REST WEB SERVICES

FRICTIONLESS FLOW OF INFORMATION

REST Web Services

- REST = **RE**presentational **S**tate **T**ransfer
- REST is an architectural style consisting of a coordinated set of architectural constraints
- First described in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine.
- RESTful is typically used to refer to web services implementing a REST architecture.
- Alternative to other distributed-computing specifications such as SOAP.
- Simple HTTP client/server mechanism to exchange data
- Everything – the UNIVERSE is available through a URI
- Utilizes HTTP: GET/POST/PUT/DELETE operations

Architectural Constraints

- Client–server
  - Separation of concerns. A uniform interface separates clients from servers.
- Stateless
  - The client–server communication is further constrained by no client context being stored on the server between requests.
- Cacheable
  - Basic WWW principle: clients can cache responses as well as servers across the network
- Layered system
  - A client cannot necessarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- Uniform interface
  - Individual resources are identified in requests, i.e.,using URIs in web-based REST systems.

RESTful Web Services

- No significant tools required to interact with the Web service
- Short learning curve
- Efficient REST can use concise message formats
- Fast (no extensive processing required)
- Close to other Web technologies in design philosophy
- Explosive growth in commercial end user applications
- ***Does NOT follow a prescribed standard beyond HTTP spec***

RESTful API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://example.com /resources	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
E http:/ /res	**Retrieve** a representation of appropriate internet media type.	Replace the	Not generally used. new entry in it.	ressed the)

POST means "create new" as in "Here is the input for creating a customer".
PUT means “create OR replace if already exists" as in "Here is the data for user 5".
PUT is Idempotent

Create support via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists.
[Not possible with DB generated PK]

Idempotent means that multiple calls with the same operation doesn’t change the representation

JSON (JavaScript Object Notation)

- {
- "productId":"P1235",
- "name":"Dell Inspiron",
- "unitPrice":700,
- "description":"Dell Inspiron 14-inch Laptop (Black) with 3rd Generation Intel Core processors",
- "manufacturer":"Dell",
- "category":"Laptop",
- "unitsInStock":1000,
- "unitsInOrder":0,
- "discontinued":false,
- "condition":null
- }

Main Point

REST is defined by architectural constraints. It is able to access information through the ubiquitous URI. Everything on the web is available through a URI.

Likewise, everything in creation is known through understanding and experience of the Unified Field of Consciousness

Spring Rest Web Service Technologies Using MVC REST-style Controller

- “Re-uses” @Controller
- Essentially means receive & send the content directly as the message body instead of structuring HTML pages.

We are **NOT** using HTML

We are using well-formed XML OR JSON

Spring support is based on the

@REQUESTBODY & @RESPONSEBODY annotations

@ResponseStatus(value = HttpStatus.***NO_CONTENT)***

For deletes, creates, updates…

RequestBody & ResponseBody

- @ResponseBody
  - Spring framework uses the "Accept" header of the request to decide the media type to send to the client
- @RequestBody
  - Spring framework will use the “*Content-Type*” header to determine the media type of the Request body received.

  - To get XML, MIME media type = "application/xml"
  - To get JSON, MIME media type = "application/json “

RESTful Web Service Controller

```
@RequestMapping("/products")
Class Product
No Request OR Response Data
@RequestMapping(value = "/{productId}", method = RequestMethod.DELETE)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
 public void deleteItem(@PathVariable("productId") String productId,
                                    HttpServletRequest request) {
```

- Response Data
- `@RequestMapping("", method = RequestMethod.GET)`

```
public @ResponseBody List<Product> getRestProduct(){
```

- Request & Response Data
- `@RequestMapping("", method = RequestMethod.POST)`

```
public @ResponseBody Product saveRestProduct(@RequestBody Product product){
```

Spring MVC Rest Controller Annotation Alternatives

```
@RestController
@RequestMapping({"/books"})
public class BookController {

@Autowired
BookService bookService;

@GetMapping("")
public List<Book> findAll() {
return bookService.findAll();
}

@PostMapping("")
public void add( @RequestBody Book book) {
bookService.save(book);
return ;
}
```

@RestController
“automatically” assumes return object is REST related [implicit @ResponseBody]

Uses composed annotations for REST - For example @GetMapping("") ==
@RequestMapping(value="", method = RequestMethod.GET).

Spring MVC [Continued]

```
• @GetMapping(value= "/{title}")
• public Book findOne(@PathVariable("title") String title) {
     return bookService.findOne(title);
•  }

• @DeleteMapping("/{title}")
• public  void delete(@PathVariable("title") String title) {
   bookService.delete(title);
•    return;
• }

• @PutMapping(value= "/{title}")
• public void update(@PathVariable("title") String title, @RequestBody
  Book updateBook) {
     bookService.delete(title);
     bookService.update(updateBook);
•    return;
• }
```

Basic and Digest Authentication

- **Basic authentication**
  - Handshake based on HTTP headers**
  - Transmits username/password as “plain text”
    - Base64 encoding
  - Used in conjunction with SSL-HTTPS**
  - Used with form-based authentication**
  - Secure data at rest
- **Digest Authentication**
  - Transmits encrypted username/password
  - “Double” handshake to get hash “seed”
  - More complex – more vulnerable

**** Web based**

Restful Web Service with Spring Security

- **Restful web service is stateless**
- **No HTTP sessions**
- **Re-authenticate on every request**

Better Solution: OAuth2

- **Server Configuration:**

```
 <!-- Stateless RESTful services use BASIC authentication -->
<security:http create-session="stateless"
                       pattern="/**" use-expressions="true">
     <security:intercept-url pattern="/**" access="hasRole('ROLE_ADMIN')"/>
     <security:http-basic/>
     <security:csrf disabled="true"/>
 </security:http>
```

Basic authentication transmits as plain text need HTTPS

Cross Site Request Forgery service - for non-browser clients disable CSRF protection

Client Side Authentication

CREATE HTTP Authorization header

```
String auth = username + ":" + password;
byte[] encodedAuth = Base64.encodeBase64(
   auth.getBytes(Charset.forName("US-ASCII")) );
String authHeader = "Basic " + new String(encodedAuth)
```

Declares Basic & includes encoded credentials

HEADER:

Authorization: Basic aHR0cHdhdGNoOmY=

Spring Rest Template
Client side access

- “Conventional” use of a REST web service is programmatic:

  NOT Browser based…

- Spring provides a convenient template class:

  RestTemplate

- Simplified Interaction with RESTful services

  Often a one-line incantation.

  Can bind data to custom domain types

  Spring RestTemplate

Also Convenience Class often used with RestTemplate :

HttpEntity

Represents HTTP Payload [Request & Response]

Contains Http headers & Body

Spring HttpEntity

Another Convenience Class:

HttpHeaders

Contains Map of HTTP headers   Spring HttpHeaders

RestTemplate Example

- **Setup HttpHeaders Example [RestHttpHeader.java]:**
- HttpHeaders requestHeader = **new** HttpHeaders();
- requestHeader.**setAccept(Collections.*singletonList(MediaType.APPLICATION_JSON))***
- requestHeader.**setContentType(MediaType.*APPLICATION_JSON);***
- requestHeader.**set("Authorization", authHeader);**

Create HttpEntity with only headers

- **Setup HttpEntity Example:**
- HttpEntity **headerOnly** = new HttpEntity(requestHeader)
- HttpEntity **headerWithBody** =
-                new HttpEntity<Member>(member,requestHeader);

Create HttpEntity with headers and body.

- **RestTemplate Examples [e.g., ProductRestServiceImpl.java]:**
- String baseUrl = "http://localhost:8080/MemberRest/members";
- String baseUrlExtended = baseUrl + "/";
-
- restTemplate.exchange(baseUrlExtended + index, HttpMethod.GET, **headerOnly** ,
-                                     Member.class).getBody());
- restTemplate.exchange(baseUrl, HttpMethod.***GET***, **headerOnly** , ***Member[].class***);
- restTemplate.postForObject(baseUrl, **headerWithBody**, Member.**class**);

Main Point

The Spring framework makes the transition to RESTful web services smooth though the flexible & adaptable design of Spring MVC controllers.

