REST API Development

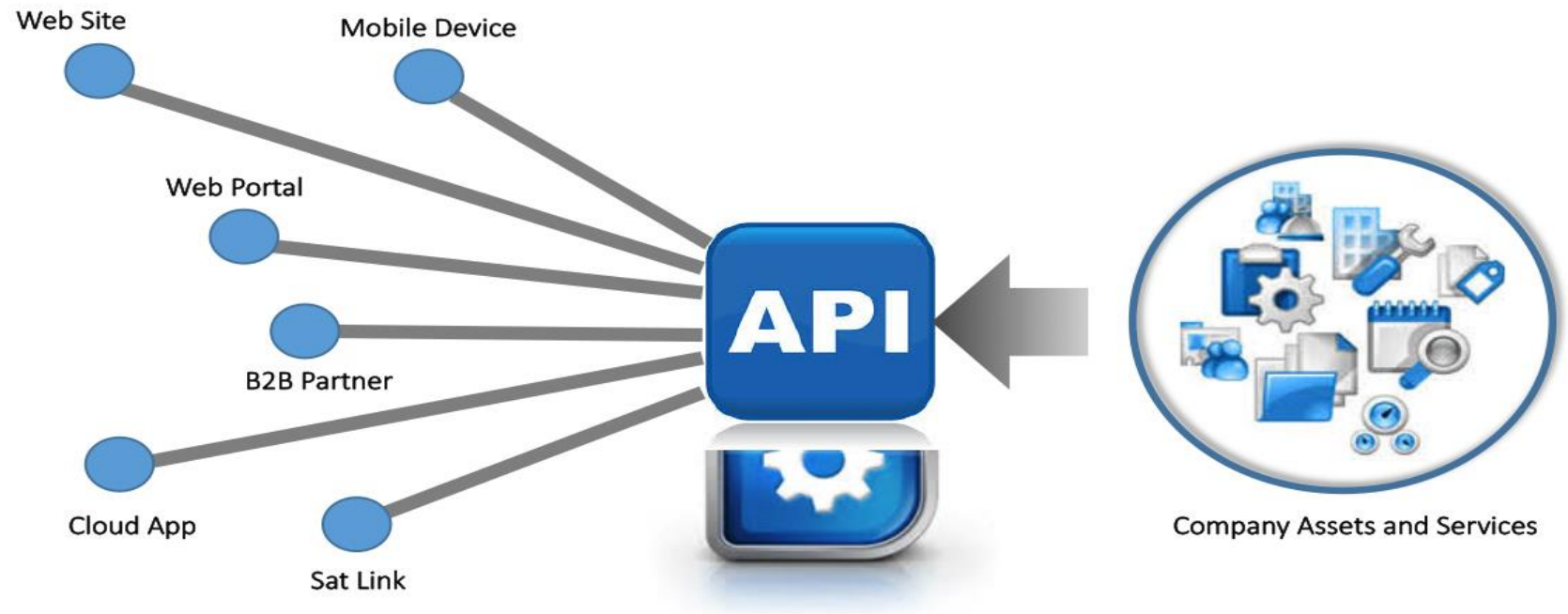
[Using JAX-RS & Spring Boot]

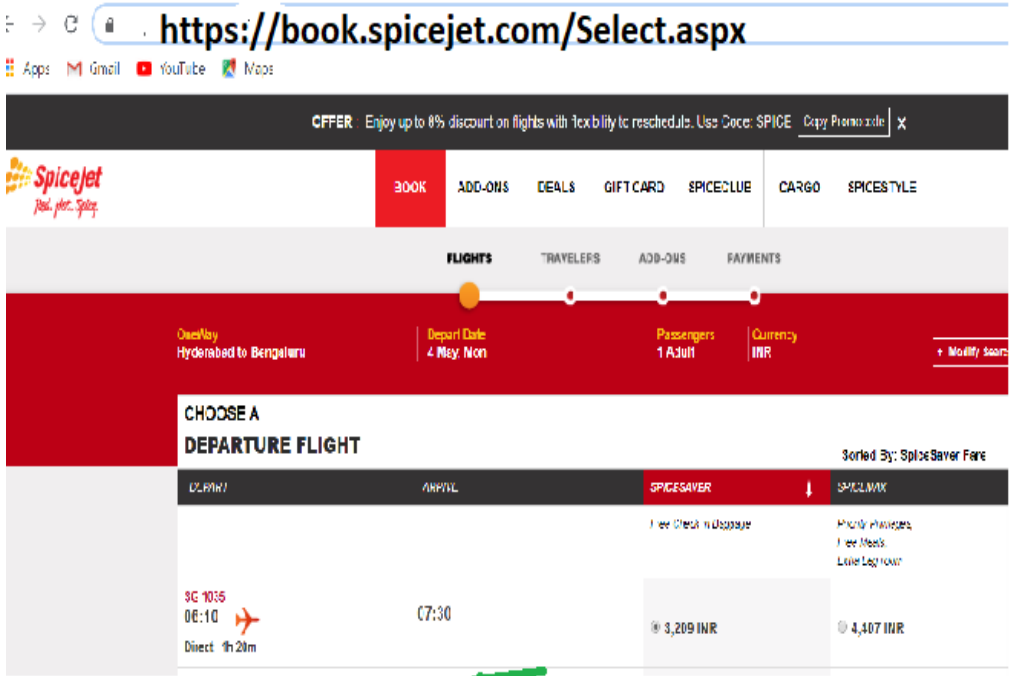
What is an API

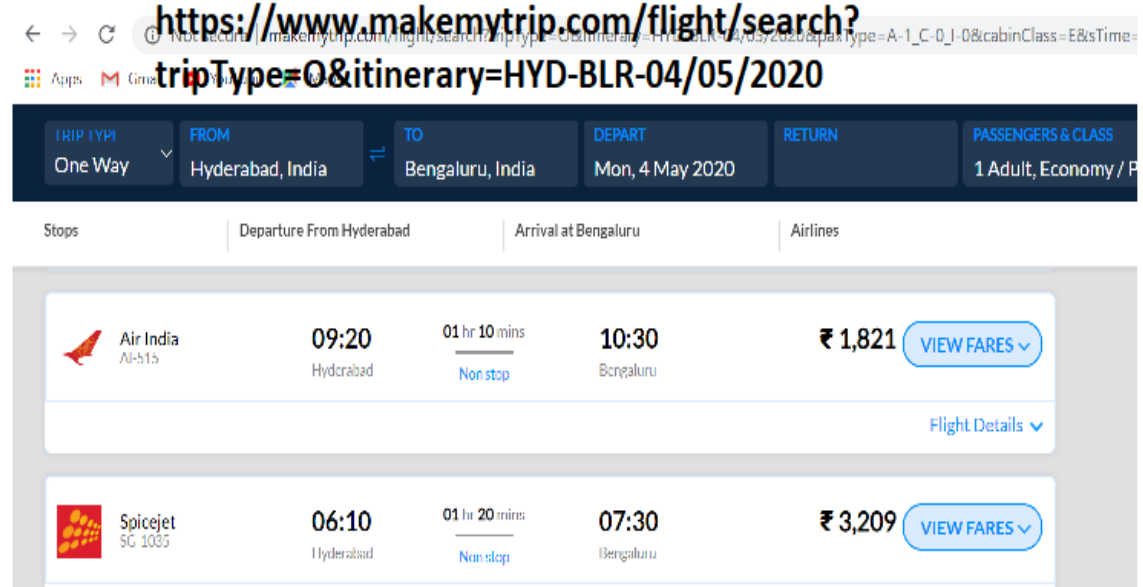
- An Application Programming Interface(API) is a set of definitions, protocols, and tools for building application software.

- An API make it easier for developers to use certain technologies in building applications by using certain predefined operations.

"API plug-ins" simplify and shorten the development life-cycle, making a developer's role more agile.







HTTP Resoponse (html)

Web MVC

API

Data (JSON/XML)

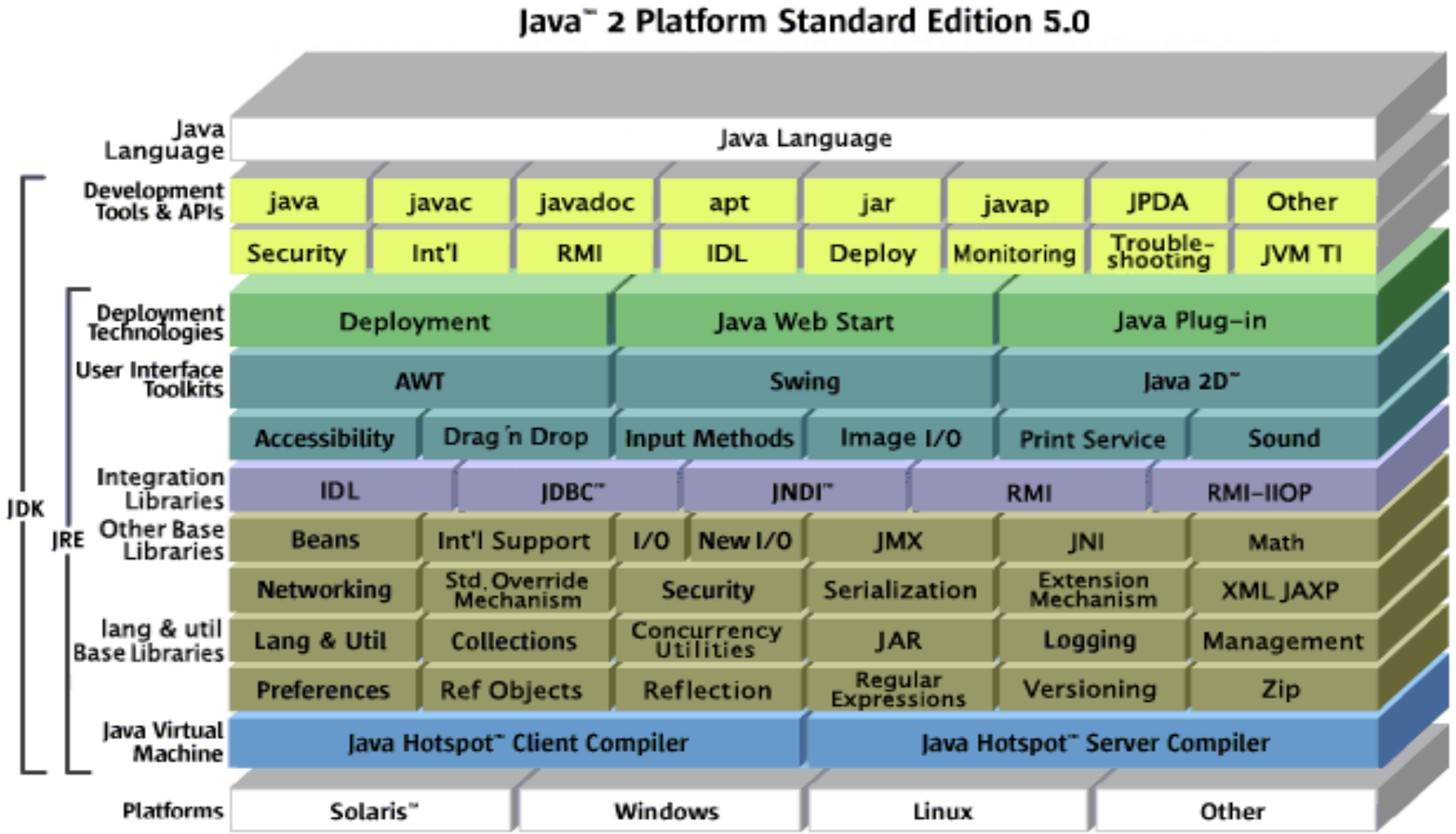
SpiceJet Flight Services

API in Java

- An API in java, is a collection of prewritten packages, classes, and interfaces with their respective methods, fields and constructors.

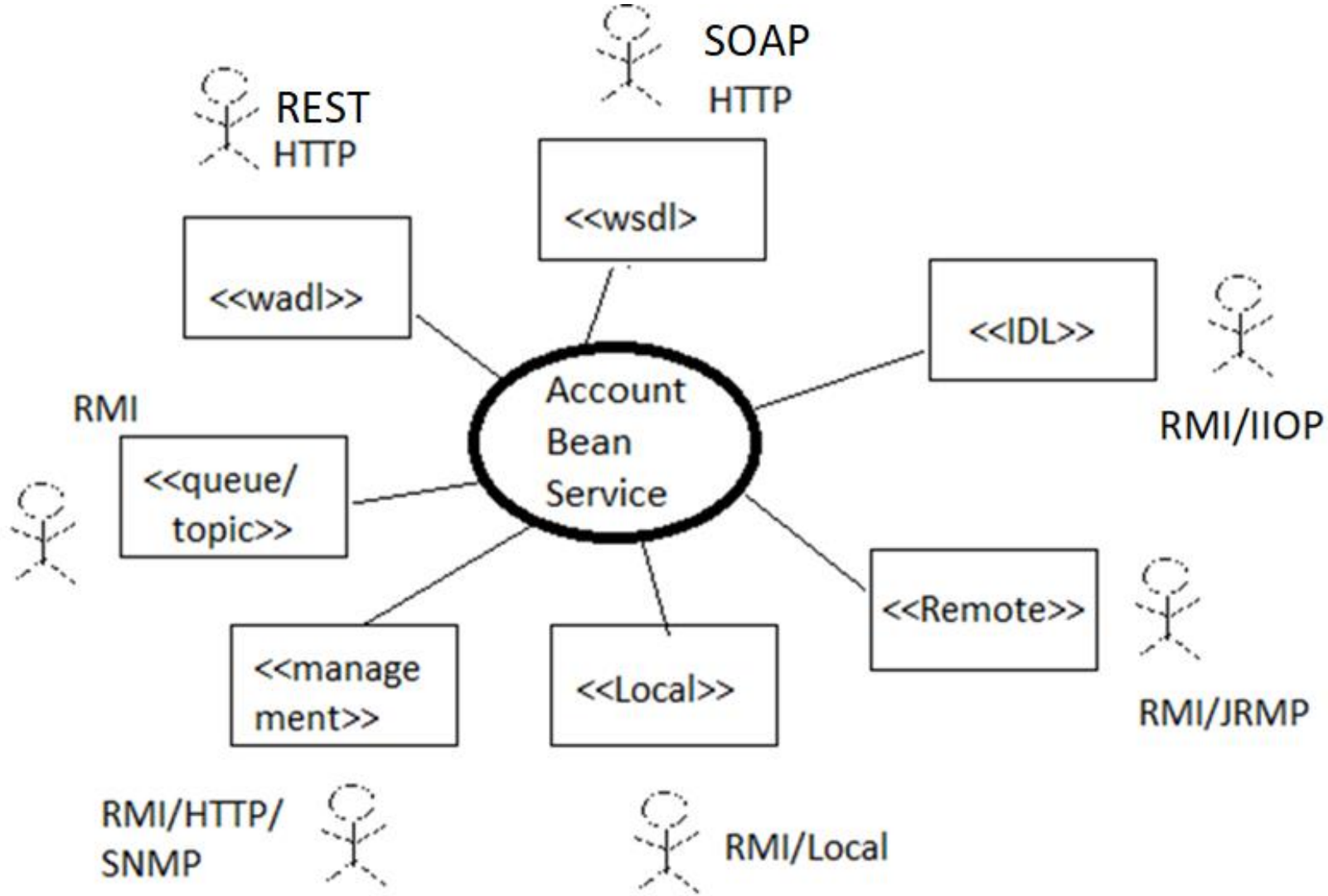
- In Java, there are over 4000+ API available for developers.

