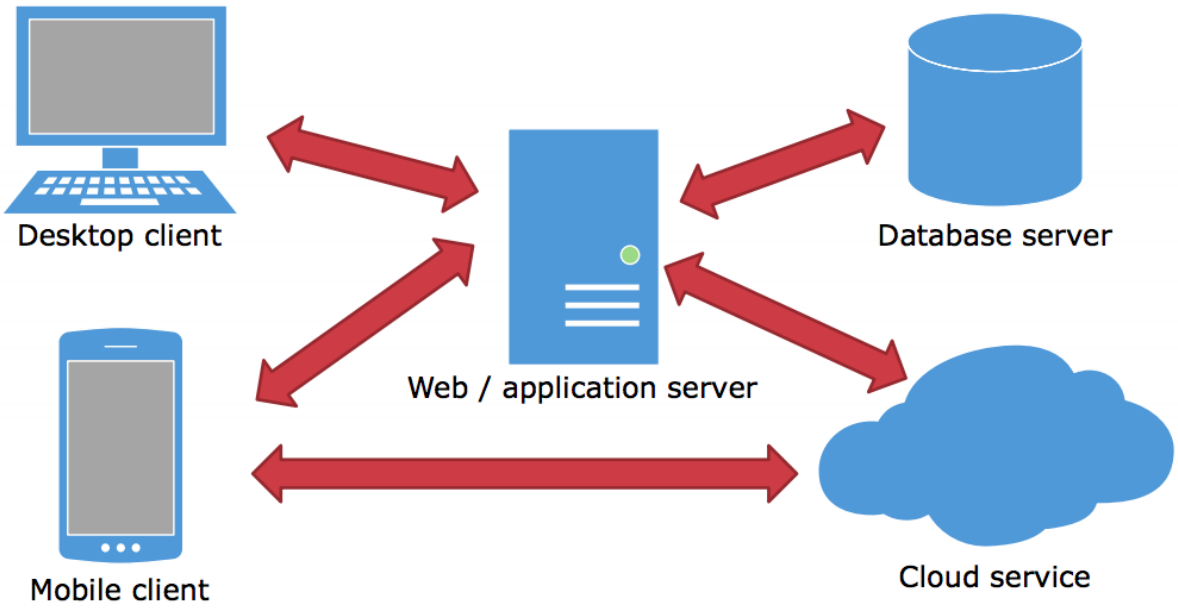
SOFTENG 325 Journal

Table of Contents

Distributed Systems

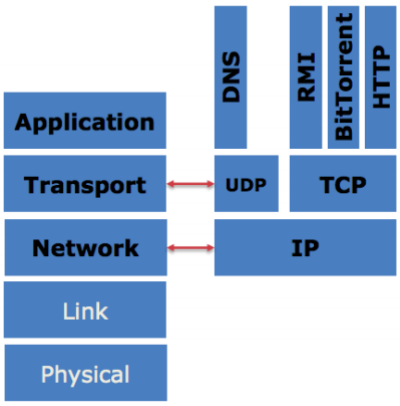


Networking Infrastructure

Networking infrastructure exhibits the following characteristics:

- Computers and links can fail independently.
- Switches have finite space for storing packets.
- Individual links vary in terms of bandwidth capacity.
- Data can be corrupted during transmission.
- Switches store routing tables that they dynamically update based on knowledge of congested links and failed switches.

Network Protocols

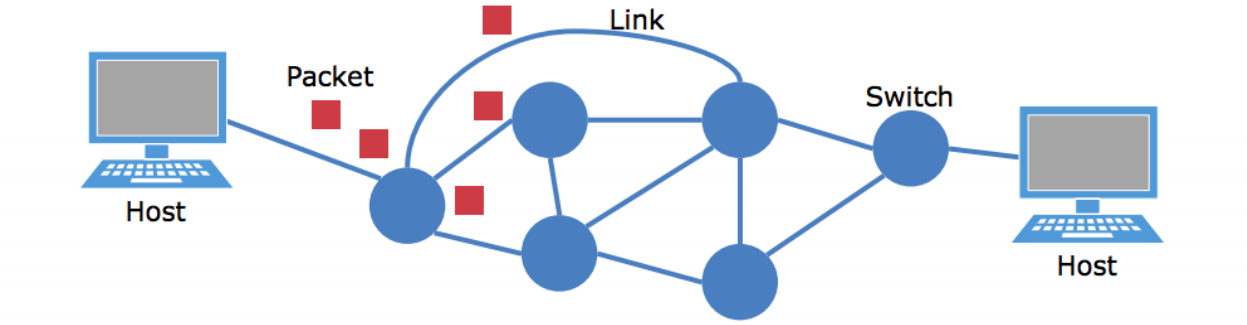


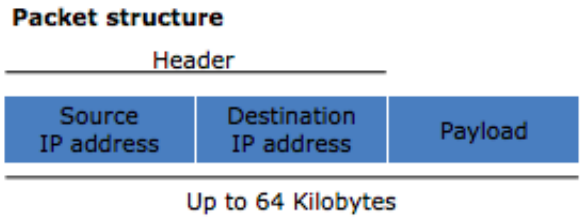
Network protocols are organized s:

- Application layer protocols address the needs of particular applications.
- A layer exposes an interface to the layer above.
- Layers hide their implementations; One implementation can be substituted for another.
- Higher layers can add reliability to unreliable lower layers.
- Transport protocols provide for process-to-process communication.
- The network layer provides a packet delivery service between host machines.

Higher-level protocols use the services of the layer directly beneath them – A layer depends on the interface of its underlying layer and not its implementation.

Internet Protocol (IP)

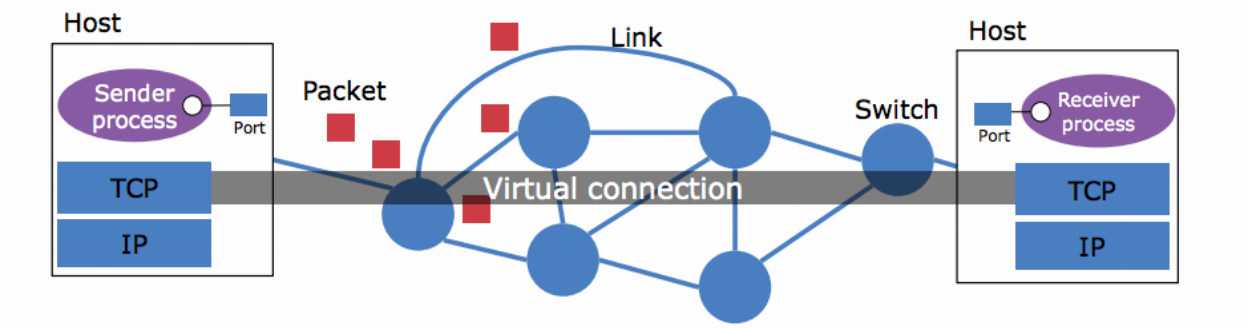




The Internet Protocol moves packets through the network to their destination - identified by the packet's destination address.

- Unreliable and can lead to packets being dropped, corrupted, arriving out of sender order and duplicated.

Transmission Control Protocol (TCP)



TCP (Transmission Control Protocol) is a transport protocol that establishes a **virtual connection** between a pair of processes - a bi-directional **stream** abstraction that hides several network characteristics:

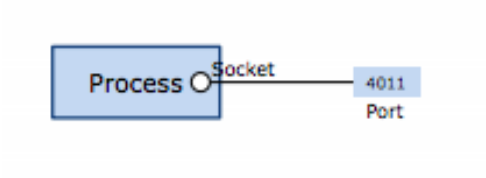
- **Message sizes / boundaries:** The application simply reads and writes data from / to the stream - the TCP layer decides how much data to accumulate in the sender before it creates packet(s) and passes them down to the IP layer.
- **Message destinations:** Once a stream has been established, the "connected" processes can use the stream without knowledge of ports and IP addresses.
- **Lost messages**
- **Message duplication and ordering**
- **Flow control**

Issue	TCP behaviour
Validity	Lost packets are detected and resent.
Integrity	A mandatory checksum is used to transform a corrupt packet into a lost packet.
Ordering	Transmitted data is processed so that once received, it is delivered in the order in which it was sent; Each packet has a sequence number.
Blocking	The Sender can be blocked inserting data into an output stream; The receiver blocks if the input stream has insufficient data.

TCP with Java

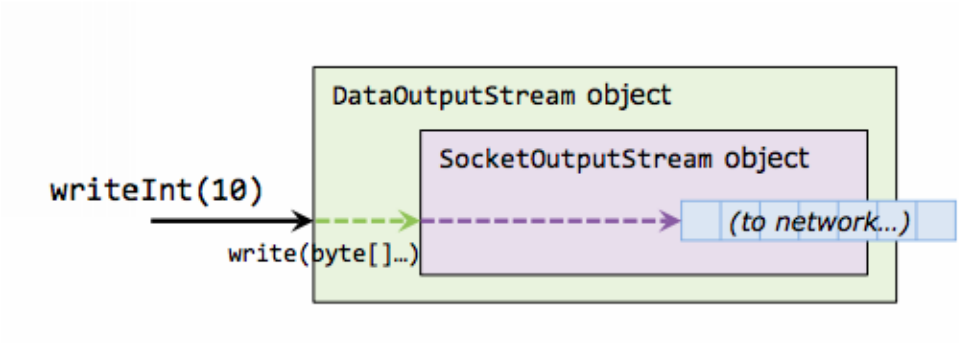
Using sockets

- To bind a process to a port number, a `Socket` is used
- For TCP, Java provides classes `Socket` and `ServerSocket`.
    - These provide methods for establishing connections and acquiring I/O streams.

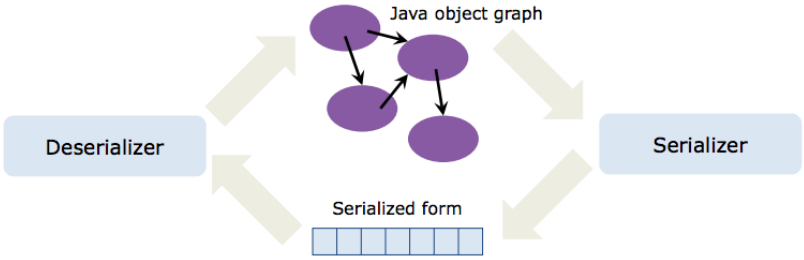


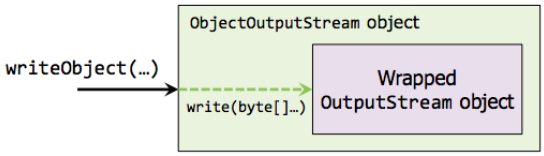
Preparing data

- Data is ultimately sent in byte form, but it is convenient to work with meaningful data types.
- Classes `DataInputStream` and `DataOutputStream` are useful for working with primitive data types.



Java Serialization

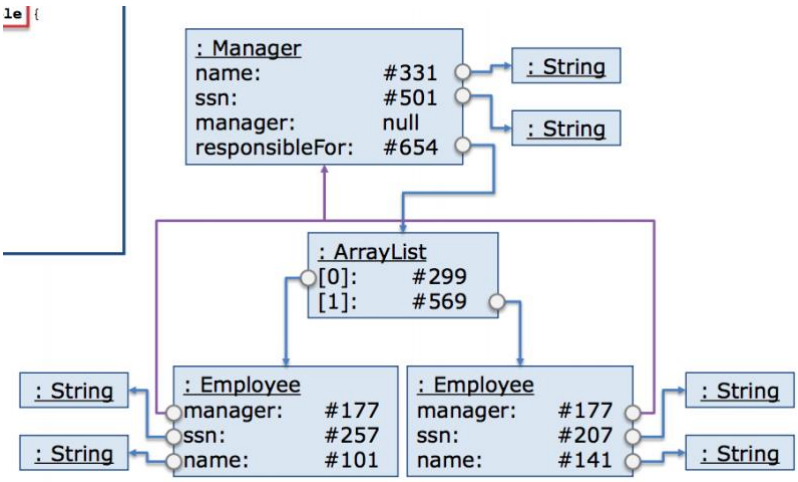




Example:

```
public class Employee implements Serializable {
      protected String name;
      protected String ssn;
      protected String manager;
      ...
}
public class Manager implements Employee
{
      private List<Employee> responsibleFor;
}
```

Address	Contents
#101	String object
#141	String object
#177	Manager object
#207	String object
#257	String object
#299	Employee object
#331	String object
#501	String object
#569	Employee object
#654	ArrayList

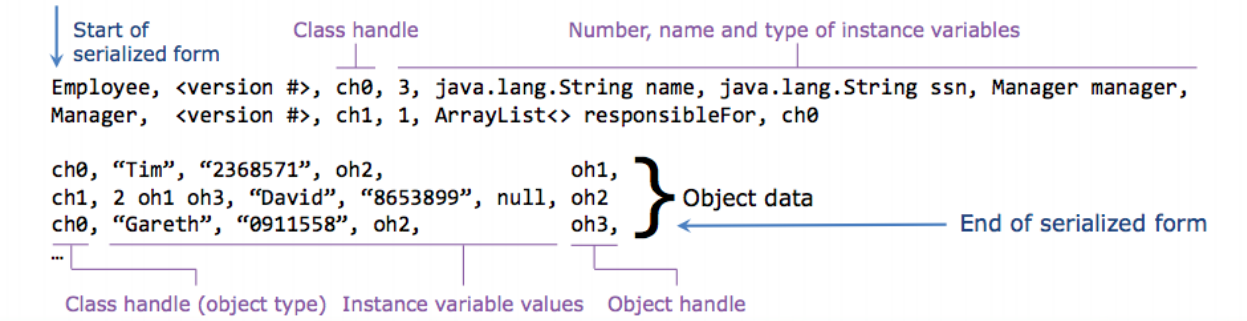


When serializing an object O, the following information is written in binary form:

- O's state - the values of O's instance variables.
- The state of all objects - once only - that are reachable from O.
- A description of each class, and its superclasses, of objects being written.

Example

```
Manager mgr = new Manager("David", "8653899");
Employee e1 = new Employee("Tim", "236571", mgr);
Employee e2 = new Employee("Gareth", "0911558", mgr);
```



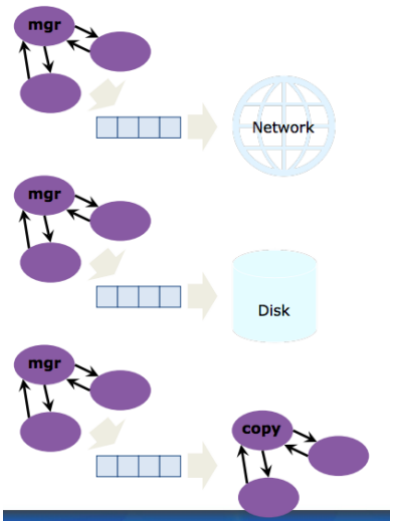
Applications of Serialization

1. Sending an object structure over a network connection

```
Manager mgr = …;
Socket socket = …;

ObjectOutputStream out =
      new ObjectOutputStream(
      socket.getOutputStream());
out.writeObject(mgr);
```

2. Persisting an object graph to disk

```
Manager mgr = …;

OutputStream file = new
      FileOutputStream("employees.ser");
ObjectOutputStream out = new
      ObjectOutputStream(file);
out.writeObject(mgr);
```

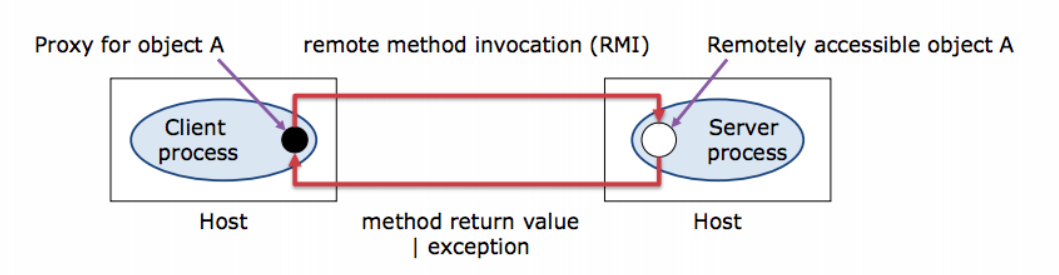


3. Making a deep copy of an object graph in memory

```
Manager mgr = …;

ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bos);
out.writeObject(mgr);

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream in = new ObjectInputStream(bis);
Employee copy = (Employee) in.readObject();
```

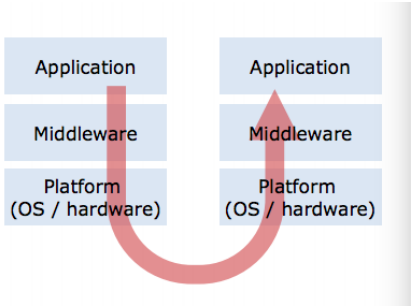
Distributed Objects & RMI

- Distributed object applications extend the familiar object-oriented programming model to the network.
- Remote objects:
    - Can be invoked by Java programs on different machines
    - Are represented by local proxy objects that implement the same interfaces as the remote objects they represent.



Middleware

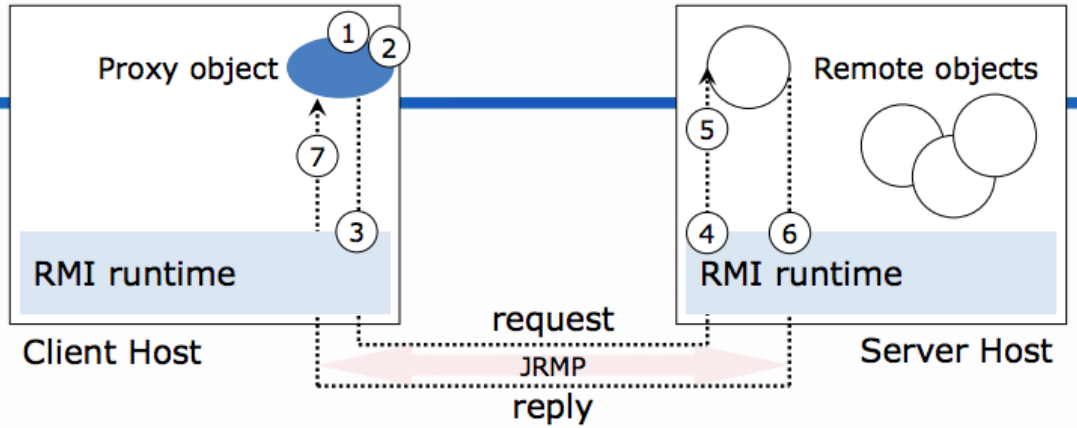
- Middleware facilitates and manages interaction between applications, and provides:
    - A programming abstraction that hides some of the complexity associated with distribution; Simplifies application development as developers can focus on application logic and not networking complexity.
    - Infrastructure to implement the programming abstraction.
    - A means to mask heterogeneity.



- Key benefits of middleware include reuse, interoperability and portability.
- Helps manage resources efficiently

Java RMI (Remote Method Invocation)

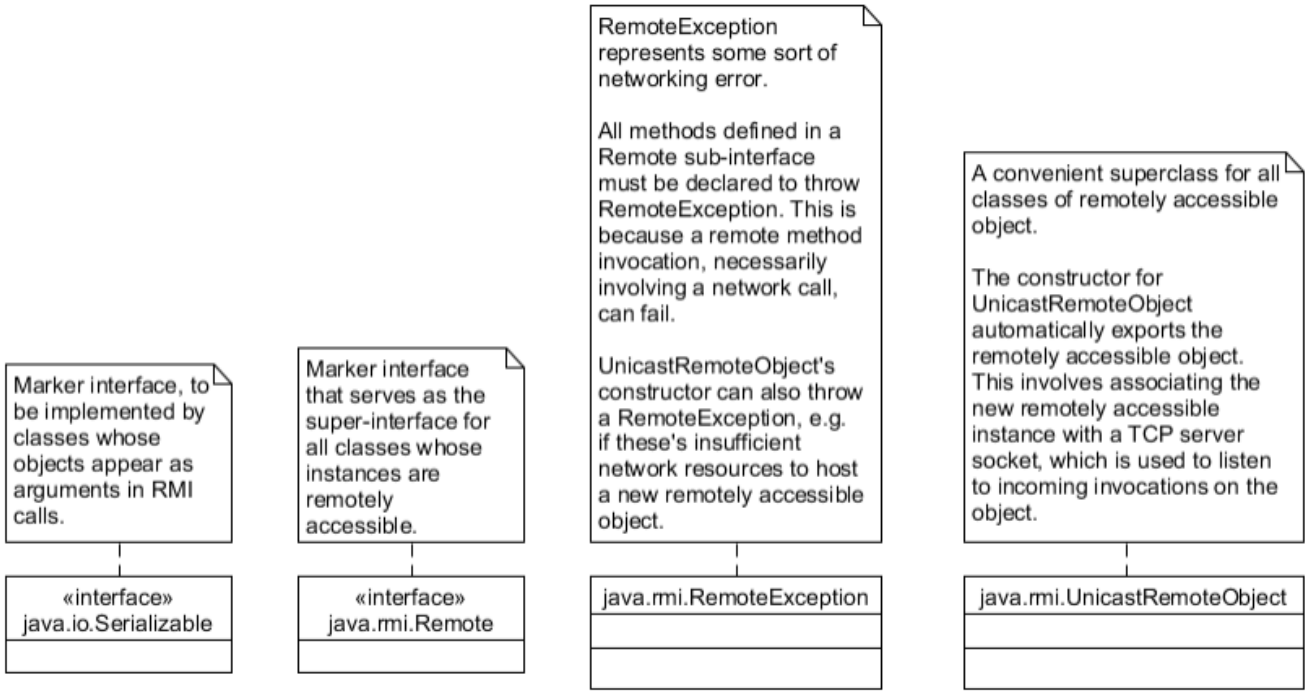
- Java RMI is an example of distributed object middleware; It provides the familiar object-oriented programming abstraction.
- Java RMI masks distribution concerns by:
    - Aims to make remote method invocation as simple as invoking a method on a local object; Provides object-invocation programming abstraction but cannot completely mask a distributed environment.
    - Parameter passing and invocation semantics are different for remote invocations, object discovery is required in a distributed environment, and Remote interfaces are necessitated for remotely accessible objects.
    - Generating proxy classes on demand, at run-time.
    - Providing a Naming service that allows proxy instances to be registered and retrieved.
    - Implementing a request-reply protocol layered on top of TCP.
    - Providing resource management covering sockets, threads, and memory



1. Client application invokes method on proxy object
2. Proxy serializes method arguments
3. Proxy passes request to client-side run-time, which creates a JRMP message identifying the remote object and describing the method to invoke. Client-side run-time then acquires a TCP socket connected to the server and sends the request message.
4. Server-side run-time locates the object whose method is to be invoked.
5. The server-side runtime deserializes any arguments and executes the method
6. Method return value (or exception) is serialized and sent to the client
7. Return value / exception is deserialized; proxy method returns control to client thread with the result

A proxy object contains sufficient information to identify a remote object. Minimally, this includes the IP address of the server host, the port number of the server process in which the remote object resides, and an ID to identify the remote object within the server process.

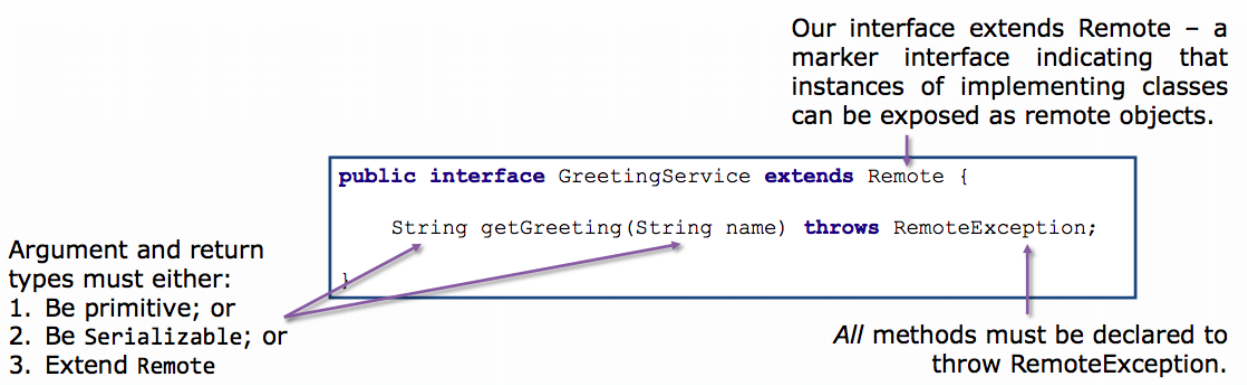
Java RMI Key Entities



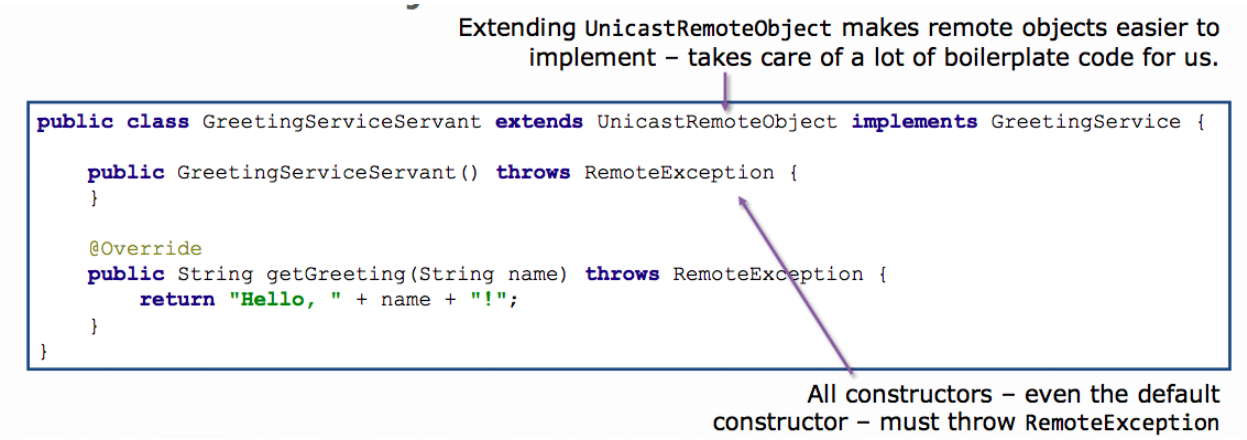
A "Hello World" Example

- Build an RMI application allowing clients to get customized greetings from the server.

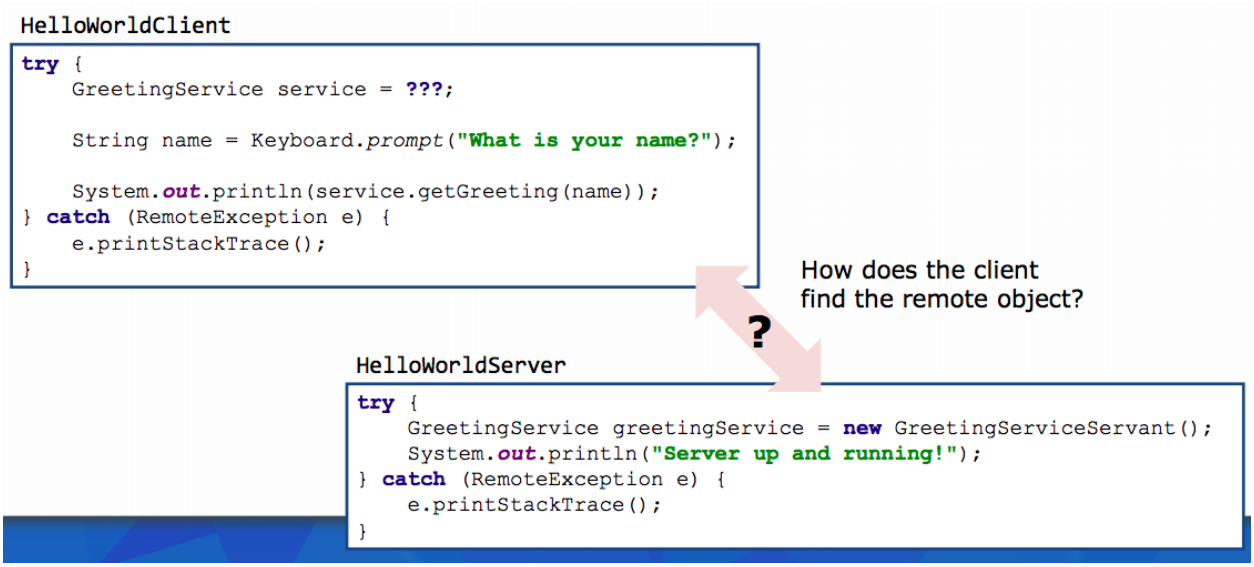
1. Define the *interface* for the remote object(s):



2. Implement the interface to create our remote object class itself:



3. Create the client and server.



Object Discovery

- Somehow, clients need to obtain proxy objects so that they can invoke methods of remote objects.
- Distributed object middleware typically offers a Naming service to store proxies.
    - Servers register proxy objects with the Naming service.
    - Clients lookup and retrieve proxy objects.
- Java RMI provides a simple white-pages style naming service called the Registry.

HelloWorldServer

```
try {
    GreetingService greetingService = new GreetingServiceServant();

    Registry lookupService = LocateRegistry.createRegistry(8080);
    lookupService.rebind("greetingService", greetingService);

    System.out.println("Server up and running!");
} catch (RemoteException e) {
    e.printStackTrace();
}
```

Creates a lookup service listening on the given port, and registers the remote object with it.

HelloWorldClient

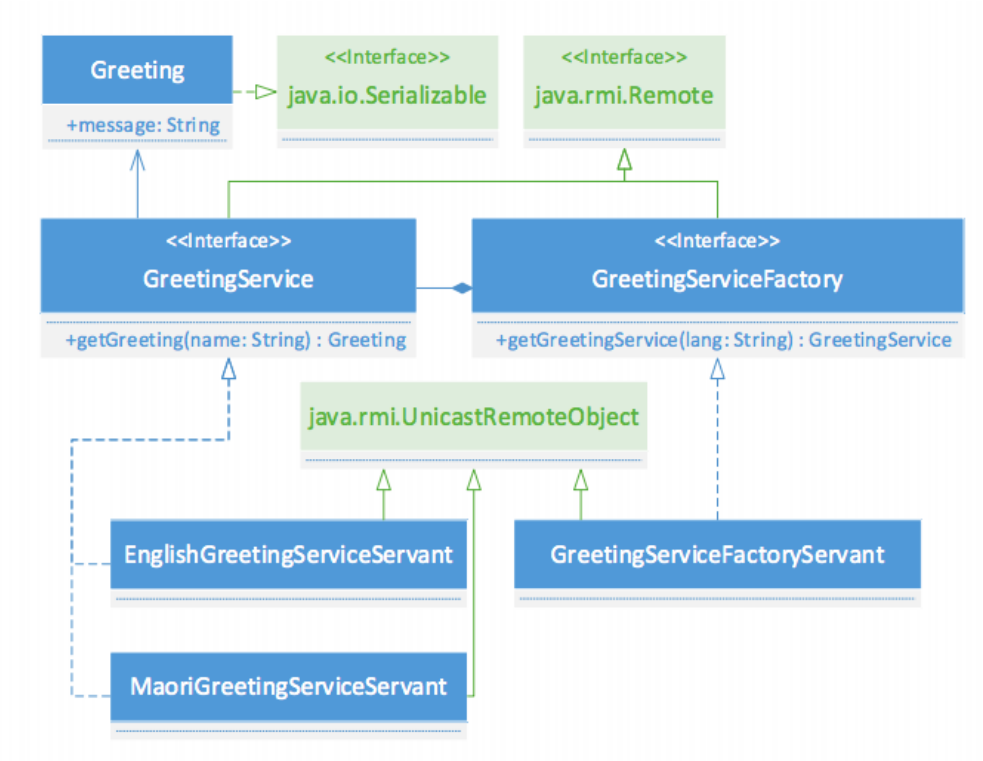
```
try {
    Registry lookupService = LocateRegistry.getRegistry("localhost", 8080);
    GreetingService service = (GreetingService) lookupService.lookup("greetingService");

    String name = Keyboard.prompt("What is your name?");

    System.out.println(service.getGreeting(name));
} catch (RemoteException e) {
    e.printStackTrace();
} catch (NotBoundException e) {
    e.printStackTrace();
}
```

Thrown if an object with the given name is not found.

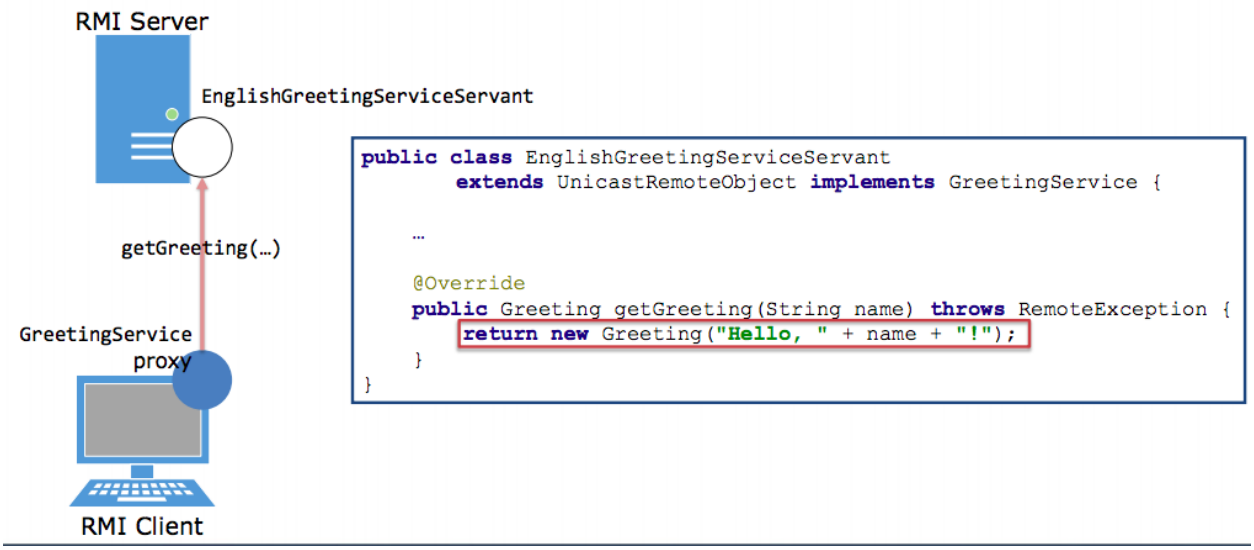
Gets the lookup service listening on the given host / port, and finds the remote object with the given name.

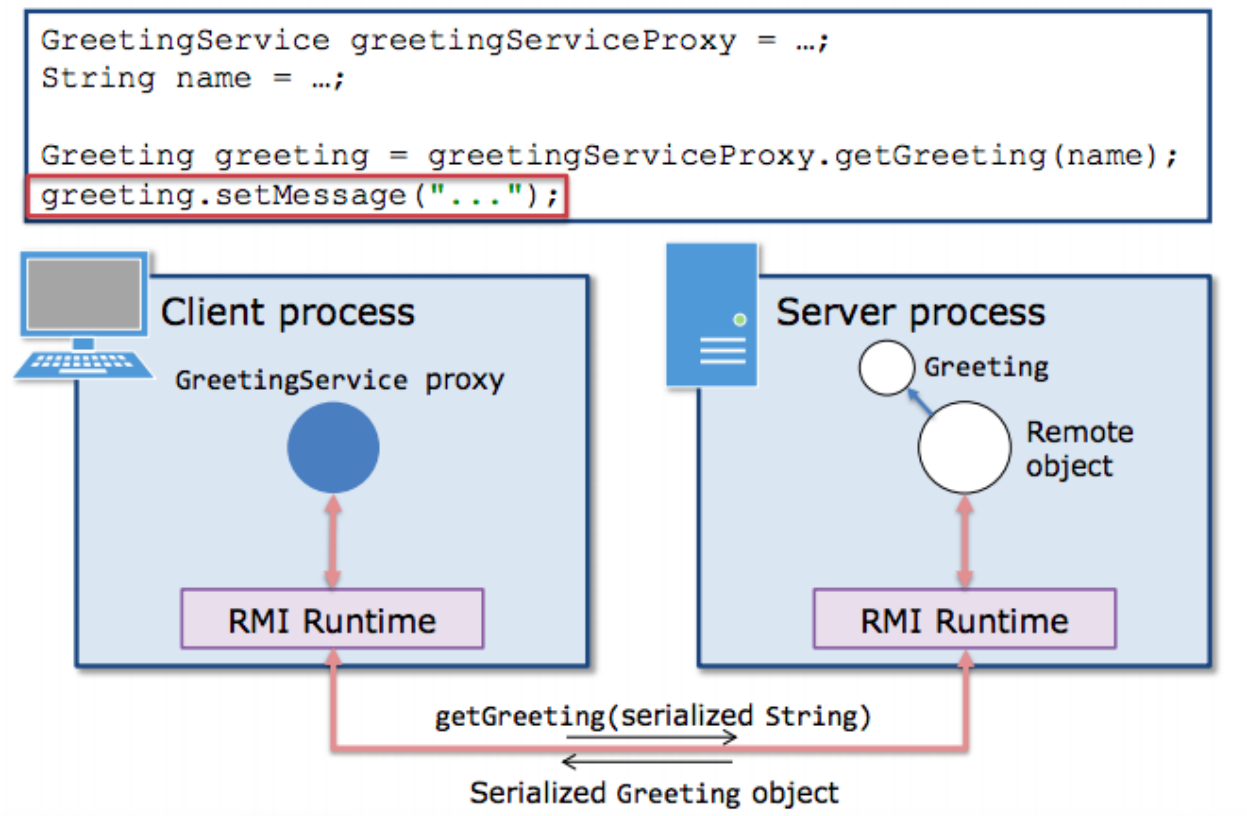
Multilingual Support Example

- Let's extend our example to include multilingual support.
    - Each language will be handled by a different remote object.
    - We will have a remote `GreetingServiceFactory` object to return the correct `GreetingService` based on the language choice.



Parameter Passing – Serializable Data

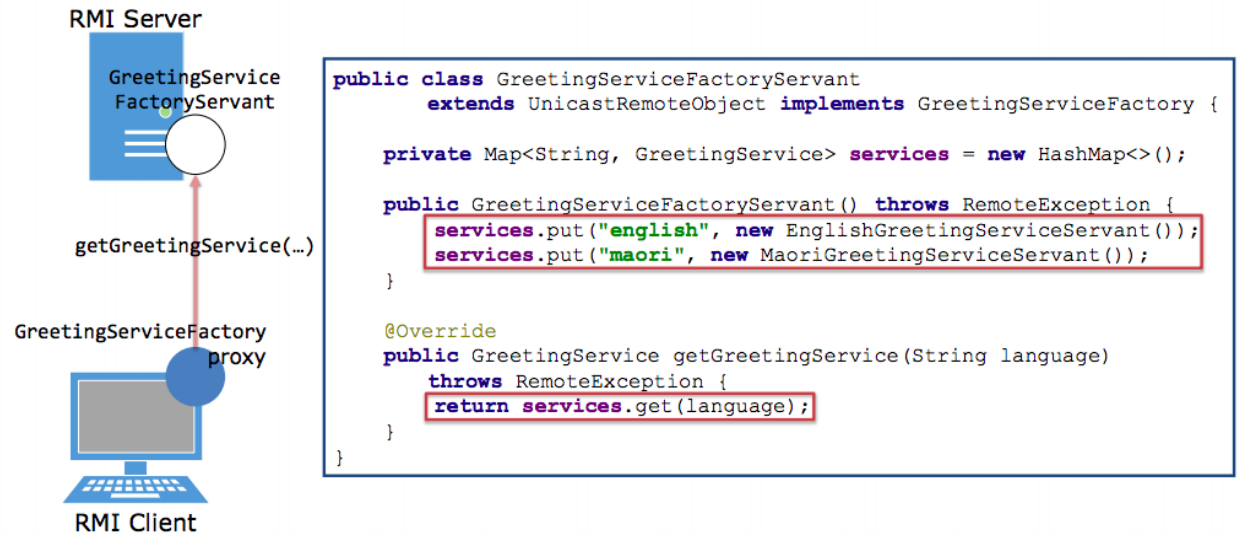




The serialization mechanism:

- Serializes the `Greeting` object in the server.
- Sends the object in its serialized form to the client.
- Deserializes the `Greeting` in the client JVM, creating a local copy.

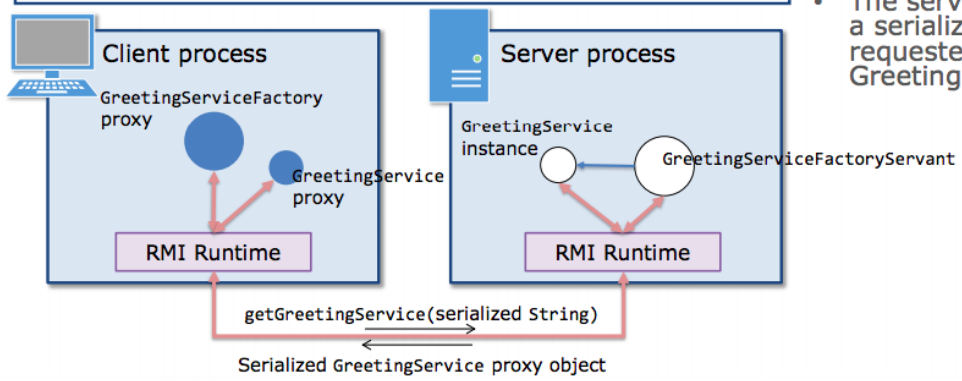
Parameter Passing – Remote Objects




```
public class GreetingServiceFactoryServant
        extends UnicastRemoteObject implements GreetingServiceFactory {

    private Map<String, GreetingService> services = new HashMap<>();

    public GreetingServiceFactoryServant() throws RemoteException {
        services.put("english", new EnglishGreetingServiceServant());
        services.put("maori", new MaoriGreetingServiceServant());
    }

    @Override
    public GreetingService getGreetingService(String language)
        throws RemoteException {
        return services.get(language);
    }
}
```

```
GreetingServiceFactory greetingFactoryProxy …;
String language = …;
GreetingService greetingServiceProxy =
        greetingFactoryProxy.getGreetingService(language);
String name = …;
Greeting greeting = greetingServiceProxy.getGreeting(name);
```

- The serialization mechanism serializes the String parameter on the client and sends it to the server
- The server then returns a serialized *proxy* to the requested remote GreetingSrervice object.

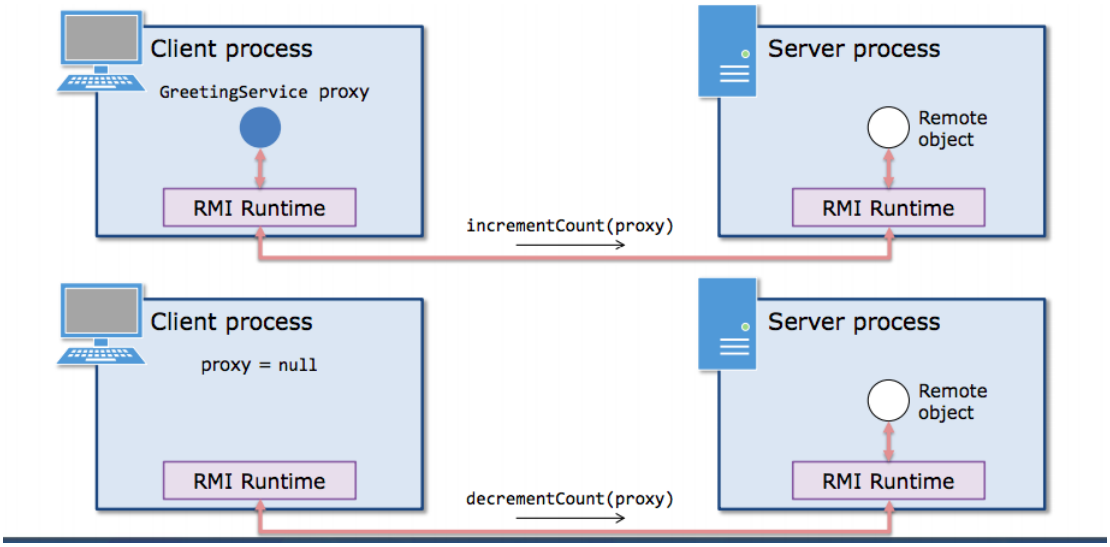


Resource Management

The RMI middleware is responsible for managing resources efficiently; Resources include:

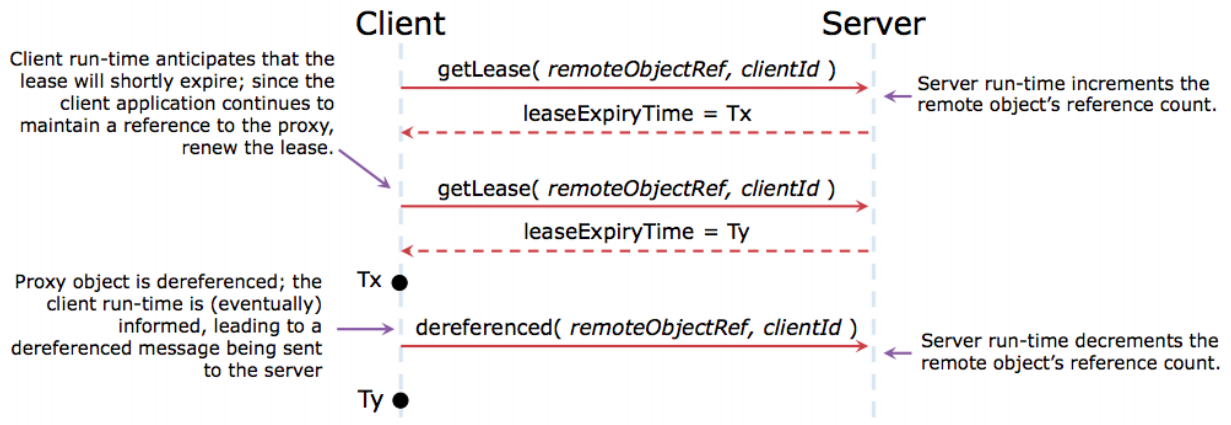
- Memory
  - Just like memory is reclaimed when a local object is no longer referenced, memory used by unreachable remotely accessible objects should be reclaimed.
  - There is a need for distributed garbage collection.
- Sockets and network resources
  - Sockets and network connections are expensive to establish and maintain.
  - The middleware should carefully manage sockets.
- Threads

A solution:



But what if errors occur when incrementing/decrementing?