The Structure of Life is flexible & adaptable.

REST API Conventions

Google API

- The resource name is organized hierarchically using collection IDs and resource IDs, separated by forward slashes. If a resource contains a sub-resource, the sub-resource's name is formed by specifying the parent resource name followed by the sub-resource's ID - again, separated by forward slashes.

Resources are **NOUNS** not verbs
members .vs. getMembers

endpoints are plural
members .vs. member

- **Relationships**
- GET /orders/12/items - Retrieves list of addresses for member 12
- GET /orders/12/items /5/products/1 - Retrieves product[s] for item 5 of order 12
- POST /orders/12/items - Creates a new item associated with order 12
- PUT /orders/12/items/5 - Updates item #5 for order #12
- DELETE /orders/12/items/5 - Deletes address #5 for member #12

Microsoft API guidelines

Microsoft API

In more complex systems, there can be URIs that enable a client to navigate through several levels of relationships:

/orders/12/items/5/products/1

The complexity level can be difficult to maintain and is inflexible

Keep URIs relatively simple.

The preceding query can be replaced with the URI

/orders/12 /items

to find the items associated with order 12, and then

/items/5/products

to get products associated with item 5.

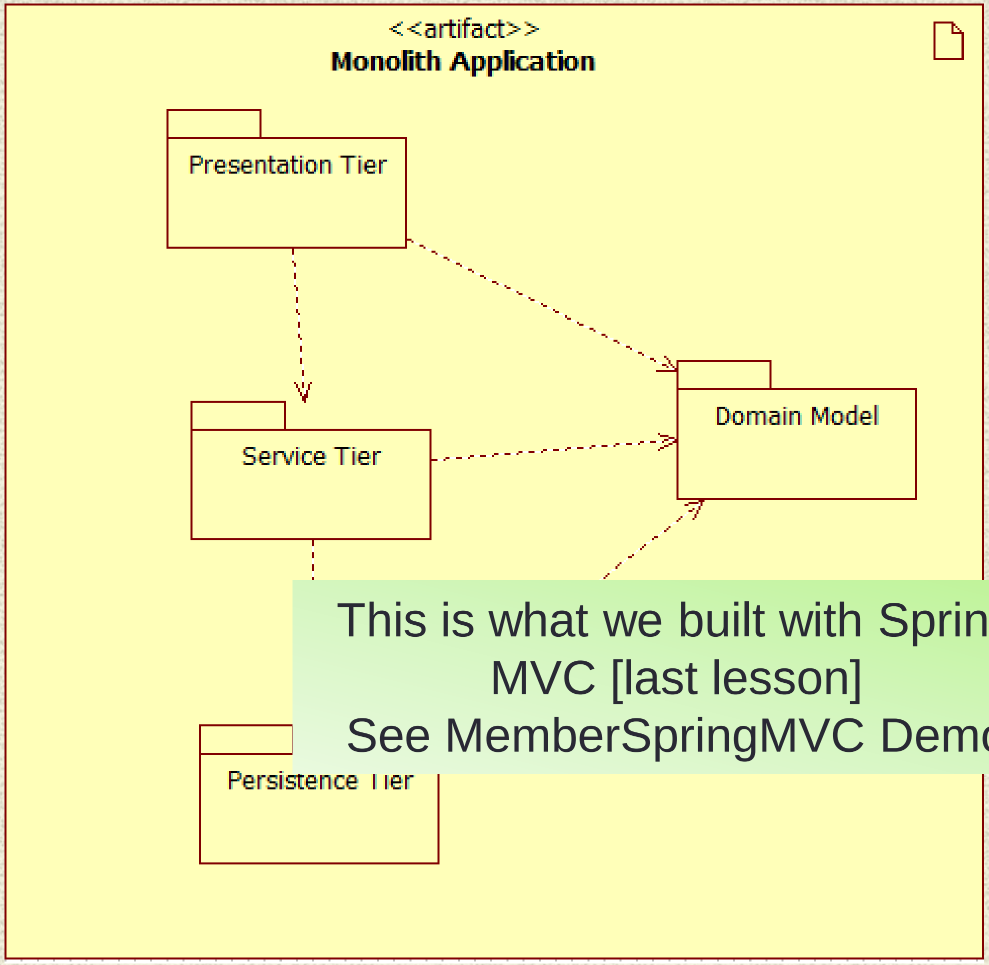
/products/1

to get product

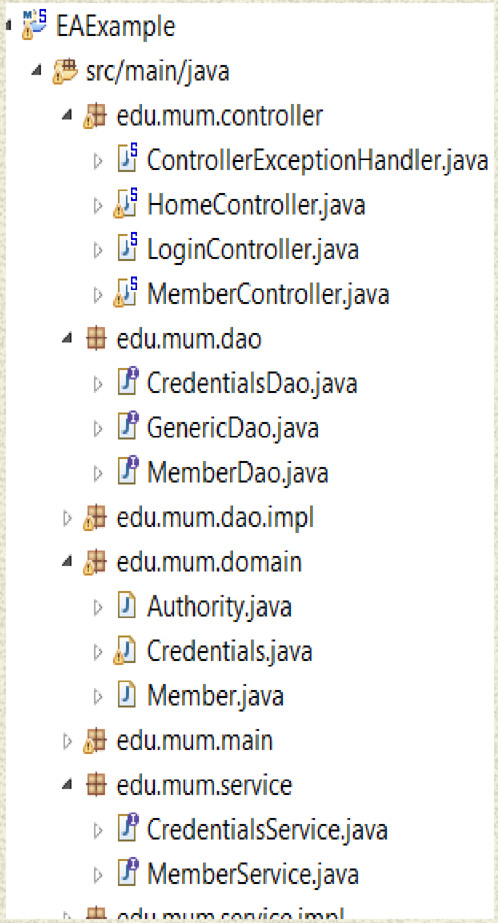
Microsoft Tip:

Avoid requiring resource URIs more complex than *collection/resource/collection*.

