Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

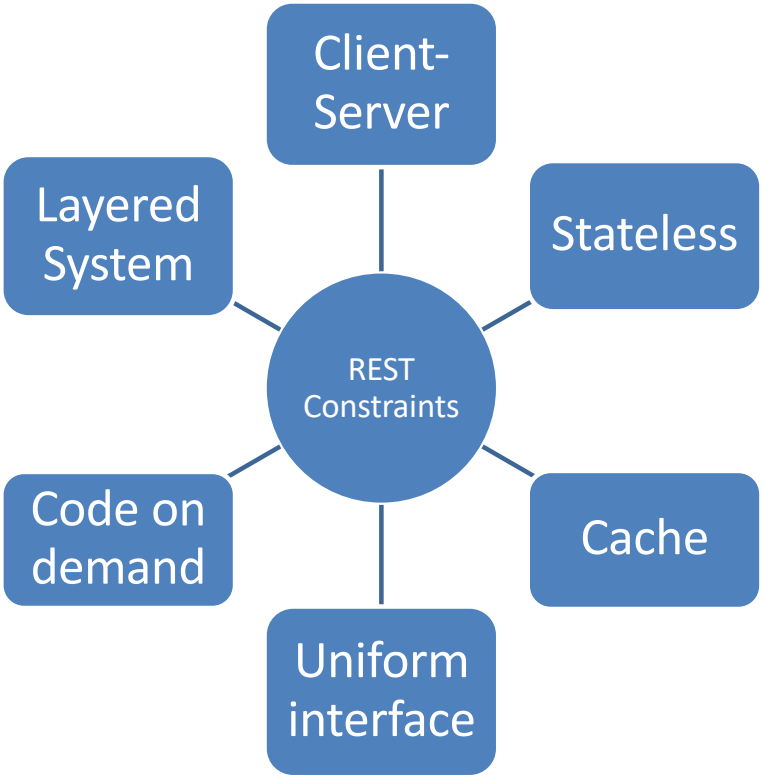
Web Services III

Agenda @Today

1. Richardson Maturity Model
   - HATEOAS Principle
2. Handling resource data
   - JPA possibilities

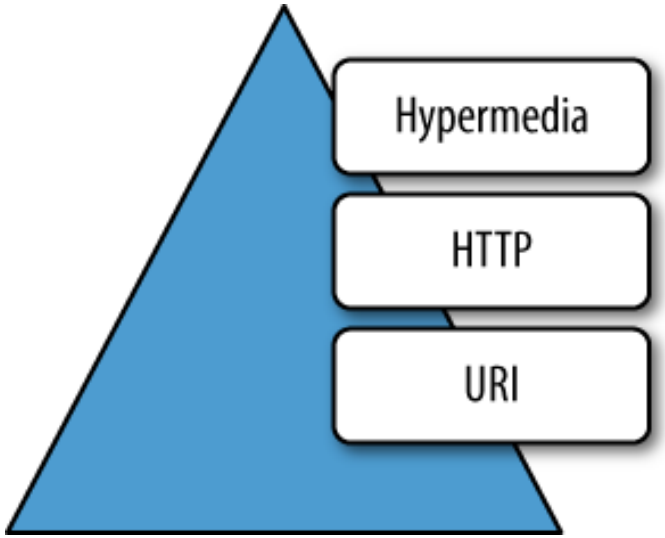
REST Architectural Constraints (Re-visit)

REST defines 6 architectural constraints to make any web service a truly RESTful API