- Browser -> https://docs.oracle.com/javase/8/docs/api/


Java™ 2 Platform Standard Edition 5.0
Java Language
Java Language
Development Tools & APIs
java
javac
javadoc
apt
jar
javap
JPDA
Other
Security
Int'l
RMI
IDL
Deploy
Monitoring
Trouble-shooting
JVM TI
Deployment Technologies
Deployment
Java Web Start
Java Plug-in
User Interface Toolkits
AWT
Swing
Java 2D™
Accessibility
Drag 'n Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC™
JNDI™
RMI
RMI-IIOP
Other Base Libraries
Beans
Int'l Support
I/O
New I/O
JMX
JNI
Math
Networking
Std. Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang & util Base Libraries
Lang & Util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Java Virtual Machine
Java Hotspot™ Client Compiler
Java Hotspot™ Server Compiler
Platforms
Solaris™
Windows
Linux
Other
JDK
JRE

Distributed Environment

DISTRIBUTED

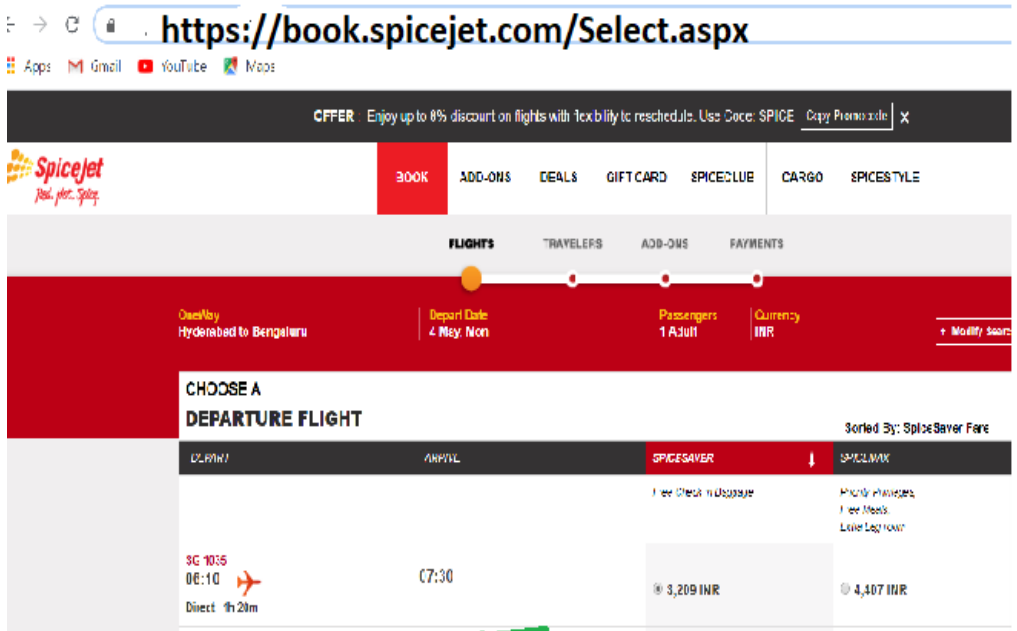


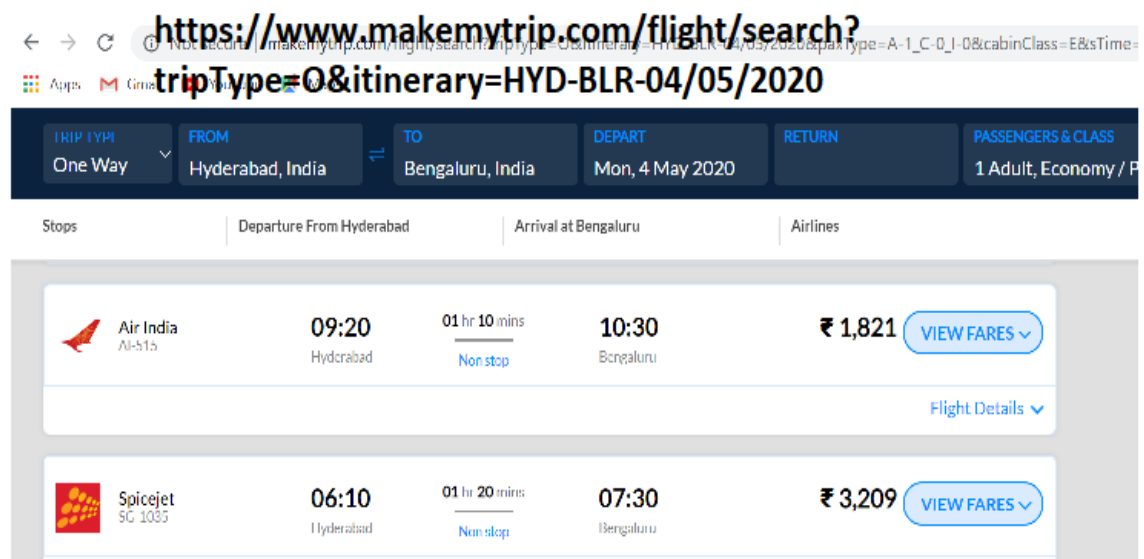
ENVIRONMENT

Rest API

- REpresentational State Transfer or REST is a web standards based architecture and uses HTTP protocol for data communication.

- HTTP methods like GET, PUT, DELETE, POST etc., are used in a REST based architecture





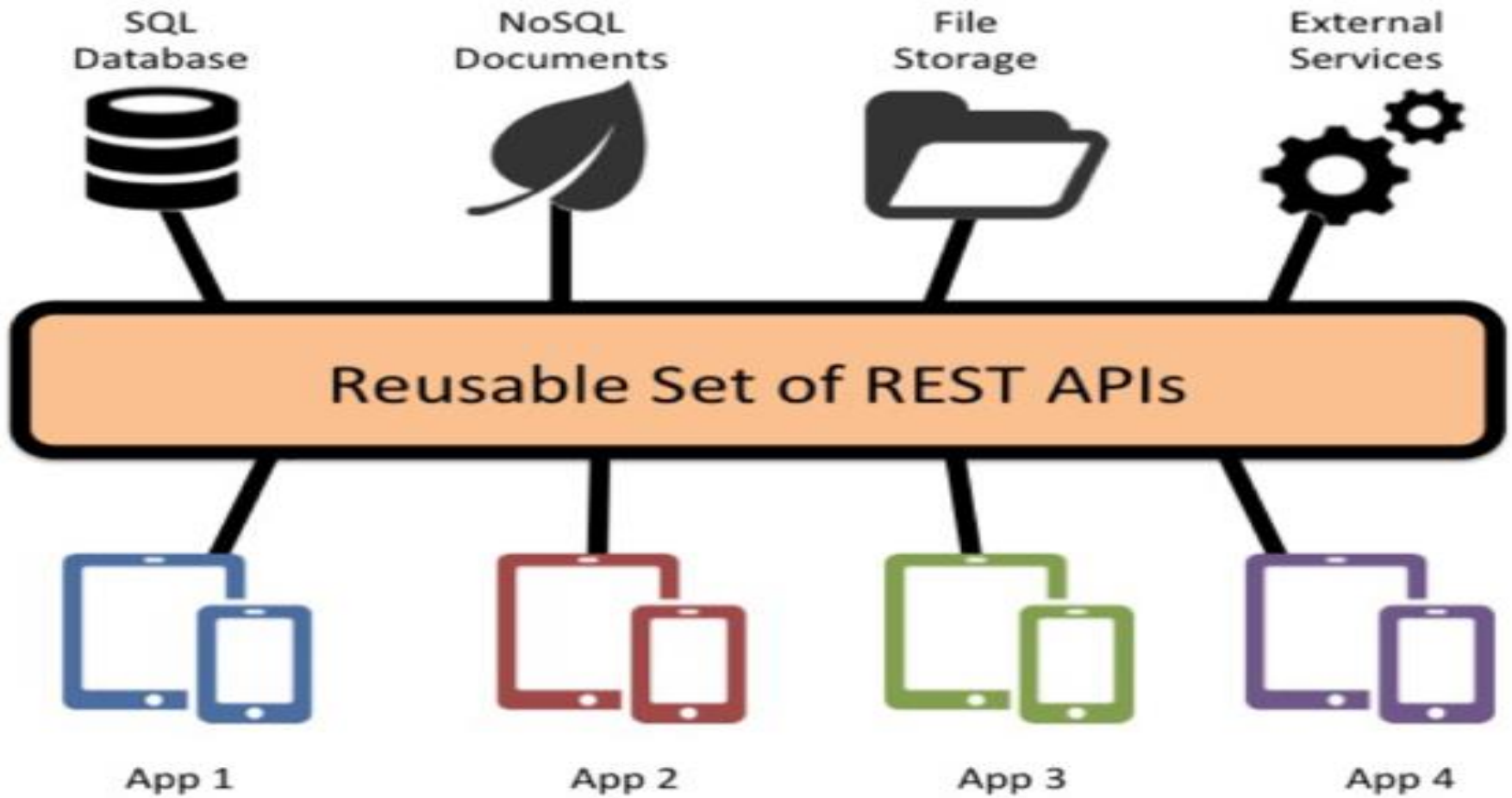
HTTP Resoponse (html)

Web MVC

REST API

Data (JSON/XML)

SpiceJet Flight Services


SQL
Database
NoSQL
Documents
File
Storage
External
Services
Reusable Set of REST APIs
App 1
App 2
App 3
App 4

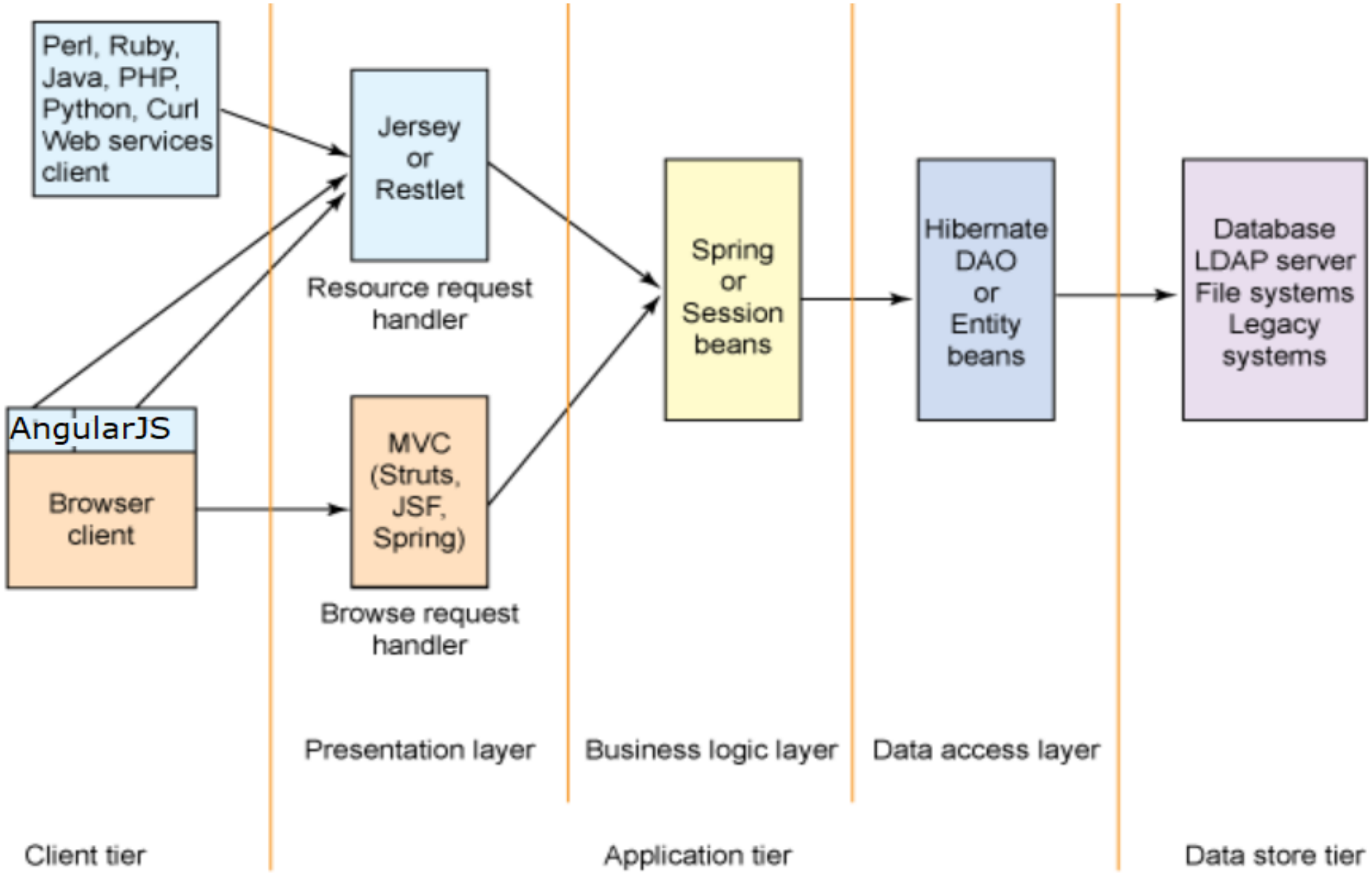
Web Application:

It is an end-to-end solution for a user.

Which means, User can:

- Open it using a browser Interact with it.
- User can click on something and after some processing, its result will be reflected in the browser screen. Human-System interaction.

Diagram of a multi-tiered Web application environment



Web API / Web service

With Web APIs alone, a user can not interact with it, because it only returns data, not views.

- ❑ It is a system which interacts with another system
- ❑ It does not return views, it returns data
- ❑ It has an endpoint set, which can be hit by other systems to get data which it provides.

- **Webservices** are services that are exposed over internet for programmatic access.

- Online API's, we can call from the code.

- Companies like Facebook, Twitter publish online API’s for the external applications (like games, YouTube etc.,) to post the messages.

- Online API’s can be built using different ways, like soap, rest etc.,

- REST is lightweight and uses HTTP protocol.

Twitter :

http://www.twitter.com [HTML]

http://api.twitter.com [XML / JSON]

Facebook:

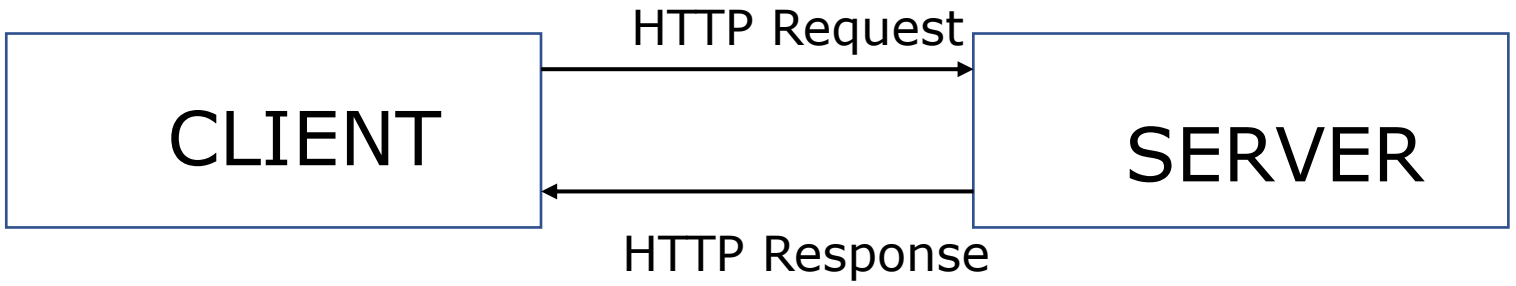
https://www.facebook.com [HTML]

https://graph.facebook.com [XML / JSON]

Web service Characteristics

(1). Exchange of data happens on Web over HTTP.

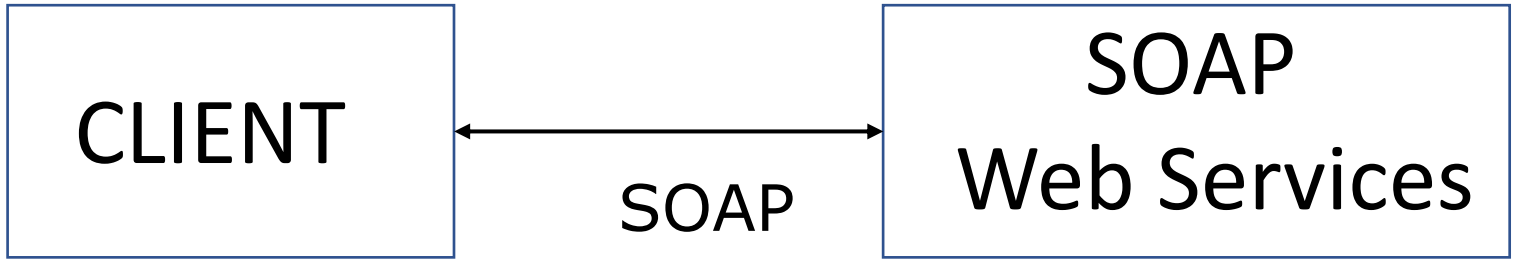
Clients sends an HTTP Request and servers returns back HTTP Response.



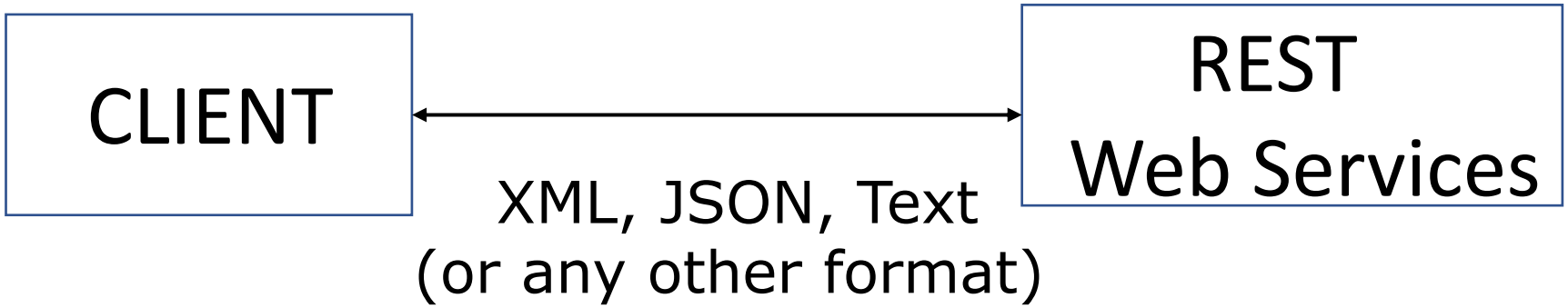
Client applications get the data and present the response in its own format.

(2). Protocol

The data exchanged between the Client and Server must follow some " Message Format", Also called as protocol.



* SOAP messages are in XML format with specific rules



* REST protocol is none. As long as client and Server understand the format, the data can be In any format.

(3). Service definition:

In java, to call a method we should know the method name, arguments and return value.

SOAP - Service Definition (WSDL)
Rules/ Specifications

Rest - No Service Definition / WADL
No Rules / Specifications;
It is a concept/idea

REpresentational State Transfer (REST)
(An Architectural Style ; Set of guidelines)

REST + Web Services = RESTFUL Web Services

Note : However, this architecture can be used for any type of services. It can be used over Tcp, websocket etc.,

REST can work over anything. HTTP is just a transport mechanism.

