BDD with SpecFlow Training Workbook

Automation with SpecFlow (API, .NET Core)

Gáspár Nagy

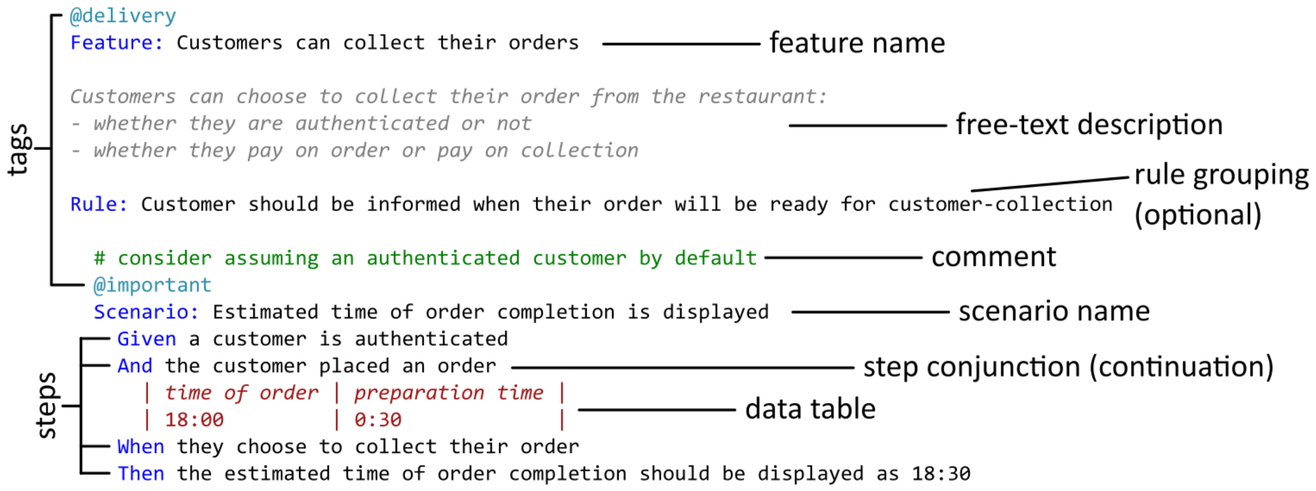
coach, trainer and BDD addict,
creator of SpecFlow

gaspar@specsolutions.eu
http://gasparnagy.com
@gasparnagy

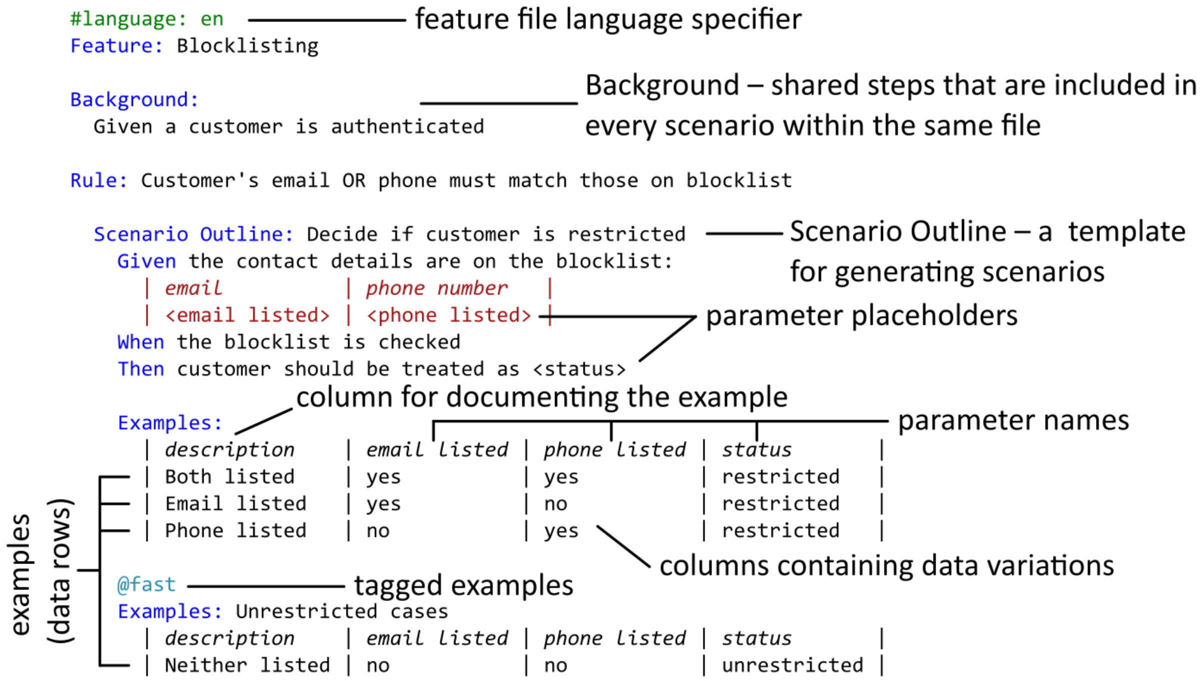
Learn more about our BDD courses at
https://www.specsolutions.eu/services/training/