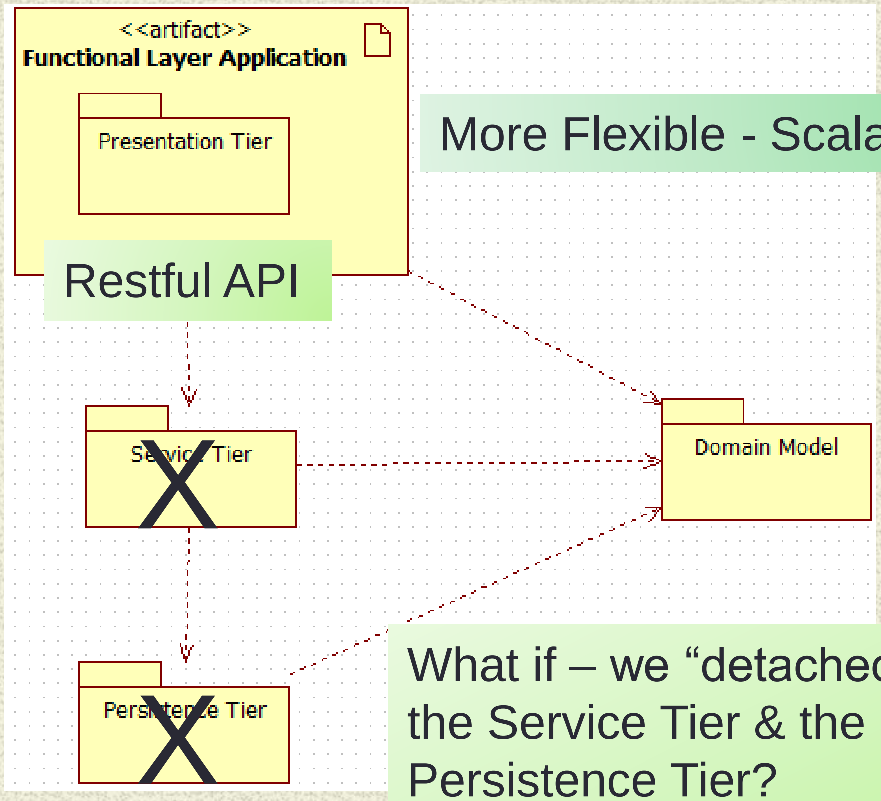
Monolith N-Tier



This is what we built with Spring MVC [last lesson]
See MemberSpringMVC Demo

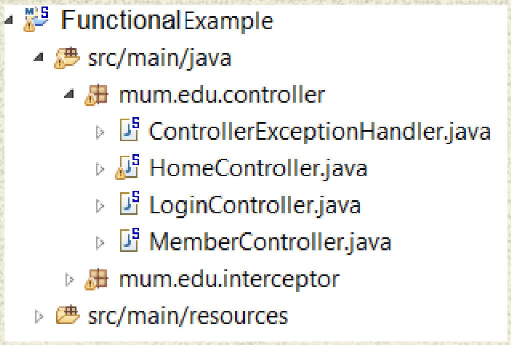

```
EAExample
  src/main/java
    edu.mum.controller
      ControllerExceptionHandler.java
      HomeController.java
      LoginController.java
      MemberController.java
    edu.mum.dao
      CredentialsDao.java
      GenericDao.java
      MemberDao.java
    edu.mum.dao.impl
    edu.mum.domain
      Authority.java
      Credentials.java
      Member.java
    edu.mum.main
    edu.mum.service
      CredentialsService.java
      MemberService.java
    edu.mum.service.impl
```

Functional N-Tier

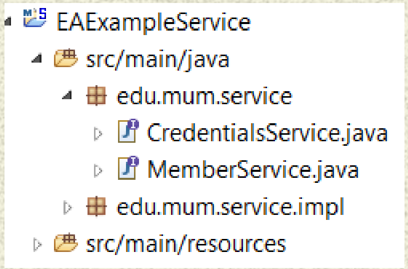


More Flexible - Scalable

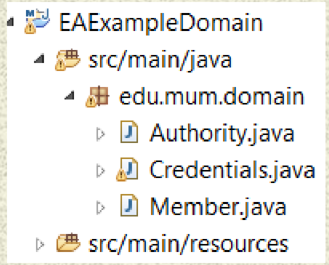
Restful API



```
FunctionalExample
  src/main/java
    mum.edu.controller
      ControllerExceptionHandler.java
      HomeController.java
      LoginController.java
      MemberController.java
    mum.edu.interceptor
  src/main/resources
```

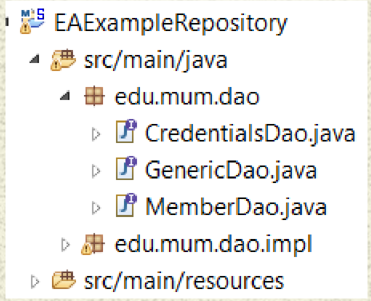


```
EAExampleService
  src/main/java
    edu.mum.service
      CredentialsService.java
      MemberService.java
    edu.mum.service.impl
  src/main/resources
```



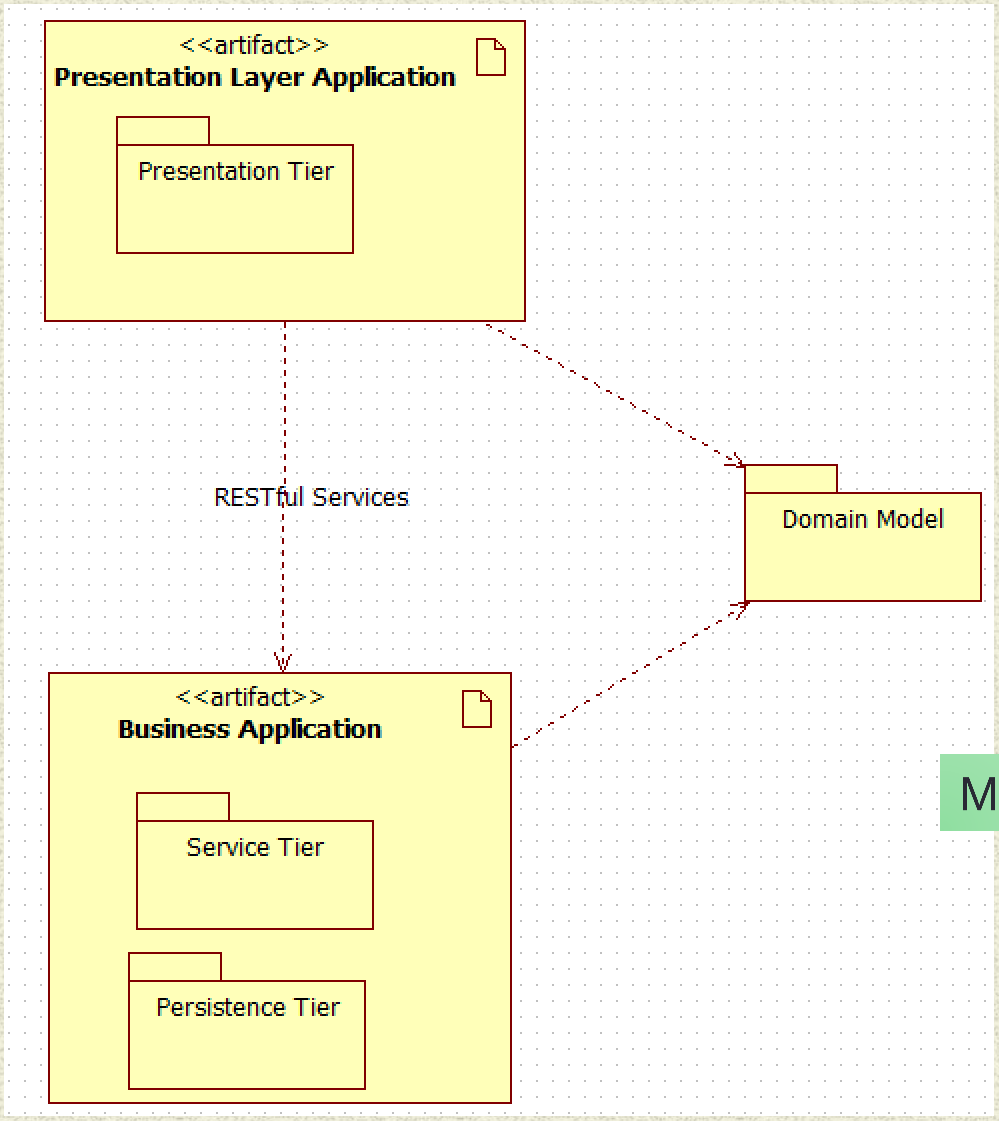
```
EAExampleDomain
  src/main/java
    edu.mum.domain
      Authority.java
      Credentials.java
      Member.java
  src/main/resources
```

