

# AMT PROJECT 2

---

Tips & hints

**AMT 2019**

**Olivier Liechti**

- In the **first project**, we have seen that pagination is important
  - Between the client and the presentation tier
  - Between the business and resource tier
- What does it mean for the **second project**, where
  - The presentation tier exposes a **RESTful API**
  - The DAOs are implemented with **Spring Data JPA**

# Pagination in the REST API

---

- How does the **client** request a page?
  - The most common approach is to use the **query string**
- How does the **server** provide paging information?
  - Custom header with count information (number of items, of pages)
  - Standardized header (Links), like in GitHub API v3
  - In the payload, as an envelope

# Pagination in REST: custom headers

```
GET /students?page=4&pageSize=10 HTTP/1.1  
Host: api.gaps.ch
```

```
HTTP/1.1 200 OK  
Content-type: application/json  
Pagination-NumberOfItems: 1053  
Pagination-Next: /students/page=5&pageSize=10
```

```
[  
  {  
    "firstName": "Olivier" ...  
  } ...  
]
```

# Pagination: standard link header

```
GET /students?page=4&pageSize=10 HTTP/1.1  
Host: api.gaps.ch
```

```
HTTP/1.1 200 OK  
Content-type: application/json  
Link: </students?page=5&pageSize=10>; rel="next", </students?  
page=106&pageSize=10>; rel="last"
```

```
[  
  {  
    "firstName": "Olivier" ...  
  } ...  
]
```

<https://tools.ietf.org/html/rfc5988>

# Pagination in the REST API: enveloppe



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

```
GET /students?page=4&pageSize=10 HTTP/1.1  
Host: api.gaps.ch
```

```
HTTP/1.1 200 OK  
Content-type: application/json
```

```
{  
  "pagination": {  
    "numberOfItems": 1053,  
    "page": 4,  
    "next": "/students/page=5&pageSize=10"  
  },  
  "data": [  
    {  
      "firstName": "Olivier" ...  
    } ...  
  ]  
}
```

# Pagination in Spring Data JPA (1)

## Example 3. CrudRepository interface

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S entity);           1  
    Optional<T> findById(ID primaryKey);      2  
    Iterable<T> findAll();                     3  
    long count();                              4  
    void delete(T entity);                     5  
    boolean existsById(ID primaryKey);         6  
    // ... more functionality omitted.  
}
```

JAVA

- 1 Saves the given entity.
- 2 Returns the entity identified by the given ID.
- 3 Returns all entities.
- 4 Returns the number of entities.
- 5 Deletes the given entity.
- 6 Indicates whether an entity with the given ID exists.

# Pagination in Spring Data JPA (2)

## Example 4. PagingAndSortingRepository interface

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

JAVA

To access the second page of `User` by a page size of 20, you could do something like the following:

```
PagingAndSortingRepository<User, Long> repository = // ... get access to a bean  
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

JAVA



# Analyzing JPA behind the scenes

---

- Using an **ORM** to interact with the database
  - ✓ Reduces the time it takes to develop an application, because there is much less “boilerplate code” to write.
  - ➡ Can lead to poor performance, if the developer does not understand what happens behind the scenes (table structure, number of queries, etc.)
- How can I find out what happens at the SQL level?

# Analyzing JPA behind the scenes

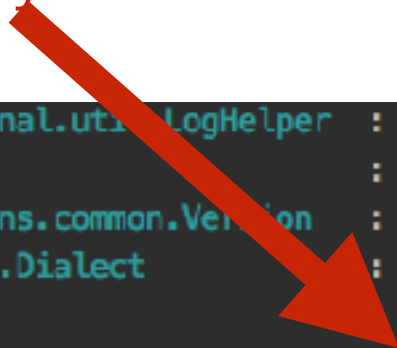
The screenshot shows an IDE with a project structure on the left and a configuration file on the right. The project structure is for a Spring Boot application named 'spring-server [swagger-spring]'. The source code is located in 'src/main/java/io/avalia/fruits/api/endpoints', where 'FruitsApiController' is the main class. The 'resources' directory contains 'api-spec.yaml' and 'application.properties'. The 'target' directory is also visible. The configuration file 'application.properties' is open in the editor, showing the following properties:

```
1 springfox.documentation.swagger.v2.path=/api-docs
2 server.servlet.context-path=/api
3 server.port=8080
4 spring.jackson.date-format=io.avalia.fruits.RFC3339DateFormat
5 spring.jackson.serialization.WRITE_DATES_AS_TIMESTAMPS=false
6
7 logging.level.org.hibernate.SQL=DEBUG
8 logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Two red arrows are present: one points from the 'application.properties' file in the project structure to the editor, and the other points from the 'FruitsApiController' class in the source code to the editor.


# Analyzing JPA behind the scenes

Schema is generated by JPA at boot time



