Engineering of Domain Specific Languages (Fall 2025)

Dr. Andrea Mocci
Assignment 03: Internal DSL (100 + 20 pts)
Deadline (Hard): December 2, 18:59
Version: v1.0, Released on November 18, 2025 at 15:50

Assignment Group. This is a **paired** assignment. Please indicate your group in the Sharepoint Excel file that you will find on iCorsi. Please contact the instructor **ASAP if you are alone because there are no remaining students for a pair.** In general, the group can be the same than the one for Assignment 01 or 02 (but not for both: if you already worked with a given person for both assignment 01 and 02, you need to change pair).

Grading. Grade is based on the following criteria:

- Submitting this assignment satisfying the **basic requirements** (listed at the end of the assignment) will normally **guarantee a base grade of 60%**. Any violation of the basic requirements will imply a penalty (your base grade will be below 60%).
- If you are fine with your base grade, you can opt out the oral discussion, and you will get that grade. You can opt out as a pair, or as a student alone.
- Extra 30% grade (thus, up to 90%) is obtained through discussion. You must answer 4 among the possible questions listed in the **advanced requirement** section, as chosen by the instructor. These questions will not involve in any form the bonus exercise(s). The quality of the submission on that specific aspect and your answer during the discussion will determine the grade. The discussion must be done in pair and each student will answer 2 out of 4 questions. The other student may complement the answer. The grade will be shared if no student will opt out. In exceptional and motivated cases, the students may discuss the assignment separately and get separate grades.
- If you want a grade over 90%, or if you are unsatisfied of the grade you obtained up to now and want to improve it, you will need to answer 2 additional arbitrary questions done by the instructor on your submission, including the bonus points. Any of the two students can opt-out on this additional discussion.

Dates and times for the discussions will be partially flexible and notified soon.

Plagiarism. Any evidence of plagiarism will be contested through mandatory discussion on arbitrary aspects of the assignment. The submission will be potentially void or considered as negative grade. Do not share solutions with other groups of students.

Policy for the use of LLMs. You are allowed to use AI tools based on Large Language Models (LLMs) like CoPilot or ChatGPT for this assignment, but you must ensure the following aspects:

- You must clearly indicate in the submission that you have used an LLM, which one, and how. You may share the interactions done with it.
- You acknowledge full ownership of the solution you provide. The solution needs to use only constructs and syntax forms of Scala that were explained during the lectures. Anything else should be properly justified and motivated, and this motivation could be invalid or contested: Any unnecessary construct will be considered invalid. In particular, in this Scala feature you should not use any advanced programming language feature that has not been explained during the lecture. If you are not sure, please ask upfront.
- In any case, undeclared or unjustified use of AI assistants (using constructs not explained during the lecture) will be considered plagiarism (see above).

Slack Day. Each of you has two slack days. You should communicate your intention to use the slack days **before** the assignment deadline via email. Since the assignment is paired, a request for a slack day will consume a day for both students.

Submission Instructions

Please use the template in iCorsi/Gitlab. Code should compile without errors, *i.e.*, `sbt compile` should run successfully.

Use comments / documentation whenever needed to clarify your choices.

For other exercises, submit a pdf, a markdown, or a text file, with images when needed. Please organize folders in a reasonable way, and submit a zip/tar file. In the rare occurrence that your submission exceeds the size of 20 MB, you have definitely included something unnecessary. *Do not include executable or compiled files (*e.g., *.class files):* you should run `sbt clean` before zipping and submitting the assignment.

Submit your solution via iCorsi as a zip file. You can submit multiple times, but only the last submission will be considered. Late submissions will not be considered.

Note: If you have any specific doubt about Scala, please write me an email with your doubts or use the forum (do not wait the last days before the deadline).

Preamble: Futures and Http Requests

A **Future**[1] in Scala represents a value which may or may not *currently* be available, but will be available at some point, or an exception if that value could not be made available.

It is used as a type that represents an asynchronous computation that will eventually terminate with a value or with a failure. Imagine it like a **Try**, but asynchronous.

In this assignment (excluding the bonus) you will *not* need to manipulate futures, but the methods you will declare may use it as a type. However, a future in scala is typically manipulated with higher order methods like **map**, **flatMap**, **transform**, etc., for which the documentation should be pretty clear. In any case, some minimal future manipulation is only needed for the bonus part.

In this assignment futures are used to represent the asynchronous result of an http GET and POST request on a given URL. The GET method requests to the http server a representation of the specified resource. Requests using GET typically just retrieve data (*e.g.*, a html page, some javascript code, some json data). A POST request instead requests that a web server accepts the data enclosed in the body of the request message. Often, this data is in JSON format (and it will be in such format for this assignment).

A request may result in a response or in a failure (for example because the domain is not valid). A response is a complex data structure which is composed of many elements, in particular:

- a *status code*, which is represented with an integer, and is used to indicate successful responses, non-existing resources, or errors. See `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`;
- a *content type*, which describes the actual content type of the resource (like `text/html`);
- a list of *header fields* or headers, see `https://en.wikipedia.org/wiki/List_of_HTTP_header_fields`;
- a body (like the html contents of a page, or JSON).

