

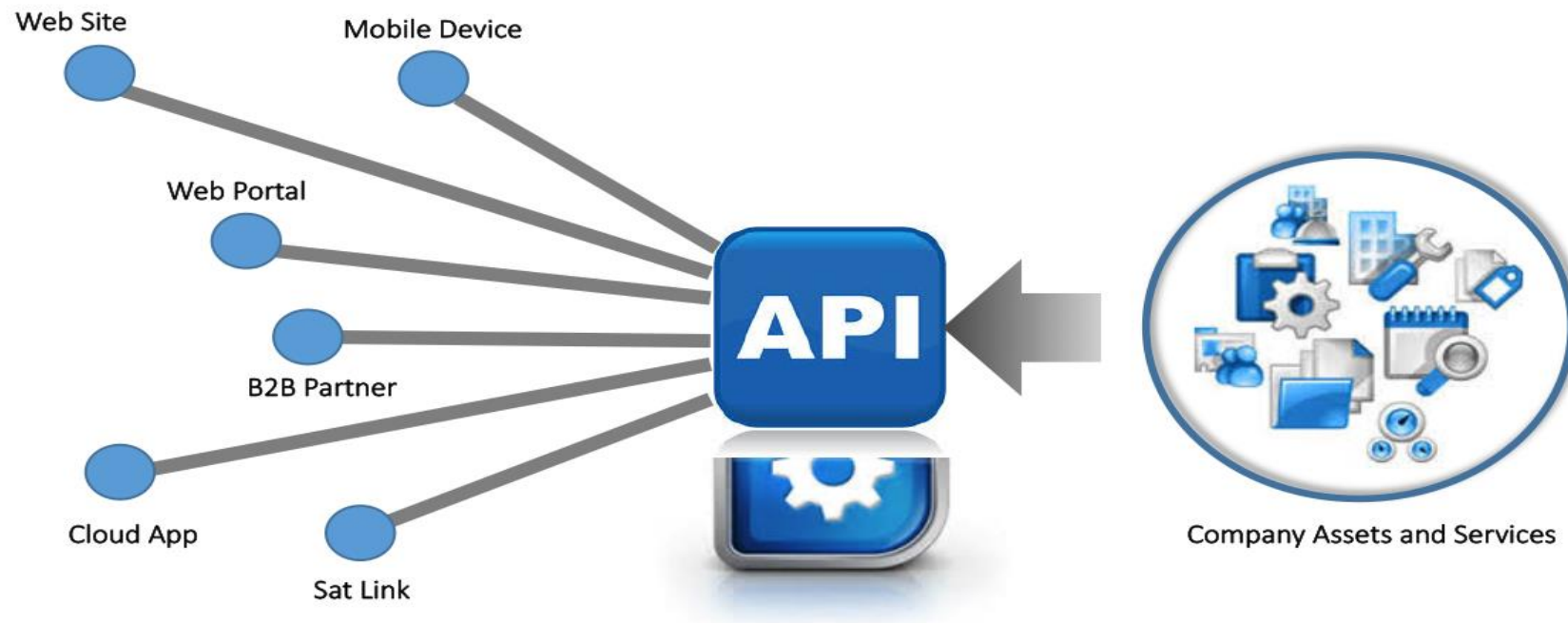
REST API Development

[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.

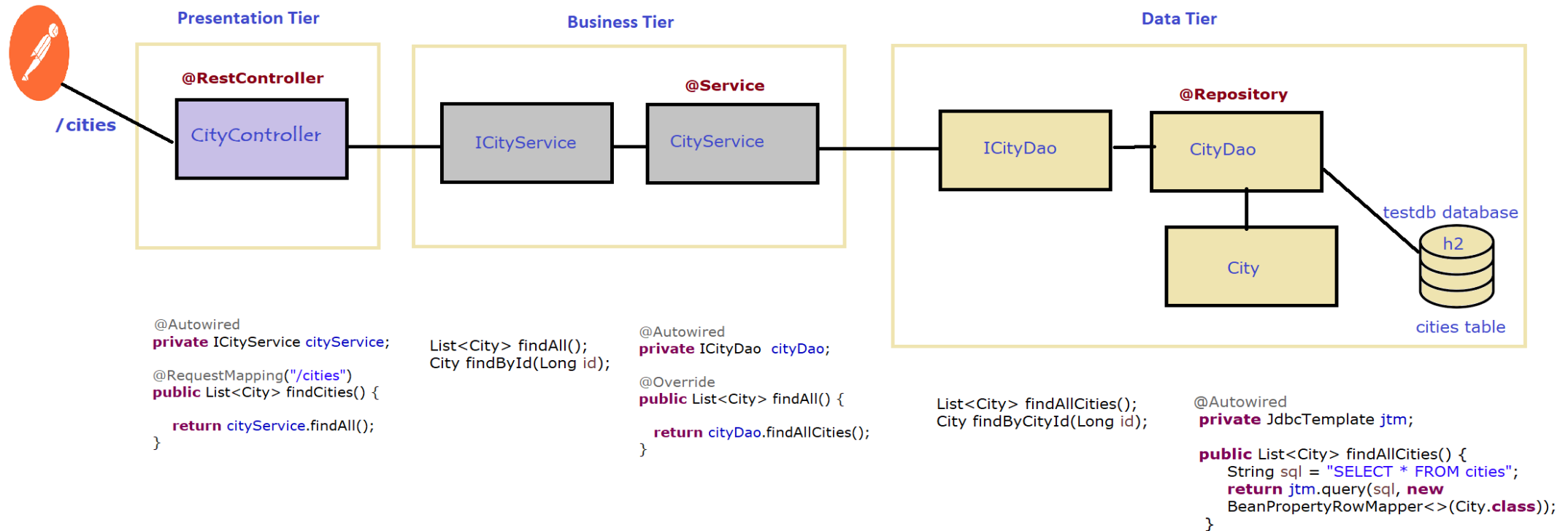
“API plug-ins” simplify and shorten the development life-cycle, making a developer’s role more agile.



An application program interface (API) is code that allows two software programs to communicate with each other.