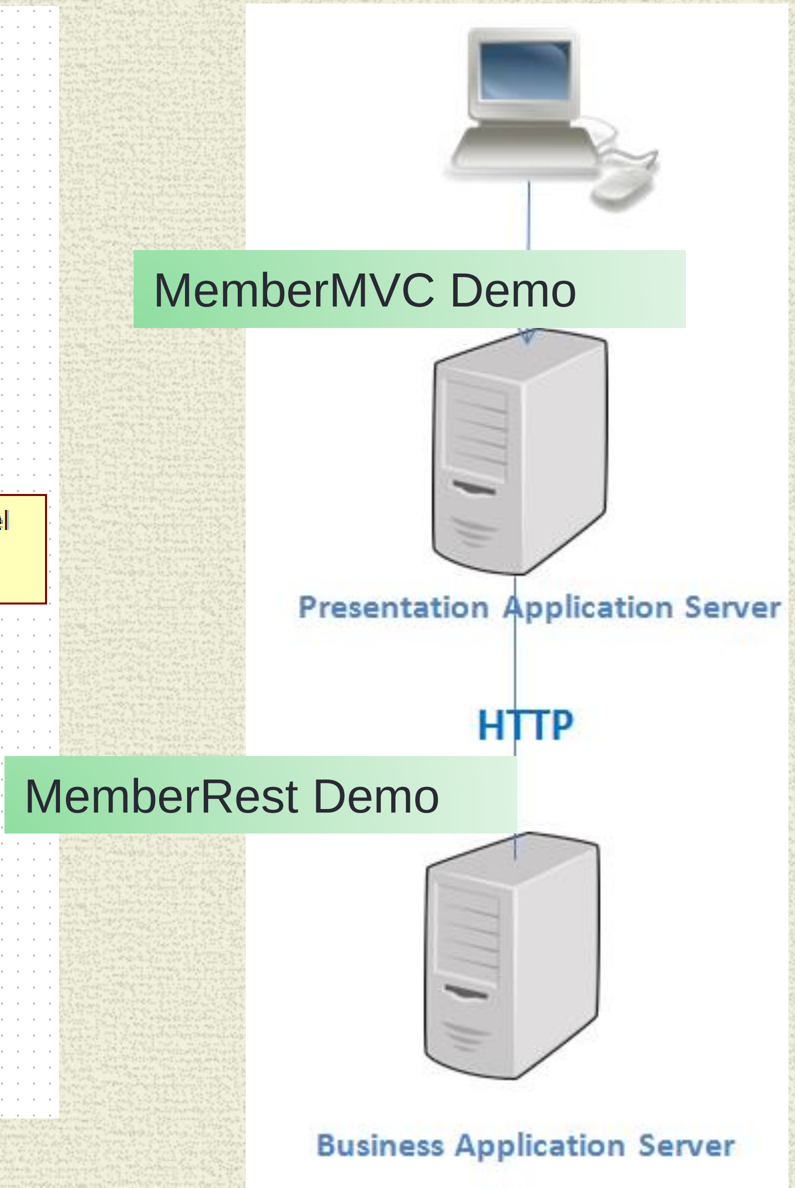
What if – we “detached” the Service Tier & the Persistence Tier?
And create a Rest based business application…



```
EAExampleRepository
  src/main/java
    edu.mum.dao
      CredentialsDao.java
      GenericDao.java
      MemberDao.java
    edu.mum.dao.impl
  src/main/resources
```

RESTful Business Layer





JSON Lazy Loading

Hibernate lazy initialization exception results from attempting JSON serialization on

JsonMappingException: failed to lazily initialize

```
    @OneToMany(fetch=FetchType.LAZY)
   private List<Address> addresses = new ArrayList<Address>();
```

SOLUTION:

Configure HibernateAwareObjectMapper

[Registers Hibernate4Module] **

Returns null if Lazy loading OR Collection if Collection hydrated

```
<mvc:message-converters>
• <bean class=
   "org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
•       <property name="objectMapper">
•               <bean class="edu.mum.rest.HibernateAwareObjectMapper" />
•       </property>
• </bean>
</mvc:message-converters>
```

Postman Demo:
http://localhost:8080/MemberRest/products/1
http://localhost:8080/MemberRest/products/1/categories

- ** Hibernate4Module : add-on module forJackson JSON processor which handles Hibernate Lazy Loading

[Some] JSON Annotations

@JsonIgnoreProperties

Class level annotation - list of properties properties to be excluded.
EXAMPLES:

In Address class:

```
@JsonIgnoreProperties(value="member")            Always ignore in Address
public class Address {
```

OR

In Member class:

```
@JsonIgnoreProperties(value="member")            Ignore when accessing Member Address List
private List<Address> addresses = new ArrayList<Address>();
```

@JsonIgnore

Field level annotation - properties to be excluded are marked one by one.

EXAMPLE:

In Address class:

```
 @JsonIgnore
  private Member  member;
```

use @JsonView if you want to dynamically determine which field(s) to skip

JSON Annotations [Cont.]

@JsonManagedReference && @JsonBackReference [Bidirectional]

@JsonManagedReference is the “front” part of reference - the one that gets serialized normally.
@JsonBackReference is the back part of reference - it will be omitted from serialization.

Example:

In Member class:

```
@JsonManagedReference()
 private List<Address> addresses = new ArrayList<Address>();
```

In Address class:

```
@JsonBackReference()
 private Member  member;                    Always ignore
```

@JsonIdentityinfo   [Bidirectional]

Serialize the first instance as full object [JSON object identity], subsequent references use reference values.

Examples:

JSON generated ID

```
@JsonIdentityInfo(generator=ObjectIdGenerators.IntSequenceGenerator.class, property="@jid")
public class UserCredentials {
```

Use Class ID

```
@JsonIdentityInfo(generator=ObjectIdGenerators.PropertyGenerator.class, property="id")
public class Member {
```

JSON Bidirectional Circular Dependency

For Jackson to work well, one of the two sides of a bidirectional relationship should *NOT* be serialized

In order to avoid an infinite recursive loop that causes a stackoverflow error.

Example Solutions….

NOTE: Postman gives access to the JSON content

JsonManagedReference /JsonBackReference

```
public class Member  {
  @JsonManagedReference
  UserCredentials userCredentials;
```

GET for Member: **[gets UserCredentials]**

```
{
    "id": 1,
    "firstName": "Curious",
    "lastName": "George",
    "age": 12,
    "title": "Boy Monkey",
    "memberNumber": 8754,
    "userCredentials": {
        "username": "admin",
        "password": "admin",
        "verifyPassword": null,
        "enabled": true,
        "authority": []
    },
    "addresses": null
```

```
public class UserCredentials {
    @JsonBackReference
    private Member member;
```

GET for UserCredentials: **[No Member]**

```
{
    "username": "admin",
    "password": "admin",
    "verifyPassword": null,
    "enabled": true,
    "authority": []
},
```

See MemberRestJSON Demo

JsonIgnoreProperties

```
 public class Member  {
@JsonIgnoreProperties
                    (value="member“)
   UserCredentials userCredentials;
```

GET for Member: [gets UserCredentials]

```
{ "id": 1,
  "firstName": "Curious",
  "lastName": "George",
  "age": 12,
  "title": "Boy Monkey",
  "memberNumber": 8754,
  "userCredentials": {
        "username": "admin",
        "password": "admin",
        "verifyPassword": null,
        "enabled": true,
        "authority": null
  },
    "addresses": null
},
```

Member

No reference to relationship

```
public class UserCredentials {
 @JsonIgnoreProperties
                (value="userCredentials“)
  private Member member;
```

GET for UserCredentials: [gets Member]

```
{
  "username": "admin",
   "password": "admin",
   "verifyPassword": null,
   "enabled": true,
   "member": {
      "id": 1,
      "firstName": "Curious",
      "lastName": "George",
      "age": 12,
      "title": "Boy Monkey",
      "memberNumber": 8754,
      "addresses": null
   },
   "authority": null
}
```

UserCredentials

See MemberRestJSON Demo

Also See Product-Category in Demo

JsonIdentityInfo

```
@JsonIdentityInfo(generator=
ObjectIdGenerators
.PropertyGenerator.class,property="id"
public class Member {
UserCredentials userCredentials;
```

GET for Member: [gets UserCredentials]

```
    "id": 1,
    "firstName": "Curious",
    "lastName": "George",
    "age": 12,
    "title": "Boy Monkey",
    "memberNumber": 8754,
    "userCredentials": {
        "@id": 1,
        "username": "admin",
        "password": "admin",
        "verifyPassword": null,
        "enabled": true,
        "member": 1,
        "authority": null   },
    "addresses": null
```

Reference to relationship

```
@JsonIdentityInfo(generator=
  ObjectIdGenerators.
 IntSequenceGenerator.class,property="@id"
public class UserCredentials {
    private Member member;
```

GET for UserCredentials: [gets Member]

```
    "@id": 1,
     "username": "admin",
     "password": "admin",
     "verifyPassword": null,
     "enabled": true,
     "member": {
        "id": 1,
        "firstName": "Curious",
        "lastName": "George",
        "age": 12,
        "title": "Boy Monkey",
        "memberNumber": 8754,
        "userCredentials": 1,
        "addresses": null  },
     "authority": null
```

Also See Product-Category in Demo

Bidirectional Considerations

Domain Driven Design:

Reduce complexity by identifying a traversal direction **

(RE: avoid bidirectional associations if possible)