[1] See `https://www.scala-lang.org/api/current/scala/concurrent/Future.html`

In this assignment, you have to implement a set of simple internal DSLs, the first one constructing json objects, the second simplified GET and POST requests, and the third one constructing assertions on such requests. Some decision on the internal DSL structure are open, and the way you solve them are part of the exercise itself.

1 Exercise 1: DSLs to build JSON objects (20 pts)

The first DSL to implement involves the construction of JSON objects. Implement these in the `JsonDSL` object in `Exercise1.scala` file.

When implementing the DSL, you can define all the supporting traits, classes, objects, and methods that you need, including extension methods, given values, etc. Add more examples if you need.

The DSL you have to implement has the following syntax:

```
// an empty json object.
val emptyJson = jobj()

/** An object with key-pairs, equivalent to:
  * { "name": "morpheus", "job": "leader" }
  */
val jsonFragment1 = jobj(
  "name" `:` "morpheus",
  "job" `:` "leader"
)

/** A more complex example **/
val jsonFragment2 = jobj(
  "title" `:` "The Matrix",
  "isSequel" `:` false,
  "duration" `:` 136,
  "directors" `:` jarray(jobj(
    "firstName" `:` "Lana",
    "lastName" `:` "Wachowski"
  ), jobj(
    "firstName" `:` "Lilly",
    "lastName" `:` "Wachowski"
  )),
  "starring" `:` jarray(
    "Keanu Reeves", "Laurence Fishburne", "Carrie-Ann Moss"
  ),
  "prequel" `:` null
)
```

To understand what exactly you need to build, please have a look at the signature of the model. In general, there is no need to implement any fluent builder. Printing well-constructed JSON objects will show them in their usual JSON representation.

Another important that you need to look up is something that we did not fully see in the lectures. How does the associativity of certain infix symbolic-named methods interact with extension methods? Before looking it up on the internet (it is fine) try to realize what is particularly hard in this case. If you have any doubt, contact the instructor.

2 Exercise 2: URLs and HTTP Methods (35 pts)

The second internal DSL you have to implement has the following syntax:

```
val getRequest1: Future[Response] =
  (https `://` "www.usi.ch" / "en" / "university").GET

val getRequest2 =
  (https `://` "reqres.in" / "api" / "user" / "1").GET `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  )

val getRequest3 = (http `://` "usi.cch").GET

val postRequest1 =
  (https `://` "reqres.in" / "api" / "users").POST `with` body == jobj(
    "name" `:` "morpheus",
    "job" `:` "leader"
  ) and headers == (
    "x-api-key" -> "reqres-free-v1"
  )

val postRequest2 =
  (https `://` "reqres.in" / "api" / "register").POST `with` body == jobj(
    "email" `:` "agent.smith@reqres.in",
    "password" `:` "OguhGnivaew"
  ) and headers == (
    "x-api-key" -> "reqres-free-v1"
  )

val postRequest3 =
  (https `://` "reqres.in" / "api" / "login").POST `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  )  and body == jobj(
    "email" `:` "morpheus@nebuchadnezzar"
  )
```

The URL structure that you need to support for this exercise includes only the scheme, the domain, and paths. You do not have to check the validity of a domain or the validity of a path element, but just to reconstruct the correct `GetRequest` or `PostRequest` objects. For the scheme, support http and https only.

For the post request, the URL must always be specified before the json object, and after the json object, the DSL must produce the corresponding Future.

Uncomment the corresponding script in the `ch.usi.si.msde.edsl.assignment_03.exercise2` method in the `Exercise2.scala` file and implement the DSL in the `HttpRequestDSL` object in the same file. The solution domain objects to be used for this DSL are implemented in the `HttpRequestModel.scala` file in the model package.

Note that the URL class is modeled with some dedicated classes. You should not touch the model classes at all. You do not necessarily have to use the fluent builder pattern, since the DSL does not use alphabetic infix methods. However, using a similar structure may help you better organize the code. For URLs, **do not support query parameters**.

If everything has been implemented properly, when running the corresponding script extending App you should obtain an output similar (but possibly not identical) to the following one:

```
[info] Success(Response with code 200 and content type text/html)
[info] Success(Response with code 200 and content type application/json)
[info] Failure(org.apache.pekko.stream.StreamTcpException: ... java.net.UnknownHostException: usi.cch)
[info] Success(Response with code 201 and content type application/json)
[info] Success(Response with code 400 and content type application/json)
[info] Success(Response with code 400 and content type application/json)
  ...
```

Please ignore the error lines at the beginning. For proper execution the DSL needs working Internet connection, or all calls may result in failures.

3 Exercise 3: Assertions on Requests (45 pts)

In this exercise, you will have to design a simple DSL for tests (assertions) on GET and POST requests (reusing the DSL of the previous exercise). The entry point to implement the DSL is the `RequestAssertionDSL` trait in the `Exercise3.scala` file.

As with the previous exercise, when implementing the DSL, you can define all the supporting traits, classes, objects, and methods that you need (including extension methods, given conversions, etc.). Add more example assertions to test your DSL, if needed.