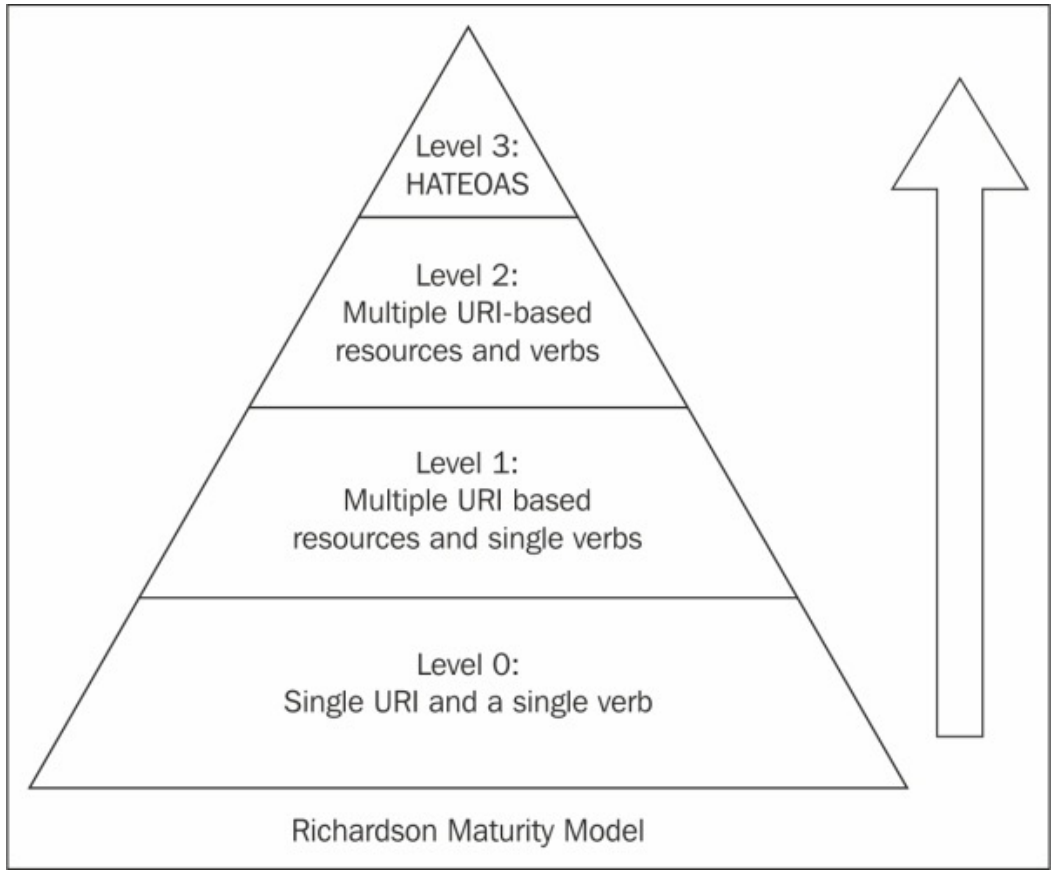


Richardson Maturity Model

- The model defines how much a web service API is REST compliant
- Richardson used three main factors to decide the maturity level
  1. URI
  2. HTTP Methods
  3. HATEOAS (Hypermedia)



Richardson Maturity Model (RMM)



- Level Three
- Level Two
- Level One
- Level Zero

RMM – Level Zero

- At this maturity level, services do not make use of URIs, HTTP Methods, and HATEOAS capabilities
- Uses a single URI, single HTTP method (typically POST)
- Example
  - SOAP Web services use a single URI to identify an endpoint
  - Use HTTP POST to transfer SOAP-based payloads
  - Ignored the rest of the HTTP verbs

POST http://localhost:8090/elibrary/api/v1/users

RMM – Level One

- At this level, a service makes use of URIs, but does not use HTTP Methods, and HATEOAS.
- The services employ many URIs but only use a single HTTP verb
  - HTTP POST
- Each resource has a unique URI
- Thus, it's better than level zero.

POST http://localhost:8090/elibrary/api/v1/users/1

RMM – Level Two

- At this level, services make use of URIs and HTTP Methods, but does not use HATEOAS
- These services support several HTTP verbs on each exposed resource
  - CRUD: Create, Read, Update, and Delete
  - State of resources can be manipulated over the network.
- Maturity level 2 is the most popular use case of the REST principles
  - Leverage HTTP APIs (GET, POST, PUT, DELETE, etc)
  - Resources are identified and mapped to distinct URIs

GET http://localhost:8090/elibrary/api/v1/users
GET http://localhost:8090/elibrary/api/v1/users/1
POST http://localhost:8090/elibrary/api/v1/users
PUT http://localhost:8090/elibrary/api/v1/users/1
DELETE http://localhost:8090/elibrary/api/v1/users/1