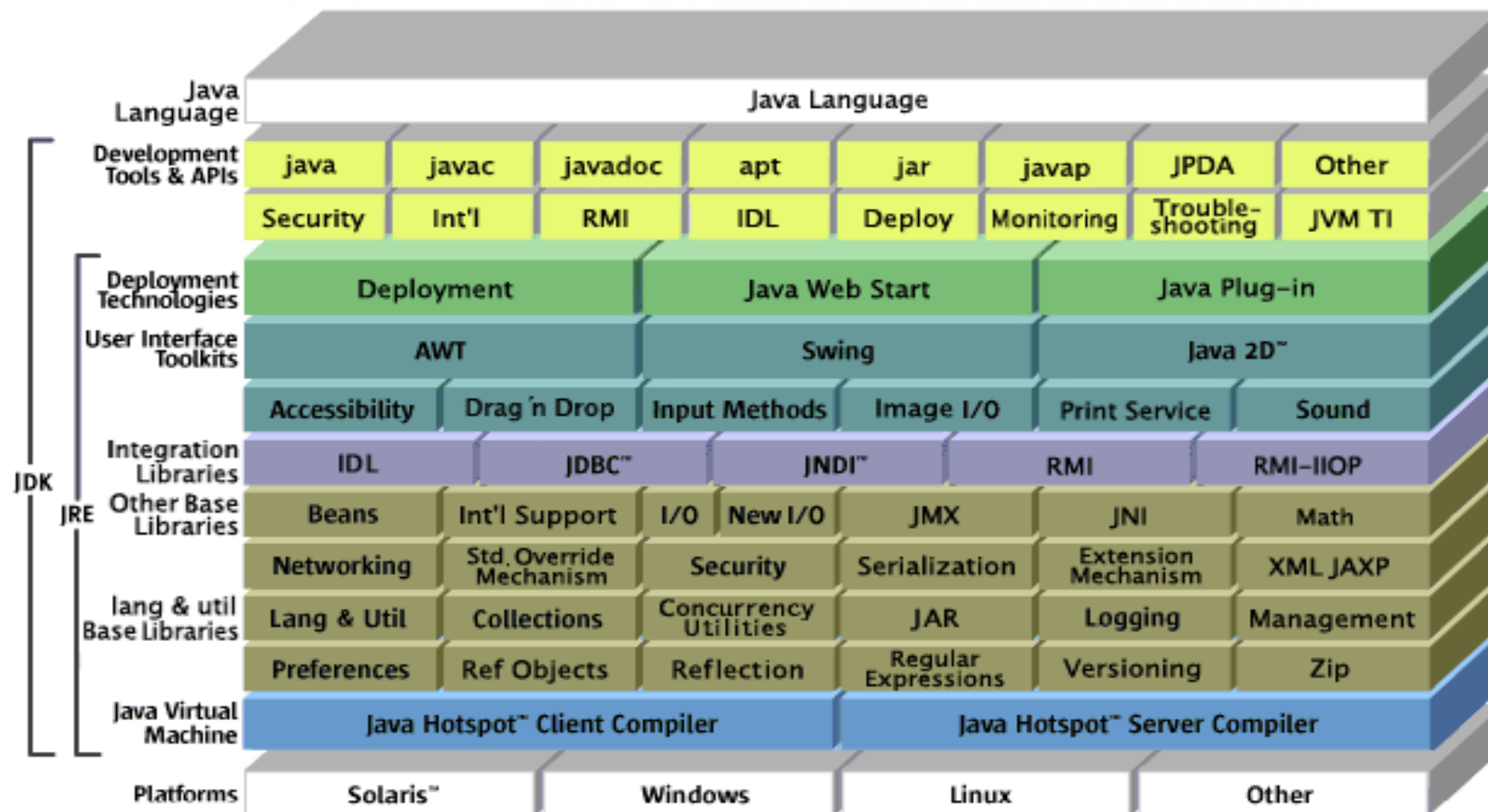
Multitier Architecture



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

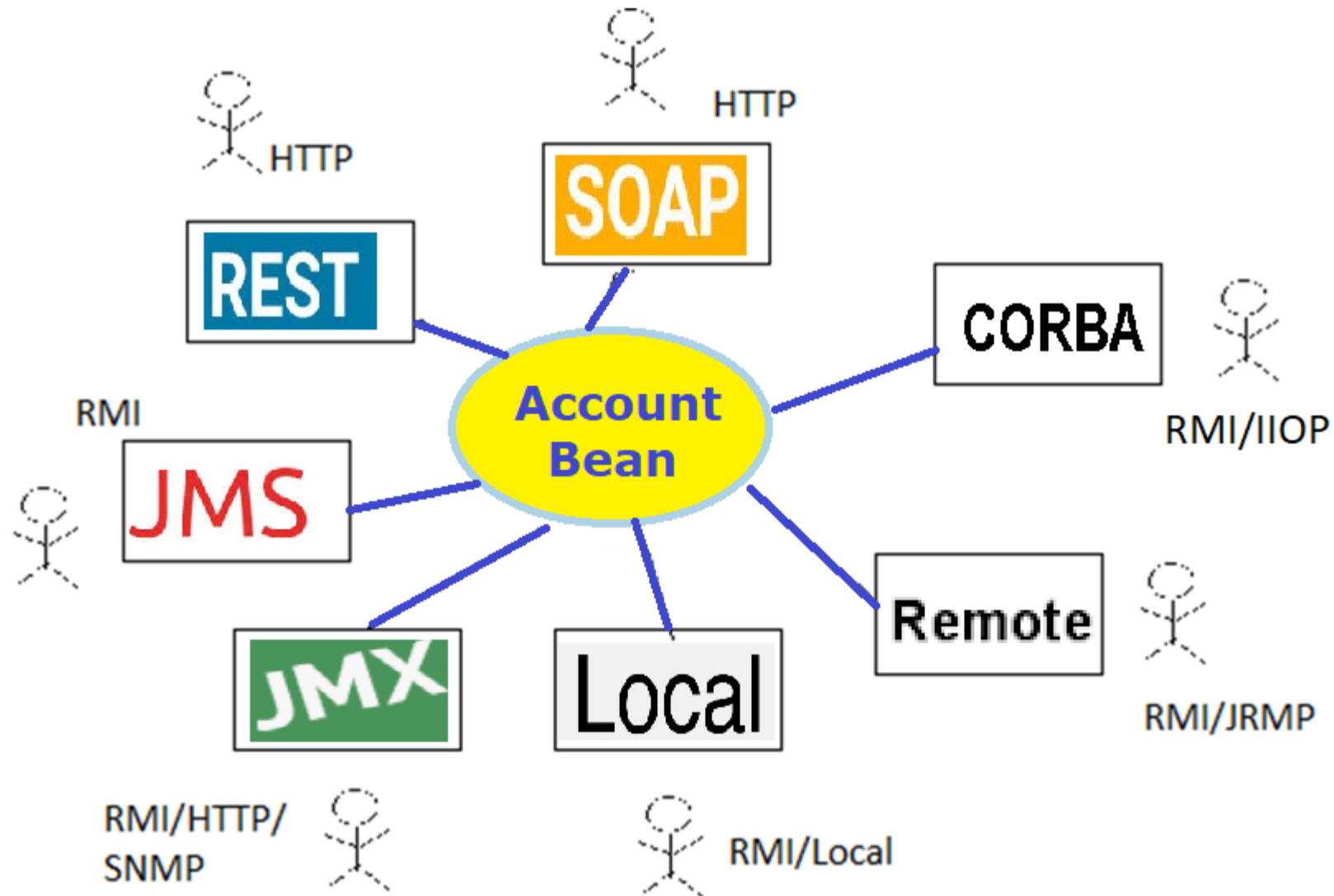


REST

REPRESENTATIONAL STATE TRANSFER

AN ARCHITECTURAL STYLE
FOR DISTRIBUTED
WEB BASED APPLICATIONS

Distributed computing and web services

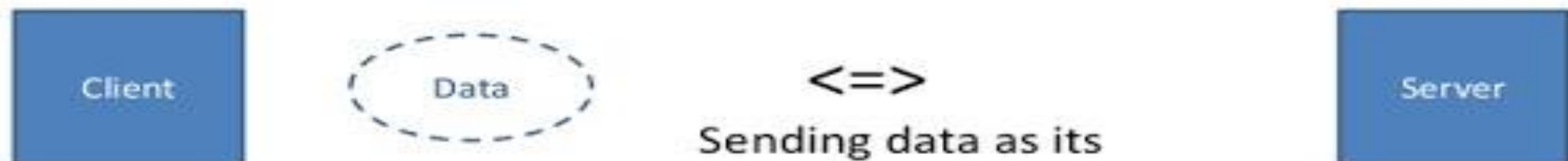


SOAP vs REST

SOAP



REST








*REST are mostly used in industry

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

← → ↻ spicejet.com

 **BOOK** ADD-ONS DEALS GIFT CARD SP

 **Flights**  Hotels  Holiday Packages  Flight S


☒ One Way ☐ Round Trip ☐ Multicity

*FROM *TO *DEPART DATE

← → ↻ yatra.com

yatra Flights Hotels

Book Flights, Hotels and Holiday Packages

Depart From
New Delhi
DEL 

<html>

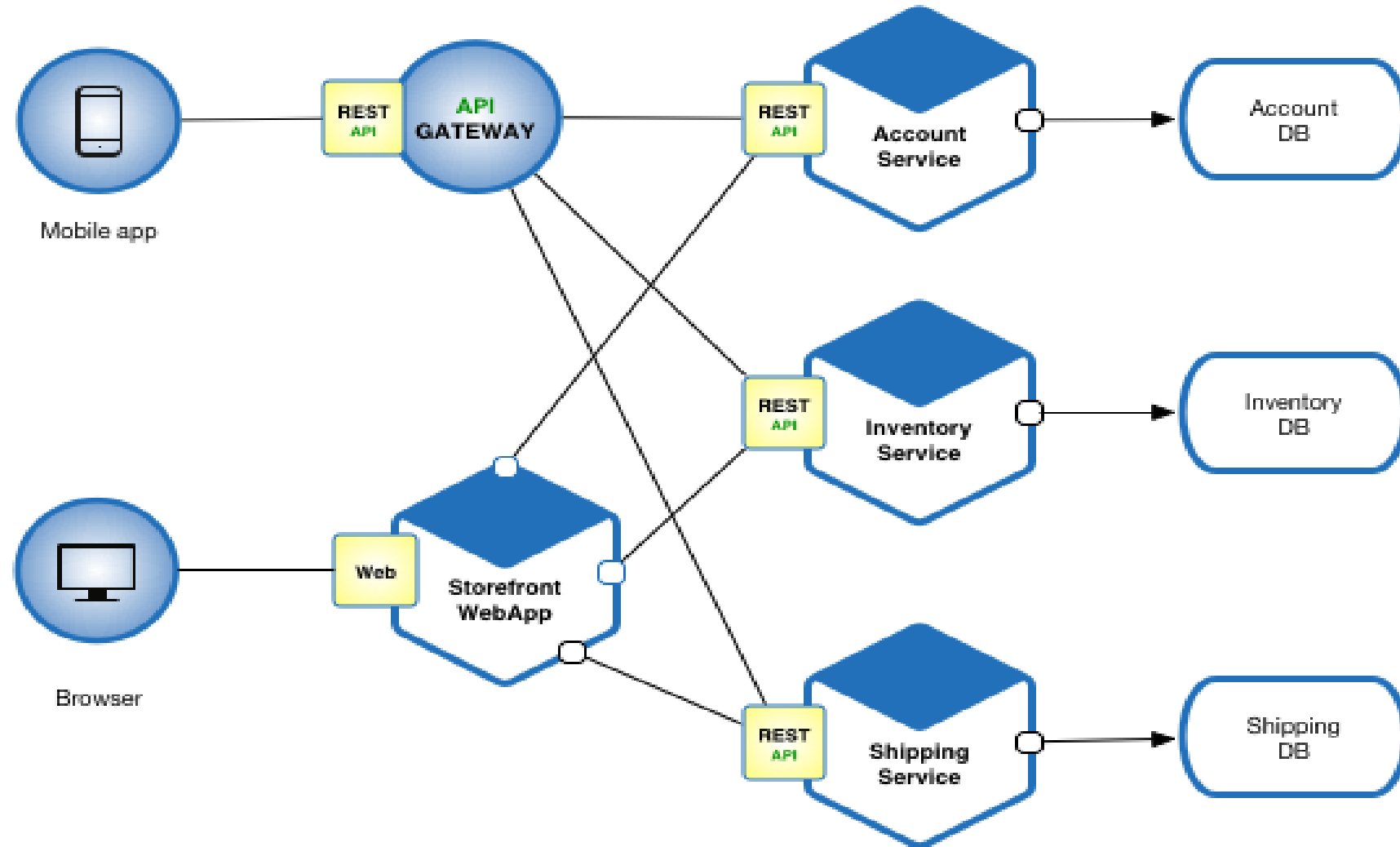
Web MVC

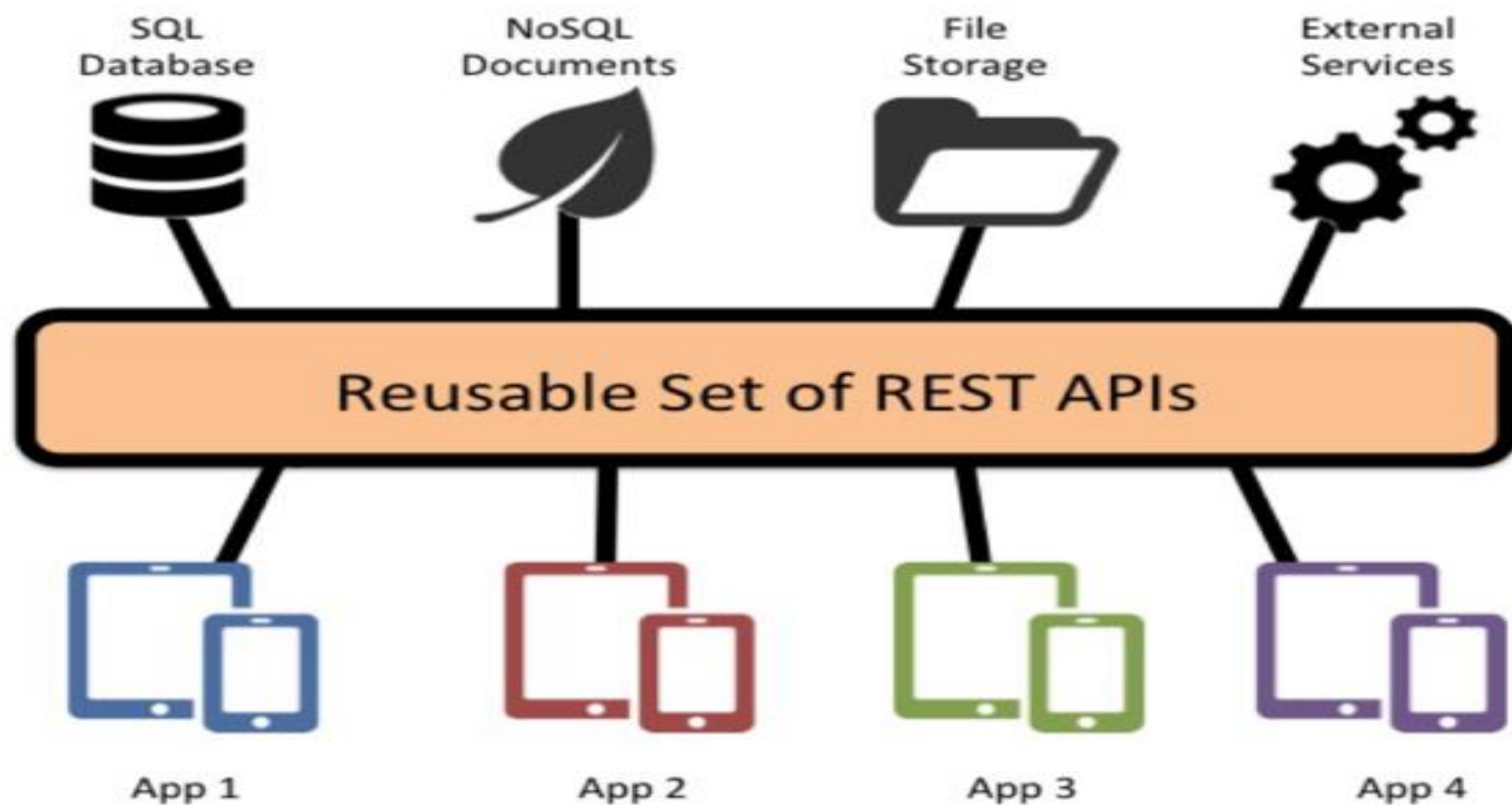
Web API

<xml />
{JSON}



Microservice Architecture





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.

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.



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

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



