D | Guides made for the Wiki–page

These guides are written for the Wiki–page for the next years student working on the GIRAF project.

D.1 Setting up the Build Environment

This is a guide in the basic steps for setting up a developer environment for the GIRAF REST API. This takes between 5–15 minutes.

Setting up Intellij

- First download and install InteliJ IDEA.
- Clone the Git repository for the REST API.
- Then start Intellij, and go though the initial start wizzard selecting your preferences.
- Then from the main start menu of Intellij select: Import Project.
- Select the folder which you've cloned the git repository to.
- Select "Import project from external model".
- Select Gradle.
- Press next.
- Select default gradle wrapper
- Press Finish.
- Press ok to import all the gradle project data.
- Intellij should now start to its main interface.
- Wait for it to stop processing (might take a while the first time on slow computers).
- Open the "Event log" in the lower right corner, it is the yellow speech bubb le with en exclamation mark.
- In it, it will show: "Frameworks detected: Spring framework is detected in the project Configure".
- Press the "Configure"–link.
- You should now see the "Setup Frameworks", it should show Spring and two sub-items.
- Press OK, with them selected.
- Now goto "File" -> "Project Structure" -> "Project".
- Under Project SDK select the newest JDK (at least 1.8).
- If no SDK are to be found, then press "New" then "JDK" then navigate to your JDK folder (C:\Program Files\Java\jdk*** on Windows).
- Under "Project language level" select "8 - Lambdas, type annotations etc.".
- Press OK to close "Project Structure".
- Intellij will now Index your files again, this might take a while.
- If everything was done sucessfully, then you should be able to Build the REST–API by pressing "Build" -> "Make Project"

Setting up Spring

- Now we will configure spring to inject the proper settings.
- Navigate to the “persistence-config.xml” file in “persistence/src/main/resources/”.
- In the top press “Change Profiles” -> then select “Project” and “Default”.
- This will configure Spring to use the local in-memory database.
- Then you should create a file called “files.properties” in “persistence/src/main/resources/”.
- This file should be the same as “files-prod.properties” but have paths which fits your machine. (On windows a path such as “C:/giraf/” is fine).
- This path is used to store the pictogramimages and usericons, which are uploaded though the API.

Setting up WildFly

- In order to test the REST–API locally, we will now setup WildFly to run locally.
- First download WildFly from `http://wildfly.org/downloads/`.
- This is made with the “Java EE7 Full & Web Distribution” in mind.gst
- Unzip it.
- Open the bin folder.
- Use the “add-user” script appropriate for your platform (.bat for windows, .sh for nix*)
- select a) Management User
- Enter a username.
- Enter a password.
- When prompted if you want to add it to any groups, just press enter.
- Type “yes” to add the user to the realm “ManagementRealm”, and again for the AS process prompt.
- Press enter to close the prompt.
- Open Intellij again if you closed it.
- Open the build configuration menu by pressing the dropdown in the upper right corner and pressing “Edit Configuration”.
- Press the plus sign to add a new configuration.
- Select “JBoss” -> “Local” (Might have to press show 33 more item in order for it to show, look at the bottom of the dropdown).
- First press “Configure”
- Then navigate to the install location of your WildFly folder.
- Once it is located press “Register schemas...”, then press OK, and OK again.
- Uncheck “After launch” (You should use a REST–client here, try out POSTMAN for Google Chrome)#SELLOUT
- Enter your username and password you set before under “JBoss Server Settings”
- If another service is using port 8080 then enter a Port offset i.e. 1. (NVIDIA Network Services does on windows).
- It should give you a warning that “No artifacts marked for deployment”, press the “Fix” next to it, select “dk.aau.giraf.rest : giraf-rest-1.0.0.ear (exploded)”.
- Press OK to exit the configuration menu.
- Try to run the WildFly server by pressing run in the upper right corner.
- This should build and deploy the WildFly server.
- After a while the Output window should show: “Artifact Gradle : dk.aau.giraf.rest : giraf-rest-1.0.0.ear (exploded): Artifact is deployed successfully”.

- And above it “Registered web context: /services-1.0.0”.
- This means that on the url `http://localhost:8080/services-1.0.0` should be the base of the REST–API.
- However nothing should be running on the root, try accessing `http://localhost:8080/services-1.0.0/pictogram/`.
- This should show some JSON for the public pictograms made in the localdata.
- Otherwise there should be an error in the Output view for you to debug.

Checkstyle

Optionally you can use the CheckStyle-IDEA plugin to check linting in real-time.

- Go to “File -> settings”, then “Plugins” then “Browse repositories”
- Search for “CheckStyle-IDEA” and press install.
- Press close, then ok.
- Restart Intellij
- Go to “File -> settings”
- Then “Other Settings -> Checkstyle”
- Under “Configuration File” press the plus icon.
- Find the “checkstyle.xml” file in the root of the Git repository.
- Be sure to mark it as “Active”
- Press Ok

Enjoy.

D.2 Hibernate Annotations

Hibernate is a tool we use for Object Relational Mapping, which e.g. makes it easier to create the relations between classes, and especially when retrieving the objects of the database.