An assertion has the following syntax:

```
"a get on user 1 with api key" should "respond with 200 ok" += eventually (
  (https `://` "reqres.in" / "api" / "user" / "1").GET `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  )
) should respond `with` statusCode(200)
```

The first part (before the += operator) is the *assertion description*, which has two strings, the *subject* and the *expected behavior*. After the += operator, each assertion is composed of a call to a GET/POST HTTP method on a url and the expected behavior, which is a response or a failure. A response can be checked against the returned status code, the content type (which is a string), or a specific json object.

```
"a get on user 1 without api key" should "respond with 200 or 401" += eventually (
  (https `://` "reqres.in" / "api" / "user" / "1").GET
) should respond `with` statusCode(200) | statusCode(401)

"a get on user 3 with api key" should "respond with some json" += eventually (
  (https `://` "reqres.in" / "api" / "user" / "3").GET `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  )
) should respond `with` contentType("application/json")

"a get on a non-existing user" should "respond with 404" += eventually (
  (https `://` "reqres.in" / "api" / "user" / "-3721").GET `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  )
) should respond `with` statusCode(404) & contentType("application/json")

"a GET on a non-existing domain" should "fail" += eventually (
  (https `://` "www.usi.chh").GET
 ) should fail

"a login without password" should "respond with contentType json" += eventually (
  (https `://` "reqres.in" / "api" / "login").POST `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  ) and body == jobj(
    "email" `:` "morpheus@nebuchadnezzar"
  )
) should respond `with` contentType("application/json")

"a login without password" should "respond with a json body containing an error" += eventually (
  (https `://` "reqres.in" / "api" / "login").POST `with` headers == (
    "x-api-key" -> "reqres-free-v1"
  ) and body == jobj(
    "email" `:` "morpheus@nebuchadnezzar"
  )
) should respond `with` responseBody(jobj(
  "error" `:` "Missing password"
)) & statusCode(400)
```

Each assertion construct needs to build a `AssertionWithDescription` object, with the description object properly constructed, and an `ExecutableAssertion` object, which is either:

- a `RequestSucceedsWithResponsePredicateAssertion`: It asserts that a request succeeds and its response satisfies a given predicate (*i.e.,* a trait holding a function `Response => Boolean` and a message).
- a `RequestWillFailAssertion`: This happens when for example there is an error processing the request (network errors, typically). Internal Server Errors (500) are not considered failures.

`RequestSucceedsWithResponsePredicateAssertion` takes a `ResponsePredicate`, of which three implementations exist: `ResponseHasStatusCodeEqualsToPredicate`, that represents a a predicate for a response having a specific status code; `ResponseHasContentTypeEqualsToPredicate` that represents a predicate for a response having a specific content type; and `ResponseContainsJson` that represents a predicate for a response having a json body matching a given JSON object. Predicates should be combined with | and & operators, which correspond to classic boolean operators.

The get and post requests **must not get started (i.e., evaluated in any form) before the assertion execution**.

You do not need to explicitly execute the assertion at any point in the DSL. Instead, you must insert the `AssertionWithDescription` object in the `namedAssertions` var in `AssertionExecutor` trait, which is already extended by the `RequestAssertionDSL` trait. Think carefully when this needs to be performed, in particular by reasoning about the precedence of methods in the whole DSL.

Please note that this DSL is not properly a fluent builder like we have seen in the second lecture of Internal DSL. A part of it resembles a fluent builder, but part of it (the predicates) is essentially composed of nested functions.

4 Exercise 4: Bonus (Hard, 20 pts)

In a new file, called `Exercise4.scala`, change the DSL in Exercise 3 to support boolean expressions on predicates but using non-symbolic methods only, called `and` and `or`. Their internal semantics needs to satisfy the same association and precedence rules of classic boolean operators, even if the expressions are parsed differently by Scala rules.

Basic Requirements for Assignment 03

- You have attempted all non-bonus (penalty proportional to exercise weight).
- You have properly designed the internal DSL according to what you have seen in the lectures (Penalty: up to 100%).
- The code compiles (Penalty: min 40%, max 80%).
- The code runs as expected, reconstructing the required objects (Penalty: proportional to 60% of each exercise where there are errors).
- The code contains only programming language constructs and features as explained up to Lecture 08. Anything else is justified properly and can has been studied properly by the student (Penalty: up to 100%).
- You do not modify the model classes for basic exercises (Penalty: up to 100%).

Questions for Advanced Requirements for Assignment 03

Whenever the questions mention a *specific* element (e.g., class, trait, rule, semantc action), the instructor will point out to a specific instance of that element in your assignment.

Questions

- How is this expression parsed in Scala?
- What is the precedence of the calls involved in this expression?
- Can you explain how this method is actually executed?
- How are the properties of this fluent builder modeled?
- Can you explain why did you use this specific language feature here?
- Can you explain how many conversions are applied in this piece of code?
- Can you discuss which methods in this piece of code are extension methods?
- Why did you use this conversion here, and where is it applied?
- What is the associativity of the calls involved in this expression?
- If I change this method name into a (symbolic, non-symbolic) method name, what changes?
- Can you explain the design of this fluent builder?