GHERKIN CHEAT-SHEET

The cheat sheet is extracted from the “BDD Books: Formulation” book (https://bddbooks.com).

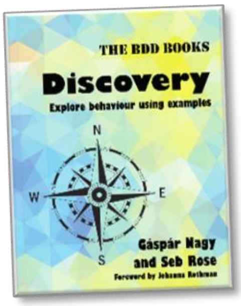


```
@delivery
Feature: Customers can collect their orders

Customers can choose to collect their order from the restaurant:
- whether they are authenticated or not
- whether they pay on order or pay on collection

Rule: Customer should be informed when their order will be ready for customer-collection

  # consider assuming an authenticated customer by default
  @important
  Scenario: Estimated time of order completion is displayed
    Given a customer is authenticated
    And the customer placed an order
      | time of order | preparation time |
      | 18:00         | 0:30             |
    When they choose to collect their order
    Then the estimated time of order completion should be displayed as 18:30
```

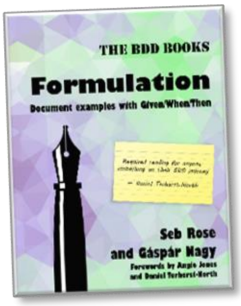


```
#language: en
Feature: Blocklisting

Background:
  Given a customer is authenticated

Rule: Customer's email OR phone must match those on blocklist

  Scenario Outline: Decide if customer is restricted
    Given the contact details are on the blocklist:
      | email          | phone number   |
      | <email listed> | <phone listed> |
    When the blocklist is checked
    Then customer should be treated as <status>

    Examples:
      | description    | email listed | phone listed | status       |
      | Both listed    | yes          | yes          | restricted   |
      | Email listed   | yes          | no           | restricted   |
      | Phone listed   | no           | yes          | restricted   |

    @fast
    Examples: Unrestricted cases
      | description    | email listed | phone listed | status       |
      | Neither listed | no           | no           | unrestricted |
```

feature file language specifier
Background – shared steps that are included in every scenario within the same file
Scenario Outline – a template for generating scenarios
parameter placeholders
column for documenting the example
parameter names
examples (data rows)
columns containing data variations
tagged examples



THE BDD BOOKS
Discovery
Explore behaviour using examples
N
W
E
S
Gáspár Nagy
and Seb Rose



Check out our books!

Find them on Amazon & Leanpub through http://bddbooks.com!

A1: Automate Menu scenario with controller automation

Automate scenario “Only active pizza menu items are listed on the menu page” using controller automation.

- Imagine what you would do to verify the scenario manually: Think it over how you would automate the different steps using controller automation.
- As you can see from the color of the steps, the “Given” step has been automated already. You will need no to automate the “When” and “Then” steps using the following steps.
- Generate step definition skeletons for the “When” and “Then” steps and add them to the existing `StepDefinitions\GeekPizzaStepDefinitions.cs` file. (To add step definitions to an existing file, you can use the “*Copy methods to clipboard*” button of the “*Define Steps*” dialog.)
- Implement step definitions to make the test fail: proves that the business rule illustrated by the scenario is not implemented yet! See Hints below, if you need help.
- Now (and only now) we can try to implement the application logic to satisfy the scenario. Check the `MenuController.GetPizzaMenu()` method, and you will see how this could be done.
- *Bonus:* Take a look at the provided automation for the “Given” step. (You can navigate to the step definition from the feature file with *Go to Definition (F12)*.) It saves the menu items directly to the database. *Is this a good idea? What are the disadvantages of it?*
- *Bonus:* If you have still some time, you can try automating the Given step using controller-automation (by using the `AdminController` class). Feel free to cleanup the code duplications as well in the Given step definition. You can also do this as a homework.