Removes coupling in domain model

Simplifies code in domain model

Removes circular dependencies

** Traversing the other direction is still possible by querying the underlying persistence system.

See demo OneToManyBiAsUni

Spring MVC Rest Web Service Alternative: Use JAX-RS

- Java API for RESTful Web Services (**JAX-RS**)
- Targeted solely for implementing REST APIs
- Implements REST architectural pattern**
- "Jersey" is the reference implementation
- Light weight
- Integrates into Spring

OBVIOUSLY JAVA Specific

- JAX-RS Specification
- ** Remember NOT a STANDARD…a "pattern"

Jersey integrated into Spring

```
• @Component
• @Path("/products")
• public class ProductRestService {
•        @GET
•        @Produces(MediaType.APPLICATION_JSON)
             public List<Product> getProducts()
             { return productService.findAll();   }

             @GET
             @Path("{id: \\d+}")
             @Produces(MediaType.APPLICATION_JSON)
             public Product getProductById(@PathParam("id") Long id)
             {return productService.findOne(id);   }
•
             @POST
•            @Consumes({ MediaType.APPLICATION_JSON })
•            @Produces(MediaType.APPLICATION_JSON)
             public Product saveProduct(Product product)
                     {return productService.save(product);  }
```

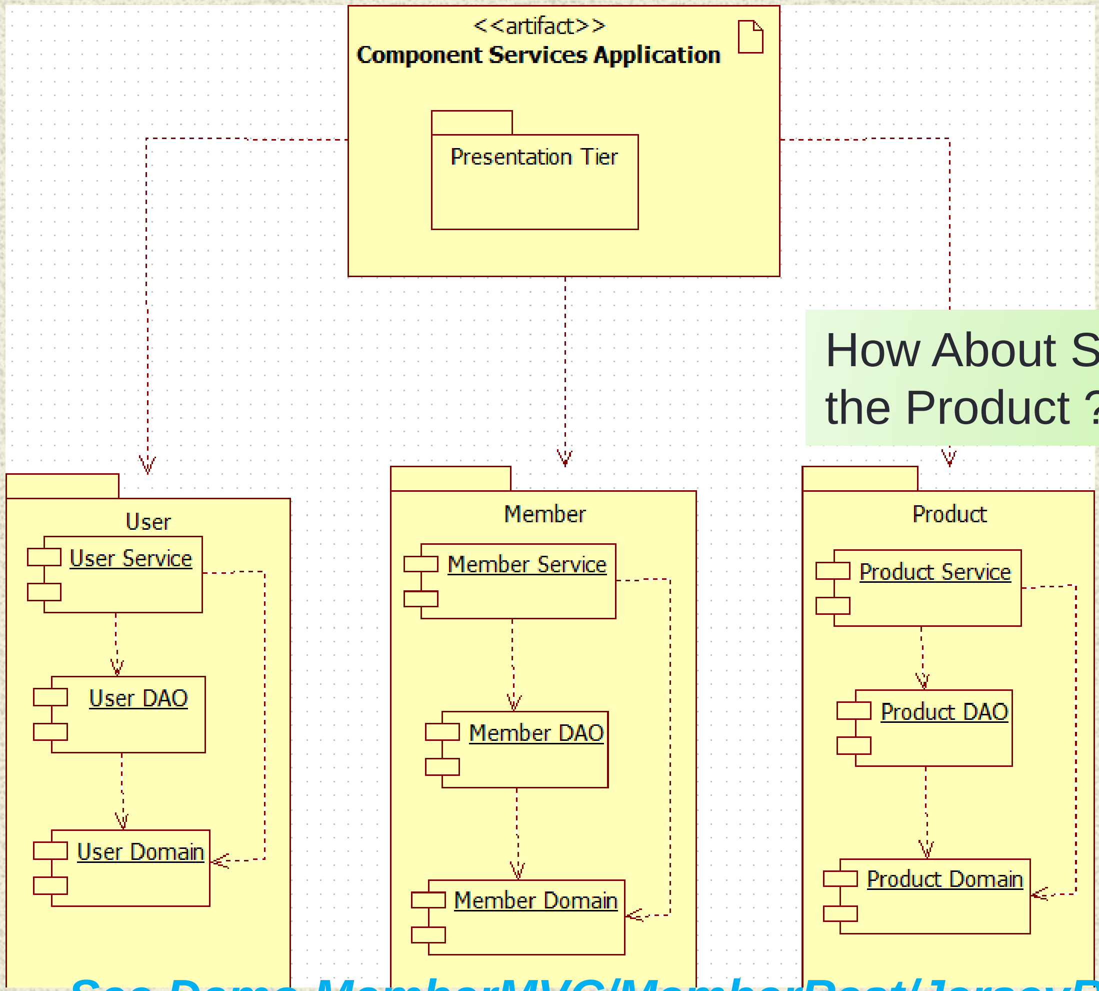
@Path .vs. “MVC” @RequestMapping

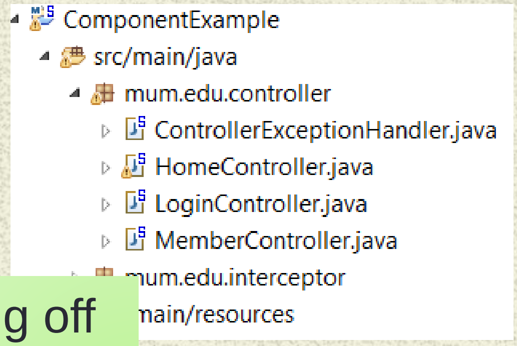
Payload type

“REST” “commands”

Regular Expression: digit only

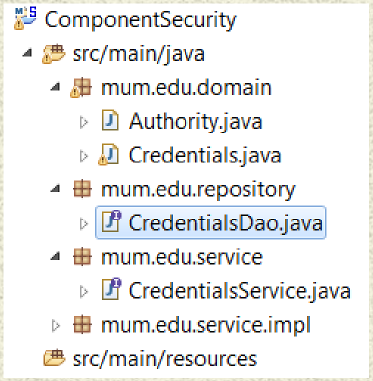
Component N-Tier with Jersey



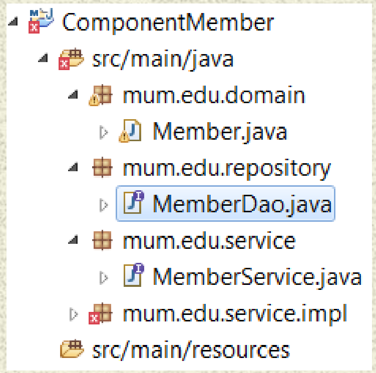


```
ComponentExample
  src/main/java
    mum.edu.controller
      ControllerExceptionHandler.java
      HomeController.java
      LoginController.java
      MemberController.java
    mum.edu.interceptor
  src/main/resources
```

How About Splitting off the Product ?