1. What is Representational State Transfer“ ?

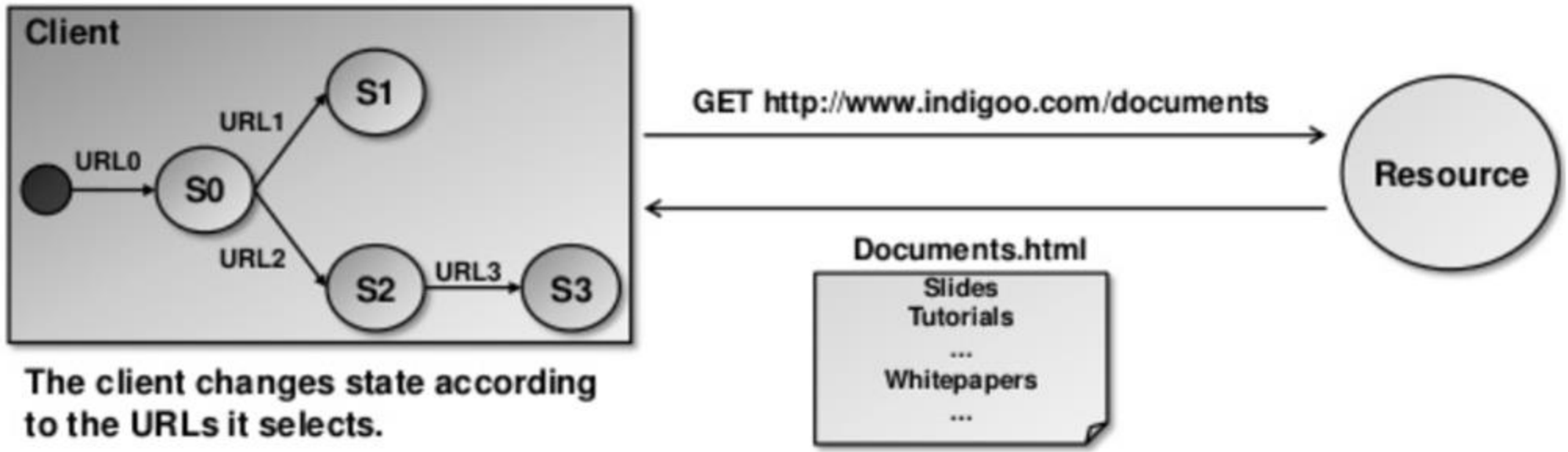
REST, unlike SOAP, is not a WS (web service) standard but an architectural style for web applications.

REST was devised by Roy Fielding in his doctoral dissertation:

> ***"Representation State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."***

- **REST is not a standard or protocol, REST is an architectural style.**
- **REST makes use of existing web standards (HTTP, URL, XML, JSON, MIME types).**
- **REST is resource oriented. Resources (pieces of information) are addressed by URIs and passed from server to client (or the other way round).**

To understand the REST principle, look at what happens in a web access of a browser:



The client changes state according to the URLs it selects.

1. The client references a web resource using a URL.
2. The web server returns a *representation* of the resource in the form of an HTML document.
3. This resource places the client into a new *state*.
4. The user clicks on a link in the resource (e.g. Documents.html) which results in another resource access.
5. The new resource places the client in a new state.

➔ The client application changes (=*transfers*) *state* with each resource *representation*.

REST is based on existing web (WWW, HTTP) principles and protocols:

Resources:
Application state and functionality are abstracted into resources (everything is a resource).

Addressability of resources:
Every resource is uniquely addressable using hyperlinks.

Uniform interface for accessing resources:
All resources share a uniform interface for the transfer of state between client and resource, consisting of

- a constrained (=limited) set of well-defined operations (GET, PUT, POST, DELETE).
- a constrained set of content types (text/html, text/xml etc.).

2. Why REST?

Scalability of WWW:
The WWW has proven to be:
a. scalable (growth)
b. simple (easy to implement, easy to use)

REST rationale:
If the web is good enough for humans, it is good enough for machine-to-machine (M2M) interaction.
The concepts behind RPC-WS (SOAP, XML-RPC) are different. RPC-WS make very little use of WWW-concepts and technologies. Such WS define an XML-based interface consisting of operations that run on top of HTTP or some other transport protocol. However, the features and capabilities of HTTP are not exploited.

The motivation for REST was to create an architectural model for web services that uses the same principles that made the WWW such a success.
The goal of REST is to achieve the same scalability and simplicity.

➔ REST uses proven concepts and technologies.
➔ REST keeps things as simple as possible.

3. Architectural constraints for REST

REST defines few architectural constraints that a system architecture should comply with to obtain scalability.

1. Client-server paradigm:

A client retrieves resources from a server or updates resources on a server.

- Separation of concerns such as presentation (client) from data storage (server).
- Portability (UI may be ported to different platforms).

2. Stateless:

A client request contains all information necessary for the server to understand the request.

- No need for storing context (state) on the server.
- Better scalability.

3. Cacheable:

Data (resources) need to be labeled as cacheable or non-cacheable.

- Improve network performance.

4. REST ‘protocol’

REST is not a protocol like SOAP.

But REST defines some core characteristics that make a system REST-ful.
N.B.: REST does not define something new, it simply makes use of existing protocols and standards (HTTP, URI).

Addressing resources:
REST uses plain *URIs* (actually URLs) to address and name resources.

Access to resources:
Unlike RPC-WS where the access method (CRUD) is mapped to and smeared over SOAP messages, REST uses the available HTTP methods as a resource interface:

Create (C)	→ HTTP POST
Read (R)	→ HTTP GET
Update (U)	→ HTTP PUT
Delete (D)	→ HTTP DELETE

REST assumes the methods GET, HEAD, PUT, DELETE to be idempotent (invoking the method multiple times on a specific resource has the same effect as invoking it once)

REST assumes the methods GET and HEAD to be safe (do not change the resource's state on the server, i.e. resource will not be modified or deleted)

Resource representations:

REST uses standard resource representations like HTML, XML, JSON, GIF, JPEG. Commonly used representations are XML and JSON (preferable to XML if the data needs to be transferred in a more compact and readable form).

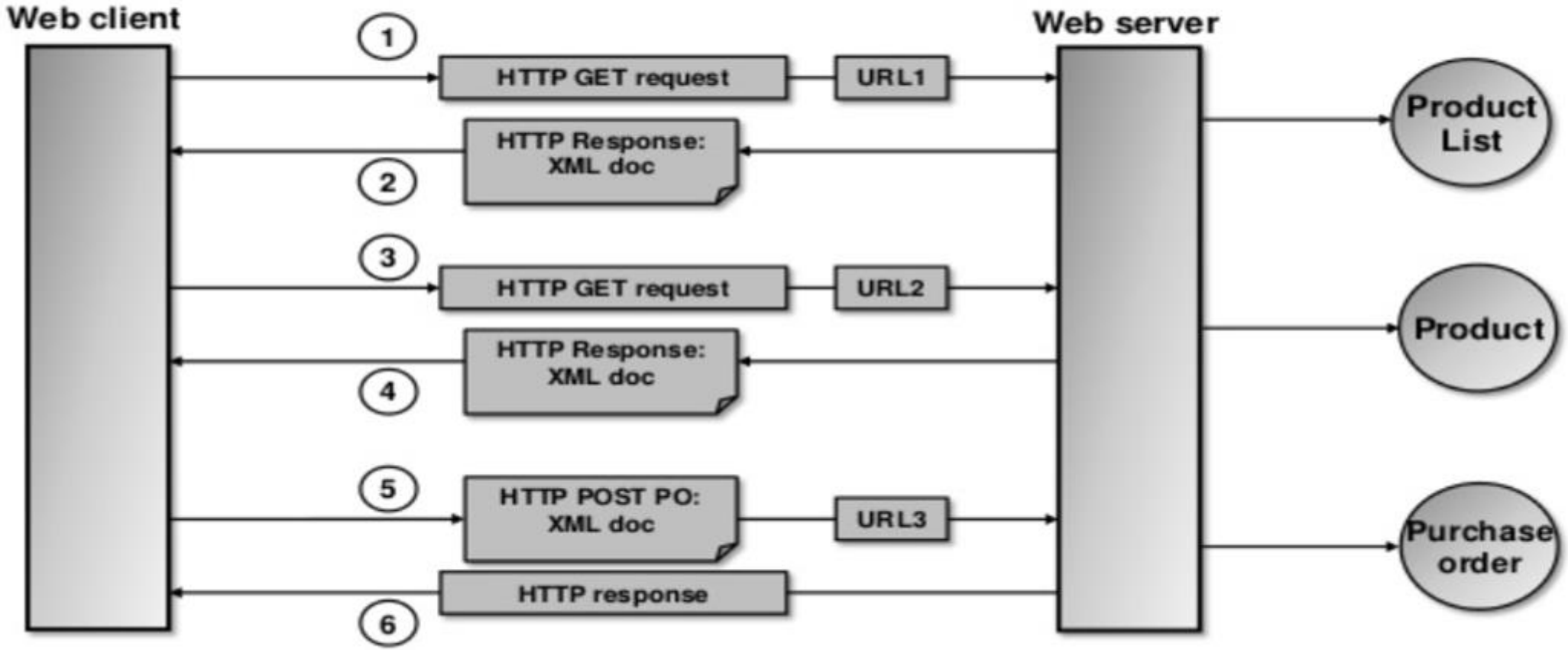
Media types:

REST uses the HTTP header *Content-type* (MIME types like text/html, text/plain, text/xml, text/javascript for JSON etc.) to indicate the encoding of the resource.

State:

Application state is to be maintained on the client. The server does not have to maintain a state variable for each client (this improves scalability).
Resource state (resource creation, update, deletion), however, is maintained on the server.

Example of a REST-ful access (1/3):



Example of a REST-ful access (2/3):

1. & 2. product list request:
The client requests a product list that is available under the URL *http://www.cool-products.com/products&flavor=xml* (URL1).
The response contains the resource encoded in XML.

3. & 4. product selection:
The client application (or the user in front of the browser) selects product 00345 by placing a request for the URL *http://www.cool-products.com/products/00345&flavor=xml* (URL2).
The response contains an XML representation of the product info for product 00345.

5. & 6. placing a purchase order:
The client application places a purchase order (PO) for product 00345 by requesting the resource under the URL *http://www.cool-products.com/products/00345/PO?quantity=7* (URL3).
The purchase order contains additional information entered on the client (customer name etc.).
Therefore the request is a POST accompanied by the XML representation of the purchase order.

Example of a REST-ful access (3/3):

Products list:

```
<?xml version="1.0"?>
<p:Products xmlns:p="http://www.cool-products.com"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
                        "http://www.cool-products.com
                         http://www.cool-products.com/products.xsd">
    <Product id="00345" xlink:href="http://www.cool-products.com/products/00345"/>
    <Product id="00346" xlink:href="http://www.cool-products.com/products/00346"/>
    <Product id="00347" xlink:href="http://www.cool-products.com/products/00347"/>
    <Product id="00348" xlink:href="http://www.cool-products.com/products/00348"/>
</p:Products>
```

Product list contains links to get detailed information about each product (e.g. using XLink). This is a core feature of REST.

Product 00345 info:

```
<?xml version="1.0"?>
<p:Product xmlns:p="http://www.cool-products.com"
           xmlns:xlink="http://www.w3.org/1999/xlink"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation=
                       "http://www.cool-products.com
                        http://www.cool-products.com/product.xsd">
    <Product-ID>00345</Product-ID>
    <Name>Widget-A</Name>
    <Description>This product is made of disposable materials</Description>
    <Specification xlink:href="http://www.cool-products.com/products/00345/specification"/>
    <PurchaseOrder xlink:href=http://www.cool-products.com/products/00345/po/>
    <UnitCost currency="CHF">1.40</UnitCost>
    <Quantity>10</Quantity>
</p:Product>
```

Another URL is provided in the product XML response for placing purchase orders.

HTTP (HyperText Transfer Protocol)

Hypertext is text displayed on a computer display or other electronic devices with references to other text that the reader can immediately access.

Hypertext documents are interconnected by Hyperlinks, which are typically activated by a mouse click, keypress set or by touching the screen.

To write the Hypertext we use html.

HTTP concepts are inspired by REST.

URLs, URIs, and URNs

- URL is a Uniform Resource Locator, tells you the how and where of something
  - [Scheme]://[Domain]:[Port]/[Path]?[QueryString]#[FragmentId]
  - http://www.wrox.com/remtitle.cgi?isgn=0470114878
- URN is a Uniform Resource Name, is simply a unique name
  - urn:[namespace identifier]:[namespace specific string]
  - urn:isbn:9780470114872
- URI is a Uniform Resource Identifier, is URL or URN

Addresses:

Website based URI: [Action based]
weatherapp.com/weatherLookup.do?zipcode=12345

Resource based URI: [Resource is already there, we are retrieving]
weatherapp.com/zipcode/12345

Ex: we want to retrieve weather based on country and country name
Weatherapp.com/countries/india

Web Application URIs (Action-based URI's)

To get an item with ID 10

/getItems.do?id=10 (OR) /retreiveItems.action?id=10

Note :

In a web application, URI are not very important. Because from main page, we navigate to other pages.

REST API, consumers have to be aware of URIs. We need to provide common and simple URI.

In REST, everything is a resource :

Examples of Resources with URI:

www.myapp.com/image/logo.gif (Image Resource)

www.myapp.com/Account/1001 - (Dynamic resource)

www.myapp.com/videos/v001 (Video Resource)

www.myapp.com/home.html (Static Resource)

Examples:

```
Action-based URL                          Resource-based URL
------------------------------------------------------------------
GET  /register                            GET    /accounts/application
POST /register                            POST   /accounts
GET  /catalog/search                      GET    /queries/form
POST /catalog/search                      POST   /queries
GET  /cart                                GET    /order/123
POST /cart/add-item                       POST   /order/123/items
POST /cart/empty                          DELETE /order/123
GET  /check-out                           GET    /order/123/invoice
POST /check-out                           POST   /order/123/payments
GET  /thank-you                           GET    /order/123/receipt
POST /sign-in                             POST   /sessions
POST /admin/delete-user?id=123            DELETE /user/123
GET  /catalog/edit?id=123                 GET    /items/123/form
POST /catalog/edit?id=123                 PUT    /items/123
```

What is difference between action based and resource base URI ?

Action-based URIs:

- ❑ Focus on the action being performed
- ❑ Usually include a verb
- ❑ Often rely on external sources to identify the resource being acted on (e.g., session state)

Resource-based URIs:

- ❑ Focus on the resource being acted on
- ❑ Usually consist of nouns
- ❑ Rely on HTTP verbs to define the action being performed (e.g., GET, PUT, POST, and DELETE)

Static Profile Pages : (Web application)

/profiles/ramana.html

Resource Based URI: (REST web service)

/profiles/<profileName>
Ex: /profiles/ramana

Note : REST is “Nouns not verbs”.

We have messages, accounts, payments etc., and not getMessages, FetchMessages.

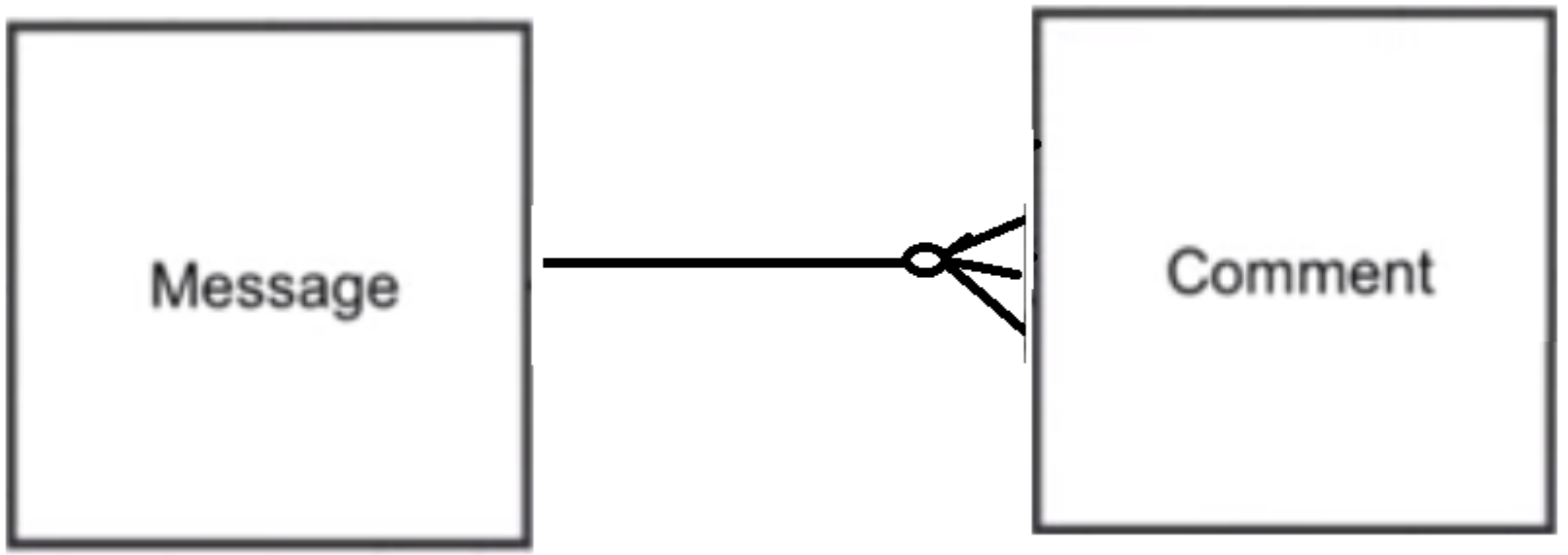
We have just messages. We have to identify the things, entity (Noun) .

