

Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

[Read the guide](#)

Usage

[Jump to bottom](#)

Johan Haleby edited this page on Mar 13 · 189 revisions

Note that if you're using version 1.9.0 or earlier please refer to the [legacy](#) documentation.

REST Assured is a Java DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests.

Contents

1. [Static imports](#)
2. [Examples](#)
 - i. [JSON Example](#)
 - ii. [JSON Schema Validation](#)
 - iii. [XML Example](#)
 - iv. [Advanced](#)
 - a. [XML](#)
 - b. [JSON](#)
 - v. [De-serialization with Generics](#)
 - vi. [Additional Examples](#)
3. [Note on floats and doubles](#)

4. [Note on syntax \(syntactic sugar\)](#)
5. [Getting Response Data](#)
 - i. [Extracting values from the Response after validation](#)
 - ii. [JSON \(using JsonPath\)](#)
 - iii. [XML \(using XmlPath\)](#)
 - iv. [Single Path](#)
 - v. [Headers, cookies, status etc](#)
 - a. [Multi-value headers](#)
 - b. [Multi-value cookies](#)
 - c. [Detailed Cookies](#)
6. [Specifying Request Data](#)
 - i. [Invoking HTTP resources](#)
 - ii. [Parameters](#)
 - iii. [Multi-value Parameter](#)
 - iv. [No-value Parameter](#)
 - v. [Path Parameters](#)
 - vi. [Cookies](#)
 - vii. [Headers](#)
 - viii. [Content-Type](#)
 - ix. [Request Body](#)
7. [Verifying Response Data](#)
 - i. [Response Body](#)
 - ii. [Cookies](#)
 - iii. [Status](#)
 - iv. [Headers](#)
 - v. [Content-Type](#)
 - vi. [Full body/content matching](#)
 - vii. [Use the response to verify other parts of the response](#)
 - viii. [Measuring response time](#)
8. [Authentication](#)
 - i. [Basic](#)
 - a. [Preemptive](#)
 - b. [Challenged](#)
 - ii. [Digest](#)
 - iii. [Form](#)
 - a. [CSRF](#)

- b. Include additional fields in Form Authentication
 - iv. OAuth
 - a. OAuth1
 - b. OAuth2
- 9. Multi-part form data
- 10. Object Mapping
 - i. Serialization
 - a. Content-Type based Serialization
 - b. Create JSON from a HashMap
 - c. Using an Explicit Serializer
 - ii. Deserialization
 - a. Content-Type based Deserialization
 - b. Custom Content-Type Deserialization
 - c. Using an Explicit Deserializer
 - iii. Configuration
 - iv. Custom
- 11. Parsers
 - i. Custom
 - ii. Default
- 12. Default Values
- 13. Specification Re-use
 - i. Querying RequestSpecification
- 14. Filters
 - i. Ordered Filters
 - ii. Response Builder
- 15. Logging
 - i. Request Logging
 - ii. Response Logging
 - iii. Log if validation fails
 - iv. Blacklist Headers from Logging
- 16. Root Path
 - i. Path Arguments
- 17. Session Support
 - i. Session Filter
- 18. SSL
 - i. SSL invalid hostname

- 19. [URL Encoding](#)
- 20. [Proxy Configuration](#)
 - i. [Static Proxy Configuration](#)
 - ii. [Request Specification Proxy Configuration](#)
- 21. [Detailed configuration](#)
 - i. [Encoder Config](#)
 - ii. [Decoder Config](#)
 - iii. [Session Config](#)
 - iv. [Redirect DSL](#)
 - v. [Connection Config](#)
 - vi. [JSON Config](#)
 - vii. [HTTP Client Config](#)
 - viii. [SSL Config](#)
 - ix. [Param Config](#)
 - x. [Failure Config](#)
- 22. [Spring Support](#)
 - i. [Spring Mock Mvc Module](#)
 - a. [Bootstrapping RestAssuredMockMvc](#)
 - b. [Asynchronous Requests](#)
 - c. [Adding Request Post Processors](#)
 - d. [Adding Result Handlers](#)
 - e. [Using Result Matchers](#)
 - f. [Interceptors](#)
 - g. [Specifications](#)
 - h. [Resetting RestAssuredMockMvc](#)
 - i. [Spring MVC Authentication](#)
 - a. [Using Spring Security Test](#)
 - b. [Injecting a User](#)
 - j. [Kotlin Extension Module for Spring MockMvc](#)
 - ii. [Spring Web Test Client Module](#)
 - a. [Bootstrapping RestAssuredWebTestClient](#)
 - b. [Specifications](#)
 - c. [Resetting RestAssuredWebTestClient](#)
 - iii. [Common Spring Module Documentation](#)
 - a. [Note on parameters](#)
- 23. [Scala Support Module](#)

24. [Kotlin](#)

- i. [Avoid Escaping "when" Keyword](#)
- ii. [Kotlin Extension Module](#)
- iii. [Kotlin Extension Module for Spring MockMvc](#)

25. [More Info](#)

Static imports

In order to use REST assured effectively it's recommended to statically import methods from the following classes:

```
io.restassured.RestAssured.*
io.restassured.matcher.RestAssuredMatchers.*
org.hamcrest.Matchers.*
```

If you want to use [Json Schema](#) validation you should also statically import these methods:

```
io.restassured.module.json.JsonSchemaValidator.*
```

Refer to [Json Schema Validation](#) section for more info.

If you're using Spring MVC you can use the [spring-mock-mvc](#) module to unit test your Spring Controllers using the Rest Assured DSL. To do this statically import the methods from [RestAssuredMockMvc](#) *instead* of importing the methods from `io.restassured.RestAssured` :

```
io.restassured.module.mockmvc.RestAssuredMockMvc.*
```

Examples

Example 1 - JSON

Assume that the GET request (to <http://localhost:8080/lotto>) returns JSON as:

```
{
  "lotto":{
    "lottoId":5,
    "winning-numbers":[2,45,34,23,7,5,3],
    "winners":[{
```

```

    "winnerId":23,
    "numbers":[2,45,34,23,3,5]
  },{
    "winnerId":54,
    "numbers":[52,3,12,11,18,22]
  }]
}
}

```

REST assured can then help you to easily make the GET request and verify the response. E.g. if you want to verify that `lottoid` is equal to 5 you can do like this:

```
get("/lotto").then().body("lotto.lottoId", equalTo(5));
```

or perhaps you want to check that the `winnerId`'s are 23 and 54:

```
get("/lotto").then().body("lotto.winners.winnerId", hasItems(23, 54));
```

Note: `equalTo` and `hasItems` are Hamcrest matchers which you should statically import from `org.hamcrest.Matchers`.

Note that the "json path" syntax uses [Groovy's GPath](#) notation and is not to be confused with Jayway's [JsonPath](#) syntax.

Returning floats and doubles as BigDecimal

You can configure Rest Assured and JsonPath to return `BigDecimal`'s instead of float and double for Json Numbers. For example consider the following JSON document:

```

{
    "price":12.12
}

```

By default you validate that price is equal to 12.12 as a float like this:

```
get("/price").then().body("price", is(12.12f));
```

but if you like you can configure REST Assured to use a `JsonConfig` that returns all Json numbers as `BigDecimal`:

```

given().
    config(RestAssured.config().jsonConfig(jsonConfig().numberReturnType(BIG_DECIM
when().
    get("/price").
then().
    body("price", is(new BigDecimal(12.12)));

```

JSON Schema validation

From version 2.1.0 REST Assured has support for [Json Schema](#) validation. For example given the following schema located in the classpath as `products-schema.json` :

```

{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product set",
  "type": "array",
  "items": {
    "title": "Product",
    "type": "object",
    "properties": {
      "id": {
        "description": "The unique identifier for a product",
        "type": "number"
      },
      "name": {
        "type": "string"
      },
      "price": {
        "type": "number",
        "minimum": 0,
        "exclusiveMinimum": true
      },
      "tags": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "minItems": 1,
        "uniqueItems": true
      },
      "dimensions": {
        "type": "object",
        "properties": {
          "length": {"type": "number"},
          "width": {"type": "number"},
          "height": {"type": "number"}
        }
      }
    }
  }
}

```

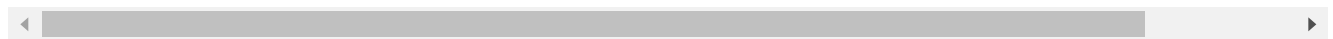
```

        "required": ["length", "width", "height"]
    },
    "warehouseLocation": {
        "description": "Coordinates of the warehouse with the product",
        "$ref": "http://json-schema.org/geo"
    }
},
"required": ["id", "name", "price"]
}
}

```

you can validate that a resource (`/products`) conforms with the schema:

```
get("/products").then().assertThat().body(matchesJsonSchemaInClasspath("products-schem
```



`matchesJsonSchemaInClasspath` is statically imported from `io.restassured.module.json.JsonSchemaValidator` and it's recommended to statically import all methods from this class. However in order to use it you need to depend on the `json-schema-validator` module by either [downloading](#) it from the download page or add the following dependency from Maven:

```

<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>json-schema-validator</artifactId>
  <version>4.3.0</version>
</dependency>

```

JSON Schema Validation Settings

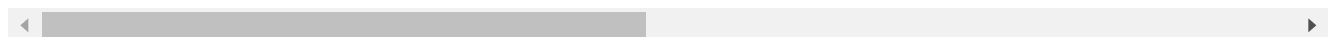
REST Assured's `json-schema-validator` module uses Francis Galiegue's [json-schema-validator](#) (`fge`) library to perform validation. If you need to configure the underlying `fge` library you can for example do like this:

```

// Given
JsonSchemaFactory jsonSchemaFactory = JsonSchemaFactory.newBuilder().setValidationConf

// When
get("/products").then().assertThat().body(matchesJsonSchemaInClasspath("products-schem

```



The `using` method allows you to pass in a `jsonSchemaFactory` instance that REST Assured will use during validation. This allows fine-grained configuration for the validation.

The `fge` library also allows the validation to be `checked` or `unchecked`. By default REST Assured uses `checked` validation but if you want to change this you can supply an instance of [JsonSchemaValidatorSettings](#) to the matcher. For example:

```
get("/products").then().assertThat().body(matchesJsonSchemaInClasspath("products-schem
```

Where the `settings` method is statically imported from the [JsonSchemaValidatorSettings](#) class.

Json Schema Validation with static configuration

Now imagine that you always want to use `unchecked` validation as well as setting the default json schema version to version 3. Instead of supplying this to all matchers throughout your code you can define it statically. For example:

```
JsonSchemaValidator.settings = settings().with().jsonSchemaFactory(  
    JsonSchemaFactory.newBuilder().setValidationConfiguration(ValidationConfigurat  
    and().with().checkedValidation(false);
```

```
get("/products").then().assertThat().body(matchesJsonSchemaInClasspath("products-schem
```

Now any `matcher` method imported from [JsonSchemaValidator](#) will use `DRAFTV3` as default version and `unchecked` validation.

To reset the `JsonSchemaValidator` to its default settings simply call the `reset` method:

```
JsonSchemaValidator.reset();
```

Json Schema Validation without REST Assured

You can also use the `json-schema-validator` module without depending on REST Assured. As long as you have a JSON document represented as a `String` you can do like this:

```
import org.junit.Test;  
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspa  
import static org.hamcrest.MatcherAssert.assertThat;  
  
public class JsonSchemaValidatorWithoutRestAssuredTest {
```

```
@Test public void
validates_schema_in_classpath() {
    // Given
    String json = ... // Greeting response

    // Then
    assertThat(json, matchesJsonSchemaInClasspath("greeting-schema.json"));
}
}
```

Refer to the [getting started](#) page for more info on this.

Anonymous JSON root validation

A JSON document doesn't necessarily need a named root attribute. This is for example valid JSON:

```
[1, 2, 3]
```

An anonymous JSON root can be verified by using `$` or an empty string as path. For example let's say that this JSON document is exposed from `http://localhost:8080/json` then we can validate it like this with REST Assured:

```
when().
    get("/json").
then().
    body("$", hasItems(1, 2, 3)); // An empty string "" would work as well
```

Example 2 - XML

XML can be verified in a similar way. Imagine that a POST request to `http://localhost:8080/greetXML` returns:

```
<greeting>
  <firstName>{params("firstName")}</firstName>
  <lastName>{params("lastName")}</lastName>
</greeting>
```

i.e. it sends back a greeting based on the `firstName` and `lastName` parameter sent in the request. You can easily perform and verify e.g. the `firstName` with REST assured:

```

given().
    parameters("firstName", "John", "lastName", "Doe").
when().
    post("/greetXML").
then().
    body("greeting.firstName", equalTo("John")).