An alternative (the real solution:



This solution helps to prevent resource starvation on servers. The dereferenced message need not arrive at the server – in this case, the server would decrement the remote object's reference count at time Ty.

Socket Sharing

- It would be grossly inefficient for each proxy object to maintain a dedicated socket connection.
- Instead, when a proxy's method is called, the middleware uses an existing socket connection to the server (if one exists and isn't in use); Otherwise a new connection is established.

Invocation Semantics

- With a method call on a local object, the method is executed exactly once.
- RMI middleware cannot guarantee exactly-once semantics, and typically offers at-most-once semantics.
    - Where an RMI call returns successfully (i.e. it doesn't throw a `RemoteException`), the remote method was executed once.
    - In the event of a `RemoteException`, the remote method may or may not have been executed.

Service-Oriented Architecture

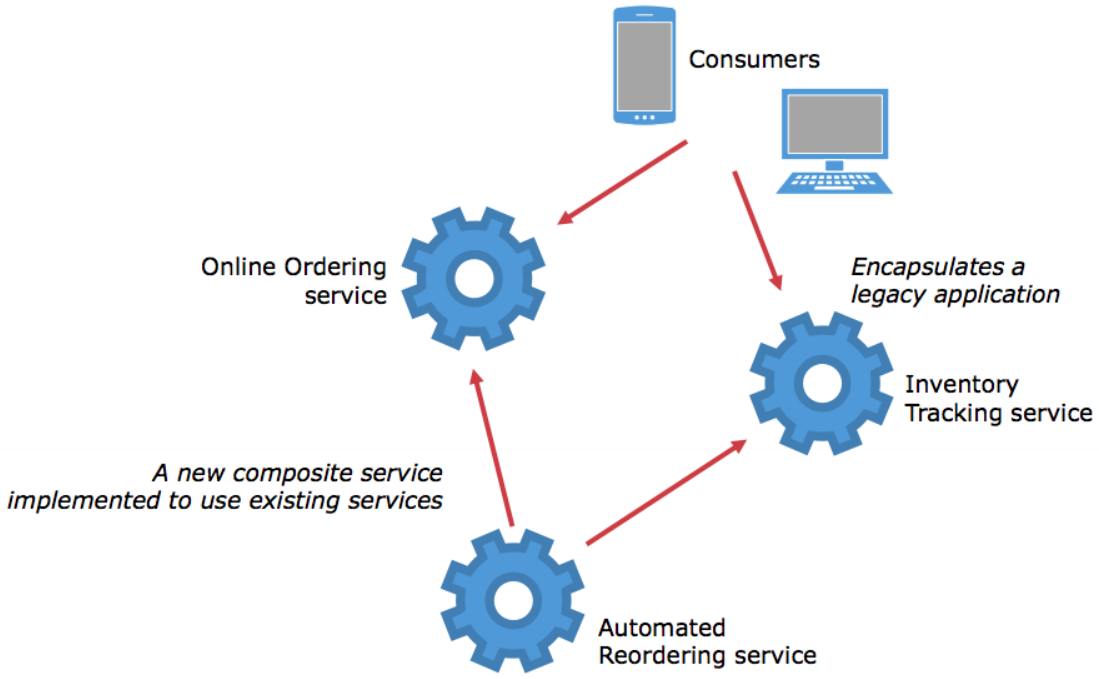
Service-oriented architectures (SOA) are distributed systems made up of software units (services). With SOA, consumers can discover and interact with services, without regard for the technologies used to implement individual services. SOA applications often cross organizational boundaries.

Service characteristics

- Distributed
- Coarse-grained units of reuse
- Well-defined interfaces, hidden implementations
- Technology independent
- Loosely coupled
- Discoverable
- Composable

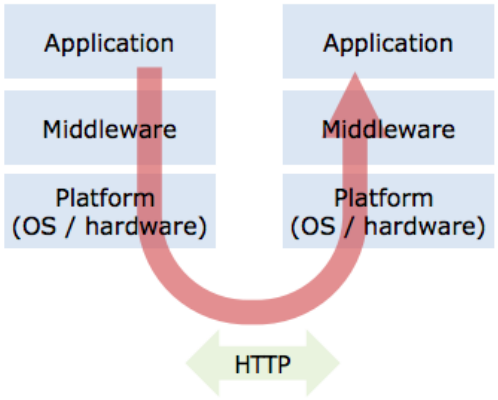
Why use SAO?

- Reduce IT costs
- Reduce time to market
- Agility
- Leverage legacy systems



Interoperability in the Presence of Heterogeneity

- Interoperability: The ability of computer systems or software to exchange and make use of information between each other.
- Heterogeneity: Software systems made up of different languages / running on different systems / using different standards.
- Interoperability necessitates use of a common communication protocol.
- HTTP is an **open** and **standardized** protocol.
- HTTP is a text-based protocol.
    - The character content of HTTP messages can be encoded in an agreed way, e.g. UTF-8 in practice.
    - Middleware converts UTF-8 encoded data to and from native formats.
- Web services use HTTP.



HTTP

- HTTP defines two messages:
    - Request

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//http://www.bbc.com/news	HTTP/ 1.1		

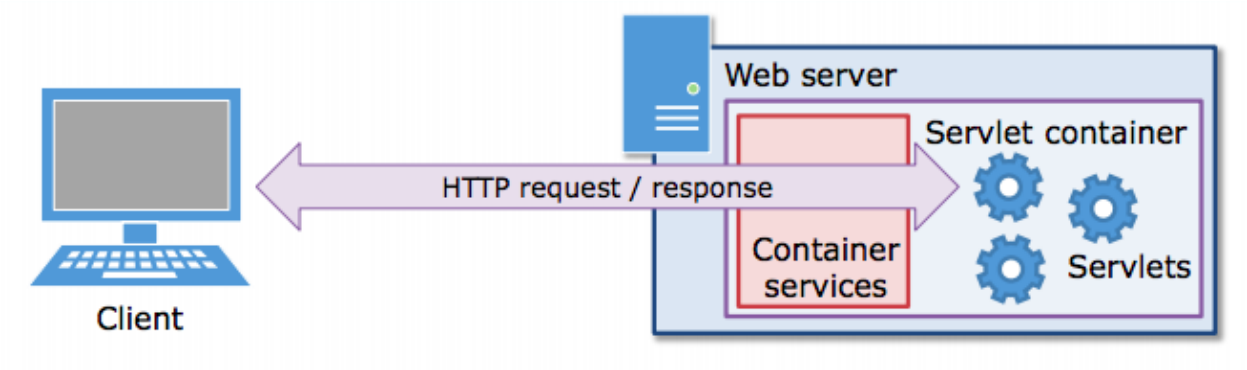
    - Reply

HTTP version	*status*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		

- URL syntax:
    - http:// serverName [:port] [/pathName] [?query]

Servlets and Servlet Containers

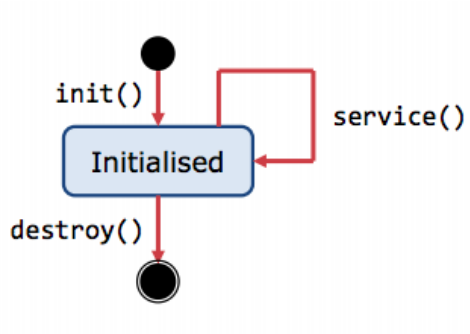
Servlet Containers



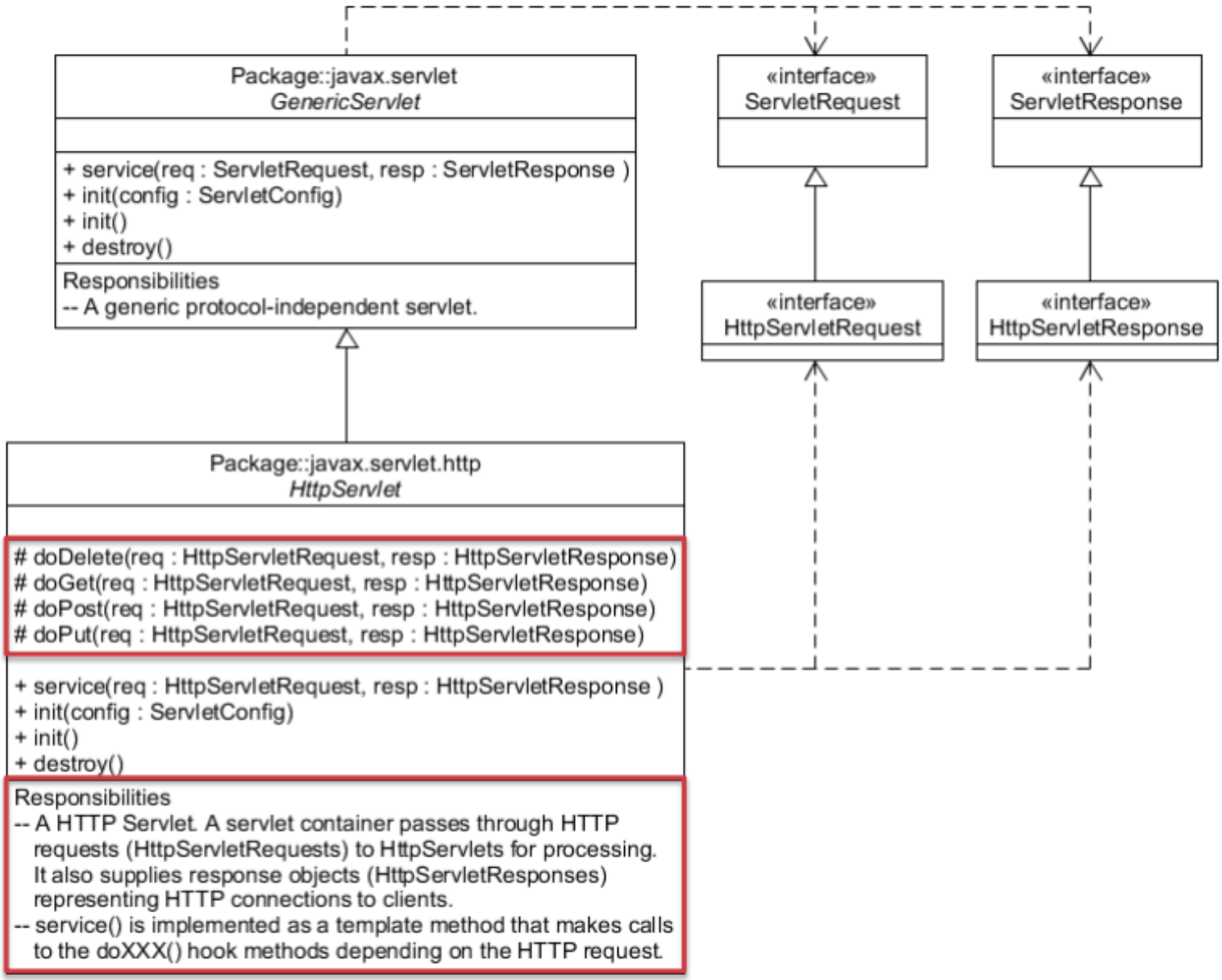
- The servlet container specification provides a managed execution environment for servlets.
  - A servlet is a Java component that extends the capabilities of a server.
  - A servlet has a lifecycle that is controlled by the container.
- Servlet containers route requests through to particular servlets.
  - Each incoming request is managed by a separated thread.
  - There is at most one instance of any servlet class – servlets need to be threadsafe.
- Servlet containers are an example of middleware.

Servlet Lifecycle

- The servlet container calls the lifecycle methods
  - `init()`
    - Initialises a servlet instance.
  - `service()`
    - Called once per incoming request for the servlet.
    - Supplies request data and a connection to the client.



  - `destroy()`
    - Called when the servlet container is shutting down or where resources need to be freed.
    - Typically implemented to save state to persistent storage.


Package::javax.servlet
GenericServlet
+ service(req : ServletRequest, resp : ServletResponse)
+ init(config : ServletConfig)
+ init()
+ destroy()
Responsibilities
-- A generic protocol-independent servlet.
«interface»
ServletRequest
«interface»
ServletResponse
«interface»
HttpServletRequest
«interface»
HttpServletResponse
Package::javax.servlet.http
HttpServlet
doDelete(req : HttpServletRequest, resp : HttpServletResponse)
doGet(req : HttpServletRequest, resp : HttpServletResponse)
doPost(req : HttpServletRequest, resp : HttpServletResponse)
doPut(req : HttpServletRequest, resp : HttpServletResponse)
+ service(req : HttpServletRequest, resp : HttpServletResponse)
+ init(config : ServletConfig)
+ init()
+ destroy()
Responsibilities
-- A HTTP Servlet. A servlet container passes through HTTP requests (HttpServletRequests) to HttpServlets for processing. It also supplies response objects (HttpServletResponses) representing HTTP connections to clients.
-- service() is implemented as a template method that makes calls to the doXXX() hook methods depending on the HTTP request.

HttpServlet Example

HelloWorldServlet.java

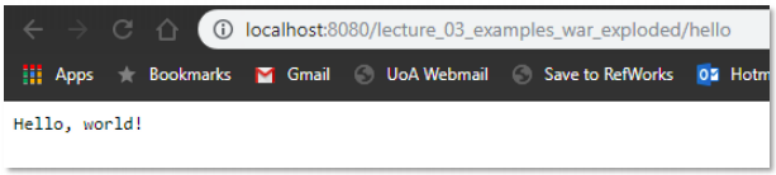
```java
public class HelloWorldServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
          throws ServletException, IOException {

        resp.setContentType("text/plain");

        PrintWriter out = resp.getWriter();
        out.println("Hello, world!");

    }
}
```

web.xml (config file)

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

    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>se325.lecture03.servlets.HelloWorldServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

HTTP Clients

- Any software implementing the HTTP protocol can act as a client – including your web browser!
- We can write HTTP clients in Java too, using the `Client` class – part of JAX-RS.

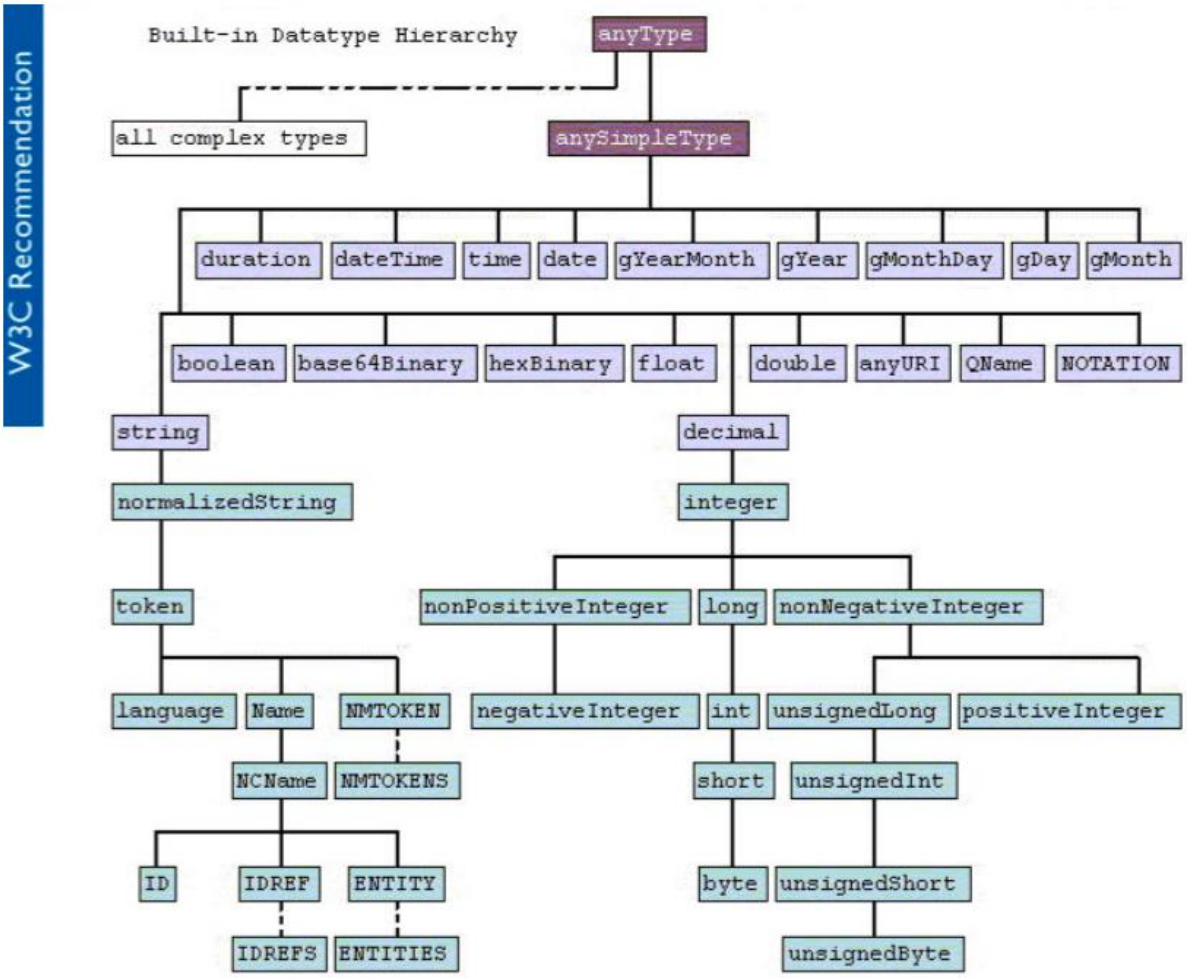



```java
Client httpClient = ClientBuilder.newClient();
Response response = httpClient.target("http://.../hello").request().get();

System.out.println("Status: " + response.getStatus());

String message = response.readEntity(String.class);
System.out.println("Message: " + message);

httpClient.close();
```

SOAP

Web Services

- HTTP is useful in developing an RMI-like service-invocation mechanism that isn't constrained to Java – but alone it's insufficient.
    - How do we represent service interfaces in a programming language-neutral way?
    - How do we describe service invocations?
    - How do we deal with different datatype systems?
- One method: SOAP over HTTP.

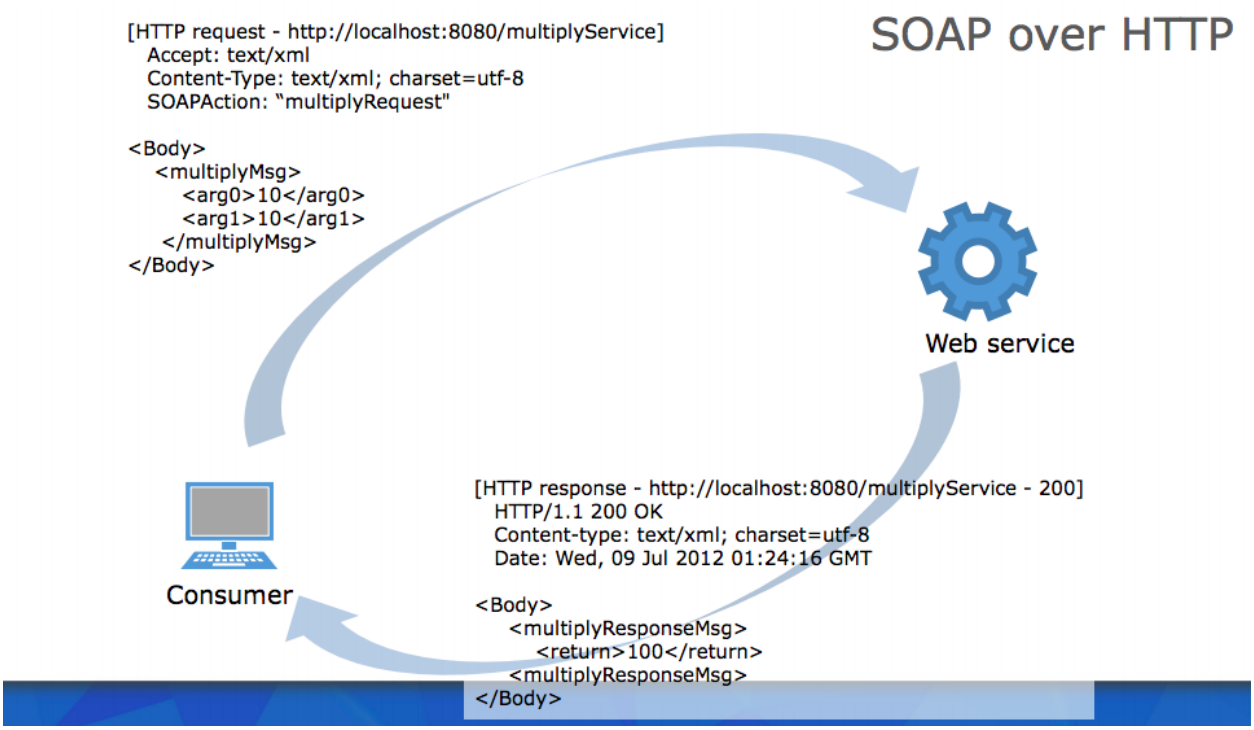


Programming-Language-Neutral Interfaces

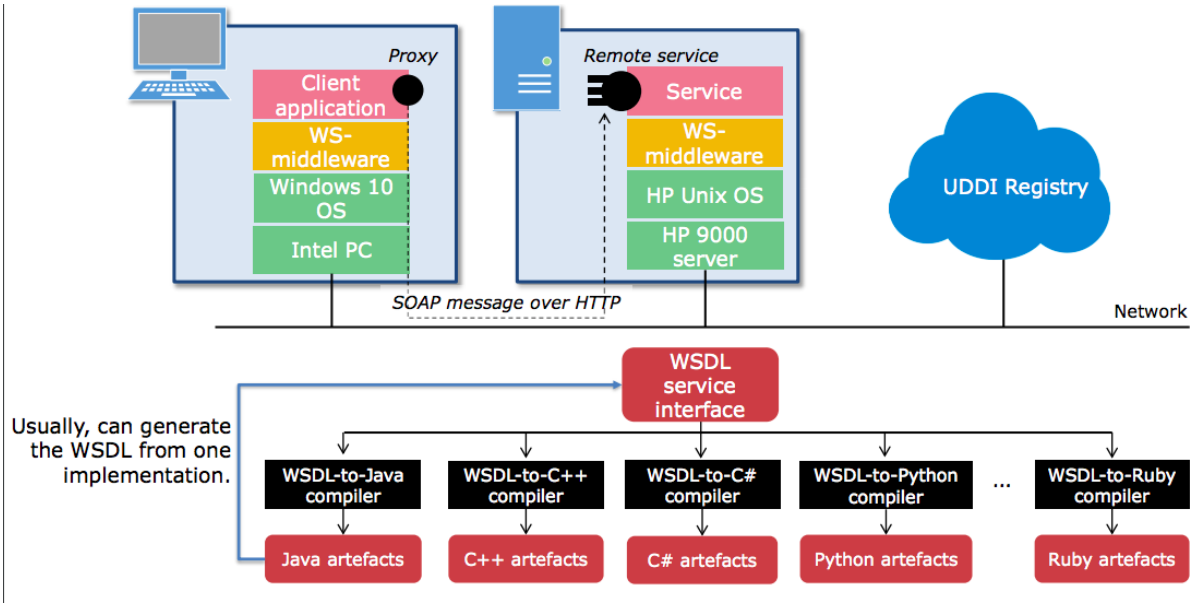
- WDSL (Web Service Description Language) is an XML dialect used to describe Web service interfaces
- A WDSL document includes several elements:
  - PortType
    - A set of named operations (like an interface).
    - Each operation is described by an input and output message.
  - Message
    - Typed messages.
  - Types
    - Datatype definitions.
  - Binding/Service
    - Communication endpoints identifying the location of a service.

```
<portType name="MultiplyService">
    <operation name="multiply">
        <input message="multiplyMsg"/>
        <output message="multiplyResponseMsg"/>
    </operation>
</portType>

<message name="multiplyMsg"/>
<message name="multiplyResponseMsg"/>

<types>
    <schema>
        <element name="multiplyMsg" type="multiplyType"/>
        <element name="multiplyResponseMsg"
           type="multiplyResponseType"/>
        <complexType name="multiplyType">
            <sequence>
                <element name="arg0" type="int"/>
                <element name="arg1" type="int"/>
            </sequence>
        </complexType>
        <complexType name="multiplyResponseType">
            <sequence>
                <element name="return" type="int"/>
            </sequence>
        </complexType>
    </schema>
</types>
```

(this is a snippet – not the whole WSDL!)



SOAP Web Services



REST

- REST (Representational State Transfer) more fully leverages the capabilities of the HTTP protocol.
- In addition to being an open request/reply protocol, HTTP:
  - Is a stateless protocol.
  - Defines methods for request messages and typed responses,
  - Supports negotiable content.

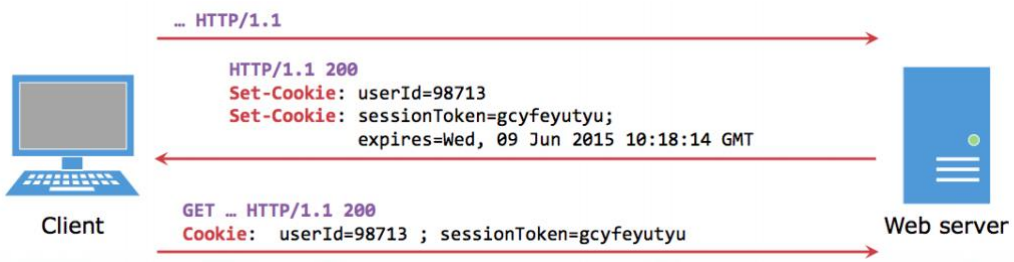
HTTP Method	Purpose
GET	• Requests specified resource • Can be made conditional on resource's last modification time
HEAD	• Similar to GET, only returns metadata (e.g. modification time, size, type)
POST	• Requests that the named resource processes data • Typically used to process form data
PUT	• Requests that the named resource is replaced with data contained in the message body
DELETE	• Requests that a resource be deleted
OPTIONS	• Requests the methods that are applicable to the named resource
TRACE	• Requests that the server simply sends back the request message

Response status code	Meaning
1xx	Informational
2xx	Success – the server received the request and successfully carried it out
3xx	Redirection
4xx	Client error. Typically used to represent: • A malformed URL supplied by the client; or • An attempt to access a resource which isn't held by the server
5xx	Server error

Stateless Protocol

As a stateless protocol, the server maintains no "session" state between requests.

It's often useful for a server to track which client it is processing requests for. A cookie allows clients to store session state and send this with each request.



Negotiable Content

- Using HTTP, clients can specify preferences for content.

REST (Representational State Transfer)

- REST originated in Roy Fielding's PhD thesis, "Architectural Styles and the Design of Network-based Software Architecture"
- Fielding proposed a set of architectural principles known as REST
  1. Addressable resources
  2. A uniform, constrained interface
  3. Representation-oriented
  4. Communicate statelessly
  5. Hypermedia As The Engine of Application State (HATEOAS)

REST Principle #1: Addressable Resources

- Every resource is reachable through a unique identifier.
- REST uses URIs to identify resources
  e.g. http://online-store.com/orders?id=111 might return the following:

```
{
  "id": 111,
  "customer": "http://online-store.com/customers/32133",
  "entries": [
    {
      "quantity": 5,
      "product": "http://online-store.com/products/111"
    },
    …
```

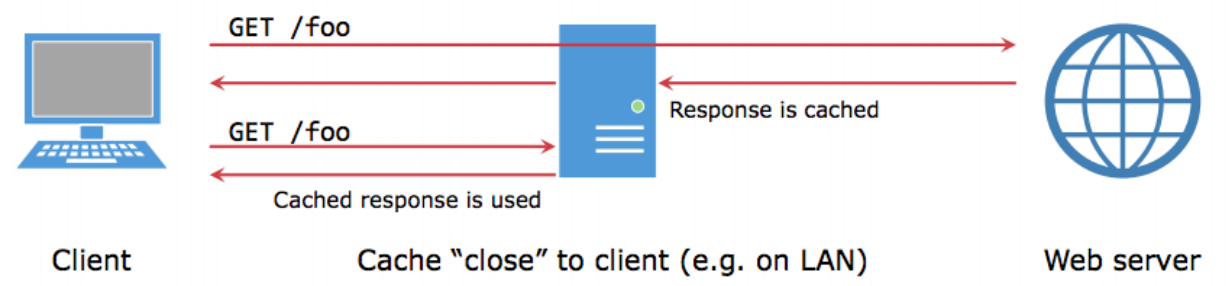
(JSON notation – we'll learn about that next week!)

REST Principle #2: Uniform, Constrained Interface

- Use the HTTP methods, as intended, in implementing the service.

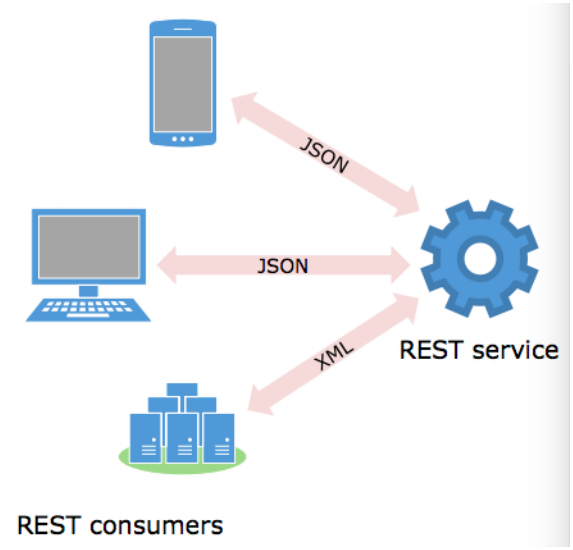
HTTP method	Description	Idempotent?	Safe?	CRUD operation
GET	A read-only operation that queries a server for specific information	Yes	Yes	Retrieve
PUT	Requests that the server stores the request's message body under the location specified in the HTTP request	Yes	No	Update
DELETE	Removes a specified resource	Yes	No	Delete
POST	Changes the state of a service based in some way, e.g. creating a new resource	No	No	Create

- Use of HTTP methods as intended can allow us to perform optimizations
    - For example, where GET's semantics are respected, GET responses can be cached, contributing to scalability.

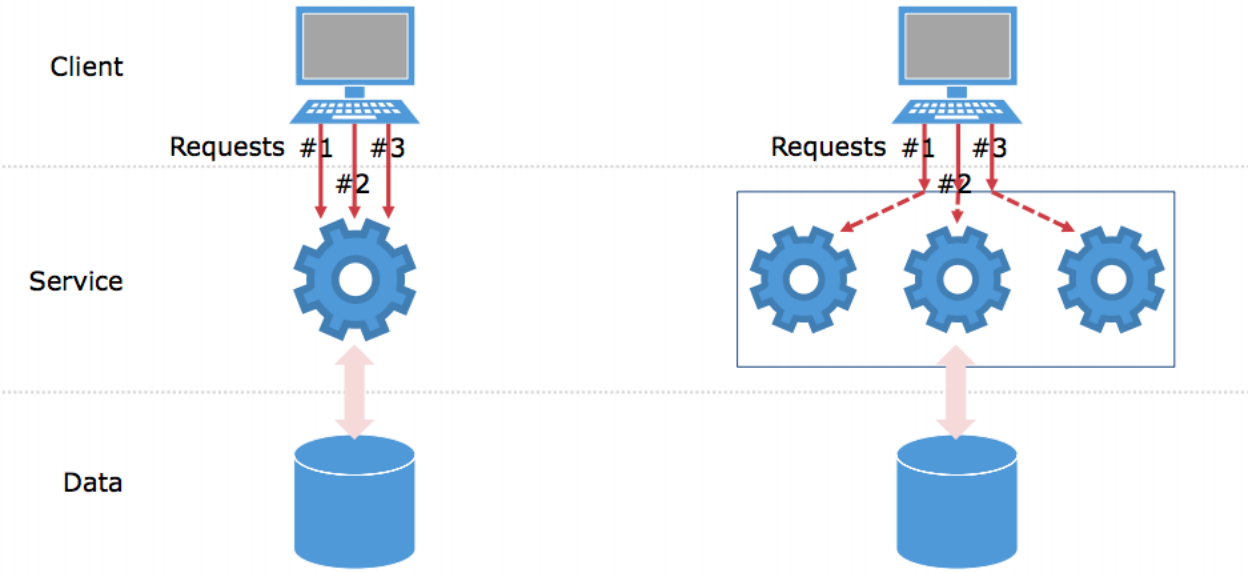


REST Principle #3: Representation-Oriented

- Consumers and services exchange representations of resources.
    - Representations can take many forms, as specified by MIME types
        - text/subtype
        - text/plain
        - text/html
        - application/xml
        - application/json
- Consumers can negotiate a representation with a service.
    - Using HTTP request `Accept` header and `Content-Type` response header.



REST Principle #4: Stateless Communication



REST Principle #5: HATEOAS

- Hypermedia As The Engine Of Application State
    - Responses contain links telling the client “where they can go next”. E.g.:

First request: `GET /products`

Response: first five entries

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: /products?startIndex=5;rel="next"
[
  {
    "id": 0,
    "name": "headphones",
    "price": "$16.99"
  },
  {
    "id": 1,
  …
]
```

Second request: `GET /products?startIndex=5`

Response: next five entries

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: /products?startIndex=0;rel="previous"
Link: /products?startIndex=10;rel="next"
[
  {
    "id": 5,
    "name": "meaning of life",
    "price": "$42.00"
  },
  {
    "id": 6,
  …
]
```

SOAP vs REST

SOAP:

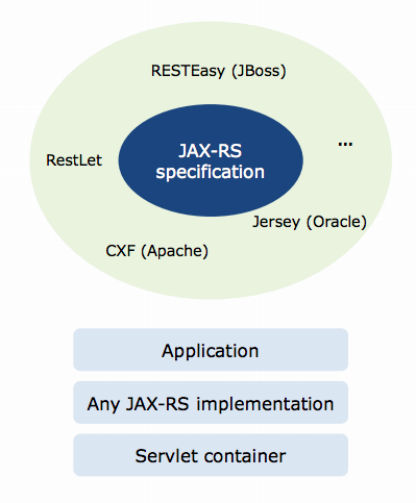
- HTTP is used as nothing more than a transport protocol.
- SOAP is based on many standards, e.g. SOAP and WSDL, and requires associated tools.
- SOAP services have formally defined contracts that specify service interfaces.

REST:

- HTTP and its features are leveraged; HTTP provides a service API.
- REST relies only on HTTP; There’s no need for other standards and tools.
- REST is ad-hoc; Service contracts are not well defined (REST interfaces don’t specify the type of data to be exchanged).

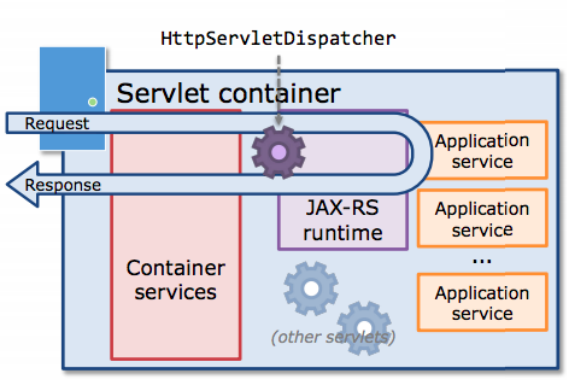
JAX-RS

JAX-RS Specification



- JAX-RS is an open specification for a framework that simplifies development of RESTful Web services.
    - As with any open specification, modifiability is promoted.
- Key features:
    - Annotation-based
    - Parameter injection
    - Message body marshallers/unmarshallers
- We will use the RESTEasy implementation in this course, but others can easily be substituted with no change to the codebase.

JAX-RS Programming Model



- JAX-RS implementations provide a servlet class, `HttpServletDispatcher`.
- `HttpServletDispatcher` loads application services and delegates processing of HTTP messages to them.
    - You no longer need to implement servlets, instead you supply application services.
- The JAX-RS runtime performs pre- and post-processing on service requests and responses.
- You should:
    - Create a Resource class with methods that process service requests.
    - Create a subclass of `Application`.
        - Override method `getSingletons()` to return a set containing an instance of the Resource class.
    - On startup, JAX-RS finds `Application` subclasses, instantiates them and calls their `getSingletons()` methods.
- JAX-RS routes all requests directed at the application service to the registered Resource object.

A "Hello World" Service Example

- We will build a web service which accepts GET requests to …/services/greatings/hello.
- Consumers can optionally supply their name as a *query parameter.* If not, a default value will be used.
- The service will return a JSON object containing a greeting. E.g.:

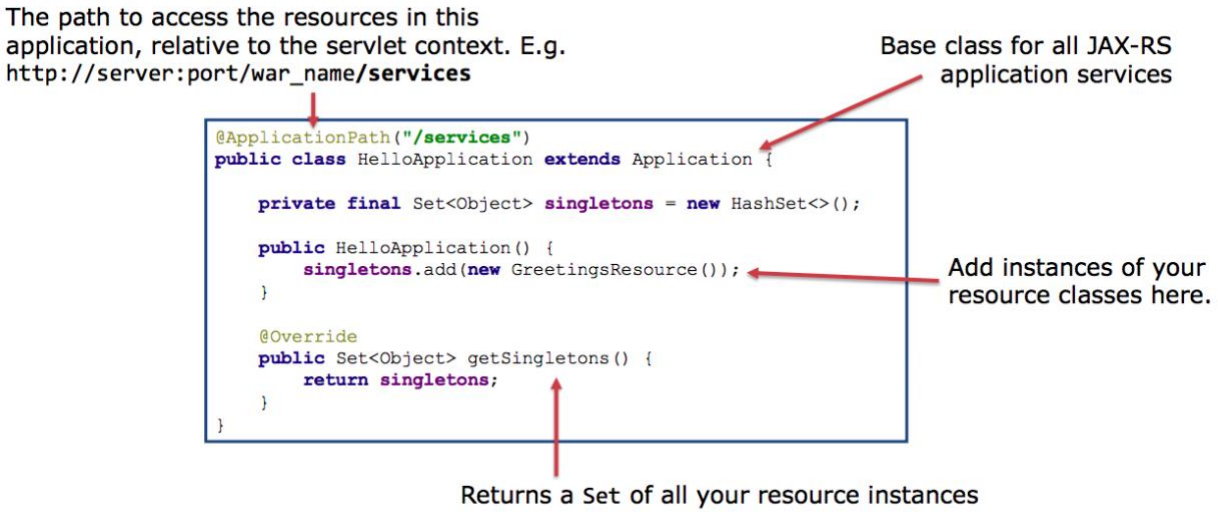
  Request: `GET http://.../services/greetings/hello?name=Bob`

  Response: {

  `"greeting": "Hello, Bob!"`

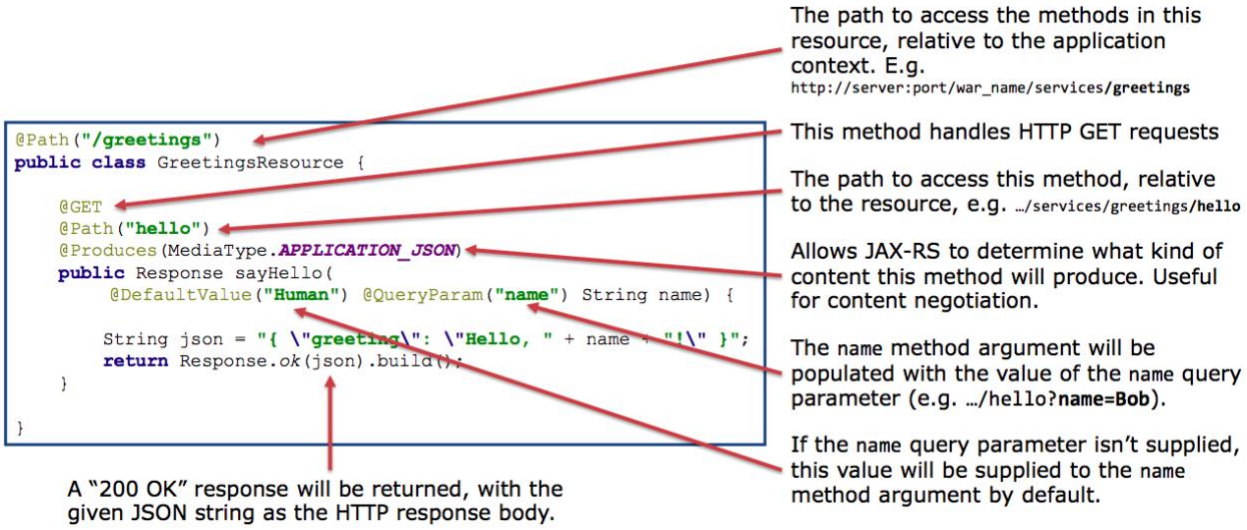
  }

HelloApplication class




```
@ApplicationPath("/services")
public class HelloApplication extends Application {

    private final Set<Object> singletons = new HashSet<>();

    public HelloApplication() {
        singletons.add(new GreetingsResource());
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

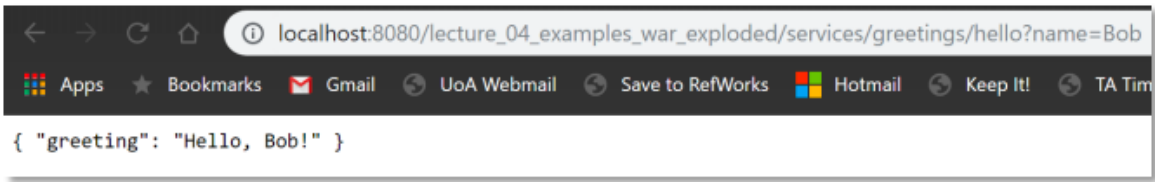
GreetingsResource class




```
@Path("/greetings")
public class GreetingsResource {

    @GET
    @Path("hello")
    @Produces(MediaType.APPLICATION_JSON)
    public Response sayHello(
        @DefaultValue("Human") @QueryParam("name") String name) {

        String json = "{ \"greeting\": \"Hello, " + name + "!\" }";
        return Response.ok(json).build();
    }

}
```

Consuming the service

- From the browser…

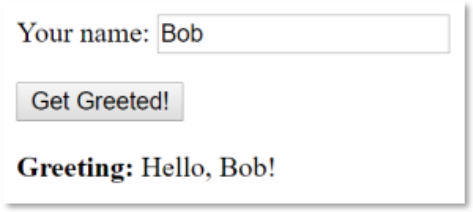


- From Java…