Hints

- The items on the menu page are retrieved using the “`GetPizzaMenu`” API controller method of the “`MenuController`”.
- How to call an API controller method? Check the step definition of the “When the client checks the home page” step!
- The method returns the items in a “`PizzaMenuModel`” structure.
- How to work with a result returned by an API controller method? Check the step definition of the “When the client checks the home page” and the “Then the home page main message should be: "Welcome to Geek Pizza!"” steps!
- To compare the result with the expected count, you can use the “`Assert.AreEqual(expected, actual)`” assertion method.

A2: Automate DataTable assertion of the Menu scenario

> Automate the missing "Then" step of the scenario "The pizzas are listed in alphabetically on the menu page" using controller automation with a simple FOR loop.

- Imagine what you would do to verify the scenario manually: Think it over how you would automate the different steps using controller automation.
- The "Given" and "When" steps have been defined (automated) already. You only need to work on the "Then" step. Review how the "Given" step is defined, you can see how you can work with DataTables (`Table` class) in SpecFlow!
- Generate step definition skeleton for the "Then" step and add them to the existing `StepDefinitions\GeekPizzaStepDefinitions.cs` file. (To add a single step definition to an existing file, the easiest way is to use the navigation command (*Go to Definition (F12)*) from the scenario, and choose *yes* on the popup dialog, in order to copy the method to the clipboard.)
- Implement step definitions **using a FOR loop first**! Make the test fail: proves that the business rule illustrated by the scenario is not implemented yet! See Hints below, if you need help.
- Implement the application logic to satisfy the scenario. It will be close to the one you did for the previous exercise.
- Now the test pass, but the logic you wrote into the "Then" step is pretty complex. Are you sure that it is bug-free? Test the test by making the scenario wrong! Try to change pizza names in the scenario or add/remove pizzas.
- *Bonus:* Extend the step definition so that it can also assert on the ingredients. Can you make it in a way that it works for both cases (with and without ingredients)? Use `Table.ContainsColumn`!

Hints

- A for loop that iterates through a list looks like this:

```
for (int i = 0; i < mylist.Count; i++)
{ … do somwthing with mylist[i] … }
```

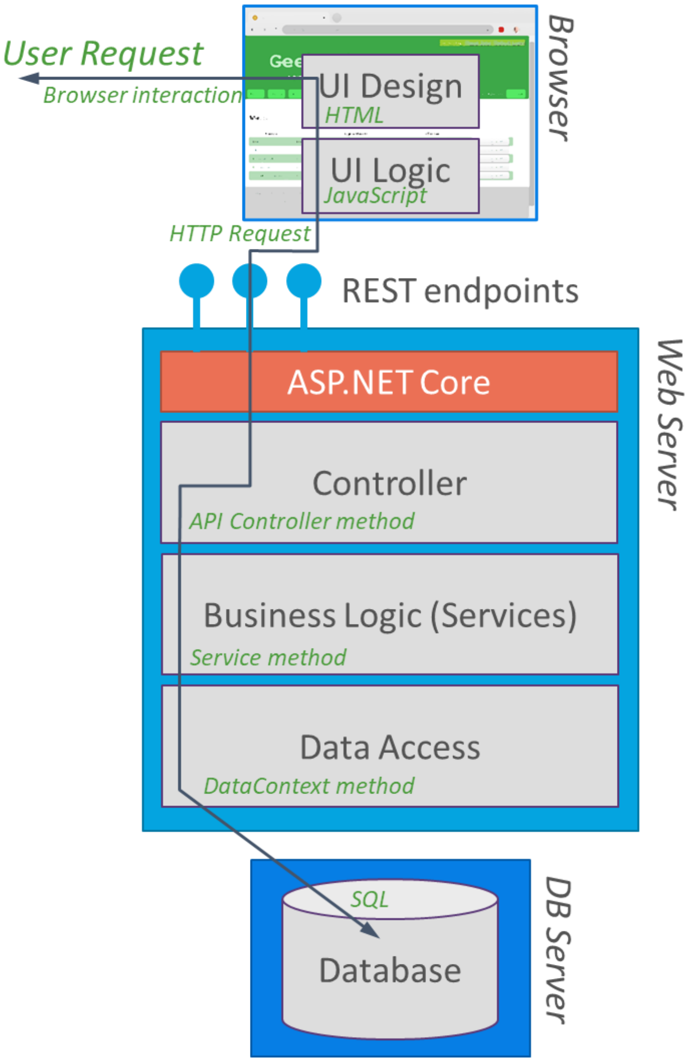
- The easiest way to write a FOR loop in Visual Studio is to type in "`for`" and press *TAB* twice.
- The length of the SpecFlow Table can be retrieved by `myTable.RowCount`.
- A row of the SpecFlow Table can be retrieved by `myTable.Rows[i]`.
- A cell within the row can be retrieved by `myRow["header"]` (or `myTable.Rows[i]["header"]`).