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

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



Twitter offers two APIs.

The REST API allows developers to access core Twitter data and the Search API provides methods for developers to interact with Twitter Search and trends data.



Google Maps APIs Web Services

A typical Directions API web service request is generally of the following form:

`https://maps.googleapis.com/maps/api/directions/output?parameters`

Add the API key to your request

we must include an API key with every Directions API request.

In the following example, replace YOUR_API_KEY with your API key.

`https://maps.googleapis.com/maps/api/directions/json
?origin=Toronto&destination=Montreal&key=YOUR_API_KEY`

SOAP



REST



SOAP

(Message Exchange Protocol)



REST



SOAP

(Message Exchange Protocol)



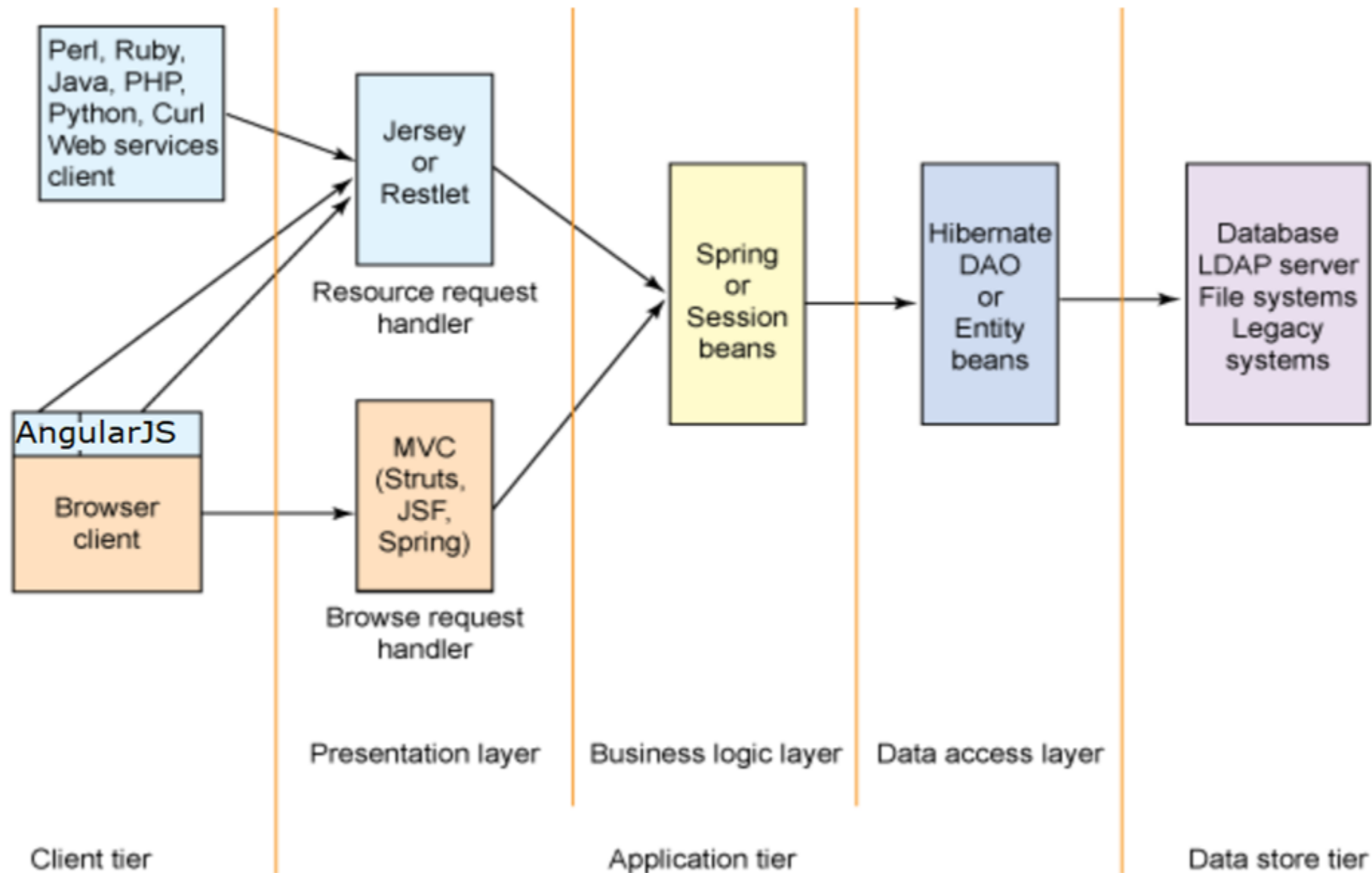
REST

(Pattern / Idea / Concept)



- ❑ **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.

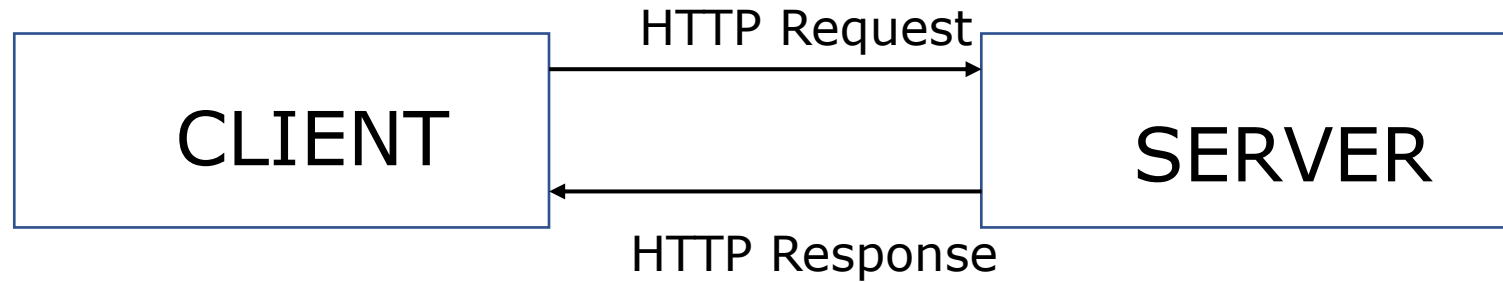
Diagram of a multi-tiered Web application environment



