* Creating a Cucumber BDD Rest API Test Automation Framework
  + Prerequisites:
    - Java Development Kit (JDK) installed
    - Integrated Development Environment (IDE) like IntelliJ IDEA or Eclipse
    - Maven for project management and build automation
    - Basic knowledge of Java programming

## Step 1: Create a New Maven Project

* Begin by creating a new Maven project in your IntelliJ IDE.
  + Make sure the following Maven package structure is created. If not already created, create them
    - ***src/main/java (Java Package)***
    - ***src/main/resources (resources directory)***
    - ***src/test/java(Java Package) (All the test code goes here)***
    - ***src/test/resources (resources directory) (All the resources needed for the resources like properties files, test data files etc go here)***
* Add the necessary dependencies to the `pom.xml` file. The essential dependencies for a Cucumber BDD Rest API framework are:
  + BDD Tool => Cucumber
    - A text on a white background

      AI-generated content may be incorrect.
  + TDD Tool => TestNG
    - A close-up of a computer code

      AI-generated content may be incorrect.
  + REST Api Testing Tool => REST Assured
    - A group of black text

      AI-generated content may be incorrect.
  + Serialization/Deserialization tool => Jackson
    - RestAssured needs this dependency to be added which it internally uses
    - A close-up of a computer code

      AI-generated content may be incorrect.
  + Dependency Injection
    - These two dependencies are needed for dependency injection to work
    - A screenshot of a computer code

      AI-generated content may be incorrect.

## Step 2: Create feature files

* + Create a ***“features”*** directory in ***src/test/resources***
  + Create one feature file per endpoint

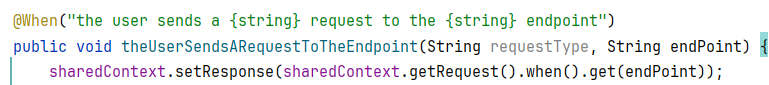
## Step 3: Create step definition classes

* + Make sure to create multiple step definition classes and logically organize the step definitions
  + Shares steps between different features should go into SharedSteps.java
  + Any steps which are only useful for a specific feature file, should be put in a dedicated step definition class
  + Implement Dependency Injection:
    - In the process of logically organising step definitions into multiple files, we end up ***spreading step definitions of a single feature file into multiple step definition classes***
    - When the steps corresponding to a single scenario are spread into multiple step definition classes, then we should take care of sharing the state between the classes. This sharing state between the classes is achieved using dependency injection
    - There are multiple dependency injection tools available:
      * Cucumber Pico Container
      * Google guice
    - In this example we use Google’s guice library for dependency injection
    - Implementation:
      * Step 1: Adding Maven dependencies
        + We need the following two dependencies
        + A screenshot of a computer code

          AI-generated content may be incorrect.
      * Step 2: Set Up “***SharedContext***” class
        + Create a SharedContext.java
        + Add all those variables which must be shared between the steps. In the following example we want to share request and response objects, so we added them to “***SharedContext***” class
        + Create getters and setters for all the variables
        + Annotate the class as @ScenarioScoped
        + A screenshot of a computer program

          AI-generated content may be incorrect.
      * Step 3: Inject the “***SharedContext***” class
        + IN every step definition class add a SharedContext variable
        + Annotate this variable with @Inject
        + A close up of a text

          AI-generated content may be incorrect.
      * Step 4: Using the shared variables in the step definition classes
        + Use the shared variables from the injected SharedContext class
        + A screen shot of a computer code

          AI-generated content may be incorrect.
        + 

## Step 4: Set up Configuration properties:

* + What are properties files?
    - All the application configuration like URLs, paths, timeouts, and other configurable values in code is stored in properties files
    - Here is an example properties file
    - A computer screen shot of a computer code

      AI-generated content may be incorrect.
    - Managing settings like paths, timeouts, and other configurable values in code can make the application difficult to maintain. Using a properties file simplifies changes because you can update settings without touching the application’s logic.
    - Different environments (development, testing, production) require different settings. A properties file makes it easy to maintain and switch between different configurations for each environment (e.g., dev.properties, prod.properties) without changing the application code itself
    - If there is a change in the value of any property, all we need to do is to update the value in properties file and restart the application to start using the new value. If there are no properties files, then we have to make the changes in code, build, test and redeploy the code, which is a time taking process
  + Implementing properties files in test framework:
    - Step 1: Creating properties file
      * Create “***config.properties”*** file under “***src/test/resources”***
      * Add the required properties to this file
      * Make sure there is no space before and after “=”
    - Step 2: Create a class to load properties:
      * A screenshot of a computer

        AI-generated content may be incorrect.
    - Step 3: Fetching the properties within the application
      * Wherever the property is needed in the application, fetch the property value using the property name:
        + 
        + 

## Step 5: Create Test Runner class:

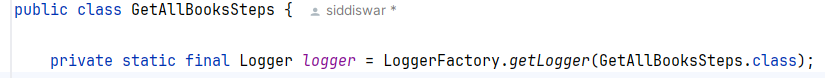
* + Create a runner class
  + A computer screen shot of a computer code

    AI-generated content may be incorrect.

## Step 6: Implement logging:

* + Why do we need logs:
    - **Debugging & Diagnosing Failures**: Logs help understand why tests fail and provide detailed insights for faster troubleshooting.
    - **Reproducibility**: Logs ensure that test failures can be reproduced and investigated in a consistent environment.
    - **Test Coverage & Quality**: Logs help identify gaps in test coverage and provide insights into system behaviour during tests.
    - **Performance Monitoring**: Logs track test performance and help identify slow tests or bottlenecks.
    - **CI & Automation**: Logs integrate with automated systems to provide real-time feedback and trigger alerts when tests fail.
    - **Audit Trails**: Logs provide a historical record and traceability of test executions, which is valuable for audits and compliance.
    - **Collaboration**: Logs facilitate communication among teams and enable a shared understanding of test results.
  + Logs Implementation:
    - Step 1: Adding dependencies
      * Add these two dependencies
      * A screenshot of a computer code

        AI-generated content may be incorrect.
    - Step 2: Creating the log configuration file
      * Create the logback.xml configuration file under src/test/resources
      * A screenshot of a computer program

        AI-generated content may be incorrect.
    - Step 3: creating log file
      * Under the project root directory, create “logs” directory
      * Under the “logs” directory, create “test-execution.log” file. We can give any name to this file
    - Step 4: Write logs in the class
      * Create a logger object in the class
        + 
      * Write the log using logger object
        + 