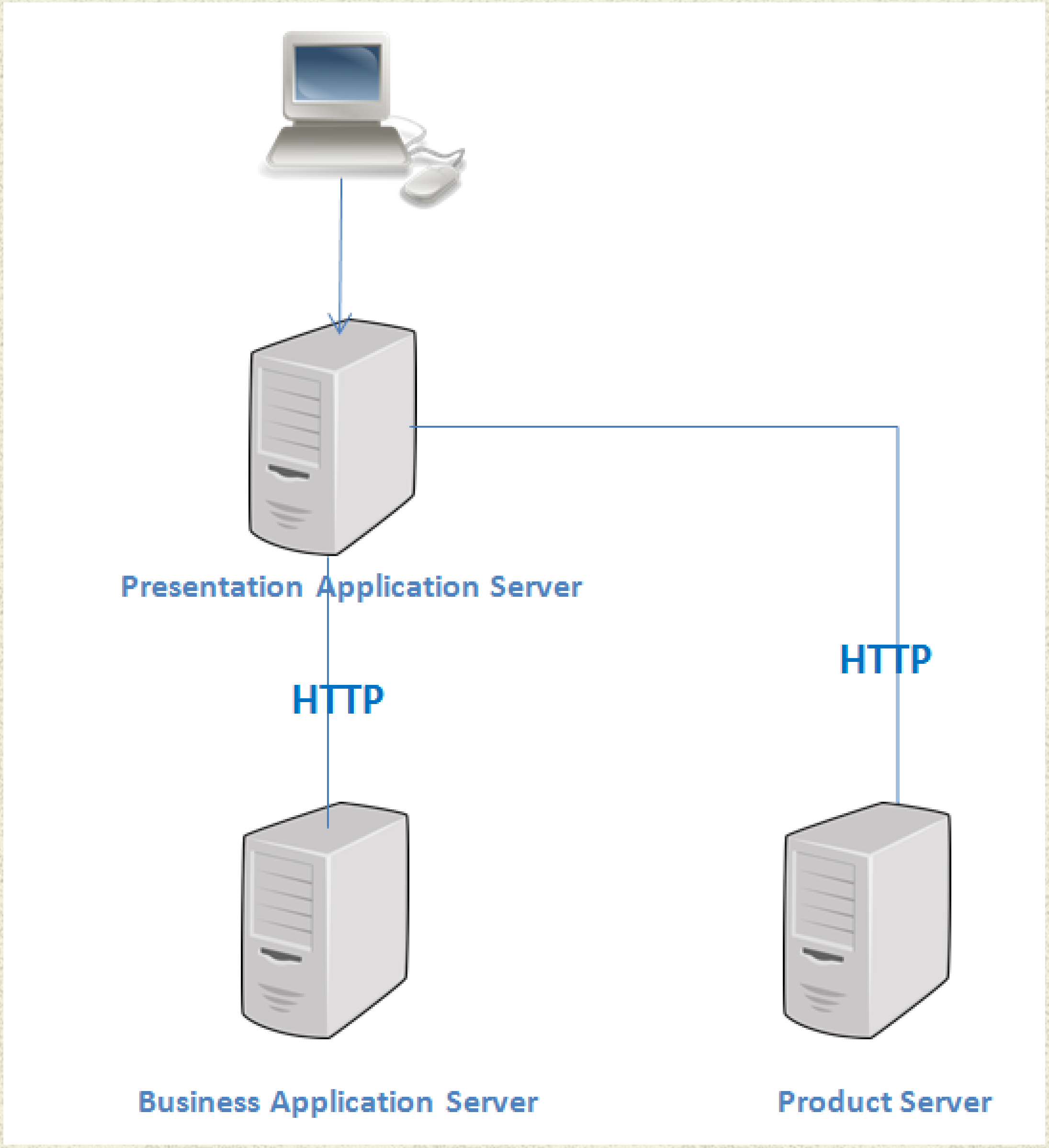
```
ComponentSecurity
  src/main/java
    mum.edu.domain
      Authority.java
      Credentials.java
    mum.edu.repository
      CredentialsDao.java
    mum.edu.service
      CredentialService.java
    mum.edu.service.impl
  src/main/resources
```



```
ComponentMember
  src/main/java
    mum.edu.domain
      Member.java
    mum.edu.repository
      MemberDao.java
    mum.edu.service
      MemberService.java
    mum.edu.service.impl
  src/main/resources
```

See Demo MemberMVC/MemberRest/JerseyRestSecurity

RESTful Product Component



Main Point

- REST fits into a well-designed N-tier application enterprise very easily.
- *Life is well designed and built in layers, accommodating change very easily.*

Oauth2

Industry-standard protocol for authorization.

Stateless - no HTTP session

Provides authentication and authorization as a service.

Allows limited application access to HTTP services [e.g. Google]

Supports Single Sign On [SSO]

Token based

Advantages of Tokens over Http Sessions:

Reduced server load

Streamlined permission management

Support for distributed and cloud-based infrastructure.

RFC 6749

OAuth2 Roles

1. **Resource owner (the User)** – an entity capable of granting access to a protected resource (for example end-user).
2. **Client** – an application making protected resource requests on behalf of the resource owner and with its authorization.
3. **Authorization server** – the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
4. **Resource server (the Rest API server)** – the server hosting the protected resources, capable of accepting/ responding to protected resource requests using access tokens.

Grant Types

Authorization Code:

Used with server-side Applications

Most commonly used because it is optimized for server- side applications

Implicit:

used with Mobile Apps or Web Applications that run on the user's device – SPAs [Angular, React]

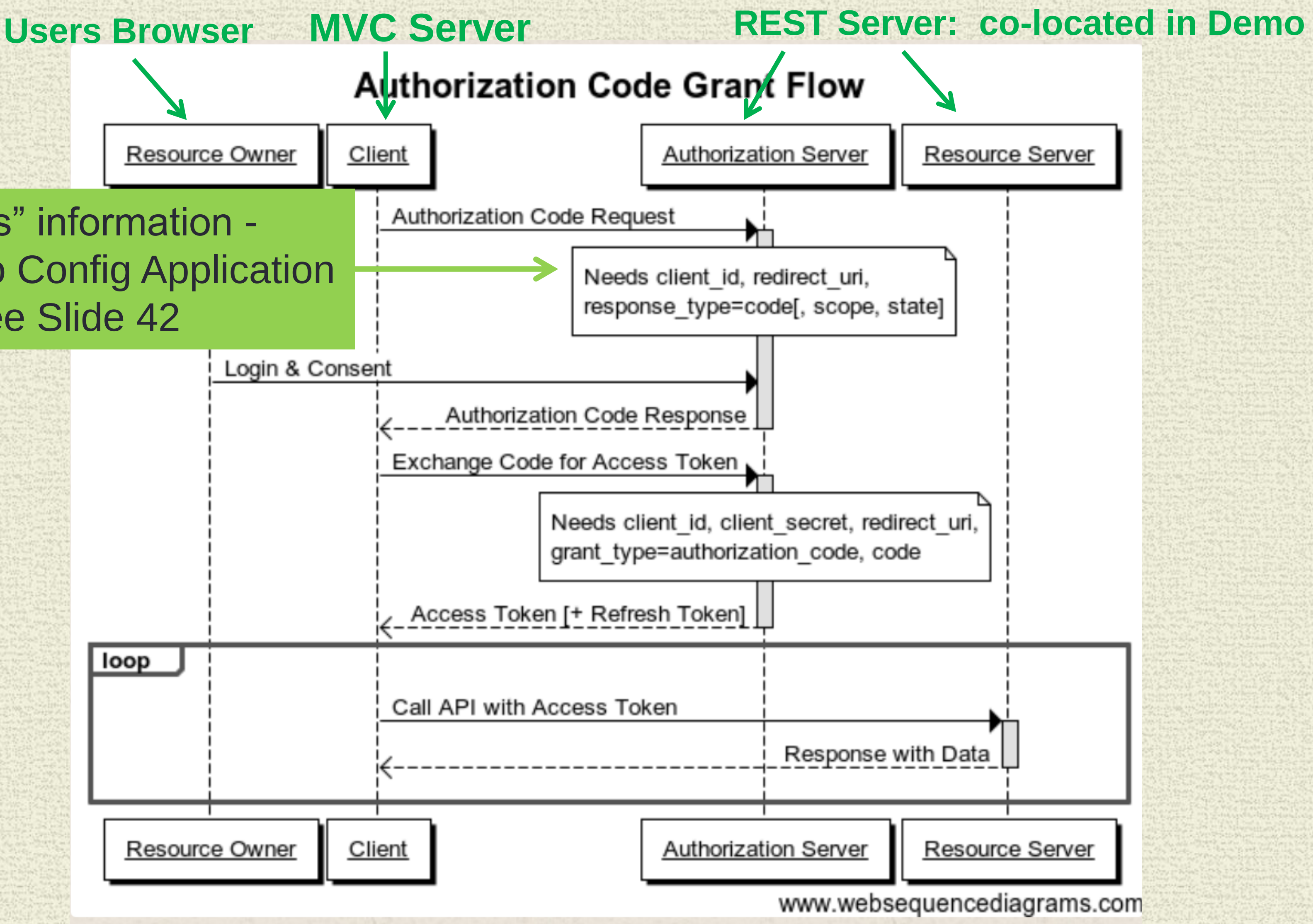
Resource Owner Password Credentials:

Used with Trusted Applications best suited when both the client and the servers are from same company.