```

If you want to verify both `firstName` and `lastName` you may do like this:

```

given().
    parameters("firstName", "John", "lastName", "Doe").
when().
    post("/greetXML").
then().
    body("greeting.firstName", equalTo("John")).
    body("greeting.lastName", equalTo("Doe"));

```

or a little shorter:

```

with().parameters("firstName", "John", "lastName", "Doe").when().post("/greetXML").the

```



See [this](#) link for more info about the syntax (it follows Groovy's [GPath](#) syntax).

XML namespaces

To make body expectations take namespaces into account you need to declare the namespaces using the [io.restassured.config.XmlConfig](#). For example let's say that a resource called `namespace-example` located at `http://localhost:8080` returns the following XML:

```

<foo xmlns:ns="http://localhost/">
  <bar>sudo </bar>
  <ns:bar>make me a sandwich!</ns:bar>
</foo>

```

You can then declare the `http://localhost/` uri and validate the response:

```

given().
    config(RestAssured.config().xmlConfig(xmlConfig().declareNamespace("test", "ht
when().
    get("/namespace-example").
then().
    body("foo.bar.text()", equalTo("sudo make me a sandwich!")).

```

```
body(":foo.:bar.text()", equalTo("sudo ")).  
body("foo.test:bar.text()", equalTo("make me a sandwich!"));
```

The path syntax follows Groovy's XmlSlurper syntax. Note that in versions prior to 2.6.0 the path syntax was *not* following Groovy's XmlSlurper syntax. Please see [release notes](#) for version 2.6.0 to see how the previous syntax looked like.

XPath

You can also verify XML responses using x-path. For example:

```
given().parameters("firstName", "John", "lastName", "Doe").when().post("/greetXML").th
```

or

```
given().parameters("firstName", "John", "lastName", "Doe").post("/greetXML").then().bo
```

To use namespaces in the XPath expression you need to enable them in the configuration, for example:

```
given().  
    config(RestAssured.config().xmlConfig(xmlConfig().with().namespaceAware(true)))  
when().  
    get("/package-db-xml").  
then().  
    body(hasXPath("/db:package-database", namespaceContext));
```

Where `namespaceContext` is an instance of [javax.xml.namespace.NamespaceContext](#).

Schema and DTD validation

XML response bodies can also be verified against an XML Schema (XSD) or DTD.

XSD example

```
get("/carRecords").then().assertThat().body(matchesXsd(xsd));
```

DTD example

```
get("/videos").then().assertThat().body(matchesDtd(dtd));
```

The `matchesXsd` and `matchesDtd` methods are Hamcrest matchers which you can import from [io.restassured.matcher.RestAssuredMatchers](#).

Example 3 - Complex parsing and validation

This is where REST Assured really starts to shine! Since REST Assured is implemented in Groovy it can be really beneficial to take advantage of Groovy's collection API. Let's begin by looking at an example in Groovy:

```
def words = ['ant', 'buffalo', 'cat', 'dinosaur']
def wordsWithSizeGreaterThanFour = words.findAll { it.length() > 4 }
```

At the first line we simply define a list with some words but the second line is more interesting. Here we search the words list for all words that are longer than 4 characters by calling the `findAll` with a Groovy closure. The closure has an implicit variable called `it` which represents the current item in the list. The result is a new list, `wordsWithSizeGreaterThanFour`, containing `buffalo` and `dinosaur`.

There are other interesting methods that we can use on collections in Groovy as well, for example:

- `find` – finds the first item matching a closure predicate
- `collect` – collect the return value of calling a closure on each item in a collection
- `sum` – Sum all the items in the collection
- `max / min` – returns the max/min values of the collection

So how do we take advantage of this when validating our XML or JSON responses with REST Assured?

XML Example

Let's say we have a resource at `http://localhost:8080/shopping` that returns the following XML:

```
<shopping>
  <category type="groceries">
    <item>Chocolate</item>
```

```

    <item>Coffee</item>
  </category>
  <category type="supplies">
    <item>Paper</item>
    <item quantity="4">Pens</item>
  </category>
  <category type="present">
    <item when="Aug 10">Kathryn's Birthday</item>
  </category>
</shopping>

```

Let's also say we want to write a test that verifies that the category of type groceries has items Chocolate and Coffee. In REST Assured it can look like this:

```
t("/shopping").
```

```
dy("shopping.category.find { it.@type == 'groceries' }.item", hasItems("Chocolate", "Cof-
```

What's going on here? First of all the XML path `shopping.category` returns a list of all categories. On this list we invoke a function, `find`, to return the single category that has the XML attribute, `type`, equal to `groceries`. On this category we then continue by getting all the items associated with this category. Since there are more than one item associated with the groceries category a list will be returned and we verify this list against the `hasItems` Hamcrest matcher.

But what if you want to get the items and not validate them against a Hamcrest matcher? For this purpose you can use [XPath](#):

```

// Get the response body as a String
String response = get("/shopping").asString();
// And get the groceries from the response. "from" is statically imported from the Xml
List<String> groceries = from(response).getList("shopping.category.find { it.@type ==

```

If the list of groceries is the only thing you care about in the response body you can also use a [shortcut](#):

```

response body as a String
> groceries = get("/shopping").path("shopping.category.find { it.@type == 'groceries' }").

```

Depth-first search

It's actually possible to simplify the previous example even further:

```
when().  
    get("/shopping").  
then().  
    body("**.find { it.@type == 'groceries' }", hasItems("Chocolate", "Coffee"));
```

`**` is a shortcut for doing depth first searching in the XML document. We search for the first node that has an attribute named `type` equal to `"groceries"`. Notice also that we don't end the XML path with `"item"`. The reason is that `toString()` is called automatically on the category node which returns a list of the item values.

JSON Example

Let's say we have a resource at `http://localhost:8080/store` that returns the following JSON document:

```
{  
  "store": {  
    "book": [  
      {  
        "author": "Nigel Rees",  
        "category": "reference",  
        "price": 8.95,  
        "title": "Sayings of the Century"  
      },  
      {  
        "author": "Evelyn Waugh",  
        "category": "fiction",  
        "price": 12.99,  
        "title": "Sword of Honour"  
      },  
      {  
        "author": "Herman Melville",  
        "category": "fiction",  
        "isbn": "0-553-21311-3",  
        "price": 8.99,  
        "title": "Moby Dick"  
      },  
      {  
        "author": "J. R. R. Tolkien",  
        "category": "fiction",  
        "isbn": "0-395-19395-8",  
        "price": 22.99,  
        "title": "The Lord of the Rings"  
      }  
    ]  
  }  
}
```

```

    }
  ]
}

```

Example 1

As a first example let's say we want to make the request to `/store` and assert that the titles of the books with a price less than 10 are "Sayings of the Century" and "Moby Dick":

```

when().
    get("/store").
then().
    body("store.book.findAll { it.price < 10 }.title", hasItems("Sayings of the Cen

```

Just as in the XML examples above we use a closure to find all books with a price less than 10 and then return the titles of all the books. We then use the `hasItems` matcher to assert that the titles are the ones we expect. Using [JsonPath](#) we can return the titles instead:

```

// Get the response body as a String
String response = get("/store").asString();
// And get all books with price < 10 from the response. "from" is statically imported
List<String> bookTitles = from(response).getList("store.book.findAll { it.price < 10 }

```

Example 2

Let's consider instead that we want to assert that the sum of the length of all author names are greater than 50. This is a rather complex question to answer and it really shows the strength of closures and Groovy collections. In REST Assured it looks like this:

```

when().
    get("/store");
then().
    body("store.book.author.collect { it.length() }.sum()", greaterThan(50));

```


First we get all the authors (`store.book.author`) and invoke the `collect` method on the resulting list with the closure `{ it.length() }`. What it does is to call the `length()` method on each author in the list and returns the result to a new list. On this list we simply call the `sum()` method to sum all the length's. The end result is `53` and we assert that it's greater than 50 by using the `greaterThan` matcher. But it's actually possible to simplify this even further. Consider the "words" example again:

```
def words = ['ant', 'buffalo', 'cat', 'dinosaur']
```

Groovy has a very handy way of calling a function for each element in the list by using the spread operator, `*`. For example:

```
def words = ['ant', 'buffalo', 'cat', 'dinosaur']
assert [3, 6, 3, 8] == words*.length()
```

I.e. Groovy returns a new list with the lengths of the items in the words list. We can utilize this for the author list in REST Assured as well:

```
when().
    get("/store");
then().
    body("store.book.author*.length().sum()", greaterThan(50)).
```

And of course we can use `JsonPath` to actually return the result:

```
// Get the response body as a string
String response = get("/store").asString();
// Get the sum of all author length's as an int. "from" is again statically imported f
int sumOfAllAuthorLengths = from(response).getInt("store.book.author*.length().sum()")
// We can also assert that the sum is equal to 53 as expected.
assertThat(sumOfAllAuthorLengths, is(53));
```



Deserialization with Generics

REST Assured 3.3.0 introduced the `io.restassured.mapper.TypeRef` class that allows you to de-serialize the response to a container with a generic type. For example let's say that you have a service that returns the following JSON for a GET request to `/products` :

```
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
      "length": 7.0,
      "width": 12.0,
      "height": 9.5
    },
    "warehouseLocation": {
      "latitude": -78.75,
      "longitude": 20.4
    }
  },
  {
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
      "length": 3.1,
      "width": 1.0,
      "height": 1.0
    },
    "warehouseLocation": {
      "latitude": 54.4,
      "longitude": -32.7
    }
  }
]
```

You can then extract the root list to a `List<Map<String, Object>>` (or a any generic container of choice) using the `TypeRef` :

```
// Extract
List<Map<String, Object>> products = get("/products").as(new TypeRef<List<Map<String, Object>>(){});

// Now you can do validations on the extracted objects:
assertThat(products, hasSize(2));
assertThat(products.get(0).get("id"), equalTo(2));
assertThat(products.get(0).get("name"), equalTo("An ice sculpture"));
assertThat(products.get(0).get("price"), equalTo(12.5));
assertThat(products.get(1).get("id"), equalTo(3));
assertThat(products.get(1).get("name"), equalTo("A blue mouse"));
assertThat(products.get(1).get("price"), equalTo(25.5));``
```

Note that currently this only works for JSON responses.

Additional Examples

Micha Kops has written a really good blog with several examples (including code examples that you can checkout). You can read it [here](#).

Also [Bas Dijkstra](#) has been generous enough to open source his REST Assured workshop. You can read more about this [here](#) and you can try out, and contribute to, the exercises available in [his](#) github repository.

Bas has also made a nice introductory screencast to REST Assured, you can find it [here](#).

Note on floats and doubles

Floating point numbers must be compared with a Java "float" primitive. For example, if we consider the following JSON object:

```
{  
  "price":12.12  
}
```

the following test will fail, because we compare with a "double" instead of a "float":

```
get("/price").then().assertThat().body("price", equalTo(12.12));
```

Instead, compare with a float with:

```
get("/price").then().assertThat().body("price", equalTo(12.12f));
```

Note on syntax

When reading blogs about REST Assured you may see a lot of examples using the "given / expect / when" syntax, for example:

```
given().  
    param("x", "y").  
expect().
```

```
        body("lotto.lottoId", equalTo(5)).
when().
    get("/lotto");
```

This is the so called "legacy syntax" which was the de facto way of writing tests in REST Assured 1.x. While this works fine it turned out to be quite confusing and annoying for many users. The reason for not using "given / when / then" in the first place was mainly technical. So prior to REST Assured 2.0 there was no support "given / when / then" which is more or less the standard approach when you're doing some kind of BDD-like testing. The "given / expect / when" approach still works fine in 2.0 and above but "given / when / then" reads better and is easier to understand for most people and is thus recommended in most cases. There's however one benefit of using the "given / expect / when" approach and that is that ALL expectation errors can be displayed at the same time which is not possible with the new syntax (since the expectations are defined last). This means that if you would have had multiple expectations in the previous example such as

```
given().
    param("x", "y").
expect().
    statusCode(400).
    body("lotto.lottoId", equalTo(6)).
when().
    get("/lotto");
```

REST Assured will report that both the status code expectation and the body expectation are wrong. Rewriting this with the new syntax

```
given().
    param("x", "y").
when().
    get("/lotto").
then().
    statusCode(400).
    body("lotto.lottoId", equalTo(6));
```

will only report an error at the first failed expectation / assertion (that status code was expected to be 400 but it was actually 200). You would have to re-run the test in order to catch the second error.

