Ersatz Server User Guide

v3.0.0 (January 2021)

Table of Contents

Introduction

The Ersatz Server is an HTTP client testing tool, which allows for request and response expectations to be configured in a flexible manner. The expectations will respond to requests in a configured manner allowing tests with different responses and/or error conditions without having to write a lot of boiler-plate code.

The "mock" server is not really a mock at all, it is an embedded Undertow HTTP server which registers the configured expectations as routes and then responds according to the configured expectation behavior. This approach may seem overly heavy; however, testing an HTTP client can involve a lot of internal state and interactions that the developer is generally unaware of (and should be) - trying to mock those interactions with a pure mocking framework will get out of hand very quickly, and Undertow starts up very quickly.

Ersatz provides a balance of mock-like expectation behavior with a real HTTP interface and all of the underlying interactions in place. This allows for rich unit testing, which is what you were trying to do in the first place.

Ersatz is written in Java 15 due to its use of the modern functional libraries; however, there is an extension library (ersatz-groovy) which provides a Groovy DSL and extensions to the base library.

Lastly, Ersatz is developed with testing in mind. It does not favor any specific testing framework, but it does work well with both the JUnit and Spock frameworks.

What's New In 3.0

- Not directly backward compatible with the 2.x codebase (see migration notes below).
- Requires Java 15+
- Removed support for legacy JUnit (< 5).
- Added new option, `ServerConfig::logResponseContent()`, to enable response content rendering (disabled by default).
- Removed Web Sockets support – this may be re-implemented later with a more stable API if there is interest.
- Removed the `ErsatzProxy` component – this may be re-implemented later if there is interest.
- Removed built-in support for JSON encoding/decoding to remove an external dependency. See the Encoding and Decoding sections for example source for implementing your own.
- Extracted the Groovy API into a separate library (`ersatz-groovy`) so that the main library could be implemented in Java without Groovy dependencies.
- Added more predefined `ContentType` constants.
- Added more encoders and decoders for common scenarios.
- Updated the dependencies and fixed some exposed dependency isolation issues.
- The User Guide is now presented as a PDF.

Migrating to 3.0

- If you use the proxy or web sockets testing support, there is no upgrade path. Please create an Issue or start a Discussion to show that you are interested in one or both of these features.
- If you use Groovy for development, you will need to change your dependency references from using the `ersatz` library to use the new `ersatz-groovy` library. The same change applies if you are using the `safe` version of the library.
- If you are using the legacy JUnit support helper, you will either need to implement the support class yourself (optionally using the source from the 2.x codebase), or you can submit an Issue or start a Discussion to show that you are interested in one or both of these features.
- If you are using the built-in JSON encoder or decoder, you will need to replace them with your own implementation (documentation and sample code are provided in the Decoders and Encoders sections).

What's New in 2.0

- Not directly backward compatible with 1.x codebase (see migration notes below).
- Requires Java 11+.

- Refactored code and packaging as well as code-conversion from Groovy to Java (no loss of support for using with Groovy).
- Removed deprecated methods.
- Refactored the HTTP method names to be uppercase.
- Added optional timeout for standard request verify calls.
- Converted the underlying response content from String to byte[] (also changed response encoder API)
- Refactored the underlying server into a more abstract integration so that it may be swapped out in the future.
- Pulled external Closure helper API code into codebase (to avoid breaking maven-central support)
- Refactored the JUnit support (4 and 5)

Migrating to 2.0

- Change all HTTP method names to uppercase (e.g. if you have a `head('/foo')` call, change it to `HEAD('/foo')`).
- Replace any deprecated method usages with appropriate replacements.
- If you use the `ErsatzProxy` or JUnit helper classes, you will need to change the package information.
- Most of the DSL classes were repackaged and will require updates to the package names imported.

What's New in 1.9

- Corrections to the closure variable scoping.
- Support for configuring the server port - though, in general, this is not recommended.
- Some added usage documentation

What's New in 1.8

- Variable scope changes – the configuration Groovy DSL closures had incorrect (or inadequate) resolution strategies specified which caused variables to be resolved incorrectly in some situations. All of the closures now use `DELEGATE_FIRST`; however, beware this may cause some issues with existing code.
- Deprecation of the `Response::content(...)` methods in favor of the new `body(...)` methods.

- ANSI color codes were added to the match failure reports to make them a bit more readable.
- A couple of helper methods were added to `ErsatzServer` to facilitate simple URL string building – see `httpUrl(String)` and `httpsUrl(String)`.
- A JUnit 5 `Extension` was added to make server management simple with JUnit 5, similar to what already existed for JUnit 4.
- Support for "chunked" responses with fixed or random delays between chunks has been added.

Getting Started

The `ersatz` and `ersatz-groovy` libraries are available via Bintray (JCenter) and the Maven Central Repository; you can add them to your project using one of the following methods:

For Gradle add the following to your `build.gradle` file:

```
testCompile 'com.stehno.ersatz:ersatz:3.0.0'
```

For Maven, add the code below to your `pom.xml` file dependencies section:

```
<dependency>
    <groupId>com.stehno.ersatz</groupId>
    <artifactId>ersatz</artifactId>
    <version>3.0.0</version>
    <scope>test</scope>
</dependency>
```

If you are working with Groovy, you can use the `ersatz-groovy` artifact instead (it pulls in the core `ersatz` library as a dependency).