```
Client client = ClientBuilder.newClient();
Response response =
client.target("http://.../services/greetings/hello?name=Bob").request().get();
String json = response.readEntity(String.class);
System.out.println("Response JSON: " + json);
client.close();
```

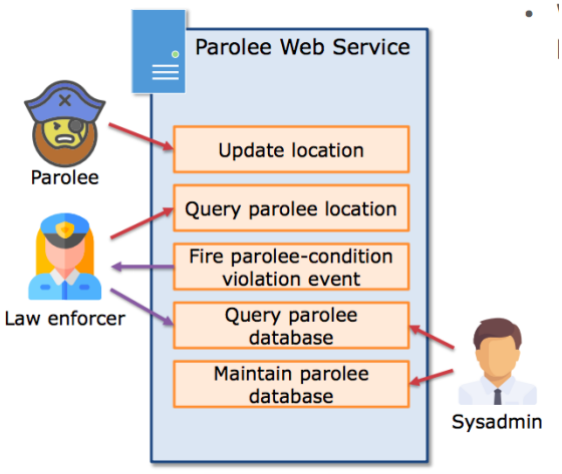
- From JavaScript…

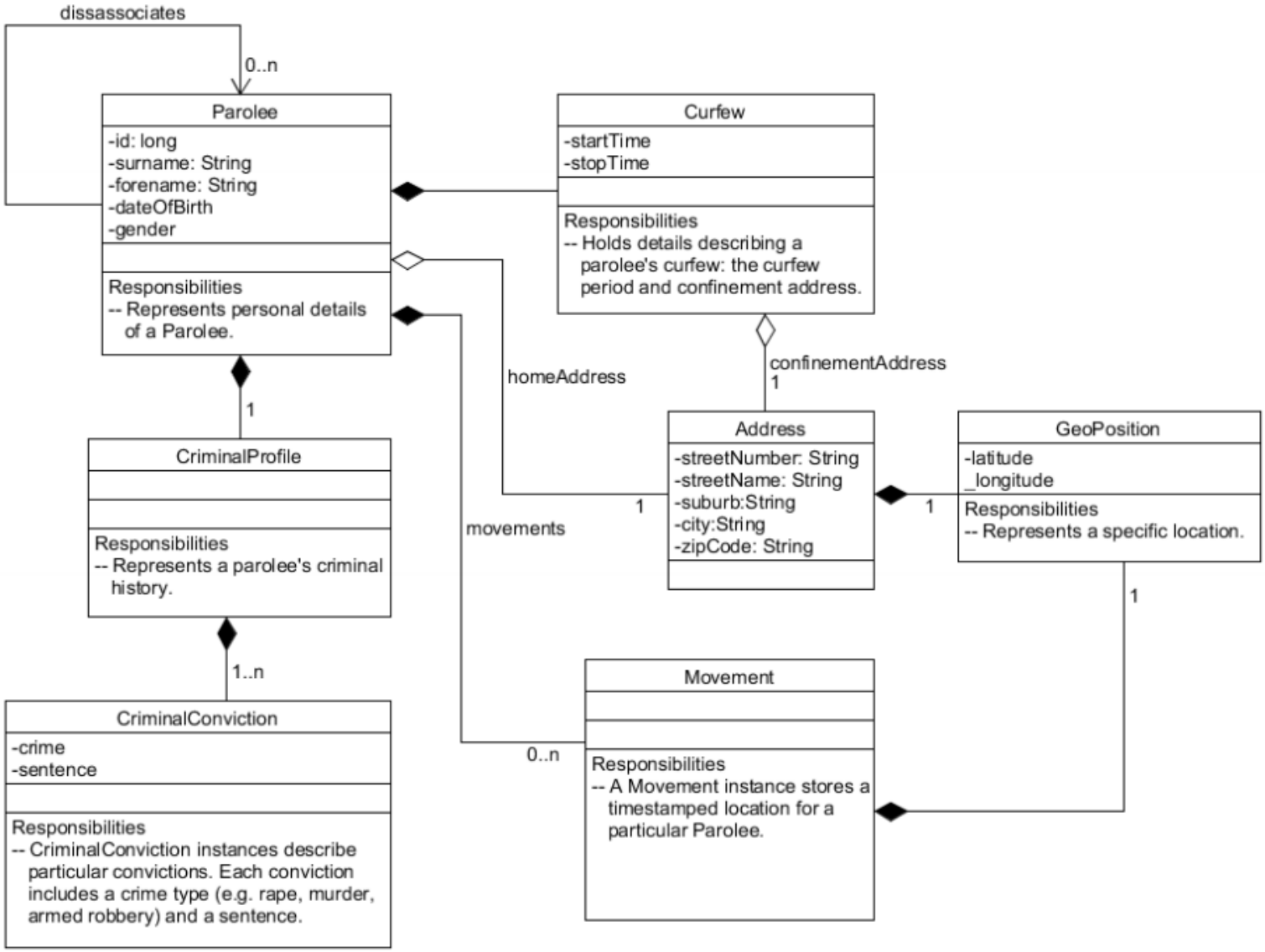
```
const input = document.querySelector('#txt-name');
const name = input.value;
const response = await fetch(`./services/greetings/hello?name=${name}`);
const json = await response.json();
const output = document.querySelector('#span-greeting');
output.innerText = json.greeting;
```



A Parolee Web Service Example

- Web service interface – possible entry points:
  - /parolees
    - Allows Administrators to obtain a list of parolees.
  - /parolees/{id}
    - Provides access to a particular parolee, identified by their id.
  - /parolees/{id}/movements
    - Allows a parolee to inform the service that he/she has moved location.
    - Enable law enforcers to query a parolee's location.



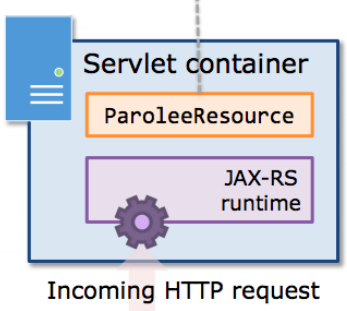

dissassociates
0..n
Parolee
-id: long
-surname: String
-forename: String
-dateOfBirth
-gender
Responsibilities
-- Represents personal details of a Parolee.
Curfew
-startTime
-stopTime
Responsibilities
-- Holds details describing a parolee's curfew: the curfew period and confinement address.
homeAddress
confinementAddress
1
1
CriminalProfile
Responsibilities
-- Represents a parolee's criminal history.
Address
-streetNumber: String
-streetName: String
-suburb:String
-city:String
-zipCode: String
1
1
GeoPosition
-latitude
_longitude
Responsibilities
-- Represents a specific location.
1
movements
1..n
CriminalConviction
-crime
-sentence
Responsibilities
-- CriminalConviction instances describe particular convictions. Each conviction includes a crime type (e.g. rape, murder, armed robbery) and a sentence.
0..n
Movement
Responsibilities
-- A Movement instance stores a timestamped location for a particular Parolee.

REST Interface

CRUD operation	HTTP method	Resource	Query params	Response headers	Message body JSON representation
Create	POST	/parolees		201 Created Location header with URI for new resource	Request message
Retrieve	GET	/parolees/{id}		200 OK (Success) 404 Not Found (Fail)	Response message
	GET	/parolees	startIndex size	200 OK (Success) 404 Not Found (Fail)	Response message
Update	PUT	/parolees/{id}		204 No content (Success) 404Not Found (Fail)	Request message
Delete	DELETE	/parolees/{id}		204 No Content (Success) 404 Not Found (Fail)	

Parameter Injection

```
@Path("/parolees")
public class ParoleeResource {
    @GET
    public StreamingOutput getParolees(
          @QueryParam("start") long start,
          @QueryParam("size") int size,
          @Context HttpHeaders headers) {…}
}
```



Incoming HTTP request

```
GET /parolees ? start = 0 & size = 10
    Accept: application/xml
```

More info: JAX-RS Javadoc

Annotations supporting injection
@javax.ws.rs.PathParam
@javax.ws.rs.MatrixParam
@javax.ws.rs.QueryParam
@javax.ws.rs.FormParam
@javax.ws.rs.HeaderParam
@javax.ws.rs.CookieParam
@javax.ws.rs.core.Context

Automatic Type Conversion

- Since HTTP is a text-based protocol, all parts of HTTP requests (other than the body) are represented as Strings.
- The JAX-RS run-time automatically converts Strings to data types.

	Injected parameter type	Conversion
1	Primitive type	String data converted to the specified primitive type
2	A class that includes a constructor with a single String argument	The constructor is called with the String data as the argument
3	A class that has a static method named `valueOf()`	The `valueOf()` method is called with the String data as the argument; `valueOf()` returns an instance of the class (all enums fall into this category)
4	A collection whose generic type satisfies criteria 2 or 3	The Collection (e.g. `java.util.List`) should have a generic type argument T where T satisfies criteria 2 or 3

- For custom conversion, see interfaces `ParamConverter` and `ParamConverterProvider` (in package `javax.ws.rs.ext`).

Default Request Processing & Error Handling

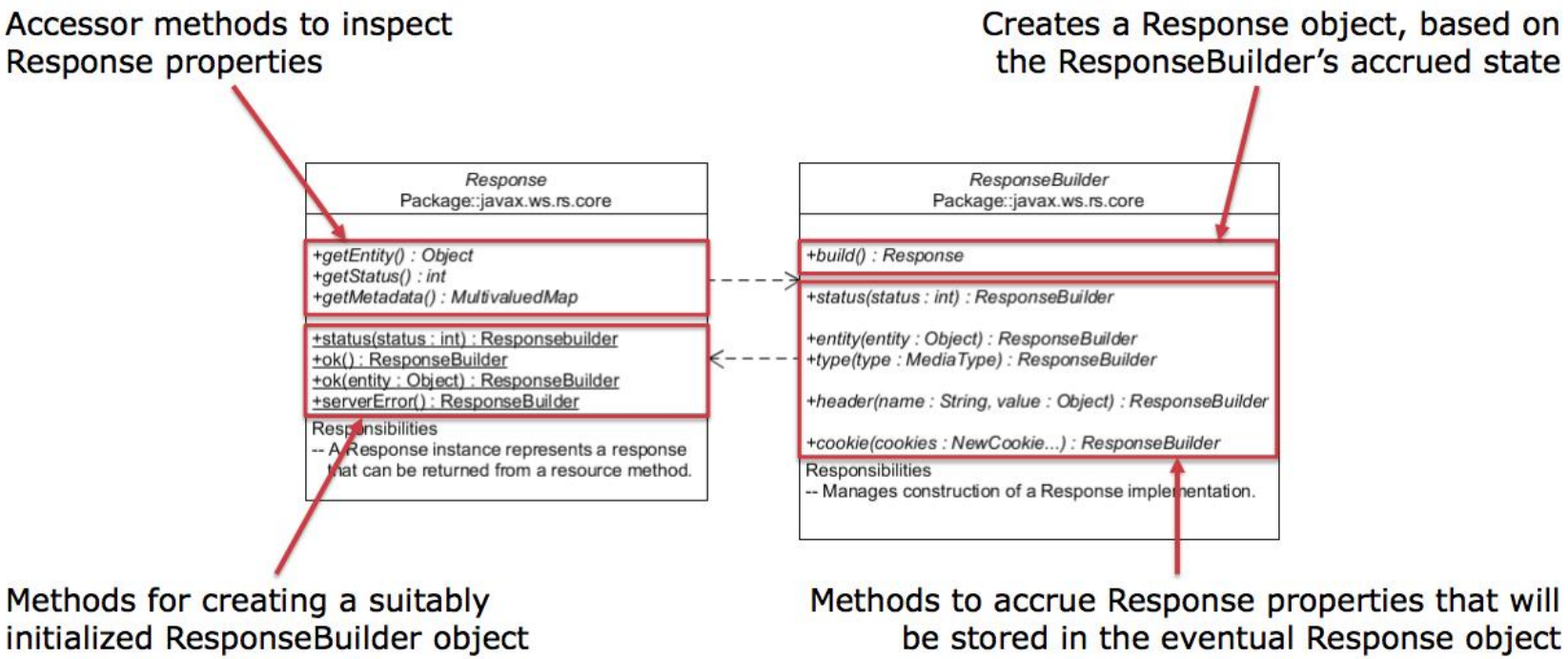
Success cases:

- Resource method with a void type results in a 204 'No content' response.
- Resource method with a non-void return type results in either a 200 'OK' response or a non-null response, and a 204 'No Content' response otherwise.

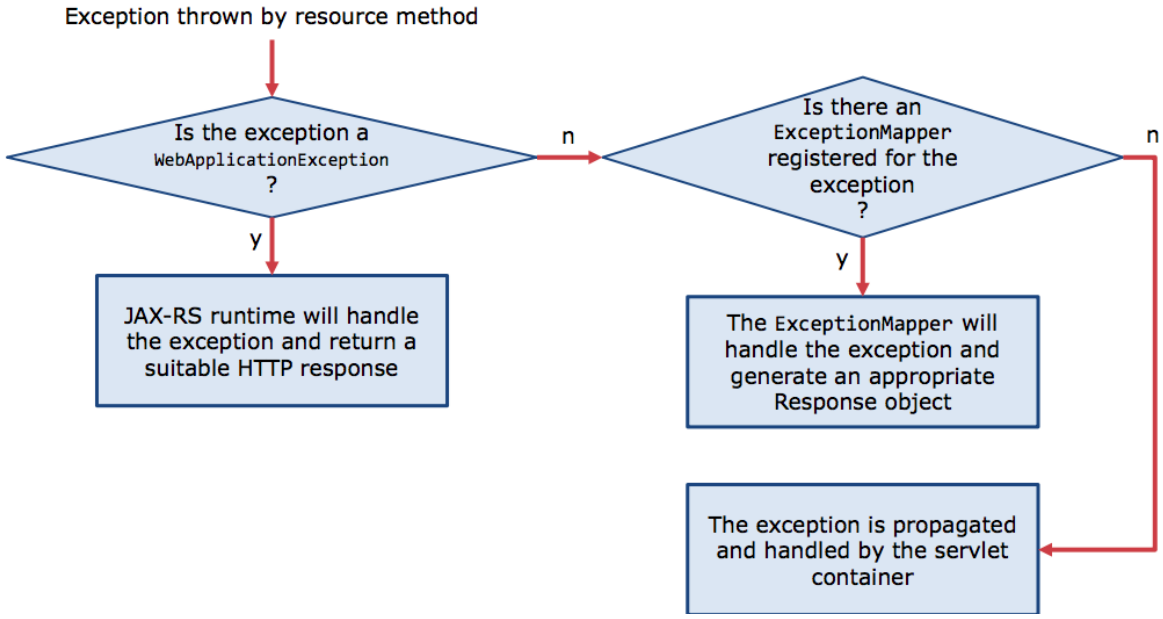
Error cases:

- If a client invokes on a misspelt URI, 404 'Not Found' is returned.
  - **GET …/parole/45**
- If a client attempts to invoke a HTTP method that's not associated with the given URI, 405 'No method' is returned.
  - **DELETE …/parolees**
- If a client requests a content type that's not supported, 406 'Not Acceptable' is returned.
  - **GET …/parolees/53**
  - Accept: application/xml

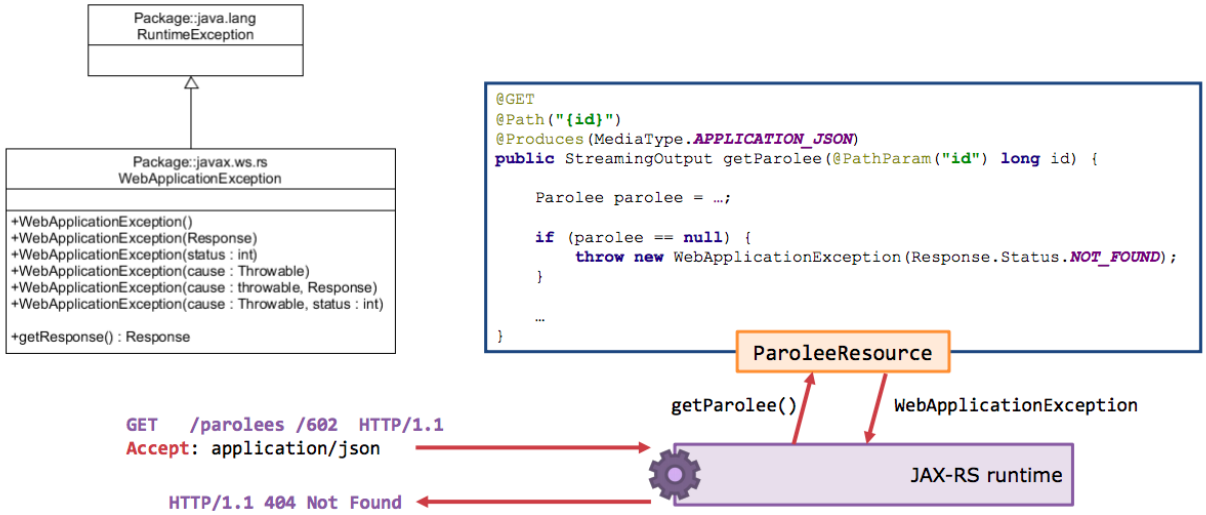
Complex Responses



Exception Handling



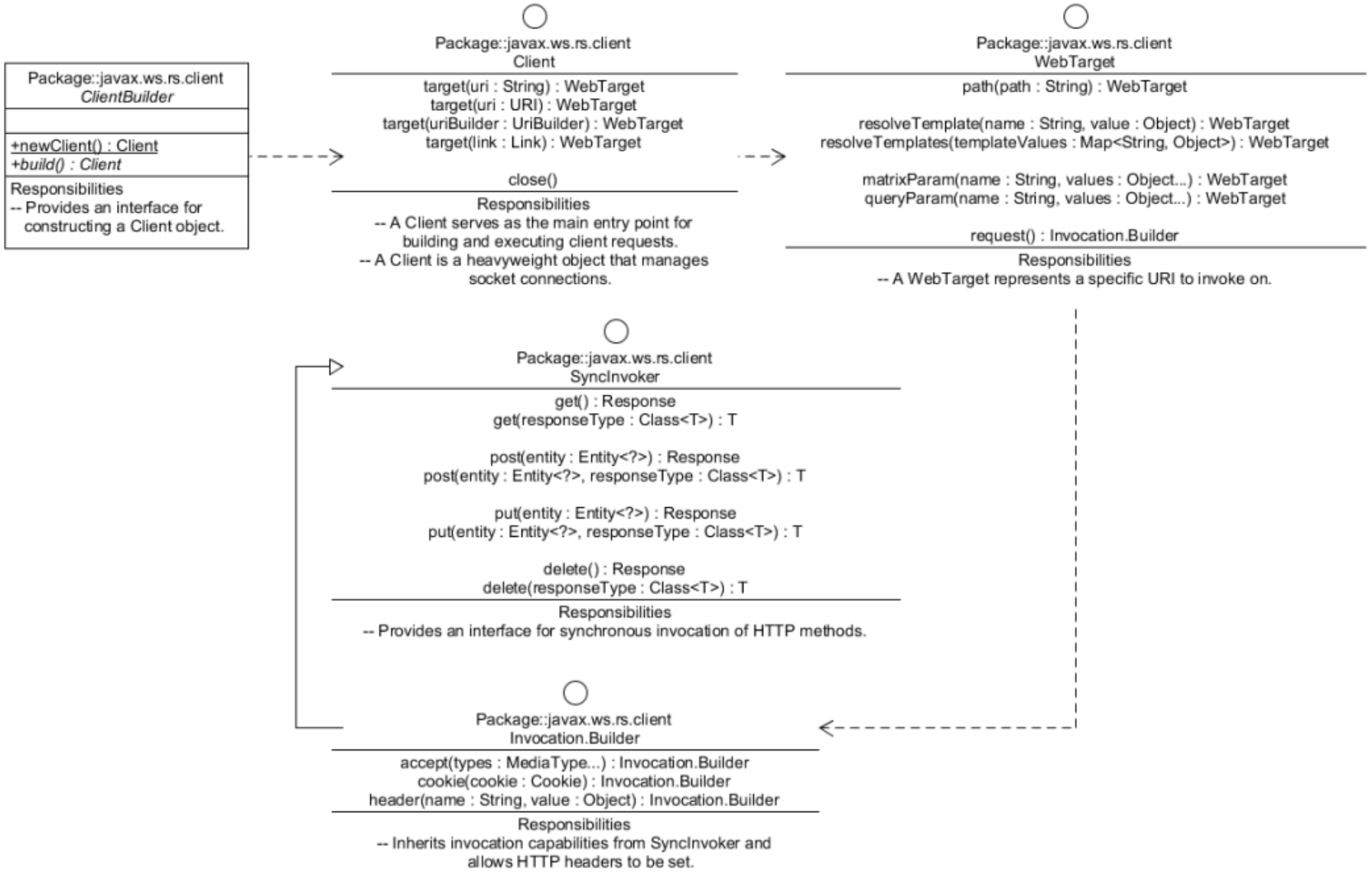
WebApplicationException



JAX-RS Client

```
client = ClientBuilder.newClient();

WebTarget target =
client.target("http://parolees.org.nz/services/parolees/3");
String paroleeAsJson = target.request().get(String.class);

client.close();
```

Client API – Key Interfaces



PUT & POST Requests

- Use this `put()/post()` method for making PUT / POST requests where you expect the Web service to return a value of a specified type (i.e. `String` in this case).
- Use this `put()/post()` method for making PUT / POST requests where either:
  - You don't expect the Web service to return any data in the HTTP response message body; Or,
  - You are interested in headers and meta data in the HTTP response message.

```
String payload = …;

String paroleeJson = client
  .target("http://.../parolees")
  .request()
  .put(Entity.json(payload) , String.class);

…
```

```
String payload = …;

Response response = client
  .target("http://.../parolees")
  .request()
  .post(Entity.json(payload));

int responseCode = response.getStatus();

…

response.close();
```

Client Exception Handling

- The `SyncInvoker` HTTP request methods may throw `ProcessingException`.
- In addition, the `SyncInvoker` methods that return application data (as opposed to Response objects) may throw `WebApplicationExceptions`.

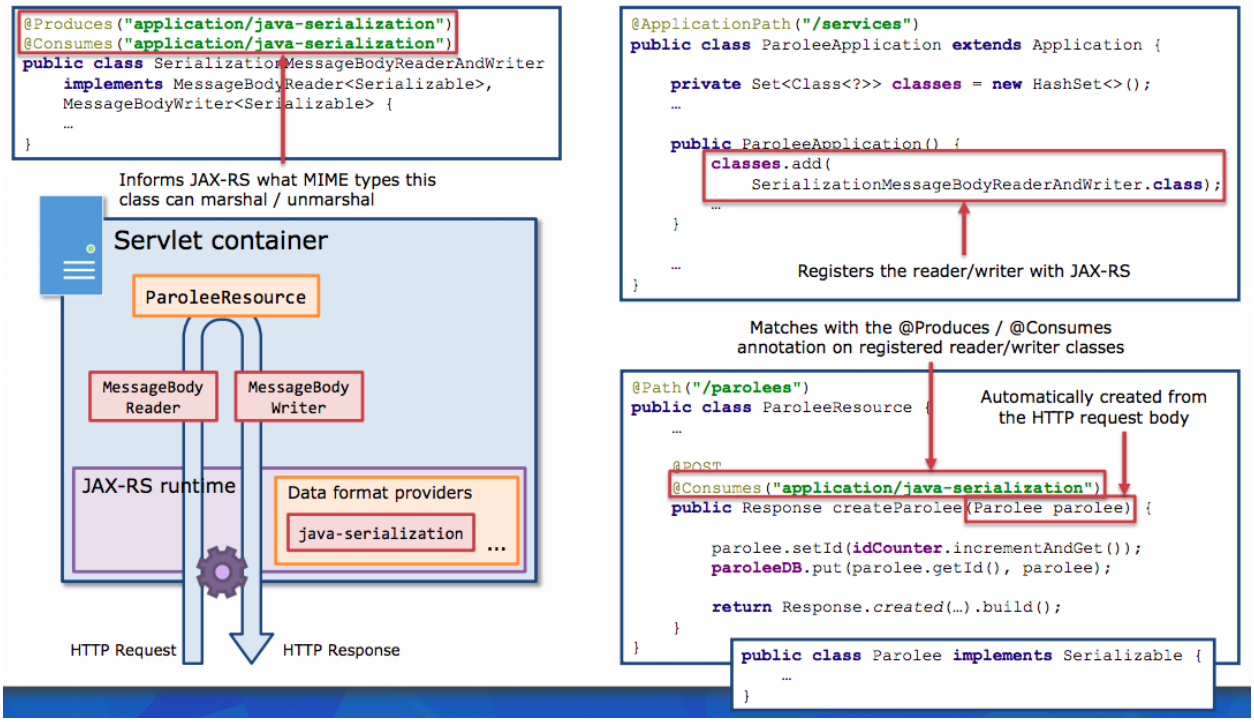
```
try {
    String paroleeAsXml = client
      .target("http://.../parolees/123")
      .request()
      .accept("application/xml")
      .get(String.class);
} catch (NotAcceptableException notAcceptable) {
    // Service couldn't provide XML content.
} catch (NotFoundException notFound) {
    // Server couldn't match the URI.
} catch (ProcessingException processing) {
    // Internal JAX-RS processing error.
}
```

- A Response object **must** be closed before making a subsequent HTTP request using the same `Client` instance.
  - Failing to do so leaks resources and results in a `ProcessingException`.
  - Can use try-with-resources syntax.

Resource Representation

`MessageBodyReading/MessageBodyWriter`

- JAX-RS contains two interfaces:
  - `MessageBodyReader`: Handles **unmarshalling** (deserialization) of an HTTP message body in some format to Java objects.
  - `MessageBodyWriter`: Handles **marshalling** (serialization) of Java objects to an HTTP method body in some format.
- Typically written in pairs, often with one class implementing both interfaces.
- Custom readers / writers can be implemented and registered with JAX-RS in the `Application` class.
- Appropriate readers / writers are chosen at runtime based on MIME types.
  - Specified using `@Produces/@Consumes` annotations within your Resource classes and your reader / writer classes.



JAX-RS `Client`

- Use the `register()` method of `ClientBuilder` to register your custom readers / writers.
- Use the `accept()` method to state your allowed response types.
- Use the `Entity.entity()` method to specify a custom MIME type for your request.

Using JSON for Resource Representation

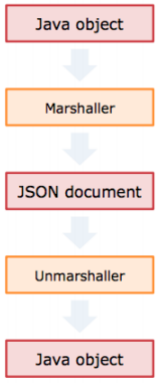
JSON (JavaScript Object Notation) Representation

- JSON is one of the most widely used *data interchange* formats.
  - XML is also widely used, but it is on the decline in some scenarios.
  - Current JSON popularity due to widespread use in JavaScript applications. JavaScript is the most widely used programming language.
- Human-readable, text-based format.
  - Lightweight compared with XML.
- Separates content from presentation
- Promotes interoperability.

```
{
  "id": 1,
  "firstName": "Al",
  "lastName": "Capone",
  "gender": "MALE",
  "dateOfBirth": "1899-01-17",
  "address": {
    "houseNumber": 123,
    "street": "Some St",
    "suburb": "Townsville",
    "city": "Citysburg"
  }
}
```

Jackson

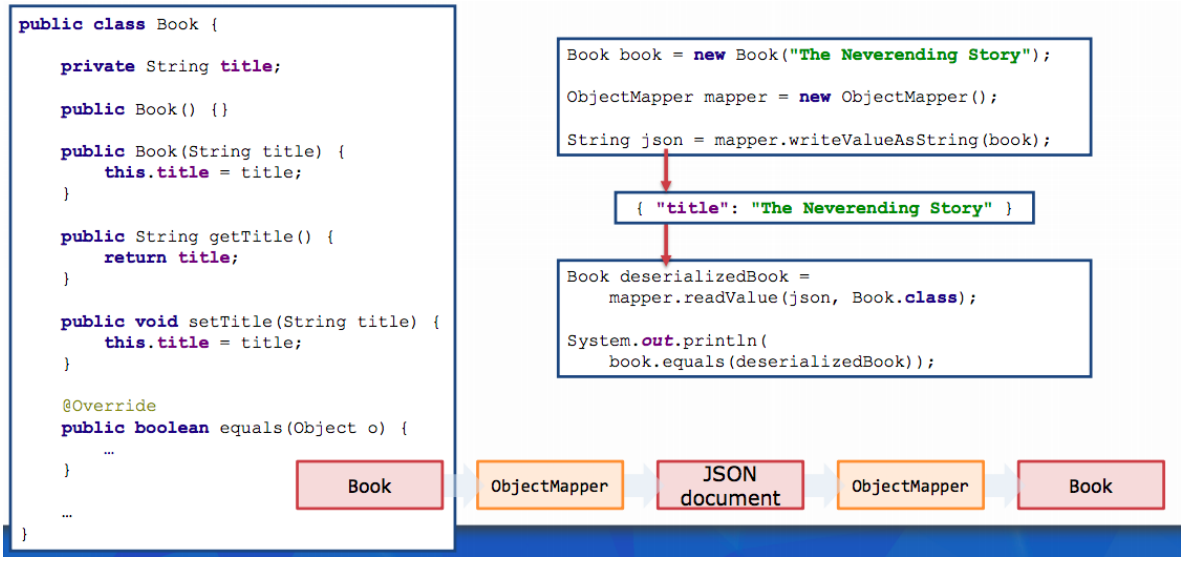
- Jackson is a specification for working with JSON.
- Jackson enables developers to work with domain objects.
  - Jackson handles the conversion of objects to / from JSON.
- JAX-RS integrates with Jackson.



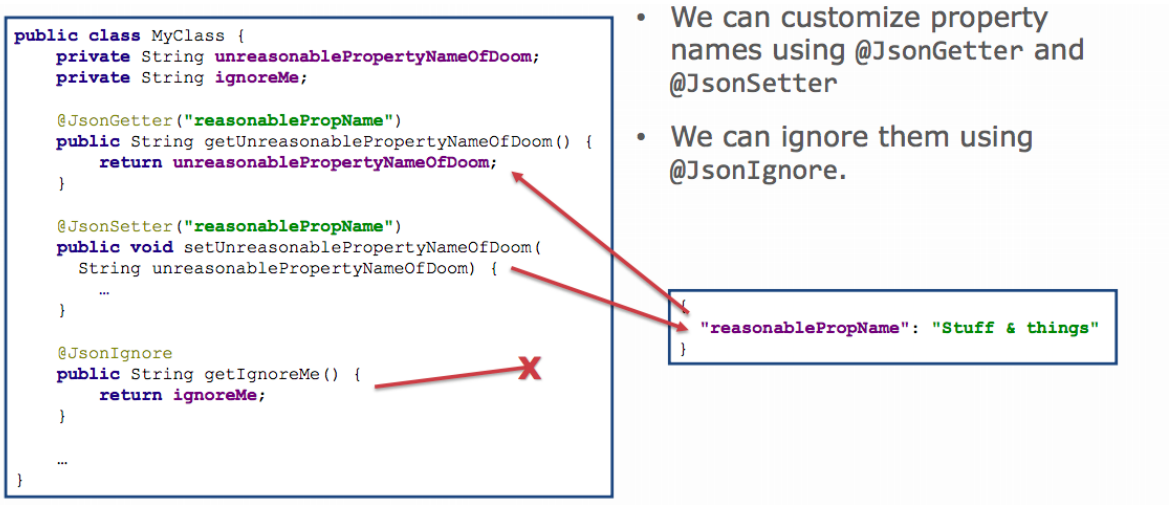
A simple example

- Jackson advocates “configuration by exception”
  - Certain behaviour is provided by default; Use annotations to override these defaults.
- By default, Jackson expects classes to conform to JavaBean conventions.
  - Each *property* of a bean will be marshalled / unmarshalled.
  - A JavaBean property refers to a getter / setter method pair (e.g. the Book class has a single property, called “title”.
  - JavaBeans must have a default (no-argument) constructor, which will be used to create instances during the unmarshalling process.

```
public class Book {

    private String title;

    public Book() {}

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public boolean equals(Object o) {
        …
    }

    …
}
```

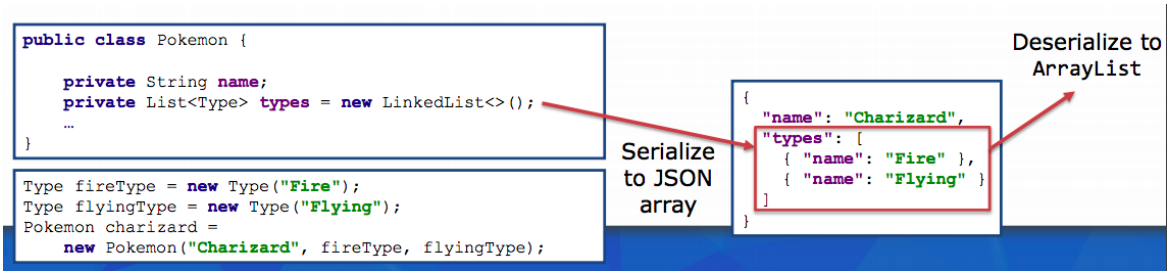


Ignoring & Changing Properties



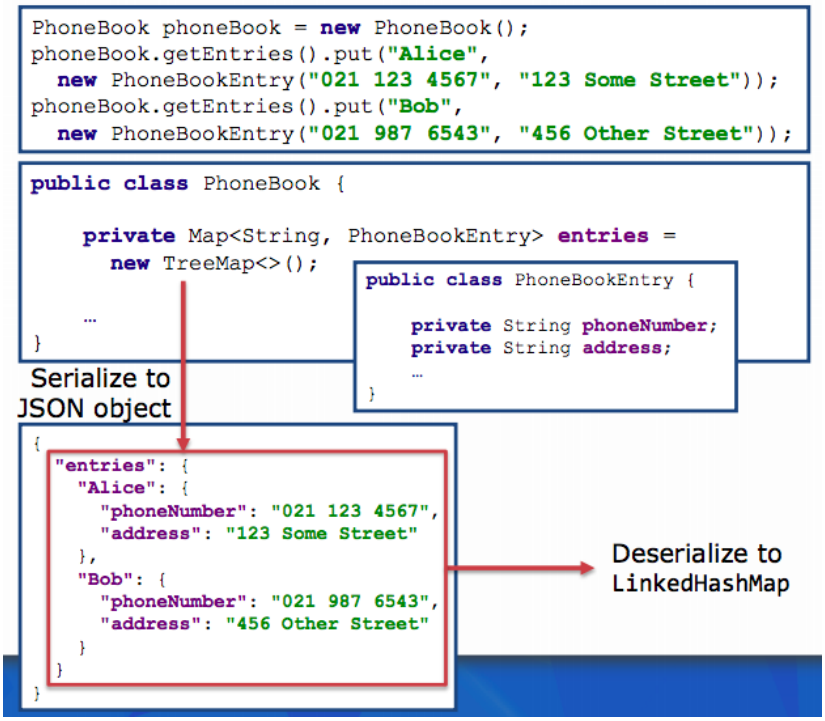
Lists & Arrays

- Lists and arrays both marshal to JSON arrays
- When unmarshalling JSON arrays:
  - If the bean property is Java array, the JSON array will be unmarshalled to a Java array.
  - If the bean property is a `List`, the JSON array will be unmarshalled to an `ArrayList` (regardless of the original list type before serialization!).



Maps

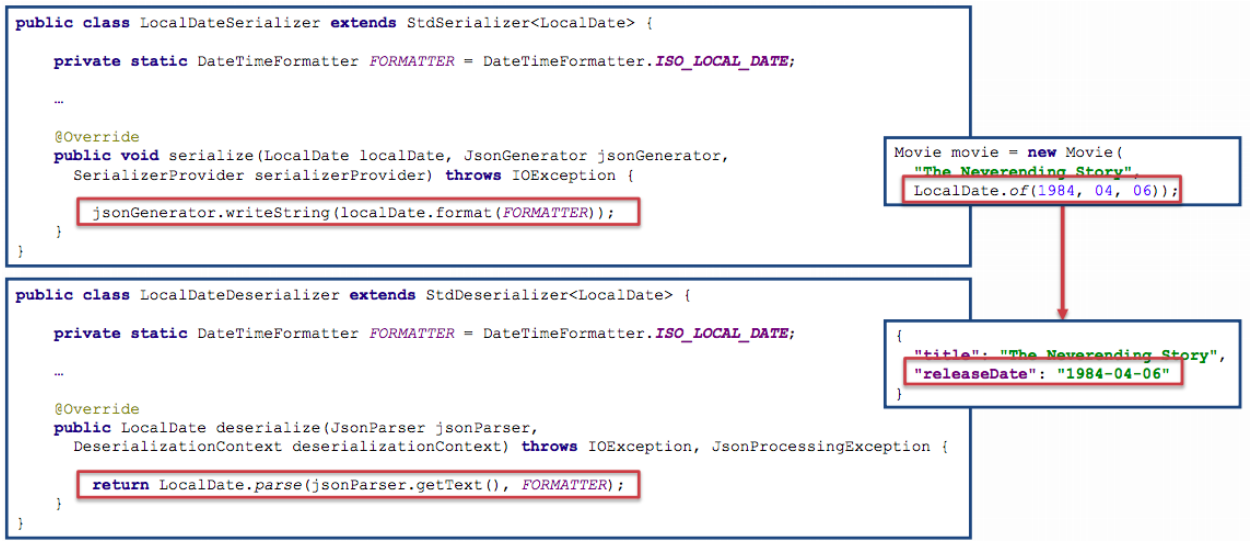
- Maps are marshalled to JSON objects.
  - Each map key is marshalled to a property name equalling the value of the key's `toString()` method.
- When unmarshalling a JSON object into a `Map`:
  - The type of `Map` created is a `LinkedHashMap`.
  - JSON object properties are unmarshalled to `String` keys.



Custom Marshalling / Unmarshalling

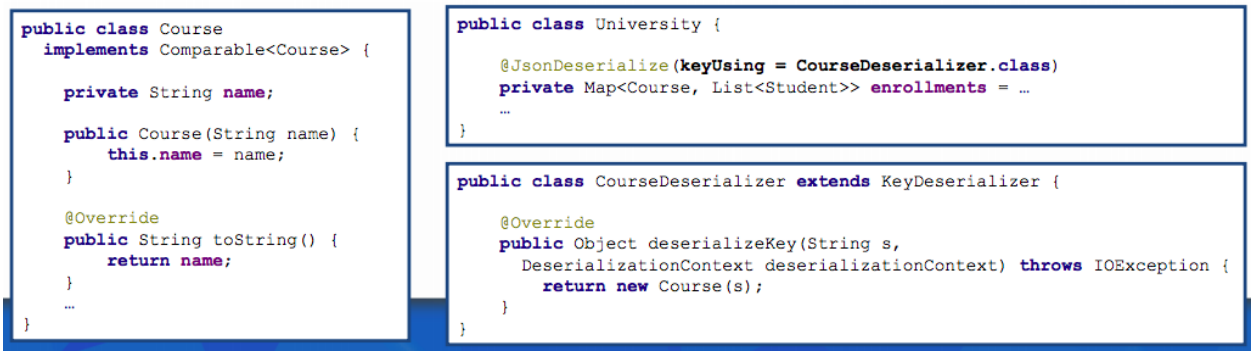
- Java classes which aren't JavaBeans may not be handled appropriately by Jackson.
  - One example includes the classes commonly used to represent date & times (`LocalDate /LocalDateTime`).
- We can define implementations of `StdSerializer` and `StdDeserializer`, and have Jackson use them where necessary via the `@JsonSerialize` and `@JsonDeserialize` annotations.

```
public class Movie {

    private String title;
    private LocalDate releaseDate;
    …

    @JsonSerialize(using = LocalDateSerializer.class)
    @JsonDeserialize(using = LocalDateDeserializer.class)
    public LocalDate getReleaseDate() { … }
}
```



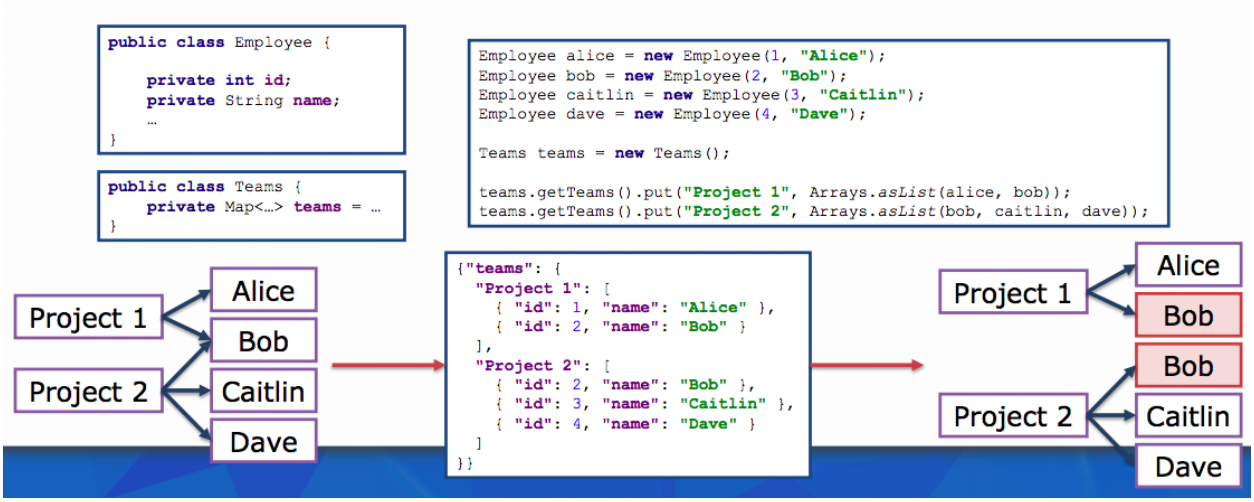
Custom Marshalling / Unmarshalling – Map keys

- Map keys are marshalled to JSON Strings by calling their `toString()` methods.
- To unmarshal a key back to anything other than a String, we must:
  - Implement a `KeyDeserializer` responsible for converting Strings to other object types,
  - Instruct Jackson to use it with the `@JsonDeserialize` attribute.

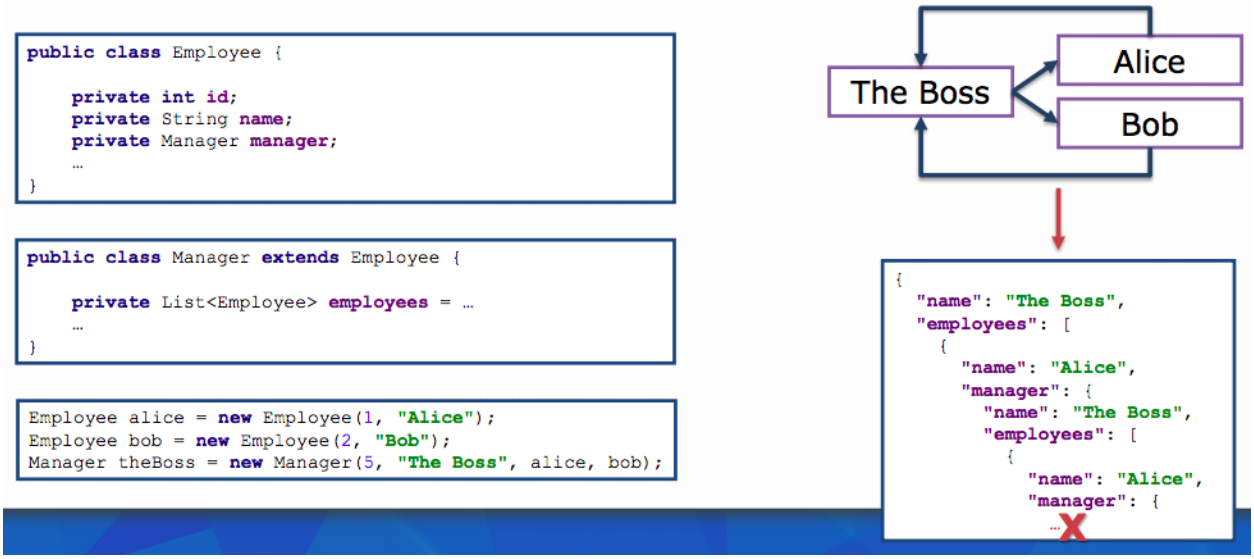


Object References

- Scenario: Suppose we marshall an object graph containing multiple references to the same object. What will happen?

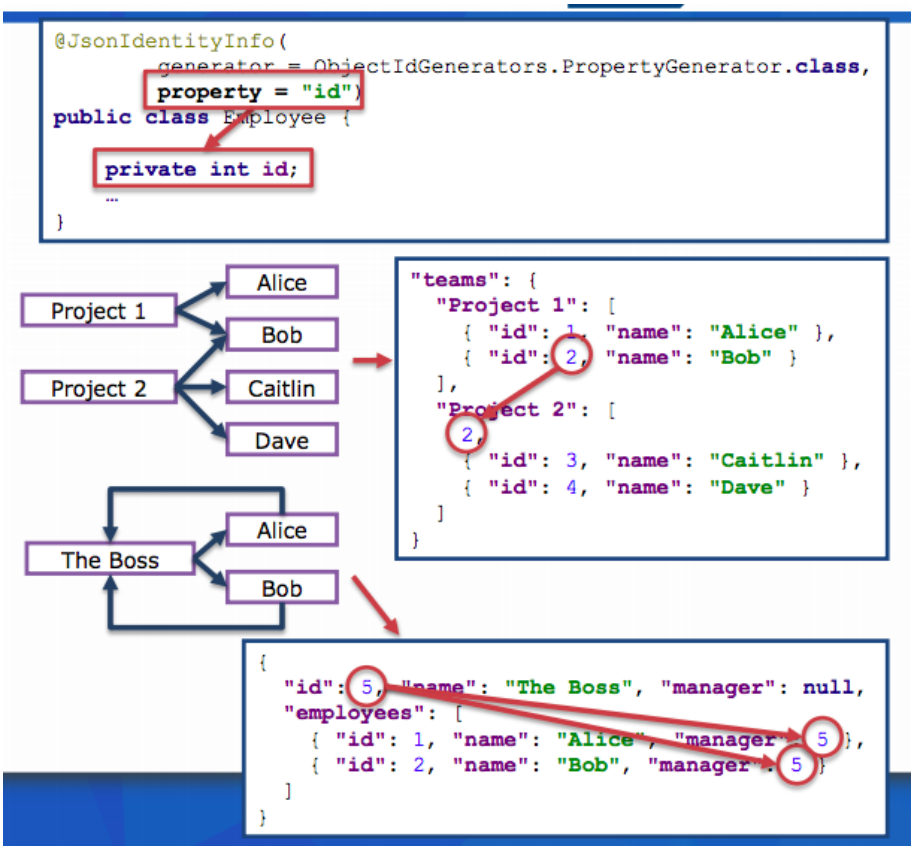


- Scenario: Suppose we marshall an object graph containing cyclic references. What will happen?



Object References - `@JsonIdentityInfo`

- Solution: We can use the `@JsonIdentityInfo` to specify an `id` property which will be unique in a given object graph.
    - When an object is first marshalled, its complete state is written out.
    - If the same object is marshalled again, only its id is written out.
- Jackson can use these ids to reconstitute the exact object graph when unmarshalling.

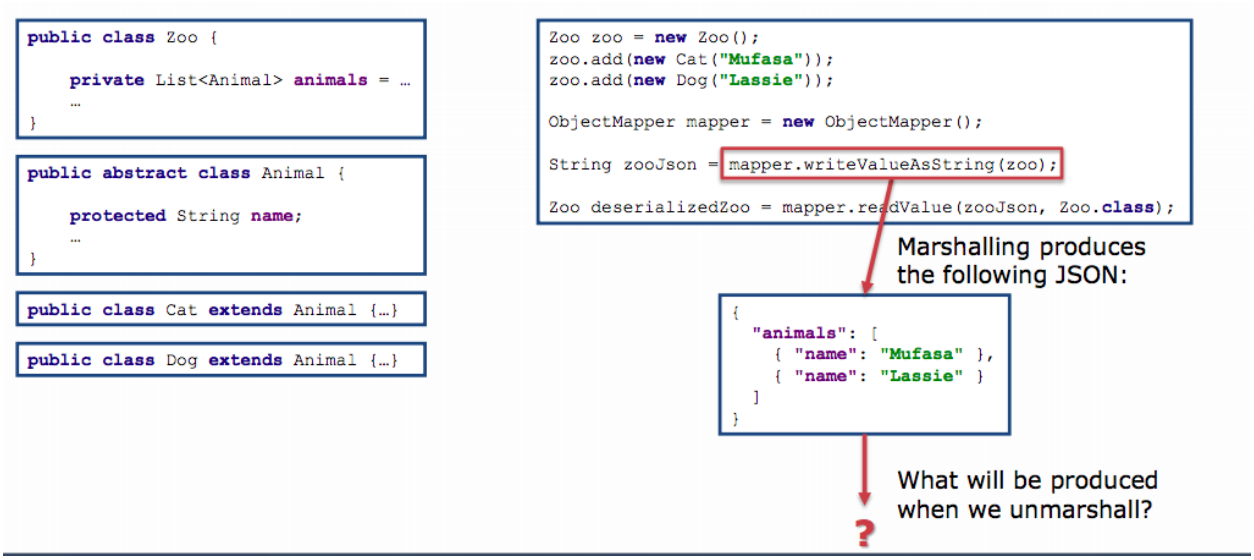


Object References - `@JsonIgnore`

- To handle cyclic references, in addition to `@JsonIdentityInfo` we can also use `@JsonIgnore`.
    - This will ignore the marked property when marshalling / unmarshalling.
    - We can annotate one side of the dependency to break the cycle.
- If we do this, we will need to ensure our own code can handle rebuilding of the cyclic references where necessary.

Polymorphism

- Consider the following code:



- Solution: We can ensure the *type* information is marshalled along with the object data itself.
  - This information can then be used to select the correct class during the unmarshalling process.
- Two approaches:
  - Supply the Java class names using `@JsonTypeInfo`.
  - Supply our own type names using `@JsonTypeInfo` and `@JsonSubTypes`.

Benefits & drawbacks of each approach?

JAX-RS Integration – Server

- Include `resteasy-jackson2-provider` as a dependency to enable Jackson and integration with JAX-RS.
    - Explicitly adding the Jackson `MessageBodyReader` and `MessageBodyWriter` classes is neither required on the client nor the server.
- Use `@Produces/@Consumes` annotations with the `application/json` MIME type.

