

# Project-Based Exercises: Appium Desktop

Problems for exercises and homework for the "Software Quality Assurance" course from the official "Applied Programmer" curriculum.

## Installing and Running the Web API in Repl.It

To run the web API, go to the "Eventures Web API Repl": <https://eventures-web-api.softuniorg.repl.co> (please give it some time it loads a bit slow!)



## Installing and Running the API on Your PC (Alternatively)

Open the project in Visual Studio, compile and run it. You need to have the following software installed:

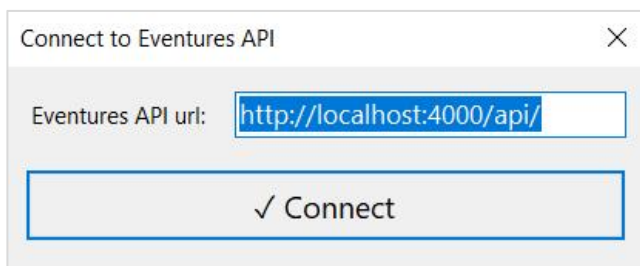
- .NET 5 or later version (<https://dotnet.microsoft.com>)
- MS SQL Server LocalDB (<https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb>)
- Visual Studio 2019 or later version (<https://visualstudio.microsoft.com>)

## 1. Appium Desktop Testing of C# Desktop App

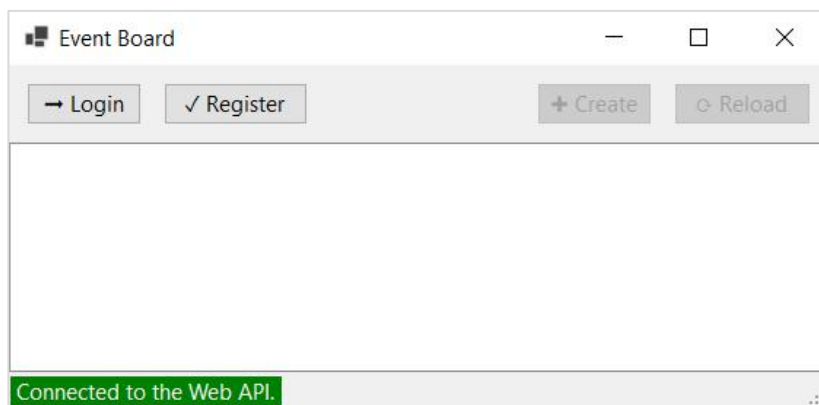
### Eventures Desktop App Overview

You are already familiar with the "Eventures" app. It also has a desktop app- "Eventures.DesktopApp", connected to the "Eventures" Web API (it should be running in order for the desktop app to connect to it). The desktop app has the following functionalities:

- Connect to the API URL:

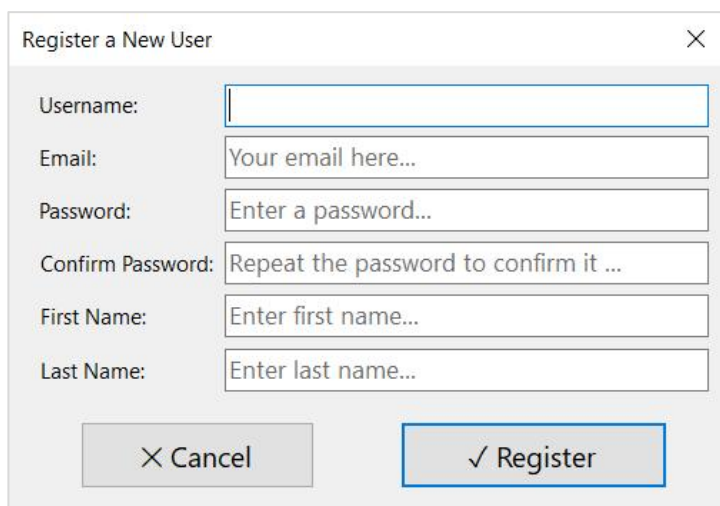


When you connect you will see the following "Event Board" window:



The "Event Board" window has a title bar with standard window controls. Below the title bar is a toolbar with four buttons: "Login" (with a right arrow icon), "Register" (with a checkmark icon), "+ Create", and "Reload" (with a circular arrow icon). The main area of the window is empty. At the bottom, there is a green status bar that reads "Connected to the Web API."

- **Register** a new user:

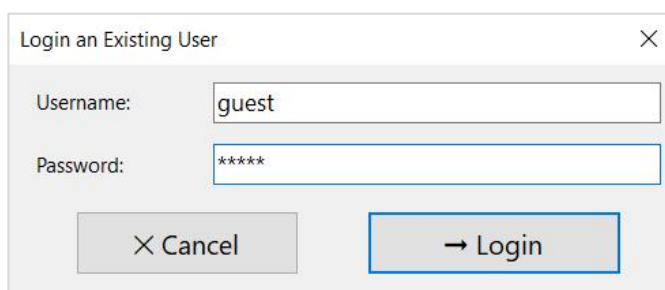


The "Register a New User" dialog box contains the following fields and buttons:

- Username:
- Email:
- Password:
- Confirm Password:
- First Name:
- Last Name:
- Buttons: "Cancel" (with an X icon) and "Register" (with a checkmark icon).