Syntactic Sugar

Another thing worth mentioning is that REST Assured contains some methods that are only there for syntactic sugar. For example the "and" method which can add readability if you're writing everything in a one-liner, for example:

```
given().param("x", "y").and().header("z", "w").when().get("/something").then().assertTl
```

This is the same thing as:

```
given().
    param("x", "y").
    header("z", "w").
when().
    get("/something").
then().
    statusCode(200).
    body("x.y", equalTo("z"));
```

Getting Response Data

You can also get the content of a response. E.g. let's say you want to return the body of a get request to "/lotto". You can get it a variety of different ways:

```
InputStream stream = get("/lotto").asInputStream(); // Don't forget to close this one
byte[] byteArray = get("/lotto").asByteArray();
String json = get("/lotto").asString();
```

Extracting values from the Response after validation

You can extract values from the response or return the response instance itself after you've done validating the response by using the `extract` method. This is useful for example if you want to use values from the response in sequent requests. For example given that a resource called `title` returns the following JSON

```
{
  "title" : "My Title",
  "_links": {
    "self": { "href": "/title" },
    "next": { "href": "/title?page=2" }
```

```
}  
}
```

and you want to validate that content type is equal to `JSON` and the title is equal to `My Title` but you also want to extract the link to the `next` title to use that in a subsequent request. This is how:

```
String nextTitleLink =  
given().  
    param("param_name", "param_value").  
when().  
    get("/title").  
then().  
    contentType(JSON).  
    body("title", equalTo("My Title")).  
extract().  
    path("_links.next.href");  
  
get(nextTitleLink). ..
```

You could also decide to instead return the entire response if you need to extract multiple values from the response:

```
Response response =  
given().  
    param("param_name", "param_value").  
when().  
    get("/title").  
then().  
    contentType(JSON).  
    body("title", equalTo("My Title")).  
extract().  
    response();  
  
String nextTitleLink = response.path("_links.next.href");  
String headerValue = response.header("headerName");
```

JSON (using JsonPath)

Once we have the response body we can then use the `JsonPath` to get data from the response body:

```
int lottoId = from(json).getInt("lotto.lottoId");  
List<Integer> winnerIds = from(json).get("lotto.winners.winnerId");
```

Or a bit more efficiently:

```
JsonPath jsonPath = new JsonPath(json).setRoot("lotto");
int lottoId = jsonPath.getInt("lottoId");
List<Integer> winnerIds = jsonPath.get("winners.winnderId");
```

Note that you can use `JsonPath` standalone without depending on REST Assured, see [getting started guide](#) for more info on this.

JsonPath Configuration

You can configure object de-serializers etc for `JsonPath` by configuring it, for example:

```
JsonPath jsonPath = new JsonPath(SOME_JSON).using(new JsonPathConfig("UTF-8"));
```

It's also possible to configure `JsonPath` statically so that all instances of `JsonPath` will shared the same configuration:

```
JsonPath.config = new JsonPathConfig("UTF-8");
```

You can read more about `JsonPath` at [this blog](#).

Note that the `JsonPath` implementation uses [Groovy's GPath](#) syntax and is not to be confused with Jayway's [JsonPath](#) implementation.

XML (using XmlPath)

You also have the corresponding functionality for XML using [XmlPath](#):

```
String xml = post("/greetXML?firstName=John&lastName=Doe").andReturn().asString();
// Now use XmlPath to get the first and last name
String firstName = from(xml).get("greeting.firstName");
String lastName = from(xml).get("greeting.lastName");

// or a bit more efficiently:
XmlPath xmlPath = new XmlPath(xml).setRoot("greeting");
String firstName = xmlPath.get("firstName");
String lastName = xmlPath.get("lastName");
```

Note that you can use `XmlPath` standalone without depending on REST Assured, see [getting started guide](#) for more info on this.

XmlPath Configuration

You can configure object de-serializers and charset for `XmlPath` by configuring it, for example:

```
XmlPath xmlPath = new XmlPath(SOME_XML).using(new XmlPathConfig("UTF-8"));
```

It's also possible to configure `XmlPath` statically so that all instances of `XmlPath` will shared the same configuration:

```
XmlPath.config = new XmlPathConfig("UTF-8");
```

You can read more about `XmlPath` at [this blog](#).

Parsing HTML with XmlPath

By configuring `XmlPath` with `compatibility mode HTML` you can also use the `XmlPath` syntax (Gpath) to parse HTML pages. For example if you want to extract the title of this HTML document:

```
<html>
<head>
  <title>my title</title>
</head>
<body>
  <p>paragraph 1</p>
  <br>
  <p>paragraph 2</p>
</body>
</html>
```

you can configure `XmlPath` like this:

```
String html = ...
XmlPath xmlPath = new XmlPath(CompatibilityMode.HTML, html);
```

and then extract the title like this:


```
xmlPath.getString("html.head.title"); // will return "mytitle"
```

In this example we've statically imported:

```
io.restassured.path.xml.XmlPath.CompatibilityMode.HTML ;
```

Single path

If you only want to make a request and return a single path you can use a shortcut:

```
int lottoId = get("/lotto").path("lotto.lottoid");
```

REST Assured will automatically determine whether to use JsonPath or XmlPath based on the content-type of the response. If no content-type is defined then REST Assured will try to look at the [default parser](#) if defined. You can also manually decide which path instance to use, e.g.

```
String firstName = post("/greetXML?firstName=John&lastName=Doe").andReturn().xmlPath()
```



Options are `xmlPath`, `jsonPath` and `htmlPath`.

Headers, cookies, status etc

You can also get headers, cookies, status line and status code:

```
Response response = get("/lotto");

// Get all headers
Headers allHeaders = response.getHeaders();
// Get a single header value:
String headerName = response.getHeader("headerName");

// Get all cookies as simple name-value pairs
Map<String, String> allCookies = response.getCookies();
// Get a single cookie value:
String cookieValue = response.getCookie("cookieName");

// Get status line
String statusLine = response.getStatusLine();
// Get status code
int statusCode = response.getStatusCode();
```

Multi-value headers and cookies

A header and a cookie can contain several values for the same name.

Multi-value headers

To get all values for a header you need to first get the [Headers](#) object from the [Response](#) object. From the `Headers` instance you can get all values using the [Headers.getValues\(\)](#) method which returns a `List` with all header values.

Multi-value cookies

To get all values for a cookie you need to first get the [Cookies](#) object from the [Response](#) object. From the `Cookies` instance you can get all values using the [Cookies.getValues\(\)](#) method which returns a `List` with all cookie values.

Detailed Cookies

If you need to get e.g. the comment, path or expiry date etc from a cookie you need get a [detailed cookie](#) from REST Assured. To do this you can use the [Response.getDetailedCookie\(java.lang.String\)](#) method. The detailed cookie then contains all attributes from the cookie.

You can also get all detailed response [cookies](#) using the [Response.getDetailedCookies\(\)](#) method.

Specifying Request Data

Besides specifying request parameters you can also specify headers, cookies, body and content type.

Invoking HTTP resources

You typically perform a request by calling any of the "HTTP methods" in the [request specification](#). For example:

```
when().get("/x"). ..;
```

Where `get` is the HTTP request method.

As of REST Assured 3.0.0 you can use any HTTP verb with your request by making use of the [request](#) method.

```
when().  
    request("CONNECT", "/somewhere").  
then().  
    statusCode(200);
```

This will send a "connect" request to the server.

Parameters

Normally you specify parameters like this:

```
given().  
    param("param1", "value1").  
    param("param2", "value2").  
when().  
    get("/something");
```

REST Assured will automatically try to determine which parameter type (i.e. query or form parameter) based on the HTTP method. In case of GET query parameters will automatically be used and in case of POST form parameters will be used. In some cases it's however important to separate between form and query parameters in a PUT or POST. You can then do like this:

```
given().  
    formParam("formParamName", "value1").  
    queryParams("queryParamsName", "value2").  
when().  
    post("/something");
```

Parameters can also be set directly on the url:

```
..when().get("/name?firstName=John&lastName=Doe");
```

For multi-part parameters please refer to the [Multi-part form data](#) section.

Multi-value parameter

Multi-value parameters are parameters with more than one value per parameter name (i.e. a list of values per name). You can specify these either by using var-args:

```
given().param("myList", "value1", "value2"). ..
```

or using a list:

```
List<String> values = new ArrayList<String>();  
values.add("value1");  
values.add("value2");  
  
given().param("myList", values). ..
```

No-value parameter

You can also specify a query, request or form parameter without a value at all:

```
given().param("paramName"). ..
```

Path parameters

You can also specify so called path parameters in your request, e.g.

```
post("/reserve/{hotelId}/{roomNumber}", "My Hotel", 23);
```

These kinds of path parameters are referred to "unnamed path parameters" in REST Assured since they are index based (`hotelId` will be equal to "My Hotel" since it's the first placeholder).

You can also use named path parameters:

```
given().  
    pathParam("hotelId", "My Hotel").  
    pathParam("roomNumber", 23).  
when().  
    post("/reserve/{hotelId}/{roomNumber}").  
then().  
    ..
```

Path parameters makes it easier to read the request path as well as enabling the request path to easily be re-usable in many tests with different parameter values.

As of version 2.8.0 you can mix unnamed and named path parameters:

```

given().
    pathParam("hotelId", "My Hotel").
when().
    post("/reserve/{hotelId}/{roomNumber}", 23).
then().
    ..

```

Here `roomNumber` will be replaced with `23`.

Note that specifying too few or too many parameters will result in an error message. For advanced use cases you can add, change, remove (even redundant path parameters) from a [filter](#).

Cookies

In its simplest form you specify cookies like this:

```

given().cookie("username", "John").when().get("/cookie").then().body(equalTo("username

```

You can also specify a multi-value cookie like this:

```

given().cookie("cookieName", "value1", "value2"). ..

```

This will create *two* cookies, `cookieName=value1` and `cookieName=value2`.

You can also specify a detailed cookie using:

```

Cookie someCookie = new Cookie.Builder("some_cookie", "some_value").setSecured(true).s
given().cookie(someCookie).when().get("/cookie").then().assertThat().body(equalTo("x")

```

or several detailed cookies at the same time:

```

Cookie cookie1 = Cookie.Builder("username", "John").setComment("comment 1").build();
Cookie cookie2 = Cookie.Builder("token", 1234).setComment("comment 2").build();
Cookies cookies = new Cookies(cookie1, cookie2);
given().cookies(cookies).when().get("/cookie").then().body(equalTo("username, token"))

```

Headers

```
given().header("MyHeader", "Something").and(). ..  
given().headers("MyHeader", "Something", "MyOtherHeader", "SomethingElse").and(). ..
```

You can also specify a multi-value headers like this:

```
given().header("headerName", "value1", "value2"). ..
```

This will create *two* headers, `headerName: value1` and `headerName: value2`.

Header Merging/Overwriting

By default headers are merged. So for example if you do like this:

```
given().header("x", "1").header("x", "2"). ..
```

The request will contain two headers, "x: 1" and "x: 2". You can change in this on a per header basis in the [HeaderConfig](#). For example:

```
given().  
    config(RestAssuredConfig.config().headerConfig(headerConfig().overwriteHeadersl  
    header("x", "1").  
    header("x", "2").  
when().  
    get("/something").  
...  
...  
...
```



This means that only one header, "x: 2", is sent to server.

Content Type

```
given().contentType(ContentType.TEXT). ..  
given().contentType("application/json"). ..
```

Request Body

```
given().body("some body"). .. // Works for POST, PUT and DELETE requests
given().request().body("some body"). .. // More explicit (optional)
```

```
given().body(new byte[]{42}). .. // Works for POST, PUT and DELETE
given().request().body(new byte[]{42}). .. // More explicit (optional)
```

You can also serialize a Java object to JSON or XML. Click [here](#) for details.

Verifying Response Data

You can also verify status code, status line, cookies, headers, content type and body.

Response Body

See Usage examples, e.g. [JSON](#) or [XML](#).

You can also map a response body to a Java Object, click [here](#) for details.