```
<!- RESTEasy Jackson (JSON) integration -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson2-provider</artifactId>
    <version>${resteasy.version}</version>
</dependency>
```

```
@Path("/hello")
public class HelloJacksonResource {

    @Path("/book")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Book echoBook(Book book) {

        return book;

    }
}
```

JAX-RS Integration – Client

- Include `resteasy-jackson2-provider` as a dependency to enable Jackson and integration with JAX-RS.
    - Explicitly adding the Jackson `MessageBodyReader` and `MessageBodyWriter` classes is neither required on the client nor the server.
- Use `.accept(MediaType.APPLICATION_JSON)` to request that the server returns JSON.
- Use `Entity.json(obj)` to make sure the the obj is serialized as JSON.
- Nothing special required for `Response.readEntity(…)`.

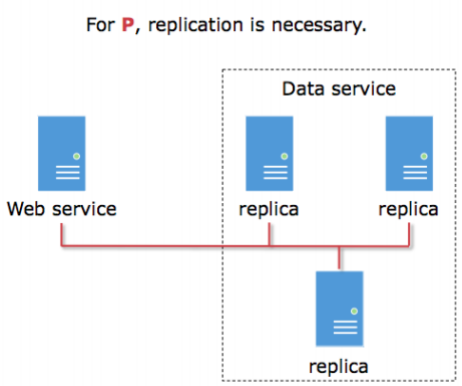
```
Client client = ClientBuilder.newClient();

Book book = new Book("The Neverending Story", Genre.Fantasy);

Response response = client.target("http://.../hello/book")
        .request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.json(book));

Book responseBook = response.readEntity(Book.class);
```

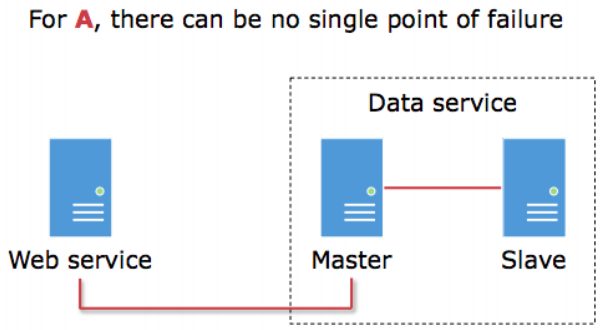
Data Management

Distributed Data Architectures

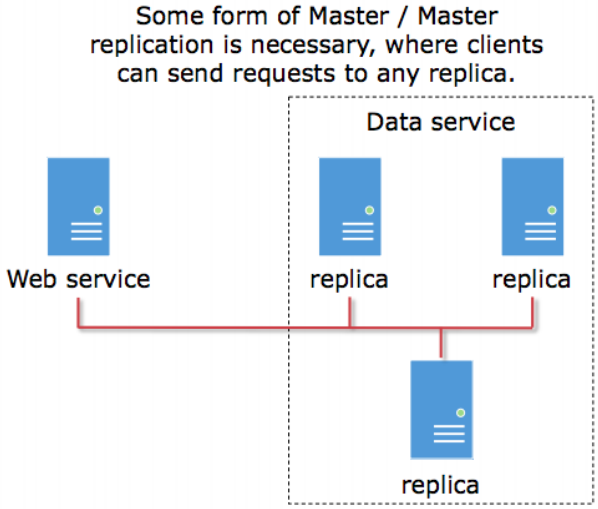
CAP Theorem

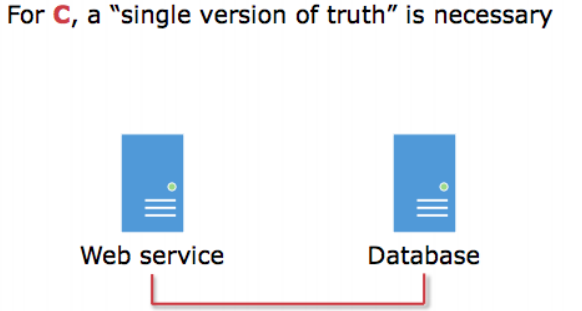
- The CAP theorem states that a distributed system cannot simultaneously be **c**onsistent, **a**vailable and tolerant against network **p**artition failures.



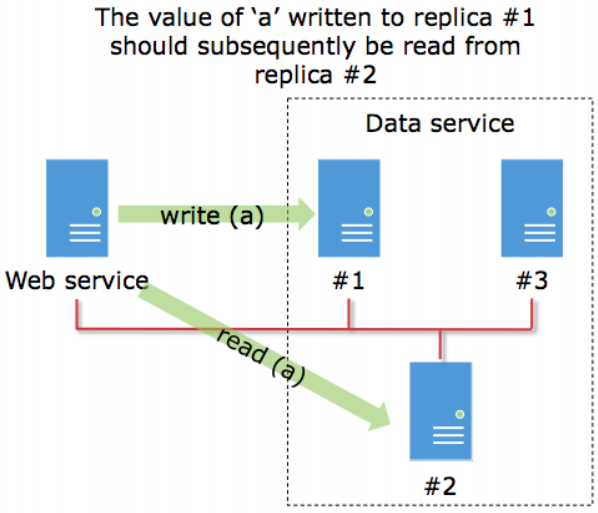


Master / Slave, where clients send all requests to a single master, are unsuitable.



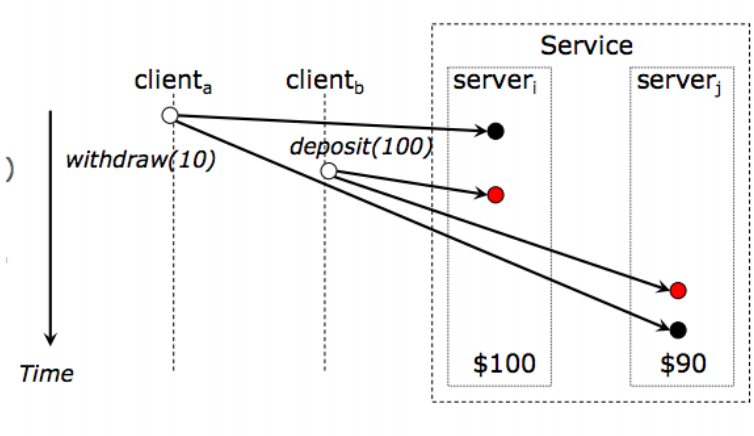


In practice, a compromise needs to be made among the requirements for data consistency, availability and tolerance of network partitions.



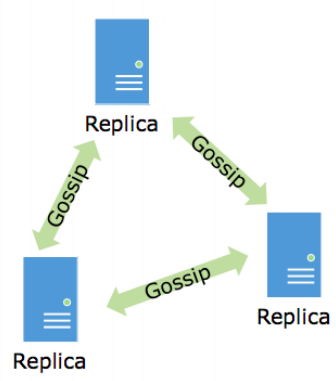
Active Replication

- Active replication involves clients sending requests to all replicas.
- Atomic multicast protocols and middleware are used to ensure order and atomicity.
    - **Order:** Given invocations op(arg) and op'(arg') by $client_a$ and $client_b$, if replicated servers i and j process the invocations they do so in the same order.
    - **Atomocity:** Given invocation op(arg) by $client_a$, if one server replica processes the request then every non-crashed replica also processes the request.



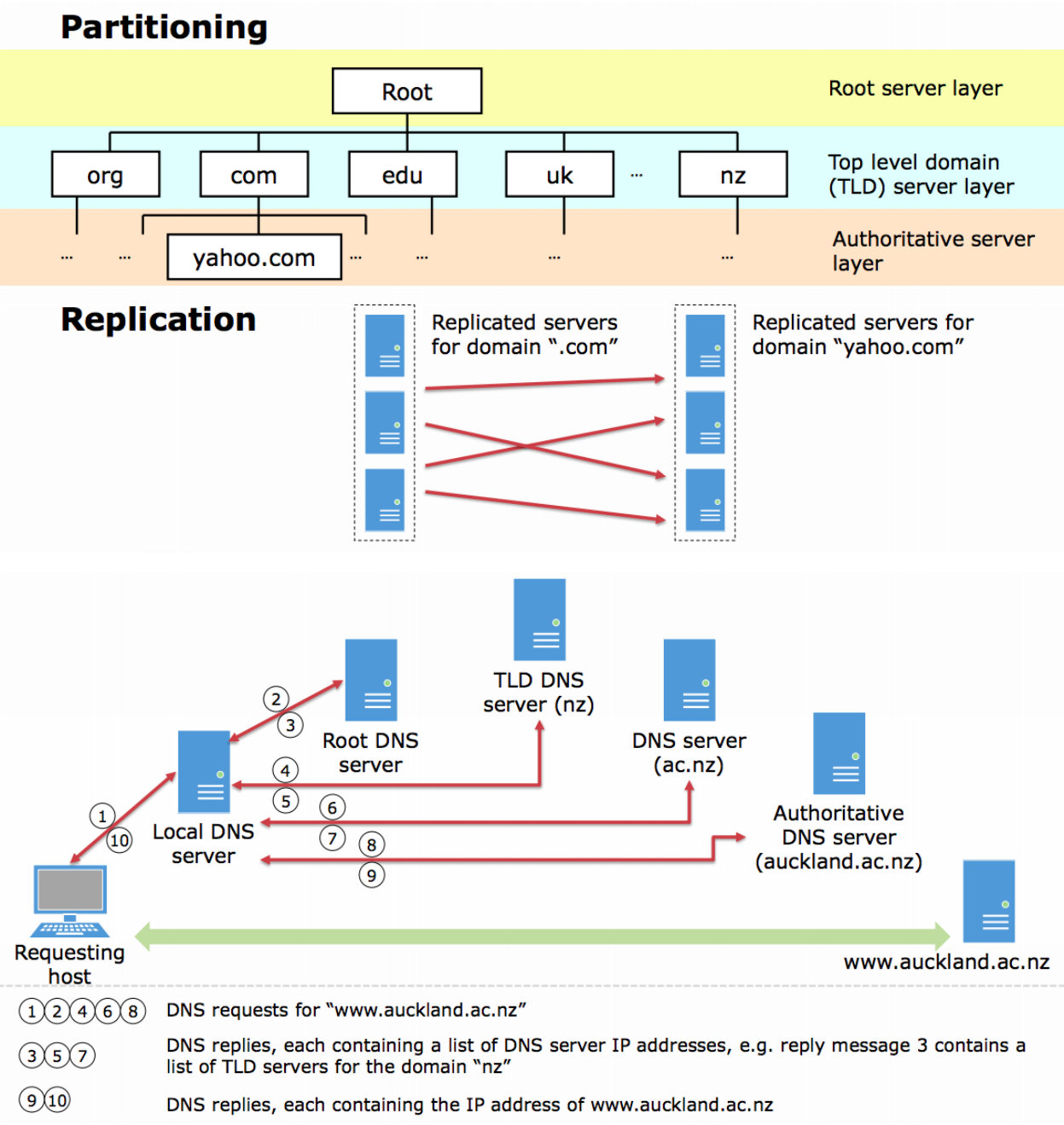
Master / Master Replication

- In practice, Master / Master replication is used where temporary inconsistency can be tolerated.
- Replicas "gossip" synchronisation messages to achieve eventual consistency.

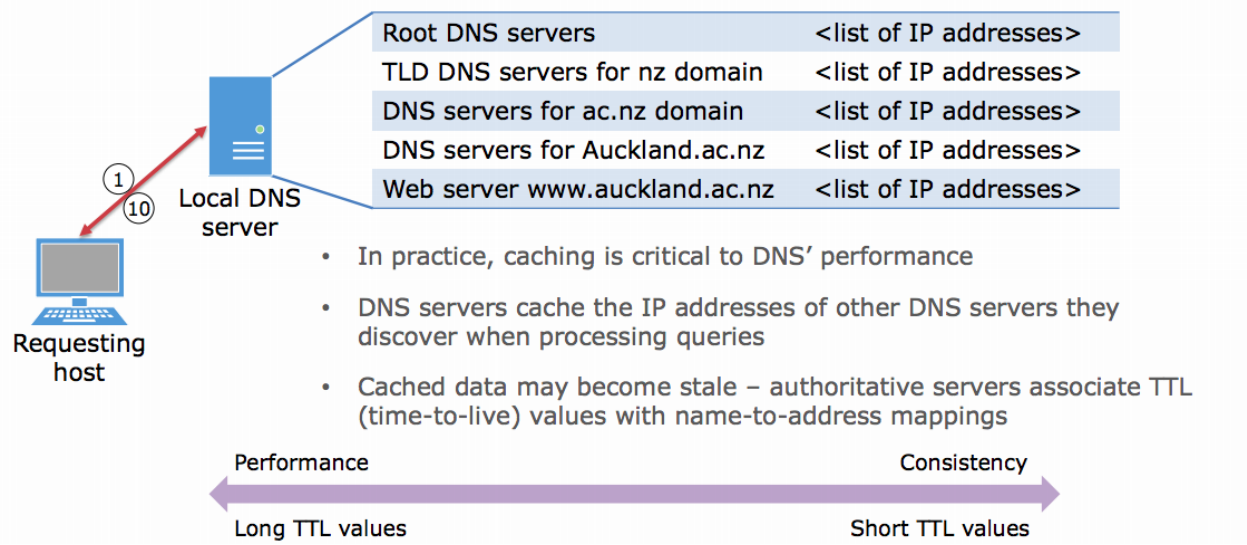


DNS: A Distributed Data Store

- The DNS (Domain Name System) is the Internet's lookup service that maps domain names to IP addresses.
- Key non-functional requirements for the DNS include scalability and availability.
    - To meet these requirements, DNS has been designed using replication, partitioning and caching tactics.

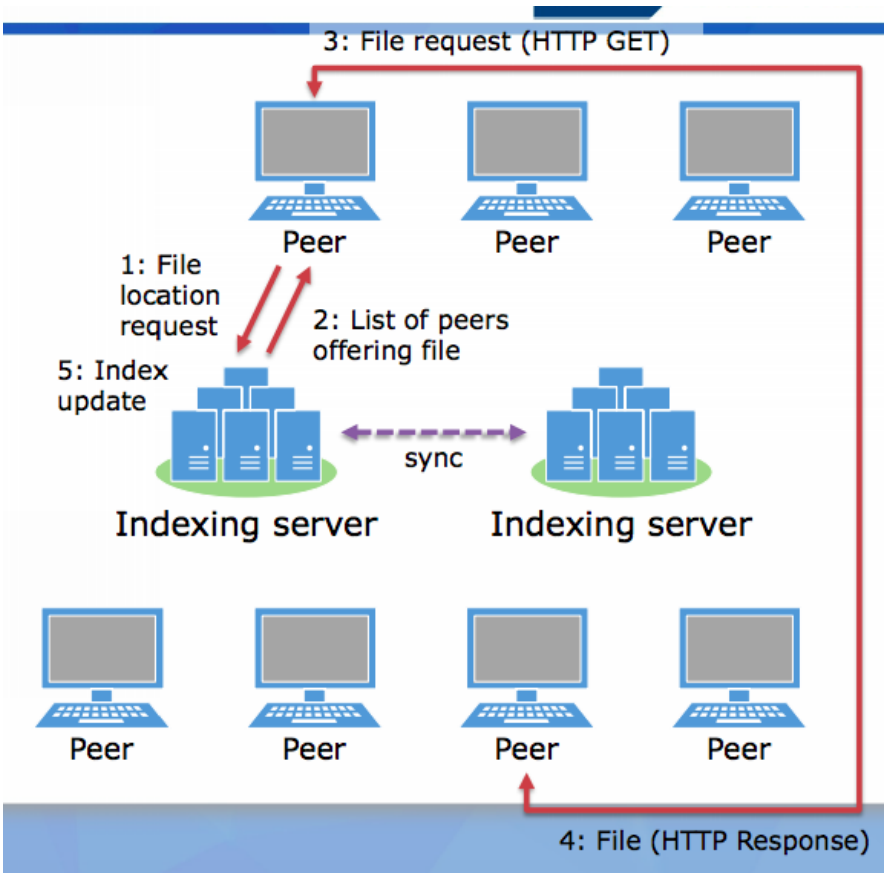


Caching

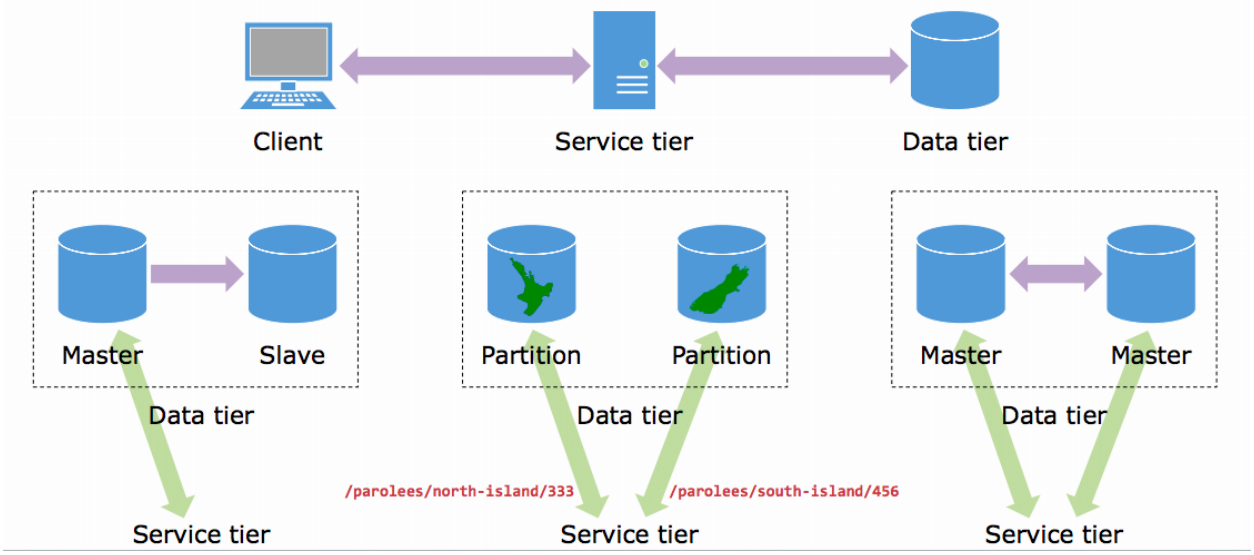


Distributed File Sharing

- Peers store content
- Indexing servers do not store content, but mappings identifying which peers store which files.
- Distributed content is immutable and arbitrarily replicated.
- Indexing servers replicate mappings – but don't guarantee consistency.



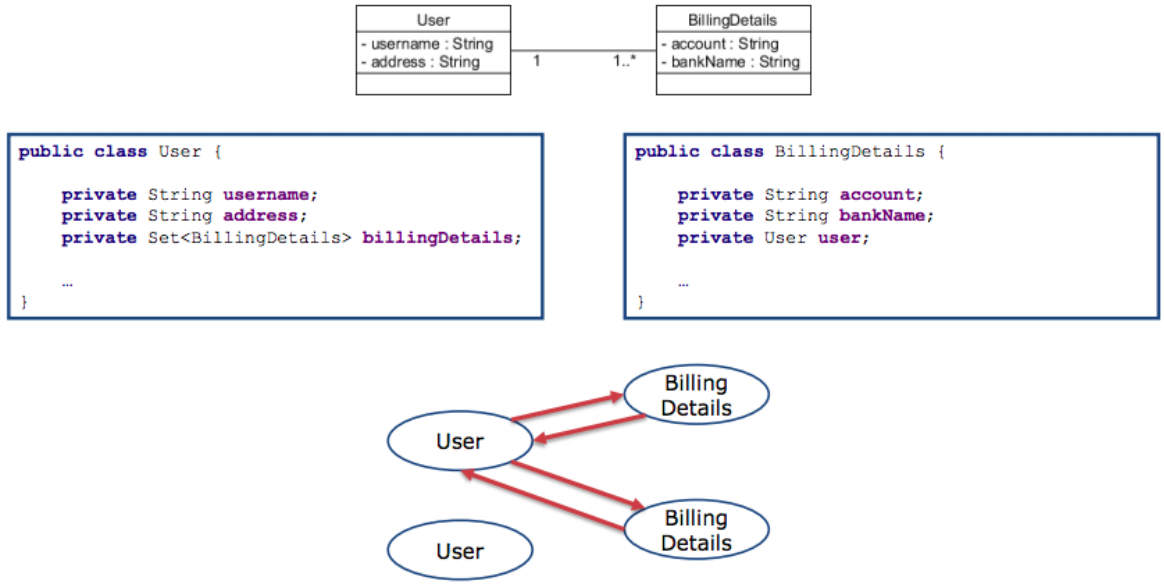
Client / Server Information Systems



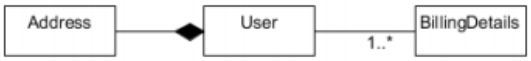
The Paradigm Mismatch: OO vs Relational Model

- There are sound reasons to use relational databases, but the mapping of objects to relational tables isn't seamless.
- Differences between the two paradigms include:
  - Granularity
  - Subtyping
  - Identity
  - Associations
  - Data navigation

Object Orientation and the Relational Model



Mismatch #1: Granularity



Option 1: Introduce an ADDRESS table

ADDRESSES

ID	ADDRESSNUMBER	ADDRESSSTREET	ADDRESSCITY
1	742	Evergreen Terrace	Springfield

USERS

ID	USERNAME	ADDRESS_ID
1	Homer Simpson	1

Option 2: Map Address data to the USERS table

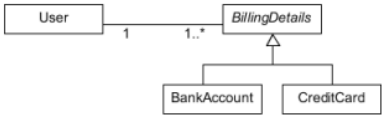
USERS

ID	USERNAME	ADDRESS NUMBER	ADDRESSSTREET	ADDRESSCITY
1	Homer Simpson	742	Evergreen Terrace	Springfield

- We often have more classes in our system than we have tables.
- Finer grained classes offer more potential for reuse and better domain models.
- Fewer tables mean better performance.

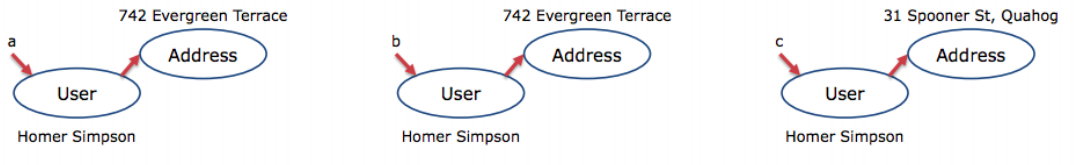
Mismatch #2: Subtyping

- Object-oriented inheritance raises basic questions for persisting using the relational model:
  - How can an inheritance hierarchy be mapped to a relational schema?
  - Polymorphism
    - How can polymorphic queries be written?
    - How can polymorphic associations be represented?



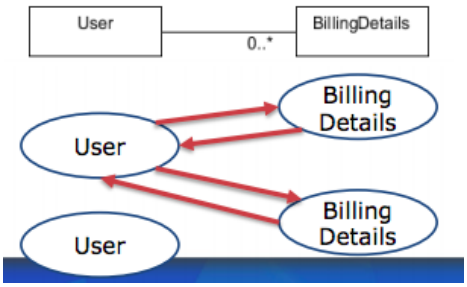
Mismatch #3: Identity

- In an object-oriented application, there are two notions of **sameness**
  - Instance identity (==)
  - Instance equality (`equals()`)
- For a relational table row, identity is defined by the row's primary key value.



ID	USERNAME	ADDRESS NUMBER	ADDRESSSTREET	ADDRESSCITY
1	Homer Simpson	742	Evergreen Terrace	Springfield

Mismatch #4: Associations



Object Orientation:

- Represents associations using object references
  - References are inherently directional.
  - Bidirectional associations need to be defined twice – once in each participating class.

Relational Databases:

- Use a foreign key-constrained column to represent an association.
  - Navigation in a particular direction has no meaning for the relational model.

USERS

ID	USERNAME	ADDRESS
1	Homer Simpson	742 Evergreen Terrace, Springfield
2	Marge Simpson	742 Evergreen Terrace, Springfield

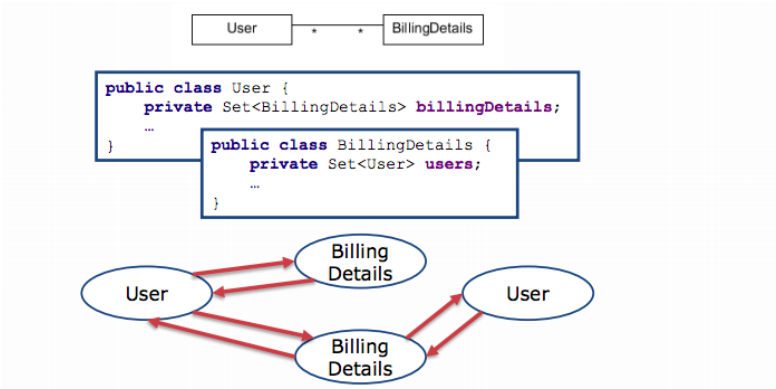
BILLINGDETAILS

ID	ACCOUNT	BANKNAME	USER_ID
1	Current	Bank of America	1
2	Credit	Bank of America	1

```
select * from USERS u
  left outer join BILLINGDETAILS bd
    on u.ID = bd.USER_ID;
```

USERNAME	ADDRESS	ACCOUNT	BANKNAME
Homer Simpson	742 Evergreen Terrace, Springfield	Current	Bank of America
Homer Simpson	742 Evergreen Terrace, Springfield	Credit	Bank of America
Marge Simpson	742 Evergreen Terrace, Springfield		

Many-to-Many Associations




```
create table USER_BILLINGDETAILS (
    USER_ID bigint,
    BILLINGDETAILS_ID bigint
    primary key (USER_ID, BILLINGDETAILS_ID),
    foreign key (USER_ID) references USERS,
    foreign key (BILLINGDETAILS_ID) references BILLINGDETAILS
)
```

USERS

ID	USERNAME	ADDRESS
1	Homer Simpson	742 Evergreen Terrace, Springfield
2	Marge Simpson	742 Evergreen Terrace, Springfield

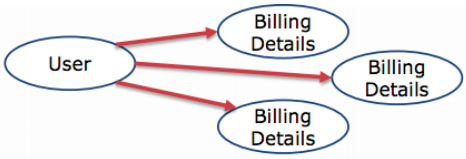
BILLINGDETAILS

ID	ACCOUNT	BANKNAME
1	Current	Bank of America
2	Credit	Bank of America

USER_BILLINGDETAILS

USER_ID	BILLINGDETAILS_ID
1	1
1	2
2	1

Mismatch #5: Data Navigation



When working with objects, it's natural to "walk the object network", navigating from one instance to the next.

```
someUser.getBillingDetails().iterator().next();
```

```
select * from USERS u where u.ID = 123
```

```
select * from BILLINGDETAILS bd where bd.USER_ID = 123
```

This piecemeal style of data access is inefficient when using a relational database. This is the n + 1 selects problem

Where you expect to retrieve a User and then subsequently visit each associated BillingDetails object, it is better to load this portion of the object graph *before* navigating it

```
select * from USERS u
  join BILLINGDETAILS bd
    on bd.USER_ID = u.ID
where u.ID = 123
```

But, loading more data than required wastes memory. When using joins, large Cartesian product results are possible …

USERS

ID	USERNAME	ADDRESS
1	Homer Simpson	742 Evergreen Terrace, Springfield
2	Marge Simpson	742 Evergreen Terrace, Springfield

ORDERS

ID	DESCRIPTION	USER_ID
1	Beer	1
2	Burgers	1
3	Chips	1

BILLINGDETAILS

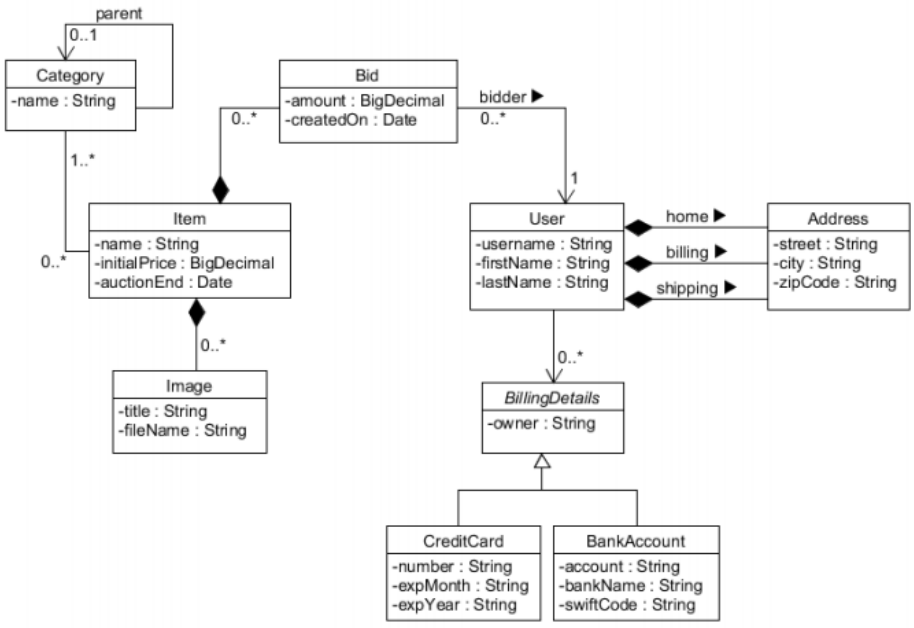
ID	ACCOUNT	BANKNAME	USER_ID
1	Current	Bank of America	1
2	Credit	Bank of America	1

```
select u.USERNAME, u.ADDRESS, bd.ACCOUNT, bd.BANKNAME, o.DESCRIPTION
    from USERS u
        left outer join BILLINGDETAILS bd on bd.USER_ID = u.ID
        left outer join ORDERS o on o.USER_ID = u.ID
    where u.ID = 1;
```

USERNAME	ADDRESS	ACCOUNT	BANKNAME	DESCRIPTION
Homer Simpson	742 Evergreen …	Current	Bank of America	Beer
Homer Simpson	742 Evergreen …	Current	Bank of America	Burgers
Homer Simpson	742 Evergreen …	Current	Bank of America	Chips
Homer Simpson	742 Evergreen …	Credit	Bank of America	Beer
Homer Simpson	742 Evergreen …	Credit	Bank of America	Burgers
Homer Simpson	742 Evergreen …	Credit	Bank of America	Chips

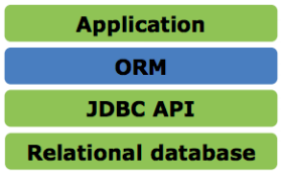
Domain Models

Persistence shouldn't leak into the domain model classes. A domain model that is decoupled from a persistence framework is easier to test, and domain classes are more reusable.



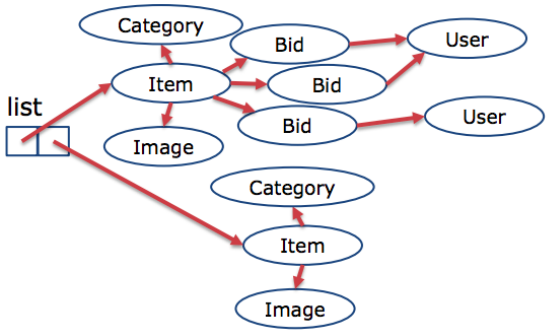
ORM (Object Relational Mapping)

- ORM is the **automated persistence of objects in an OO application to the tables in a SQL database.**



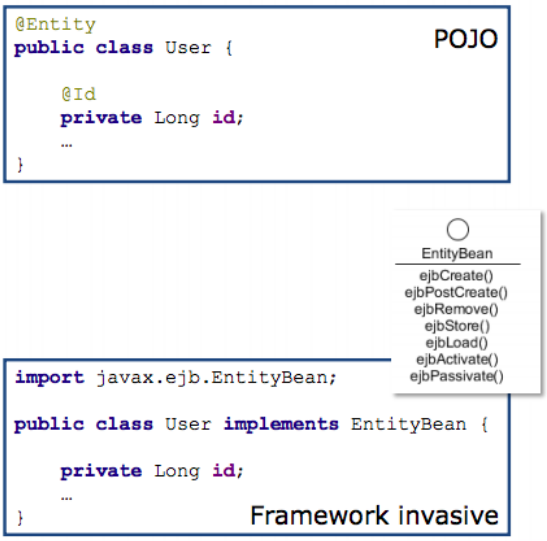
- ORM technology is relatively mature and is widely adopted by applications that operate on domain models.

```
EntityManager em = …
List<Item> items = em
    .createQuery("from Item", Item.class)
    .getResultList();
```



Java Persistence API (JPA)

- JPA is an ORM specification that defines:
  - A facility for specifying the mapping of persistent classes to a database schema.
  - An API for performing CRUD operations on persistent instances.
  - A language (JPQL) and API for specifying queries using classes and properties of classes.
  - The behaviour of the persistence engine in performing dirty checking, lazy association fetching, caching, etc.
  - Lightweight, uses annotations on Java objects.



Transparency and Automation

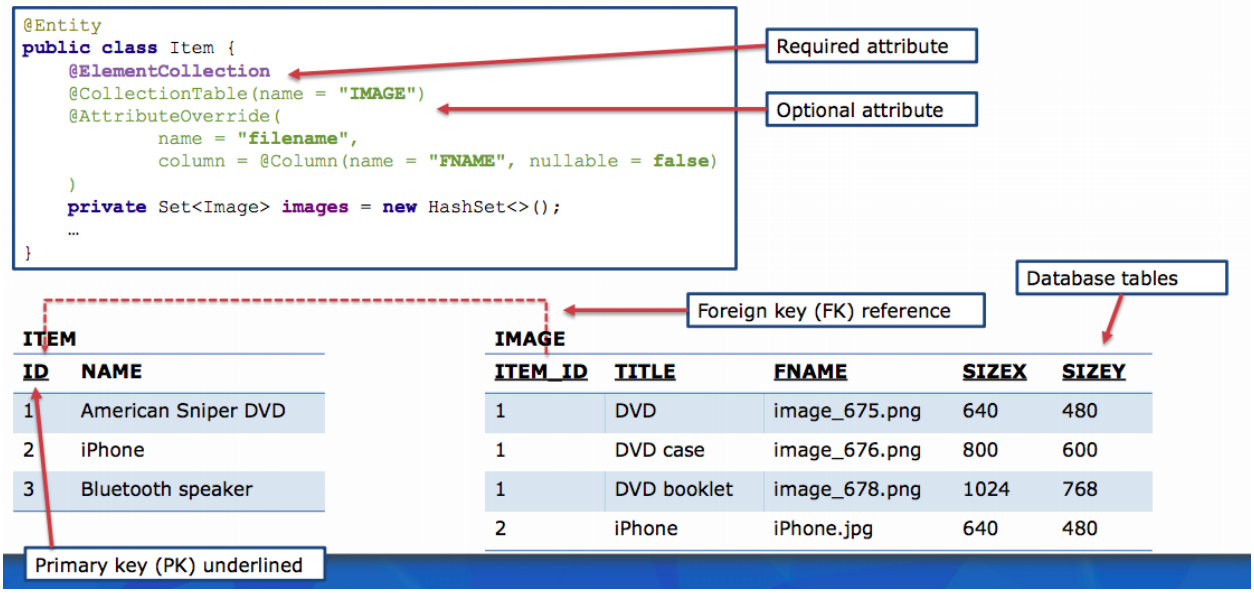
- JPA aims to provide transparent and automated persistence.
  - **Transparency** provides for a separation of concerns between persistent classes and the persistence mechanism.
  - **Automation** relieves developers of low-level persistence details.
- Using JPA:
  - Minimally, `@Entity` and `@Id` need to be used on a persistent class.
  - Persistent classes must provide a default constructor.
  - Collection attributes must be typed to an interface, not an implementation.
  - Persistent object fields can be accessed directly or be accessor methods.
  - Where accessors are used, the persistent class must conform to JavaBean conventions.

Benefits of using ORM and JPA

- Productivity
  - JPA allows developers to focus on business logic and be relieved of low-level data access concerns.
- Maintainability (modifiability)
  - Automated ORM reduces LOC, making a system easier to understand and easier to change.
- Performance
  - In general, JPA implementations apply performance optimisations; JPA deployments can be configured where necessary to address performance issues.
- Vendor independence
  - JPA is a specification that is implemented by many vendors.
  - Using an ORM promotes portability – it insulates application code from particular RDBMS products.

Mapping Persistent Classes

Notation

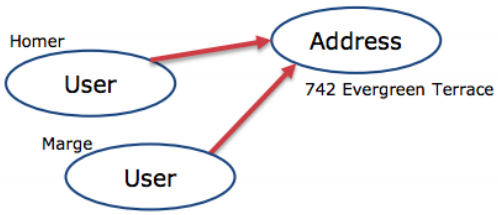


Mapping Value Types

Entity vs Value Types

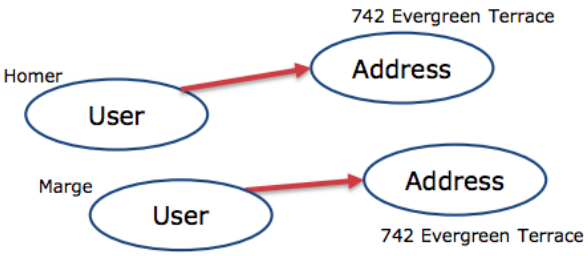
Entity types

- An entity has a **database identity,** allowing the entity to be queried.
- An entity instance has its own lifecycle; Entity instances can exist independently.
- A reference to an entity object is persisted as a foreign key constrained value.



Value types

- A value-type instance forms part of some other object.
- The lifespan of a value-type instance is dictated by its owning object.
- Value-type instances shouldn't be shared.



Mapping Entities and Value Types

```
@Entity
public class User {

    @Id
    @GeneratedValue
    private Long id;

    private String username;
    private String firstName;
    private String lastName;

    private Address homeAddress;
    …

}
```

Entity classes must be annotated with `@Entity`.

Entity classes must have a database identity field, annotated with `@Id`. To have the database generate and assign the value for this field, use `@GeneratedValue`.

Because the `@Id` annotation is used on a field, JPA accesses fields directly (without needing to invoke accessor methods).

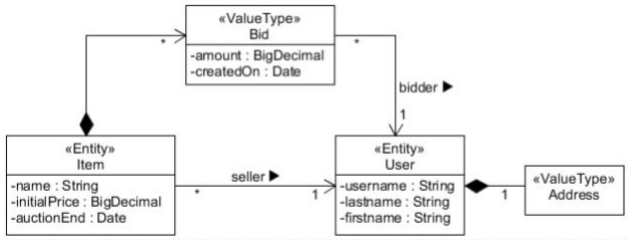
Property-based access can also be specified – requiring use of a get/set pair of methods for each property. JPA will then use these methods instead of accessing fields directly.

