Create war file
- mvn package war:war

4) Copy the war file to ROOT.war
- cd target
- cp js-example.war ROOT.war

5) Go to the ElasticBeanstalk console

6) Create application
- Choose Tomcat as the application platform
- Choose the option to upload ROOT.war
- Choose the default options for the rest of the options
- Deploy the application

7) Once the application is READY, you will see the application box on the web console becomes green.

8) Access the application url and the html files at:
   http://<app-name>-<region>.elasticbeanstalk.com/test.html
   http://<app-name>-<region>.elasticbeanstalk.com/ajaxexample.html

References:
http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html
http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_Java.html
http://beanstalker.ingenieux.com.br/beanstalk-maven-plugin/usage.html

Deploying web app on Google App Engine (PaaS approach)

References:
https://cloud.google.com/appengine/docs/java/tools/maven