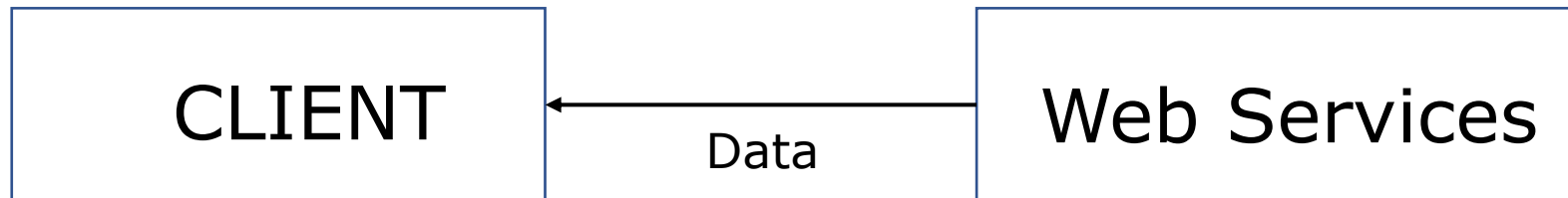
Web service Characteristics

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

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

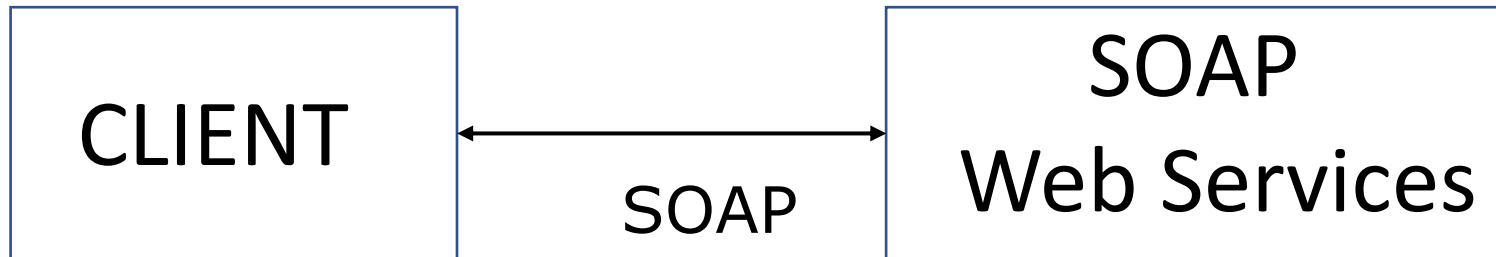


Client applications get the data and present the response in its own format.

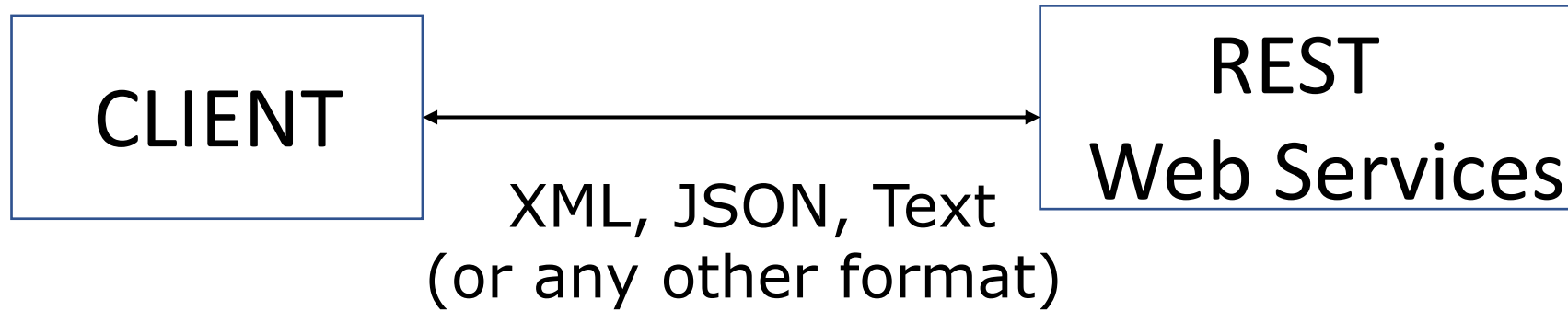


(2). Protocol

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

What is

Representational State Transfer

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.

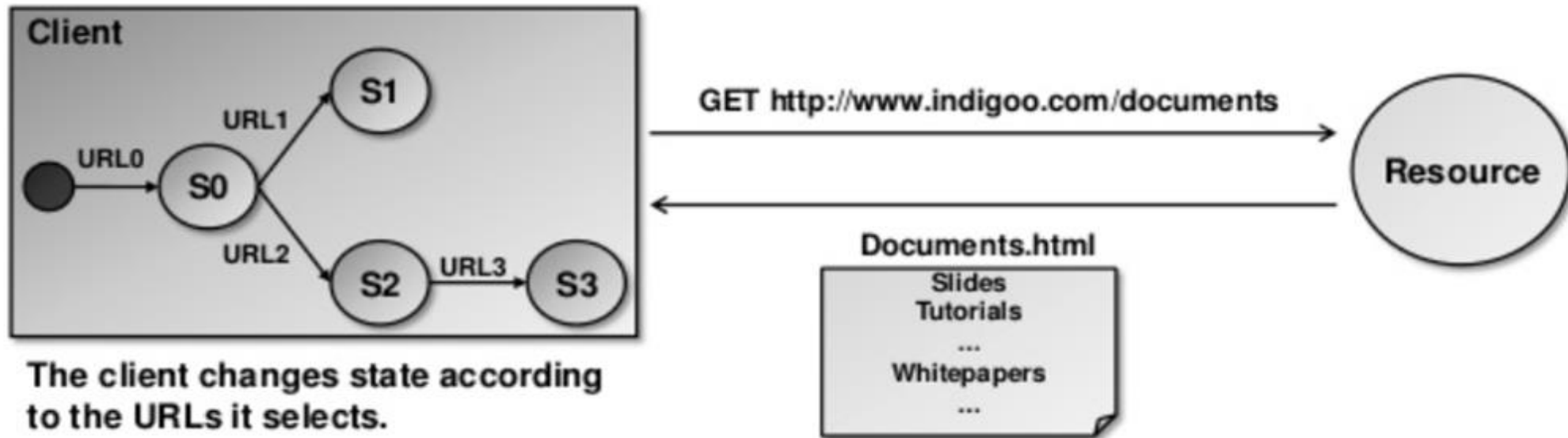
REST, unlike SOAP, is not a WS (web service) standard but an architectural style for web applications.

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.).

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.

Architectural Constraints

REST defines 6 architectural constraints which make any web service – a true RESTful API.

- ☐ **Uniform interface**
- ☐ **Client–server**
- ☐ **Stateless**
- ☐ **Cacheable**
- ☐ **Layered system**
- ☐ **Code on demand (optional)**

Uniform interface

A resource in the system should have only one logical URI, and that should provide a way to fetch related or additional data.

Any single resource should not be too large and contain each and everything in its representation.

Whenever relevant, a resource should contain links (HATEOAS) pointing to relative URIs to fetch related information.

The resource representations across the system should follow specific guidelines such as naming conventions, link formats, or data format (XML or/and JSON).

“Once a developer becomes familiar with one of your APIs, he should be able to follow similar approach for other APIs.”

Client-server

Client application and server application MUST be able to evolve separately without any dependency on each other.

A client should know only resource URIs, keep it simple.

“Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.”

Stateless

Roy fielding got inspiration from HTTP, so it reflects in this constraint.

Make all client-server interactions stateless.

The server will not store anything about the latest HTTP request the client made. It will treat every request as new. No session, no history.

“No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.”

Cacheable

Caching of data and responses is of utmost important wherever they are applicable/possible.

Caching brings performance improvement for the client-side and better scope for scalability for a server because the load has reduced. Improve network performance.

In REST, caching shall be applied to resources when applicable, and then these resources **MUST declare themselves cacheable. Caching can be implemented on the server or client-side.**

“Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.”

Layered system

Layers (or tiers) are aimed at the decomposition of the system functionality.

Decomposition of system functionality into client, server and intermediary.

REST allows you to use a layered system architecture where we deploy the APIs on server A, and store data on server B and authenticate requests in Server C, for example.

Client will not have any idea about these servers.

Code on demand (optional)

This constraint is optional. Most of the time, we will be sending the static representations of resources in the form of XML or JSON.

But when the client need some executable code, the server can return the same.

e.g., Clients may call your API to get a UI widget rendering code. It is permitted.

REST protocol

REST is not a protocol like SOAP.

But REST defines some core characteristics that make a system REST-ful.

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 (<i>C</i>)	→ HTTP POST
Read (<i>R</i>)	→ HTTP GET
Update (<i>U</i>)	→ HTTP PUT
Delete (<i>D</i>)	→ 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:

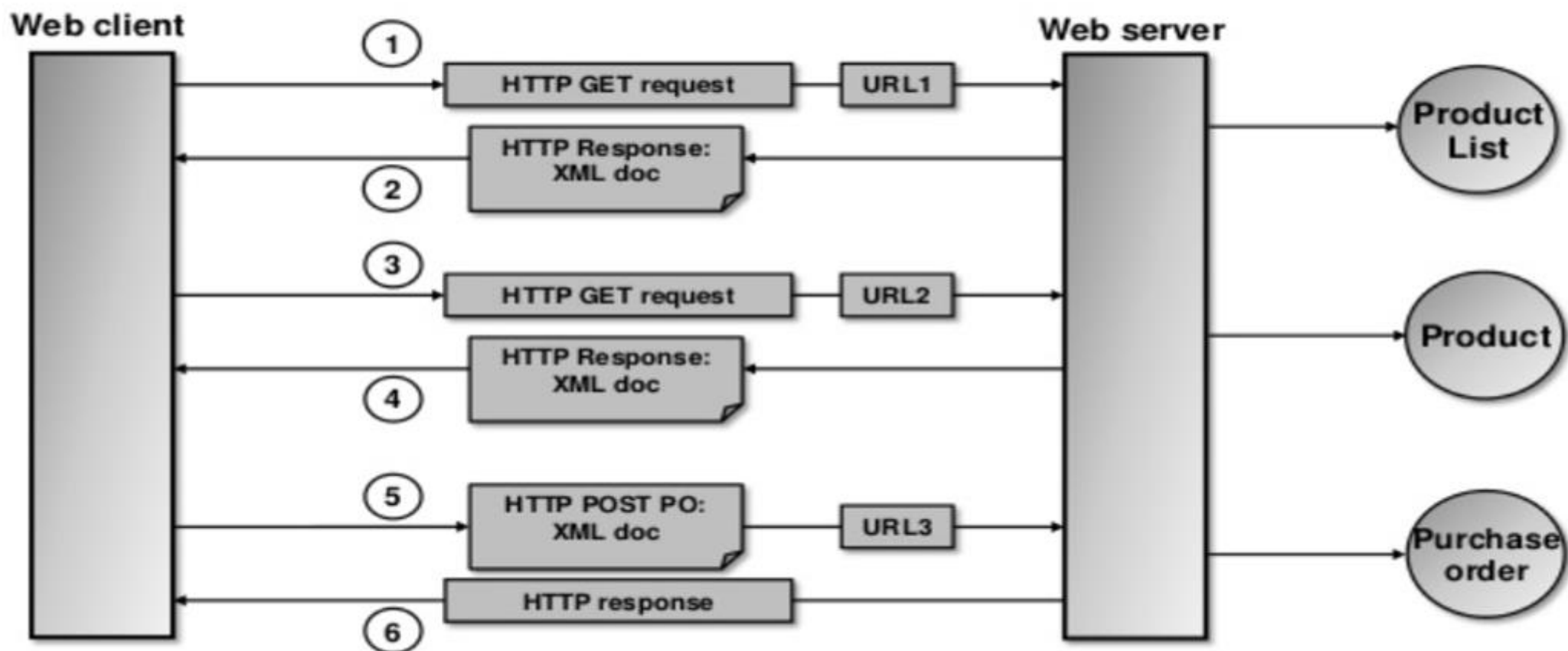
REST uses the HTTP header *Content-type* (MIME types like text/html, text/plain, text/xml, text/javascript for JSON etc.) to indicate the encoding of the resource.