A3: Split Feature files and Step definition classes

Split feature file and step definition class to ensure a logical structure

- This is a refactoring exercise, so make sure the tests are passing before you would make any changes.
- Move Menu-related scenarios to a new feature file: `Menu.feature`. *Do they still pass? Did you have to make any changes in the step definitions to make it pass? Why?*
- The two "Given" step definitions are all related to the "admin" interface of the application. Extract them to a separate step definition class: `AdminStepDefinitions.cs`. *How do you have to decorate the new class to make the tests pass again?*
- Split the remaining step definitions in `GeekPizzaStepDefinitions` to `HomeStepDefinitions` and `MenuStepDefinitions` accordingly.

Architecture of the Geek Pizza App



A4: SHARE DATA BETWEEN STEP DEFINITIONS IN DIFFERENT CLASSES

Practice data sharing between step definitions placed in different classes.

- This is a refactoring exercise, so make sure the tests are passing before you would make any changes.
- The step definitions that are related to the same features of the system should be kept in the same step definition class. We have seen this in `HomeStepDefinitions` (the `HomePageModel` was shared) and also in `MenuStepDefinitions` (the list of pizzas was shared).
- In this exercise, there is a new scenario introduced in `Home.feature`: “The logged-in user name is shown on home page”. The related step definitions are currently automated in the `HomeStepDefinitions` class, but to keep a healthy structure, we would like to move the `GivenTheClientIsLoggedIn` method to the `AuthStepDefinitions`, because that does not belong to the home page but to the authentication.
- We have learned how to move step definitions to another class, but this one shares data (the authentication token, stored in the `_authToken` field) with the `WhenTheClientChecksTheHomePage` method, that supposed to remain in `HomeStepDefinitions`. To fix this, we will use the **Context Injection** feature of SpecFlow by performing the following steps:
    1. Review the context class that has been prepared for you already: `AuthContext`. (Normally you would need to create such a class with properties for the data you want to share.)
    2. Inject the context into *both step definition classes*, using constructor injection based on the following example:
       ```
       public class SomeStepDefinitions
       {
           private readonly AuthContext _authContext;
           public SomeStepDefinitions(AuthContext authContext)
           {
               _authContext = authContext;
           }
       }
       ```
    3. Wherever the code used the `_authToken` field (for reading or writing), use the `AuthToken` property of the injected context class instead: `_authContext.AuthToken`
- The scenarios are passing again! This means that SpecFlow created instances of the AuthContext class whenever it was needed. *How many `AuthContext` instance are created if you run all scenarios. Why?*
- *Bonus:* The `ThenTheUserNameOfTheClientShouldBeOnTheHomePage` method of the `HomeStepDefinitions` has the user name “Marvin” hard-coded. We need that because that is the user name we use for login in the `AuthStepDefinitions`. Can you change the automation code in a way that it only mentions “Marvin” once – in the `AuthStepDefinitions`? Hint: You need to extend the `AuthContext` class.