RMM – Level Three

- Level three maturity makes use of all three
  - URIs, HTTP Methods, and HATEOAS
- This is the highest maturity for RESTful web services that make them truly RESTful
  - Promotes easy discoverability
  - Responses can be self-descriptive

Truly RESTful API

- Discoverability & Self-descriptive
  - It should be possible to navigate the entire set of resources without prior knowledge of the URI scheme
- HTTP GET request should
  - return information necessary to find the resources related to the requested object through hyperlinks
  - provide operations possible on each of these resources
- This principle is known as HATEOAS
  - Hypertext(media) as the Engine of Application State

HATEOAS principle

- In many applications, the allowed actions on a resource depend on the **state** of that resource
- HATEOAS provides a means for the server to say what is allowed on a resource
  - Instead to require the consumer to understand and code for the allowed state

HATEOAS

- Consider a bank account (12345) with NOK100 balance
- A REST query on the resource might return a response showing that deposit, withdrawal, and transfer are the allowed actions

```
{
  "account_number”:”12345”’
  “balance”: 100.00,
  “links”:[
      {“rel”: "deposit",  “href”:"/account/12345/deposit"},
      {“rel”: "withdraw", “href":"/account/12345/withdraw"},
      {“rel”: "transfer", “href”:"/account/12345/transfer"}
  ]
}
```

Example from: https://apiguide.readthedocs.io/en/latest/build_and_publish/hateos.html

HATEOAS

- However, if the same account is overdrawn -25NOK balance.
- Then, the only possible action is deposit

```
{
  "account_number”:”12345”’
  “balance”: -25.00,
  “links”:[
     {“rel”: "deposit",  “href”:"/account/12345/deposit"}
     ]
}
```

Brainstorming – eLibrary REST APIs

- Applying HATEOAS principle to our APIs
- Looking at the next slide, what possible actions should we link to based on the state of the resource?

eLibrary REST API Endpoints

Resource	API Method	Endpoint (URI Path)	HTTP Method	HATEOAS LinksTo
Create/Register a user	createUser	/users	POST	
Create a borrow book order for a user	borrowBook	/users/{id}/orders	POST	
Delete a user	deleteUser	/users/{id}	DELETE	
Delete (Return/cancel) a book order	returnBook	/orders/{id}	DELETE	
List all borrow orders	getBorrowOrders	/orders	GET	
Details of a borrow order	getBorrowOrder	/orders/{id}	GET	

Something more!
If a book expires in a few days, it can be intuitive to provide links to return or renew book if possible
These links can be processed automatically by the backend reminder system

HATEOAS in Spring Framework

Spring Framework provides support for HATEOAS

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

```
public class Order extends RepresentationModel<Order>
```

```
private void addLinks(Set<Order> orders) throws OrderNotFoundException {
    for(Order order : orders) {
        Link link = linkTo(methodOn(OrderController.class)
                    .returnBookOrder(order.getId()))
                    .withRel("Update_Return_or_Cancel");
        order.add(link);
    }
}
```

HATEOAS in Spring Framework

POST /elibrary/api/v1/users/1/orders


```
{
    "id": 1,
    "isbn": "ghijk1234",
    "expiry": "2023-11-12",
    "_links": {
        "Update_Return_or_Cancel": {
            "href": "http://localhost:8090/elibrary/api/v1/orders/1"
        }
    }
}
```