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):



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?isbn=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) `/retrieveItems.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)

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)

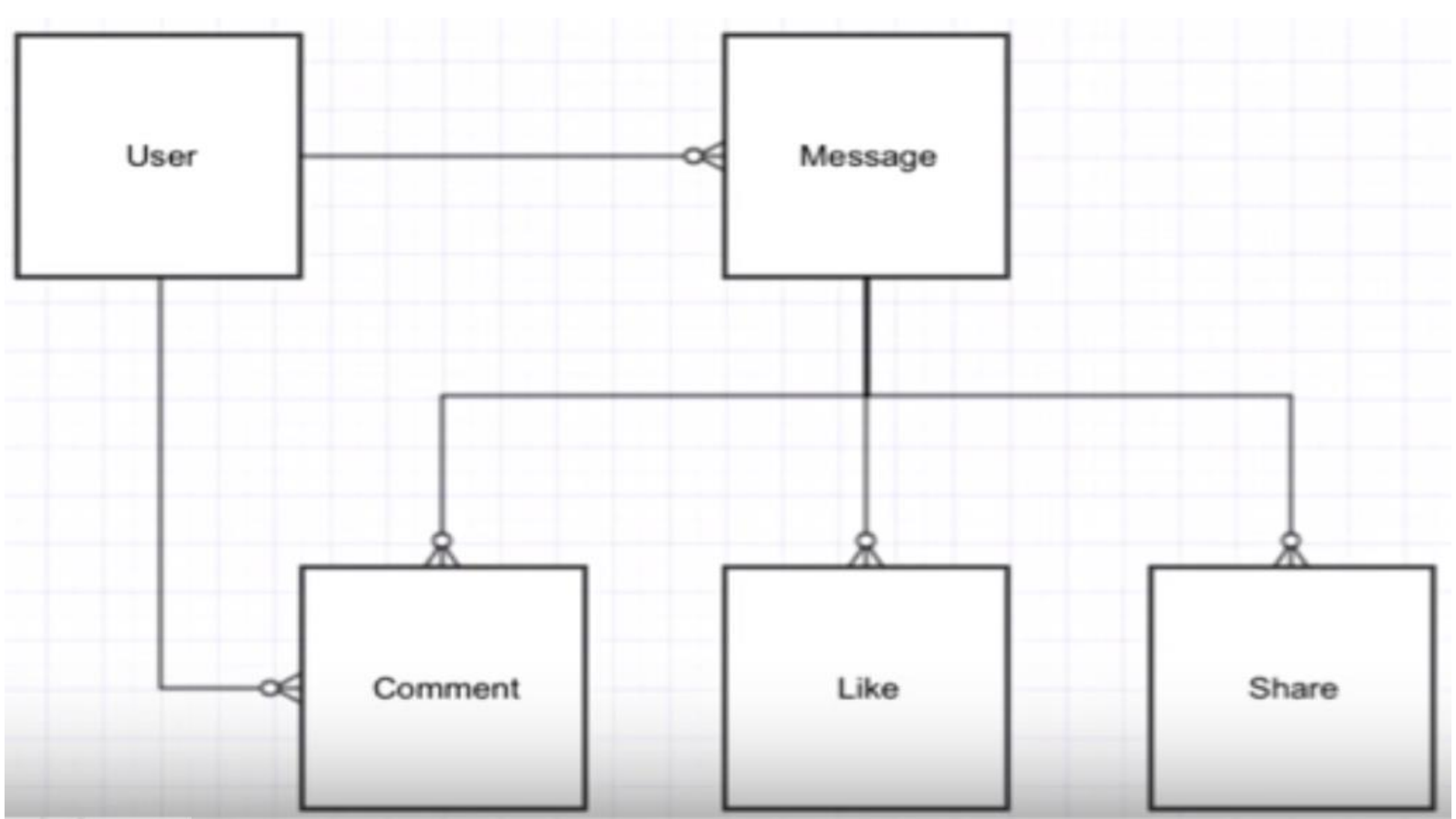
Resource-based URIs:

- ☐ 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)



Messenger - A social media application API

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



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

/messages

/messages/{messageId}/comments

/messages/{messageId}/likes

/messages/{messageId}/shares

How to get all comments

/message/comment [not recommended]

/comment [recommended]

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 Codes		
Level 200 (Success) 200 : OK 201 : Created 203 : Non-Authoritative Information 204 : No Content	Level 400 400 : Bad Request 401 : Unauthorized 403 : Forbidden 404 : Not Found 409 : Conflict	Level 500 500 : Internal Server Error 503 : Service Unavailable 501 : Not Implemented 504 : Gateway Timeout 599 : Network timeout 502 : Bad Gateway

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: 'rushy',
  age: 32
}
```

```
// PATCH request
{
  age: 33
}
```

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

PATCH Non-idempotent example:

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

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

```
// New resource
{
  name: 'rushy',
  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.

Jersey

The word "SETUP" is written in bold blue capital letters and is centered within a bright yellow horizontal oval.

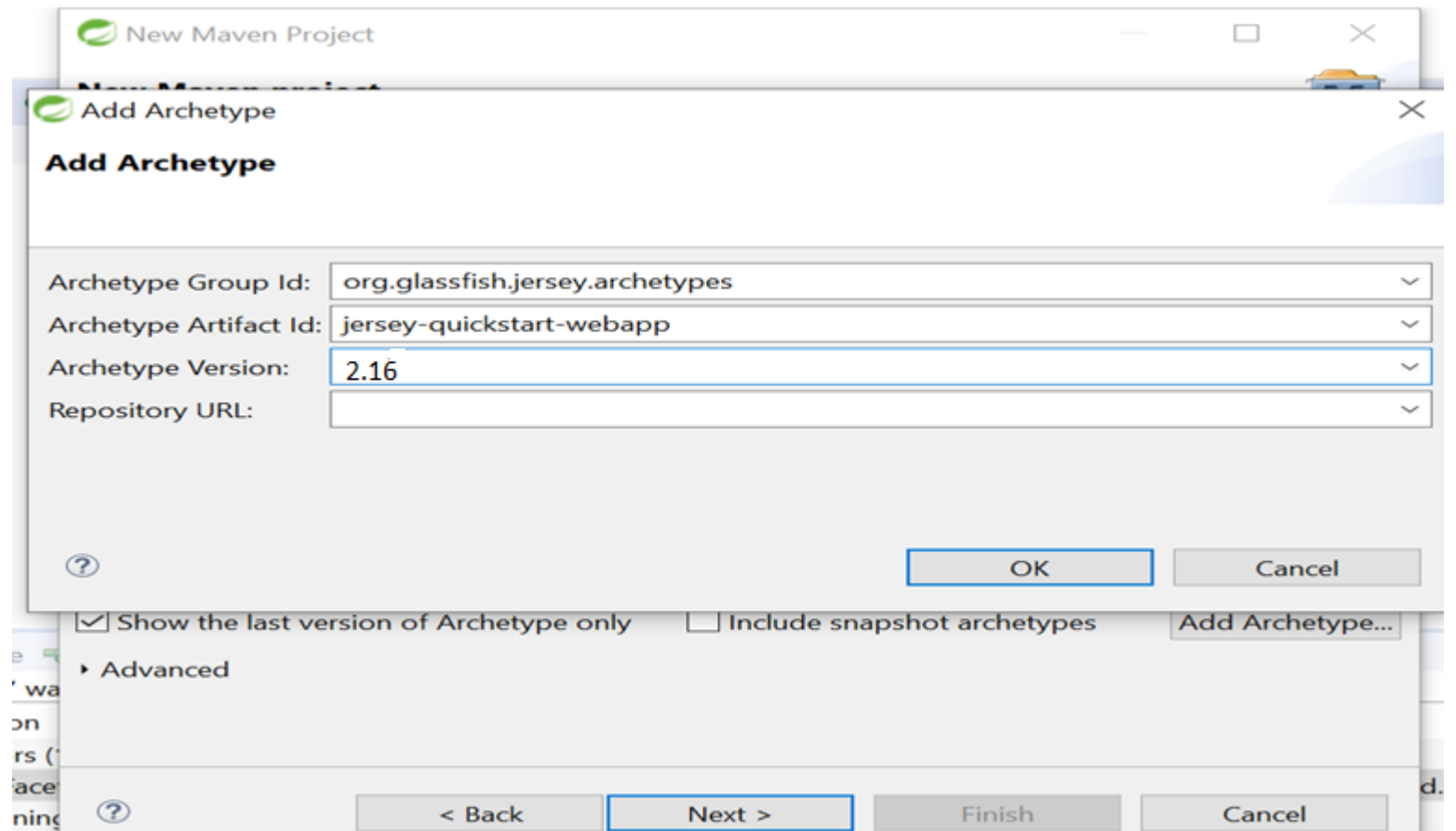
SETUP

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

Download

Jersey is distributed mainly via Maven and it offers some extra modules. Check the [How to Download](#) page or see our list of [dependencies](#) for details.

