<https://github.com/cucumber/cucumber-js/blob/main/docs/support_files/world.md>

**Hooks:**

BeforeAll(), AfterAll(): Before and after running ALL scenarios.

Before(), After(): before and after each scenarios

BeforeStep(), AfterStep(): before and after each steps in a scenario.

**Tagged Hooks:** you can use ‘tags’ inside the hook functions to let the hooks function know that they should run only for the tagged scenarios.

**World**, is an isolated scope for each scenario, exposed to the steps and most hooks as this. It allows you to set variables in one step and recall them in a later step. All variables set this way are discarded when the scenario concludes. It is managed by a world class, either the default one or one you create. Each scenario is given an new instance of the class when the test starts, even if it is a [retry run](https://github.com/cucumber/cucumber-js/blob/main/docs/retry.md).

The world is not available to the hooks BeforeAll or AfterAll as each of these executes outside any particular scenario.

**Custom World**

To do this, you can write your own world implementation with its own properties and methods that help with your instrumentation, and then call setWorldConstructor to tell Cucumber about it:

Let's walk through a typical scenario, setting up world that manages a browser context. We'll use the ES6 module syntax for this example. First, let's set up our custom world. Class files should not be loaded as steps - they should be imported. So in this example we'll presume it is in a classes folder next to the steps folder.

// CustomWorld.js

import { World } from '@cucumber/cucumber';

import seleniumWebdriver from "selenium-webdriver";

/\*

\* The only method to be inherited from the default world is

\* the constructor, so if you want to handle the options in

\* an entirely customized manner you don't have to extend from

\* World as seen here.

\*/

export default class extends World {

driver = null;

/\*

\* A constructor is only needed if you have custom actions

\* to take after the Cucumber parses the options or you

\* want to override how the options are parsed.

\*

\* The options are an object with three members

\* {

\* log: Cucumber log function,

\* attach: Cucumber attachment function,

\* params: World Parameters object

\* }

\*/

constructor(options) {

/\*

\* If you don't call the super method you will need

\* to bind the options here as you see fit.

\*/

super(options);

// Custom actions go here.

}

/\*

\* Constructors cannot be asynchronous! To work around this we'll

\* use an init method with the Before hook

\*/

async init(scenario) {

this.driver = await seleniumWebdriver.Builder()

.forBrowser(this.params.browser)

.build();

}

}

Why constructors cant be asynchronous?

In JavaScript, a constructor is a special method that is called when an object is created from a class. It is used to initialize the object's properties and set up its state. Constructors cannot be asynchronous because **they are called synchronously during the object creation process**.

This means that if a constructor contains any asynchronous code, such as **await** statements, it will cause the object creation to block until the asynchronous code completes, which can lead to unpredictable behavior.

To work around this limitation, the **init** method is used as a workaround. The **init** method is called asynchronously using a Cucumber **Before** hook, which runs before each scenario. This allows the **init** method to contain any asynchronous code needed to set up the object state without blocking the object creation process.

In the code you provided, the **init** method is used to create an instance of the Selenium Webdriver for the current scenario being executed. Since creating a new instance of the driver for each scenario is a common practice, using the **init** method with the **Before** hook provides an efficient and reliable way to achieve this.

We have the need to do this because constructor in JavaScript is called synchronously and the playwright uses asynchronous calls for the test steps. By writing asynchronous code, we make sure that the tests don’t get flacky.

If we had moved all the code that is inside the **init** method and put inside the **constructor**, we may have a problem where javascript will attempt to create a **page** before opening a **browser** or create a **context** before creating a **browser**. So, we need this code inside an async function but then we can’t put the async function inside the constructor because javascript does not allow a constructor to be asynchronous. So, we had to define the async init method to put the asynchronous code inside it.

Now we'll use a step file to setup this custom world and declare the before hook.

// setup.js

import { Before, setWorldConstructor } from '@cucumber/cucumber';

import CustomWorld from "../classes/CustomWorld.js"

setWorldConstructor(CustomWorld);

Before(async function(scenario) {

this.init(scenario);

});

Custom world classes can also be used as a point to anchor methods used by multiple steps. In a small project it is sufficient to put these methods in the world class file.

import { Given } from '@cucumber/cucumber';

Given("I'm viewing the admin settings", async function(){

this.login("administrator");

this.page.navigateTo("/admin/settings");

});

This pattern allows for cleaner feature files. Remember that, ideally, scenarios should be between 3-5 lines and communicate **what** the user is doing clearly to the whole team without going into the details of **how** it will be done. While steps can be reused that should not come at the expense of feature clarity.

Explanation:

This code is using Cucumber, a testing framework, to set up a custom world and declare a before hook. The purpose of a custom world class is to provide a set of reusable functions that can be used across multiple steps in a test scenario. In this case, the **CustomWorld** class is defined in a separate file (../classes/CustomWorld.js) and is imported into this file (setup.js) using ES6 module syntax.

The **setWorldConstructor** function is used to set the **CustomWorld** class as the world constructor for Cucumber. This means that when Cucumber runs the test scenarios, it will create a new instance of **CustomWorld** for each scenario.

The **Before** function is used to define a hook that runs before each scenario. In this case, the hook calls the **init** method of the **CustomWorld** instance, passing in the current scenario as an argument.

The second code block shows an example of how a step definition can be written using the **Given** function. In this example, the step is defined as "I'm viewing the admin settings". When this step is executed, the **login** and **navigateTo** methods of the **CustomWorld** instance are called, passing in the appropriate arguments. These methods are defined in the **CustomWorld** class and provide a way to abstract away the implementation details of the test, making it easier to read and maintain.

The final comment emphasizes the importance of writing clear and concise scenarios that communicate what the user is doing, without getting bogged down in implementation details. Steps can be reused across scenarios, but this should not be done at the expense of clarity.

The **Before** hook function in the above code expects the **scenario** parameter to be passed in as an argument. If you remove this parameter from the function definition, the code would still compile and execute, but you would not have access to the **scenario** object in the hook function.

The **scenario** object provides information about the current scenario being executed, such as its name, tags, and feature file location. If you need to access this information in your hook function or any of your step definitions, you must include **scenario** as a parameter in the function definition.

If you do not need to use the **scenario** object in your hook function, you can remove the parameter without any issues. However, it's generally a good idea to leave it in place, even if you're not using it, to make it clear that the hook function expects the **scenario** object to be available.