```
7720 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
7720 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate Core {5.4.8.Final}
7720 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
7720 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
7720 --- [ restartedMain] org.hibernate.SQL : drop table fruit_entity if exists
7720 --- [ restartedMain] org.hibernate.SQL : create table fruit_entity (id bigint generated by default as identity, colour
7720 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction
7720 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
7720 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
7720 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
7720 --- [ restartedMain] uration$JodaDateTimeJacksonConfiguration : Auto-configuration of Jackson's Joda-Time integration is deprecated in favor o
7720 --- [ restartedMain] uration$JodaDateTimeJacksonConfiguration : spring.jackson.date-format could not be used to configure formatting of Joda's
7720 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
7720 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path '/api'
7720 --- [ restartedMain] io.avalia.fruits.Swagger2SpringBoot : Started Swagger2SpringBoot in 4.355 seconds (JVM running for 5.052)
7720 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/api] : Initializing Spring DispatcherServlet 'dispatcherServlet'
7720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
7720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
7720 --- [nio-8080-exec-4] org.hibernate.SQL : insert into fruit_entity (id, colour, kind, size) values (null, ?, ?, ?)
7720 --- [nio-8080-exec-4] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [null]
7720 --- [nio-8080-exec-4] o.h.type.descriptor.sql.BasicBinder : binding parameter [2] as [VARCHAR] - [null]
7720 --- [nio-8080-exec-4] o.h.type.descriptor.sql.BasicBinder : binding parameter [3] as [VARCHAR] - [null]
```

a POST request has been received - a record is inserted  
in the database



# Swagger 2.0 vs OpenAPI 3.0

---

- When you write to API specification, be aware that there are different versions of the language.
- The 2 reference projects (Fruits, OpenAffect) use the Swagger 2.0 syntax. Be aware that all the tooling (maven plugins and dependencies) vary between the two formats.
- I have migrated to OpenAPI - it is possible. However, I did not have the time to do a lot of tests, so I recommend you to stick with Swagger 2.0 format for the project.
- Syntax and examples:
  - OpenAPI 3: <https://swagger.io/docs/specification/basic-structure/>
  - Swagger 2: <https://swagger.io/docs/specification/2-0/basic-structure/>

# Swagger: Top-Down vs Bottom-Up

---



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

- **Process**

- **Top-Down** = read the API specification and generate source code
- **Bottom-Up** = read source code with annotations and generate API spec

- **What about the AMT project?**

- We combine both approaches!
- At the very beginning, I used the Top-Down process a first time to generate the initial skeleton. I was not completely happy (e.g. Spring Boot version), so I changed it.
- In the pom.xml, I have added the codegen plugin, so that a Top-Down process is executed every time we do a mvn clean install. However, we do not re-generate everything. We generate the DTOs (types), the controller interfaces. We leave your controller implementations alone.
- The code that is generated top-down contains swagger annotations. When the Spring Boot application starts, the springfox library scans these annotations and re-generates the API spec. What you see in the interactive documentation is the result of this bottom-up process.

# CRUD vs Commands

- We are all familiar with **CRUD** operations and how they map with HTTP methods (POST, GET, PUT/PATCH, DELETE).
- There is an other way to **model** operations, with “commands”

```
PATCH /students/837 HTTP/1.1
Host: api.gaps.ch
Content-type: application/json
```