- 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



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

Software framework

A software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.

Jersey, Spring, Hibernate, MyBatis etc.,

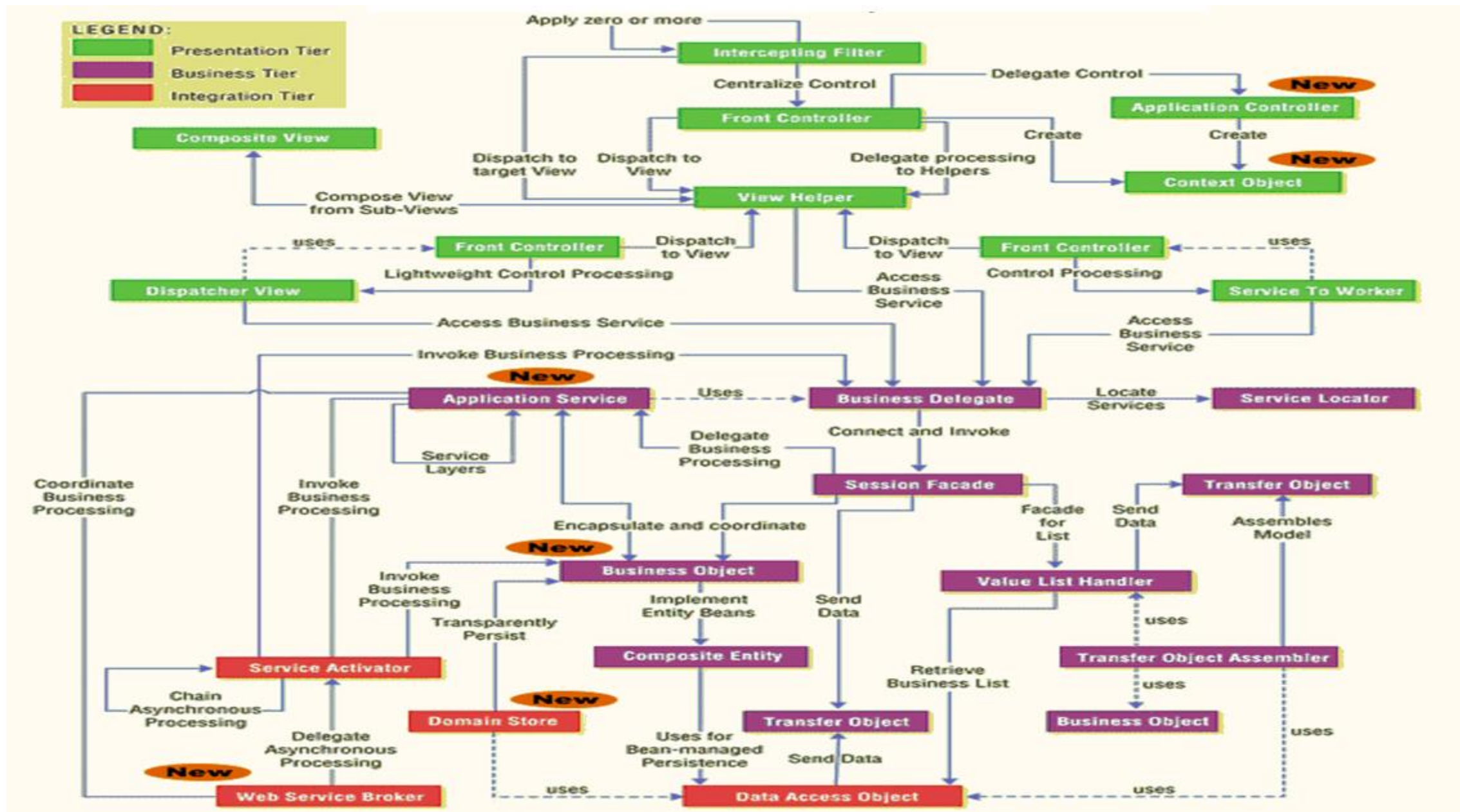
Design Pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

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





JavaEE Patterns

Architectural patterns:

MVC, SOA, Restful, Microservices, MOM

Enterprise Integration Patterns:

Splitter, Aggregator, Dynamic Router, Scatter-Gather, Content Enricher, Content Filter, Message Filter etc.,

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

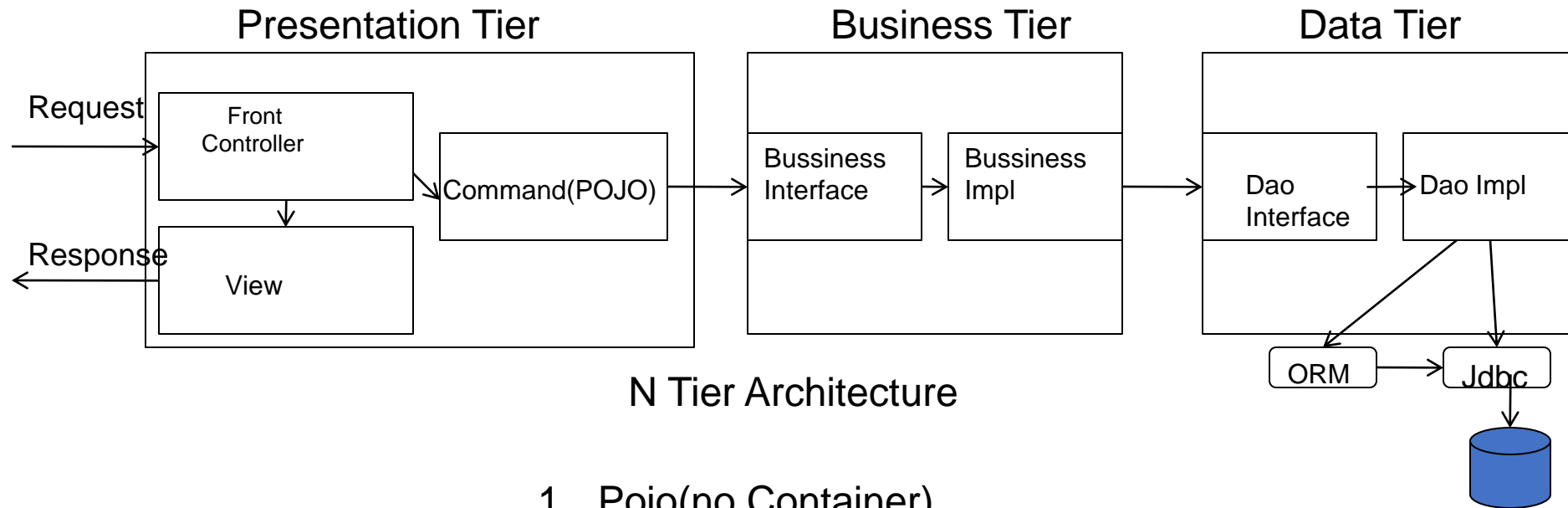
Container design patterns (Distributed Systems)

- ❑ The sidecar design pattern
- ❑ The ambassador design pattern
- ❑ The adapter design pattern
- ❑ The leader election design pattern
- ❑ The work queue design pattern
- ❑ The scatter/gather design pattern

Domain-Driven Design patterns

Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.





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

> Any MVC + Spring

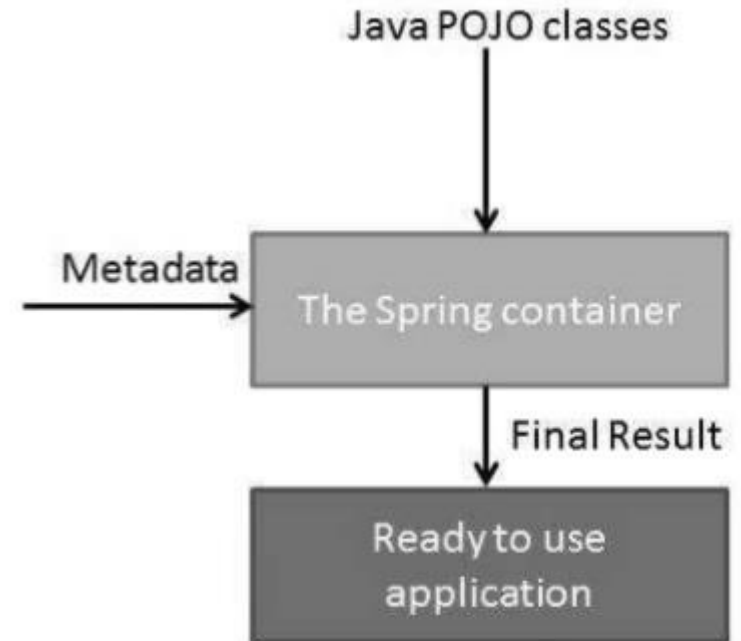
1. Pojo(no Container)
2. Ejb 2.x(HW Container)
 - Session Bean
 - Mdb
3. Pojo + LW Container
 - Spring
 - Microcontainer
 - Xwork
4. Ejb3.0

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

+ Spring Templates

What is a container?

