Cucumber - Reusing steps

Reusing **common steps across scenarios** is a best practice in test automation frameworks (especially in **Rest Assured**, **Cucumber**, or other BDD frameworks). It promotes **code reusability**, **maintainability**, and **readability**. Below is a complete beginner-friendly explanation with examples using **Cucumber + Rest Assured (Java)** as the context.

🧩 1. Why Reuse Steps?

Suppose you have multiple scenarios that start with similar steps like:

```gherkin
Given the API is up
And I set the base URI
And I set headers
```

Writing their step definitions repeatedly would violate DRY (Don't Repeat Yourself). Instead, we define **common step definitions once** and reuse them across scenarios.

🛠️ 2. Folder Structure Example

```sql
project/
│
├── features/
│   └── apiTests.feature
│
├── stepdefinitions/
│   ├── CommonSteps.java       ✅ common setup steps
│   └── GetUserSteps.java      ✅ specific to Get User API
│
└── utils/
    └── ApiUtils.java          ✅ reusable logic
```

📄 3. Sample Feature File (apiTests.feature)

```gherkin
Feature: User API Tests

  Scenario: Get user by ID
    Given the API is up
    And I set the base URI
    And I set headers
    When I send GET request to "/users/1"
    Then the response status code should be 200

  Scenario: Get all users
    Given the API is up
    And I set the base URI
    And I set headers
    When I send GET request to "/users"
    Then the response status code should be 200
```

👣 4. Common Step Definitions (CommonSteps.java)

```java
package stepdefinitions;

import io.cucumber.java.en.Given;
import io.restassured.RestAssured;
import io.restassured.specification.RequestSpecification;

public class CommonSteps {
    public static RequestSpecification request;

    @Given("the API is up")
    public void api_is_up() {
        System.out.println("API server is assumed up");
    }

    @Given("I set the base URI")
    public void set_base_uri() {
        RestAssured.baseURI = "https://jsonplaceholder.typicode.com";
        request = RestAssured.given();
    }

    @Given("I set headers")
    public void set_headers() {
        request.header("Content-Type", "application/json");
    }
}
```

🚀 5. Scenario-Specific Steps (GetUserSteps.java)

```java
package stepdefinitions;

import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import io.restassured.response.Response;

import static org.junit.Assert.assertEquals;

public class GetUserSteps {

    Response response;

    @When("I send GET request to {string}")
    public void send_get_request(String endpoint) {
        response = CommonSteps.request.get(endpoint);
    }

    @Then("the response status code should be {int}")
    public void validate_status_code(int expectedStatusCode) {
        assertEquals(expectedStatusCode, response.getStatusCode());
    }
}
```

✅ 6. Benefits of Reusing Steps

Benefit	Description
🧹 Cleaner Code	Step definitions are not duplicated
🛠️ Maintainable	Easy to update common logic in one place
🔁 Reusable	Steps can be reused across different test files
📚 Modular	Promotes separation of concerns

🔄 7. Tips

- Keep common steps in a separate class (e.g., `CommonSteps.java`)
- Use static variables/methods for shared data (like `RequestSpecification`)
- Prefix reusable steps clearly (e.g., "I set headers", "I set base URI")
- Avoid using duplicate regex patterns in different classes

Would you like an example using **pure Java (without Cucumber)** or in another language like **Python (Behave)** or **JavaScript (Cypress or Playwright)**?

Keeping your steps **modular and readable** is essential for building a maintainable and scalable automation framework. Whether you're using **Rest Assured with Cucumber** or any other BDD/test framework, modularity and readability go hand in hand.

✅ What Does "Modular and Readable" Mean?

- **Modular** → Each step should do one thing only, and be reusable in other scenarios.
- **Readable** → Anyone (even non-technical team members) can understand the scenario steps.

🔧 How to Keep Steps Modular and Readable

1. ✅ Use Descriptive Step Names in Gherkin

Bad:

```gherkin
Given I call the API
```

Good:

```gherkin
Given the API is up
And I set the base URI to "https://jsonplaceholder.typicode.com"
And I set the request headers
```

Each step clearly states what it's doing — no ambiguity.

---

2. ✅ Keep Step Definitions Focused (One Job Per Method)

Bad:

```java
@Given("I initialize the request")
public void initRequest() {
    RestAssured.baseURI = "https://jsonplaceholder.typicode.com";
    request = RestAssured.given().header("Content-Type", "application/json");
}
```

Good:

```java
@Given("I set the base URI to {string}")
public void setBaseUri(String uri) {
    RestAssured.baseURI = uri;
    request = RestAssured.given();
}

@Given("I set the request headers")
```

```
public void setHeaders() {
    request.header("Content-Type", "application/json");
}
```

➡️ This separation makes each method reusable and easier to test/debug.