B1: REVIEW SCENARIO AUTOMATED USING WEB API AUTOMATION

Review the automation of the scenario "The user name should be shown on the home page if logged in" to find potential maintenance and scaling problems.

- We have switched over to a new automation solution that is in the `BddWithSpecFlow.GeekPizza.API.Specs` project. For the sake of simplicity, the scenarios we have automated using the API controllers have been removed.
- The current codebase contains a single scenario: "The user name should be shown on the home page if logged in" in the `Home.feature`. The scenario has been automated already using the step definitions in the `WebApiStepDefinitions` class. The scenario should be passing (make sure it does).
- The current automation solution is a "simplest possible" approach and while it makes the scenario pass it has a few problems. *Review the step definitions and make a note of the ones you find.*
- The following orientation questions might help:
  1. Is the code clean? Are there any code duplications?
  2. Assuming that the automation solution will contain many other similar GET and POST requests, would you use these automation blocks as samples for those? Why?
  3. Do these step definitions support reusability (reusing the steps in other scenarios, potentially in different order)?
  4. Does the current structure of the code support structure the step definitions into different step definition classes?
- *How could you fix these issues?* (You should not start fixing it, just make notes so that we can discuss together.)
- *Bonus:* What is the difference between sanity and functional checks? Why are they separated?

B2: AUTOMATE SCENARIOS USING WEB API AUTOMATION

Automate "Menu" and "Registration" scenarios using Web API automation.

- The "Given" step has been already automated in `AdminStepDefinitions.cs` (it adds the expected pizzas to the database).
- Automate the remaining steps by making GET and POST requests. The exact method, endpoint and input/output data you can find in the feature files as `#TODO` comments.
- The (empty) step definition classes are prepared (`MenuStepDefinitions` and `RegistrationStepDefinitions`) and the `WebApiContext` has been injected already.

- Check `HomeStepDefinitions.WhenTheClientChecksTheHomePage` and `AuthStepDefinitions.GivenTheClientIsLoggedIn` methods to see how you can use the WebApi Context to make a GET and POST request.
- *Was it easy? Did you enjoy the separation of the infrastructural details from the functional parts of the automation?*

B3: Eliminate Incidental Details

> Refactor scenario "Customer has a few different pizzas in the basket" in MyOrder.feature so that you eliminate incidental details.

- This is a refactoring exercise, so make sure the tests are passing before you would make any changes.
- For the sake of this exercise, you should focus only on *this* scenario and do not change other scenarios having similar problems. In order to focus on a small set of exercises, a good practice to tag those scenarios temporarily (I use `@focus` – the scenarios I am focusing on) and filter the *Test Explorer* window to only show the tagged scenarios by specifying the `Trait:focus` filter criteria. Try what "*Run All*" does when the list is filtered!
- There are many things to improve in this scenario, let's fix them one-by-one. Make sure that the scenario still passes after each step.
    1. The values for "ingredients", "calories" and "inactive" are not important in this scenario. Remove these columns from the "Given" step and make sure that the step definition populates the menu items with some meaningful defaults (if they are not provided). (Hint: you can select a rectangular area by pressing the *Alt* button when performing the selection)
    2. Let's suppose, we want to use these pizza menu items for all scenarios in this project, so we would move it to a test baseline (baseline database) and remove it from the scenario. Hint: check `DatabaseHooks.cs`. *How would you document the baseline so that the entire team knows what pizzas are available for testing?*
    3. Let's add a default user as well to our baseline (so remove "Given there is a user registered with user name 'Ford' and password '1423'"), and use that for login.
    4. Get rid of the concrete user name and password for login and replace it with a "Given the client is logged in" step that logs in the default user. (We already have a step definition for this anyway.)
    5. Move the "Given I am logged in" step to a "Background" section, in order to be able to share it with other scenarios in the same file. (Check Gherkin Cheat-Sheet on page 2 for the exact "Background" syntax.)
    6. To see an alternative for the "Background", comment out the Background and tag the scenario with @login or @authenticated. Implement a hook that runs for every scenario marked with the selected tag and logs in the user. For now, implement this in the `AuthStepDefinitions`

class. *What is the difference between the test baseline, the "Background" section and the tags with hooks providing automation logic?*

7. *Bonus:* Get rid of the duplicated DataTable in the "Given" and "Then" steps by replacing the "Then" step with "Then the ordered items should be listed on the my order page". Hint: try to "remember" the data table of the "Given" step, so that you can use it in the "Then" step.