```
@Embeddable
public class Address {

    private String street;
    private String city;
    private String zipCode;
    …
}
```

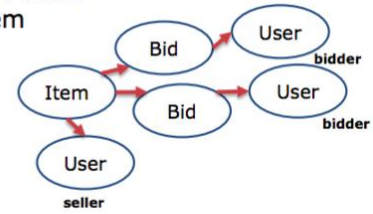
Specify a value-type using the `@Embeddable` annotation

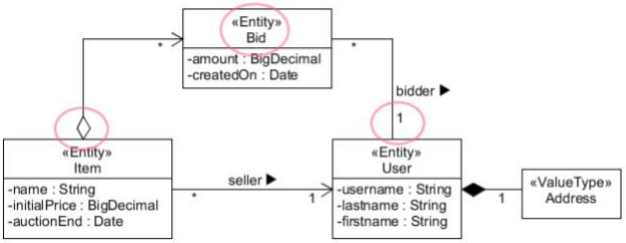
«Table» USERS
ID «PK»
USERNAME
LASTNAME
FIRSTNAME
STREET
CITY
ZIPCODE

Entity vs Value Types

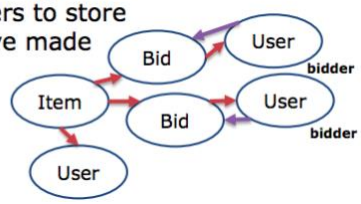


A bid shouldn't exist without an Item





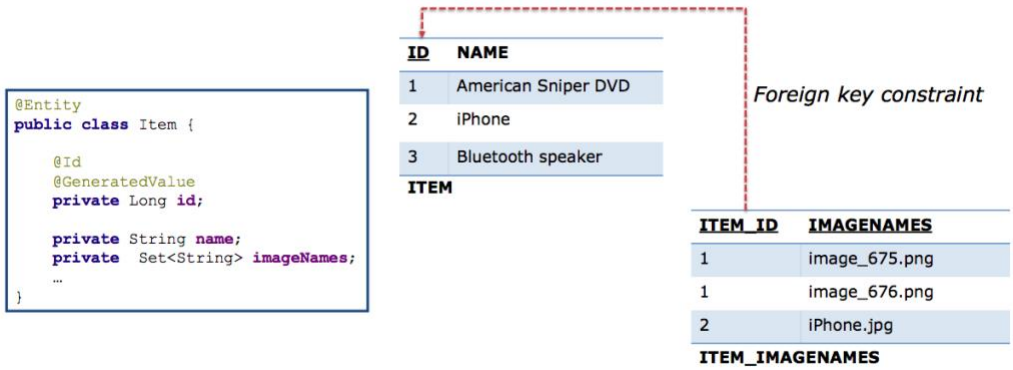
An alternate design that allows for Users to store all Bids they've made

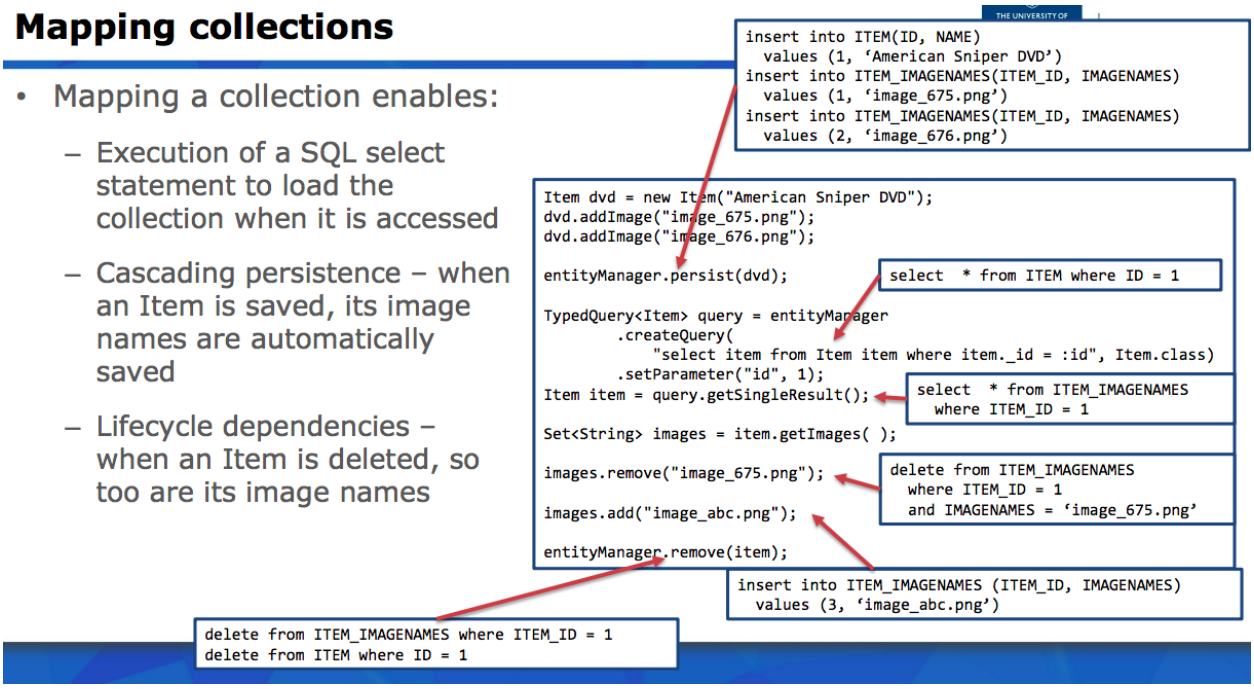


Type Mapping

Field in Owner class	JPA mapping behaviour
Typed to a primitive Java type, or a wrapper class	The field is mapped to a column in the Owner table
The type of the field is a class annotated with `@Embeddable`	The fields of the embeddable are mapped to columns in the Owner table
Annotated with `@Transient`	Fields marked `@Transient` are not mapped
An enumerated type	The field is mapped to a column whose value is the property's ordinal value
An enumerated type annotated with `@Enumerated`	The field is mapped to a column whose value is the field's String representation
A java.util.Date type annotated with `@Temporal(TemporalType.DATE | TemporalType.TIME | TemporalType.TIMESTAMP)`	The Date property is mapped to a column whose date/time precision is determined by the `TemporalType` argument
None of the above	JPA throws an exception

Mapping Collections





Matching a Set

```
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(
            name = "IMAGE",
            joinColumns = @JoinColumn(name = "ITEM_ID"))
    @Column(name = "FILENAME")
    private Set<String> imageNames = new HashSet<>();
    …
}
```

Required for a collection of value-typed elements, `@ElementCollection` is processed to generate a table to store the collection elements.

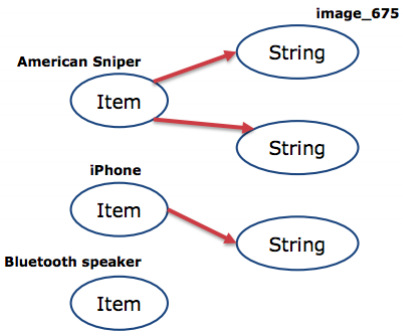
`@CollectionTable` is used to (optionally) override default properties for the table used to store the collection.

ID	NAME
1	American Sniper DVD
2	iPhone
3	Bluetooth speaker

ITEM

ITEM_ID	FILENAME
1	image_675.png
1	image_676.png
2	iPhone.jpg

IMAGE



Collection Interfaces

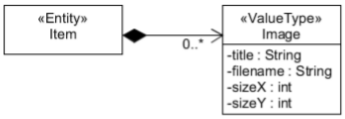
```
<interface-type<generic-type>> collection-name =
    new <implementation-class<generic-type>>( );
```

```
Collection<String> imageNames = new ArrayList<>();
```

- Collections should be typed to an interface.
- The interface determines the collection's mapping to a database.
- JPA expects a suitable implementation, and may wrap it with an implementation to handle dirty checking and lazy loading.

Interface	Suitable implementation	Semantics
`java.util.Set`	`java.util.HashSet`	Set is an unordered collection that can't contain duplicate elements
`java.util.Collection`	`java.util.ArrayList`	Bag semantics – similarly to a Set, a Bag is an unordered collection, but a Bag can contain duplicates
`java.util.List`	`java.util.ArrayList`	An ordered collection, the position of each element is preserved when persisted
`java.util.Map`	`java.util.HashMap`	A set of key/value pairs that are persisted in a database

Collections of Value Types




```
someItem.getImages().add( new Image( "Foo", "foo.jpg", 640, 480 ) );
someItem.getImages().add( new Image( "Bar", "bar.jpg", 800, 600 ) );
someItem.getImages().add( new Image( "Baz", "baz.jpg", 1024, 768 ) );
someItem.getImages().add( new Image( "Baz", "baz.jpg", 1024, 768 ) );
```

```
@Embeddable
public class Image {

    @Column(nullable = false)
    private String fileName;
    @Column(nullable = false)
    private String title;
    private int sizeX;
    private int sizeY;

    …

    @Override
    public boolean equals(Object o) {…}

    @Override
    public int hashCode() {…}
}
```

```
@Entity
public class Item {
    …
    private Set<Image> images = new HashSet<>();
```

- For components, `equals()` should be overridden to test for **value** equality.
- Whenever `a.equals(b)`, `a.hashCode()` must generate the **same** code as `b.hashCode()`.
- `equals()` and `hashCode()` should therefore be overridden together.

- IntelliJ and Eclipse can make equals() and hashCode() for you.

Mapping Collections of Value Types

```
@Entity
public class Item {
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @AttributeOverride(
            name = "filename",
            column = @Column(name = "FNAME", nullable = false)
    )
    private Set<Image> images = new HashSet<>();
    …
}
```

Similarly to before, `@ElementCollection` is used to specify a collection of embedded components.

`@CollectionTable` changes the default name for the collection table (ITEM_IMAGES) to IMAGE.

`@AttributeOverride` is used to override a column name in the IMAGE table. By default, each Image attribute maps to a column of the same name. With `@AttributeOverride`, Image's filename attribute maps to FNAME.

ITEM

ID	NAME
1	American Sniper DVD
2	iPhone
3	Bluetooth speaker

IMAGE

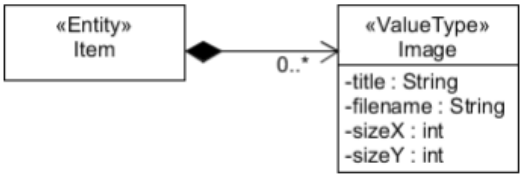
ITEM_ID	TITLE	FNAME	SIZEX	SIZEY
1	DVD	image_675.png	640	480
1	DVD case	image_676.png	800	600
1	DVD booklet	image_678.png	1024	768
2	iPhone	iPhone.jpg	640	480

Mapping Entity Associations

Entity Associations

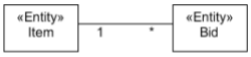
Fully dependent lifecycle:

- Part components (Images) are fully dependent on their owning entities (Items).
- Components are saved, updated and removed with their own entities.
- Collections of components are appropriate for dependent lifecycles.

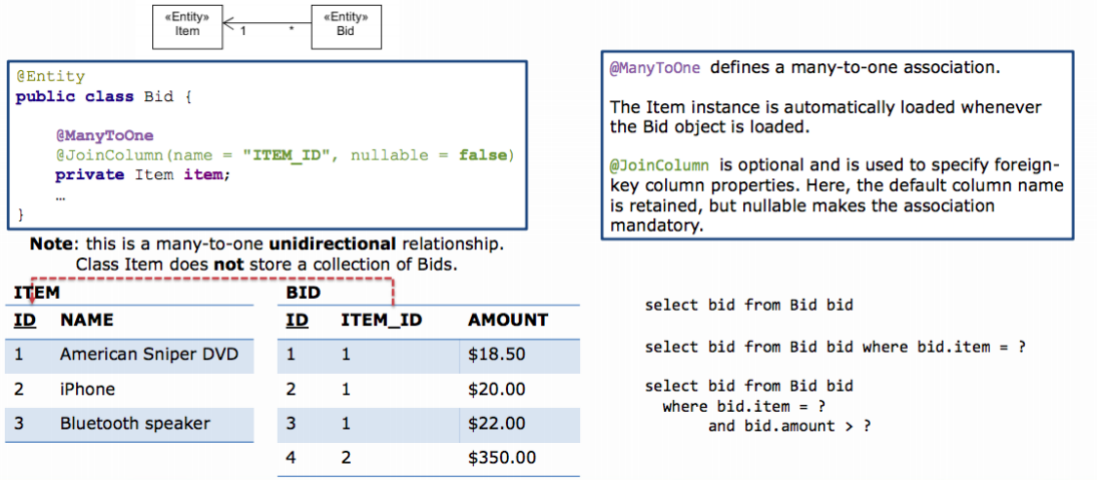


Independent lifecycle

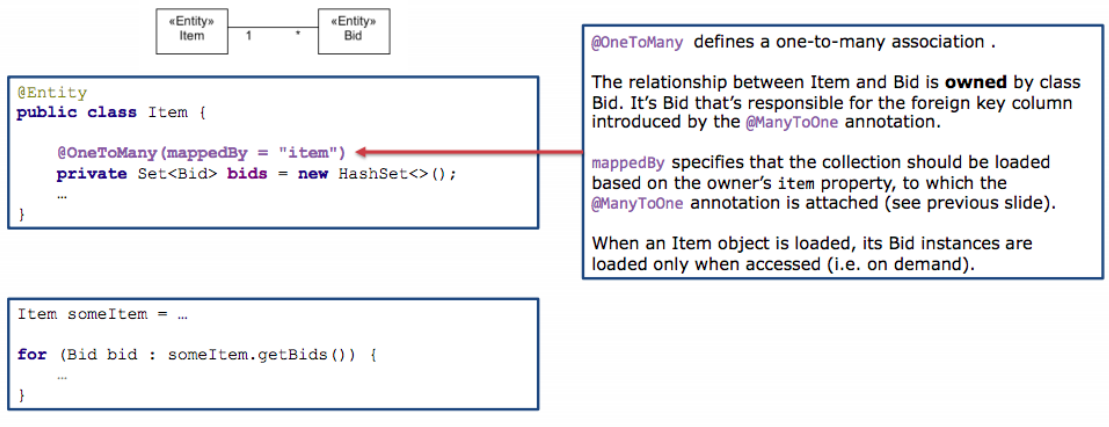
- Two entities are related, but they don't have a dependent lifecycle.
- One instance can be saved, updated or removed independently of the other.
- More fine-grained control is required to model entity associations.



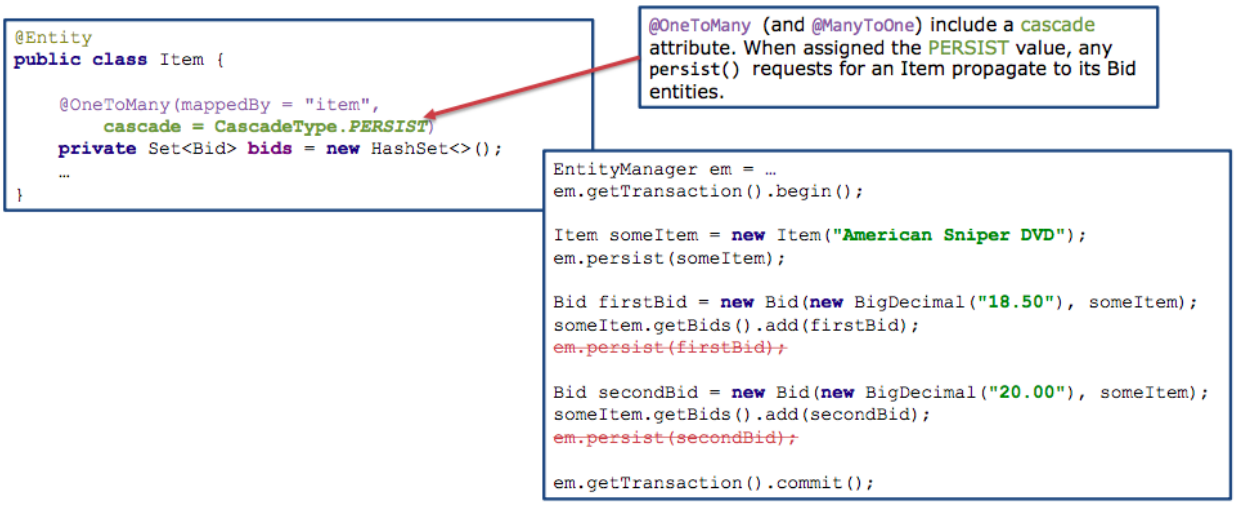
Many-to-One Unidirection Association



Making the Association Bidirectional



Cascading State



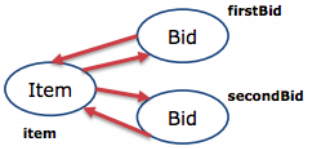
- Allows developers to work with domain model and be less concerned with the system specials.

Cascading Deletion

```
EntityManager em = null;
em.getTransaction().begin();

Item item = em.find(Item.class, 1);

for (Bid bid : item.getBids()) {
    em.remove(bid);
}
em.remove(item);

em.getTransaction().commit();
```

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
        cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Set<Bid> bids = new HashSet<>();
    …
}
```

```
EntityManager em = null;
em.getTransaction().begin();

Item item = em.find(Item.class, 1);
em.remove(item);

em.getTransaction().commit();
```



ITEM

ID	NAME
~~1~~	~~American Sniper DVD~~
2	iPhone
3	Bluetooth speaker

BID

ID	ITEM_ID	AMOUNT
~~1~~	~~1~~	~~$18.50~~
~~2~~	~~1~~	~~$20.00~~
3	2	$350.00

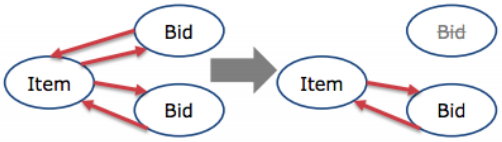
- Specify multiple cascading types using a list.

Orphan Removal

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
        cascade = CascadeType.PERSIST, orphanRemoval = true)
    private Set<Bid> bids = new HashSet<>();
    …
}
```

`@OneToMany` includes an `orphanRemoval` tag that subsumes `CascadeType.REMOVE`.

When an entity stored in a collection is removed, the entity's row in its database table is removed automatically.




```
EntityManager em = null;
em.getTransaction().begin();

Item item = em.find(Item.class, 1);

Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);
firstBid.setItem(null);

em.getTransaction().commit();
```

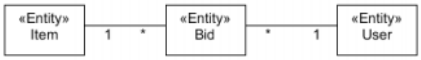
ITEM

ID	NAME
1	American Sniper DVD
2	iPhone
3	Bluetooth speaker

BID

ID	ITEM_ID	AMOUNT
~~1~~	~~1~~	~~$18.50~~
2	1	$20.00
3	2	$350.00

Database:



ITEM

ID	NAME
1	American Sniper DVD

BID

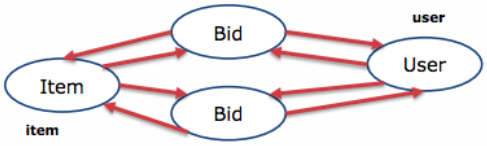
ID	ITEM_ID	USER_ID	AMOUNT
1	1	1	$18.50
2	1	1	$20.00

USER

ID	NAME
1	Alice
2	Bob

```
EntityManager em = null;
em.getTransaction().begin();

Item item = em.find(Item.class, 1);
User user = em.find(User.class, 1);
Bid firstBid = item.getBids().iterator().next();
item.getBids().remove(firstBid);
firstBid.setItem(null);

em.getTransaction().commit();
```

Object graph in memory:



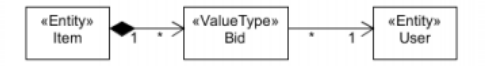
- Inconsistent state when link between bid and item removed.

A simpler design

```
@Entity
public class Item {

    @ElementCollection
    private Set<Bid> bids = new HashSet<>();
    …
}
```

```
@Embeddable
public class Bid {
    private BigDecimal amount;

    @ManyToOne
    private User bidder;
}
```

```
@Entity
public class User {
    private String name;
}
```




```
select bid from Bid bid where bid.bidder = ?
```

- User does not have any reference to bid so if bid deleted, user won't have dangling references.
- Downside: Given user, cannot just get all the bids.

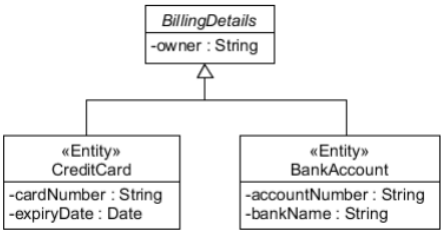
Mapping Inheritance

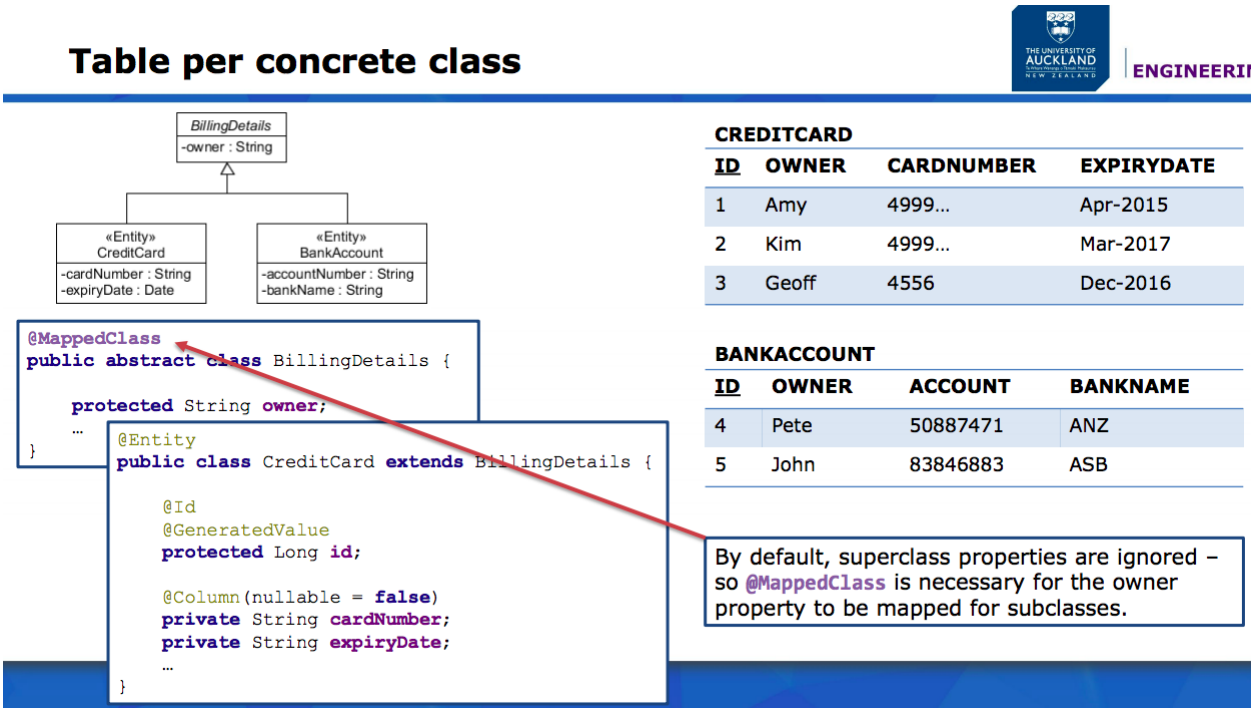
- Inheritance is such a visible structural mismatch between the object-oriented and relational worlds.
    - The OO model supports both has-a and is-a relationships.
    - The relational paradigm offers only has-a.
- There are several strategies for mapping an inheritance hierarchy to a relational schema.
    - The best strategy to use is dependent on application characteristics; each strategy involves trade-offs.
    - The simplest strategy is table per concrete class.

Strategies

Table per Concrete Class

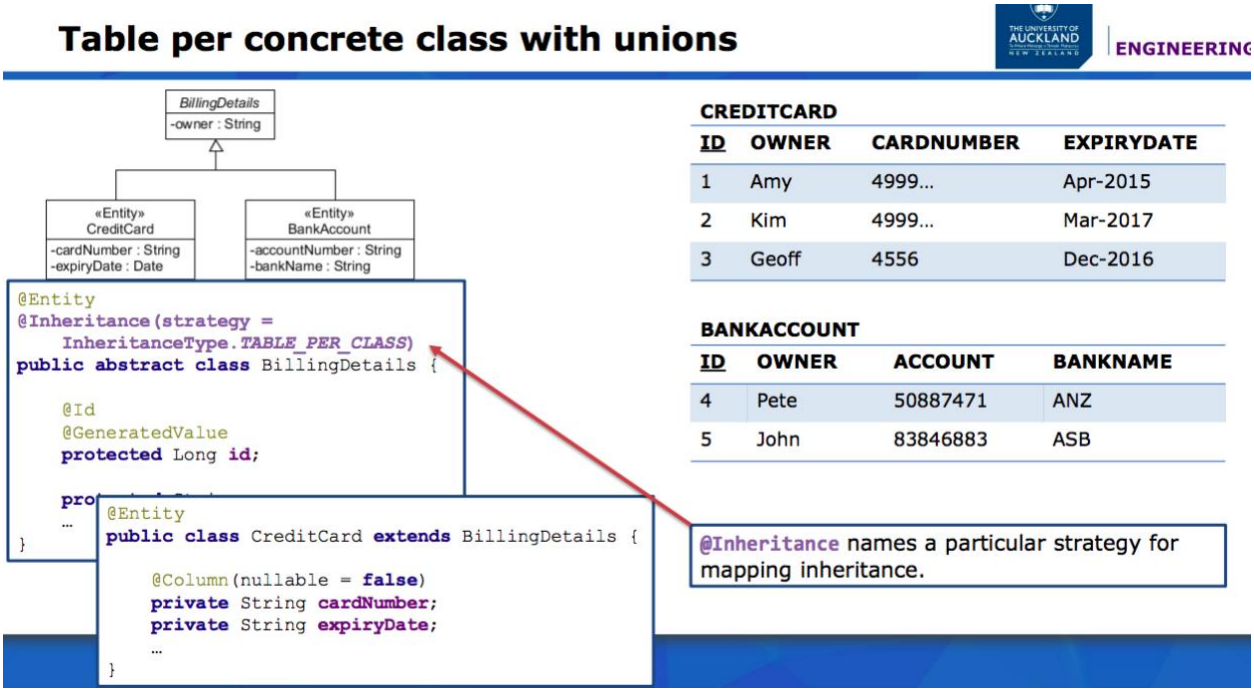
- Construct the database schema.
- How well does the schema support polymorphic queries?
    - `select bd from BillingDetails bd`
- How do you model a polymorphic association?
    - Class User has a many-to-one association with BillingDetails
- What are the limitations of this strategy?





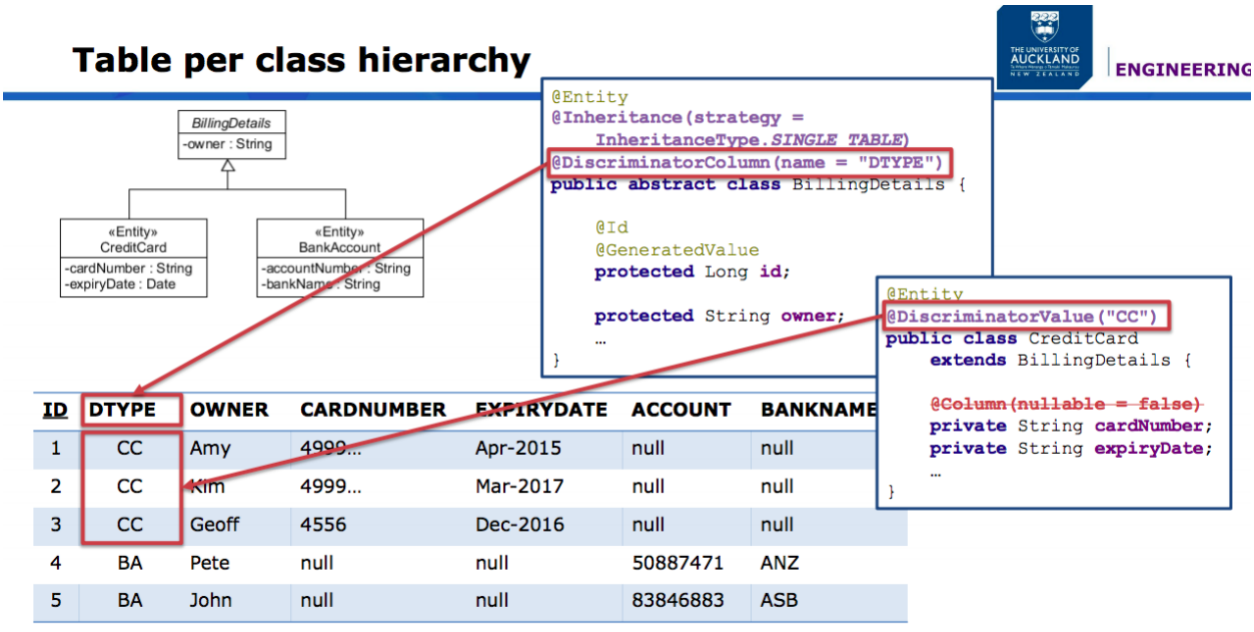
- Generally not used since it doesn't support polymorphic associations and makes schema evolution difficult.

Table per Class with Unions



- Improves strategy 1 by providing for polymorphic associations, but like strategy 1 columns from different tables share the same semantics.

Table per Class Hierarchy



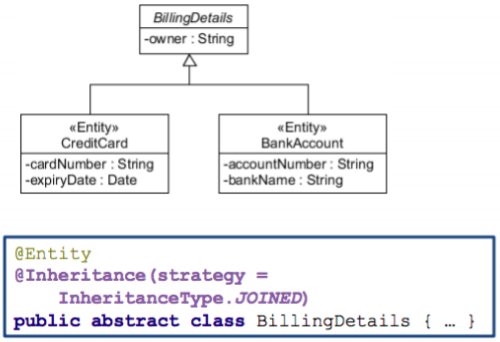
The `DiscrimatorColumn` helps the entity distinguish what type of "concrete" entity it actually is.

This method is simpler because there is only one table and therefore it is easier to write SQL queries for it. It's also more performant because polymorphic and non-polymorphic queries either select all billing details or just one, there's no joins in either case.

The big drawback is comprised data. Violates requirement for third normal form. Since null is used to replace columns that are not needed, we can't use nullable=false.

Table per Class with Joins

- With this strategy, the table of a concrete `@Entity` contains:
  - Columns for each **non-inherited** property, declared by the subclass itself.
  - A primary key that is also a foreign key linking to the superclass table.



BILLINGDETAILS

ID	OWNER
1	Pete
2	Amy
3	Kim
4	John
5	Geoff

CREDITCARD

ID	CARDNUMBER	EXPIRYDATE
2	4999...	Apr-2015
3	4999...	Mar-2017
5	4556	Dec-2016

BANKACCOUNT

ID	ACCOUNT	BANKNAME
1	50887471	ANZ
4	83846883	ASB

- Normalises database schema, no need for null values.
- Makes schema evolution and integrity constraint definition straightforward.
- Performance can be unacceptable for complex class hierarchies.

Polymorphic Queries

Table per Concrete Class

Polymorphic queries – Strategy #1

THE UNIVERSITY OF AUCKLAND | ENGINEERING

Strategy #1:
Table per concrete class

```
select bd from BillingDetails bd
```

```
select
    ID, OWNER, ACCOUNT, BANKNAME
from
    BANKACCOUNT

select
    ID, OWNER, CARDNUMBER, EXPIRYDATE
from
    CREDITCARD
```

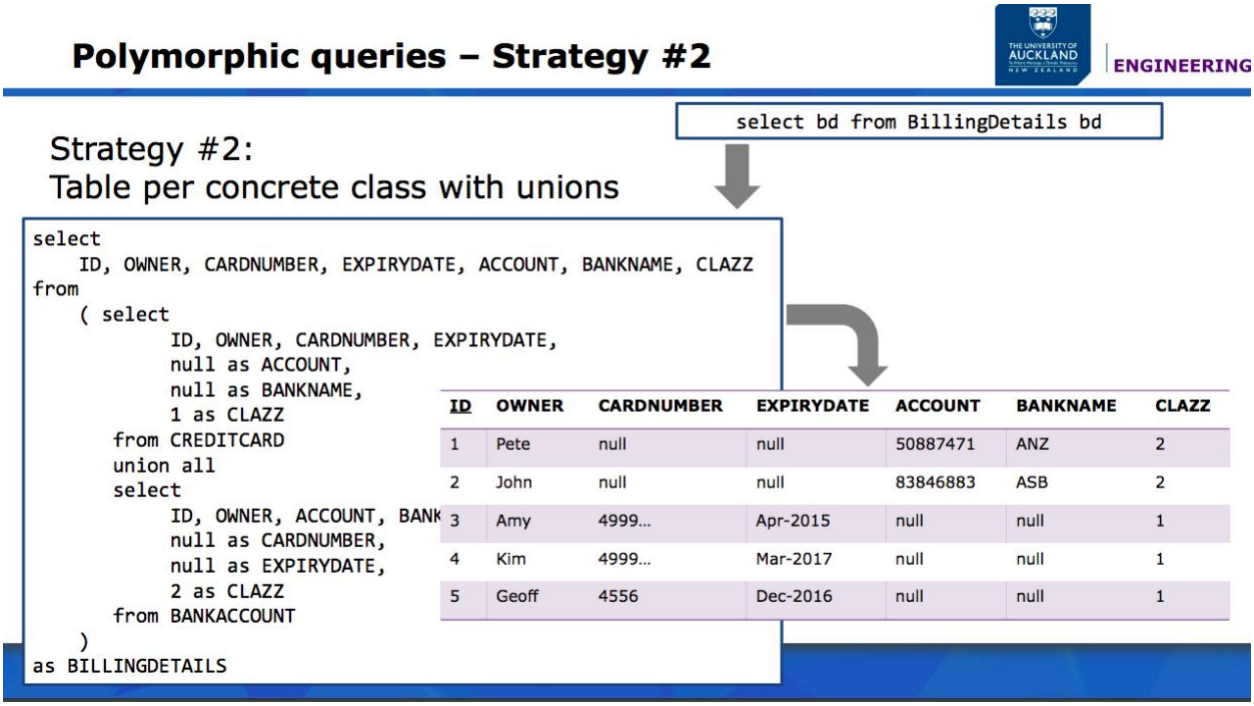
CREDITCARD

ID	OWNER	CARDNUMBER	EXPIRYDATE
1	Amy	4999...	Apr-2015
2	Kim	4999...	Mar-2017
3	Geoff	4556	Dec-2016

BANKACCOUNT

ID	OWNER	ACCOUNT	BANKNAME
4	Pete	50887471	ANZ
5	John	83846883	ASB

Table per Class with Unions



Likely to perform better than strategy one where more than one result is created, one for each subclass and using multiple queries. Gets database to generate one result set rather than running multiple queries and joining in memory.

Table per Class Hierarchy

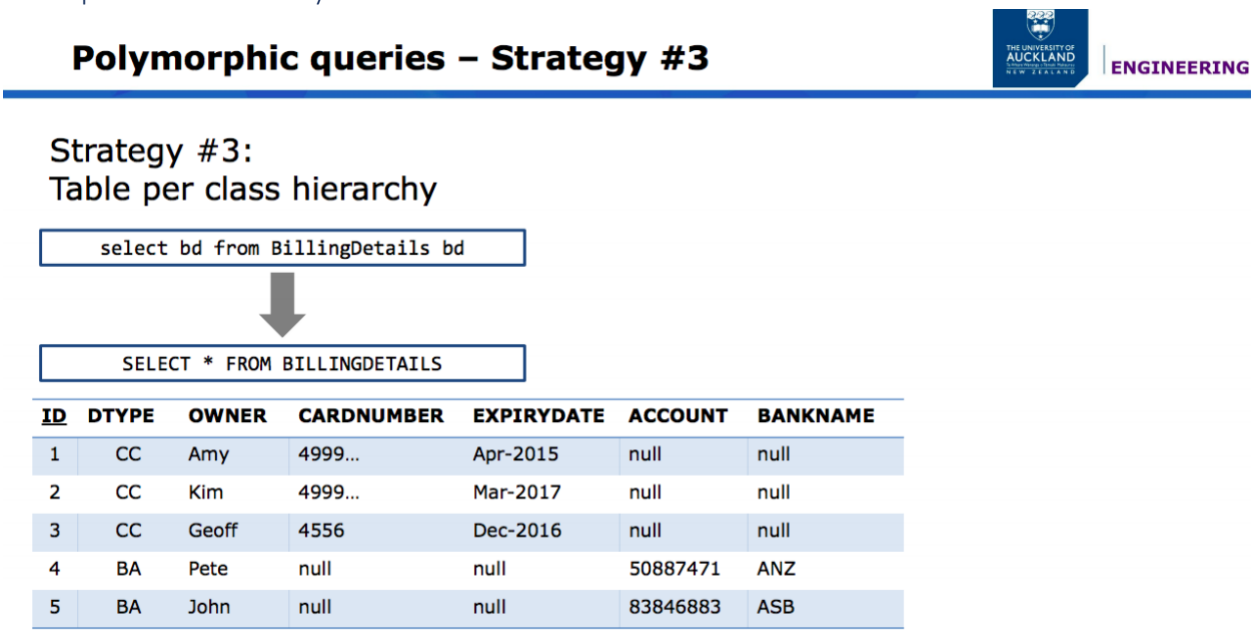


Table per Class with Joins

Polymorphic queries – Strategy #4

ENGINEERING

Strategy #4:
Table per class with joins

```
select bd from BillingDetails bd
```

```
select
    BD.ID, BD.OWNER,
    CC.CARDNUMBER, CC.EXPIRYDATE,
    BA.ACCOUNT, BA.BANKNAME,
    case
        when CC.ID is not null then 1
        when BA.ID is not null then 2
    end as CLAZZ
    from
        BILLINGDETAILS BD
    left outer join
        BANKACCOUNT BA on BD.ID = BA.ID
    left outer join
        CREDITCARD CC on BD.ID = CC.ID
```

ID	OWNER	CARDNUMBER	EXPIRYDATE	ACCOUNT	BANKNAME	CLAZZ
2	Amy	4999...	Apr-2015	null	null	1
3	Kim	4999...	Mar-2017	null	null	1
1	Pete	null	null	50887471	ANZ	2
4	John	null	null	83846883	ASB	2
5	Geoff	4556	Dec-2016	null	null	1

Polymorphic queries – Strategy #4

ENGINEERING

Strategy #4:
Table per class with joins