```
{
  "phone": "079 111 22 33"
}
```

```
POST /commands HTTP/1.1
Host: api.gaps.ch
Content-type: application/json
```

```
{
  "operation": "changePhone",
  "student": "/students/837",
  "phone": "079 111 22 33"
}
```

- What's the best?



# CRUD vs Commands

- The first approach is very common and works well for simple domain models and applications.
- The second one encourages to reason about business processes, asynchronous operations, and immutable event streams (similar to what front-end frameworks like React and Vue do with event stores)
- To learn more about this design approach, look for information on the CQRS and Event Sourcing design patterns (which become increasingly popular)

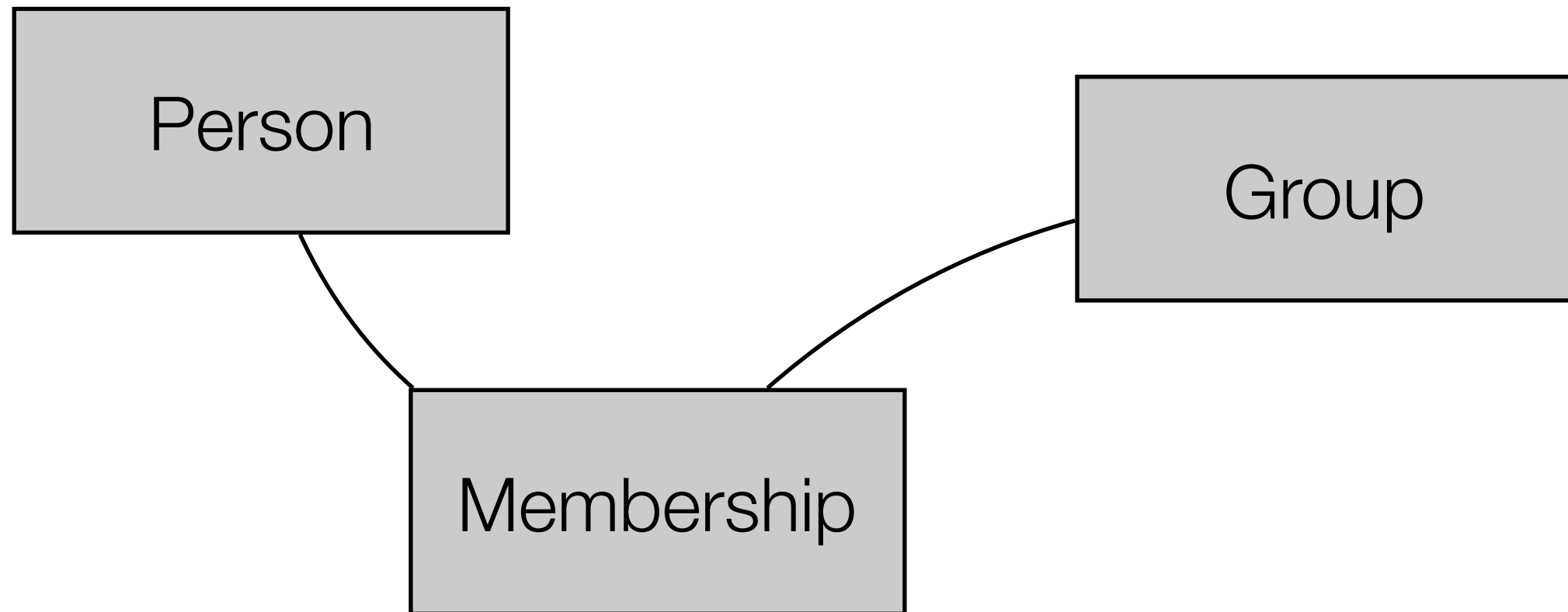
```
PATCH /students/837 HTTP/1.1
Host: api.gaps.ch
Content-type: application/json
```

```
{
  "phone": "079 111 22 33"
}
```

```
POST /commands HTTP/1.1
Host: api.gaps.ch
Content-type: application/json
```

```
{
  "operation": "changePhone",
  "student": "/students/837",
  "phone": "079 111 22 33"
}
```

# CRUD vs Commands



```
DELETE /memberships/34 HTTP/1.1  
Host: api.gaps.ch
```

```
POST /resignations/ HTTP/1.1  
Host: api.gaps.ch
```

```
{  
  "person": "/people/23",  
  "group": "/groups/2"  
}
```

Which one better captures the  
business workflow?



# Couple of Spring Boot features (1)

- **Auto-configuration:**

## 4. Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, if `HSQLDB` is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

In the Fruits project, you can see that I have a dependency on the H2 (in-memory database). Replace that with a dependency on the mysql driver and Spring Boot will automatically adapt the JPA behavior.

# Couple of Spring Boot features (1)

- **Externalized configuration:**

## 2. Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's `Environment` abstraction, or be [bound to structured objects](#) through `@ConfigurationProperties`.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order:

In the Fruits project, I use the `application.properties` file to define application variables. Checkout the documentation to see the list of configuration sources and the order in which they have priority.

Something that is EXTREMELY useful is to override what is defined in `application.properties` with an Environment Variable. Especially if you do that in a `docker-compose.yml` (think JDBC URL!!!)