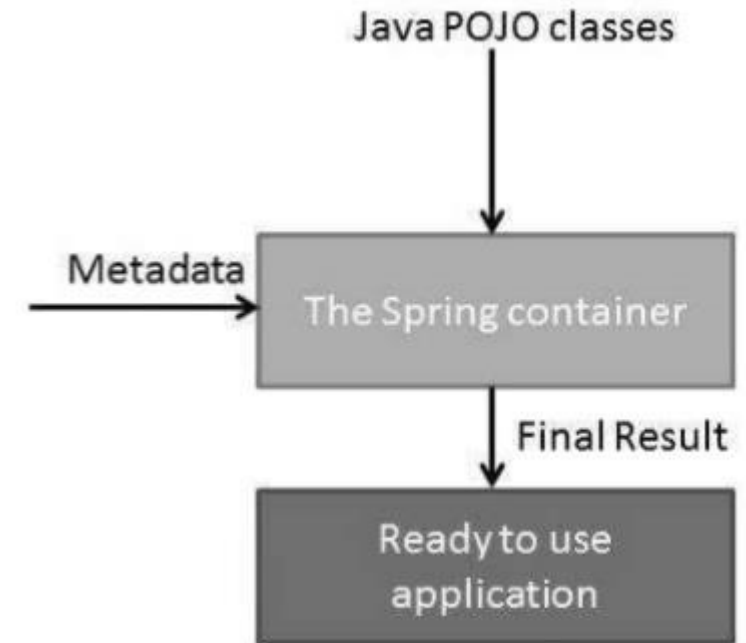
- ✓ The container will create the objects,
- ✓ wire them together,
- ✓ configure them,
- ✓ and manage their complete life cycle from creation till destruction.



The container gets its instructions on

- ✓ what objects to instantiate,
- ✓ configure,
- ✓ and assemble by reading the configuration metadata provided.
- ✓ The configuration metadata can be represented either by XML or Annotation.

- ❑ Apache Tomcat is a Servlet Container.
- ❑ Weblogic/WebSphere/JBoss provides EJB Container
- ❑ Spring is a POJO container.



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.

Spring XML Configuration

```
package demo.web;
```

```
Class AccountDaoJdbc implements AccountDao  
{
```

```
DataSource dataSource;
```

```
Public void setDataSource(DataSource datasource)  
{  
dataSource = datasource;  
}  
..  
}
```

```
<bean id = "accountDao" class="demo.web.AccountDaoJdbc">  
  <property name="datasource">  
    <ref bean="dataSource"/>  
  </property>  
</bean>
```


Spring Annotation Configuration

```
package demo.web;
```

```
@Repository("accountDao")
```

```
Class AccountDaoJdbc implements AccountDao
```

```
{
```

```
@Autowired
```

```
DataSource dataSource;
```

```
Public void setDataSource(DataSource datasource)
```

```
{
```

```
dataSource = datasource;
```

```
}
```

```
..
```

```
}
```

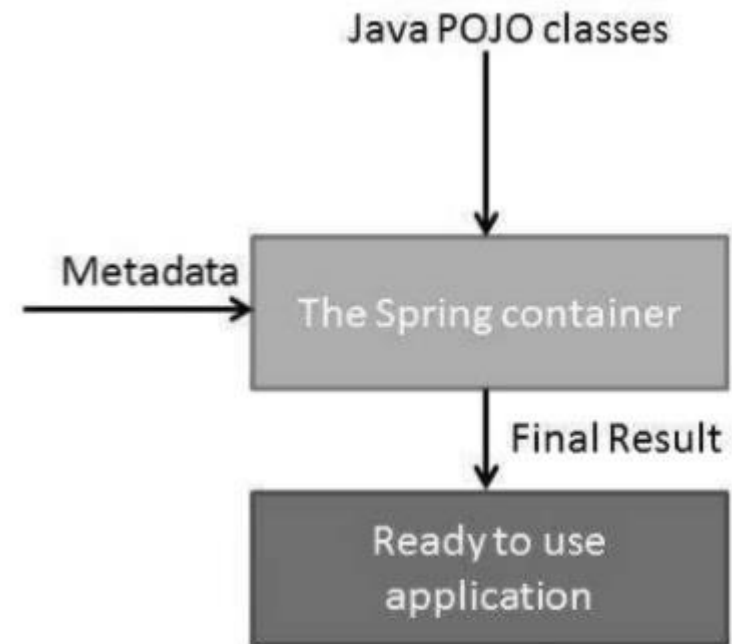
```
<bean id="accountDao" class="demo.web.AccountDaoJdbc">  
  <property name="dataSource">  
    <ref bean="dataSource"/>  
  </property>  
</bean>
```

Lifecycle Management

Bean Managed Life cycle

~~AccountBean acc = new AccountBean();
acc.setAmt(5000);
....
acc=null;~~

Container Managed Life cycle



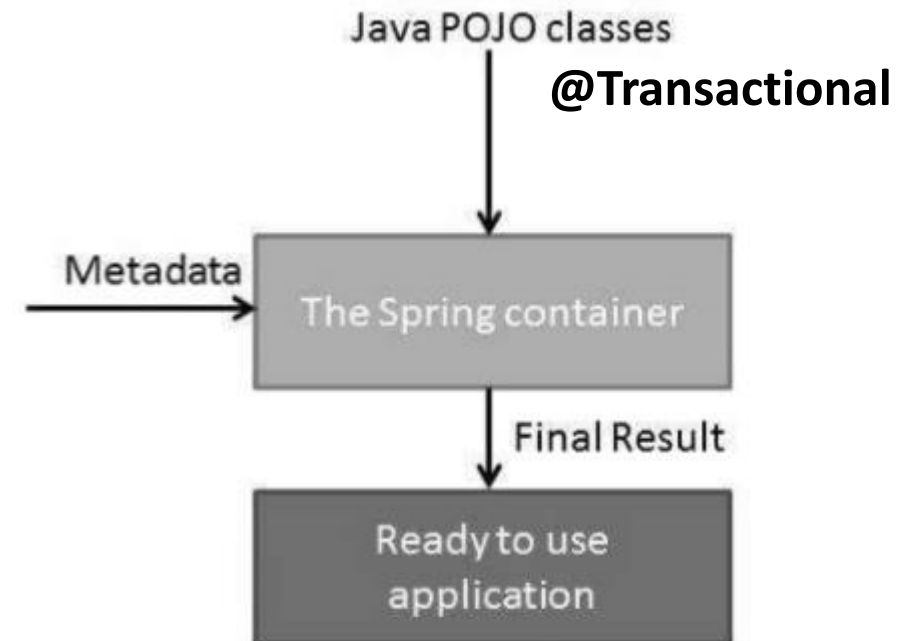
ApplicationContext ctx =
AccountBean acc = ctx.getBean("account");

Container Managed Services

Bean managed transactions

~~Transaction tx=
tx.begin;
database operations;
tx.commit() / tx.rollback();~~

Container managed transactions



Spring Framework Features:

1. Core Container - IOC, DI & AOP
Application Context (Object creation – Singleton/Prototype)
Dependency Injection (Object Graph Creation)
2. Data Access Layer (Template pattern)
3. Spring MVC & REST
4. Other features like Security, Messaging, Spring Integration, Spring Batch, Spring Cloud etc.,

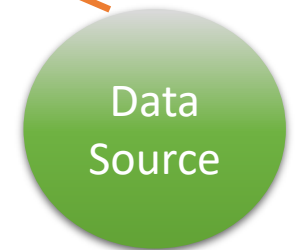
Presentation Tier



Business Tier



Data Tier



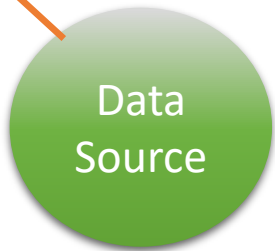
Presentation Tier



Business Tier



Data Tier



@RestController

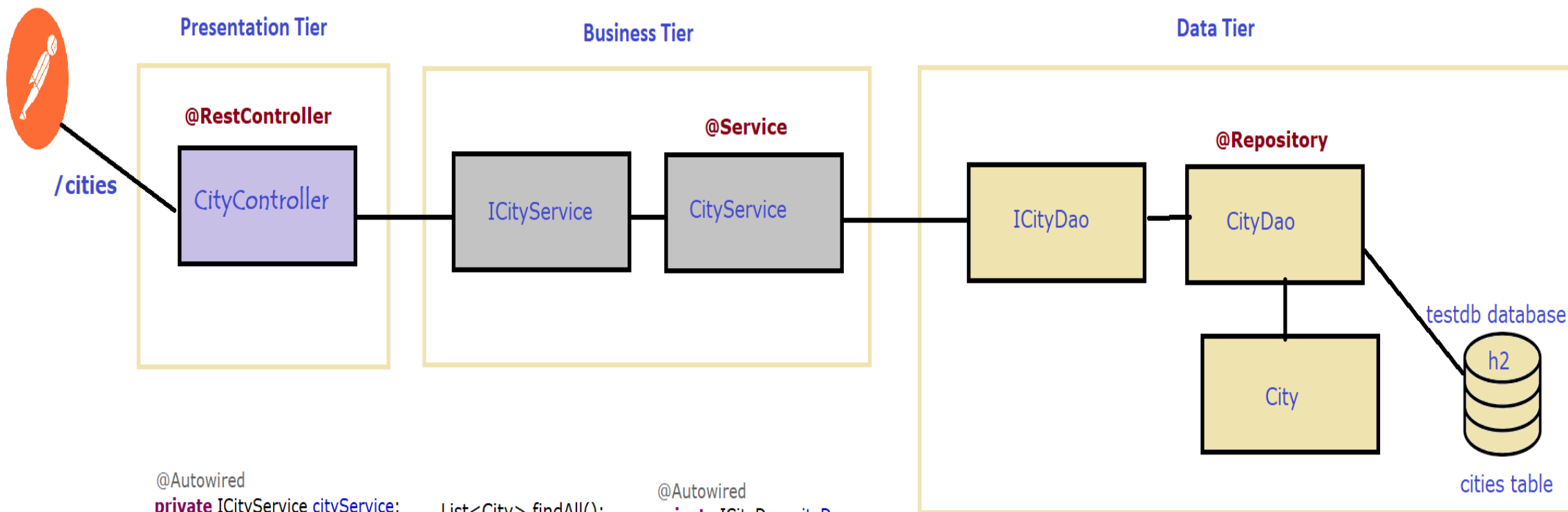
```
Class AccountController
{
  @Autowired
  AccountService accService;
  ....
}
```

@Service

```
Class AccountServiceBean
  implements AccountService
{
  @Autowired
  AccountDao accountDao;
  ....
}
```