```
select cc from CreditCard cc
```

```
select
    ID, OWNER, CARDNUMBER, EXPIRYDATE
from
    CREDITCARD
    inner join BILLINGDETAILS on
        CREDITCARD.ID = BILLINGDETAILS.ID
```

BILLINGDETAILS

ID	OWNER
1	Pete
2	Amy
3	Kim
4	John
5	Geoff

CREDITCARD

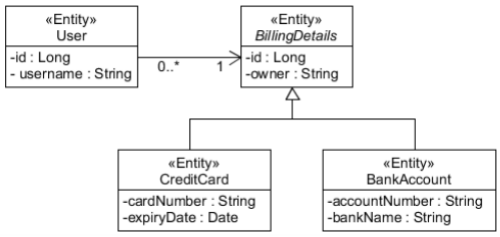
ID	CARDNUMBER	EXPIRYDATE
2	4999...	Apr-2015
3	4999...	Mar-2017
5	4556	Dec-2016

Polymorphic Associations

Table per Concrete Class

Polymorphic associations – Strategy #1

Strategy #1:
Table per concrete class



USER

ID	USERNAME	BILLINGDETAILS_ID

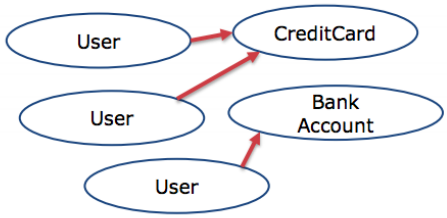
???

CREDITCARD

ID	OWNER	CARDNUMBER	EXPIRYDATE

BANKACCOUNT

ID	OWNER	ACCOUNT	BANKNAME

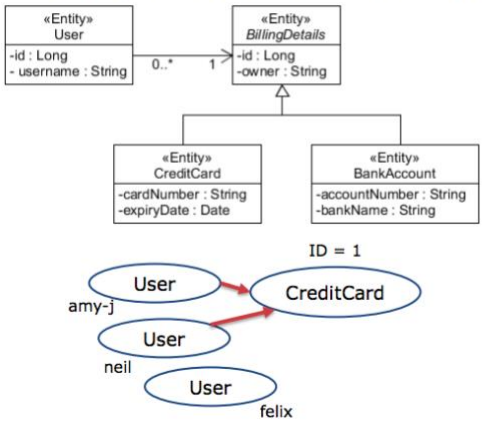


Association not supported.

Table per Class with Unions

Polymorphic associations – Strategy #2

Strategy #2:
Table per concrete class with unions



USER

ID	USERNAME	BILLINGDETAILS_ID
21	amy-j	1
22	neil	1
23	felix	null

CREDITCARD

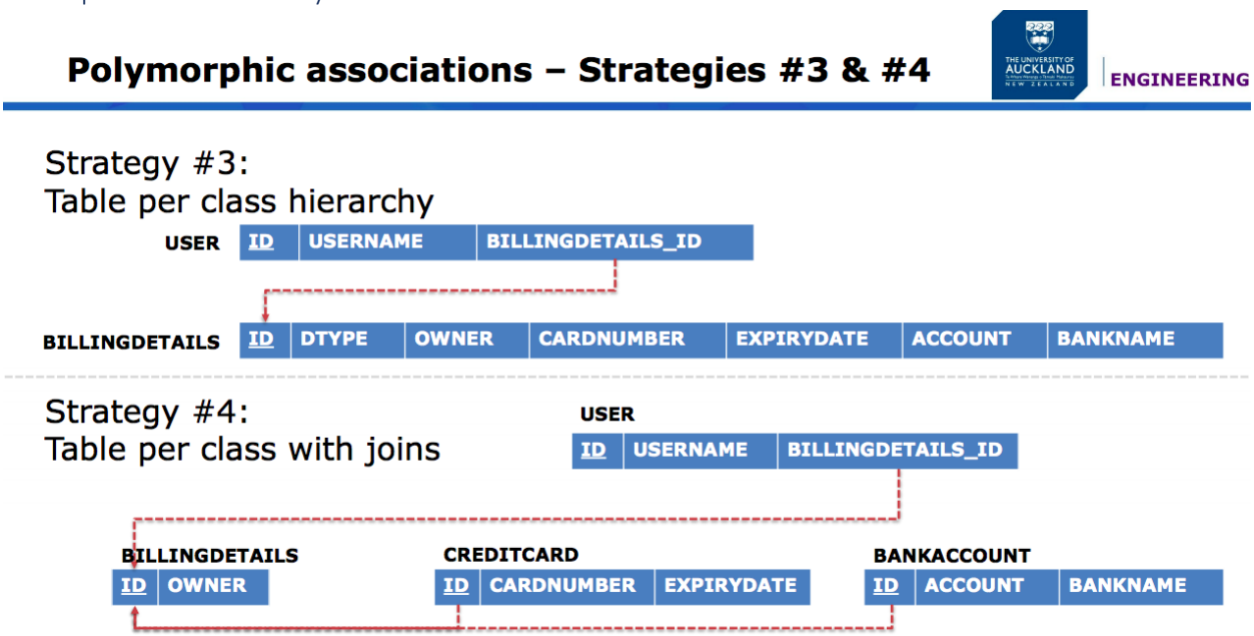
ID	OWNER	CARDNUMBER	EXPIRYDATE
1	Amy	4999...	Apr-2015
2	Kim	4999...	Mar-2017
3	Geoff	4556	Dec-2016

BANKACCOUNT

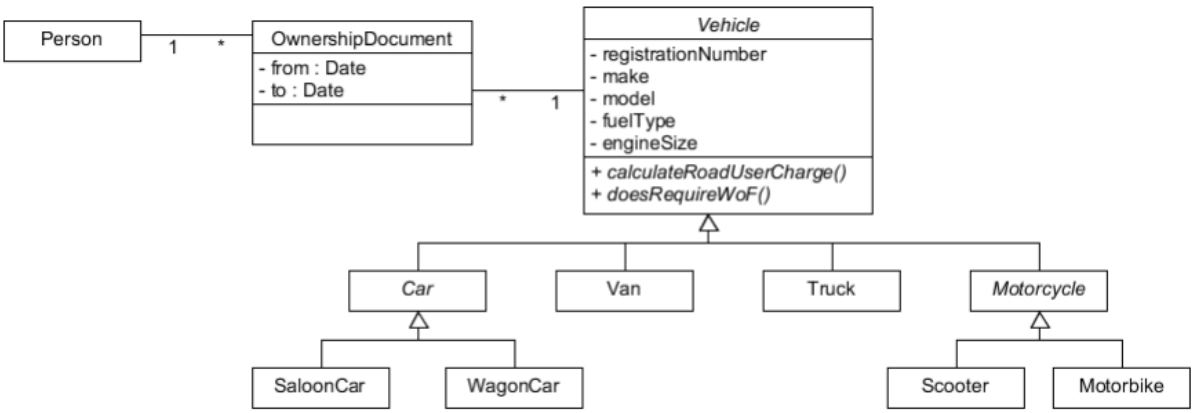
ID	OWNER	ACCOUNT	BANKNAME
4	Pete	50887471	ANZ
5	John	83846883	ASB

Primary keys in CREDITCARD and BANKACCOUNT need to be unique from each other.

Table per Class Hierarchy

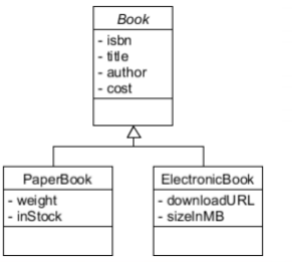


Example 1



- Strategy 1 can't support polymorphism.
- Strategy 2 might have a of duplicated columns due to deep hierarchy.
- Strategy 4 might require a lot of joins.

Example 2

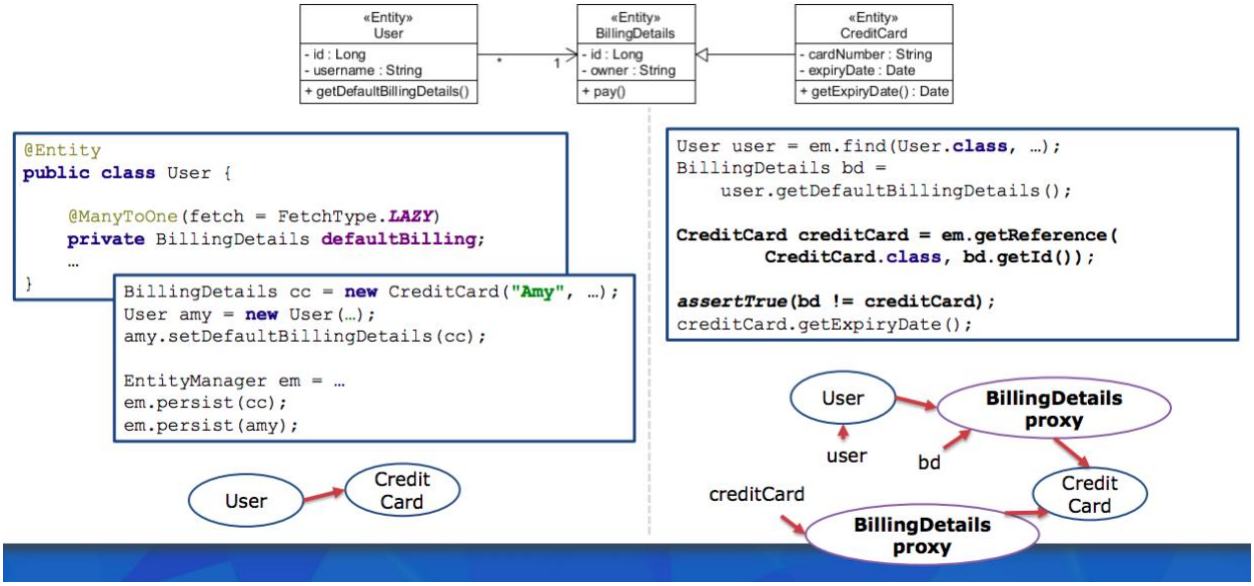


- We want to enforce that the values can't be null.
- Strategy 1 is do-able since there are no polymorphic associations.
- Strategy 2 is ok but will be duplicated columns.
- Strategy 3 would have a lot of null values and can't enforce null.
- Strategy 4 best option, not many joins required, data is normalized, we can enforce data integrity and non-null columns.

Strategy Summary

	Strategy 1	Strategy 2	Strategy 3	Strategy 4
Polymorphic queries	Requires 1 SQL select per concrete class table	Requires use of SQL union involving all concrete class tables	Performs best – all subclass instance data is held in one table	Requires use of SQL join operator on multiple tables
Polymorphic associations	Not supported	Supported by simulated foreign key	Supported simply with all data in one table	Supported by true foreign key relationships
Schema evolution	Poor – changing a superclass property impacts multiple tables	As for strategy 1	Good – any change to a class requires the single table to be changed	Good
Other			Lacks data integrity Relational schemas aren't normalised	Normalised schemas Potential performance issues with wide/deep inheritance hierarchies

Lazy Loading



We can set up lazy loading to load from the database only on demand. For example, if we hardly need to access a user's billing details, we don't need to load them immediately when we load the user.

Need to explicitly create a credit card proxy using the entity manager.

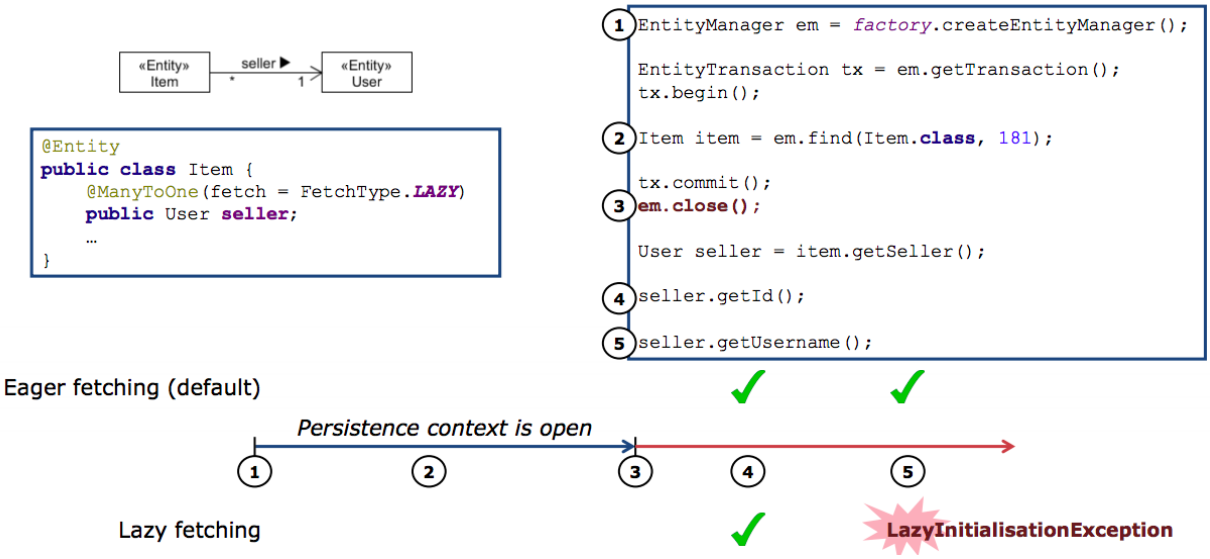
Fetch Plans & Strategies

Fetch Plans

JPA supports two fundamental techniques for loading data.

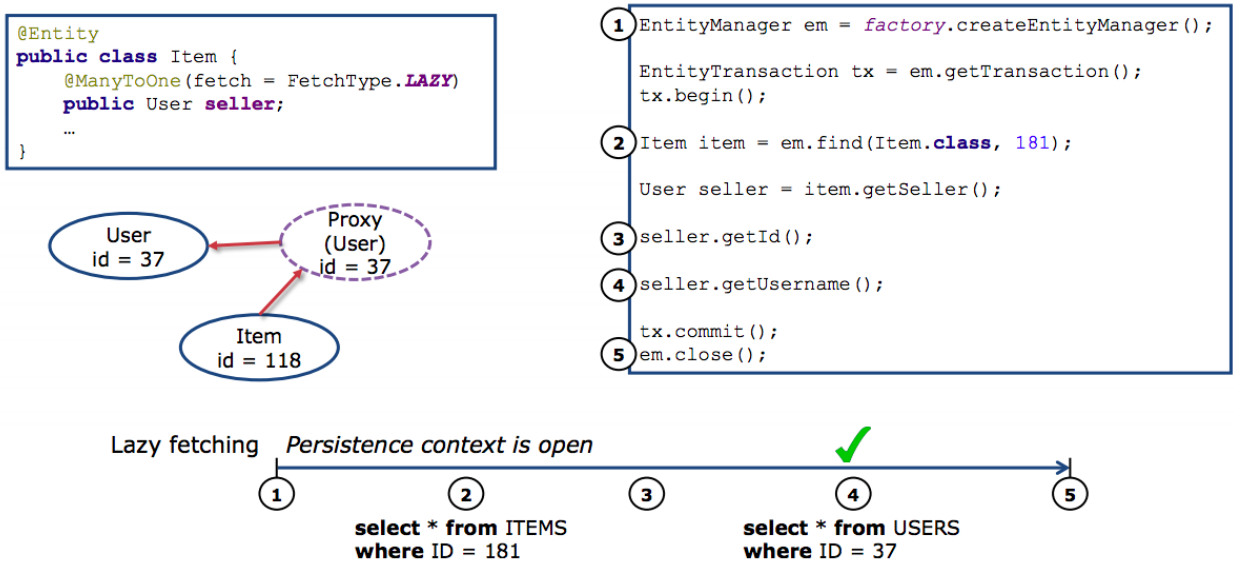
- Lazy loading
  - A domain-model object graph is loaded in small parts, as the graph is navigated.
- Eager fetching
  - A domain-model object graph is loaded by minimizing the number of database requests.

Loading Associations



Since the connection to the database is closed and the proxy does not contain the username, it will throw an exception

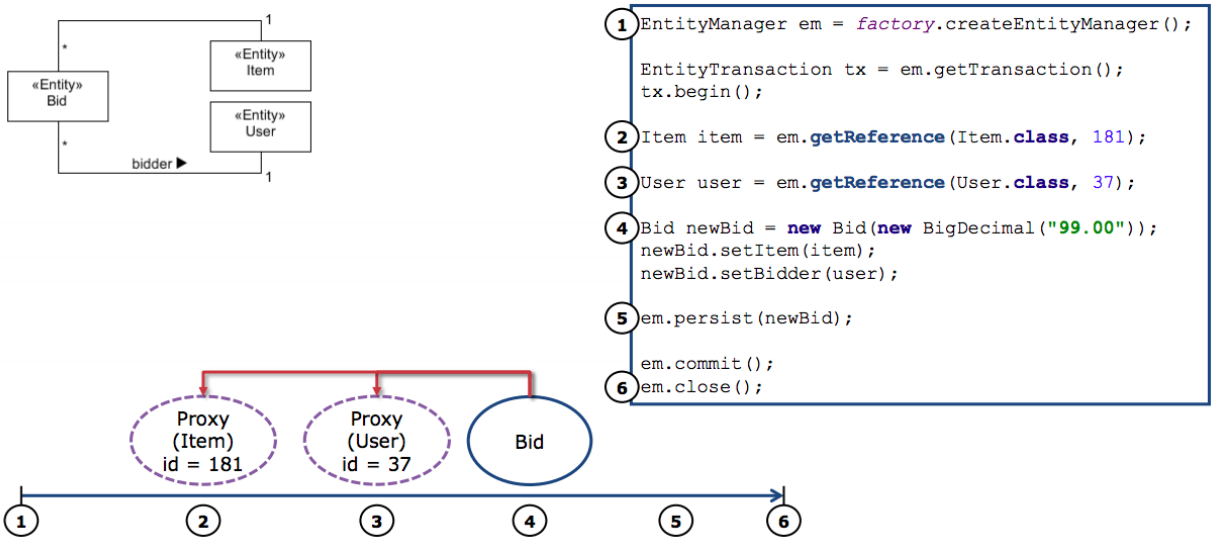
Entity Proxies



When created the proxy only knows the ID of the data it represents. With any method to access the data for this class the JPA then accesses the database. This only works persistent context remains open.

- GetClass() returns Proxy instead of the class. When overriding equals(), use instanceof instead of getClass(). Use accessor methods instead of direct field access.

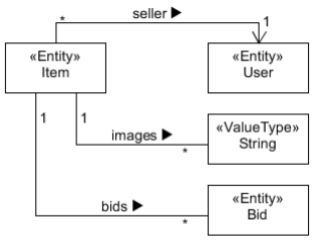
Entity proxies



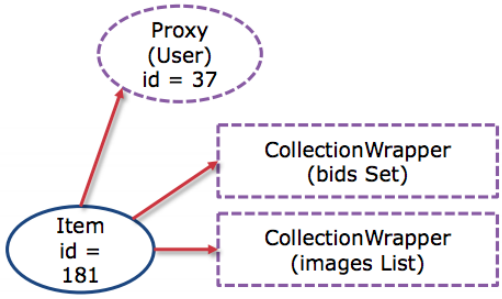
Create proxies for the items so there only needs to be one database access when the new bid is added.

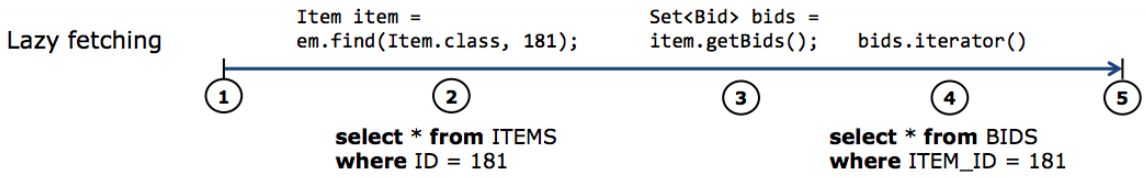
Lazy Persistent Connections

Persistent collections that are mapped with @ElementCollection, @OneToMany and @ManyToMany are, by default, lazily loaded




```
1 EntityManager em = factory.createEntityManager();

  EntityTransaction tx = em.getTransaction();
  tx.begin();

2 Item item = em.find(Item.class, 181);

3 Set<Bid> bids = item.getBids();

  PersistenceUtil persistenceUtil =
          Persistence.getPersistenceUtil();

  assertFalse(persistenceUtil.isLoaded(item, "bids" ));
  assertTrue(Set.class.isAssignableFrom(bids.getClass()));
  assertNotEquals(bids.getClass(), HashSet.class);
  assertEquals(bids.getClass(),
          org.hibernate.collection.internal.PersistentSet.class);

4 Bid firstBid = bids.iterator().next();

  tx.commit();
5 em.close();
```



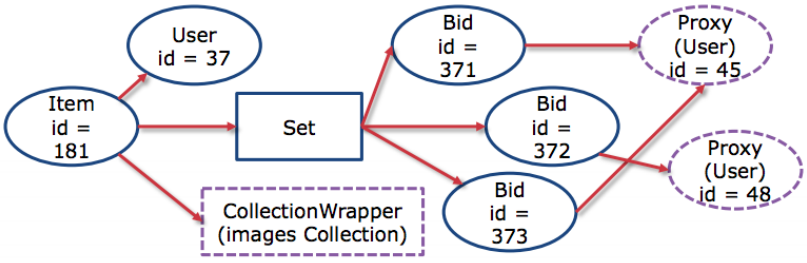


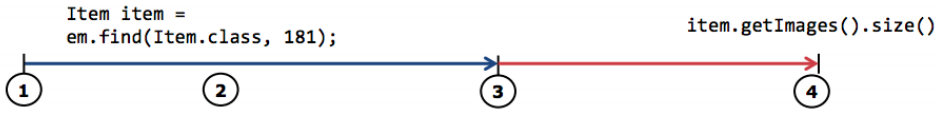
Eager Fetching

```java
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<>();

    @ElementCollection
    private Collection<String> images = new ArrayList<>();
}
```

```java
1 EntityManager em = factory.createEntityManager();

EntityTransaction tx = em.getTransaction();
tx.begin();

2 Item item = em.find(Item.class, 181);

tx.commit();
3 em.close();

item.getBids().size();
item.getBids().iterator().next().getAmount();
item.getSeller().getUsername();

4 item.getImages().size();
```

Eager fetching & lazy loading



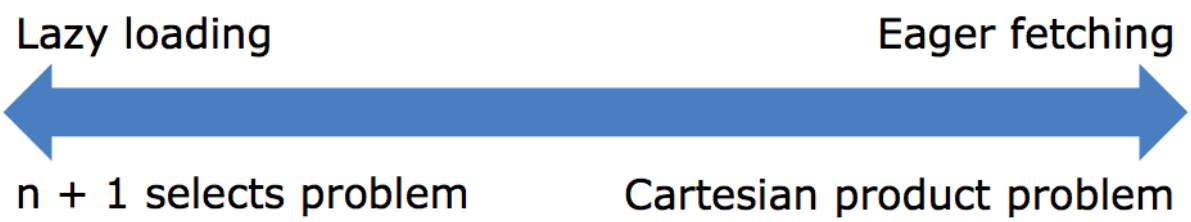



```
select i.*, u.*, b.*
from ITEMS i
   left outer join USERS u on u.ID = i.SELLER_ID
   left outer join BIDS b on b.ITEM_ID = i.ID
where i.ID = 181
```

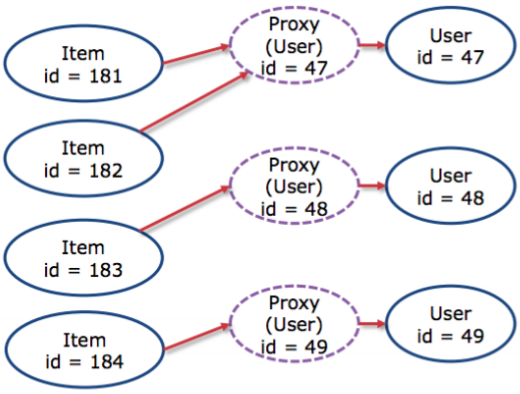
LazyInitialisationException

Fetch Strategies

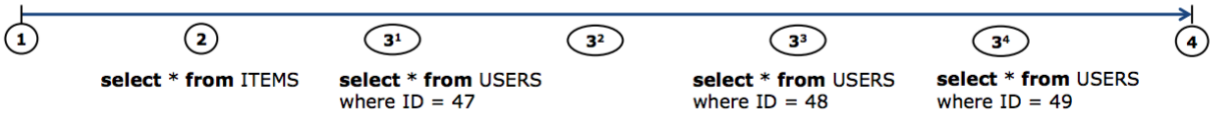
- A fetch plan should generally use lazy loading for associations and collections.
- The goal is to minimise the number of SQL requests and to simplify them.



The n+1 Selects Problem




```
1 EntityManager em = factory.createEntityManager();

  EntityTransaction tx = em.getTransaction();
  tx.begin();

2 List<Item> items = em
          .createQuery("select i from Item i")
          .getResultList();

  for (Item item : items) {
3*    assertNotNull(item.getSeller().getUsername());
  }

  tx.commit();
4 em.close();
```



- Lazily loading data in this way is very inefficient and will negatively impact performance.
- Where a use case dictates that, e.g. for all Items their bids must be accessed, eager fetching appears to be a better solution…but this can lead to the Cartesian product problem.

```
EntityManager em = factory.createEntityManager();

EntityTransaction tx = em.getTransaction();
tx.begin();

List<Item> items = em
        .createQuery("select i from Item i")
        .getResultList();

for (Item item : items) {
    item.getBids().size();
}
tx.commit();
em.close();
```

The Cartesian Product Problem

```
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    …
}
```

```
Item item = em.find( Item.class, 181 );
```

```
select i.*, u.*
    from ITEMS i
        left outer join USERS u on u.ID =
i.SELLER_ID
    where i.ID = 181
```

Result set

i.ID	i.NAME	i.SELLER_ID	...	u.ID	u.USERNAME	...
1	Item 1	2		2	John	

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item",
        fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<>();

    @ElementCollection(fetch = FetchType.EAGER)
    private Collection<String> images = new
ArrayList<>();

    …
}
```

```
Item item = em.find( Item.class, 181 );
```

```
select i.*, b.*, img.*
   from ITEM i
      left outer join BIDS b on b.ITEM_ID = i.ID
      left outer join IMAGES img on img.ITEM_ID = i.ID
   where i.ID = 181
```

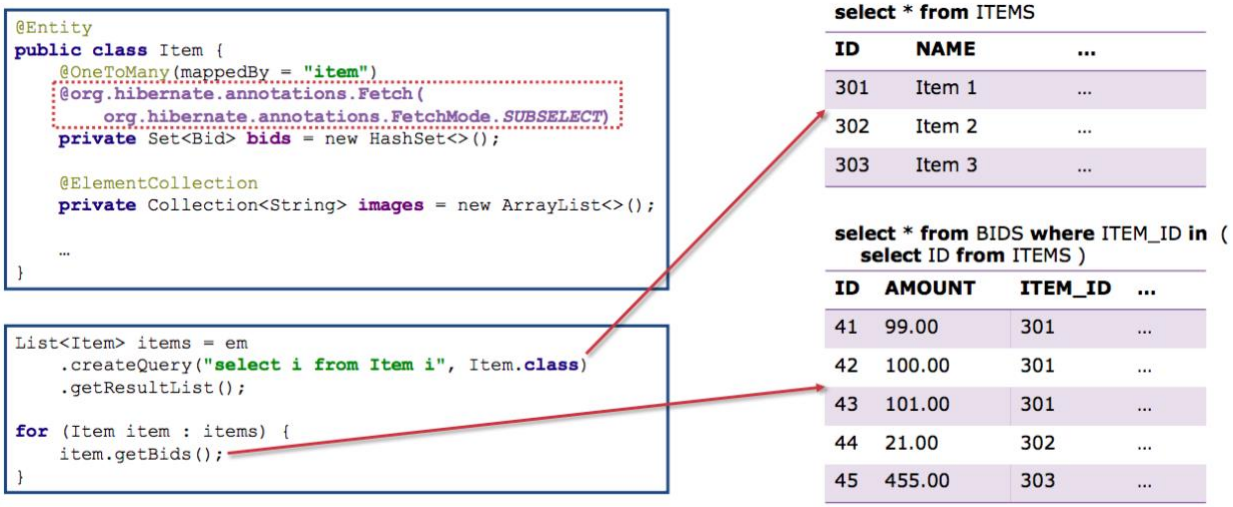
i.ID	i.NAME	...	b.ID	b.AMOUNT	img.FILENAME
181	Item 1	...	1	99.00	Foo.jpg
181	Item 1	...	1	99.00	Bar.jpg
181	Item 1	...	1	99.00	Baz.jpg
181	Item 1	...	2	100.00	Foo.jpg
181	Item 1	...	2	100.00	Bar.jpg
181	Item 1	...	2	100.00	Baz.jpg
181	Item 1	...	3	101.00	Foo.jpg
181	Item 1	...	3	101.00	Bar.jpg
181	Item 1	...	3	101.00	Baz.jpg

- A lot of duplicated data, only the ones circled in red are unique.
- Can consume considerable resources to process, including database time, network bandwidth, and application memory.

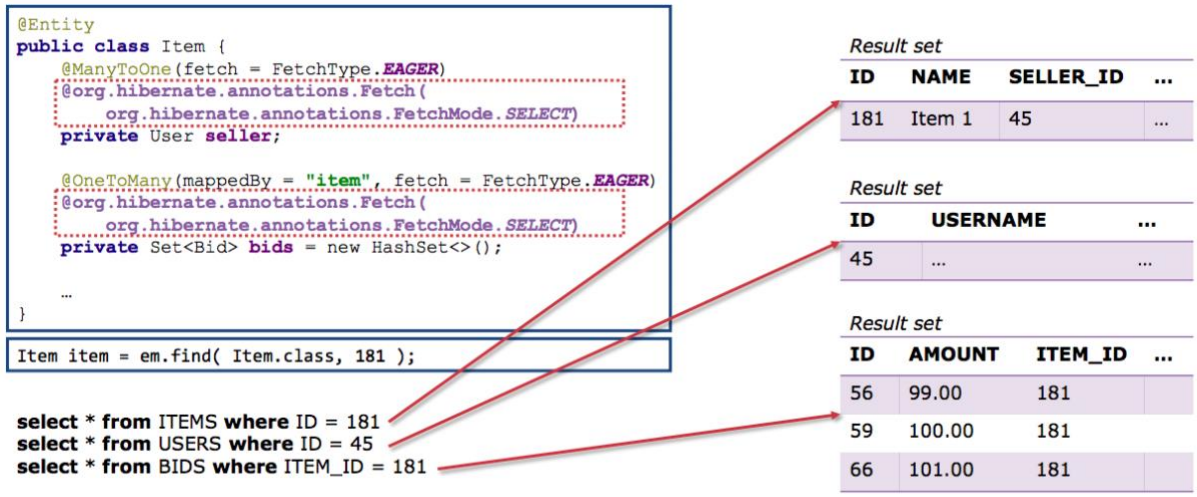
Optimizations

- The Hibernate JPA provider offers strategies for addressing the n+1 selects and Cartesian product problems.
- N+1 selects:
    - Collections can be pre-fetched using SQL sub-select statements. This allows many collections to be loaded with a single round trip to the database.
- Cartesian product:
    - Hibernate can eagerly load data using additional SQL select statements instead of join operations.

Collection Pre-Fetching using Sub-Selects



Using Additional Selects instead of Joins



We can run the queries separately and Hibernate will in memory combine the result sets as needed (not gigantic Cartesian produce result set).

Dynamic Eager Fetching

```
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.LAZY)
    private User seller;

    …
}
```

Adopt a global fetch plan for lazy loading of all entity associations and collections, and "override" it in particular work-units that really do need eager fetching.

```
List<Item> items = em
        .createQuery("select i from Item i join fetch i.seller")
        .getResultList();

…
em.close();
…

for (Item item : items) {
    item.getSeller().getUsername();
}
```

```
select i.*, u.*
  from ITEMS i
    inner join USERS u
      on u.ID = i.SELLER_ID
```

i.ID	i.NAME	i.SELLER_ID	...	u.ID	u.USERNAME	...
181	Item 1	47	...	47	Brad	...
182	Item 2	47	...	47	Brad	...
183	Item 3	48	...	48	Clint	...
184	Item 4	49	...	49	Bruce	...

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item", fetch = FetchType.LAZY)
    private Set<Bid> bids = new HashSet<>();

    …
}
```

```
List<Item> items = em
        .createQuery("select i from Item i left join fetch i.bids")
        .getResultList();

…
em.close();
…

for (Item item : items) {
    item.getBids().size();
}
```

```
select i.*, b.*
  from ITEMS i
    left outer join BIDS b
      on b.ITEM_ID = i.ID
```

i.ID	i.NAME	...	b.ID	b.AMOUNT	...
181	Item 1	...	371	99.00	...
181	Item 1	...	372	100.00	...
181	Item 1	...	373	101.00	...
182	Item 2	...	398	450.00	...
183	Item 3	...	null	null	...
184	Item 4	...	null	null	...

JPQL (Java Persistent Query Language)

Overview

- A JPQL query is developed in 3 steps:
  - **Creation:** A query is expressed in terms of selection [from], restriction [where] and projection [select] elements.
  - **Preparation:** Query parameters are bound to values.
  - **Execution:** The query is executed against the database and retrieves data.
- In JPQL, queries are expressed in terms of entity classes and property (field) names.
- JPQL queries are created by the `EntityManager`.

```
select i from Item I

select i from Item i where i.name like '%DVD%'

select o from Object o
```

```
EntityManager em = …;

Query query = em.createQuery(
   "select i from Item i where i.id = :id")
        .setParameter("id", 1);

Item result = (Item)query.getSingleResult();
```

Query Preparation and Execution

```
EntityManager em = …;

Query itemQuery = em
        .createQuery(
        "select i from Item i where i.name = :itemName")
        .setParameter("itemName", searchString);

List<Item> items =
        (List<Item>) itemQuery.getResultList();


Item someItem = …;
Query bidQuery = em
        .createQuery(
        "select b from Bid b where b.item = :item")
        .setParameter("item", someItem);

List<Bid> bids = (List<Bid>) bidQuery.getResultList();
```

```
EntityManager em = …;

TypedQuery<Item> itemQuery = em
        .createQuery(
            "select i from Item i where i.id = :id",
            Item.class)
        .setParameter("id", ITEM_ID);

Item item = itemQuery.getSingleResult();


Item someItem = …;
TypedQuery<Bid> bidQuery = em
        .createQuery(
            "select b from Bid b where b.item = :item",
            Bid.class)
        .setParameter("item", someItem);

List<Bid> bids = bidQuery.getResultList();
```

TypedQuery<> - no need for casting. Use if result class known.

getSingleResult() – use if a single result is expected.
Can throw NonUniqueResultException or NoResultException.

Query Restriction

```
select i from Item i where i.name = 'Foo'
```

```
select b from Bid b where b.amount between 99 and 100
```

```
select u from User u where u.username in ( 'Homer', 'Marge' )
```

```
select u from User u where u.username like 'H%'
```

```
select u from User u where u.username not like 'H%'
```

```
select b from Bid b where (b.amount / 2) – 0.5 > 49
```

```
select i from Item i where i.name like 'Fo%' and i.buyNowPrice is not null
```

```
select c from Category c where c.items is not empty
```

```
select c from Category c where size(c.items) > 1
```

```
select c from Category c where :item member of c.items
```

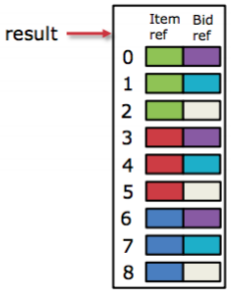
JPQL operator	Description
.	Navigation path operator
+, -, *, /	Mathematical operators
=, <>, <, >, >=	Binary comparison operators with SQL semantics
[not] between, [not] like, in, is [not] null	Binary operators with SQL semantics
is [not] empty, [not] member of	Binary operators for persistent collections
not, and, or	Logical expressions

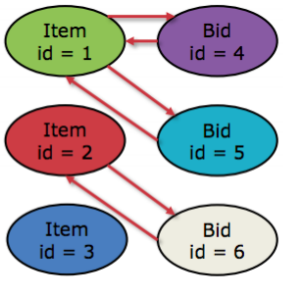
JPA function	Description
upper(s), lower(s), concat(s), length(s)	String manipulators
current_date, current_time, current_timestamp	Return the date and/or time from the DBMS
abs(n), sqrt(n), mod(dividend, divisor)	Mathematical functions
size(c)	Returns the size of a collection

Query Projection


```
TypedQuery<Object[]> query = em.createQuery(
    "select i, b from Item i, Bid b",
    Object[].class);
List<Object[]> result = query.getResultList();
```

ITEM		BID		
ID	**NAME**	**ID**	**AMOUNT**	**ITEM_ID**
1	American Sniper DVD	4	18.50	1
2	iPhone	5	22.00	1
3	Bluetooth speaker	6	200.00	2





Revision – SQL Joins

Revision - SQL joins

ITEM		BID		
ID	**NAME**	**ID**	**AMOUNT**	**ITEM_ID**
1	American Sniper DVD	4	18.50	1
2	iPhone	5	22.00	1
3	Bluetooth speaker	6	200.00	2

```
select i.*, b.*
   from ITEM i
      inner join BID b on i.ID = b.ITEM_ID
```

i.ID	i.NAME	b.ID	b.AMOUNT	b.ITEM_ID
1	American Sniper DVD	4	18.50	1
1	American Sniper DVD	5	22.00	1
2	iPhone	6	200.00	2

```
select i.*, b.*
   from ITEM i
      left outer join BID b on i.ID = b.ITEM_ID
```

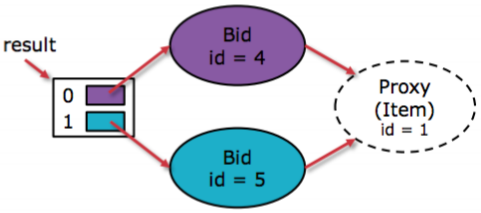
i.ID	i.NAME	b.ID	b.AMOUNT	b.ITEM_ID
1	American Sniper DVD	4	18.50	1
1	American Sniper DVD	5	22.00	1
2	iPhone	6	200.00	2
3	Bluetooth speaker	null	null	null

```
select b.*, i.*
   from BID b
      right outer join ITEM i on b.ITEM_ID = i.ID
```

b.ID	b.AMOUNT	b.ITEM_ID	i.ID	i.NAME
4	18.50	1	1	American Sniper DVD
5	22.00	1	1	American Sniper DVD
6	200.00	2	2	iPhone
null	null	null	3	Bluetooth speaker

Implicit Association Joins

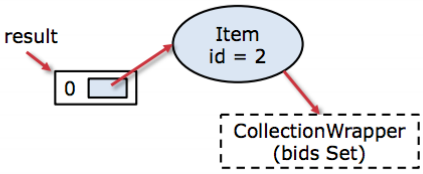
ITEM		BID		
ID	**NAME**	**ID**	**AMOUNT**	**ITEM_ID**
1	American Sniper DVD	4	18.50	1
2	iPhone	5	22.00	1
3	Bluetooth speaker	6	200.00	2




```
TypedQuery<Bid> queryForBid = em.createQuery(
        "select b from Bid b where b.item.name like '%DVD%'",
        Bid.class);
List<Bid> result = queryForBid.getResultList();
```

Explicit Joins for Restriction

ITEM		BID		
ID	**NAME**	**ID**	**AMOUNT**	**ITEM_ID**
1	American Sniper DVD	4	18.50	1
2	iPhone	5	22.00	1
3	Bluetooth speaker	6	200.00	2



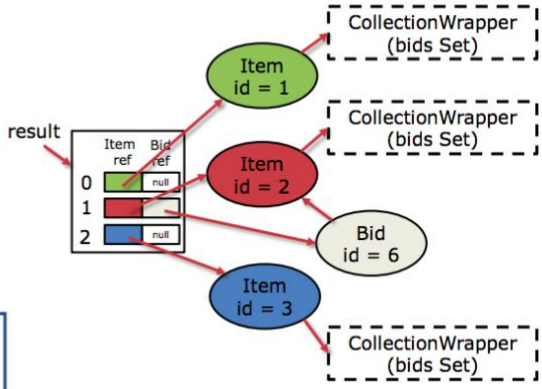

```
TypedQuery<Item> queryForItem = em.createQuery(
        "select i from Item i " +
                "inner join i.bids b " +
                "where b.amount > 100", Item.class);
List<Item> result = queryForItem.getResultList();
```

Step #1: Perform an inner join to match Items and their Bids. **b** is an alias to the joined association.

Step #2: Restrict the results with reference to **b** elements.

Explicit Joins for Inclusion

ITEM		BID		
ID	**NAME**	**ID**	**AMOUNT**	**ITEM_ID**
1	American Sniper DVD	4	18.50	1
2	iPhone	5	22.00	1
3	Bluetooth speaker	6	200.00	2




```
TypedQuery<Object[]> queryForItemBidPairs = em.createQuery(
            "select i, b from Item i " +
            "left outer join i.bids b on b.amount > 100",
                Object[].class);
List<Object[]> result = queryForItemBidPairs.getResultList();
```

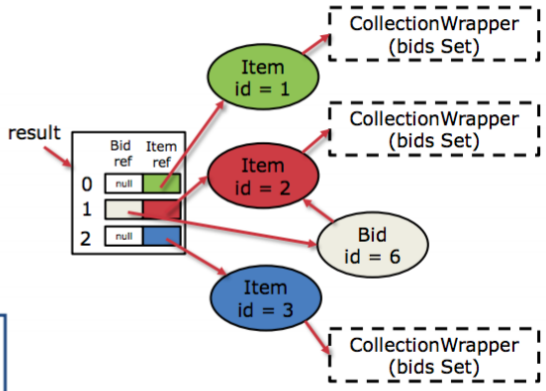
ITEM

ID	NAME
1	American Sniper DVD
2	iPhone
3	Bluetooth speaker

BID

ID	AMOUNT	ITEM_ID
4	18.50	1
5	22.00	1
6	200.00	2


```
TypedQuery<Object[]> queryForBidItemPairs = em.createQuery(
            "select b, i from Bid b " +
            "right outer join b.item i on b.amount > 100",
                Object[].class);
List<Object[]> result = queryForBidItemPairs.getResultList();
```



Explicit Joins with Eager Fetching

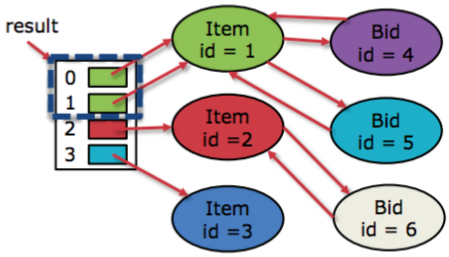
ITEM

ID	NAME
1	American Sniper DVD
2	iPhone
3	Bluetooth speaker

BID

ID	AMOUNT	ITEM_ID
4	18.50	1
5	22.00	1
6	200.00	2


```
TypedQuery<Item> query = em.createQuery(
            "select i from Item i " +
            "left join fetch i.bids", Item.class);
List<Item> result = query.getResultList();
```

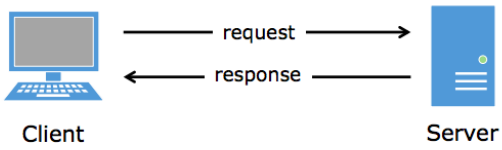


There are duplicate rows for item 1 because it is in two bids. Shoving it into a set will get rid of duplicates.

Asynchronous Web Services

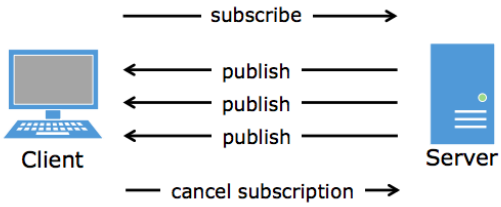
Communication Patterns

Synchronous communication



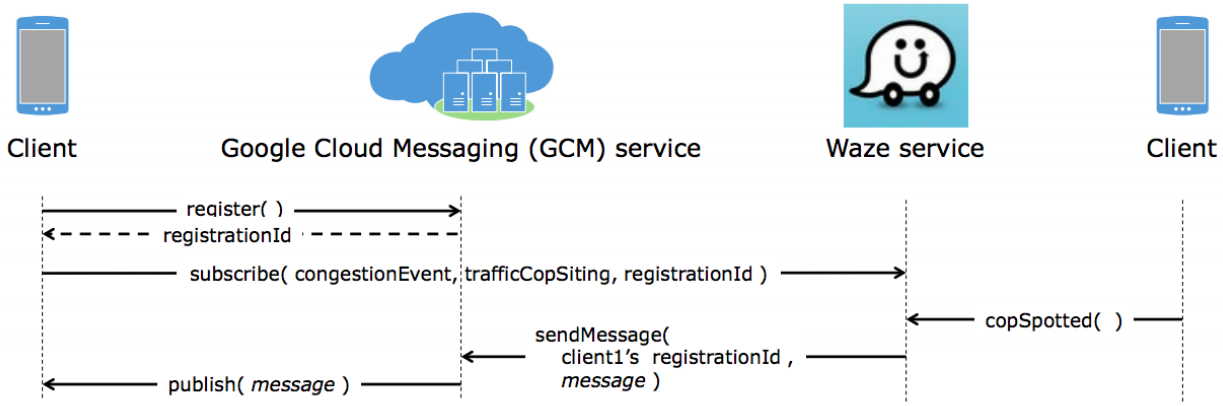
- Clients communicate synchronously with the server over a request reply protocol
- Clients block while waiting for a reply

Publish / Subscribe communication



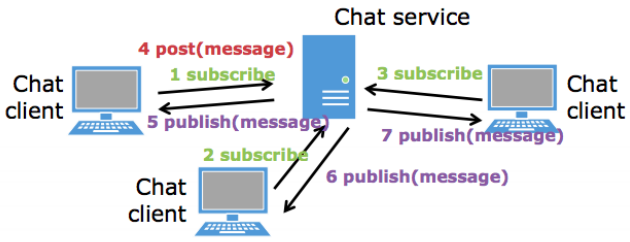
- Clients communicate with servers asynchronously using message-based protocols
- Having sent a message, senders do not block
- Once subscribed, a client receives updates (publish messages) from the server

Publish Subscribe Communication

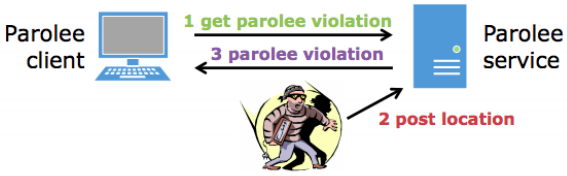


Use Cases for Asynchronous Communication

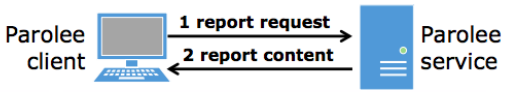
- Publish/Subscribe services



- Server-side push services

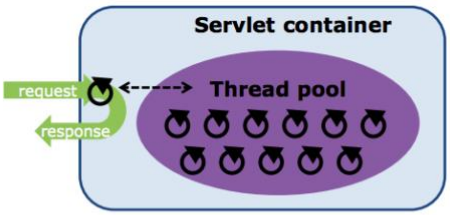


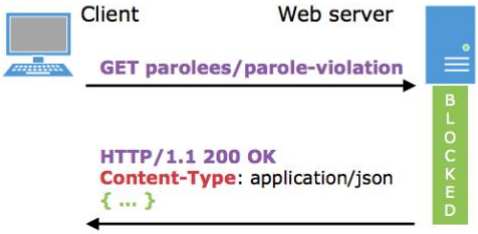
- Services with long running tasks or priority scheduling



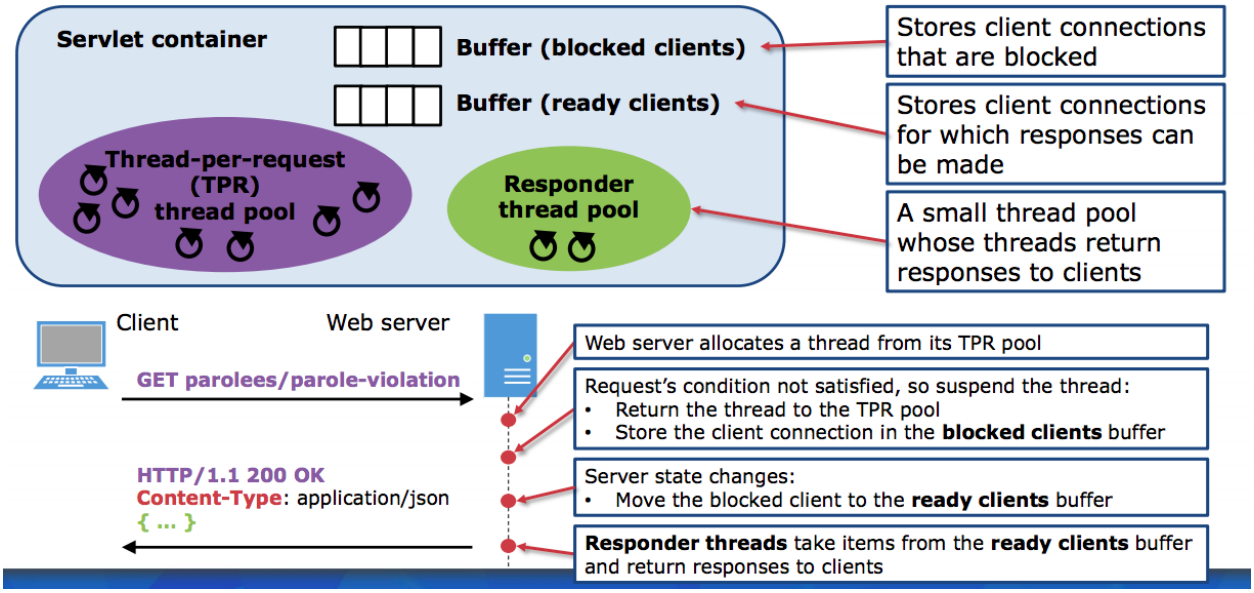
Server-Side Push

- HTTP requests tend to have short processing durations.
- HTTP servers generally implement a thread-per-request model.
    - An incoming request is services by a thread taken from the server's threadpool; afterwards, the thread is returned to the pool.
- Server-side push can be implemented over HTTP by the client making a "long poll" request.
    - The client blocks until something happens on the server (e.g. a state change) that allows the server to return a response.
    - The server maintains the connection with the client for the duration of the request.

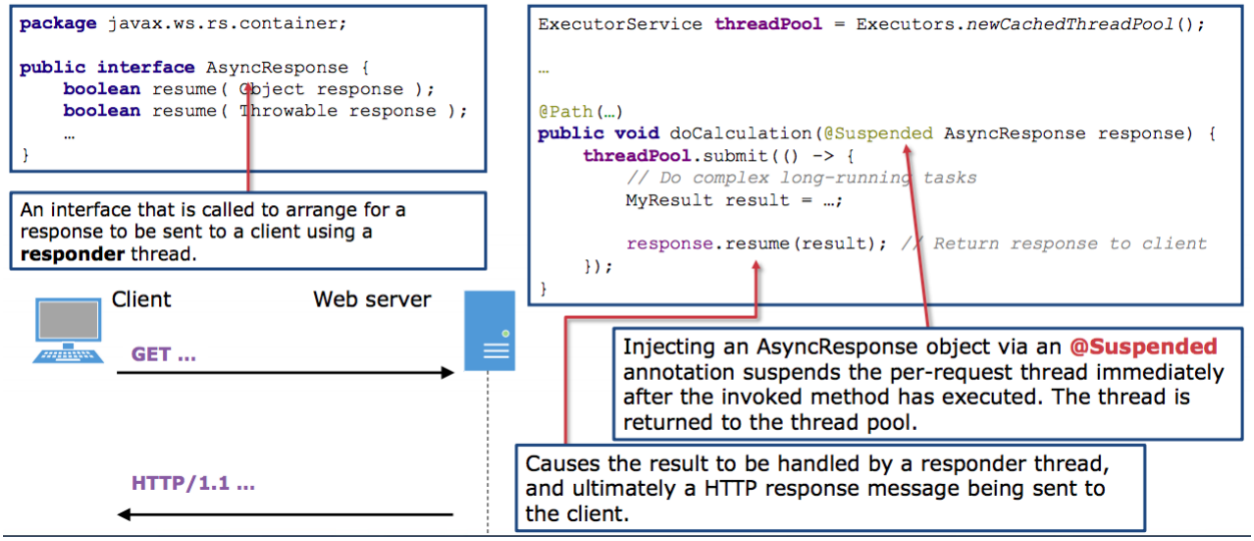




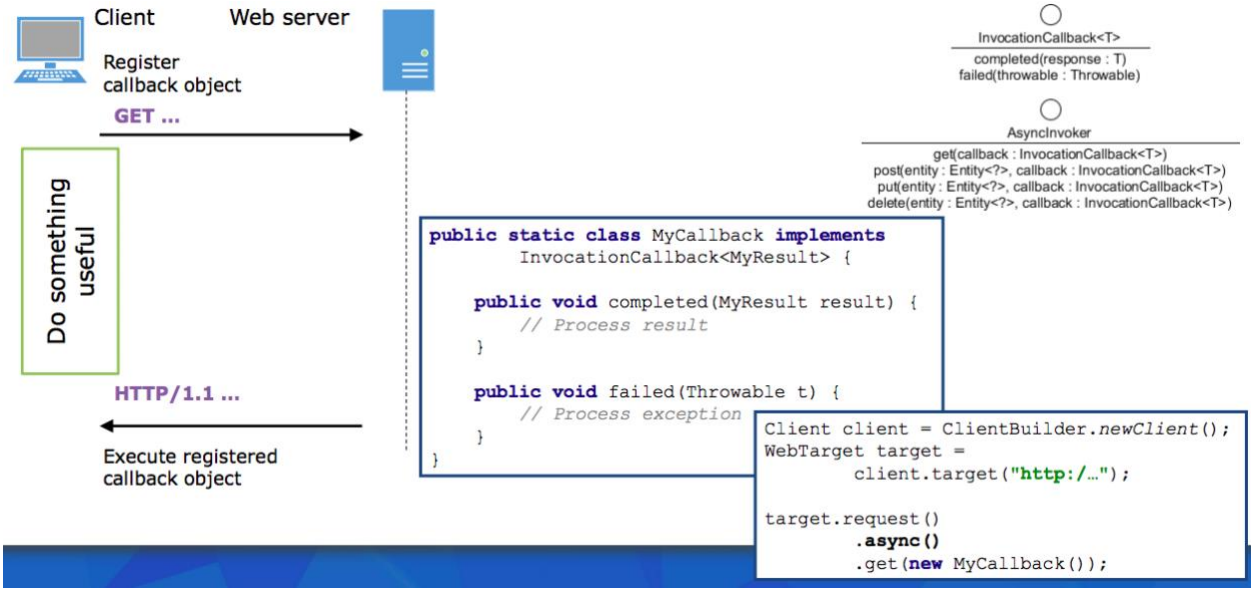
Servlet 3 Specification



JAX-RS: AsyncResponse



JAX-RS: AsyncInvoker & InvocationCallback



Alternative: Use of Future<>

- Rather than use InvocationCallback, we can instead obtain a Future object from the asynchronous request.
- At some point in the future, we call the future's get() method to receive the response (or exception).