---

3. ✅ Reuse Utility Functions Behind the Scenes

Extract the actual implementation logic into utility classes:

ApiUtils.java

```java
public class ApiUtils {
    public static RequestSpecification createRequestWithHeaders() {
        return RestAssured.given()
                          .header("Content-Type", "application/json");
    }
}
```

Then in your step:

```java
@Given("I set the request headers")
public void setHeaders() {
    request = ApiUtils.createRequestWithHeaders();
}
```

➡️ Keeps steps readable, logic clean.

---

4. ✅ Use Variables to Avoid Hardcoding

Make use of scenario-level variables to store dynamic content:

```java
public class CommonSteps {
    public static RequestSpecification request;
    public static Response response;
}
```

This allows multiple step classes to **share the same request and response objects**.

---

5. ✅ Group Steps Logically

- **CommonSteps.java** → for base URI, headers, auth
- **GetUserSteps.java** → for `/users` endpoints
- **PostUserSteps.java** → for POST logic
- **ValidationSteps.java** → for all assertions

---

6. ✅ Write Gherkin Scenarios Like Test Cases

Bad:

```gherkin
When I hit endpoint "/users"
Then I check status
```

Good:

```gherkin
When I send a GET request to "/users"
Then the response status code should be 200
And the response should contain user data
```

✅ Summary

Best Practice	Why It Matters
🧱 One Responsibility per Step	Easier to reuse and maintain
✍️ Clear and Descriptive Steps	Improves readability and collaboration
🛠️ Utility Functions for Logic	Keeps step definitions clean
📂 Logical Class Grouping	Organizes code better
🔄 Parameterized Steps	Makes steps dynamic and reusable

Would you like a **mini framework structure template** or a **ready-made step definitions boilerplate** for practice?

Great! Here's a **ready-made modular and readable Cucumber + Rest Assured framework template** that you can directly use or customize. This is designed for beginners aiming to keep step definitions clean, organized, and reusable.

✅ Mini Framework Structure (Java + Cucumber + Rest Assured)

```bash
api-automation-framework/
│
├── src/
│   └── test/
│       ├── java/
│       │   ├── stepdefinitions/
│       │   │   ├── CommonSteps.java
│       │   │   ├── GetUserSteps.java
│       │   │   └── ValidationSteps.java
│       │   └── utils/
│       │       └── ApiUtils.java
│       └── resources/
│           └── features/
│               └── UserAPI.feature
│
├── pom.xml
└── README.md
```

🧪 Example Feature File — `UserAPI.feature`

```gherkin
Feature: User API

  Scenario: Get user by ID
    Given the API is up
    And I set the base URI to "https://jsonplaceholder.typicode.com"
    And I set request headers
    When I send a GET request to "/users/1"
    Then the response status code should be 200
    And the response should contain the key "username"
```

👣 Step Definitions

🔹 `CommonSteps.java`

```java
package stepdefinitions;

import io.cucumber.java.en.Given;
import io.restassured.RestAssured;
import io.restassured.specification.RequestSpecification;

public class CommonSteps {
    public static RequestSpecification request;

    @Given("the API is up")
    public void api_is_up() {
        System.out.println("API is assumed up.");
    }

    @Given("I set the base URI to {string}")
    public void set_base_uri(String uri) {
        RestAssured.baseURI = uri;
        request = RestAssured.given();
    }

    @Given("I set request headers")
    public void set_headers() {
        request.header("Content-Type", "application/json");
    }
}
```

🔹 `GetUserSteps.java`

```java
package stepdefinitions;

import io.cucumber.java.en.When;
import io.restassured.response.Response;

public class GetUserSteps {
    Response response;

    @When("I send a GET request to {string}")
    public void send_get_request(String endpoint) {
        response = CommonSteps.request.get(endpoint);
        CommonSteps.request = CommonSteps.request; // keep the chain intact
        CommonSteps.request.then().log().all(); // optional logging
        CommonSteps.request.response = response;
    }
}
```

🔹 `ValidationSteps.java`

```java
package stepdefinitions;

import io.cucumber.java.en.Then;
import io.restassured.response.Response;
import static org.junit.Assert.assertEquals;
import static org.hamcrest.Matchers.*;
import static io.restassured.RestAssured.*;

public class ValidationSteps {
    Response response = CommonSteps.request.response;

    @Then("the response status code should be {int}")
    public void validate_status_code(int expectedStatusCode) {
        assertEquals(expectedStatusCode, response.getStatusCode());
    }

    @Then("the response should contain the key {string}")
    public void response_should_contain_key(String key) {
        response.then().body("$", hasKey(key));
    }
}
```