Cookies

```
assertThat().cookie("cookieName", "cookieValue"). ..
assertThat().cookies("cookieName1", "cookieValue1", "cookieName2", "cookieValue2"). ..
assertThat().cookies("cookieName1", "cookieValue1", "cookieName2", containsString("Value2
```

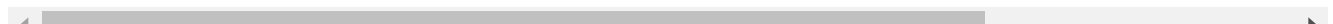


Status

```
get("/x").then().assertThat().statusCode(200). ..
get("/x").then().assertThat().statusLine("something"). ..
get("/x").then().assertThat().statusLine(containsString("some")). ..
```

Headers

```
get("/x").then().assertThat().header("headerName", "headerValue"). ..
get("/x").then().assertThat().headers("headerName1", "headerValue1", "headerName2", "h
get("/x").then().assertThat().headers("headerName1", "headerValue1", "headerName2", co
```



It's also possible to use a mapping function when validating headers. For example let's say you want to validate that the `Content-Length` header is less than 1000. You can then use a mapping function to first convert the header value to an int and then use an `Integer` before validating it with a Hamcrest matcher:

```
get("/something").then().assertThat().header("Content-Length", Integer::parseInt, less
```

Content-Type

```
get("/x").then().assertThat().contentType(ContentType.JSON). ..
```

Full body/content matching

```
get("/x").then().assertThat().body(equalTo("something")). ..
```

Use the response to verify other parts of the response

You can use data from the response to verify another part of the response. For example consider the following JSON document returned from service x:

```
{ "userId" : "some-id", "href" : "http://localhost:8080/some-id" }
```

You may notice that the `href` attribute ends with the value of the `userId` attribute. If we want to verify this we can implement a `io.restassured.matcher.ResponseAwareMatcher` and use it like this:

```
get("/x").then().body("href", new ResponseAwareMatcher<Response>() {  
    public Matcher<?> matcher(Response response) {  
        return equalTo("http://localhost:8080/" + re  
    }  
});
```

If you're using Java 8 you can use a lambda expression instead:


```
get("/x").then().body("href", response -> equalTo("http://localhost:8080/" + response.
```



There are some predefined matchers that you can use defined in the

`io.restassured.matcher.RestAssuredMatchers` (Or

`io.restassured.module.mockmvc.matcher.RestAssuredMockMvcMatchers` if using the spring-mock-mvc module). For example:

```
get("/x").then().body("href", endsWithPath("userId"));
```

`ResponseAwareMatchers` can also be composed, either with another `ResponseAwareMatcher` or with a Hamcrest Matcher. For example:

```
get("/x").then().body("href", and(startsWith("http://localhost:8080/"), endsWithPath("u
```



The `and` method is statically imported from

`io.restassured.matcher.ResponseAwareMatcherComposer`.

Measuring Response Time

As of version 2.8.0 REST Assured has support measuring response time. For example:

```
long timeInMs = get("/lotto").time()
```

or using a specific time unit:

```
long timeInSeconds = get("/lotto").timeIn(SECONDS);
```

where `SECONDS` is just a standard `TimeUnit`. You can also validate it using the validation DSL:

```
when().  
    get("/lotto").  
then().  
    time(lessThan(2000L)); // Milliseconds
```

or

```
when().  
    get("/lotto").  
then().  
    time(lessThan(2L), SECONDS);
```

Please note that response time measurement should be performed when the JVM is hot! (i.e. running a response time measurement when only running a single test will yield erroneous results). Also note that you can only vaguely regard these measurements to correlate with the server request processing time (since the response time will include the HTTP round trip and REST Assured processing time among other things).

Authentication

REST assured also supports several authentication schemes, for example OAuth, digest, certificate, form and preemptive basic authentication. You can either set authentication for each request:

```
given().auth().basic("username", "password"). ..
```

but you can also define authentication for all requests:

```
RestAssured.authentication = basic("username", "password");
```

or you can use a [specification](#).

Basic Authentication

There are two types of basic authentication, preemptive and "challenged basic authentication".

Preemptive Basic Authentication

This will send the basic authentication credential even before the server gives an unauthorized response in certain situations, thus reducing the overhead of making an additional connection. This is typically what you want to use in most situations unless you're testing the servers ability to challenge. Example:

```
given().auth().preemptive().basic("username", "password").when().get("/secured/hello")
```

Challenged Basic Authentication

When using "challenged basic authentication" REST Assured will not supply the credentials unless the server has explicitly asked for it. This means that REST Assured will make an additional request to the server in order to be challenged and then follow up with the same request once more but this time setting the basic credentials in the header.

```
given().auth().basic("username", "password").when().get("/secured/hello").then().statu
```

Digest Authentication

Currently only "challenged digest authentication" is supported. Example:

```
given().auth().digest("username", "password").when().get("/secured"). ..
```

Form Authentication

[Form authentication](#) is very popular on the internet. It's typically associated with a user filling out his credentials (username and password) on a webpage and then pressing a login button of some sort. A very simple HTML page that provide the basis for form authentication may look like this:

```
<html>
  <head>
    <title>Login</title>
  </head>

  <body>
    <form action="j_spring_security_check" method="POST">
      <table>
        <tr><td>User:&nbsp;</td><td><input type='text' name='j_username'></td></tr>
        <tr><td>Password:</td><td><input type='password' name='j_password'></td></tr>
        <tr><td colspan='2'><input name="submit" type="submit"/></td></tr>
      </table>
    </form>
  </body>
</html>
```

I.e. the server expects the user to fill-out the "j_username" and "j_password" input fields and then press "submit" to login. With REST Assured you can test a service protected by form authentication like this:

```
given().
    auth().form("John", "Doe").
when().
    get("/formAuth");
then().
    statusCode(200);
```

While this may work it's not optimal. What happens when form authentication is used like this in REST Assured an additional request have to made to the server in order to retrieve the webpage with the login details. REST Assured will then try to parse this page and look for two input fields (with username and password) as well as the form action URI. This may work or fail depending on the complexity of the webpage. A better option is to supply the these details when setting up the form authentication. In this case one could do:

```
given().
    auth().form("John", "Doe", new FormAuthConfig("/j_spring_security_check", "j_u
when().
    get("/formAuth");
then().
    statusCode(200);
```



This way REST Assured doesn't need to make an additional request and parse the webpage. There's also a predefined FormAuthConfig called `springSecurity` that you can use if you're using the default Spring Security properties:

```
given().
    auth().form("John", "Doe", FormAuthConfig.springSecurity()).
when().
    get("/formAuth");
then().
    statusCode(200);
```

CSRF

Today it's common for the server to supply a [CSRF](#) token with the response in order to avoid these kinds of attacks. REST Assured has support for automatically parsing and supplying the CSRF token to the server. In order for this to work REST Assured *must* make an additional request and parse (parts) of the website.

You can enable CSRF support by doing the following:

```
given().
    auth().form("John", "Doe", formAuthConfig().withAutoDetectionOfCsrf()).
when().
    get("/formAuth");
then().
    statusCode(200);
```

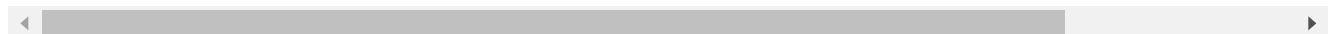
Now REST Assured will automatically try to detect if the webpage contains a CSRF token. In order to assist REST Assured and make the parsing more robust it's possible to supply the CSRF field name (here we imagine that we're using Spring Security default values and thus can make use of the predefined `springSecurity FormAuthConfig`):

```
given().
    auth().form("John", "Doe", springSecurity().withCsrfFieldName("_csrf")).
when().
    get("/formAuth");
then().
    statusCode(200);
```

We've now told REST Assured to search for the CSRF field name called `"_csrf"` (which is it both faster and less prone to error).

By default the CSRF value is sent as a form parameter with the request but you can configure to send it as a header instead if that's required:

```
given().
    auth().form("John", "Doe", springSecurity().withCsrfFieldName("_csrf").sendCsrf()
when().
    get("/formAuth");
then().
    statusCode(200);
```



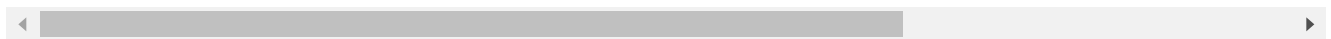
Include additional fields in Form Authentication

Since version 3.1.0 REST Assured can include additional input fields when using form authentication. Just use the `FormAuthConfig` and specify the additional values to include. For example if you have an html page that looks like this:

```
<html>
<head>
  <title>Login</title>
</head>
<body>
<form action="/login" method="POST">
  <table>
    <tr>
      <td>User:&nbsp;</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td colspan="2"><input name="submit" type="submit"/></td>
    </tr>
  </table>
  <input type="hidden" name="firstInputField" value="value1"/>
  <input type="hidden" name="secondInputField" value="value2"/>
</form>
</body>
</html>
```

and you'd like to include the value of form parameters `firstInputField` and `secondInputField` you can do like this:

```
given().auth().form("username", "password", formAuthConfig().withAdditionalFields("fir
```



REST Assured will automatically parse the HTML page, find the values for the additional fields and include them as form parameters in the login request.

OAuth

In order to use OAuth 1 and OAuth 2 (for query parameter signing) you need to add [Scribe](#) to your classpath (if you're using version 2.1.0 or older of REST Assured then please refer to the [legacy](#) documentation). In Maven you can simply add the following dependency:

```
<dependency>
  <groupId>com.github.scribejava</groupId>
  <artifactId>scribejava-apis</artifactId>
  <version>2.5.3</version>
  <scope>test</scope>
</dependency>
```

If you're not using Maven [download](#) a Scribe release manually and put it in your classpath.

OAuth 1

OAuth 1 requires [Scribe](#) in the classpath. To use auth 1 authentication you can do:

```
given().auth().oauth(..). ..
```

OAuth 2

Since version 2.5.0 you can use OAuth 2 authentication without depending on [Scribe](#):

```
given().auth().oauth2(accessToken). ..
```

This will put the OAuth2 `accessToken` in a header. To be more explicit you can also do:

```
given().auth().preemptive().oauth2(accessToken). ..
```

There reason why `given().auth().oauth2(..)` still exists is for backward compatibility (they do the same thing). If you need to provide the OAuth2 token in a query parameter you currently need [Scribe](#) in the classpath. Then you can do like this:

```
given().auth().oauth2(accessToken, OAuthSignature.QUERY_STRING). ..
```

Custom Authentication

Rest Assured allows you to create custom authentication providers. You do this by implementing the `io.restassured.spi.AuthFilter` interface (preferably) and apply it as a [filter](#). For example let's say that your security consists of adding together two headers together in a new header called "AUTH" (this is of course not secure). Then you can do that like this (Java 8 syntax):

```

given().
    filter((requestSpec, responseSpec, ctx) -> {
        String header1 = requestSpec.getHeaders().getValue("header1");
        String header2 = requestSpec.getHeaders().getValue("header2");
        requestSpec.header("AUTH", header1 + header2);
        return ctx.next(requestSpec, responseSpec);
    }).
when().
    get("/customAuth").
then().
    statusCode(200);

```

The reason why you want to use a `AuthFilter` and not `Filter` is that `AuthFilters` are automatically removed when doing `given().auth().none(). ...`

Multi-part form data

When sending larger amount of data to the server it's common to use the multipart form data technique. Rest Assured provide methods called `multiPart` that allows you to specify a file, byte-array, input stream or text to upload. In its simplest form you can upload a file like this:

```

given().
    multiPart(new File("/path/to/file")).
when().
    post("/upload");

```

It will assume a control name called "file". In HTML the control name is the attribute name of the input tag. To clarify let's look at the following HTML form:

```

<form id="uploadForm" action="/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="file" size="40">
    <input type="submit" value="Upload!">
</form>

```

The control name in this case is the name of the input tag with name "file". If you have a different control name then you need to specify it:

```

given().
    multiPart("controlName", new File("/path/to/file")).
when().
    post("/upload");

```


It's also possible to supply multiple "multi-parts" entities in the same request:

```
byte[] someData = ..
given().
    multiPart("controlName1", new File("/path/to/file")).
    multiPart("controlName2", "my_file_name.txt", someData).
    multiPart("controlName3", someJavaObject, "application/json").
when().
    post("/upload");
```

For more advanced use cases you can make use of the [MultiPartSpecBuilder](#). For example:

```
Greeting greeting = new Greeting();
greeting.setFirstName("John");
greeting.setLastName("Doe");

given().
    multiPart(new MultiPartSpecBuilder(greeting, ObjectMapperType.JACKSON_2)
        .fileName("greeting.json")
        .controlName("text")
        .mimeType("application/vnd.custom+json").build()).
when().
    post("/multipart/json").
then().
    statusCode(200);
```

You can specify, among other things, the default `control` name and filename using the [MultiPartConfig](#). For example:

```
given().config(config().multiPartConfig(multiPartConfig().defaultControlName("something-else"));
```



This will configure the default control name to be "something-else" instead of "file".