This guide will show how we created some relations in the REST API using Hibernate, i.e. how to do a many-to-many, one-to-one, many-to-one. In a code block `[...]` means that the other code is omitted from the example.

First up however, any class which you want modelled in the database, needs to have the following annotations above the class definition: `@Entity` and `@Table(name = <nameoftable>)`

A One-to-One relationship

We have a One-to-One in the class: Pictogram, which has a single PictogramImage. You simply write: `@OneToOne(<fillWithFields>)`. In Pictogram it looks like this:

```
@OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, optional =
    true)
private PictogramImage pictogramImage;
```

Listing D.1: A One-to-One relationship using `PictogramImage`, on `Pictogram`.

- Cascade means that if a Pictogram is deleted or created, the action will cascade so PictogramImage will know of e.g. the delete, and will also be deleted. Please see the Hibernate Documentation for more information on the CascadeTypes.
- FetchType.LAZY means that the object will not be loaded in the same transaction as the pictogram is loaded, the opposite is FetchType.EAGER and this will load it in the same transaction.
- optional means that the field is not required.

In the SQL a simple foreign key constraint is set on the table. The table Pictogram contains the property as long with its constraint:

```
(
  [...]
    pictogramimage_id BIGINT,
    CONSTRAINT Pictogram__id_PictogramImage FOREIGN KEY
    (pictogramimage_id) REFERENCES PictogramImage (id),
  [...]
}
```

Listing D.2: The SQL to create the required

A many-to-one relationship

A many-to-one, is used in PictoFrame to set Department. The new thing here is that a `JoinColumn` has to be set which specifies which column will contain the foreign key for the table.

```
@ManyToOne(fetch = FetchType.EAGER, optional = true)
@JoinColumn(name = "department_id", nullable = true)
protected Department department;
```

Listing D.3: The Annotations required for a `Many-to-One`

The SQL needs to be created as follows to create this:

```
 CREATE TABLE PictoFrame
(
    [...]
    department_id BIGINT,
```

```
    [...]
);
ALTER TABLE PictoFrame
    ADD CONSTRAINT PictoFrame__department_id
        FOREIGN KEY (department_id) REFERENCES Department (id);
```

Listing D.4: The SQL needed for a `Many-to-One`

Simply adding the department_id as specified in the annotation to the SQL, and adding the constraint, which means that the difference of create a `One-to-One` and a `Many-to-One`, is simply adding the `@JoinColumn` annotation to the field.

A Many-to-Many relationship

A many-to-many is created in two different ways in the REST API. With an extra class to model more data which is not just an index, and without an extra class.

The first will be without the extra class which happens in the class Sequence.

```
    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(name = "Sequence__Frame",
            joinColumns = {
                @JoinColumn(name = "sequence_id")},
            inverseJoinColumns = {
                @JoinColumn(name = "frame_id")})
    @OrderColumn(name = "index")
    private List<Frame> elements;
```

Listing D.5: The Annotations required for a `Many-to-Many` relationship without creating an extra class to model it.

Instead of creating a `JoinColumn` we know create a `JoinTable`, the tables data is made using the annotations, it is given a name and two `JoinColumns` are created. A `JoinColumn` which is the id of the class this is being modelled in, and an `inverseJoinColumns` which is the opposite direction, which is then the id of the other side, in this case `Frame`.

`@OrderColumn` is created here, and tell hibernate that the List should be ordered according to the property Index, this is how the ordering of the sequence elements is created, to make sure they always come in the same order.

Now the opposite side also needs to be modelled for a `Many-to-Many` in Hibernate so it contains the following as well:

```
 @ManyToMany(fetch = FetchType.LAZY, mappedBy = "elements", cascade =
    CascadeType.ALL)
    private Collection<Sequence> partOfSequences;
```

Listing D.6: The opposite side of the relations needs to say it is mappedBy the other end.

Here the annotation `mappedBy` is needed to specify which field on the opposite side(Sequence) models it in the database.

For the SQL we need to add the following:

```
CREATE TABLE Sequence__Frame
(
    id BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    index INT NOT NULL,
    sequence_id BIGINT NOT NULL,
    frame_id BIGINT NOT NULL
);
ALTER TABLE Sequence__Frame
    ADD CONSTRAINT Sequence__Frame__choice_id
        FOREIGN KEY (sequence_id) REFERENCES Sequence (frame_id);

```

```
ALTER TABLE Sequence__Frame
    ADD CONSTRAINT Sequence__Frame__pictoFrame_id
        FOREIGN KEY (frame_id) REFERENCES Frame (id);
```

Listing D.7: The SQL for the above mentioned `Many-to-Many` relationship.

The second with extra classes is created like this:
You need three classes, in the REST API example we have a Weekday, a Frame, and a Weekday-Frame. You may create a `@one-to-many` on the two classes, e.g. Frame, and Weekday:

```
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "weekday_id", nullable = false)
    @OrderColumn(name = "weekdayframe_index", updatable = false,
    insertable = false)
    private List<WeekdayFrame> frames;
```

Listing D.8: One of the relations on the opposite side to create the `One-to-Many` to the relationship class.