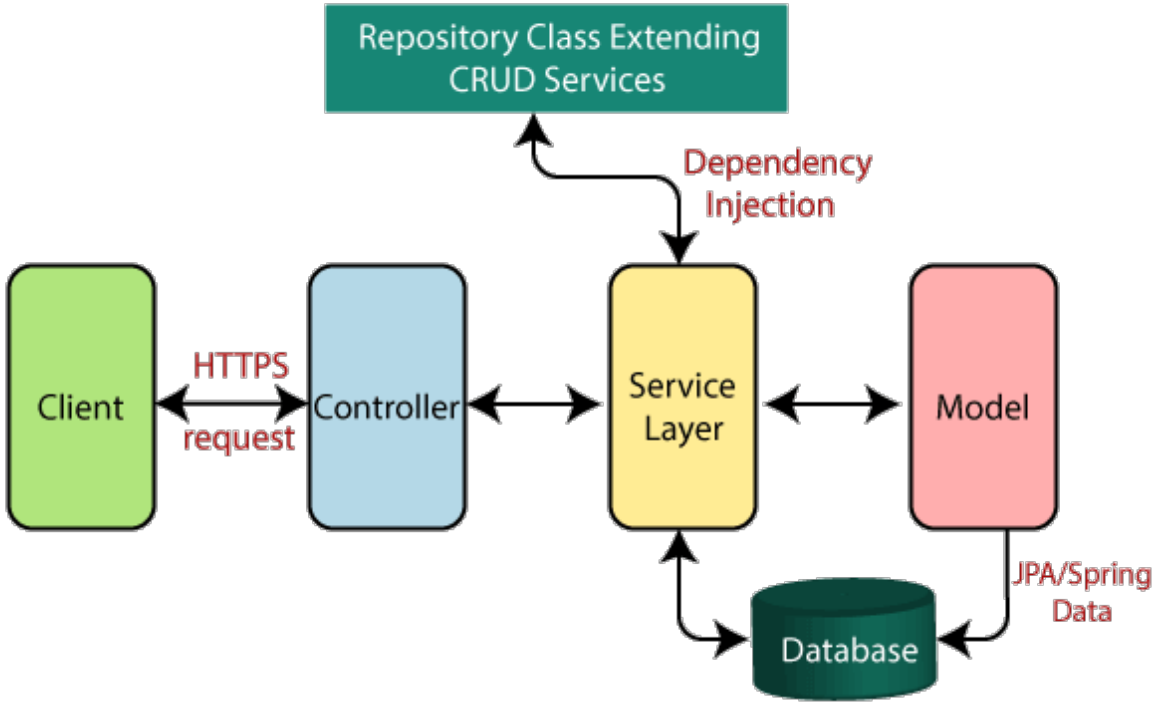
Part 2: Handling Resource Data

Handling Resource Data

- RESTful APIs (e.g. GET /orders) can pull large data from the database
- In addition, it should be possible to present data based on different business conditions

Handling Resource Data

- The Spring Data/JPA provides ‘Query API’ support to fetch data based on different conditions



https://www.javatpoint.com/spring-boot-architecture

Spring Data - CRUD JPA

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  <S extends T> S save(S entity);
  Optional<T> findById(ID primaryKey);
  Iterable<T> findAll();
  long count();
  void delete(T entity);
  boolean existsById(ID primaryKey);