Client Credentials:

Used with non interactive application [e.g Batch].Token issued directly to application

Authorization Code

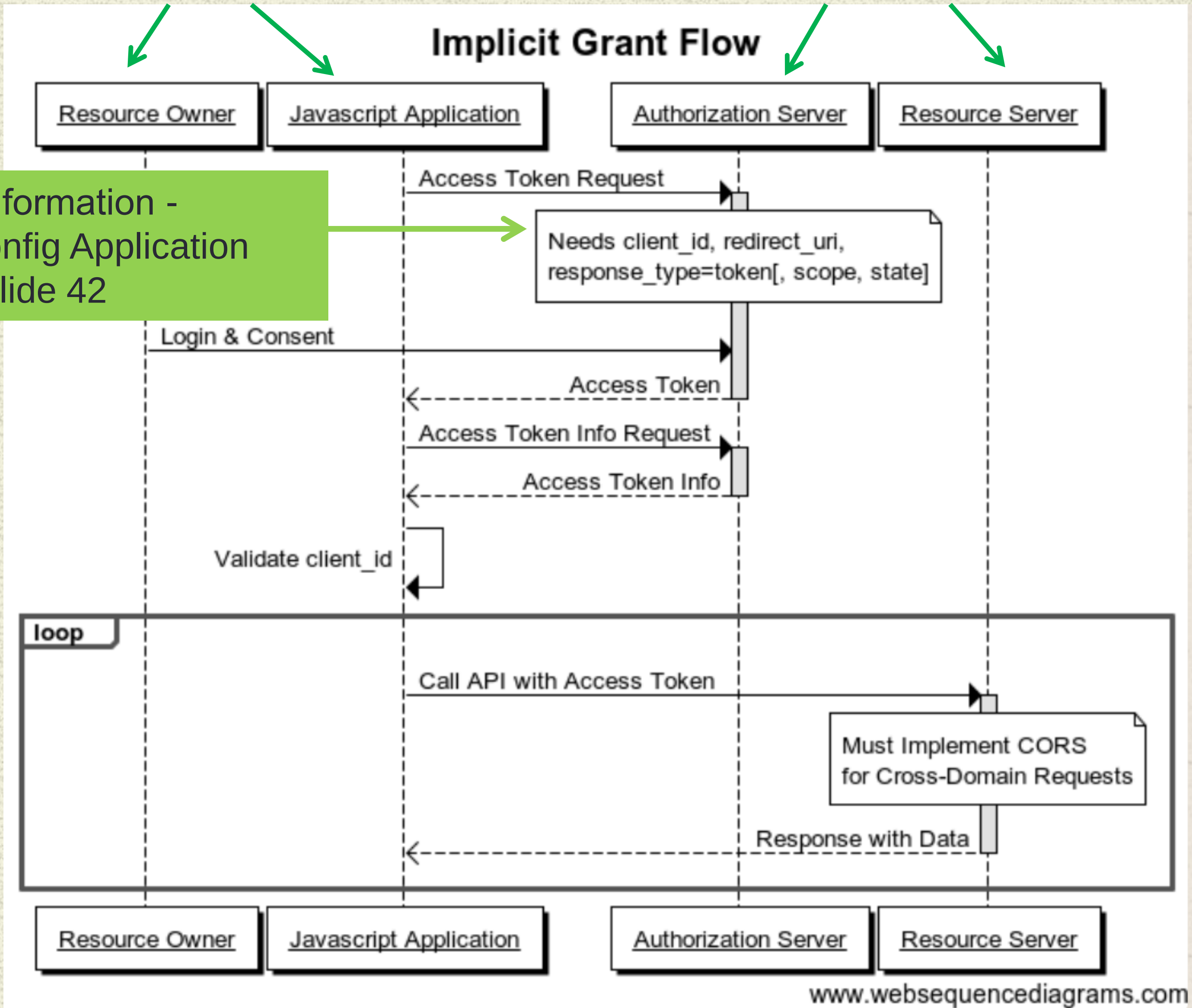

Users Browser
MVC Server
REST Server: co-located in Demo
Authorization Code Grant Flow
Resource Owner
Client
Authorization Server
Resource Server
"properties" information - Needed to Config Application See Slide 42
Authorization Code Request
Needs client_id, redirect_uri, response_type=code[, scope, state]
Login & Consent
Authorization Code Response
Exchange Code for Access Token
Needs client_id, client_secret, redirect_uri, grant_type=authorization_code, code
Access Token [+ Refresh Token]
loop
Call API with Access Token
Response with Data
Resource Owner
Client
Authorization Server
Resource Server
www.websequencediagrams.com

Implicit Grant

Users Browser

REST Server: co-located in Demo

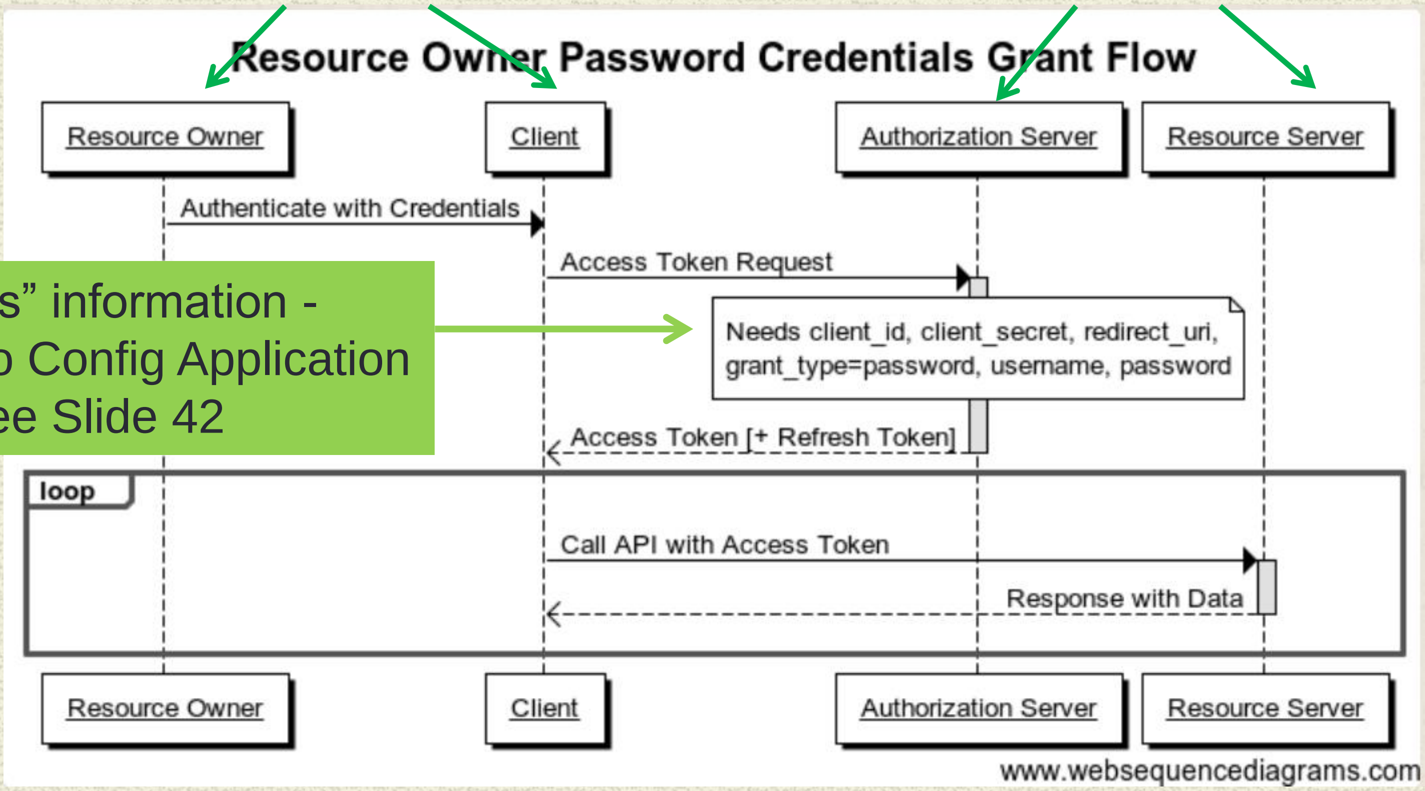
“properties” information - Needed to Config Application
See Slide 42