@Repository

```
Class AccountDaoJdbc
  implements AccountDao
{
  @Autowired
  DataSource dataSource;
  ....
}
```



```

@Autowired
private ICityService cityService;

@RequestMapping("/cities")
public List<City> findCities() {
    return cityService.findAll();
}

```

```

List<City> findAll();
City findById(Long id);

```

```

@Autowired
private ICityDao cityDao;

@Override
public List<City> findAll() {
    return cityDao.findAllCities();
}

```

```

List<City> findAllCities();
City findById(Long id);

```

```

@Autowired
private JdbcTemplate jtm;

public List<City> findAllCities() {
    String sql = "SELECT * FROM cities";
    return jtm.query(sql, new
        BeanPropertyRowMapper<>(City.class));
}

```

JAX-RS Vs Spring MVC[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

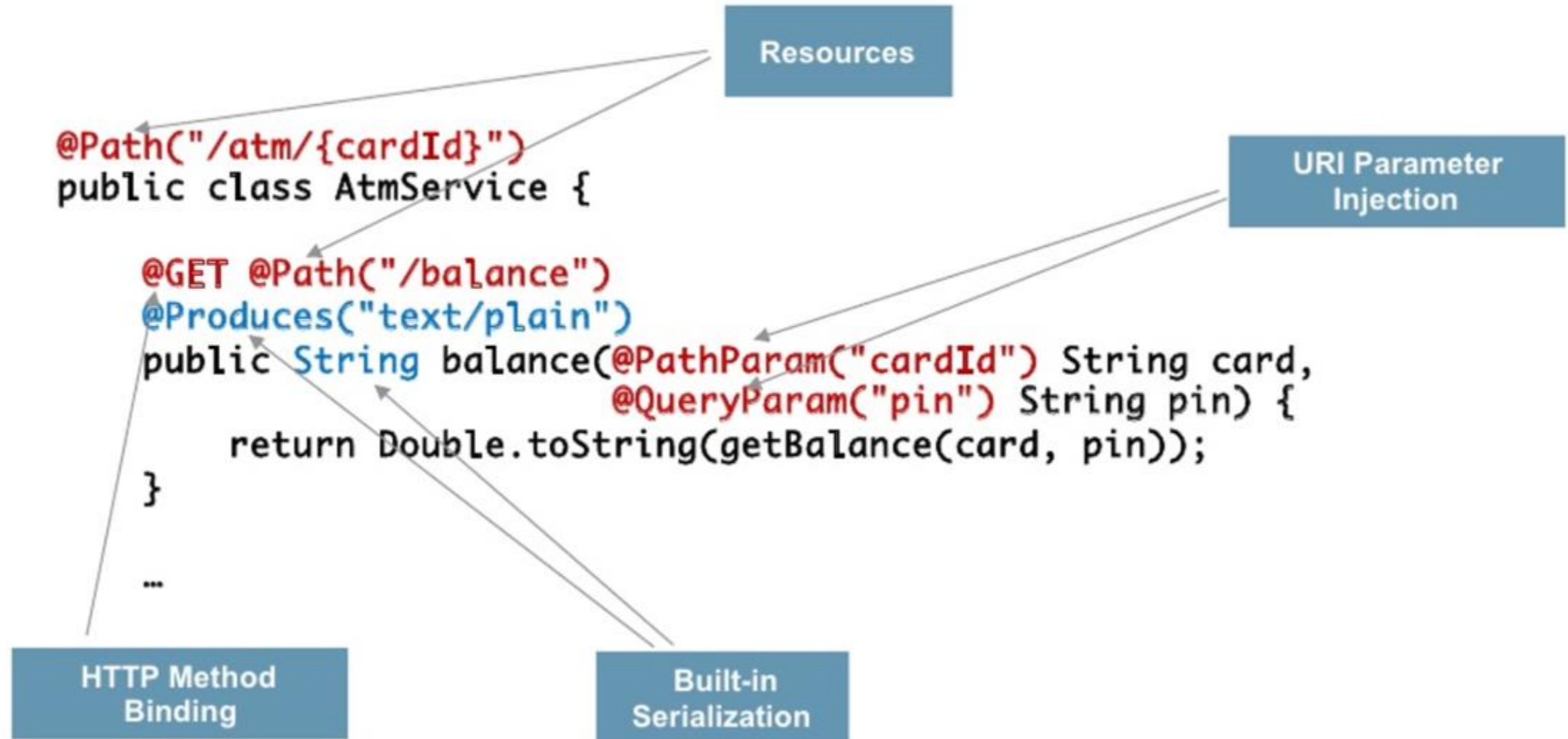
The Spring Framework is a full framework that allows us to create Java enterprise applications.

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.


Example: JAX-RS API



Example: JAX-RS API (contd.)

```
...  
  
@POST @Path("/withdrawal")  
@Consumes("text/plain")  
@Produces("application/json")  
public Money withdraw(@PathParam("card") String card,  
                      @QueryParam("pin") String pin,  
                      String amount){  
    return getMoney(card, pin, amount);  
}  
}
```

Custom Serialization



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

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`

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.

1. Maps HTTP Request with REST Resource based on 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 request, 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.

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`

Spring MVC Annotations

@Controller
@RequestMapping
@RequestParam
@PathVariable
@RequestBody &
@ResponseBody
@RestController

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

Java API for RESTful Web Services

Example: JAX-RS API

```
@Path("/atm/{cardId}")
public class AtmService {

    @GET @Path("/balance")
    @Produces("text/plain")
    public String balance(@PathParam("cardId") String card,
        @QueryParam("pin") String pin) {
        return Double.toString(getBalance(card, pin));
    }

    ...
}
```

Resources

URI Parameter Injection

HTTP Method Binding

Built-in Serialization

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.findByName(firstName);
    return contact;
}
```

@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 its 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());

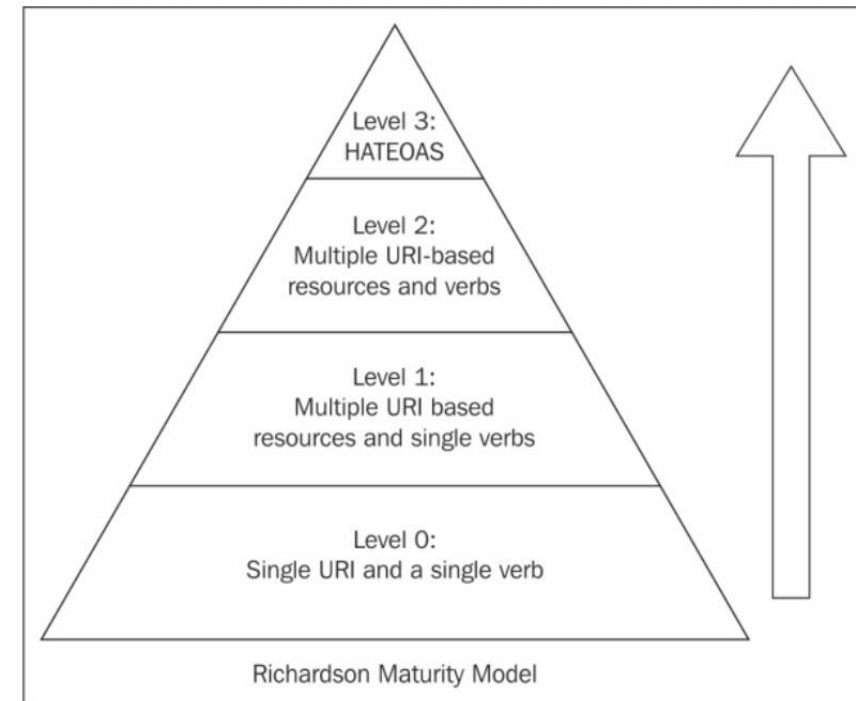
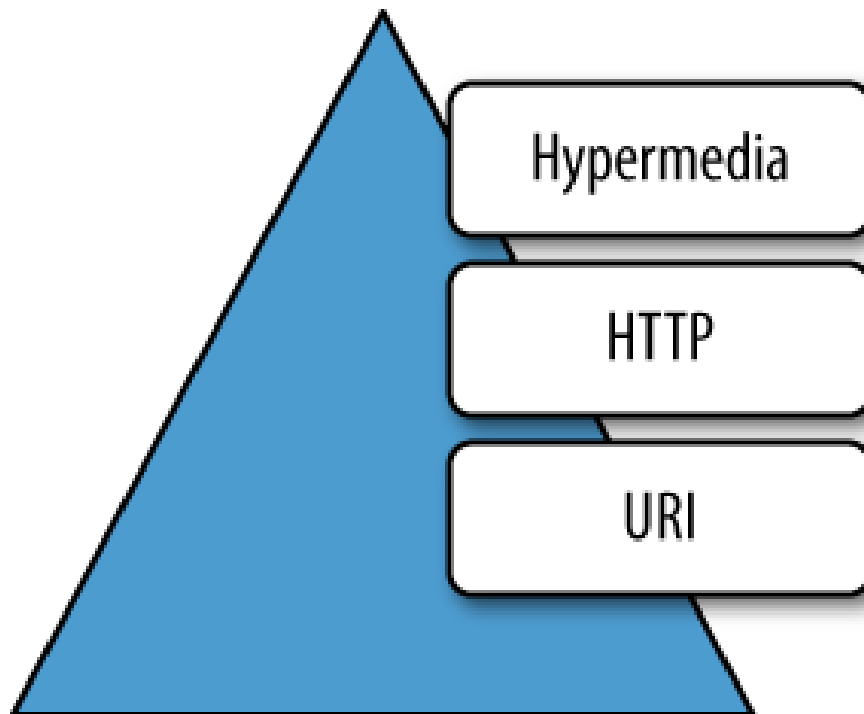
        System.out.println(target.path("hello").request().accept(MediaType.TEXT_PLAIN).get(String.class));
        System.out.println(target.path("hello").request().accept(MediaType.TEXT_XML).get(String.class));
        System.out.println(target.path("hello").request().accept(MediaType.TEXT_HTML).get(String.class));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

    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.