For additional info refer to [this](#) blog post.

Object Mapping

REST Assured supports mapping Java objects to and from JSON and XML. For JSON you need to have either Jackson, Jackson2, Gson or Johnzon in the classpath and for XML you need JAXB.

Serialization

Let's say we have the following Java object:

```
public class Message {  
    private String message;  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```

and you want to serialize this object to JSON and send it with the request. There are several ways to do this, e.g:

Content-Type based Serialization

```
Message message = new Message();  
message.setMessage("My messagee");  
given().  
    contentType("application/json").  
    body(message).  
when().  
    post("/message");
```

In this example REST Assured will serialize the object to JSON since the request content-type is set to "application/json". It will first try to use Jackson if found in classpath and if not Gson will be used. If you change the content-type to "application/xml" REST Assured will serialize to XML using JAXB. If no content-type is defined REST Assured will try to serialize in the following order:

1. JSON using Jackson 2 (Faster Jackson (databind))
2. JSON using Jackson (databind)
3. JSON using Gson
4. JSON using Johnzon
5. JSON-B using Eclipse Yasson
6. XML using JAXB

REST Assured also respects the charset of the content-type. E.g.

```
Message message = new Message();
message.setMessage("My messagee");
given().
    contentType("application/json; charset=UTF-16").
    body(message).
when().
    post("/message");
```

You can also serialize the `Message` instance as a form parameter:

```
Message message = new Message();
message.setMessage("My messagee");
given().
    contentType("application/json; charset=UTF-16").
    formParam("param1", message).
when().
    post("/message");
```

The message object will be serialized to JSON using Jackson (databind) (if present) or Gson (if present) with UTF-16 encoding.

Create JSON from a HashMap

You can also create a JSON document by supplying a Map to REST Assured.

```
Map<String, Object> jsonAsMap = new HashMap<>();
jsonAsMap.put("firstName", "John");
jsonAsMap.put("lastName", "Doe");

given().
    contentType(JSON).
    body(jsonAsMap).
when().
    post("/somewhere").
then().
    statusCode(200);
```

This will provide a JSON payload as:

```
{ "firstName" : "John", "lastName" : "Doe" }
```

Using an Explicit Serializer

If you have multiple object mappers in the classpath at the same time or don't care about setting the content-type you can specify a serializer explicitly. E.g.

```
Message message = new Message();
message.setMessage("My messagee");
given().
    body(message, ObjectMapperType.JAXB).
when().
    post("/message");
```

In this example the Message object will be serialized to XML using JAXB.

Deserialization

Again let's say we have the following Java object:

```
public class Message {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

and we want the response body to be deserialized into a Message object.

Content-Type based Deserialization

Let's assume then that the server returns a JSON body like this:

```
{"message": "My message"}
```

To deserialize this to a Message object we simply do like this:

```
Message message = get("/message").as(Message.class);
```

For this to work the response content-type must be "application/json" (or something that contains "json"). If the server instead returned

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<message>
  <message>My message</message>
</message>
```

and a content-type of "application/xml" you wouldn't have to change the code at all:

```
Message message = get("/message").as(Message.class);
```

Custom Content-Type Deserialization

If the server returns a custom content-type, let's say "application/something", and you still want to use the object mapping in REST Assured there are a couple of different ways to go about. You can either use the [explicit](#) approach or register a parser for the custom content-type:

```
Message message = expect().parser("application/something", Parser.XML).when().get("/me
```



or

```
Message message = expect().defaultParser(Parser.XML).when().get("/message").as(Message
```



You can also register a default or custom parser [statically](#) or using [specifications](#).

Using an Explicit Deserializer

If you have multiple object mappers in the classpath at the same time or don't care about the response content-type you can specify a deserializer explicitly. E.g.

```
Message message = get("/message").as(Message.class, ObjectMapperType.GSON);
```

Configuration

You can configure the pre-defined object mappers by using a [ObjectMapperConfig](#) and pass it to [detailed configuration](#). For example to change GSON to use lower case with underscores as field naming policy you can do like this:

```
RestAssured.config = RestAssuredConfig.config().objectMapperConfig(objectMapperConfig(
    new GsonObjectMapperFactory() {
        public Gson create(Class cls, String charset) {
            return new GsonBuilder().setFieldNamingPolicy(LOWER_CASE_WITH_UNDERSCORES);
        }
    }
));
```

There are pre-defined object mapper factories for GSON, JAXB, Jackson, Faster Jackson and Eclipse Yasson (JSON-B).

Custom

By default REST Assured will scan the classpath to find various object mappers. If you want to integrate an object mapper that is not supported by default or if you've rolled your own you can implement the [io.restassured.mapper.ObjectMapper](#) interface. You tell REST Assured to use your object mapper either by passing it as a second parameter to the body:

```
given().body(myJavaObject, myObjectMapper).when().post("..")
```

or you can define it statically once and for all:

```
RestAssured.config = RestAssuredConfig.config().objectMapperConfig(new ObjectMapperConfig(myObjectMapper));
```

For an example see [here](#).

Custom parsers

REST Assured provides predefined parsers for e.g. HTML, XML and JSON. But you can parse other kinds of content by registering a predefined parser for unsupported content-types by using:

```
RestAssured.registerParser(<content-type>, <parser>);
```

E.g. to register that mime-type 'application/vnd.uoml+xml' should be parsed using the XML parser do:

```
RestAssured.registerParser("application/vnd.uoml+xml", Parser.XML);
```

You can also unregister a parser using:

```
RestAssured.unregisterParser("application/vnd.uoml+xml");
```

Parsers can also be specified per "request":

```
get(..).then().using().parser("application/vnd.uoml+xml", Parser.XML). ..;
```

and using a [response specification](#).

Default parser

Sometimes it's useful to specify a default parser, e.g. if the response doesn't contain a content-type at all:

```
RestAssured.defaultParser = Parser.JSON;
```

You can also specify a default parser for a single request:

```
get("/x").then().using().defaultParser(Parser.JSON). ..
```

or using a [response specification](#).

Default values

By default REST assured assumes host localhost and port 8080 when doing a request. If you want a different port you can do:

```
given().port(80). ..
```

or simply:

```
..when().get("http://myhost.org:80/doSomething");
```

You can also change the default base URI, base path, port and authentication scheme for all subsequent requests:

```
RestAssured.baseURI = "http://myhost.org";
RestAssured.port = 80;
RestAssured.basePath = "/resource";
RestAssured.authentication = basic("username", "password");
RestAssured.rootPath = "x.y.z";
```

This means that a request like e.g. `get("/hello")` goes to:

<http://myhost.org:80/resource/hello> with basic authentication credentials "username" and "password". See [rootPath](#) for more info about setting the root paths. Other default values you can specify are:

```
RestAssured.filters(..); // List of default filters
RestAssured.requestSpecification = .. // Default request specification
RestAssured.responseSpecification = .. // Default response specification
RestAssured.urlEncodingEnabled = .. // Specify if Rest Assured should URL encoding the
RestAssured.defaultParser = .. // Specify a default parser for response bodies if no n
RestAssured.registerParser(..) // Specify a parser for the given content-type
RestAssured.unregisterParser(..) // Unregister a parser for the given content-type
```

You can reset to the standard baseURI (localhost), basePath (empty), standard port (8080), standard root path (""), default authentication scheme (none) and url encoding enabled (true) using:

```
RestAssured.reset();
```

Specification Re-use

Instead of having to duplicate response expectations and/or request parameters for different tests you can re-use an entire specification. To do this you define a specification using either the [RequestSpecBuilder](#) or [ResponseSpecBuilder](#).

E.g. let's say you want to make sure that the expected status code is 200 and that the size of the JSON array "x.y" has size 2 in several tests you can define a [ResponseSpecBuilder](#) like this:


```

ResponseSpecBuilder builder = new ResponseSpecBuilder();
builder.expectStatusCode(200);
builder.expectBody("x.y.size()", is(2));
ResponseSpecification responseSpec = builder.build();

// Now you can re-use the "responseSpec" in many different tests:
when().
    get("/something").
then().
    spec(responseSpec).
    body("x.y.z", equalTo("something"));

```

In this example the data defined in "responseSpec" is merged with the additional body expectation and all expectations must be fulfilled in order for the test to pass.

You can do the same thing if you need to re-use request data in different tests. E.g.

```

RequestSpecBuilder builder = new RequestSpecBuilder();
builder.addParam("parameter1", "parameterValue");
builder.addHeader("header1", "headerValue");
RequestSpecification requestSpec = builder.build();

given().
    spec(requestSpec).
    param("parameter2", "paramValue").
when().
    get("/something").
then().
    body("x.y.z", equalTo("something"));

```

Here the request's data is merged with the data in the "requestSpec" so the request will contain two parameters ("parameter1" and "parameter2") and one header ("header1").

Querying RequestSpecification

Sometimes it's useful to be able to query/extract values from a RequestSpecification. For this reason you can use the `io.restassured.specification.SpecificationQuerier`. For example:

```

RequestSpecification spec = ...
QueryableRequestSpecification queryable = SpecificationQuerier.query(spec);
String headerValue = queryable.getHeaders().getValue("header");
String param = queryable.getFormParams().get("someparam");

```

Filters

A filter allows you to inspect and alter a request before it's actually committed and also inspect and [alter](#) the response before it's returned to the expectations. You can regard it as an "around advice" in AOP terms. Filters can be used to implement custom authentication schemes, session management, logging etc. To create a filter you need to implement the [io.restassured.filter.Filter](#) interface. To use a filter you can do:

```
given().filter(new MyFilter()). ..
```

There are a couple of filters provided by REST Assured that are ready to use:

1. `io.restassured.filter.log.RequestLoggingFilter` : A filter that'll print the request specification details.
2. `io.restassured.filter.log.ResponseLoggingFilter` : A filter that'll print the response details if the response matches a given status code.
3. `io.restassured.filter.log.ErrorLoggingFilter` : A filter that'll print the response body if an error occurred (status code is between 400 and 500).

Ordered Filters

As of REST Assured 3.0.2 you can implement the [io.restassured.filter.OrderedFilter](#) interface if you need to control the filter ordering. Here you implement the `getOrder` method to return an integer representing the precedence of the filter. A lower value gives higher precedence. The highest precedence you can define is `Integer.MIN_VALUE` and the lowest precedence is `Integer.MAX_VALUE`. Filters not implementing [io.restassured.filter.OrderedFilter](#) will have a default precedence of `1000`. Click [here](#) for some examples.

Response Builder

If you need to change the [Response](#) from a filter you can use the [ResponseBuilder](#) to create a new Response based on the original response. For example if you want to change the body of the original response to something else you can do:

```
Response newResponse = new ResponseBuilder().clone(originalResponse).setBody("Somethin
```



Logging

In many cases it can be useful to print the response and/or request details in order to help you create the correct expectations and send the correct requests. To do help you do this you can use one of the predefined [filters](#) supplied with REST Assured or you can use one of the shortcuts.

Request Logging

Since version 1.5 REST Assured supports logging the [request specification](#) before it's sent to the server using the [RequestLoggingFilter](#). Note that the HTTP Builder and HTTP Client may add additional headers then what's printed in the log. The filter will *only* log details specified in the request specification. I.e. you can NOT regard the details logged by the [RequestLoggingFilter](#) to be what's actually sent to the server. Also subsequent filters may alter the request *after* the logging has taken place. If you need to log what's *actually* sent on the wire refer to the [HTTP Client logging docs](#) or use an external tool such [Wireshark](#).

Examples:

```
given().log().all(). .. // Log all request specification details including parameters,
given().log().params(). .. // Log only the parameters of the request
given().log().body(). .. // Log only the request body
given().log().headers(). .. // Log only the request headers
given().log().cookies(). .. // Log only the request cookies
given().log().method(). .. // Log only the request method
given().log().path(). .. // Log only the request path
```

Response Logging

If you want to print the response body regardless of the status code you can do:

```
get("/x").then().log().body() ..
```

This will print the response body regardless if an error occurred. If you're only interested in printing the response body if an error occur then you can use:

```
get("/x").then().log().ifError(). ..
```

You can also log all details in the response including status line, headers and cookies:

```
get("/x").then().log().all(). ..
```

as well as only status line, headers or cookies:

```
get("/x").then().log().statusLine(). .. // Only log the status line
get("/x").then().log().headers(). .. // Only log the response headers
get("/x").then().log().cookies(). .. // Only log the response cookies
```

You can also configure to log the response only if the status code matches some value:

```
get("/x").then().log().ifStatusCodeIsEqualTo(302). .. // Only log if the status code i
get("/x").then().log().ifStatusCodeMatches(matcher). .. // Only log if the status code
```

Log if validation fails

Since REST Assured 2.3.1 you can log the request or response only if the validation fails. To log the request do:

```
given().log().ifValidationFails(). ..
```

To log the response do:

```
.. .then().log().ifValidationFails(). ..
```

It's also possible to enable this for both the request and the response at the same time using the [LogConfig](#):

```
given().config(RestAssured.config().logConfig(logConfig().enableLoggingOfRequestAndRes
```

This will log only the headers if validation fails.

There's also a shortcut for enabling logging of the request and response for all requests if validation fails:

```
RestAssured.enableLoggingOfRequestAndResponseIfValidationFails();
```

Blacklist Headers from Logging

As of REST Assured 4.2.0 it's possible to blacklist headers so that they are not shown in the request or response log. Instead the header value will be replaced with [BLACKLISTED]. You can enable this per header basis using the [LogConfig](#):

```
given().config(config().logConfig(logConfig().blacklistHeader("Accept"))). ..
```

The response log will then print:

```
Request method:   GET
Request URI:      http://localhost:8080/something
Proxy:           <none>
Request params:  <none>
Query params:    <none>
Form params:     <none>
Path params:     <none>
Headers:         Accept=[ BLACKLISTED ]
Cookies:         <none>
Multiparts:      <none>
Body:            <none>
```

Root path

To avoid duplicated paths in body expectations you can specify a root path. E.g. instead of writing:

```
when().
    get("/something").
then().
    body("x.y.firstName", is(..)).
    body("x.y.lastName", is(..)).
    body("x.y.age", is(..)).
    body("x.y.gender", is(..));
```

you can use a root path and do:

```
when().
    get("/something").
then().
    root("x.y"). // You can also use the "root" method
    body("firstName", is(..)).
    body("lastName", is(..)).
    body("age", is(..)).
    body("gender", is(..));
```

You can also set a default root path using:

```
RestAssured.rootPath = "x.y";
```

In more advanced use cases it may also be useful to append additional root arguments to existing root arguments. To do this you can use the `appendRoot` method, for example:

```
when().  
    get("/jsonStore").  
then().  
    root("store.%s", withArgs("book")).  
    body("category.size()", equalTo(4)).  
    appendRoot("%s.%s", withArgs("author", "size()")).  
    body(withNoArgs(), equalTo(4));
```

It's also possible to detach a root. For example:

```
when().  
    get("/jsonStore").  
then().  
    root("store.category").  
    body("size()", equalTo(4)).  
    detachRoot("category").  
    body("size()", equalTo(1));
```

Path arguments

Path arguments are useful in situations where you have e.g. pre-defined variables that constitutes the path. For example

```
String someSubPath = "else";  
int index = 1;  
get("/x").then().body("something.%s[%d]", withArgs(someSubPath, index), equalTo("some value"));
```

will expect that the body path " something.else[0] " is equal to "some value".

Another usage is if you have complex [root paths](#) and don't wish to duplicate the path for small variations:

```

when().
    get("/x").
then().
    root("filters.filterConfig[%d].filterConfigGroups.find { it.name == 'GroupName'
    body(withArgs(0), hasItem("first")).
    body(withArgs(1), hasItem("second")).
    ..

```

The path arguments follows the standard [formatting syntax](#) of Java.

Note that the `withArgs` method can be statically imported from the [io.restassured.RestAssured](#) class.

Sometimes it's also useful to validate a body without any additional arguments when all arguments have already been specified in the root path. This is where `withNoArgs` come into play. For example:

```

when().
    get("/jsonStore").
then().
    root("store.%s", withArgs("book")).
    body("category.size()", equalTo(4)).
    appendRoot("%s.%s", withArgs("author", "size()")).
    body(withNoArgs(), equalTo(4));

```

Session support

REST Assured provides a simplified way for managing sessions. You can define a session id value in the DSL:

```

given().sessionId("1234"). ..

```

This is actually just a short-cut for:

```

given().cookie("JSESSIONID", "1234"). ..

```

You can also specify a default `sessionId` that'll be supplied with all subsequent requests:

```

RestAssured.sessionId = "1234";

```

By default the session id name is `JSESSIONID` but you can change it using the [SessionConfig](#):

```
RestAssured.config = RestAssured.config().sessionConfig(new SessionConfig().sessionIdName("JSESSIONID"))
```

You can also specify a sessionId using the `RequestSpecBuilder` and reuse it in many tests:

```
RequestSpecBuilder spec = new RequestSpecBuilder().setSessionId("value1").build();

// Make the first request with session id equal to value1
given().spec(spec). ..
// Make the second request with session id equal to value1
given().spec(spec). ..
```

It's also possible to get the session id from the response object:

```
String sessionId = get("/something").sessionId();
```

Session Filter

As of version 2.0.0 you can use a [session filter](#) to automatically capture and apply the session, for example:

```
SessionFilter sessionFilter = new SessionFilter();

given().
    auth().form("John", "Doe").
    filter(sessionFilter).
when().
    get("/formAuth").
then().
    statusCode(200);

given().
    filter(sessionFilter). // Reuse the same session filter instance to automatically
when().
    get("/x").
then().
    statusCode(200);
```

To get session id caught by the `SessionFilter` you can do like this:


```
String sessionId = sessionFilter.getSessionId();
```

SSL

In most situations SSL should just work out of the box thanks to the excellent work of HTTP Builder and HTTP Client. There are however some cases where you'll run into trouble. You may for example run into a `SSLPeerUnverifiedException` if the server is using an invalid certificate. The easiest way to workaround this is to use "relaxed HTTPs validation". For example:

```
given().relaxedHTTPSValidation().when().get("https://some_server.com"). ..
```

You can also define this statically for all requests:

```
RestAssured.useRelaxedHTTPSValidation();
```

or in a [request specification](#).

This will assume an `SSLContext` protocol of `SSL`. To change to another protocol use an overloaded version of `relaxedHTTPSValidation`. For example:

```
given().relaxedHTTPSValidation("TLS").when().get("https://some_server.com"). ..
```

You can also be more fine-grained and create Java keystore file and use it with REST Assured. It's not too difficult, first follow the guide [here](#) and then use the keystore in Rest Assured like this:

```
given().keystore("/pathToJksInClassPath", <password>). ..
```

or you can specify it for every request:

```
RestAssured.keystore("/pathToJksInClassPath", <password>);
```

You can also define a keystore in a re-usable [specification](#).

If you already loaded a keystore with a password you can use it as a truststore:

```
RestAssured.trustStore(keystore);
```

You can find a working example [here](#).

For more advanced SSL Configuration refer to the [SSL Configuration](#) section.

SSL invalid hostname

If the certificate is specifying an invalid hostname you don't need to create and import a keystore. As of version 2.2.0 you can do:

```
RestAssured.config = RestAssured.config().sslConfig(sslConfig().allowAllHostnames());
```



to allow all hostnames for all requests or:

```
given().config(RestAssured.config().sslConfig(sslConfig().allowAllHostnames())). .. ;
```

for a single request.

Note that if you use "relaxed HTTPs validation" then `allowAllHostnames` is activated by default.

URL Encoding

Usually you don't have to think about URL encoding since Rest Assured provides this automatically out of the box. In some cases though it may be useful to turn URL Encoding off. One reason may be that you already have some parameters encoded before you supply them to Rest Assured. To prevent double URL encoding you need to tell Rest Assured to disable its URL encoding. E.g.

```
String response = given().urlEncodingEnabled(false).get("https://jira.atlassian.com:44  
..
```



or

```
RestAssured.baseURI = "https://jira.atlassian.com";  
RestAssured.port = 443;
```

```
RestAssured.urlEncodingEnabled = false;
final String query = "project%20=%20BAM%20AND%20issuetype%20=%20Bug";
String response = get("/rest/api/2.0.alpha1/search?jql={q}", query);
..
```

Proxy Configuration

Starting from version 2.3.2 REST Assured has better support for proxies. For example if you have a proxy at localhost port 8888 you can do:

```
given().proxy("localhost", 8888). ..
```

Actually you don't even have to specify the hostname if the server is running on your local environment:

```
given().proxy(8888). .. // Will assume localhost
```

To use HTTPS you need to supply a third parameter (scheme) or use the `io.restassured.specification.ProxySpecification`. For example:

```
given().proxy(host("localhost").withScheme("https")). ..
```

where `host` is statically imported from `io.restassured.specification.ProxySpecification`.

Starting from version 2.7.0 you can also specify preemptive basic authentication for proxies. For example:

```
given().proxy(auth("username", "password")).when() ..
```

where `auth` is statically imported from `io.restassured.specification.ProxySpecification`. You can of course also combine authentication with a different host:

```
given().proxy(host("http://myhost.org").withAuth("username", "password")). ..
```

Static Proxy Configuration

It's also possible to configure a proxy statically for all requests, for example:

```
RestAssured.proxy("localhost", 8888);
```

or:

```
RestAssured.proxy = host("localhost").withPort(8888);
```

Request Specification Proxy Configuration

You can also create a request specification and specify the proxy there:

```
RequestSpecification specification = new RequestSpecBuilder().setProxy("localhost").build()
given().spec(specification) ..
```

Detailed configuration

Detailed configuration is provided by the [RestAssuredConfig](#) instance with which you can configure the parameters of [HTTP Client](#) as well as [Redirect](#), [Log](#), [Encoder](#), [Decoder](#), [Session](#), [ObjectMapper](#), [Connection](#), [SSL](#) and [ParamConfig](#) settings. Examples:

For a specific request:

```
given().config(RestAssured.config().redirect(redirectConfig().followRedirects(false)))
```

or using a RequestSpecBuilder:

```
RequestSpecification spec = new RequestSpecBuilder().setConfig(RestAssured.config().re
```

or for all requests:

```
RestAssured.config = config().redirect(redirectConfig().followRedirects(true)).and().ma
```

`config()` and `newConfig()` can be statically imported from `io.restassured.config.RestAssuredConfig`.

Encoder Config

With the [EncoderConfig](#) you can specify the default content encoding charset (if it's not specified in the content-type header) and query parameter charset for all requests. If no content charset is specified then ISO-8859-1 is used and if no query parameter charset is specified then UTF-8 is used. Usage example:

```
RestAssured.config = RestAssured.config().encoderConfig(encoderConfig().defaultContent
```

You can also specify which encoder charset to use for a specific content-type if no charset is defined explicitly for this content-type by using the `defaultCharsetForContentType` method in the [EncoderConfig](#). For example:

```
RestAssured.config = RestAssured.config(config().encoderConfig(encoderConfig().default
```

This will assume UTF-16 encoding for "application/xml" content-types that does explicitly specify a charset. By default "application/json" is specified to use "UTF-8" as default content-type as this is specified by [RFC4627](#).

Avoid adding the charset to content-type header automatically

By default REST Assured adds the charset header automatically. To disable this completely you can configure the `EncoderConfig` like this:

```
RestAssured.config = RestAssured.config(config().encoderConfig(encoderConfig().appendD
```

Decoder Config

With the [DecoderConfig](#) you can set the default response content decoding charset for all responses. This is useful if you expect a different content charset than ISO-8859-1 (which is the default charset) and the response doesn't define the charset in the content-type header. Usage example:

```
RestAssured.config = RestAssured.config().decoderConfig(decoderConfig().defaultContent
```

You can also use the `DecoderConfig` to specify which content decoders to apply. When you do this the `Accept-Encoding` header will be added automatically to the request and the response body will be decoded automatically. By default GZIP and DEFLATE decoders are enabled. To for example to remove GZIP decoding but retain DEFLATE decoding you can do the following:

```
given().config(RestAssured.config().decoderConfig(decoderConfig().contentDecoders(DEFL
```



You can also specify which decoder charset to use for a specific content-type if no charset is defined explicitly for this content-type by using the "defaultCharsetForContentType" method in the [DecoderConfig](#). For example:

```
RestAssured.config = config(config().decoderConfig(decoderConfig().defaultCharsetForCo
```



This will assume UTF-16 encoding for "application/xml" content-types that does explicitly specify a charset. By default "application/json" is using "UTF-8" as default charset as this is specified by [RFC4627](#).

Session Config

With the session config you can configure the default session id name that's used by REST Assured. The default session id name is `JSESSIONID` and you only need to change it if the name in your application is different and you want to make use of REST Assured's [session support](#). Usage:

```
RestAssured.config = RestAssured.config().sessionConfig(new SessionConfig().sessionIdN
```



Redirect DSL

Redirect configuration can also be specified using the DSL. E.g.