🧰 Utility Class — `ApiUtils.java` (Optional Enhancements)

```java
package utils;

import io.restassured.RestAssured;
import io.restassured.specification.RequestSpecification;

public class ApiUtils {
    public static RequestSpecification getBaseRequest(String uri) {
        RestAssured.baseURI = uri;
        return RestAssured.given()
                          .header("Content-Type", "application/json");
    }
}
```

🧾 `pom.xml` Dependencies

```xml
<dependencies>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>7.11.0</version>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>7.11.0</version>
    </dependency>
    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId>
        <version>5.3.0</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

✅ How to Run It

1. Install Maven and Java (8+).
2. Run from terminal:

```bash
mvn test
```

✅ Summary: Key Best Practices Followed Here

Best Practice	✅ Implemented
Common reusable setup steps	✅

Best Practice	✅ Implemented
Clear and descriptive Gherkin	✅
Utility class for request setup	✅
Modular step definitions	✅
Dynamic URI and endpoint support	✅

Would you like me to package this into a downloadable ZIP, or would you prefer a GitHub-ready structure with test cases for POST, PUT, and DELETE too?

Here's a **clean, modular folder structure** for a **Cucumber + Rest Assured API Test Framework** with **easy reuse and scalability in mind** — ideal for beginners and professionals alike.

✅ Common Folder Structure for API Automation with Reusability

```vbnet
api-automation-framework/
│
├── src/
│   └── test/
│       ├── java/
│       │   ├── stepdefinitions/        ✅ Gherkin step definitions
│       │   │   ├── CommonSteps.java
│       │   │   ├── UserSteps.java
│       │   │   ├── PostSteps.java
│       │   │   └── ValidationSteps.java
│       │   │
│       │   ├── runners/                ✅ Cucumber Test Runner
│       │   │   └── TestRunner.java
│       │   │
│       │   ├── utils/                  ✅ Common reusable utilities
│       │   │   ├── ApiUtils.java
│       │   │   ├── ConfigReader.java
│       │   │   └── PayloadManager.java
│       │   │
│       │   └── context/                ✅ Test context & shared data
│       │       └── TestContext.java
│       │
│       └── resources/
│           └── features/              ✅ Feature files (Gherkin)
│               ├── UserAPI.feature
│               ├── PostAPI.feature
│               └── HealthCheck.feature
│
├── target/                            ✅ Test output (generated)
├── logs/                              ✅ Log files (optional)
├── reports/                           ✅ Cucumber or custom reports
│
├── pom.xml                            ✅ Maven dependencies
└── README.md                          ✅ Project instructions
```

🗂️ Folder/Package Responsibilities

Folder/Package	Responsibility
`stepdefinitions/`	Gherkin step logic. One class per domain (e.g., **`UserSteps`**, **`PostSteps`**)
`runners/`	Cucumber JUnit runner config
`utils/`	Common methods: base URI setup, payloads, headers, config loading
`context/`	Share request/response/context across steps
`resources/features/`	**`.feature`** files for each module (GET, POST, auth, etc.)
`logs/`	Logging output using Log4j/SLF4J (optional)
`reports/`	HTML or JSON test reports (optional with plugins)

🧱 Example: `TestContext.java` (for reusability)

```java
package context;

import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;

public class TestContext {
    private RequestSpecification request;
    private Response response;

    public RequestSpecification getRequest() {
        return request;
    }

    public void setRequest(RequestSpecification request) {
        this.request = request;
    }

    public Response getResponse() {
        return response;
    }

    public void setResponse(Response response) {
        this.response = response;
    }
}
```

Use this context class in all step classes for shared request/response objects.

🧰 Example: `ApiUtils.java`

```java
package utils;

import io.restassured.specification.RequestSpecification;
import static io.restassured.RestAssured.*;

public class ApiUtils {

    public static RequestSpecification getRequestSpec() {
        return given().header("Content-Type", "application/json");
    }
}
```

📁 Optional Extensions

- **`config/` folder**: for external property files (e.g., `config.properties`)
- **`enums/` package**: for managing constants or endpoint enums
- **`hooks/` package**: for `@Before` and `@After` test logic (e.g., setup/teardown)
- **`data/` folder**: for test data JSONs or Excel files

✅ Summary Checklist

Good Practice	Implemented in Structure
Modular step classes	✅ Yes
Centralized context handling	✅ Yes
Reusable request utils	✅ Yes
Environment configs (extensible)	✅ Yes
Easily expandable	✅ Yes

Would you like a **GitHub-ready starter project** with this structure and 2–3 working test scenarios (GET/POST) to start with?