Once you have the library configured as a dependency, you could use it in a JUnit 5 test as follows (Java):

```
package com.stehno.ersatz.examples;

import com.stehno.ersatz.ErsatzServer;
import com.stehno.ersatz.junit.ErsatzServerExtension;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

import static com.stehno.ersatz.cfg.ContentType.TEXT_PLAIN;
import static java.net.http.HttpClient.newHttpClient;
import static java.net.http.HttpResponse.BodyHandlers.ofString;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

@ExtendWith(ErsatzServerExtension.class)
```

```
class HelloTest {

    private ErsatzServer server;

    @Test void sayHello() throws Exception {
        server.expectations(expect -> {
            expect.GET("/say/hello", req -> {
                req.called(1);
                req.query("name", "Ersatz");
                req.responder(res -> {
                    res.body("Hello, Ersatz", TEXT_PLAIN);
                });
            });
        });

        final var request = HttpRequest
            .newBuilder(new URI(server.httpUrl("/say/hello?name=Ersatz")))
            .GET()
            .build();

        final var response = newHttpClient().send(request, ofString());

        assertEquals(200, response.statusCode());
        assertEquals("Hello, Ersatz", response.body());
        assertTrue(server.verify());
    }
}
```

The server is configured to expect a single GET /say/hello request with name=Ersatz on the query string. When it receives that request, the server will respond with status code 200 (by default), and a response with content-type “text/plain” and “Hello, Ersatz” as the body content. If the server does not receive the expected request, the `verify()` call will fail, likewise, the expected response content would not be returned.

A similar test could be written in Groovy, also using JUnit 5, as follows:

```
package com.stehno.ersatz.examples

import com.stehno.ersatz.GroovyErsatzServer
import com.stehno.ersatz.junit.ErsatzServerExtension
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.extension.ExtendWith

import java.net.http.HttpRequest

import static com.stehno.ersatz.cfg.ContentType.TEXT_PLAIN
import static java.net.http.HttpClient.newHttpClient
import static java.net.http.HttpResponse.BodyHandlers.ofString
import static org.junit.jupiter.api.Assertions.assertEquals
import static org.junit.jupiter.api.Assertions.assertTrue

@ExtendWith(ErsatzServerExtension)
class HelloGroovyTest {

    private GroovyErsatzServer server

    @Test void 'say hello'() {
        server.expectations {
            GET('/say/hello') {
                called 1
                query 'name', 'Ersatz'
```

```
                responder {
                    body 'Hello, Ersatz', TEXT_PLAIN
                }
            }
        }

        final var request = HttpRequest
                .newBuilder(new URI(server.httpUrl('/say/hello?name=Ersatz')))
                .GET()
                .build()

        final var response = newHttpClient().send(request, ofString())

        assertEquals 200, response.statusCode()
        assertEquals 'Hello, Ersatz', response.body()
        assertTrue server.verify()
    }
}
```

Note that the configuration is almost identical between the two, though with Groovy it's just a bit cleaner. Also, note that for the Groovy version the `GroovyErsatzServer` is used instead of the `ErsatzServer` – this provides additional Groovy DSL support.

Ersatz Server Lifecycle

The core component of the Ersatz Server framework is the `ErsatzServer` class. It is used to manage the server lifecycle as well as providing the configuration interface.

The lifecycle of the server can be broken down into four states: Configuration, Matching, Verification, and Cleanup. Each is detailed in the following sections.

Configuration

The first lifecycle state is "configuration", where the server is instantiated, request expectations are configured and the server is started.

An Ersatz server is created as an instance of either `ErsatzServer` or `GroovyErsatzServer` with optional configuration performed by providing a `Consumer<ServerConfig>` or a `Closure` respectively. Both will have an instance of `ServerConfig` passed into them for the configuration to be applied.

Global decoders and encoders may also be configured with the server, as such they will be used as defaults across all configured expectations.

At this point, there is no HTTP server running and it is ready for further configuration, as well specifying the request expectations (using the `expectations(…)` and `expects()` methods).

Once the request expectations are configured, if auto-start is enabled (the default), the server will automatically start. If auto-start is disabled (using `autoStart(false)`), the server will need to be started using the `start()` method. If the server is not started, you will receive connection errors during testing.

The server is now ready for “matching”.

Matching

The second state of the server, is “matching”, where request/response interactions are made against the server.

Any HTTP client can be used to make requests against an Ersatz server. The `ErsatzServer` instance has some helpful methods for use by the client, in order to get the URLs, ports as exposed by the server.

Verification

Once the testing has been performed, it may be desirable to verify whether or not the expected requests were matched the expected number of times (using the `Request::called(…)` methods) rather than just that they were called at all.

To execute verification, one the `ErsatzServer::verify(…)` must be called, which will return a boolean value of true if the verification passed.

Verification is optional and may simply be skipped if you have no need for counting the requests.

Cleanup

After matching and verification, when all test interactions have completed, the server must be stopped in order to free up resources and close connections. This is done by calling the `ErsatzServer::stop()` method or its alias `close()`. This is an important step, as odd test failures have been noticed during multi-test runs if the server is not properly stopped.

If you use JUnit 5, you can use the `ErsatzServerExtension` to perform server instantiation and cleanup for you.

If you use Spock, you can use the `@AutoCleanup` annotation on the ErsatzServer to perform the cleanup automatically.

Note: A stopped server may be restarted, though if you want to clean out expectations, you may want to call the `ErsatzServer::clearExpectations()` method before starting it again.

Server Configuration