Resource Base URI

/profiles/{profileName}
/messages/{messageID}
/comments/{commentID}

- They are not depending on the frameworks (in the case of structs we add .do)

Resource Relation



A REST API defines a set of functions which developers can perform requests and receive responses via HTTP protocol such as GET and POST.

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.

It is important to note that REST is a style of software architecture as opposed to a set of standards.

As a result, such applications or architectures are sometimes referred to as RESTful or REST-style applications or architectures.

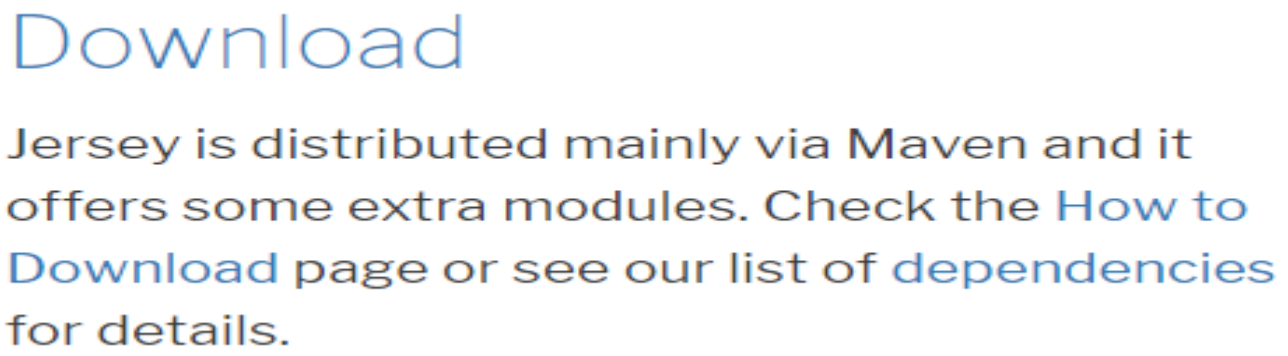
REST has proved to be a popular choice for implementing Web Services.

An application or architecture considered RESTful or REST-style is characterized by:

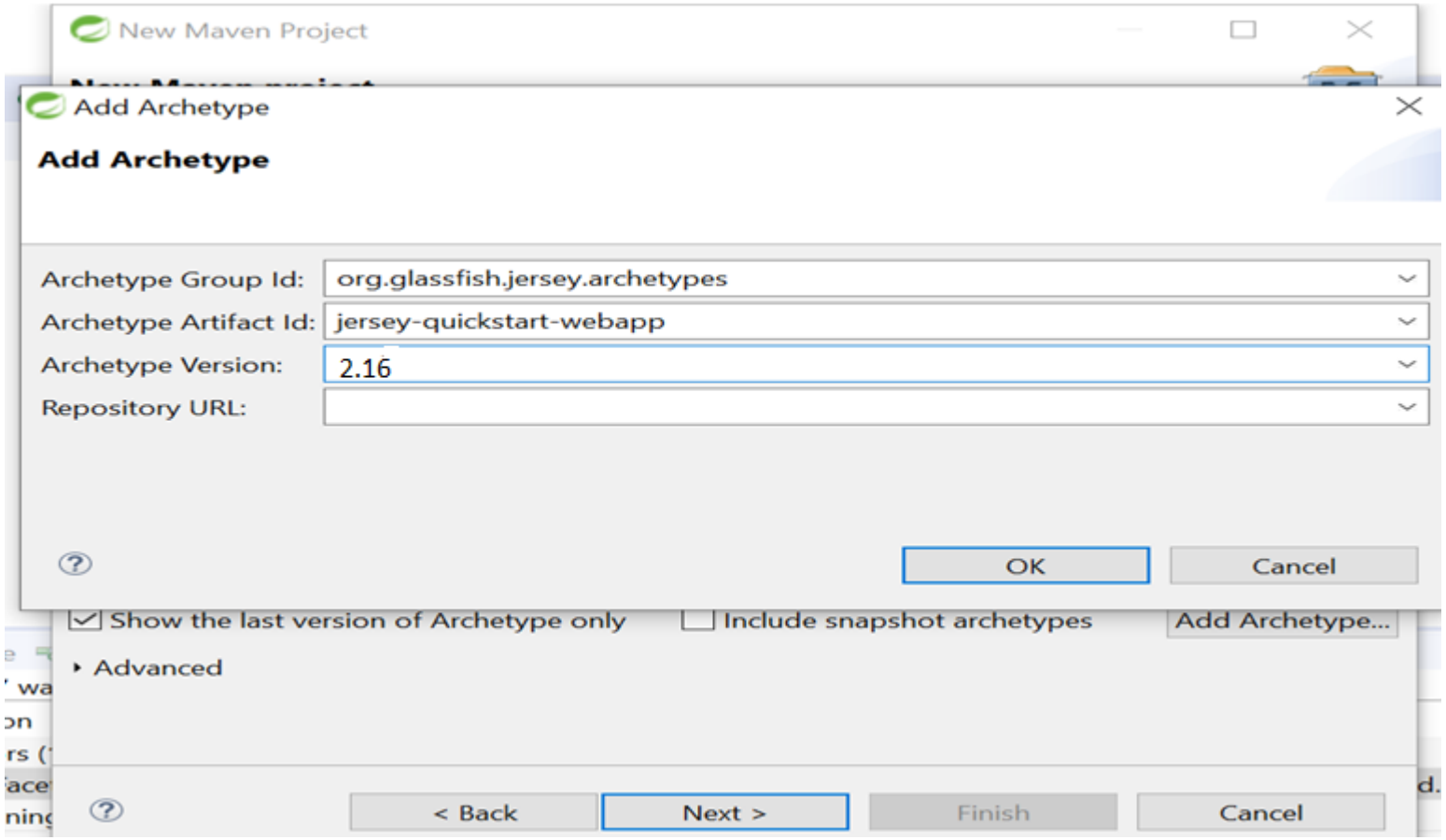
- State and functionality are divided into distributed resources
- Every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet)
- The protocol is client/server, stateless, layered, and supports caching

Setup

https://eclipse-ee4j.github.io/jersey/

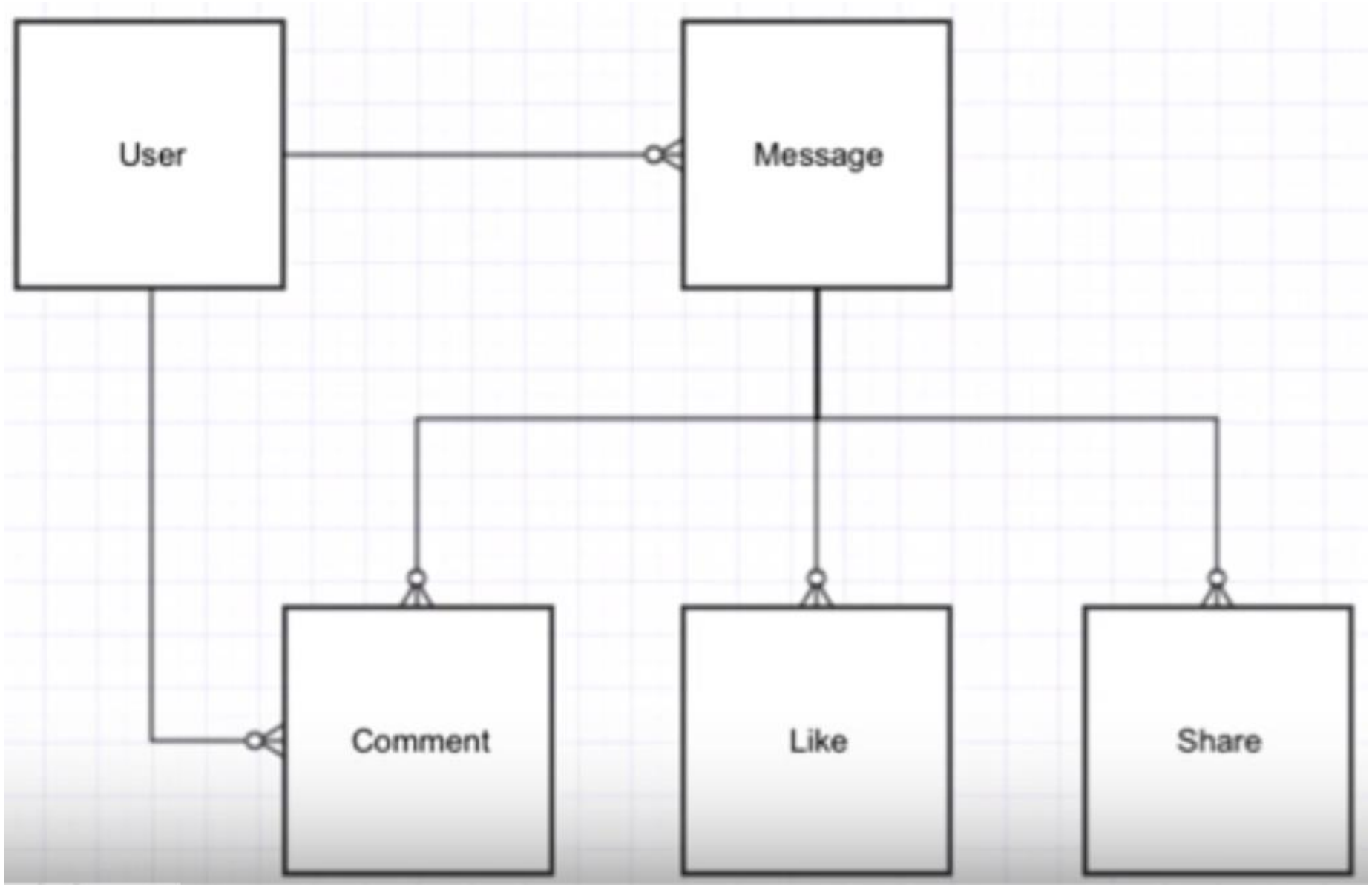


- Create a Maven Project
- Add Archetype for jersey
  Group Id: org.glassfish.jersey.archetypes
  Artifact Id: jersey-quickstart-webapp
  Version : 2.16
  Repository URL : not required



Messenger - A social media application API

- ❑ Post messages
- ❑ Comment on messages
- ❑ Like and share messages
- ❑ User profiles

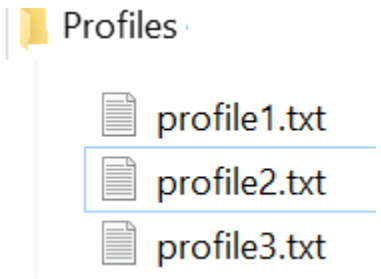

User
Message
Comment
Like
Share

Initially, when websites came; it was all static pages.

Each page use to have separate html (like index.html, product.html etc.,)

Resource URL's uses similar concept

Static profile pages:



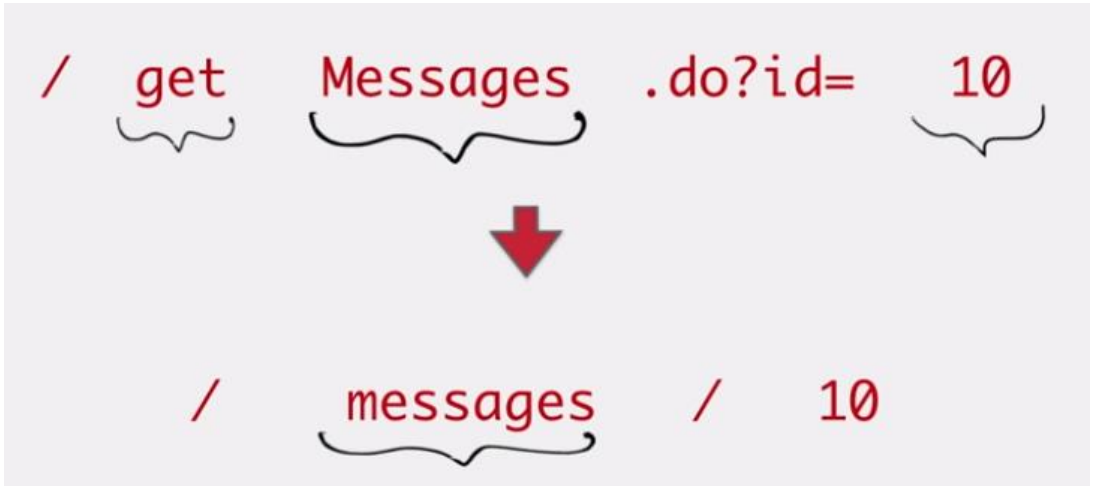
Resource based URI

/profiles/profile1.html
/profiles/profile2.html
/profiles/profile3.html

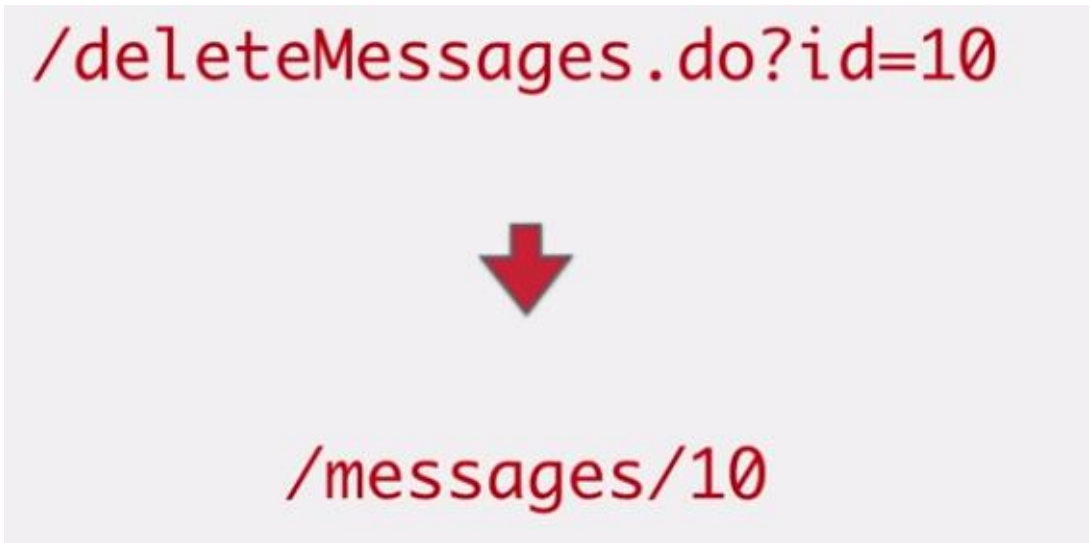
remove .html

/profiles/profile1
/profiles/profile2
/profiles/profile3

* Generic uri would be /profile/{profileName}


/ get Messages .do?id= 10
/ messages / 10


/postMessages.do?id=10
/messages/10


/deleteMessages.do?id=10
/messages/10

URI's are Nouns [not verbs]

messages
profiles
comments

[not]

getMessages
setProfiles
DelteComments

/messages/{messageId}

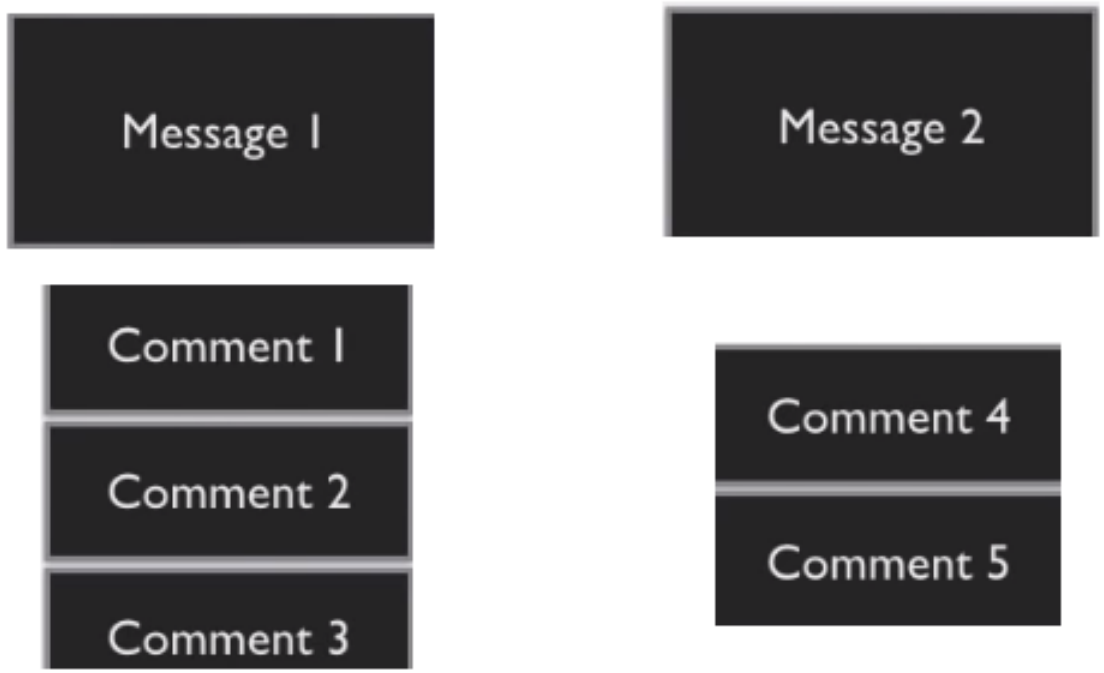
[not]