```
// Create an async request to the subscription service and set it going
Future<Response> future = CLIENT
        .target(WEB_SERVICE_URI + "/subscribeParoleViolations")
        .request().async().get();

// Do useful stuff
…

// Get response. Will block until the response is received.
Response response = future.get(5, TimeUnit.SECONDS);
```

A Publish/Subscribe Chat Service

```
@GET
@Path("/sub")
@Produces(MediaType.APPLICATION_JSON)
public void subscribeToMessage(@Suspended AsyncResponse sub) {
    subs.add(sub);
}

@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response postMessage(Message message) {

    synchronized (subs) {
        for (AsyncResponse sub : subs) {
            sub.resume(message);
        }
        subs.clear();
    }

    return Response.ok().build();

}
```

```
Message message = …

Response response = chatClient
    .target(WEB_SERVICE_URI)
    .request().post(Entity.json(message));
```

```
subClient.target(WEB_SERVICE_URI + "/sub")
        .request().accept(MediaType.APPLICATION_JSON)
        .async().get(new InvocationCallback<Message>() {

    @Override
    public void completed(Message message) {
        // Received a message. Display it and re-sub.
        System.out.println(message);
        subscribe(subClient);
    }

    @Override
    public void failed(Throwa
        // Handle failure
    }
});
```

Remember to re-subscribe if required, to continue receiving new messages.

Server-Side Push

```
@GET
@Path("/subscribeParoleViolations")
@Produces(MediaType.APPLICATION_JSON)
public void subscribeToParoleViolations(@Suspended AsyncResponse sub) {
    paroleViolationSubscriptions.add(sub);
}

@POST
@Path("{id}/movements")
@Consumes(MediaType.APPLICATION_JSON)
public void createMovementForParolee(@PathParam("id") long id, Movement movement) {
    Parolee parolee = …;
    parolee.addMovement(movement);

    processParoleViolation(parolee, movement);
}

private void processParoleViolation(Parolee parolee, Movement movement) {
    if (isParoleViolation(parolee, movement)) {
        for (AsyncResponse sub : paroleViolationSubscriptions) {
            sub.resume(…);
        }
        paroleViolationSubscriptions.clear();
    }
}
```

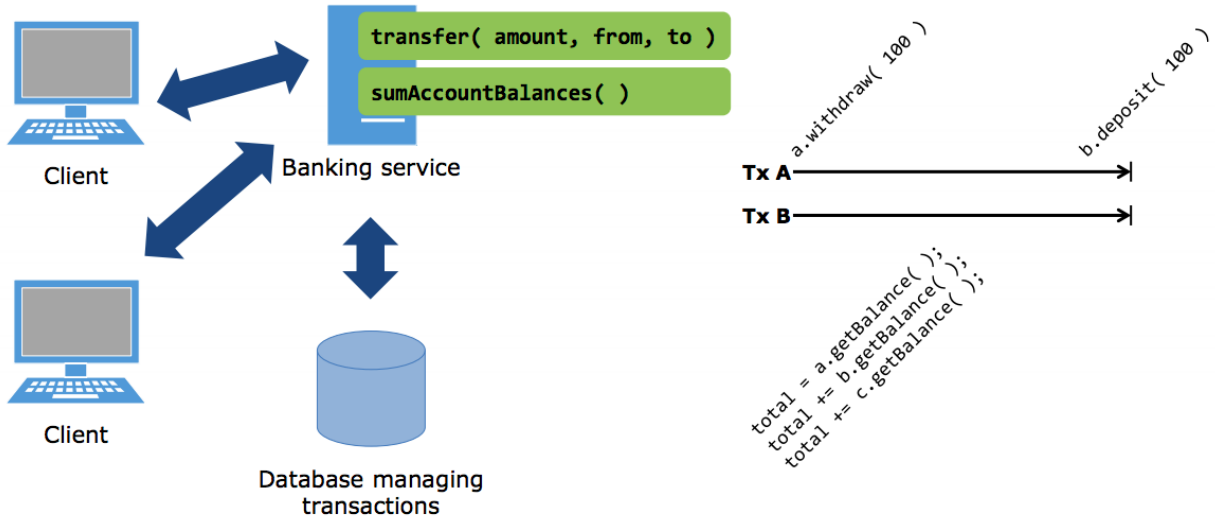
Priority Scheduling

- A service may have operations that are computationally intensive.
    - Executing too many such requests concurrently may starve clients making simple fast requests.
- Only bottleneck long running tasks.

```
ExecutorService threadPool =
    Executors.newSingleThreadExecutor();

@GET
@Path("/{id}/reoffendingRisk")
@Produces(MediaType.APPLICATION_JSON)
public void generateReport(@PathParam("id") long id,
        @Suspended AsyncResponse response) {

    threadPool.submit(() -> {
        // Do complex long-running tasks, then...
        Report resport = ...;

        response.resume(report);
    });
}
```

Transactions

Transaction Essentials



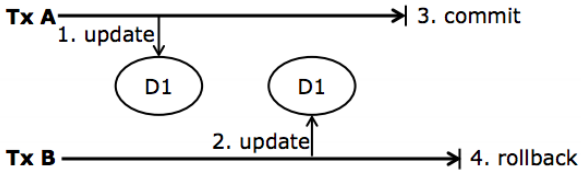
ACID Attributes

- **A**tomicity
  - All operations in a transaction execute as an atomic unit.
- **C**onsistency
  - Concurrently executing transactions do not compromise the data they operate on.
- **I**solation
  - The effects of concurrently executing incomplete transactions shouldn't be visible.
- **D**urability
  - Changes made by a transaction should be durable, even if the system fails after the transaction has been completed successfully.

Programming Transactions with JPA

```
EntityManagerFactory factory = …
EntityManager em = factory.createEntityManager( );

try {
    EntityTransaction tx = em.getTransaction( );

    tx.begin( );

    Execute queries / make EntityManager invocations

    tx.setRollbackOnly( );

    tx.commit();
} catch(Exception e) {
    Exception handling code
    logger.error( … )
} finally {
    if( em != null && em.isOpen( ) ) {
        em.close( );
    }
}
```

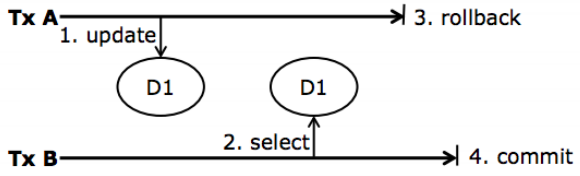
Database-level Concurrency Control



Problem #1 **Lost Update**

Transaction A updates data D1 ; transaction B does likewise. Transaction A commits and then transaction B aborts.

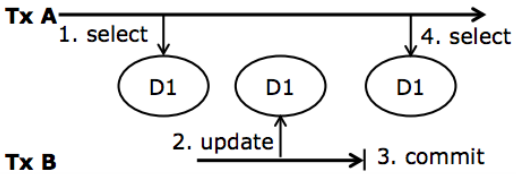
B's abortion returns D1 to its state when B started, undoing A's update.



Problem #2 **Dirty Read**

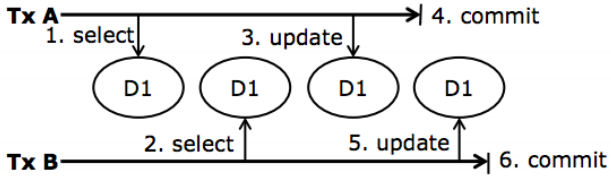
Transaction A updates data D1 ; transaction B reads the updated (and uncommitted) value for D1. Transaction A aborts while B commits.

B may have acted incorrectly based on the state of D1 that it read.



Problem #3 **Unrepeatable Read**

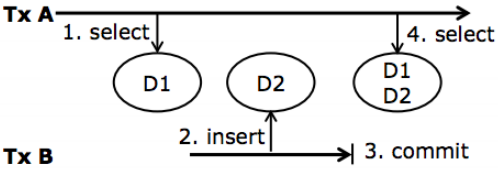
Transaction A reads data item D1 and sees a different value each time. Between the reads, transaction B has committed a write to D1.



Problem #4 **Last Commit Wins**

Transactions A and B each read data item D1 and then go on to write to D1.

The last transaction (B) to commit its write wins, overwriting the earlier transaction's write.

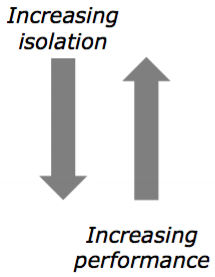


Problem #5 **Phantom Read**

A Phantom Read occurs where a transaction (A) executes a query twice, and the second result includes data that wasn't visible in the first result.

A concurrently executing transaction (B) inserts data between the other transaction's queries.

Isolation Levels



Problems that can occur

Isolation level	Lost updates	Dirty reads	Unrepeatable reads	Phantom reads
Read Uncommitted	No	Yes	Yes	Yes
Read Committed	No	No	Yes	Yes
Repeatable Read	No	No	No	Yes
Serializable	No	No	No	No

Read uncommitted: very rarely appropriate. Dangerous to use. We could have one transaction's uncommitted changes in another transaction.

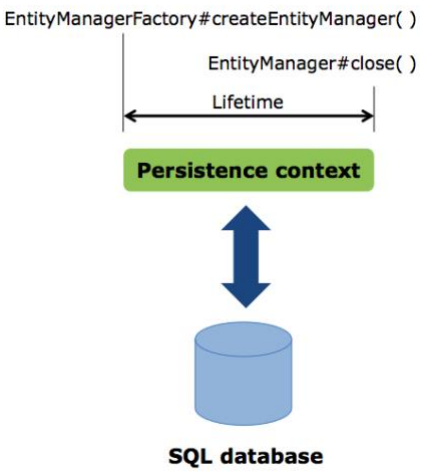
Serializable: rare used because phantom reads are generally not problematic, and this level seriously impacts performance (essentially sequential).

Repeatable read: alright, but not always needed.

Read committed: Default for most databases.

The Persistent Context

- JPA's persistence context manages persistent objects; the context:
  - Serves as a cache that enforces repeatable read semantics.
  - Performs dirty checking of objects and syncs them with the database.
  - Provides a guaranteed scope of object identity.
    - If entityA == entityB then entityA.getId().equals(entity.getId())



Optimistic Concurrency Control (OCC)

- Optimistic concurrency control involves working on a local copy of data and then updating the database.
- OCC uses version numbers for data.
  - Whenever an entity is updated, its version number is incremented.

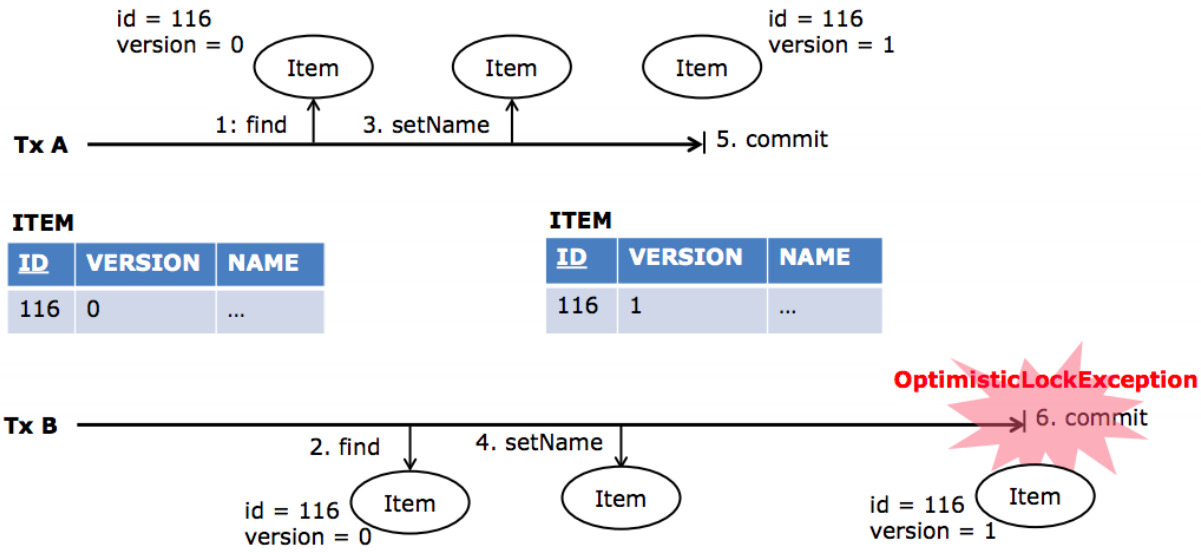
```
@Entity
public class Item {
    @Version
    private long version;
    …
}
```

```
EntityManagerFactory factory = …
EntityManager em = factory.createEntityManager( );

try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );
    Item item = em.find( Item.class, 116 );
    assertEquals( 0, item.getVersion( ) );
    item.setName( "New name" );
    tx.commit( );
} catch(Exception e) {
    logger.error( … );
} finally {
    if( em != null && em.isOpen( ) ) {
        em.close( );
    }
}
```

select * from ITEM **where** ID = 116

update ITEM
set NAME = "New name", VERSION = 1
where ID = 116 **and** VERSION = 0

Last Commit Wins -> First Commit Wins



A Repeatable Read Problem

Assume a many-to-one relationship between Item and Category.

CATEGORY A	CATEGORY B	CATEGORY C	
Item #1, price = $10 Item #2, price = $20	Item #3, price = $15 Item #4, price = $10	Item #5, price = $25	
Sum total = $30	Sum total = $25	Sum total = $25	**Sum total = $80**

CATEGORY A	CATEGORY B	CATEGORY C	
Item #1, price = $10	Item #3, price = $15 Item #4, price = $10	Item #5, price = $25 Item #2, price = $20	
Sum total = $30	Sum total = $25	Sum total = $45	**Sum total = $100**

Calculating the sum of all items across several categories requires that the "get items in each category" is a repeatable read.

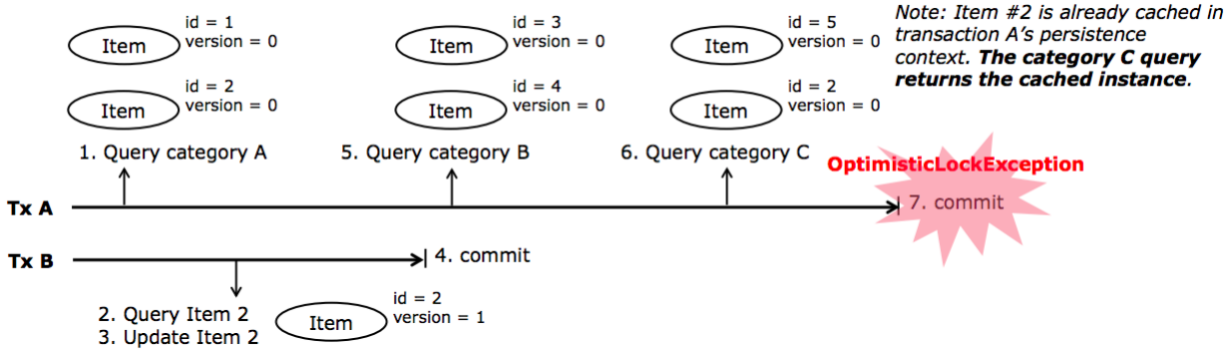
Forced Version Checking

```
try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );

    BigDecimal sumTotal = new BigDecimal( 0 );

    for( Long categoryId : categories ) {
        List< Item > items =
            em.createQuery( "select i from Item i where i.category.id = :catId" )
                    .setLockMode( LockModeType.OPTIMISTIC )
                    .setParameter( "catId", categoryId )
                    .getResultList( );

        for( Item item : items ) {
            sumTotal = sumTotal.add( item.getBuyNowPrice( ) );
        }
    tx.commit( );
} …
```

For each Category, query all Item instances using an OPTIMISTIC "lock" mode.

For each Item object loaded, the JPA implementation will execute a SQL **select** statement to compare the Item's version value with that of its corresponding row in the database.

If the version values differ or if the row no longer exists, commit( ) throws an OptimisticLockException.



Note: Item #2 is already cached in transaction A's persistence context. ***The category C query returns the cached instance.***

ITEMS table prior to transaction execution

ID	VERSION	CATEGORY
1	0	A
2	0	A
3	0	B
4	0	B
5	0	C

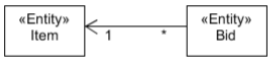
ITEMS table after transaction B has committed.

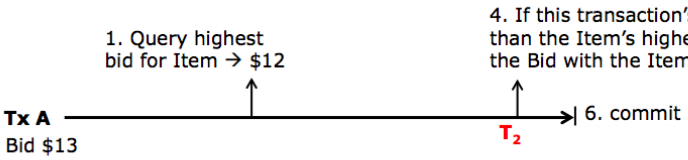
ID	VERSION	CATEGORY
1	0	A
2	**1**	**C**
3	0	B
4	0	B
5	0	C

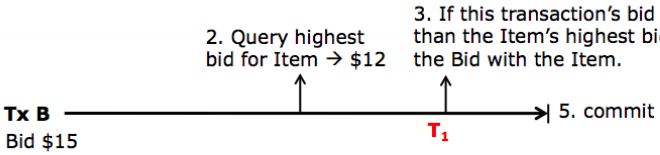
An "Invisible Conflict" Problem

Assumptions:

- Two transactions are bidding on a common Item.
- The Item currently has 1 bid of $12.
- Item and Bid are entities related by a many-to-one unidirectional relationship.
- A Bid has a timestamp; the most recent bid should be the winning bid.







BIDS

ID	ITEM_ID	TIMESTAMP	AMOUNT
1	116	23/08/15 10:26pm	$12.00
2	116	24/08/15 9:30am	$15.00
3	116	24/08/15 9:31am	$13.00

Forced Version Increment

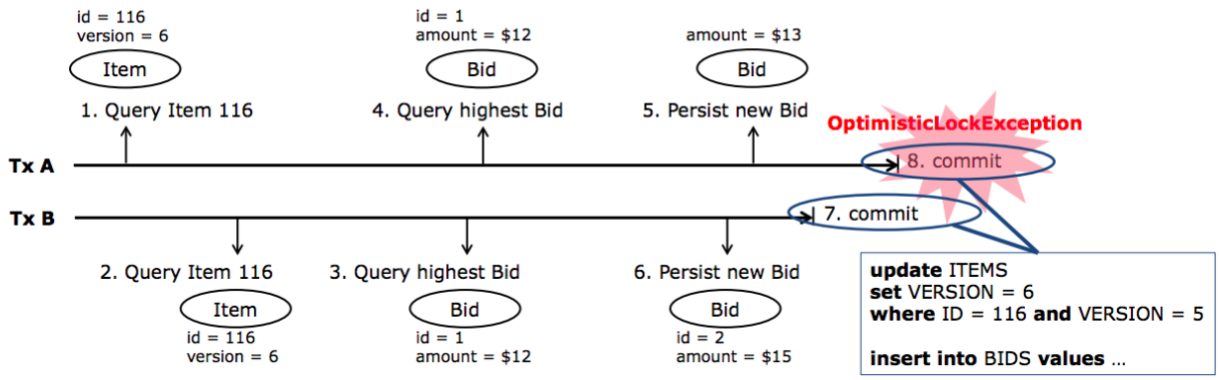
```
try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );

    Item item = em.find( Item.class,
            116,
            LockModeType.OPTIMISTIC_FORCE_INCREMENT );

    Bid highestBid = queryHighestBid( em, item );

    try {
        Bid newBid = new Bid( new BigDecimal( 15.00 ), item, highestBid );
        em.persist( newBid );
    } catch( InvalidBidException e ) {
        // Bid too low.
    }

    tx.commit( );
} ...
```

On loading the Item, the JPA implementation will increment the object's version number – even though the Item isn't going to be modified.

This persists a new Bid, as long as the bid is higher than the Item's present highest bid.

At commit time, the JPA implementation executes an SQL **insert** for the new Bid ***and*** an **update** for the Item, to update its version number.

The update will fail if the Item's version has since been incremented, e.g. by other transactions running this place-bid work unit.



Database tables prior to transaction execution

ITEMS

ID	VERSION
116	5

BIDS

ID	ITEM_ID	TIMESTAMP	AMOUNT
1	116	23/08/15 10:26pm	$12.00

Database tables after transaction B has committed

ITEMS

ID	VERSION
116	**6**

BIDS

ID	ITEM_ID	TIMESTAMP	AMOUNT
1	116	23/08/15 10:26pm	$12.00
2	116	24/08/15 9:30am	$15.00

Pessimistic Locking

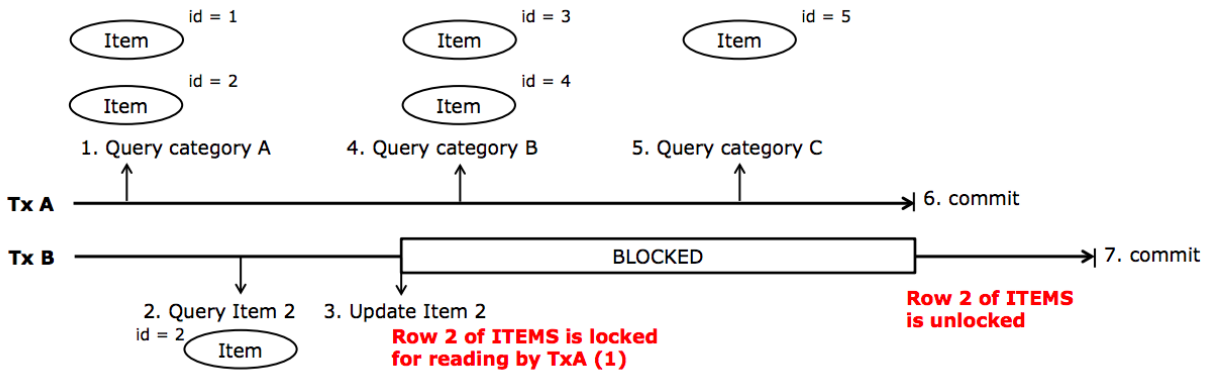
```
try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );

    BigDecimal sumTotal = new BigDecimal( 0 );

    for( long categoryId : categories ) {
        List< Item > items =
            em.createQuery( "select i from Item i where i.category.id = :catId" )
                    .setLockMode( LockModeType.PESSIMISTIC_READ )
                    .setHint( "javax.persistence.lock.timeout", 5000 )
                    .setParameter( "catId", categoryId )
                    .getResultList( );

        for( Item item : items ) {
            sumTotal = sumTotal.add( item.getBuyNowPrice( ) );
        }
    tx.commit( );
} …
```

For each Category, query all Item instances, using a database-level read-lock to lock ITEM table rows.

This guarantees repeatable reads.

If the required locks can't be obtained within 5 seconds, the query throws either a LockTimeoutException or a PessimisticLockException.

The acquired locks are released.



ITEMS table prior to transaction execution

ID	CATEGORY
1	A
2	A
3	B
4	B
5	C

ITEMS table after transaction A has committed.

ID	CATEGORY
1	A
2	A
3	B
4	B
5	C

ITEMS table after transaction B has committed.

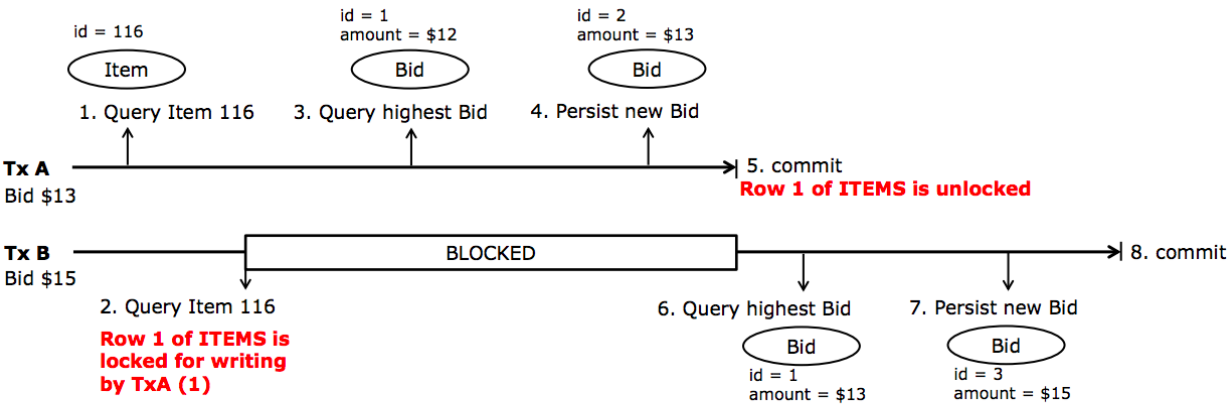
ID	CATEGORY
1	A
2	**C**
3	B
4	B
5	C

```
try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );

    Item item = em.find( Item.class,
            116,
            LockModeType.PESSIMISTIC_WRITE );

    Bid highestBid = queryHighestBid( em, item );

    try {
        Bid newBid = new Bid( new BigDecimal( 15.00 ), item, highestBid );
        em.persist( newBid );
    } catch( InvalidBidException e ) {
        // Bid too low.
    }

    tx.commit( );
} ...
```

On loading the Item, the JPA implementation will attempt to acquire a database-level write-lock on the ITEM table row – even though the loaded Item won't be modified.

This persists a new Bid, as long as the bid is higher than the Item's present highest bid.

At commit time, the JPA implementation simply executes a SQL **insert** for a new valid Bid. The database-level lock is then released.



ID	ITEM_ID	TIMESTAMP	AMOUNT
1	116	23/08/15 10:26pm	$12.00
2	116	24/08/15 9:30am	$13.00
3	116	24/08/15 9:31am	$15.00

BIDS

Only this row exists prior to transaction execution

Row inserted when transaction A commits

Row inserted when transaction B commits

A Problem with Locking

```
try {
    EntityTransaction tx = em.getTransaction( );
    tx.begin( );

    Item item1 = em.find( Item.class, 116 );
    item1.setName( "First new name" );

    Item item2 = em.find( Item.class, 117 );
    item2.setName( "Second new name" );

    Bid highestBid = queryHighestBid( em, item );

    tx.commit( );
    em.close( );
} …
```

```
update ITEM set … where ID = 116;
update ITEM set … where ID = 117;
```

Locks ITEM row 116

Locks ITEM row 117

TxA

```
update ITEM set … where ID = 116;
update ITEM set … where ID = 117;
```

TxB

```
update ITEM set … where ID = 117;
update ITEM set … where ID = 116;
```

Solutions to Deadlock

- Use the Serializable isolation level

Lock whole tables and not rows.

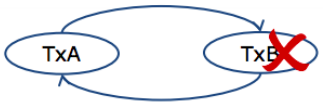
- Order database updates based on primary key values

Set property hibernate.order_updates to true in persistence.xml.

- Use a deadlock detection and resolution strategy
  - Lock timeouts

Where a transaction (A) has held a lock for more than a specified time, if another transaction (B) requests the lock, transaction A is aborted – releasing the lock.

  - Break deadlock cycles



Tutorials

Tutorial 1

How might you develop a TCP-like protocol that provides reliable and ordered communication over a virtual connection?

Issue	TCP behaviour
Validity	Lost packets are detected and resent
Integrity	A mandatory checksum is used to transform a corrupt packet into a lost packet
Ordering	Transmitted data is processed so that once received, it is delivered in the order in which it was sent; each packet has a sequence number
Blocking	The Sender can be blocked inserting data into an output stream; the receiver blocks if the input stream has insufficient data

Tutorial 2

1. What use do Serializable objects play in an RMI-based distributed system?
   For sending data between different parties. Parameters for remote objects are serialized.
2. What is the purpose of extending UnicastRemoteObject?
   Makes it easier to create remote objects, make it available to be consumed by clients.
3. In the context of Java RMI, what is a proxy? Where are they generated? And how does a client obtain one?
   Client interacts with object through proxy. Server adds remote object to registry, proxy created. Client gets it later on.
4. Consider the code below. Why does the server not stop running once this code is completed?

```
try {
    GreetingService greetingService = new GreetingServiceServant();
    System.out.println("Server up and running!");
} catch (RemoteException e) {
    e.printStackTrace();
}
```

   There are multiple threads running in the background for RMI "stuff", e.g. one for the socket, one for the naming service, leasing mechanism…
5. What invocation semantics does RMI offer? Suggest why this is the "best" we can do for such a system?
   Invocation semantics is when you invoke something, what is the behaviour expected. RMI offers "at most once" semantics. We don't know when we get an exception if the method has been called or not.
6. In a Java-based distributed system using HTTP as a communication protocol, what role does a servlet container play?

TomCat. In charge of receiving HTTP requests from all clients and making those requests available to servlets inside the container. Manage lifecycle of servlets. Security and authentication.

7. When entering a URL into a web browser, what kind of HTTP request is sent to the server?
   GET.
8. What is the role of WSDL in a SOAP-based web service?
   Describes a web service (Web Service Description Language). Contains information about endpoints/services in the web service. Web service version of interface.
9. In REST, which four HTTP methods correspond to the CRUD operations Create, Retrieve, Update, & Delete?
   Create: POST. Retrieve: GET. Update: PUT. Delete: DELETE.
10. What are the five REST principles? Describe each one briefly (one – two sentences each).
    Stateless communication: Each request/response is completely isolated. Good for multithreading.
    Hypermedia as the Engine of Application State: Response contains information about possible future pathways.
    Addressable Resources: Everything is represented by an unique identifier (URI).
    Uniform Constraint Interface: Well-defined interface and using it exactly as intended. Not trying to add extra features on top.
    Representation-Oriented: Client and server communicate between each other.
11. Compare Java RMI vs SOAP vs REST for creating distributed applications. For each one, give one scenario where that technology might be preferred over the alternatives.
    Java RMI can be preferred when everything is written in Java. Serialization is easy, RMI is easy.
    REST can be preferred because it is designed to use HTTP which is the backbone of the internet. Open and interoperable. Different kinds of the system don't need much more than HTTP.
    SOAP can be preferred when REST services considered too ad hoc. It is well-defined. The description is built right into the service itself. Discovery mechanisms better.

Tutorial 3

1. In REST, one of the principles is "stateless communication". Given this, how could one implement an authentication service – which is supposed to identify individual clients – using REST?
   Cookies. Authentication code which can be sent when needed.
2. What is the purpose of the Application class in a JAX-RS service?
   Describes what the service is to the JAX RS runtime. Let the JAX RS know the resources. Configuration of marshalling/unmarshalling…
3. What is the purpose of each of the following JAX-RS annotations:
   @Produces, @Path, @POST, @QueryParam?
   @Produces: Annotate a web method, lets you specify what it returns. Serialization.
   @Path: Specify URL that client needs to enter to enter the method.
   @Post: Specifies web method can be accessed with post.
   @QueryParam: The part of the URL after the ? is the query string, queryParam gets the param from the URL.

4. What HTTP status code should normally be returned from a request to update certain information, assuming the request is successful?
   204: No content.
5. Describe the steps that would be required to support Java serialization as a data transfer format in a JAX-RS web service?
   Implement MessageBodyReader and MessageBodyWriter. Need @Produces and @Consumes annotation. Give Application class a list of classes including the custom readers and writers.
6. How are Java Bean properties usually represented in terms of Java methods? Give an example.
   Getters and setters. Use all lower case except first letter.
7. Consider the following code:

```
public abstract class BillingDetails {

    private String user;

    public BillingDetails() { }
    public String getUser() { return user; }
    public void setUser(String user) { this.user = user; }
}

public class CreditCard extends BillingDetails {

    private String number;

    public CreditCard() { }

    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number;
}
```

   a) Describe the problems that would occur when attempting to marshall / unmarshall an instance of CreditCard.
      JAX RS will try to create a new BillingDetails instead of a CreditCard. Use @JsonTypeInfo to solve this problem.
   b) Give two solutions, using Jackson annotations, that can be used to solve this problem
      Use @JsonTypeInfo to include the CreditCard, or to use custom names.
   c) Compare the two solutions. What are the benefits / drawbacks of each?
      Benefit of first approach: only one annotation on super class. The result can look hard to read, and package information is tied to Json. Second approach: no package information. Drawback: Every time new class is added, would have to modify the application to include that class.
8. Describe a problem that might occur when marshalling / unmarshalling object graphs with cyclic dependencies using Jackson. Give two solutions to this problem, again using Jackson annotations. What are the benefits and drawbacks of each approach?
   Infinite loop. Stack overflow error.
   First solution: Use @Unique and the first time it will make the object, other times it will use the id. Benefit: reconstitute object graph. Drawback: Another system that doesn't use Jackson isn't necessarily usable.
   Second solution: Don't serialize using @Ignore. Benefit: Json produced doesn't have id values. Drawback: Extra work required to recreate the object graph.

Design question

A new web service is required for the city library, and you have been contracted to design the REST service interface.

Users of the system have an id, username, and a password. The system must be able to create new user accounts. Once a user's account is created, they must be able to authenticate with the system and be provided with an authentication token they can use on subsequent requests.

Whether or not a user is authenticated, they should be able to query a list of all **Books** in the library. Books have an id, title, author name, and a value indicating whether or not the book is currently on loan. Optionally, users may supply a start index and a size when making this query; in this case, only the specified number of books will be returned, along with links to the "previous" and "next" books in the collection. Users may also choose to view only books which are not currently on loan.

While authenticated, users must also be able to query a list of books which they currently have on loan. In addition, authenticated users must be able to borrow a book which is not already on loan.

If any request fails (due to an authentication error, or an attempt to borrow a non-existent book or one which is already on loan), and appropriate error code should be reported back to the client.

Describe the **REST endpoints** which you would implement in order to create the web service introduced in the above description. For each endpoint, provide the following information:

- HTTP method (e.g. GET, POST)
- Resource (URI -- e.g. `/users/{id}`)
- Required path, query, and cookie parameters (if any)
- Possible HTTP response status codes (on success and failure)
- Required HTTP request payload (if any)
- Expected HTTP response payload (if any)
- Other response data (e.g. cookies, if any)
- Description covering any detail you think is important which isn't included in the above points.

CRUD Operation	**HTTP method**	**Resource**	**Query params**	**Response headers**	**Message body JSON representation**
Create	POST	/users/signup	User info	201	Response message
Update	POST	/users/login	Usename password	200 401 fail	Cookie (authentication code)
Retrieve	GET	/books	(optional) startIndex size onLoan	200	List of books. (json)
Retrieve	GET	/{id}/books	Cookie	200 401 or 403 fail.	List of books
Update	POST	/books/borrow	Cookie, bookId	204 success 401 or 403 fail. 404 no book.	

Tutorial 4

1. With reference to the CAP theorem, what does the P stand for? What is a strategy that can improve P in a distributed system? What is a drawback / tradeoff of such an approach?
   Tolerant of partition failures. Can use replica / backup copies.
   Will reduce consistency – different parts of the system make it harder to ensure single version of truth. Can use other protocols to increase consistency but at the cost of performance.
2. What is a tradeoff that needs to be considered when introducing caching into a distributed system? Use DNS as an example.
   Benefit: Increases performance.
   Tradeoff: The longer you cache data for, the greater the chance it will become stale. Longer TTL values = more performance, shorter TTL values = more consistency.
3. Peer-to-peer file-sharing systems are suitable for distributing immutable data. Why do they work well for this purpose, but not for other kinds of data?
   So many different peers have different copies of the data. We don't need to do much to ensure consistency with immutable data, but changes to the data would need to somehow be propagated throughout the network – P2P systems aren't designed this way.
   Differences in files detected using hashing / checksum of the file contents.
4. With reference to the OO and relational models, what is meant by the term "granularity"? Give an example of where the granularity of a data representation might be different between the two models.
   Classes (OO) vs tables (relational). Example of a difference: Person has an Address, if this is one-to-one we can represent these two classes as one table in the database.
5. A one-to-many unidirectional association between two entity classes is being persisted in a database. Later, the developer is required to make that association bidirectional. What changes, if any, will need to be made to the database schema, and why?
   No changes to database schema – all FK relationships are two-way as it is.
6. JPA aims to provide transparent and automated persistence. Explain what is meant by "transparency" in this case? Suggest why complete transparency cannot be achieved?
   Transparency: Separation of concerns between your OO implementation and the underlying database. Transparency can't be 100% because of the need to add some annotations (@Entity and @Id at minimum), and the need to have PK fields in your OO classes.
7. Study the partial class definitions for Concert and Performer overleaf. Then, construct the relational schema that would be generated from these classes using Hibernate. The schema should include tables, columns, and primary and foreign keys (with correct names where appropriate).

```
@Entity
public class Concert {
  @Id
  @GeneratedValue
  private Long id;

  private String title;

  @ElementCollection
  private Set<java.time.LocalDateTime> dates;

  @ManyToOne(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
  private Performer performer;
  ...
}

@Entity
public class Performer
  @Id
  @GeneratedValue
  private Long id;

  @Enumerated
  private Genre genre;
  ...
}
```

- Concert (id, title, performer_id (FK on Performer))
- Performer (id, genre (String))
- Concert_dates (Concert_id (FK on Concert), dates)

Note naming conventions are default _ between the class/field.

Tutorial 5

1. Examine the code below, which is intended to form part of a vehicle rental system. Then, answer the questions below.

```
@Entity
public class User {
    @Id
    @GeneratedValue
    protected Long id;
    private String name;
    @OneToMany(cascade = CascadeType.ALL)
    private Set<Rental> rentals = new HashSet<>();
}

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Rental {

    @Id
    @GeneratedValue
    protected Long id;
    protected int year;
    protected String make;
    protected String model;
}

@Entity
public class Car extends Rental {

    protected boolean hasTowbar;
}

@Entity
public class Van extends Rental {

    protected int numSeats;
}
```

Figure 1: Partial class definitions for car rental system.

a) Write the relational schema that will be generated by JPA / Hibernate to persist the classes shown above. The schema should include any table names, columns, and keys.
Note InheritanceType.JOINED specifies there should be a table for all the entities.
User(id, name)
Rental (id, year, make, model, User_id [FK on User])
Car (id, hasTowbar)
Van (id, numSeats)

b) Suppose the inheritance strategy highlighted in bold is changed to InheritanceType.SINGLE_TABLE. What other annotations, if any, would need to be added to the class definitions?
@DiscriminatorColumn on Rental (defines the new column we use to tell cars and vans apart),
@DiscriminatorValue on Car and Van (defines the unique value that identifies as either Car or Van).

c) Assuming the changes identified in (b) are made, how would this affect the generated schema? What impact would this have on query performance and data integrity? Why? Which of the two strategies would you recommend in this case, and why?

All cars and vans (and other vehicles) would be stored in a single table, with a discriminator column to tell them apart. Query performance will increase – no joins.Data integrity will decrease – we aren't allowed nullable values, and some of our data is not dependent on the PK (e.g. if a row represents Van, then its hasTowbar field is irrelevant).

Could prefer single table as there are a lot of common fields, so the number of fields affected by the negative data integrity is small. Is worth it to achieve better query performance.

Could prefer to stick with joined. Small number of classes so won't be many joins, performance hit will be small. Not worth sacrificing data integrity and the ability to have compulsory fields.

2. The code below presents partial class definitions for a system which stores health records.

```
@Entity
@Table(name = "HEALTH_RECORD")
public class HealthRecord {

    @Id @GeneratedValue
    private Long id;
    private String name;
    private String address;

    @OneToMany(mappedBy="healthRecord", fetch=FetchType.LAZY,
cascade=CascadeType.ALL)
    private Set<Consultation> consultations = new HashSet<>();

    @ElementCollection
    private Set<String> ongoingConditions = new HashSet<>();

    public Set<Consultation> getConsultations() {
        return consultations; }

    public Set<String> getOngoingConditions() {
        return ongoingConditions; }
}

@Entity
@Table(name = "CONSULTATION")
public class Consultation {

    @Id @GeneratedValue
    private Long id;
    private String summary;

    @ManyToOne
    @JoinColumn(name = "HEALTH_RECORD_ID")
    private HealthRecord healthRecord;

    public String getSummary() { return summary; }
}
```

Figure 2: Partial class definitions for health record system

a) Write the relational schema that will be generated by JPA / Hibernate to persist the classes shown above. The schema should include any table names, columns, and keys.
HEALTH_RECORD (id, name, address)
CONSULTATION (id, summary, HEALTH_RECORD_ID [FK on HEALTH_RECORD])
HealthRecord_ongoingConditions (HealthRecord_id [FK on HealthRecord], ongoingConditions)

b) Study the code in figure 3 overleaf, which operates on instances of these classes. Explain the effect of this code on the database. With reference to the code, explain why this is the effect.
The transaction persists a HealthRecord entity, and its associated Consultation entity, and the Consultation is automatically persisted because of cascading persistence. The transaction also persists the two ongoingConditions rows. So therefore the effect is that one Consultation row, one HealthRecord row, and two ongoingConditions rows are added to the database.

c) Study the code in figure 4, that again uses instances of these classes.
   a. Assuming the database stores 300 health records, and each patient record has information about 3 consultations, how many database queries would be made in executing the transaction? Explain your answer.
   301. One from the initial fetch of all HealthRecords. Since fetch type is lazy, we don't get the information for the Consultations in this fetch. We then need 300 fetches, one for all the Consultation for each Health Record.
   b. Name and describe an optimization that can be applied to improve the efficiency of the code in figure 4.
   We could use FetchMode.SELECT/SUBSELECT to specify eager fetching for the Consultations. Then the first time we try to get the Consultations for any HealthRecord, it gets all the Consultations for all HealthRecords. Therefore we would only be using 2 query fetches instead of 301. There also won't be any joins for the query that gets the consultations. Changing it directly to always eager fetching is not a good answer. We could have an enormous Cartesian product.

```
EntityManager em;
...

em.getTransaction().begin();

HealthRecord record = new HealthRecord("Bob", "123 Some Street");
Consultation consult = new Consultation("Examined for common cold symptoms");
record.getConsultations().add(consult);
record.getOngoingConditions().add("Coughing");
record.getOngoingConditions().add("Runny nose");

em.persist(record);

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

Figure 3: Adding a new HealthRecord

```
EntityManager em;
…

em.getTransaction().begin();

List<HealthRecord> records = em
    .createQuery("select r from HealthRecord r", HealthRecord.class)
    .getResultList();

for (HealthRecord r : records) {
    for (Consultation c : r.getConsultations()) {
        System.out.println("Consultation details: " + c.getSummary());
    }
}

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

Figure 4: Querying information about health records

Tutorial 6

1. This question refers to asynchronous web services.
    a. Describe, using examples, two situations where an asynchronous communication protocol would be beneficial compared with a traditional request-reply protocol.
    b. Consider the code below. With reference to the code, the HTTP protocol, and the Servlet 3 specification, describe how a client would subscribe to this service, and how they would eventually receive any notifications.
    c. Again with reference to the code, and your answer to (b) above, describe why the HTTP protocol may not be ideal for some forms of asynchronous communication.