- If you followed all these steps, you have a nice and simple scenario like this. Yay!

```
@login
Scenario: Customer has a few different pizzas in the basket
       Given the client has the following items in the basket
              | name       | size   |
              | Margherita | Small  |
              | BBQ        | Medium |
       When the client checks the my order page
       Then the ordered items should be listed on the my order page
```

B4: Handling relative dates

Make the Order Details scenarios pass with relative date parameters, like "today", "tomorrow" or "5 days later"

- All steps of the Order Details scenarios (`OrderDetails.feature`) are defined, but they still fail, because the step definitions expect a `DateTime` parameter, but the scenarios use "`today`", "`tomorrow`" or "`5 days later`". If you would change these to concrete dates (in *MM/DD/YYYY* format), they would pass. (Try it!)
- Make the scenarios pass with the relative dates. For that, you need to implement step argument transformations that extend the conversion system of SpecFlow. (You can find examples among the hints below.)
- *Bonus:* Change the parameter type of `WhenTheClientSpecifiesDateAtTimeAsDeliveryTime` to `TimeSpan` and make it pass. Make sure you can specify "`noon`" or "`5pm`" as time.

Hints

- A step argument transformation that converts "five":

```
[StepArgumentTransformation("five")]
public int ConvertFive()
{
     return 5;
}
```

- A step argument transformation that converts "one more than X":

```
[StepArgumentTransformation(@"one more than (.*)")]
public int ConvertOneMoreThan(int what)
{
     return what + 1;
}
```

- Get today and tomorrow with `DateTime.Today` and `DateTime.Today.AddDays(1)`

B5: Practice using Scenario Outlines

> Convert the scenario “Pizza is ordered for tomorrow” to a Scenario Outline and provide different dates and times as examples.

- Define a Scenario Outline for the different date and time options. What should be the scenario outline name? (Check Gherkin Cheat-Sheet on page 2 for the exact “Scenario Outline” syntax.)

date	Time
today	12:30
today	18:00
tomorrow	10:00
2 days later	13:30

- Add an additional first column (e.g. “description”) to the Examples table to describe the purpose of the individual examples (be creative!). Check the name of the generated tests!
- Add a second “Examples” block and provide further test cases.

B6: Save log on error

> Setup the automation infrastructure in a way that it automatically saves the API call log to a file when the scenario fails.

- A scenario failure can be detected in an `[AfterScenario]` hook by checking if the `_scenarioContext.TestError` property contains an exception (and it is not `null`).
- Implement such a hook in the `WebApiHooks.cs` file and save the log file to the output folder if the scenario fails. You can use the `_webApiContext.SaveLog()` method to save the file. Does it save the files to the right folder?

Hints

- Use the provided `TestFolders` class to access the “correct” output folder.

C1: Driver for POST API call (UserApiDriver)

Extract automation logic of the RegistrationStepDefinitions to use to a driver class.

- Create a driver class for the user registration in `RegistrationStepDefinitions`. Call the new class `UserApiDriver` and save it to the `Drivers` folder. You can use the `AuthApiDriver` class as an example.
- Update the step definition methods in `RegistrationStepDefinitions` to use the driver object. (You need to inject the driver object to the step definition class first.)

C2: Driver for GET API call (HomeApiDriver)

Extract the automation logic of the HomeStepDefinitions call to a driver class.

- Create a driver class for the home page automation in `HomeStepDefinitions`. Call the new class `HomeApiDriver` and save it to the `Drivers` folder.
- Remove the `WebApiContext` field and its injection from the `HomeStepDefinitions` class, because it is not needed anymore. *Is it valid that our step definition class does not need the `WebApiContext`, even though we automate through Web API?*
- *Bonus:* Refactor the remaining step definitions that directly access the `WebApiContext` into driver classes. (The C2.1, C2.2 in the `Solutions` folder contain implementations for these: MenuApiDriver and OrderApiDriver.)