/message/{messageId}

Note :
These Resource URI’s are not depending on the frameworks (.do / .controller)
These Resource URI’s would be same, even if frameworks are changed
Consumer really don’t care about which framework was used to develop the resources

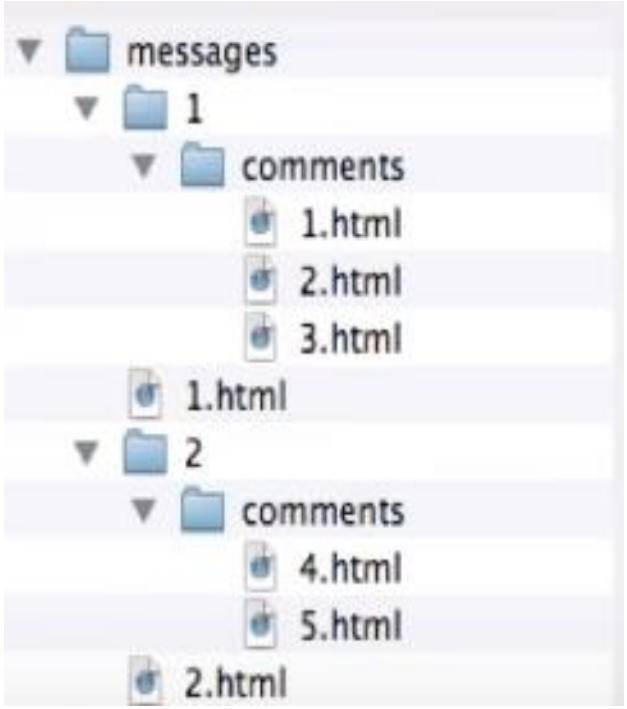
Resource Relations [Resources depend on other resources]

Resource Relations



To retrieve comment id 10, the below URI is used /comments/10

But, comments will have a relation with messages



/messages/1/comments/2

i.e.,

/messages/{messageId}/comments/{commentID}

Note : This URI makes clear that comment belong to particular message Id

Similar URI’s are:

/messages/{messageId}/comments/{commentId}
/messages/{messageId}/likes/{likeId}
/messages/{messageId}/shares/{shareId}
/profiles/{profileId}/messages/{messageId}

* This is one-to-many relationship

There are two types URIs

- ❑ Instance resource URI’s
- ❑ Collection resource URI’s

Instance Resource URIs

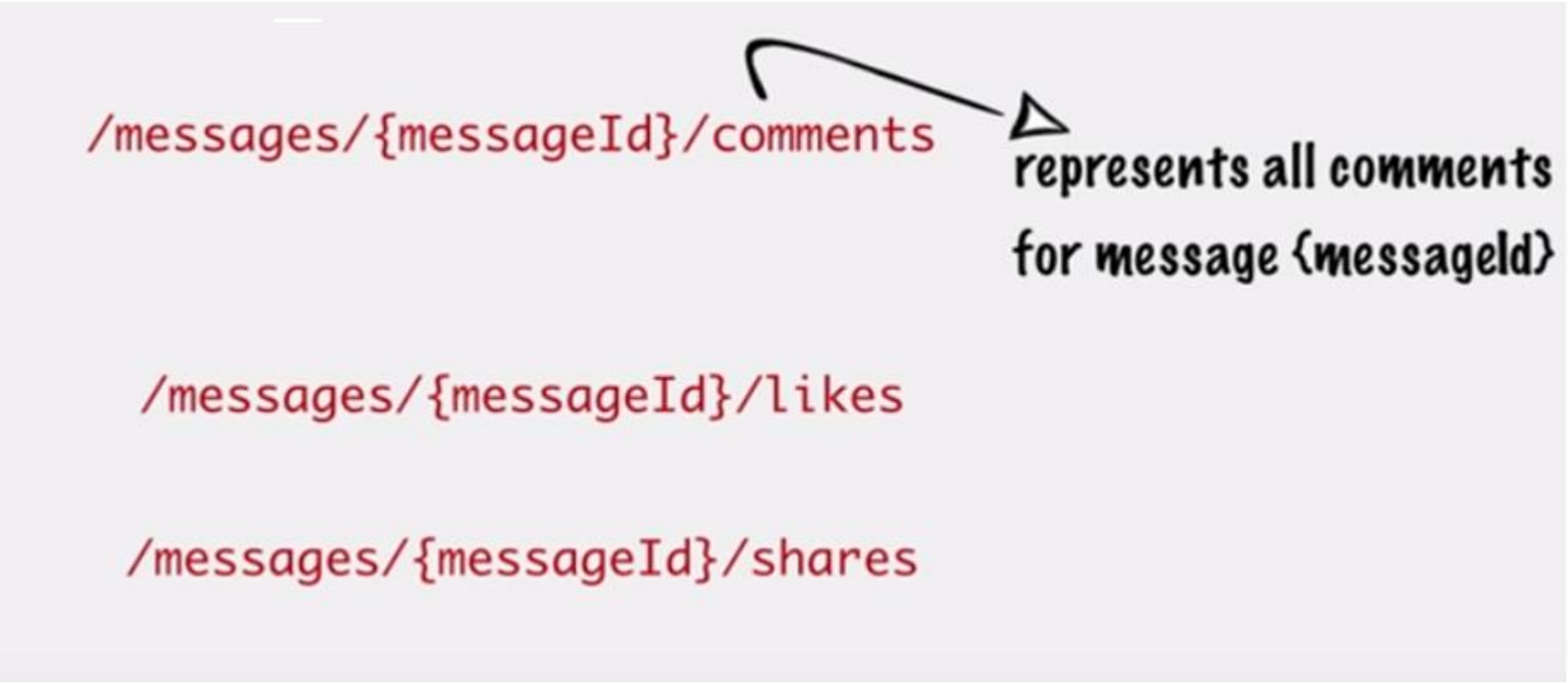
first level:

/profiles/{profileName}
/messages/{messageId}

second level:

/messages/{messageId}/comments/{CommentId}
/messages/{messageId}/likes/{likeId}
/messages/{messageId}/shares/{shareId}
/profiles/{profileName}/messages/{messageId}

Collection Resource URIs



How to get all comments

/messages/{messageId}/comments [will not work]

/comments [would work]

But; using /comments, we can't get comments for a particular message

Note:
This is where, we have to make trade-off.
Normally, comments are retrieved per message id.

Custom filter:

/messages?year=2014

- However, this can be used with pagination

  /messages? year=2014 &offset=30&limit=10

  [offset is ‘starting point’ and limit is ‘page size’]

Note : to get these values in our service, we use @QueryParam

HTTP Methods

GET: Retrieve data from a specified resource

POST: Submit data to be processed to a specified resource

PUT: Update a specified resource

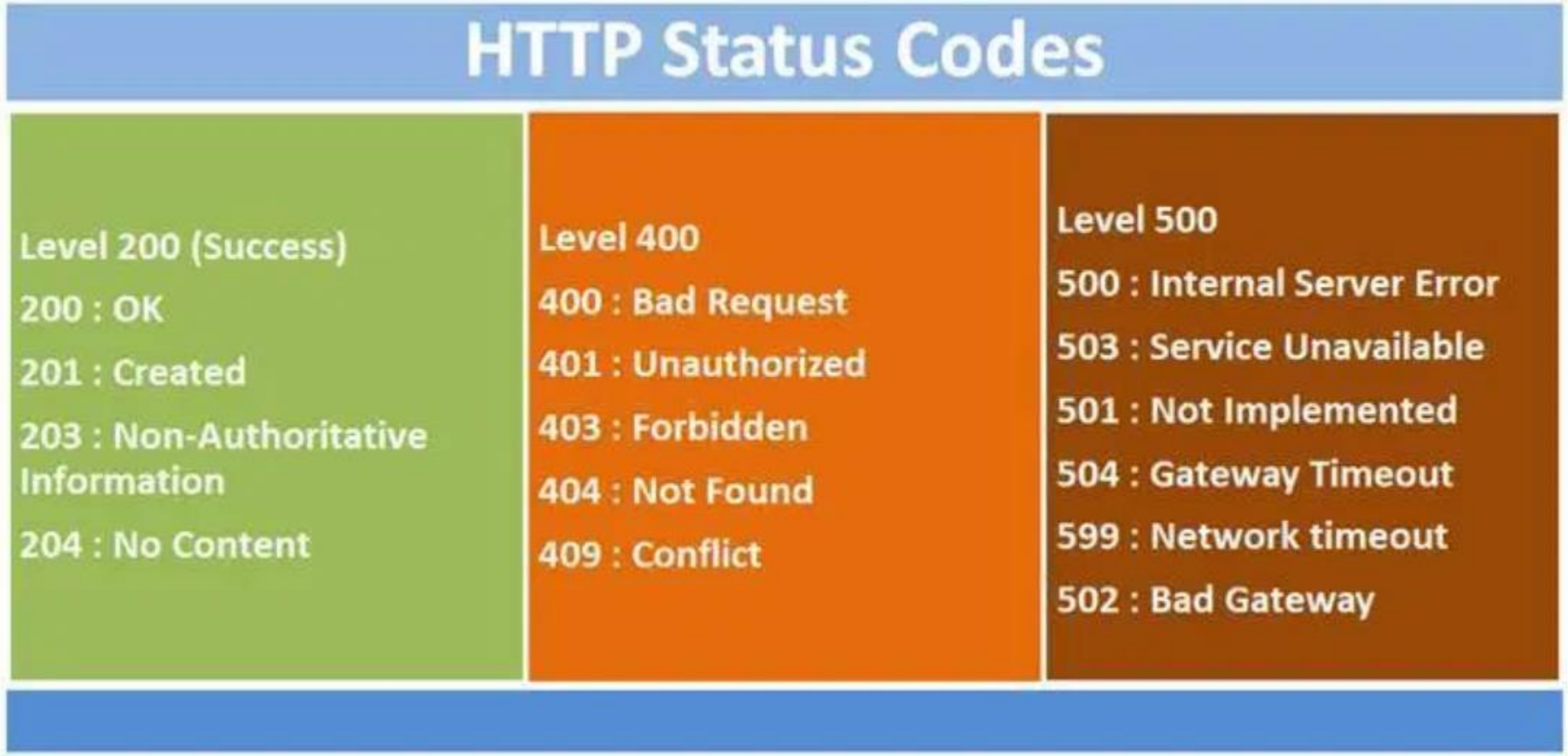
DELETE: Delete a specified resource

HEAD: Same as get but does not return a body

OPTIONS: Returns the supported HTTP methods

PATCH: Update partial resources

Http response contain metadata. Http Status code is one among them.



Http Status Code

- In the case of html page, based on the status code; appropriate message is sent to client.

- But, in REST; client is another piece of code.

Message Headers: [Accept & Content-Type]

Accept and Content-type are both headers sent from a client(browser say) to a service.

Accept header is a way for a client to specify the media type of the response content it is expecting.

Content-type is a way to specify the media type of request being sent from the client to the server.

Content types are : text/xml, application/json

Note : It is also called Context Negotiation.

To design a REST API's, we need to have the below things:

- ❑ Resource based URIs
- ❑ Http methods
- ❑ Http status codes
- ❑ Message headers (content-type)

Resource Based URI

The URI/URL where api/service can be accessed by a client application

GET https://online.trainings.com/api/users

GET https://online.trainings.com/api/users/1 (OR)
https://online.trainings.com/api/users/details/1

POST https://online.trainings.com/api/users

PUT https://online.trainings.com/api/users/1 (OR)
https://online.trainings.com/api/users/update/1

DELETE https://online.trainings.com/api/users/1 (OR)
https://online.trainings.com/api/users/delete/1

HTTP Methods

- read-only method : GET
- write methods : PUT, POST,PUT, DELETE
- safely repeatable methods (Idempotent) : GET, PUT, DELETE
- can’t be repeated safely (non-Idempotent) : POST

IDEMPOTENCE

WHEN PERFORMING AN OPERATION AGAIN GIVES THE SAME RESULT

HTTP METHOD	IDEMPOTENCE	SAFETY
GET	YES	YES
HEAD	YES	YES
PUT	YES	NO
DELETE	YES	NO
POST	NO	NO
PATCH	NO	NO

Idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application. The concept of idempotence arises in a number of places in abstract algebra and functional programming. Wikipedia

PATCH request can be idempotent if we define the merging rules to be idempotent.

Idempotent example:

```
// Original resource
{
  name: 'rushi',
  age: 32
}

// PATCH request
{
  age: 33
}

// New resource
{
  name: 'rushi',
  age: 33
}
```

PATCH Non-idempotent example:

```
// Original resource
{
  name: 'rushi',
  age: 32
}

// PATCH request
{
  $increment: 'age'
}

// New resource
{
  name: 'rushi',
  age: 33
}
```

This is not idempotent, as sending the same request multiple times would result in different results each time.

Remember :

- We can cache the resources for GET request.
- Client has to safe guard, browser refresh button.
- Otherwise, if previous request was make using POST method; by clicking the refresh button; duplicate record will be inserted. Because POST is not idempotent method.

Specification, Framework & Pattern

Specification

Provides API , standards, recommended practices, codes and technical publications, reports and studies.

JCP - Java Community Process
JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)
The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)
The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)
Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 340: Java Servlet 3.1 Specification

JSR 318: EJB 3.1

JSR 250: Common Annotations for java

JSR 303: Java Bean Validations

JSR 224: JAX-WS 2.0

JSR 370: JAX-RS 2.1

JSR 299: Context & DI

JSR 330: DI

GOF Design Patterns

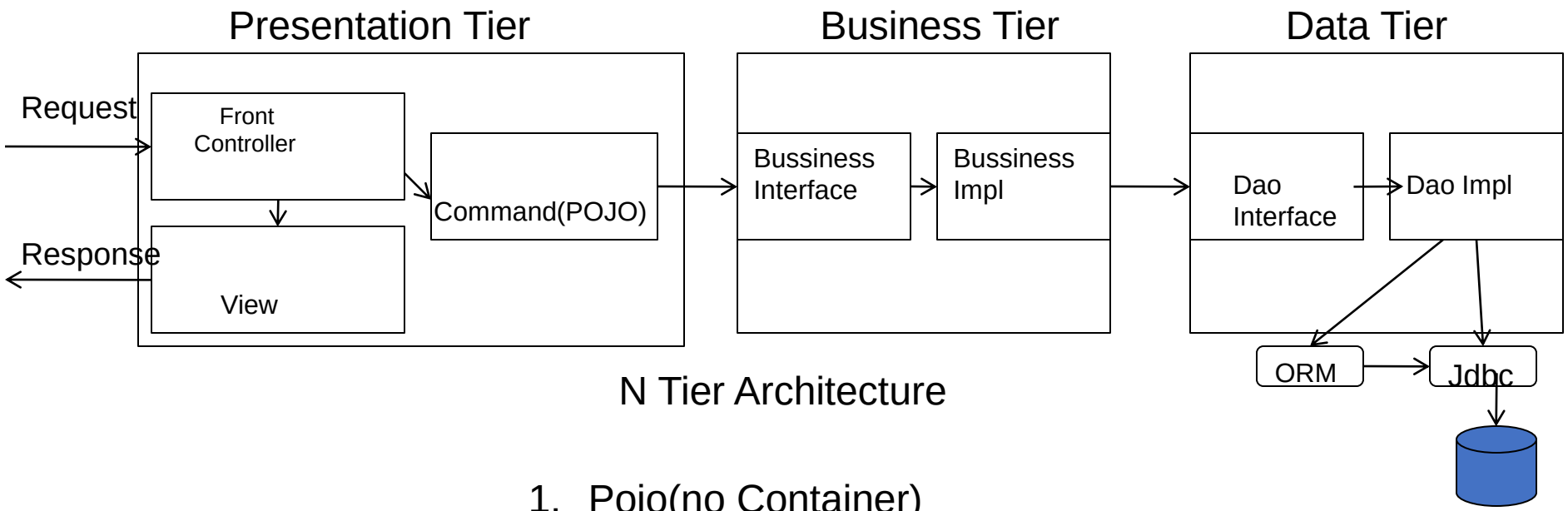
		Purpose		
		Creational	Structural	Behavioral
SCOPE	Class	Factory Method	Class Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

RESTful API Patterns

- ❑ Statelessness
- ❑ Content Negotiation
- ❑ URI Templates
- ❑ Pagination
- ❑ Versioning
- ❑ Authorization
- ❑ API facade
- ❑ Discoverability
- ❑ Idempotent
- ❑ Circuit breaker

Microservice Patterns:

- ❑ API gateway
- ❑ Service registry
- ❑ Circuit breaker
- ❑ Messaging
- ❑ Database per Service
- ❑ Access Token
- ❑ Saga
- ❑ Event Sourcing & CQRS



N Tier Architecture

1. Servlet/jsp
2. MVC
   Struts
   JSF
   Flex
   Gwt
   Spring MVC
   ...

> Any MVC + Spring

1. Pojo(no Container)
2. Ejb 2.x(HW Container)
   -Session Bean
   -Mdb
3. Pojo + LW Container
   - Spring
   - Microcontainer
   - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
   - Hibernate
   - Kodo
   - Toplink
   - MyBatis
5. JPA

\+ Spring Templates

JAX-RS Vs Spring REST

JSR 370: Java™ API for RESTful Web Services (JAX-RS 2.1) Specification

JAX-RS 2.1

JAX-RS is a specification for implementing REST web services in Java, currently defined by the JSR-370.