Resource Owner Password Credentials

Client Server

REST Server


Resource Owner Password Credentials Grant Flow
Resource Owner
Client
Authorization Server
Resource Server
Authenticate with Credentials
Access Token Request
Needs client_id, client_secret, redirect_uri, grant_type=password, username, password
Access Token [+ Refresh Token]
loop
Call API with Access Token
Response with Data
www.websequencediagrams.com

“properties” information - Needed to Config Application See Slide 42

Oauth2 Server Configuration

```
void configure(final ClientDetailsServiceConfigurer clients) throws Exception {
    clients.inMemory()
        .withClient("SampleClientId")
        .secret(passwordEncoder.encode("secret"))
        .authorizedGrantTypes("authorization_code","refresh_token")
        .scopes("user_info")
        .redirectUris("http://localhost:8083/MemberMVCOauthTest/login")
        .accessTokenValiditySeconds(30)
        .and()
            .withClient("MemberMVCOauthPwd")
            .secret(passwordEncoder.encode("FooBar"))
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read")
            .accessTokenValiditySeconds(10)
        .and()
            .withClient("ImplicitClientId")
            .secret(passwordEncoder.encode("FooBar"))
            .authorizedGrantTypes("implicit")
            .redirectUris("http://localhost:8181/MemberOauthImplicit/index.html")
           .scopes("read");
```

See MemberRestOauth Demo

Authorization Code Client W/ SSO

```
@EnableOAuth2Sso
@Configuration
public class UiSecurityConfig extends WebSecurityConfigurerAdapter {
```

```
server:
    port: 8083
    servlet:
      context-path: /MemberMVCOauthTest
security:
  oauth2:
    client:
      clientId: SampleClientId
      clientSecret: secret
      accessTokenUri: http://localhost:8080/MemberRestOauth/oauth/token
      userAuthorizationUri: http://localhost:8080/MemberRestOauth/oauth/authorize
    resource:
      userInfoUri: http://localhost:8080/MemberRestOauth/user
spring:
  autoconfigure:
```

See MemberRestOauthCode Demo

Token-based Resource Server [REST] Calls

```
public HttpHeaders getHttpHeaders() {

Authentication authentication =
             SecurityContextHolder.getContext().getAuthentication();

OAuth2AuthenticationDetails details =
            (OAuth2AuthenticationDetails)  authentication.getDetails();
  String accessToken = details.getTokenValue();
  String authHeader = "Bearer " + accessToken;

  HttpHeaders requestHeaders = new HttpHeaders();
  requestHeaders.set("Authorization", authHeader);
```

Google as SSO with Authorization Code based token

```
server:
    port: 8083
    servlet:
      context-path: /MemberMVCOauthGoogle
security:
  oauth2:
    client:
      clientId: 634044519076...apps.googleusercontent.com
      clientSecret: Cyd4EgRs-H-XoQDK2W5NovYs
      accessTokenUri: https://www.googleapis.com/oauth2/v3/token
      userAuthorizationUri: https://accounts.google.com/o/oauth2/auth
      token-name: oauth_token
clientAuthenticationScheme: form
      scope: profile email
    resource:
      userInfoUri: https://www.googleapis.com/userinfo/v2/me
```

Implicit Client

index.html:

```
<button type="button"  class='btn btn-lg btn-success btn-mini'
         onclick="location.href=
           'http://localhost:8080/MemberRestOauth/oauth/authorize‘
          + '?client_id=ImplicitClientId'
          + '&response_type=token'
          + '&state=S5678' "> Login
  </button>
```

Main.js:

```
 var baseURL = 'http://localhost:8080/MemberRestOauth';

 //This represents the token you got after login
   var authToken = getParams(window.location.href);

function getParams(url) {
   var tokenString = "access_token=";
   var start = url.search(tokenString) + tokenString.length;
   var token = url.substring(start,end);
     return token;
```

Implicit Resource Server Call

```
getMember = function() {
    $.ajax({
        url: baseURL + "/members/1",
        type: "get",
        dataType: "json",
 headers: {
    "Authorization":   ("Bearer " + authToken)
      },
      success: function (response) { }
```

Resource Owner Password Credentials Client

```
server:
    port: 8082
    servlet:
      context-path: /MemberMVCOauthPwd
security:
  oauth2:
    client:
     clientId: MemberMVCOauthPwd
     clientSecret: FooBar
     accessTokenUri: http://localhost:8080/MemberRestOauth/oauth/token
     userAuthorizationUri: http://localhost:8080/MemberRestOauth/oauth/authorize
    resource:
      userInfoUri: http://localhost:8080/MemberRestOauth/user/me
spring:
  autoconfigure:
```

Client-Configure Token & Spring Template

```
protected OAuth2ProtectedResourceDetails resource() {
       ResourceOwnerPasswordResourceDetails resource =
                                          new ResourceOwnerPasswordResourceDetails();
        resource.setAccessTokenUri(accessTokenUri);
       resource.setClientId(clientId);
       resource.setClientSecret(clientSecret);
       resource.setGrantType("password");
       resource.setScope(scopes);

       return resource;
 }

public OAuth2RestTemplate restTemplate() {
   AccessTokenRequest accessTokenRequest = new DefaultAccessTokenRequest();
   // Client parameters[resource()] & DefaultAccessTokenRequest passed to
       template constructor
   return new OAuth2RestTemplate(resource(),
                     new DefaultOAuth2ClientContext(accessTokenRequest));
   }
```

Simulate Credentials & Access Resource Server

```
private void mainInternal() throws IOException {

ResourceOwnerPasswordResourceDetails resource =
        (ResourceOwnerPasswordResourceDetails) oAuth2RestTemplate.getResource();
// "Simulate getting user's credentials
    while(true) {
      String name = in.readLine();
      String password = in.readLine();
      resource.setUsername(name);
      resource.setPassword(password);
```

```
public List<Member> findAll()  throws OAuth2AccessDeniedException {
// Use OAuth2RestTemplate - token setup in config & populated with
                   username/password from console input [see previous slide]
    OAuth2RestTemplate oAuth2RestTemplate = restHelper.getOAuth2RestTemplate();
    ResponseEntity<Member[]> responseEntity =
                     oAuth2RestTemplate.getForEntity(baseUrl, Member[].class);
    List<Member> userList = Arrays.asList(responseEntity.getBody());
    return userList;
```

Access Token Strategy

Managed by Authorization Server

Two types

- **Self-contained**
  - Token has credentials as content
    - e.g. JWT
  - Token is signed
    - Public Key made available to Resource Server
- **Reference**
  - Token is a random “unique” string
  - Key to credentials stored
    - in-memory, cached or datastore

See OAuth2 Guide - Managing Tokens Section

JWT Token

Three Parts

Header JWT Content

Type of token and hashing algorithm

"alg": "HS256",

"typ": "JWT"

Payload

defined by “**registered claims**” **See** Registered JWT Claims

Additional custom claims are allowed **for sub, iss, exp definitions**

"**sub**": “admin",

"scopes": **Custom claim**

“authority": "ROLE_ADMIN“

"**iss**": "http://mum.edu",

"**exp**": 1508625322

Signature

Signature ensures token not tampered with - signed with a private key

HMACSHA256(base64(header) + "." + base64 (payload), **secret**)

Reference Token

Database Example used by Reference Token

Access Token is Key to Database

Access is ***directly*** from Resource Server

Table: **oauth_access_token**

Columns:

token_id VARCHAR(256)**,** **Access Token**

token LONG VARBINARY,

authentication_id VARCHAR(256) PRIMARY KEY,

user_name VARCHAR(256),

client_id VARCHAR(256),

authentication LONG VARBINARY, **Authentication/Authorities Blob**

refresh_token VARCHAR(256)

Spring OAuth2 Schema

Main Point

OAuth2 is a token based protocol for authorization. Tokens provide streamlined permission management and support for distributed and cloud-based infrastructure.

Science of Consciousness: *Cosmic Consciousness is characterized by the right action at the right time for full effectiveness.*