```
@Path("/notifications")
public class NotificationResource {

    private final List<AsyncResponse> subs = new ArrayList<>();

    @GET
    @Path("/sub")
    @Produces(MediaType.TEXT_PLAIN)
    public void subscribe(@Suspended AsyncResponse sub) {
        synchronized(subs) { subs.add(sub); }
    }

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public Response sendNotification(String notification) {

        synchronized (subs) {
            for (AsyncResponse sub : subs) {
                sub.resume(message);
            }
            subs.clear();
        }

        return Response.ok().build();
    }
}
```

2. This question deals with Transactions.
    a. Describe the difference between optimistic and pessimistic concurrency control. Suggest an example where optimistic concurrency control might be preferred.
    b. Consider the code overleaf, showing a BankAccount class and a transaction involving accounts. Describe how two instances of this transaction could conflict.
    c. Identify any changes you would make to address the conflict using optimistic concurrency control techniques supported by JPA. Describe the effect of the changes.
    d. Identify any changes you would make to address the conflict using pessimistic concurrency control techniques supported by JPA. Describe the effect of the changes.

```
@Entity
public class BankAccount {
    @Id @GeneratedValue
    private Long id;

    private String number;
    private BigDecimal balanceInCents;

    public void deposit(BigDecimal amount) {
        balanceInCents = balanceInCents.add(amount);
    }

    public boolean withdraw(BigDecimal amount) {
        boolean success = false;

        /* Withdaw funds if the balance after
           withdrawing is non-negative. */
        if(balanceInCents.subtract(amount).doubleValue() >= 0.0) {
            balanceInCents = balanceInCents.subtract(amount);
            success = true;
        }
        return success;
    }

    public BigDecimal getBalance() { return balanceInCents; }
}
```

```
BigDecimal depositAmount = ...;

EntityTransaction tx = em.getTransaction();
tx.begin();
BankAccount account = em.find(BankAccount.class, 1088);
account.deposit(depositAmount);
tx.commit();
```

Software Architecture Introduction

The meaning of Architecture

The process and the product of planning, designing, and constructing buildings or other structures.

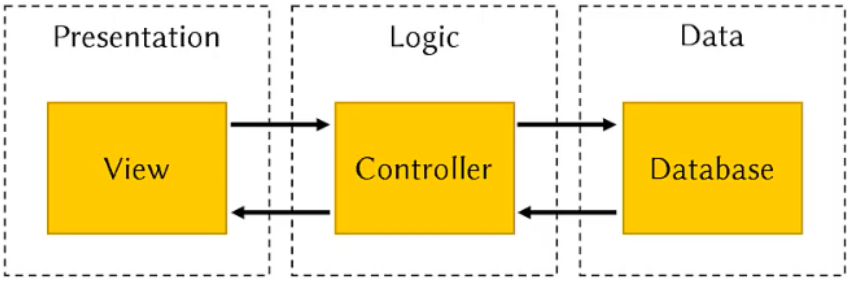
Why Architecture is Important to Learn

- Understand
- Communicate
- Proven solution

Each view provides different information

Software Architecture

The set of structures needed to reason about the system, which comprise externally visible software elements, relations among them, and the properties of both.



- Every system has an architecture
- Architecture is not a random decision
    - Choose certain types of architecture based on the needs.
- Architecture is not a box-and-line drawing
    - We can express a lot with pictures, but it is not a single view.
- We must document architecture
- It includes software architecture and the hardware that the software runs on.

Software Architecture Types

- **Classification:** What kinds (or “styles”) of architectures are there?
- **Analysis:** How to reason about an architecture effectively?
- **Develop:** How do we decide on an architecture for a system? What issues should we keep in mind?
- **Documentation:** How should we describe architectures? What are the important things to document?

Software Development Lifecycle

- Requirements
- Specification
- Architecture
- Design (in detail)
- Implementations

Quality Attributes

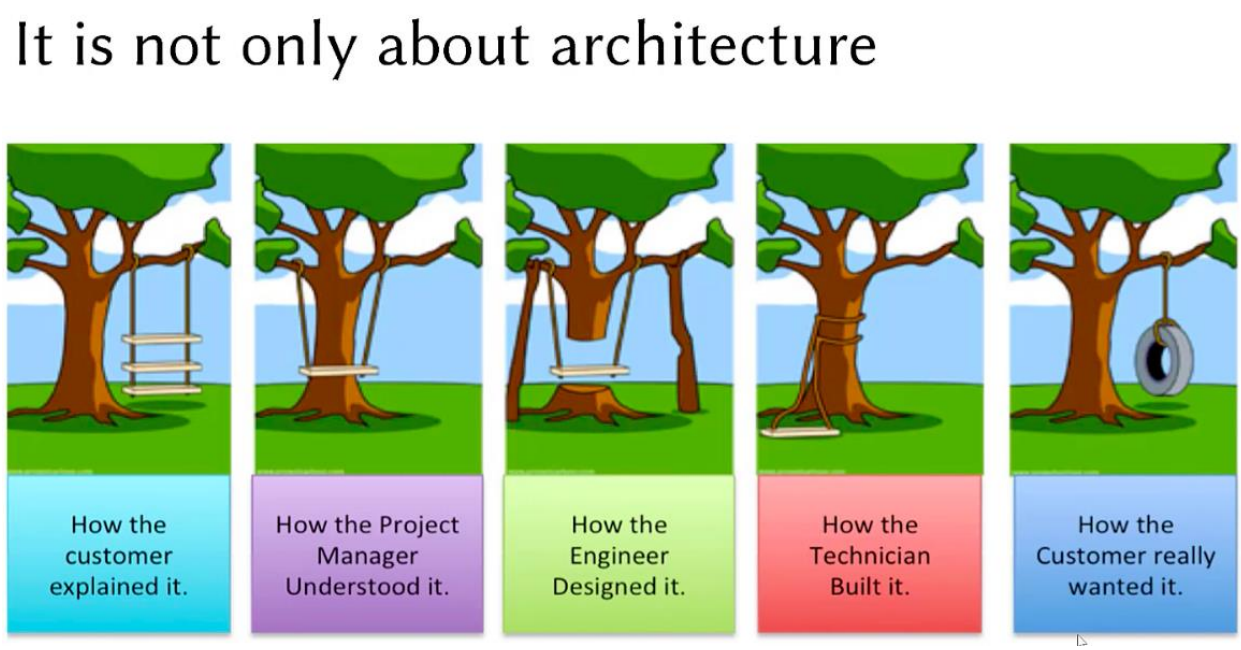
A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.

Examples:

- Availability
- Modifiability
- Performance
- Usability
- Security

Business Quality Attributes:

- Market
- Time to market
- System lifetime
- Integration with legacy systems



How to Ensure a Certain Requirement is Met?

- We should understand the client's expectations
- We need "operational" description of quality requirements

Quality Attribute Scenarios

- Stimulus: Event that needs to be occurred at (some part of) the system.
- Source of stimulus: Some entity that generates a stimulus.
- Environment: Conditions under which the stimulus occurs.
- Artifact: The things that are stimulated.r
- Response: What the artifact should do on arrival of the stimulus.
- Response measure: How to measure response to make sure that it is satisfactory.

Example: Fast v1

- Stimulus: Form submission.
- Source of stimulus: The user.
- Environment: Normal working conditions (network is up, server is not overloaded, database is accessible).
- Artifact: The system.
- Response: Load and display page indicating successful purchase.
- Response measure: The page is loaded in less than 2 seconds 95% of the time.

Example: Fast v2

- Stimulus: Form submission.
- Source of stimulus: The user.
- Environment: Normal working conditions (network is up, server is not overloaded, database is accessible).
- Artifact: The server.
- Response: Verify and record purchase.
- Response measure: Process 1000 transactions per second.

Example: Reliable v1

- Stimulus: Single CPU failure.
- Source of stimulus: Random event.
- Environment: Normal working conditions.
- Artifact: Server.
- Response: Notify operator.
- Response measure: No loss of service.

Example: Reliable v2

- Stimulus: Request a service.
- Source of stimulus: The user.
- Environment: Receiving more than 1000 requests per second.
- Artifact: The server.
- Response: Reject request, report to operator.
- Response measure: Operator must be notified within 5 minutes.

Example: Secure v1

- Stimulus: Attempts to change details on the web server.
- Source of stimulus: External user.
- Environment: Normal working conditions.
- Artifact: Server.
- Response: Notify operator with details.
- Response measure: No loss of service.

Example: Secure v2

- Stimulus: Attempts to change details on the web server.
- Source of stimulus: Internal user.
- Environment: Normal working conditions.
- Artifact: Server.
- Response: Record access.
- Response measure: Identity of the person making the change is available.

Quality Attribute Scenarios

Finding Relevant Scenarios

- The client dictates which are the important ones
- Develop general attribute scenarios and move towards concrete scenarios.

General Scenario: Performance

Portion of scenario	Possible values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response measure	Latency, deadline. throughput, jitter, miss rate

Definitions

- Periodic: Stimuli occur at regular (and known) intervals.
- Sporadic: Time between stimuli is completely unpredictable.
- Stochastic: Time between stimuli follows particular probability distribution (e.g. “Likelihood decreases over time’)
- Latency: Time between stimulus and response.
- Deadline: Time by when the response must be ready (independent of stimulus).
- Jitter: Variation in latency or deadline.
- Throughput: The number of requests that can be processed within a particular time interval.

- Miss rate: Events that cannot be processed

Concrete Scenario: Performance

	General value	Concrete value
Source	Independent source	The user
Stimulus	Sporadic event	Form submission
Artifact	System	System
Environment	Normal mode	Server is not overloaded
Response	Process the stimulus	Load and display page indicating successful purchase
Response measure	Latency	The page is loaded in less than two seconds

General Scenario: Availability

Portion of scenario	Possible values
Source	Internal/external: people, hardware, software, physical infrastructure
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, over1oaded operation
Response	Prevent the fault from becoming a failure (detect and recover)
Response measure	Time interval when the system must be available, availability percentage, time to detect the fault, time to repair the fault, time interval in which system can be in degraded mode, proportion of class of faults that the system prevents or handles without failing

Concrete Scenario: Availability

	General value	Concrete value
Source	Internal to system	Random event
Stimulus	Crash	Single processor failure
Artifact	System's processor	Server
Environment	Normal operation	Normal working condition
Response	Notify related parties	Notify operator
Response measure	Availability percentage	No loss of service (i.e. 100% availability)

General Scenario: Modifiability

Portion of scenario	Possible values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifact	Code, data, interfaces, components, resources, configurations, etc.
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: make modification, test modification, or deploy modification
Response measure	Cost in terms of number, size, complexity of affected artifacts; effort; calendar time; money; extent to which this modification affects other functions or quality attributes; or new defects introduced

Concrete Scenario: Modifiability

	General value	Concrete value
Source	System admin	The owner
Stimulus	Change capacity	Request new site to be added with 10-100 users
Artifact	Platform	The system
Environment	Runtime	Normal working condition
Response	Deploy modifications	New site is set up
Response measure		Downtime does not exceed 48 hours

General Scenario: Security

Portion of scenario	Possible values
Source	Internal or external human or another system that may be unknown to the system
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	System continues its normal operation without disclosing protected information, and it records any "suspicious" activity
Response measure	Harms against the system in terms of time, services, assets, etc.

Security-Related Response

- Data or services are protected from unauthorized access.
- Data or services are not being manipulated without authorization.
- Parties to a transaction are identified with assurance.
- The parties to the transaction cannot repudiate (reject) their involvements.
- Recording attempts to access data, resources or services.
- Notifying appropriate entities (people or systems) when an apparent attack is occurring.

Security-Related Response Measure

- How much of a system is compromised when a particular component or data value is compromised?
- How much time was passed before an attack was detected?
- How many attacks were resisted?
- How long does it take to recover from a successful attack?

Concrete Scenario: Security

	General value	Concrete value
Source	External human	Unknown user
Stimulus	Unauthorized access	Try to access system without authentication
Artifact	System data	Database
Environment	Online system	Online web
Response	Block access	Refuse any access to sensitive data
Response measure	No data disclosure	Detect the attack after 10 requests

General Scenario: Usability

Portion of scenario	Possible values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system
Artifact	System or the specific portion of the system with which the user is interacting
Environment	Runtime or configuration time
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs

Concrete Scenario: Usability

	General value	Concrete value
Source	End user	
Stimulus	Trying to use a system efficiently	
Artifact	System	
Environment	Runtime	
Response	The system should provide the user with the required features	
Response measure	User should figure out how to do the task shortly	

General Scenario: Interoperability

Portion of scenario	Possible values
Source	A system initiates a request to interoperate with another system
Stimulus	A request to exchange Information among system(s)
Artifact	The systems that wish to interoperate
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime
Response	• The request is (appropriately) rejected and appropriate entities (people or systems) are notified • The request is (appropriately) accepted and information is exchanged successfully • The request is logged by one or more of the involved systems
Response measure	Percentage of information exchanges correctly processed/rejected

Concrete Scenario: Interoperability

	General value	Concrete value
Source		
Stimulus		
Artifact		
Environment		
Response		
Response measure		

General Scenario: Testability

Portion of scenario	Possible values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding task, the completed Integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer
Artifact	The portion of the system being tested
Environment	Design time, development time, compile time, integration time, deployment time, run time
Response	One or more of the following: execute. test suite and capture resuts, capture activity that resulted in the fault, control and monitor the state of the system
Response measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure

Concrete Scenario: Testability

General value	Concrete value
Source	
Stimulus	
Artifact	
Environment	
Response	
Response measure	

Evaluation of Quality Attribute Scenarios

- Is each concrete value consistent with the general value?
- Is the scenario appropriate for the quality attribute?
- Does the artifact receive the stimulus?
- Does the source generate the stimulus?
- Are the environment details relevant to the stimulus and response?
- Is the response relevant to the stimulus?
- Is the “measure” really a measure, in particular, testable?
- Does the measure apply to the response?

Structures and Views

Structure

A set of elements held together by a relation.

- Software systems are composed of different types of structures.
    - Static, Runtime, and Allocation
- No single structure holds claim to be the architecture.
- Each type of structure provides a particular view (useful to reason about specific quality attributes).
- Each structure deals with specific types of elements (classes, processes, etc).

Static/Module Structures

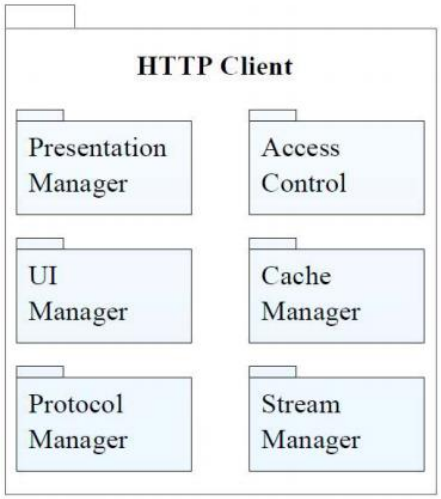
They focus on the way the system's functionality is divided up and assigned to in terms of units of implementation.

Why a module structure is useful?

- What is the primary functional responsibility assigned to each module?
- What other software elements is a module allowed to use?
- What other software does it actually use and depend on?
- What modules are related to other modules by generalization or specialization (i.e. inheritance) relationships?

Module Structure: Decomposition View

It shows how modules are decomposed into smaller modules (i.e., sub-modules) recursively until the modules are small enough to be easily understood.

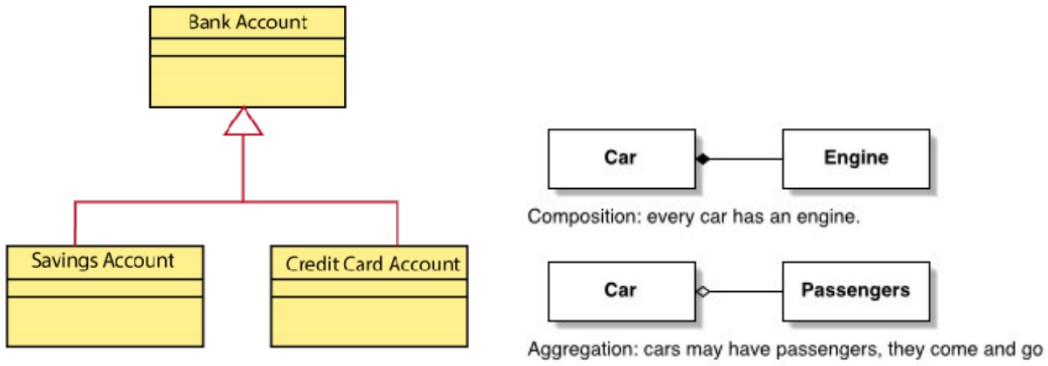


Module Structure: Uses View

A unit of software (for example, a class) uses another if the correctness of the first requires the presence of a correctly functioning version of the second.

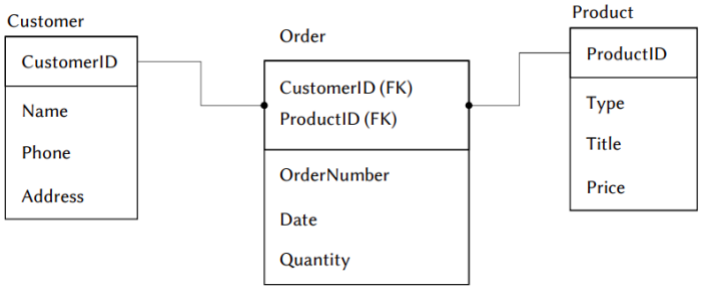
Module Structure: Class View

It presents the relations such as “inherits from” or “is an instance of” between classes.



Module Structure: Data Model View

Data model describes data entities and their relationships.



Runtime/Component-and-Connector Structures

They focus on the way the elements interact with each other at runtime to carry out the system’s functions.

Why a runtime structure is useful

- What are the major executing components and how do they interact at runtime?
- What are the major shared data stores?
- Which parts of the system are replicated?
- How does data progress through the system?
- What parts of the system can run in parallel?
- Can the system’s structure change as it executes and if so, how?

C&C Structure: Service View

It shows how services interoperate with each other by service coordination mechanisms.

C&C Structure: Concurrency View

It allows the architect to determine opportunities for parallelism and the locations where resource contention may occur.

Allocation Structures

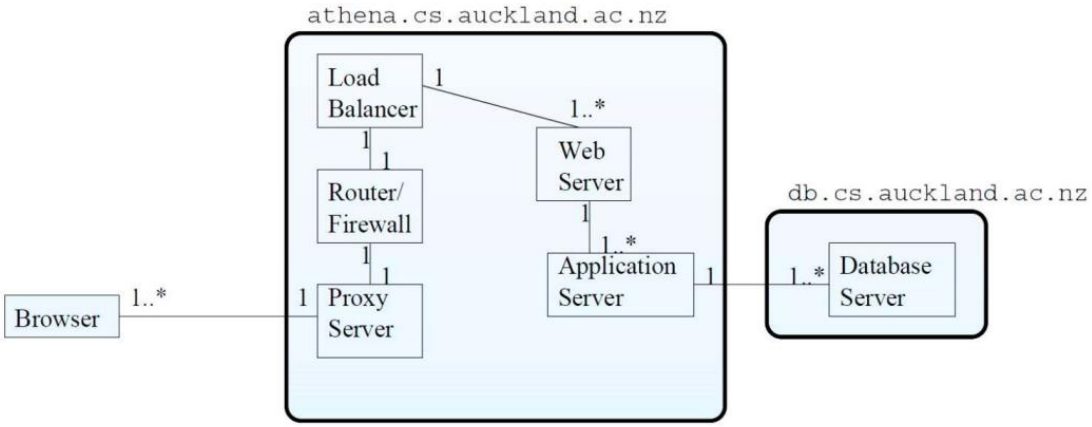
They define how the elements from C&C or module structures map onto things that are not software such as hardware, development teams, file systems, etc.

Why an Allocation Structure is useful

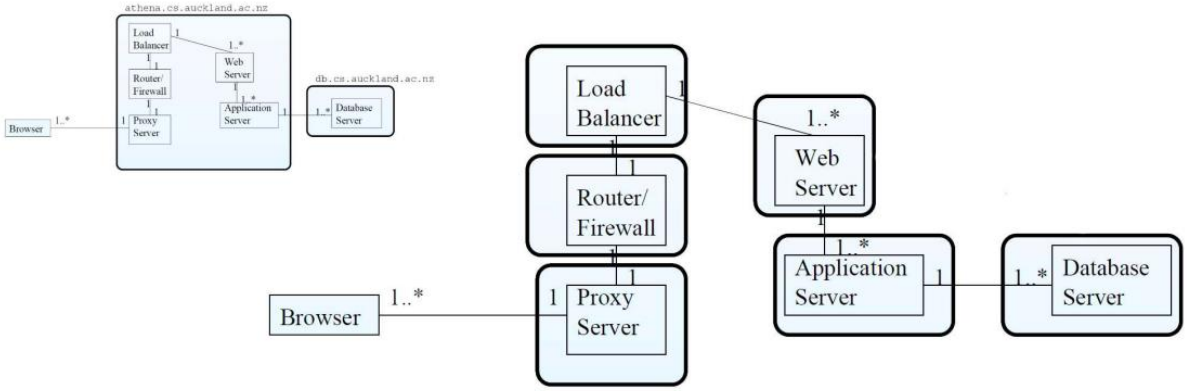
- What processor does each software element execute on?
- In what directories or files is each element stored during development, testing, and system building?
- What is the assignment of each software element to development teams?

Allocation Structure: Deployment View

It shows how software is assigned to hardware processors and communication elements. Helpful to reason about, e.g., performance, security, and availability.



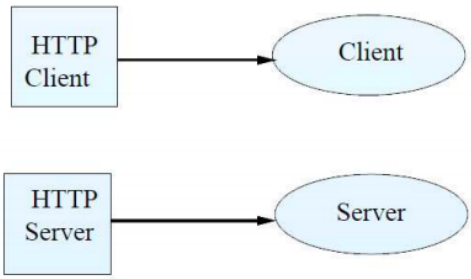
- Servers are in round rectangles and processes are the non-rounded rectangles.
- A lot of processes are on the athena which may cause performance issues.
- Deployment view shows which physical units the software elements reside and migrates to if the allocation is dynamic.



- Instead of putting the processes in the athena server, we put each process on a new server.
- Causes improvement in performance. (more computing devices)
- Athena server will no longer be the bottleneck in the system.

Allocation Structure: Implementation View

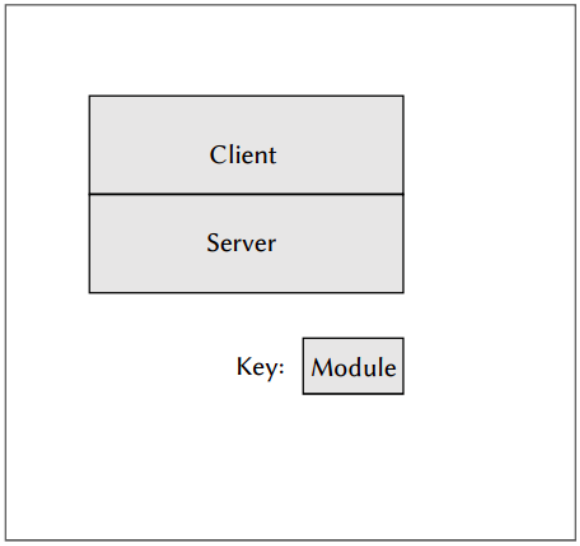
This structure shows how software elements are mapped to the file structure(s) in the system's development, integration, or configuration control environments.



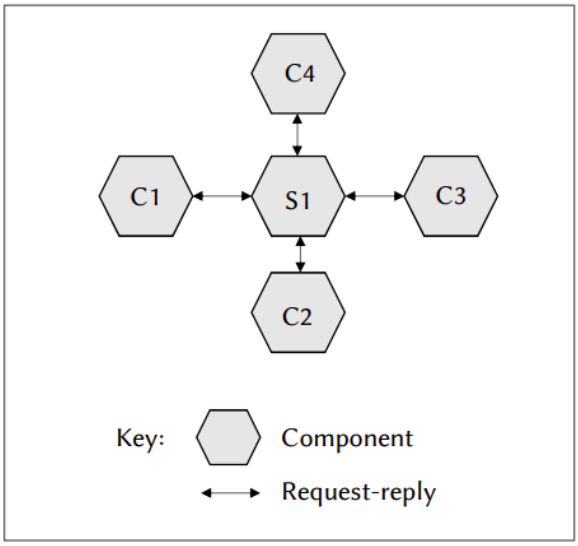
Allocation Structure: Work Assignment View

It assigns responsibility for implementing and integrating the modules to the teams who will carry it out.

Do we need all architectural structures?



Decomposition view



Component-and-connector view

Although each view shows a different system perspective, they are not independent.

A module in a decomposition structure may be manifested as one part of one or se

What structures to use?

In general, you should design and document a structure only if doing so brings you a positive return on the investment, usually in terms of decreased development or maintenance costs.

Question 1

In order to determine whether or not an architecture meets modifiability scenarios relating to change of functionality, what kinds of software elements should be considered?

- Looking at code -> modular structure
- Most useful are uses view and decomposition views to show dependencies and help with impact analysis.

Question 2

In order to determine whether or not an architecture meets modifiability scenarios relating to change run-time performance behaviour, what kinds of software elements should be considered?

- Runtime effects -> C&C Views
- Communicating process view should be helpful to make sure that the changes do not introduce deadlock. It is similar to service view, instead of services we have processes that communicate together.

Question 3

In order to determine which things to run on the same machine, what kinds of relationships should be considered?

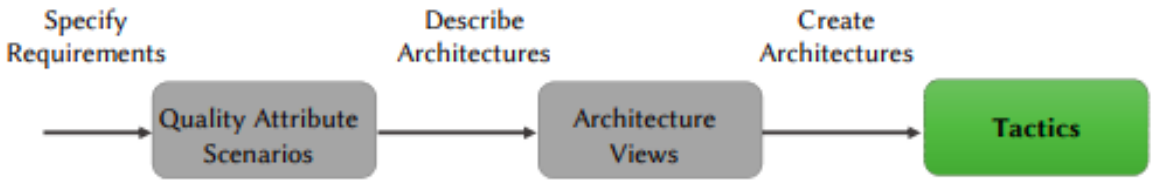
- Running stuff on hardware -> allocation structure, deployment view

Question 4

In order to determine whether or not an architecture meets its security goals, what kinds of relationships should be considered?

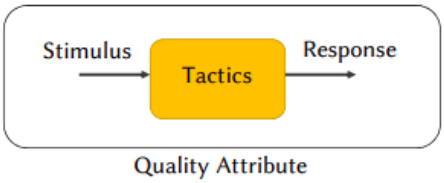
- Deployment view shows outside connections.
- CNC structure shows data flow which helps tracking where information goes and is exposed.
- Module decomposition helps to identify where the security related concerns are related.
- Work assignment view might help us identify who are responsible for developing security sensitive features and whether they have required expertise.

Tactics - Availability and Performance



Tactics

Techniques that an architect can use to achieve the required quality attributes.

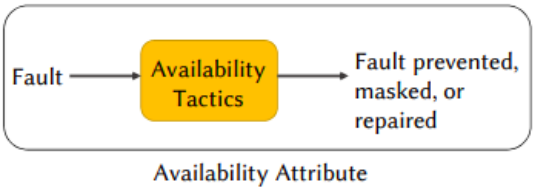


- There are often multiple tactics to improve a particular quality attribute.
- These tactics will overlap.
- Which one to use depends on factors such as tradeoffs among other quality attributes and the cost to implement.

Availability

Availability means that if we have a working system it should continue to work and deliver its services based on its specification.

- e.g. Availability of ATM: 24/7. Availability of stock machine: 8am-12pm.
- Could include down-time of maintenance
- Faults could be:
    - Omission: fails to respond to input
    - Crash: omission occurs repeatedly
    - Incorrect timing: response too early/late
    - Incorrect response: response result
- Mechanisms to prevent failure from fault, or mask fault from user.
- Goal of availability is to maximum mean time to failure minimize mean time to repair.



Detection

In order to detect whether something is wrong or not we cannot really rely on the human. We need some automated approaches. We usually rely on a monitor that can quickly inform us regarding issues.

- Ping: Monitor actively inquires the status of the target component.
- Heartbeat: Worker responsibility to signal monitor in a given time interval.
- Self-test: The component receives an input and produces an output. Component performs some tests on its own to make sure it's functioning properly. If it isn't, it may notify the monitor.
- Time stamp: Used in order to ensure that the order of the sequence of events are properly received and sent. E.g. Client may add timestamp/order number to packets to server.
- Exceptions: When there is an error in the component, we raise the exception and related parties (e.g. monitor) is informed.
- There is an overhead from the ping and heartbeat in the system since we need to send packets. Instead of sending these packets individually, we could send them along with other communication between the monitor and the component. Minimize overhead.

Recovery – Repair

Once we identify there is an issue, we need to repair it. There are possibilities to make sure that we repair the issue.

- Active redundancy: For a given request there are different replicas of a component. The client uses the fastest response and discards the rest. High load on network because for every request there is a connection between clients and all the servers. At the same time all the servers are working. If one server goes down, the client can still receive the response from the other servers.
- Voting (triple modular redundancy): There is a single point (voter) which distributes the request to these servers, and the voter collected the responses from the servers and takes what the majority says to send back to the client. Assumes that the minority is out of order, notifies system maintainer.
- Passive redundancy: There is one server that responds to a client's request. There are other servers that are not actively responding to the client's requests, but they have the capability to. If something goes wrong with one server, the client request will be sent to another server with the same capability.
- Spare: Most overhead. We need to call maintenance team to come and run a new system which is a spare and can provide the expected service to the clients. Until we provide this per system there might be an availability in our system which may or may not be accepted. If timing matters, we need to rely on more automated approaches.

Recovery – Reintroduction

The component that was malfunctioning needs to be reintroduced into the system.

- Shadow: Run in parallel to check correction works.
    - Make sure the new component works in parallel with the other components.
- State synchronization: ensure state matches current state.
    - Need to make sure we bring new component to state of system.
- Rollback: Keep record of valid states.
    - When states are not valid (failure), we need to rollback and make sure all components are in valid states that does not yield fault and then failure.

Prevention

Avoid failure in the first place.

- Removal from service: scheduled downtime to deal with problems.
- Transactions: only allow "valid" states and rollback the rest.
- Monitor the system: determine that some part of the system is in a faulty state before failure occurs.

Exercise

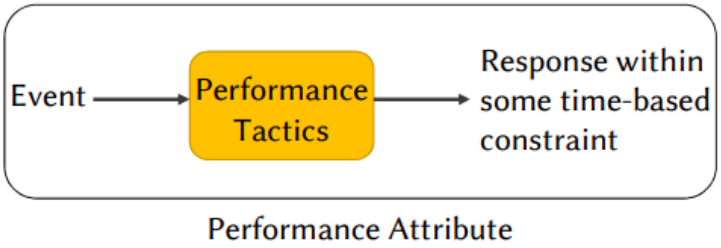
What tactics are useful to develop an architecture that meets this requirement?

Part	Details
Stimulus	A crash occurs
Source	A random event
Artifact	One of the servers
Environment	No other system or networking problems
Response	Record the failure
Measure	... within 10 minutes and there is no loss of service

- We don't want loss of service, so we need to look at availability and redundancy. The type of redundancy to use depends on our requirements. We don't want a loss of service so we should come up with automated approaches like active redundancy or passive redundancy.
- We want to make sure that the failure is recorded within 10 minutes. We need a tactic to take care of this detection. We can use the heartbeat or the ping. The time interval should not be more than 10 minutes. We don't want loss of service so we should have a frequent check.

Performance

Performance is the time taken to correctly answer to a request. We might have a set of events (stimuli). The response should be ready within a specific time constraint.



Performance Attribute

Performance Factors

- **Processing Time**
  - Resource consumption
  - Some of the processing is not directly task related (e.g. marshalling parameters for network communication)
- **Blocked Time**
  - Contention for resources
  - Unavailable resources
  - Dependency on other components

Performance-related Tactics

- Resource control (demand side)
  - Produce smaller demand on the resources that we have to service the events.
- Resource management (response side)
  - Use resources that we have more effectively in handling the demands.
- Resource arbitration
  - Arbitration decides to which request one should allocate a resource.
  - There may be policies for what requests to service first, etc

Control Resource Demand

- Reduce resources required for processing stimuli
  - Increase computational efficiency (e.g. use better algorithms)
  - Reduce computational overhead (e.g. minimize inter-component communication)
- Decrease number of events to process
  - Control frequency of sampling
  - Limit event response (relevant to discrete events)
- Control resources consumed
  - Bound execution times
  - Bound queue sizes

Example – Online Shopping

- Increase computational efficiency – use faster search algorithms
- Reduce computational overhead – co-locate processes to avoid remote communication
- Manage event rate – decrease the frequency of checking the availability of a product
- Control frequency of sampling – profiling user activity in longer time interval
- Bound execution times – if a search takes too long just give up
- Bound queue sizes – if queueing search requests on a processor and its queue is full just drop new jobs

Resource Management

- Increase available resources
    - E.g. adding new servers, increase configuration of servers, better cpus, larger memories
- Introduce concurrency
    - Different servers that can process requests at once.
    - Might have load balancer that decides where to direct requests
- Maintain multiple copies of either data or computations
    - Prevent single point of failure

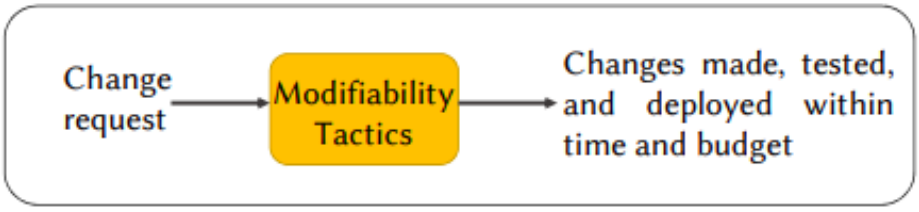
Arbitration

- Scheduling
    - Optimal resource usage
    - Minimizing number of resources used
    - Ensure fairness
    - Prevent starvation
    - Optimal processing of “important” requests
- Dispatching
    - Can occur any time
    - Can occur only at specific pre-emption points
    - Can occur only when nothing else is executing

Scheduling Policies

- First in/First out
    - Not that good an option for urgent requests
- Fixed priority
    - Semantic importance
    - Deadline importance
- Dynamic priority
    - Round robin
        - Not that good an option for urgent requests
    - Earliest deadline first

Modifiability



Modifiability Attribute

- What can change? Functionality, quality attributes, hardware.
- Who can make the change? The developer, system admin, end user.
- At what time a change can happen? Implementation, compilation, installation, execution.

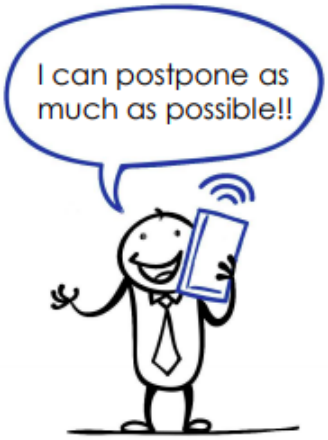
Tradeoff in modifiability

Modifiability Goals

- Reduce size
- Reduce coupling – the probability that a change to one module will propagate to another module.
- Increase cohesion – the relevance of responsibilities in a module.
- Defer binding time.

Defer Binding Time

- Critical.
- To reach high modifiability, we need to bind values as late as possible.
- E.g. there is a system where you can specify how many users are allowed to use your system. If this number is hardcoded, and later if there needs to be a change to this number, we need to adapt the code, recompile it, and deploy it. Expensive. Instead, we could extract these values from a config file which we can simply change at any time.
- Sometimes we do not have access to the running environment. E.g. getting users to update their apps is difficult.



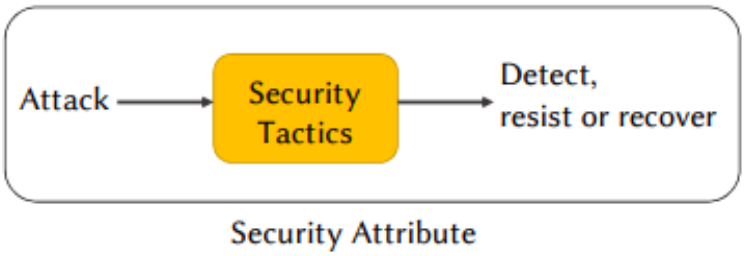
Modifiability Tactics

- Reduce size
    - Split modules – understanding code for small modules will be easier. Take into account cohesion.
- Increase cohesion
    - Increase semantic coherence – whatever exists within one module should be semantically relevant.
- Reduce coupling
    - Encapsulation – defining interfaces, making sure other modules are connected through the interface. If there are changes in our module, those will be hidden from those that are dependent on our module.
    - Using intermediaries – e.g. If we have a consumer and producer, we could also have a database that acts as the intermediary between the two.
    - Restrict dependencies – Controlling what modules one can access.
    - Refactor – To change module.
    - Abstract common services – Having services that are used in different places in the system. E.g. GPS service that can provide services to different parts of the application.
- Defer binding time
    - Compile time – Parameterialized compilers.
    - Deployment time – Configuration files.
    - Initialization time – Resource files.
    - Runtime – Polymorphism. Look up resources and connect to them.

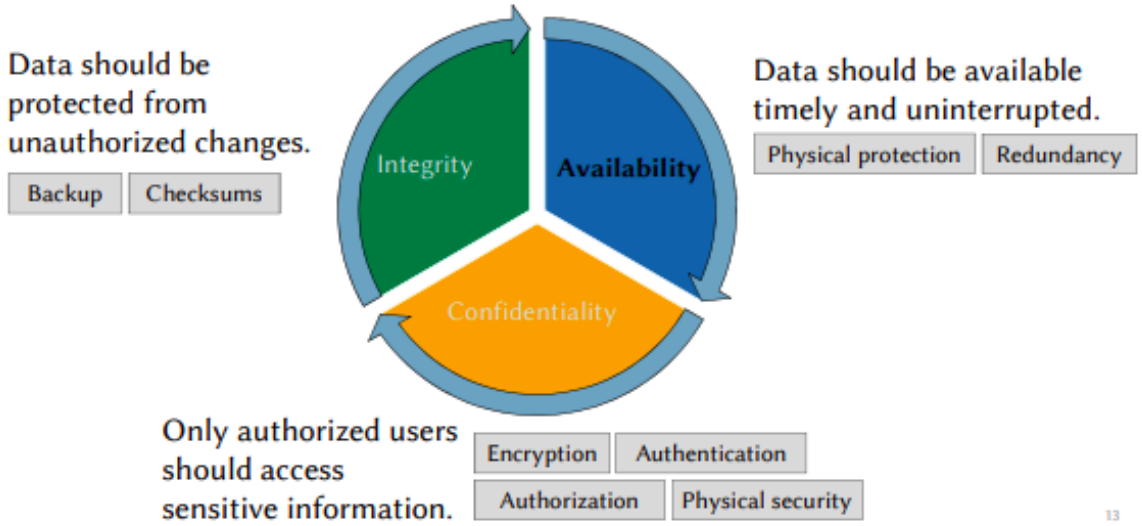
Online Shopping Example

- It has to communicate with different payment systems. E.g. PayPal, bank transfer.
    - Use a single payment module to increase semantic coherence.
- It should provide the payment status for all systems in the same way.
    - Use an intermediary and abstract common services.
- Airlines can join or leave any time.
    - Runtime registration.

Security



Key Principles to Develop Security



Confidentiality – only authorized users should be able to access sensitive information.

- We can apply encryption such that only those with a decryption key are able to read data.
- Authentication – the process of verifying who a user is.
- Authorization – the process verifying what that user has access to.
- Physical security is important as well.

Integrity – the data should be protected from unauthorized changes.

- Backup
- Checksums – Can identify changes in data.

Availability – data should be available timely and uninterrupted. Depends on availability specification.

- Physical protection
- Redundancy

React to attacks

- Revoke access
  - Limit users or uses.
- Lock computer
  - Prevent users or uses.
- Inform actors
  - Notify relevant parties.

Recover from attacks

- Restore state
  - Similar to availability.
- Audit trial
  - Record each transaction and the identity associated to it.

Testability

Testability Attribute

Once we complete a unit, we want to make sure that we are able to detect faults in the unit or the system caused by that unit before it gets into production.

Testability Tactics

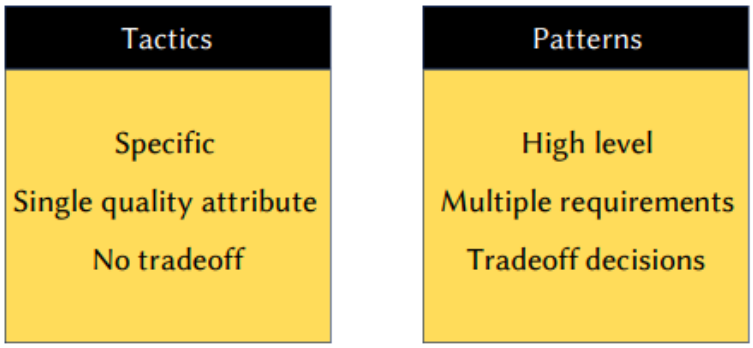
- Specialized interfaces – specialized testing interfaces like getters and setters, report networks or reset methods that allow us to control or capture variable values. These interfaces and methods should be clearly identified or be separated from other methods that belong to your actual functionality so that they can be removed if needed. We use these interfaces during testing in order to be able to monitor what's the status of each value.
- Record and playback – Make sure that the states that cause a fault in the system can be captured. If there is a fault in the system, it can be difficult to recreate what was the exact value that the system failed. Recording the state helps that you can use these states later to play the system back and to recreate the fault. We record what caused the fault and play back so that we can test our system.
- Localize state storage – important to start a system subsystem or module in an arbitrary estate for a test.
- Abstract data sources
- Sandboxing – Isolating an interface of the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- Executable assertions – Usually hand coded and placed at a desired location in the tests to indicate when and where a program is in a faulty state.
- Limit structural complexity

- Limit nondeterminism – Flaky. Sometimes it runs, sometimes it doesn't. There are tactics that help with this if it is unavoidable.

Architectural Patterns

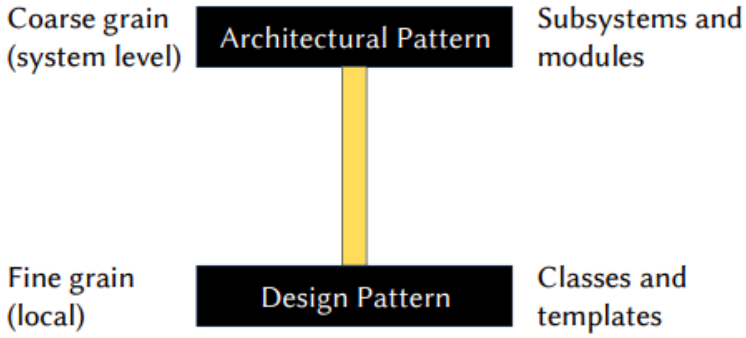
A pattern is a specific composition of elements that solve multiple (functional and non-functional) system requirements.

Pattern vs Tactics



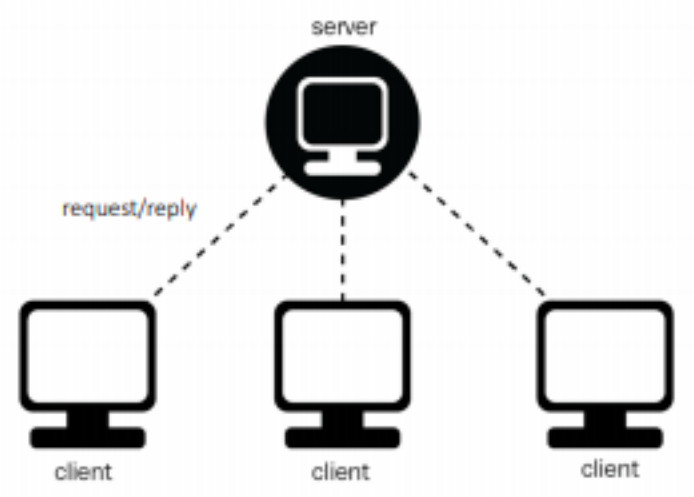
Tactics are building blocks of patterns. Each pattern is composed of multiple tactics.

Architectural Patterns vs Design Patterns



Both design decisions. Architectural patterns have system-wide consequences. We are concerned about subsystems and modules while in design patterns we are very fine-grained. We only care about the design of classes and templated. Design patterns aren't in the scope of this course.

Client-Server Architecture



In this setup clients only request the server and the server responds to clients. This pattern separates client applications from the services they use. It simplifies systems by factoring out common services which are reusable because servers can be accessed by any number of clients it is easy to add new clients to a system. Similarly, servers may be replaced to support the scalability or availability.

- Some components may act as both clients and servers.
- There may be one or multiple distributed servers.
- In some Client-Server patterns, servers are also permitted to initiate certain actions on their clients, e.g. notifications.
- Common example is the websites that we use regularly. Clients = browsers, servers = web server.

Why Client-Server Architecture?

- Modifiability: High. We can always add more servers and change servers. This will happen without clients noticing -> high availability.
- Availability
- Performance: Load balancer which sends requests to different servers so we can scale up the system.
- Security: We can add security mechanisms, since there is only one single point of service. E.g. Firewalls, intrusion detection systems, authentication mechanisms.
- Since we have a dedicated number of servers, they could be bottlenecks, but it is easy to fix by adding more resources.