```
given().redirects().max(12).and().redirects().follow(true).when(). ..
```

Connection Config

Lets you configure connection settings for REST Assured. For example if you want to force-close the Apache HTTP Client connection after each response. You may want to do this if you make a lot of fast consecutive requests with small amount of data in the response. However if you're downloading (especially large amounts of) chunked data you must not close connections after each response. By default connections are *not* closed after each response.

```
RestAssured.config = RestAssured.config().connectionConfig(connectionConfig().closeIdleConnections(true));
```

Json Config

[JsonPathConfig](#) allows you to configure the Json settings either when used by REST Assured or by [JsonPath](#). It let's you configure how JSON numbers should be treated.

```
RestAssured.config = RestAssured.config().jsonConfig(jsonConfig().numberReturnType(Number.class));
```

HTTP Client Config

Let's you configure properties for the HTTP Client instance that REST Assured will be using when executing requests. By default REST Assured creates a new instance of http client for each "given" statement. To configure reuse do the following:

```
RestAssured.config = RestAssured.config().httpClient(httpClientConfig().reuseHttpClient(true));
```

You can also supply a custom HTTP Client instance by using the `httpClientFactory` method, for example:

```
RestAssured.config = RestAssured.config().httpClient(httpClientConfig().httpClientFactory(new HttpClientConfig.HttpClientFactory() {  
    @Override  
    public HttpClient createHttpClient() {  
        return new SystemDefaultHttpClient();  
    }  
}));
```

Note that currently you need to supply an instance of `AbstractHttpClient`.

It's also possible to configure default parameters etc.

SSL Config

The [SSLConfig](#) allows you to specify more advanced SSL configuration such as truststore, keystore type and host name verifier. For example:


```
RestAssured.config = RestAssured.config().sslConfig(sslConfig().with().keystoreType(<type>);
```



Param Config


[ParamConfig](#) allows you to configure how different parameter types should be updated on "collision". By default all parameters are merged so if you do:

```
given().queryParam("param1", "value1").queryParam("param1", "value2").when().get("/x")
```



REST Assured will send a query string of `param1=value1¶m1=value2`. This is not always what you want though so you can configure REST Assured to *replace* values instead:

```
given().  
    config(config().paramConfig(paramConfig().queryParamsUpdateStrategy(REPLACE)))  
    queryParam("param1", "value1").  
    queryParam("param1", "value2").  
when().  
    get("/x"). ..
```



REST Assured will now replace `param1` with `value2` (since it's written last) instead of merging them together. You can also configure the update strategy for each type of for all parameter types instead of doing it per individual basis:

```
given().config(config().paramConfig(paramConfig().replaceAllParameters())). ..
```

This is also supported in the [Spring Mock Mvc Module](#) (but the config there is called [MockMvcParamConfig](#)).

Failure Config

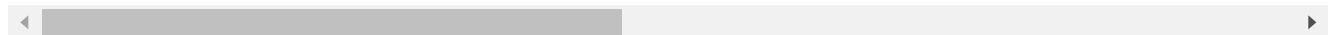
Added in version 3.3.0 the [FailureConfig](#) can be used to get callbacks when REST Assured validation fails. This is useful if you want to do some custom logging or store data available in the request/response specification or in the response itself somewhere. For example let's say that you want to be notified by email when the following test case fails because the status code is not 200:

```
given().
    param("x", "y")
when().
    get("/hello")
then().
    statusCode(200);
```

You can then implement a [ResponseValidationFailureListener](#) and add it to the [FailureConfig](#):

```
ResponseValidationFailureListener emailOnFailure = (reqSpec, respSpec, resp) -> emailS

given().
    config(RestAssured.config().failureConfig(failureConfig().with().failureListeners(
        param("x", "y")
when().
    get("/hello")
then().
    statusCode(200);
```



Spring Support

REST Assured contains two support modules for testing Spring Controllers using the REST Assured API:

- [spring-mock-mvc](#) - For unit testing standard Spring [MVC](#) Controllers
- [spring-web-test-client](#) - For unit testing (reactive) Spring [Webflux](#) Controllers

Spring Mock Mvc Module

REST Assured 2.2.0 introduced support for [Spring Mock Mvc](#) using the `spring-mock-mvc` module. This means that you can unit test Spring Mvc Controllers. For example given the following Spring controller:

```

@Controller
public class GreetingController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping(value = "/greeting", method = GET)
    public @ResponseBody Greeting greeting(
        @RequestParam(value="name", required=false, defaultValue="World") String name) {
        return new Greeting(counter.incrementAndGet(), String.format(template, name));
    }
}

```

you can test it using [RestAssuredMockMvc](#) like this:

```

given().
    standaloneSetup(new GreetingController()).
    param("name", "Johan").
when().
    get("/greeting").
then().
    statusCode(200).
    body("id", equalTo(1)).
    body("content", equalTo("Hello, Johan!"));

```

i.e. it's very similar to the standard REST Assured syntax. This makes it really fast to run your tests and it's also easier to bootstrap the environment and use mocks (if needed) than standard REST Assured. Most things that you're used to in standard REST Assured works with RestAssured Mock Mvc as well. For example (certain) configuration, static specifications, logging etc etc. To use it you need to depend on the Spring Mock Mvc module:

```

<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>spring-mock-mvc</artifactId>
    <version>4.3.0</version>
    <scope>test</scope>
</dependency>

```

Or [download](#) it from the download page if you're not using Maven.

Bootstrapping RestAssuredMockMvc

First of all you should statically import methods in:

```
io.restassured.module.mockmvc.RestAssuredMockMvc.*
io.restassured.module.mockmvc.matcher.RestAssuredMockMvcMatchers.*
```

instead of those defined in

```
io.restassured.RestAssured.*
io.restassured.matcher.RestAssuredMatchers.*
```

Refer to [static import](#) section of the documentation for additional static imports.

In order to start a test using RestAssuredMockMvc you need to initialize it with either a set of Controllers, a MockMvc instance or a WebApplicationContext from Spring. You can do this for a single request as seen in the previous example:

```
given().standaloneSetup(new GreetingController()). ..
```

or you can do it statically:

```
RestAssuredMockMvc.standaloneSetup(new GreetingController());
```

If defined statically you don't have to specify any Controllers (or MockMvc or WebApplicationContext instance) in the DSL. This means that the previous example can be written as:

```
given().
    param("name", "Johan").
when().
    get("/greeting").
then().
    statusCode(200).
    body("id", equalTo(1)).
    body("content", equalTo("Hello, Johan!"));
```

Asynchronous Requests

Both RestAssuredMockMvc and As of version 2.5.0 RestAssuredMockMvc has support for asynchronous requests. For example let's say you have the following controller:

```
@Controller
public class PostAsyncController {
```

```

@RequestMapping(value = "/stringBody", method = POST)
public @ResponseBody
Callable<String> stringBody(final @RequestBody String body) {
    return new Callable<String>() {
        public String call() throws Exception {
            return body;
        }
    };
}
}

```

You can test this like so:

```

given().
    body("a string").
when().
    async().post("/stringBody").
then().
    body(equalTo("a string"));

```

This will use the default timeout of 1 second. You can change the timeout by using the DSL:

```

given().
    body("a string").
when().
    async().with().timeout(20, TimeUnit.SECONDS).post("/stringBody").
then().
    body(equalTo("a string"));

```

It's also possible to configure a default timeout by using the [AsyncConfig](#), for example:

```

given().
    config(config().asyncConfig(withTimeout(100, TimeUnit.MILLISECONDS))).
    body("a string").
when().
    async().post("/stringBody").
then().
    body(equalTo("a string"));

```

`withTimeout` is statically imported from `io.restassured.module.mockmvc.config.AsyncConfig` and is just a shortcut for creating an `AsyncConfig` with a given timeout. Apply the config globally to apply to all requests:

```
RestAssuredMockMvc.config = RestAssuredMockMvc.config().asyncConfig(withTimeout(100, T

// Request 1
given().
    body("a string").
when().
    async().post("/stringBody").
then().
    body(equalTo("a string"));

// Request 2
given().
    body("another string").
when().
    async().post("/stringBody").
then().
    body(equalTo("a string"));
```

Both request 1 and 2 will now use the default timeout of 100 milliseconds.

Adding Request Post Processors

Spring MockMvc has support for [Request Post Processors](#) and you can use these in RestAssuredMockMvc as well. For example:

```
given().postProcessors(myPostProcessor1, myPostProcessor2). ..
```

Note that it's recommended to add `RequestPostProcessors` from `org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors` (i.e. `authentication RequestPostProcessors`) to `auth` instead for better readability (result will be the same):

```
given().auth().with(httpBasic("username", "password")). ..
```

where `httpBasic` is statically imported from [SecurityMockMvcRequestPostProcessor](#).

Adding Result Handlers

Spring MockMvc has support for [Result Handlers](#) and you can use these in RestAssuredMockMvc as well. For example let's say you want to use the native MockMvc logging:

```
.. .then().apply(print()). ..
```

where `print` is statically imported from

`org.springframework.test.web.servlet.result.MockMvcResultHandlers`. Note that if you're using REST Assured 2.6.0 or older you used the `resultHandlers` method:

```
given().resultHandlers(print()). ..
```

but this was deprecated in REST Assured 2.8.0.

Using Result Matchers

Spring MockMvc provides a bunch of [Result Matchers](#) that you may find useful.

RestAssuredMockMvc has support for these as well if needed. For example let's say that for some reason you want to verify that the status code is equal to 200 using a `ResultMatcher`:

```
given().
    param("name", "Johan").
when().
    get("/greeting").
then().
    assertThat(status().isOk()).
    body("id", equalTo(1)).
    body("content", equalTo("Hello, Johan!"));
```

where `status` is statically imported from

`org.springframework.test.web.servlet.result.MockMvcResultMatchers`. Note that you can also use the `expect` method which is the same as `assertThat` but more close to the syntax of native MockMvc.

Interceptors

For more advanced use cases you can also get ahold of and modify the [MockHttpServletRequestBuilder](#) before the request is performed. To do this define a [MockHttpServletRequestBuilderInterceptor](#) and use it with RestAssuredMockMvc:

```
given().interceptor(myInterceptor). ..
```

Spring Mock Mvc Specifications

Just as with standard Rest Assured you can use [specifications](#) to allow for better re-use. Note that the request specification builder for RestAssuredMockMvc is called [MockMvcRequestSpecBuilder](#). The same [ResponseSpecBuilder](#) can be used in RestAssuredMockMvc as well though. Specifications can be defined statically as well just as with standard Rest Assured. For example:

```
RestAssuredMockMvc.requestSpecification = new MockMvcRequestSpecBuilder().addQueryParam(
RestAssuredMockMvc.responseSpecification = new ResponseSpecBuilder().expectStatusCode(

given().
    standaloneSetup(new GreetingController()).
when().
    get("/greeting").
then().
    body("id", equalTo(1));
```

Resetting RestAssuredMockMvc

If you've used any static configuration you can easily reset RestAssuredMockMvc to its default state by calling the `RestAssuredMockMvc.reset()` method.

Spring MVC Authentication

Version 2.3.0 of `spring-mock-mvc` supports authentication. For example:

```
given().auth().principal(..). ..
```

Some authentication methods require Spring Security to be on the classpath (optional). It's also possible to define authentication statically:

```
RestAssuredMockMvc.authentication = principal("username", "password");
```

where the `principal` method is statically imported from [RestAssuredMockMvc](#). It's also possible to define an authentication scheme in a request builder:

```
MockMvcRequestSpecification spec = new MockMvcRequestSpecBuilder.setAuth(principal("us
```

Using Spring Security Test

Since version 2.5.0 there's also better support for Spring Security. If you have `spring-security-test` in classpath you can do for example:

```
given().auth().with(httpBasic("username", "password")). ..
```

where `httpBasic` is statically imported from [SecurityMockMvcRequestPostProcessor](#). This will apply basic authentication to the request. For this to work you need apply the [SecurityMockMvcConfigurer](#) to the `MockMvc` instance. You can either do this manually:

```
MockMvc mvc = MockMvcBuilders.webApplicationContextSetup(context).apply(SecurityMockMvcConfigu
```

or `REStAssuredMockMvc` will automatically try to apply the `springSecurity` configurer automatically if you initialize it with an instance of [AbstractMockMvcBuilder](#), for example when configuring a "web app context":

```
given().webApplicationContextSetup(context).auth().with(httpBasic("username", "password")). ..
```

Here's a full example:

```
import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.web.context.WebApplicationContext;

import static io.restassured.module.mockmvc.RestAssuredMockMvc.given;
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcReq
import static org.springframework.security.test.web.servlet.response.SecurityMockMvcRe

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = MyConfiguration.class)
@WebAppConfiguration
public class BasicAuthExample {

    @Autowired
    private WebApplicationContext context;
```



```

@Before public void
rest_assured_is_initialized_with_the_web_application_context_before_each_test() {
    RestAssuredMockMvc.webAppContextSetup(context);
}

@After public void
rest_assured_is_reset_after_each_test() {
    RestAssuredMockMvc.reset();
}

@Test public void
basic_auth_example() {
    given().
        auth().with(httpBasic("username", "password")).
    when().
        get("/secured/x").
    then().
        statusCode(200).
        expect(authenticated().withUsername("username"));
}
}

```

You can also define authentication for all request, for example:

```
RestAssuredMockMvc.authentication = with(httpBasic("username", "password"));
```

where `with` is statically imported from `io.restassured.module.mockmvc.RestAssuredMockMvc`. It's also possible to use a [request specification](#).

Injecting a User

It's also possible use to of Spring Security test annotations such as [@WithMockUser](#) and [@WithUserDetails](#). For example let's say you want to test this controller:

```

@Controller
public class UserAwareController {

    @RequestMapping(value = "/user-aware", method = GET)
    public
    @ResponseBody
    String userAware(@AuthenticationPrincipal User user) {
        if (user == null || !user.getUsername().equals("authorized_user")) {
            throw new IllegalArgumentException("Not authorized");
        }

        return "Success";
    }
}

```

```
}  
}
```

As you can see the `userAware` method takes a [User](#) as argument and we let Spring Security inject it by using the [@AuthenticationPrincipal](#) annotation. To generate a test user we could do like this:

```
@WithMockUser(username = "authorized_user")  
@Test public void  
spring_security_mock_annotations_example() {  
    given().  
        webApplicationContextSetup(context).  
    when().  
        get("/user-aware").  
    then().  
        statusCode(200).  
        body(equalTo("Success")).  
        expect(authenticated().withUsername("authorized_user"));  
}
```

Spring Web Test Client Module

REST Assured 3.2.0 introduced support for testing components of the [Spring Reactive Web](#) stack using the `spring-web-test-client` module. This means that you can unit test reactive Spring (Webflux) Controllers. For example let's say that the server defines a controller that returns JSON using this DTO:

```
public class Greeting {  
  
    private final long id;  
    private final String content;  
  
    public Greeting(long id, String content) {  
        this.id = id;  
        this.content = content;  
    }  
  
    public long getId() {  
        return id;  
    }  
  
    public String getContent() {  
        return content;  
    }  
}
```

The reactive Controller might look like this:

```
@RestController
public class GreetingController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @GetMapping(value = "/greeting", produces = "application/json")
    public Mono<Greeting> greeting(@RequestParam(value="name") String name) {
        return Mono.just(new Greeting(counter.incrementAndGet(), String.format(template, name)));
    }
}
```

you can test it using [RestAssuredWebTestClient](#) like this:

```
package io.restassured.module.webtestclient;

import io.restassured.module.webtestclient.setup.GreetingController;
import org.junit.Test;
import static io.restassured.module.webtestclient.RestAssuredWebTestClient.given;

public class GreetingControllerTest {

    @Test
    public void greeting_controller_returns_json_greeting() {
        given().
            standaloneSetup(new GreetingController()).
            param("name", "Johan").
        when().
            get("/greeting").
        then().
            statusCode(200).
            body("id", equalTo(1)).
            body("content", equalTo("Hello, Johan!"));
    }
}
```

i.e. it's very similar to the standard REST Assured syntax. This makes it really fast to run your tests and it's also easier to bootstrap the environment and use mocks (if needed) than standard REST Assured. Most things that you're used to in standard REST Assured works with `RestAssuredWebTestClient` as well. For example (certain) configuration, static specifications, logging etc etc. To use it you need to depend on the `spring-web-test-client` module:

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>spring-web-test-client</artifactId>
  <version>4.3.0</version>
  <scope>test</scope>
</dependency>
```

Or [download](#) it from the download page if you're not using Maven.

Bootstrapping RestAssuredWebTestClient

First of all you should statically import methods in:

```
io.restassured.module.webtestclient.RestAssuredWebTestClient.*
io.restassured.module.webtestclient.matcher.RestAssuredWebTestClientMatchers.*
```

instead of those defined in

```
io.restassured.RestAssured.*
io.restassured.matcher.RestAssuredMatchers.*
```

Refer to [static import](#) section of the documentation for additional static imports.

In order to start a test using RestAssuredWebTestClient you need to initialize it with either a set of Controllers, a WebTestClient instance or a WebApplicationContext from Spring. You can do this for a single request as seen in the previous example:

```
given().standaloneSetup(new GreetingController()). ..
```

or you can do it statically:

```
RestAssuredWebTestClient.standaloneSetup(new GreetingController());
```

If defined statically you don't have to specify any Controllers in the DSL. This means that the previous example can be written as:

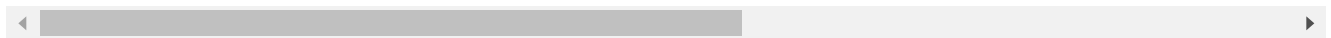
```
given().
  param("name", "Johan").
when().
  get("/greeting").
then().
```

```
statusCode(200).  
body("id", equalTo(1)).  
body("content", equalTo("Hello, Johan!"));
```

Spring Web Test Client Specifications

Just as with standard Rest Assured you can use [specifications](#) to allow for better re-use. Note that the request specification builder for RestAssuredWebTestClient is called [WebTestClientRequestSpecBuilder](#). The same [ResponseSpecBuilder](#) can be used in RestAssuredWebTestClient as well though. Specifications can be defined statically as well just as with standard Rest Assured. For example:

```
RestAssuredWebTestClient.requestSpecification = new WebTestClientRequestSpecBuilder().  
RestAssuredWebTestClient.responseSpecification = new ResponseSpecBuilder().expectStatus  
  
given().  
    standaloneSetup(new GreetingController()).  
when().  
    get("/greeting").  
then().  
    body("id", equalTo(1));
```



Resetting RestAssuredWebTestClient

If you've used any static configuration you can easily reset RestAssuredWebTestClient to its default state by calling the `RestAssuredWebTestClient.reset()` method.

Common Spring Module Documentation

Note on parameters

Neither RestAssuredMockMvc nor RestAssuredWebTestClient differentiates between parameters types, so `param`, `formParam` and `queryParams` currently just delegates to `param` in MockMvc. `formParam` adds the `application/x-www-form-urlencoded` content-type header automatically though just as standard Rest Assured does.

Scala Support Module

REST Assured 2.6.0 introduced the [scala-support](#) module that adds an alias to the "then" method defined in the [Response](#) or [MockMvcResponse](#) called "Then". The reason for this is that `then` might be a reserved keyword in Scala in the future and the compiler issues a warning when using a method with this name. To enable the use of `Then` simply import the `io.restassured.module.scala.RestAssuredSupport.AddThenToResponse` class from the `scala-support` module. For example:

```
import io.restassured.RestAssured.when
import io.restassured.module.scala.RestAssuredSupport.AddThenToResponse
import org.hamcrest.Matchers.equalTo
import org.junit.Test

@Test
def `trying out rest assured in scala with implicit conversion`() {
    when().
        get("/greetJSON").
    Then().
        statusCode(200).
        body("key", equalTo("value"))
}
```

Note that this is also supported for the [Spring Mock Mvc Module](#).

To use it do like this:

SBT:

```
libraryDependencies += "io.rest-assured" % "scala-support" % "4.3.0"
```

Maven:

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>scala-support</artifactId>
  <version>4.3.0</version>
  <scope>test</scope>
</dependency>
```

Gradle:

```
testImplementation 'io.rest-assured:scala-support:4.3.0'
```

No build manager:

Download the [distribution file](#) manually.

Kotlin

Avoid Escaping "when" Keyword

Kotlin is a language developed by [JetBrains](#) and it integrates very well with Java and REST Assured. When using it with REST Assured there's one thing that can be a bit annoying. That is you have to escape `when` since it's a reserved keyword in Kotlin. You can do this either by using [Kotlin Extension Module](#) (recommended) or you can simply create your own extension method (the approach shown below). For example:

```
@Test
fun `kotlin rest assured example`() {
    given().
        param("firstName", "Johan").
        param("lastName", "Haleby").
    `when`().
        get("/greeting").
    then().
        statusCode(200).
        body("greeting.firstName", equalTo("Johan")).
        body("greeting.lastName", equalTo("Haleby"))
}
```

To get around this, create an [extension function](#) that creates an alias to `when` called `When` :

```
fun RequestSpecification.When(): RequestSpecification {
    return this.`when`()
}
```

The code can now be written like this:

```
@Test
fun `kotlin rest assured example`() {
    given().
        param("firstName", "Johan").
        param("lastName", "Haleby").
    When().
        get("/greeting").
}
```

```

    then().
        statusCode(200).
        body("greeting.firstName", equalTo("Johan")).
        body("greeting.lastName", equalTo("Haleby"))
}

```

Notice that we don't need any escaping anymore. For more details refer to [this](#) blog post.

Kotlin Extension Module

REST Assured 4.1.0 introduced a new module called "kotlin-extensions". This module provides some useful extension functions when working with REST Assured from Kotlin. First you need to add the module to the project:

```

<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>kotlin-extensions</artifactId>
    <version>4.3.0</version>
    <scope>test</scope>
</dependency>

```

and then import `Given` from the `io.restassured.module.kotlin.extensions` package. You can then use it like this:

```

val message: String =
    Given {
        port(7000)
        header("Header", "Header")
        body("hello")
    } When {
        put("/the/path")
    } Then {
        statusCode(200)
        body("message", equalTo("Another World"))
    } Extract {
        path("message")
    }

```

Besides a more pleasing API for Kotlin developers it also has a couple of major benefits to the Java API:

1. All failed expectations are reported at the same time
2. Formatting the code in your IDE won't mess up indentation

Note that the names of the extension functions are subject to change in the future (although it's probably not likely). You can read more about the rationale and benefits of the Kotlin API in [this](#) blog post.

Kotlin Extension Module for Spring MockMvc

REST Assured 4.1.0 introduced Kotlin extension support for the [Spring MockMvc](#) module. This allows one to write tests like this:

```
class RestAssuredMockMvcKotlinExtensionsTest {

    @Test
    fun example() {
        val mockMvc =
            MockMvcBuilders.standaloneSetup(GreetingController())
                .build()

        val id: Int =
            Given {
                mockMvc(mockMvc)
                param("name", "Johan")
            } When {
                get("/greeting")
            } Then {
                body(
                    "id", Matchers.equalTo(1),
                    "content", Matchers.equalTo("Hello, Johan!")
                )
            } Extract {
                path("id")
            }

        assertThat(id).isEqualTo(1)
    }
}
```

To use it depend on:

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>spring-mock-mvc-kotlin-extensions</artifactId>
  <version>4.3.0</version>
  <scope>test</scope>
</dependency>
```

and import the extension functions from the
`io.restassured.module.mockmvc.kotlin.extensions` package.

More info

For more information refer to the [javadoc](#):

- [RestAssured](#)
- [RestAssuredMockMvc Javadoc](#)
- [Specification package](#)

You can also have a look at some code examples:

- REST Assured [tests](#)
- [JsonPathTest](#)
- [XmlPathTest](#)

If you need support then join the [mailing list](#).

For professional support please contact [johanhaleby](#).

► Pages 33

- [Getting Started](#)
- [Downloads](#)
- [Usage Guide \(Legacy\)](#)
- [Snapshot dependencies](#)
- [Release Notes](#)
- [FAQ](#)
- [Support](#)

Clone this wiki locally

`https://github.com/rest-assured/rest-assured.wiki.git`