Jersey (shipped with GlassFish and Payara) is the JAX-RS reference implementation, however there are other implementations such as RESTEasy (shipped with JBoss EAP and WildFly) and Apache CXF (shipped with TomEE and WebSphere).

Spring Framework

The Spring Framework is a full framework that allows us to create Java enterprise applications.

The REST capabilities are provided by the Spring MVC module (same module that provides model-view-controller capabilities).

It is not a JAX-RS implementation and can be seen as a Spring alternative to the JAX-RS standard.

The Spring ecosystem also provides a wide range of projects for creating enterprise applications, covering persistence, security, integration with social networks, batch processing, etc.

IOC is used to decouple common task from implementation.

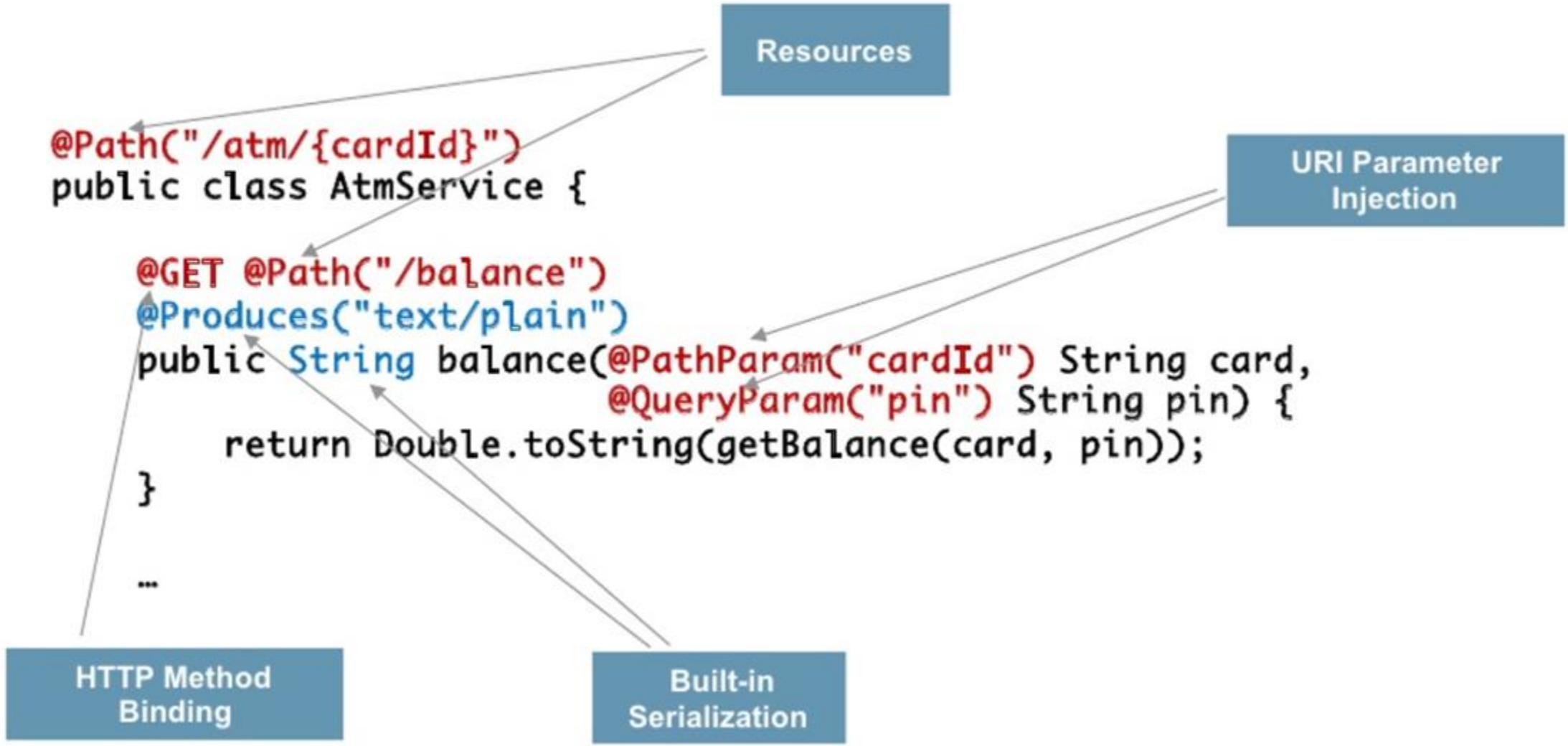
Six basic techniques to implement Inversion of Control.

These are:

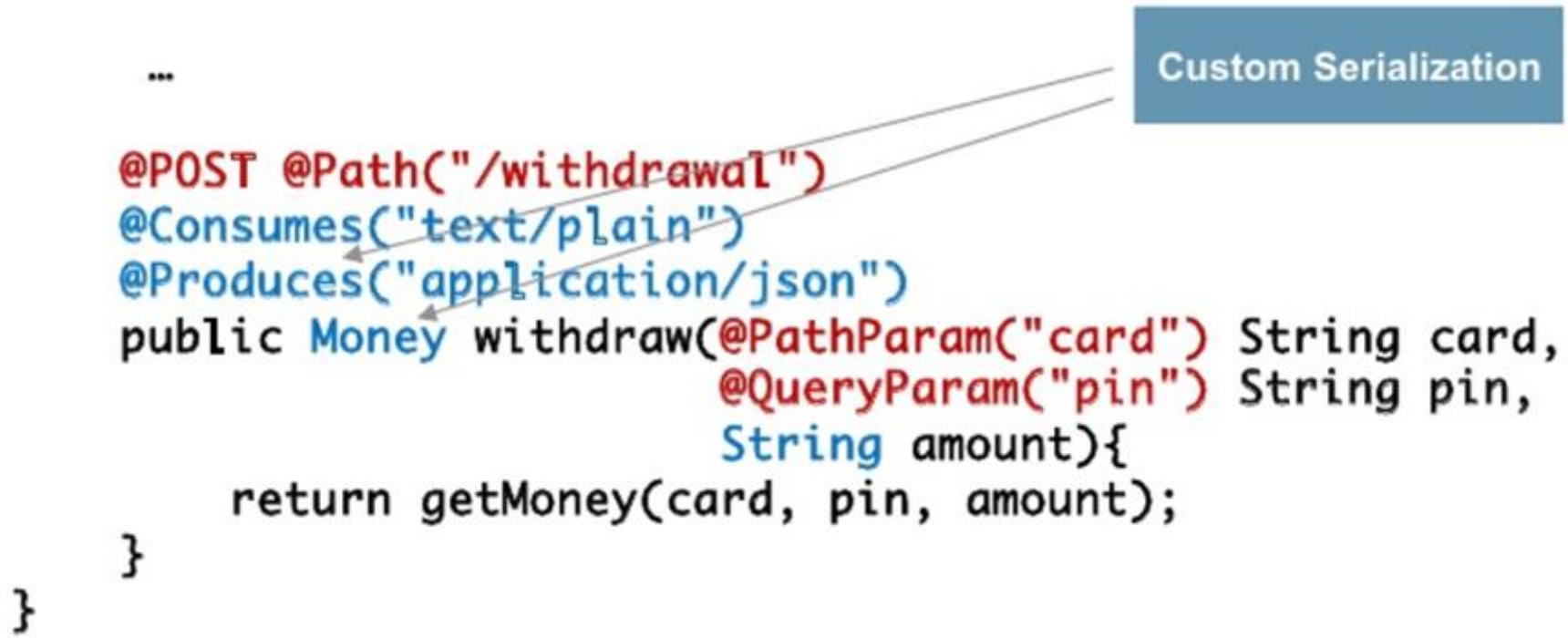
1. using a factory pattern
2. using a service locator pattern
3. using a constructor injection
4. using a setter injection
5. using an interface injection
6. using a contextualized lookup

Constructor, setter, and interface injection are all aspects of Dependency injection.

Example: JAX-RS API



Example: JAX-RS API (contd.)



JAX-RS API Annotations

1. *@javax.ws.rs.GET*
2. *@javax.ws.rs.POST*
3. *@javax.ws.rs.PUT*
4. *@javax.ws.rs.DELETE*
5. *@javax.ws.rs.HEAD*
6. *@javax.ws.rs.OPTIONS*

`Other basic Annotations provided by JAX-RS API:`

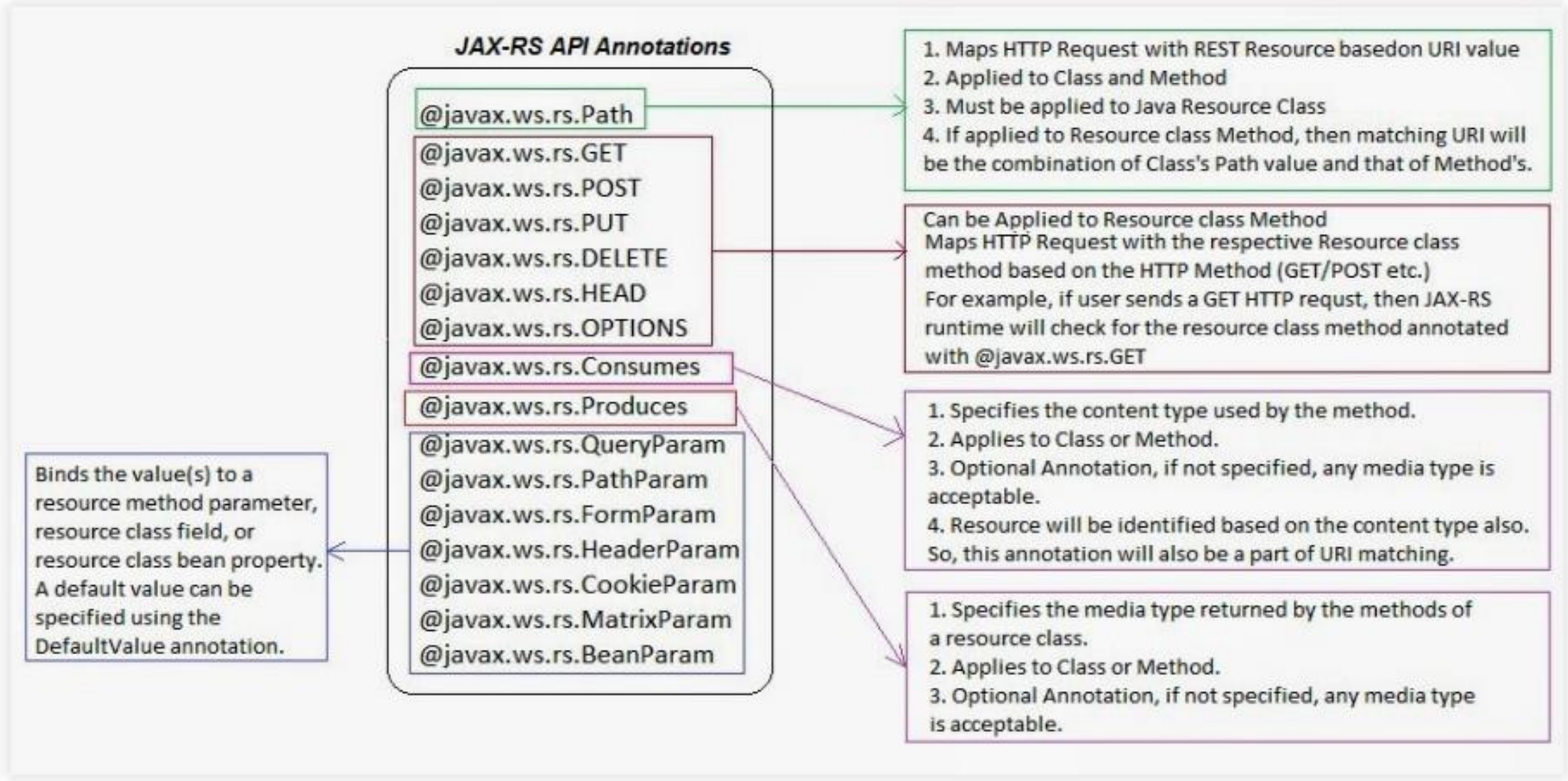
1. *@javax.ws.rs.Path*
2. *@javax.ws.rs.Consumes*
3. *@javax.ws.rs.Produces*
4. *@javax.ws.rs.QueryParam*
5. *@javax.ws.rs.PathParam*
6. *@javax.ws.rs.HeaderParam*
7. *@javax.ws.rs.FormParam*
8. *@javax.ws.rs.CookieParam*
9. *@javax.ws.rs.MatrixParam*
10. *@javax.ws.rs.BeanParam*

JAX-RS API

Method	Purpose	Annotation
GET	Read, possibly cached	@GET
POST	Update or create without a known ID	@POST
PUT	Update or create with a known ID	@PUT
DELETE	Remove	@DELETE
HEAD	GET with no response	@HEAD
OPTIONS	Supported methods	@OPTIONS

JAX-RS API (contd.)

Annotation	Sample
@PathParam	Binds the value from URI, e.g. @PathParam("id")
@QueryParam	Binds the value of query name/value, e.g. @QueryParam("name")
@CookieParam	Binds the value of a cookie, e.g. @CookieParam("JSESSIONID")
@HeaderParam	Binds the value of a HTTP header , e.g. @HeaderParam("Accept")
@FormParam	Binds the value of an HTML form, e.g. @FormParam("name")
@MatrixParam	Binds the value of a matrix parameter, e.g. @MatrixParam("name")


JAX-RS API Annotations
@javax.ws.rs.Path
@javax.ws.rs.GET
@javax.ws.rs.POST
@javax.ws.rs.PUT
@javax.ws.rs.DELETE
@javax.ws.rs.HEAD
@javax.ws.rs.OPTIONS
@javax.ws.rs.Consumes
@javax.ws.rs.Produces
@javax.ws.rs.QueryParam
@javax.ws.rs.PathParam
@javax.ws.rs.FormParam
@javax.ws.rs.HeaderParam
@javax.ws.rs.CookieParam
@javax.ws.rs.MatrixParam
@javax.ws.rs.BeanParam
1. Maps HTTP Request with REST Resource basedon URI value
2. Applied to Class and Method
3. Must be applied to Java Resource Class
4. If applied to Resource class Method, then matching URI will be the combination of Class's Path value and that of Method's.
Can be Applied to Resource class Method
Maps HTTP Request with the respective Resource class method based on the HTTP Method (GET/POST etc.)
For example, if user sends a GET HTTP requst, then JAX-RS runtime will check for the resource class method annotated with @javax.ws.rs.GET
1. Specifies the content type used by the method.
2. Applies to Class or Method.
3. Optional Annotation, if not specified, any media type is acceptable.
4. Resource will be identified based on the content type also. So, this annotation will also be a part of URI matching.
1. Specifies the media type returned by the methods of a resource class.
2. Applies to Class or Method.
3. Optional Annotation, if not specified, any media type is acceptable.
Binds the value(s) to a resource method parameter, resource class field, or resource class bean property. A default value can be specified using the DefaultValue annotation.

Matrix vs Query parameters

Matrix parameters are alternative to Query parameters. Both can insert optional parameters in a URL.