But you only need to create the **@many-to-one** for both classes on their relationship-table, or join-table, i.e. `WeekdayFrame` like so:

```
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "weekday_id", insertable = false, updatable =
    false)
    private Weekday weekday;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "frame_id")
    private Frame frame;
```

Listing D.9: The two relations needed on the relationship class in order to create the `Many-to-Many` relationship.

Finally you create the SQL as you would normally:

```
CREATE TABLE Weekday__Frame
(
    id                  BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    weekday_id          BIGINT NOT NULL,
    frame_id            BIGINT NOT NULL,
    weekdayframe_index INT,
    pictoFrameProgress INT,
    CONSTRAINT WeekdayFrame__id_Weekday FOREIGN KEY (weekday_id)
    REFERENCES Weekday (id),
    CONSTRAINT WeekdayFrame__id_Frame FOREIGN KEY (frame_id) REFERENCES
    Frame (id)
);
```

Listing D.10: The SQL needed for the relationship class to create the `Many-to-Many`.

D.3 Javadocs

In order to make the code more readable and easier to use for new developers as well as developers that may not have been included in that specific segment we use Javadocs. For each class, method, non–empty constructor, non–default setters and getters and an enum value Javadocs should be present such that the purpose and how to use the aforementioned constructions can be understood with no more than simply skimming the code. For classes and enum values the Javadocs is quite simple but should exist nonetheless.

```
/**
 * A Weekday is used to contain Frames for the week schedule.
 */
```

The above may be enough for a class. For methods, non–empty constructors and non–default getters and setters a little more information is required, purpose, parameters and return value should all be present.

```
/**
 *  Adds a user to the week schedule.
 *
 * @param user user to add to list.
 * @return boolean
 */
```

Specifically for the service layer it is important that the Javadocs uses link to reference classes as shown in this example, this is required for us to achieve the endpoint documentation mentioned next.

```
/**
 * Add a new {@link User user} to the department. Only guardians can add
    users.
 *
 * @param currentUser The currently authenticated {@link User user}
 * @param newUser     The new {@link User user}to insert
 * @return The {@link User user} that was inserted
 */
```

For the REST API Javadocs serves as more than just documentation and an easy way to understand the cohesion in the code. The Javadocs are also used to provide a clean overview of all the endpoints available in the REST API. This is done through enunciate which uses the Javadocs to create full HTML documentation making implementation for the client side significantly easier by providing hypertext documentation for the endponts. To create and view this, run `./gradlew enunciate` and open the `index.html` file in the `services/dist/docs/api` folder.

D.4 Testing

For the unit tests we use JUnit. As a baseline all methods must be tested regardless of their simplicity. For the Core and Persistence layer we use unit tests and integration tests. The service layer has no formal testing established as it should not contain anything new, still during development one should test whether or not the JSON response is correct.

It should be noted that the integration tests are in a developer environment, thus there is not actual guarantee it works in other environments. The integration tests, located exclusively in the persistence layer alongside unit tests, tests whether the DAO works with the model.

The tests are split into 3 segments, firstly resources are added with the `@Resource` annotation, secondly data is prepared if required using the `@Before` notation and lastly tests, each of which has the `@Test` notation to determine it is a test. For those familiar with the AAA syntax from NUnit `@Resource` and `@Before` serves as the arrange part for all tests. For the Core layer, the unit tests, there will be no `@Resource` as the Core layer is not testing how a resource works with the rest of the system, but simply methods and objects, as such only instantiating objects is necessary, which is handled with the `@Before` notation. For the integration tests `@Resource` is required for all DAOs tested in a given test class.

Unit/Core test example:

```
@Before
public void prepareData(){
    department1 = new Department("meme");
    department1.setId(69L);
    department2 = new Department("dank");
    department2.setId(911L);
    user1 = new User(department1, "test1", "hunter2");
    user2 = new User(department2, "test2", "hunter2");
    user1.setId(1L);
    user2.setId(2L);
}

@Test
public void testHasAccessToPublicPictogram(){
    pictogram1 = new Pictogram("rare", AccessLevel.PUBLIC, user1);
    Assert.assertTrue(pictogram1.hasPermission(user1));
    Assert.assertTrue(pictogram1.hasPermission(user2));
}
```

Integration/Persistence test example:

```
@Resource
private PictogramDao pictogramDao;

@Resource
private UserDao userDao;

@Resource
private DepartmentDao departmentDao;

@Before
public void getDepartment() {
    department = departmentDao.byName("monstertruck");
}

@Test
public void testPictogramGetAll() throws Exception {
    Collection<Pictogram> pictograms =
    pictogramDao.getAllPublicPictograms();
    Assert.assertFalse(pictograms.isEmpty());
}

@Test
public void testPictogramGetAllWithUser() throws Exception {
    User u = userDao.byId(department, 1337L);
    Collection<Pictogram> pictograms = pictogramDao.getAll(u);
    Assert.assertFalse(pictograms.isEmpty());
    Assert.assertEquals(pictograms.size(), 4);
}
```