  // ... more functionality omitted.
}
```

```
public interface BookRepository extends CrudRepository<Book, Long> {}
```

By extending the CrudRepository class, our object can then have access to the CRUD operations.

Custom JPA Query Methods

- Sometimes, we need to add extra functionalities than what the default JPA methods provide.
- JPA supports these using 2 approaches
  - Derived Query Creation from method names.
  - Using SQL query mapped to the method with **@Query** annotation
- Typical cases are filtering, paging, searching, and sorting supports

GET localhost:8090/elibrary/api/v1/orders?expiry=2023-11-30&page=0&size=5

GET localhost:8090/elibrary/api/v1/orders?expiry=2023-11-30&limit=5&offset=0

Derived JPA Query Methods

- Derived Query method has 2 elements
  - Subject (the action)
  - Predicate (the conditions)
- Subject: introduces clause
  - *find…By*, *exists…By*, *count…By*
  - May contain further expressions (between *find/exists/count* and *By)* for result-limiting keywords such as Distinct or Top/First
- Predicate: this is placed after the **subject**.
  - It can be entity properties (concatenated with And/Or) followed by one or more keywords (*StartingWith*, *EndingWith*, *Containing*, *IgnoreCase*, …)

Full list here -> https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject

Derived JPA Query Methods

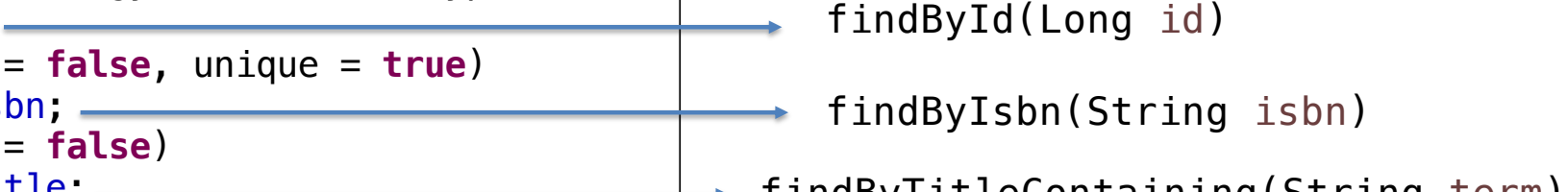
Predicate	**Subject**	**Example**	**JPQL snippet**
Lastname…Firstname	findDistinctBy…And…	**findDistinctBy**Lastname**And**Firstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
Lastname…Firstname	findBy…And…	**findBy**Lastname**And**Firstname	… where x.lastname = ?1 and x.firstname = ?2
StartDate	findBy … Between	**findBy**StartDate**Between**	… where x.startDate between ?1 and ?2
Lastname…Firstname	findBy … Or	**findBy**Lastname**Or**Firstname	… where x.lastname = ?1 or x.firstname = ?2
Age	findBy … LessThan	findByAge**LessThan**	… where x.age < ?1
Firstname	findBy … Like	findByFirstnameLike	… where x.firstname like ?1

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

Examples

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    @Column(nullable = false, unique = true)
    private String isbn;
    @Column(nullable = false)
    private String title;
    ...
}
```




```
public interface BookRepository extends JpaRepository<Book, Long> {
    Book findByIsbn(String isbn);
    List<Book> findByIsbn(String isbn, Pageable pageable);
    ...
}
```

Custom JPA Query Methods

- JPA also provides 2 additional possibilities (using @Query annotation)
  1. Using JPQueryLanguage (JPQL) to construct custom method bound to repository methods
  2. Using native queries bound to repository methods

```
@Query("SELECT b FROM Book b WHERE b.isbn = :isbn")
Book findBookByISBN(@Param("isbn") String isbn);
```

```
@Query(value = "SELECT userid FROM orders WHERE id = id", nativeQuery=true)
Long findUserID(@Param("id") Long id);
```

Example

```
@Query("SELECT b FROM Book b WHERE b.isbn = :isbn")
Book findBookByISBN(@Param("isbn") String isbn);
```

Using JPQL

```
Book findByISBN(String isbn);
```

Using Derive Query Method

Therefore, take a look at the derived JPA query before you write your own custom query.

Filtering, Paging and Sorting

- JPA provides us with some possibilities
  - Using the PagingAndSortingRepository abstraction with sort and pageable methods
  - Using native queries bound to repository methods
  - Using Derived Query Methods (Where possible)

```
public interface PagingAndSortingRepository<T, ID> {
    Iterable<T> findAll(Sort sort); Page<T>
    findAll(Pageable pageable);
}
```

```
List<Book> findByIsbn(String isbn, Pageable pageable);
```

```
List<Tutorial> findByTitleAndSort(String title, Sort sort);
```

Examples

- Filtering
  - Filter by isbn
    - e.g. Book findByIsbn(String isbn);
  - Filter by expiry date
    - e.g. List<Order> findByExpiryBefore(LocalDate expiry);
- Pagination
  - Present 20 records per page
  - e.g. Pageable paging = PageRequest.of(0, 20); page=0, size=20
- Sorting
  - Order by title ‘desc or asc’
  - e.g. List<Tutorial> findByTitleOrderByLevelAsc(String title);
- Search
  - findByTitleContaining(“error”)

Oblig#2 – Next Week

- Task #1: RESTful API Services
- Task #2: HATEOAS, Pagination and Filtering
- Task #3: Securing resource endpoints with JWT Access tokens
- Task #4: Securing resource endpoints with JWT Access tokens from OAuth2 Server