Matrix Parameter format:
http://www.example.com/example-page;field1=value1;field2=value2;field3=value3

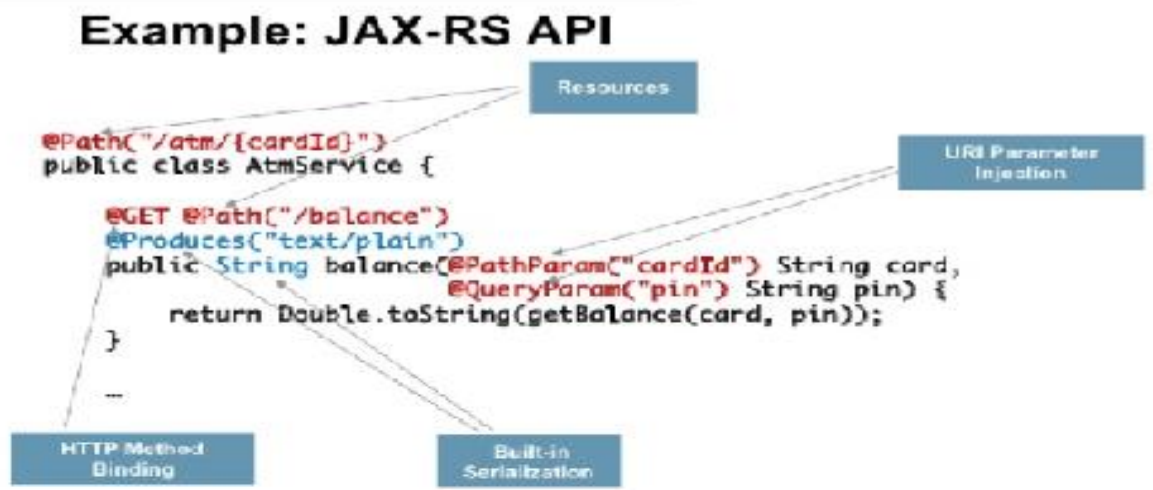
Query Parameter format:
http://www.example.com/example-page?field1=value1&field2=value2&field3=value3

Matrix parameter is more flexible, most importantly it can accept parameter anywhere in the path and not limited to the end:

http://www.example.com/example-page;field1=value1;field2=value2;field3=value3/other-example-page

The @RestController annotation in Spring MVC is nothing but a combination of @Controller and @ResponseBody annotation

Java API for RESTful Web Services



JAX-RS: Java API for RESTful Web Services is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer architectural pattern. Wikipedia

Developed by: Oracle Corporation (initial code from Sun Microsystems)

Written in: Java

Platform: Java virtual machine

@GET

Annotate your Get request methods with @GET.

```
@GET
public String getHTML() {
  ...
}
```

@Produces

@Produces annotation specifies the type of output this method (or web service) will produce.

```
@GET
@Produces("application/xml")
public Contact getXML() {
  ...
}

@GET
@Produces("application/json")
public Contact getJSON() {
  ...
}
```

@Path

@Path annotation specify the URL path on which this method will be invoked.

```
@GET
@Produces("application/xml")
@Path("xml/{firstName}")
public Contact getXML() {
  ...
}
```

@PathParam

We can bind REST-style URL parameters to method arguments using @PathParam annotation as shown below.

```
@GET
@Produces("application/xml")
@Path("xml/{firstName}")
public Contact getXML(@PathParam("firstName") String firstName) {
  Contact contact = contactService.findByFirstName(firstName);
  return contact;
}
```

```
1 @GET
```

@QueryParam

Request parameters in query string can be accessed using @QueryParam annotation as shown below.

```
@GET
@Produces("application/json")
@Path("json/companyList")
public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {
  CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
  return list;
}
```

The example above returns a list of companies (with server side pagination)

@POST

Annotate POST request methods with @POST.

```
@POST
@Consumes("application/json")
@Produces("application/json")
public RestResponse<Contact> create(Contact contact) {
...
}
```

@Consumes

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.

```
@PUT
@Consumes("application/json")
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> update(Contact contact) {
...
}
```

@FormParam

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using @QueryParam annotation.

```
@POST
public String save(@FormParam("firstName") String firstName,
    @FormParam("lastName") String lastName) {
      ...
  }
```

@PUT

Annotate PUT request methods with @PUT.

```
@PUT
@Consumes("application/json")
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> update(Contact contact) {
...
}
```

@DELETE

Annotate DELETE request methods with @DELETE.

```
@DELETE
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {
...
}
```

Form.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Strict//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Form Page</title>
</head>
<body>
<h1>Submit the following form</h1>

   <form action="rest/members/info" method="post">
      <p>
         First Name : <input type="text" name="fname" />
      </p>
      <p>
         Last Name : <input type="text" name="lname" />
      </p>
      <input type="submit" value="Submit" />
   </form>
</body>
</html>
```

REST Service

```
package demo.rest;

import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("/members")
public class HelloWorldREST {

    @POST
    @Path("/info")
    public Response responseMsg(@FormParam("fname") String fname,
@FormParam("lname") String lname ) {

        String output = "This all the info about "+fname +" "+lname;
        return Response.status(200).entity(output).build();

    }
}
```

@BeanParam

The annotation that may be used to inject custom JAX-RS "parameter aggregator" value object into a resource class field, property or resource method parameter.

The JAX-RS runtime will instantiate the object and inject all it's fields and properties annotated with either one of the @XxxParam annotation (@PathParam, @FormParam ...) or the @Context annotation.

For the POJO classes same instantiation and injection rules apply as in case of instantiation and injection of request-scoped root resource classes.

For example:

```
public class MyBean {
  @FormParam("myData")
  private String data;

  @HeaderParam("myHeader")
  private String header;

  @PathParam("id")
  public void setResourceId(String id) {...}

  ...
}

@Path("myresources")
public class MyResources {
  @POST
  @Path("{id}")
  public void post(@BeanParam MyBean myBean) {...}

  ...
}
```

@MatrixParam annotation

matrix parameters are an arbitrary set of name-value pairs embedded in a uri path segment.

Example :URI pattern

"**/inventory/switch;company=cisco;model=nexus-5596**"

```
import javax.ws.rs.GET;
import javax.ws.rs.MatrixParam;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;

@Path("/inventory")
public class InventoryService {

    @GET
    @Path("{deviceType}")
    public Response getInventoryDetails(@PathParam("deviceType")
                                                String deviceType,
            @MatrixParam("company") String company,
            @MatrixParam("model") String model){

        String resp = "Received request for device: "+deviceType+
                      ", comany: "+company+" and model: "+model;
        return Response.status(200).entity(resp).build();
    }
}
```

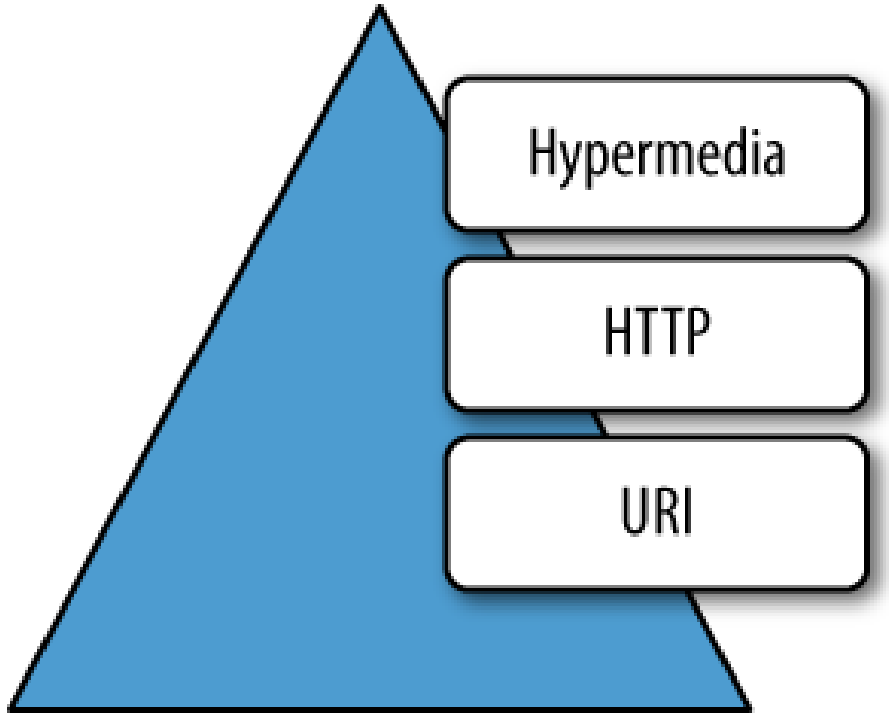
Client API

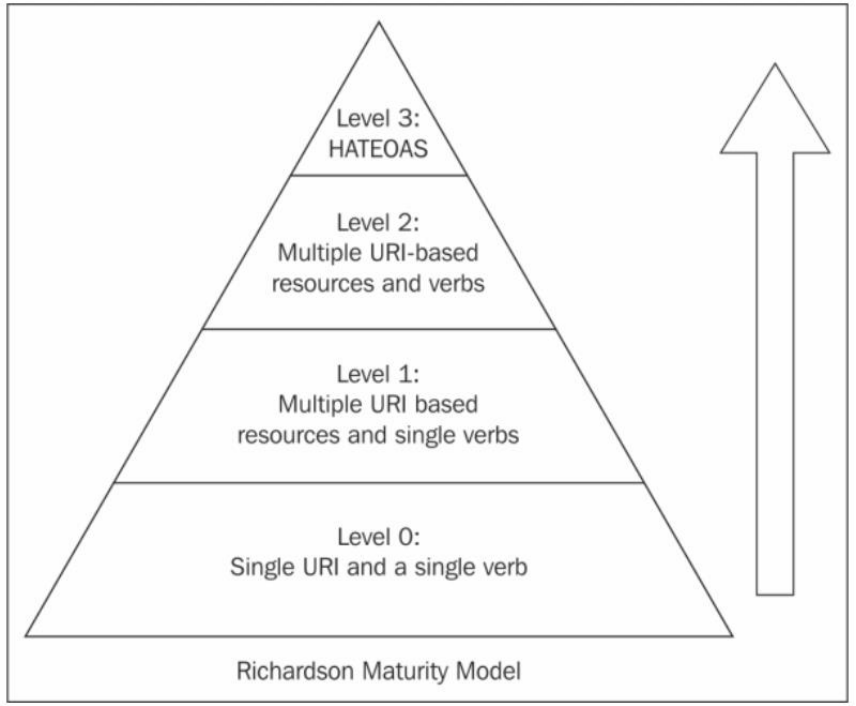
```
import java.net.URI;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;
import org.glassfish.jersey.client.ClientConfig;

public class ClientTest {
  public static void main(String[] args) {
    ClientConfig config = new ClientConfig();
    Client client = ClientBuilder.newClient(config);
    WebTarget target = client.target(getBaseURI());

    System.out.println(target.path("hello").request().accept(MediaType.TEXT_PLAIN).get(String.class));
    System.out.println(target.path("hello").request().accept(MediaType.TEXT_XML).get(String.class));
    System.out.println(target.path("hello").request().accept(MediaType.TEXT_HTML).get(String.class));
  }

  private static URI getBaseURI() {
    return UriBuilder.fromUri("http://localhost:8080/lab1-Rest-Basic/webapi/").build();
  }
}
```

Richardson Maturity Model

Leonard Richardson analyzed a hundred different web service designs and divided them into four categories based on how much they are REST compliant.

This model of division of REST services to identify their maturity level – is called Richardson Maturity Model.





Level Zero

Level zero of maturity does not make use of any of URI, HTTP Methods, and HATEOAS capabilities.

These services have a single URI and use a single HTTP method (typically POST). For example, most Web Services (WS-*)-based services use a single URI to identify an endpoint, and HTTP POST to transfer SOAP-based payloads, effectively ignoring the rest of the HTTP verbs.

Similarly, XML-RPC based services which send data as Plain Old XML (POX). These are the most primitive way of building SOA applications with a single POST method and using XML to communicate between services.

Level One

Level one of maturity makes use of URIs out of URI, HTTP Methods, and HATEOAS.

These services employ many URIs but only a single HTTP verb – generally HTTP POST. They give each individual resource in their universe a URI. Every resource is separately identified by a unique URI – and that makes them better than level zero.

Level Two

Level two of maturity makes use of URIs and HTTP out of URI, HTTP Methods, and HATEOAS.

Level two services host numerous URI-addressable resources. Such services support several of the HTTP verbs on each exposed resource – Create, Read, Update and Delete (CRUD) services. Here the state of resources, typically representing business entities, can be manipulated over the network.

Here service designer expects people to put some effort into mastering the APIs – generally by reading the supplied documentation.

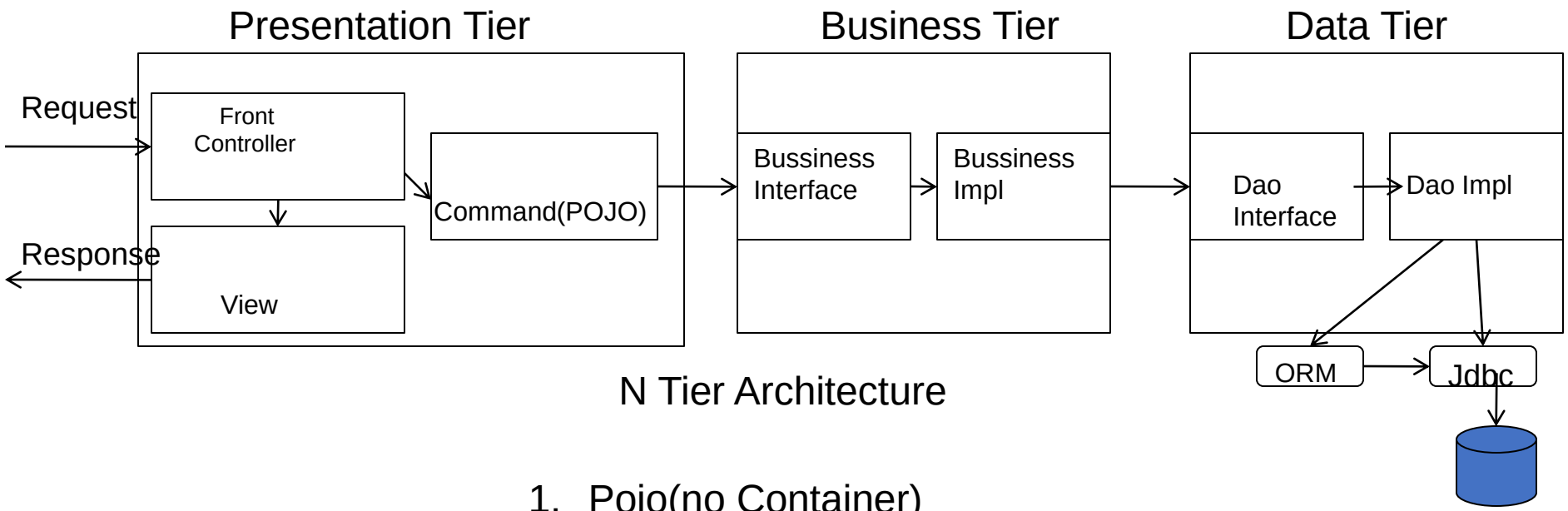
Level 2 is the good use-case of REST principles, which advocate using different verbs based on the HTTP request methods and the system can have multiple resources.

Level Three

Level three of maturity makes use of all three i.e. URIs and HTTP and HATEOAS.

This is the most mature level of Richardson’s model which encourages easy discoverability and makes it easy for the responses to be self-explanatory by using HATEOAS.

The service leads consumers through a trail of resources, causing application state transitions as a result.



N Tier Architecture

1. Servlet/jsp
2. MVC
   Struts
   JSF
   Flex
   Gwt
   Spring MVC
   ...

> Any MVC + Spring

1. Pojo(no Container)
2. Ejb 2.x(HW Container)
   -Session Bean
   -Mdb
3. Pojo + LW Container
   - Spring
   - Microcontainer
   - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
   - Hibernate
   - Kodo
   - Toplink
   - MyBatis
5. JPA

\+ Spring Templates

Spring Framework (IOC, DI &AOP)

IOC is used to decouple common task from implementation.

Six basic techniques to implement Inversion of Control.

These are:

1. using a factory pattern
2. using a service locator pattern
3. using a constructor injection
4. using a setter injection
5. using an interface injection
6. using a contextualized lookup

Constructor, setter, and interface injection are all aspects of Dependency injection.

Life cycle of a bean within Spring application context

1. Instantiate
2. Populate properties(DI)
3. Aware interfaces [BeanNameAware's setBeanName()
   ApplicationContextAware's setApplicationContext ()
4. Pre-initialization Bean Post processors
5. Any method with @PostConstruct
6. InitializingBean's afterPropertiesSet()
7. Call custom init-method
8. Post-initialization BeanPostProcessors
9. Now Bean is ready to use

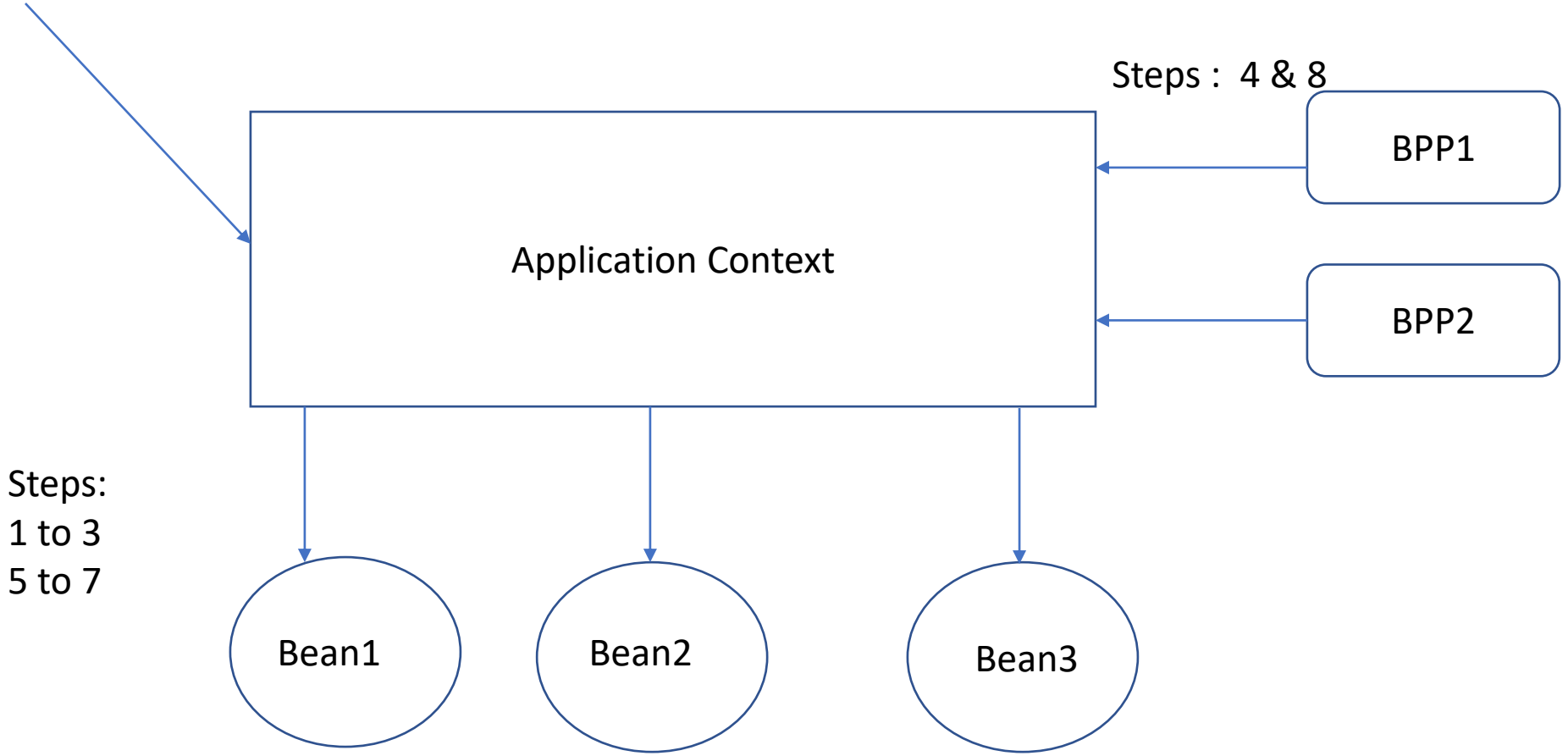
Container is shutdown

1. Any method with @PreDestroy
2. DisposableBean's destroy
3. Call custom destroy-method

Bean is removed from the bean factory in two ways:

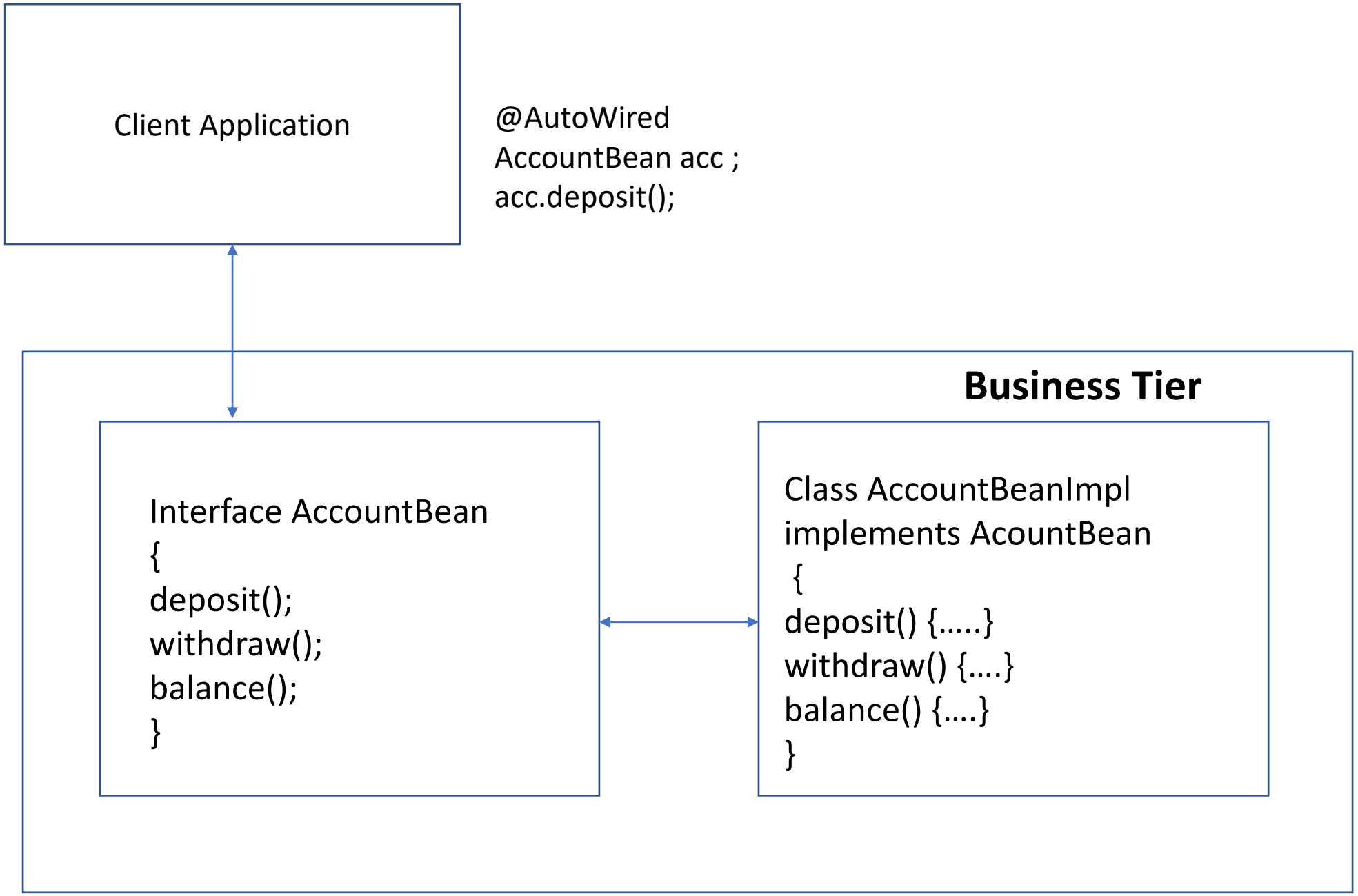
1. If the bean implements the DisposableBean interface,
   the destroy() method is called
2. If a custom destroy-method is specified, it will be called

ApplicationContext context = new ……
Bean1 bean1 = context.getBean(“bean1”)

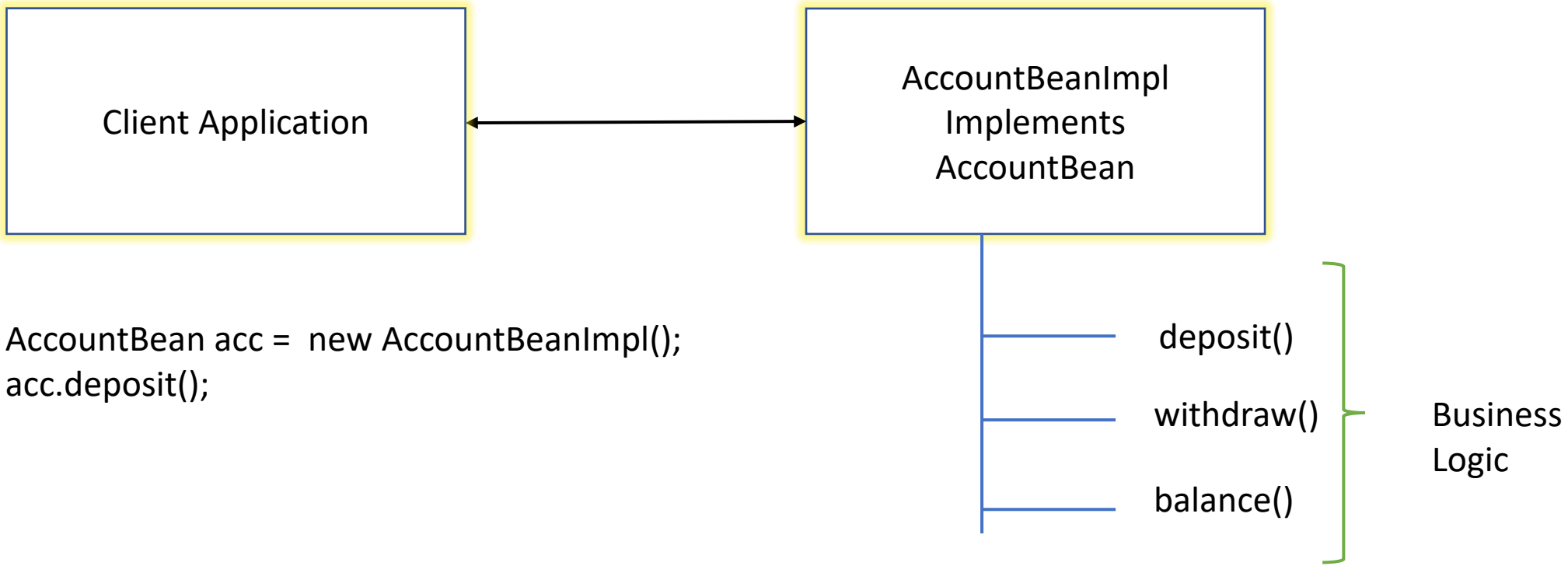


Spring AOP

Spring AOP enables Aspect-Oriented Programming in **spring** applications. In **AOP**, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed crosscutting concerns).

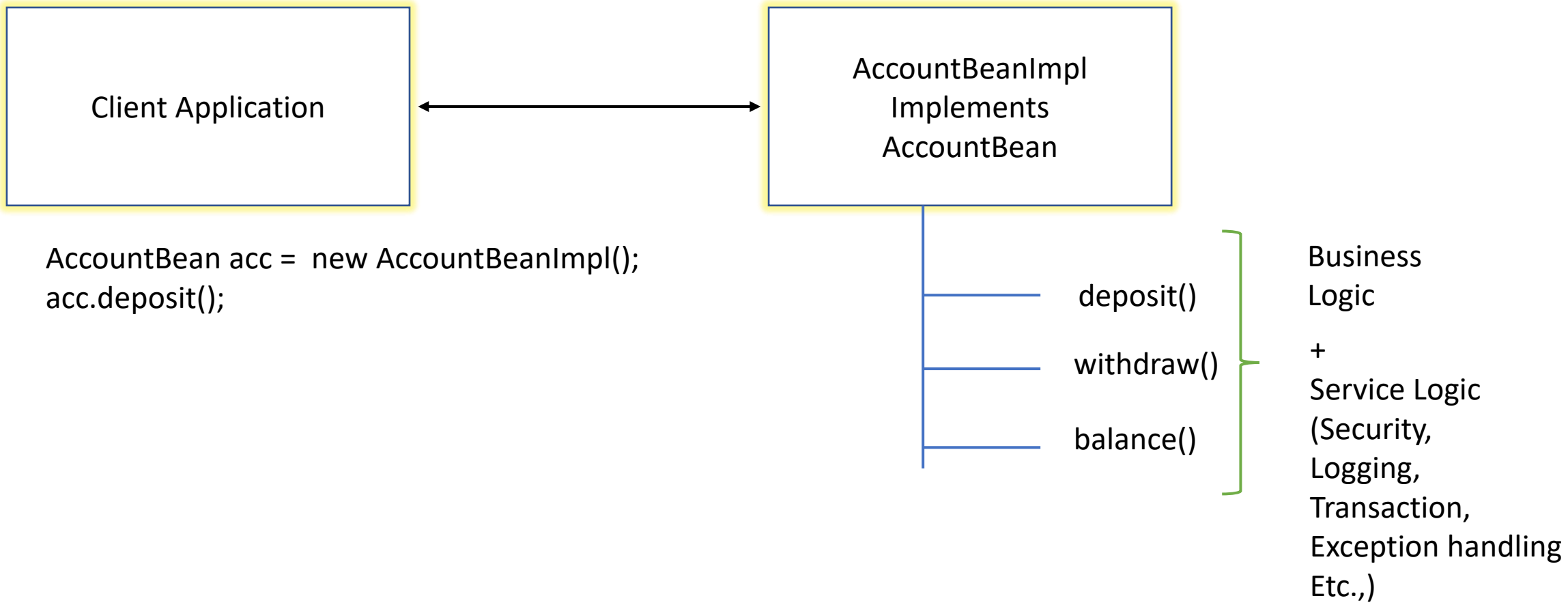

Client Application
@AutoWired
AccountBean acc ;
acc.deposit();
Business Tier
Interface AccountBean
{
deposit();
withdraw();
balance();
}
Class AccountBeanImpl
implements AcountBean
{
deposit() {.....}
withdraw() {....}
balance() {....}
}

Account Bean with business logic

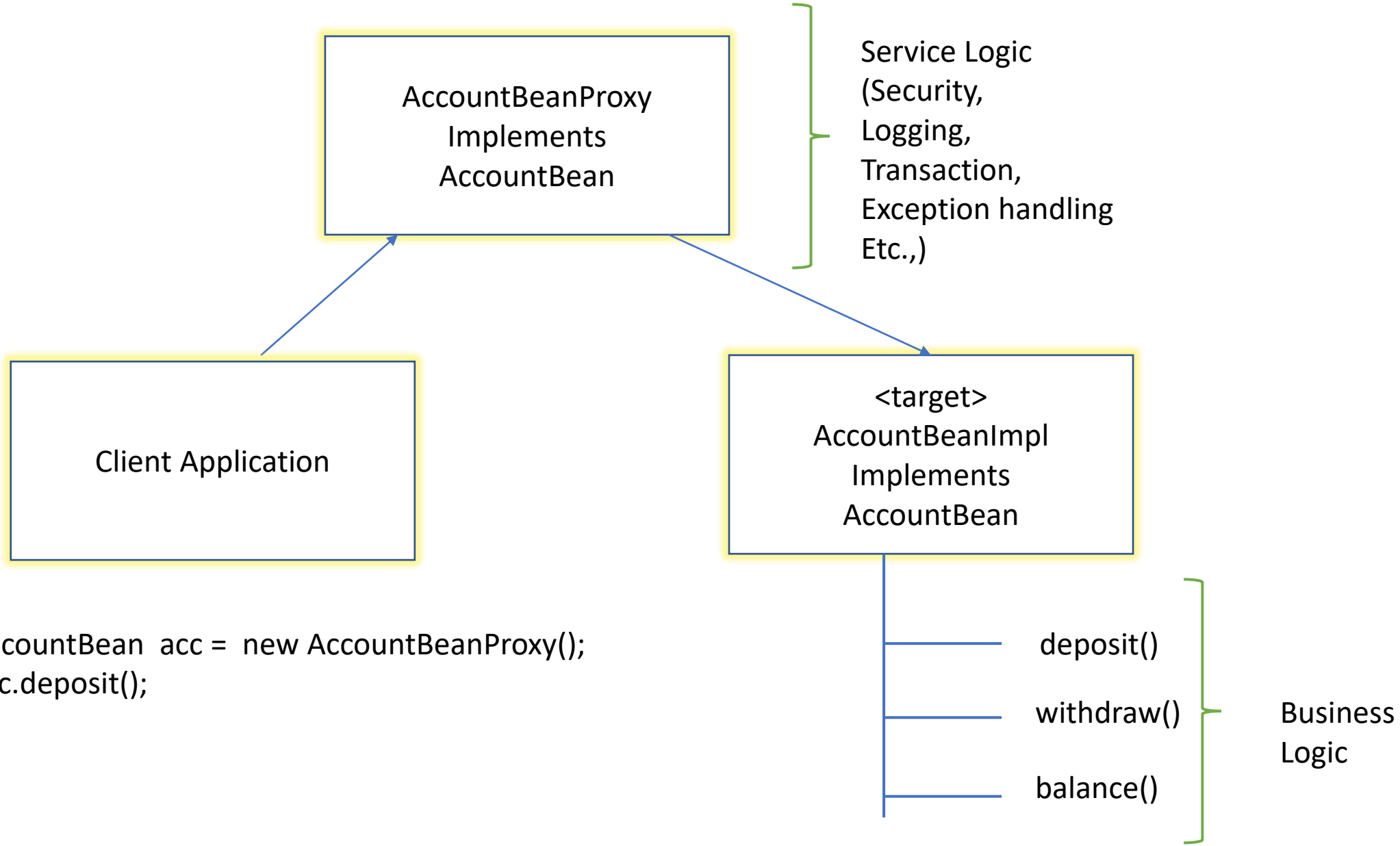



```
AccountBean acc =  new AccountBeanImpl();
acc.deposit();
```

Account Bean with business & service logic (programmatic)



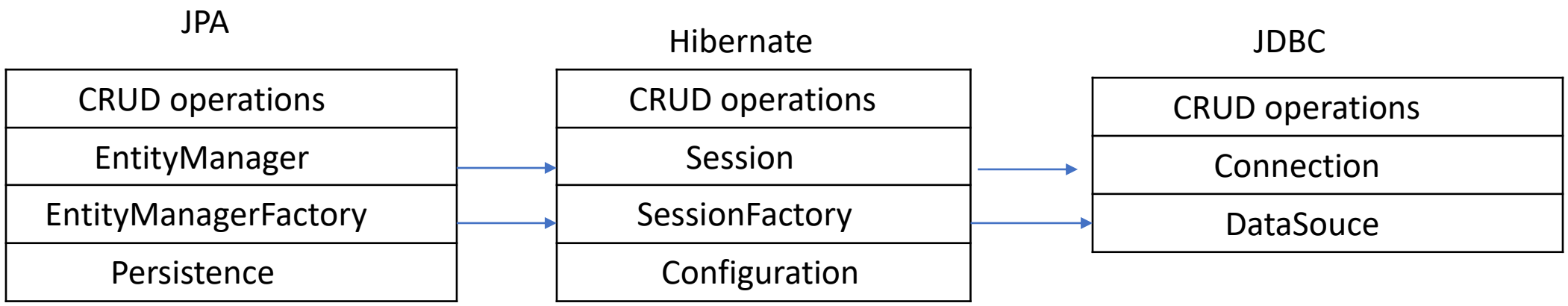
Account Bean with business & service logic (Declarative – using proxy)




```
AccountBean  acc =  new AccountBeanProxy();
acc.deposit();
```

Data Access Layer

JPA / Hibernate / JDBC



Persistence.xml

Javax.persistence.provide = Hibernate

+

Hibernate-jpa.jar

```
@Entity
Class User
{
String name;
....
getName(){}

setName(){}
}
```

```
interface UserDao
{

List<User>  getUsers();

User getUser(long id);

void updateUser(User user);

void deleteUser(User user);

}
```

```
@Repository
Class UserDaoJpa  implements UserDao
{

@PersistenceContext
EntityManager  manager;


List<User>  getUsers()
{   return manager.createQuery("from User"); }

User getUser(long id)
{ return  manager.find(User.class, id); }

void updateUser(User user)  { ... }

void deleteUser(User user)   { ... }

}
```