If you want to **register** a user to the app, **fill in the fields** and **click on [Register]**. If you already **have an account**, use the “**Login**” functionality to **authenticate**.

- **Login** an existing user:



The "Login an Existing User" dialog box contains the following fields and buttons:

- Username:
- Password:
- Buttons: "Cancel" (with an X icon) and "Login" (with a right arrow icon).

To **log in**, **fill in the form** with your **valid credentials**. Upon successful registration or login, **all events will be displayed** and the **[Create]** and **[Reload]** buttons will be **enabled**.

- **View all events** from the API on the “**Event Board**”:

Event Board

→ Login

✓ Register

+ Create

↻ Reload

Id	Name	Place	Start	End	Tickets	Price	Owner
1	Softuniada 2022	Sofia	17-04-2022 07:44	18-04-2022 07:44	200	12.50	guest
2	OpenFest 2022	Online	11-02-2023 07:44	11-02-2023 15:44	500	10.00	guest
3	Microsoft Build 2022	Online	26-07-2022 07:44	26-07-2022 19:44	1000	0.00	guest

<

>

Load successful: 3 events loaded.

- **Create a new event:**

Create a New Event

Name:

Place:

Start:  End:

Total Tickets:  Price Per Ticket:

✕ Cancel

✓ Create

Note that if any step is **unsuccessful**, a different **error message** will appear in the “**Status Box**” or an **error window** will appear:

✕

Name field is required.

Place field is required.

OK

**Start** the **Web API** and the **Desktop App**. **Examine** the Desktop App by yourself. Let’s now start writing test for it.

## Create the Tests Base Class

As you already now, it is a **good practice** to do the **setups** in a **separate class** from the tests when possible. Let’s create the “**AppiumTestsBase**” class and write its “**OneTimeSetUpBase()**” method. We will need the following **fields** in the class:

```
public class AppiumTestsBase
{
    protected TestDb testDb;
    protected ApplicationDbContext dbContext;
    private TestEventuresApp<Startup> testEventuresApp;
    protected string baseUrl;
    private AppiumLocalService appiumLocalService;
    private string ApiPath = @"../../../../../Eventures.WebAPI";
    private string AppPath = @"../../../../../Eventures.DesktopApp/bin/Debug" +
        @"/net5.0-windows/Eventures.DesktopApp.exe";
    protected WindowsDriver<WindowsElement> driver;
    protected WebDriverWait wait;
```

First, in the `OneTimeSetUpBase()` method, we shall **initialize** a **testing database** with a **database context**. Also, we need to **initialize** the `TestEventuresApp<TStartup>` class, which will **run** our **Web API** on a **server** and get its **URI**, which is different from the one which we connect with through the Desktop App. You already know about these **test helper classes** and what they do (if you don't, look it up in the **"API-Testing"** exercise), so use them in the `OneTimeSetUpBase()` method like this:

```
[OneTimeSetUp]
public void OneTimeSetUpBase()
{
    this.testDb = new TestDb();
    this.dbContext = testDb.CreateDbContext();
    this.testEventuresApp = new TestEventuresApp<Startup>(testDb, ApiPath);
    this.baseUrl = this.testEventuresApp.ServerUri;
```

Next, you should **initialize Appium Local Service** and **start it** in order to **start the Appium server** automatically:

```
// Initialize Appium Local Service to start the Appium server automatically
appiumLocalService = new AppiumServiceBuilder().UsingAnyFreePort().Build();
appiumLocalService.Start();
```

To **connect to Appium**, you should set **Appium options** with a **relative path to the Desktop App .exe start file**. The **.exe** file is created when the **Desktop App** is **run at least one time** (run the app if you haven't done it already). The **relative path** is given to you as a **field value**, so you should just **initialize the AppiumOptions()**:

```
var appiumOptions = new AppiumOptions() { PlatformName = "Windows" };
var fullPathName = Path.GetFullPath(AppPath);
appiumOptions.AddAdditionalCapability("app", fullPathName);
```

Then, **initialize the "WindowsDriver<WindowsElement>" class** with the **Appium Local Service** and **Appium options** and **set an implicit wait** to the driver:

```
// Initialize the Windows driver with Appium local service and options
driver = new WindowsDriver<WindowsElement>(
    appiumLocalService, appiumOptions);

// Set an implicit wait for the UI interaction
driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(10);
```

Finally, **set an explicit wait** for the UI interaction with the `WebDriverWait` class:

```
// Set an explicit wait for the UI interaction
wait = new WebDriverWait(driver, TimeSpan.FromSeconds(60));
}
```

These are all the initializations and classes you need in order to **run local Appium tests on a desktop app**. The last thing you should do is create the “**OneTimeTearDownBase()**” method, which should **close the app** and **quit the driver**, **dispose of the Appium Local Service**, and **dispose of the local Web API service**:

```
[OneTimeTearDown]
0 references
public void OneTimeTearDownBase()
{
    // Close the app and quit the driver
    driver.CloseApp();
    driver.Quit();

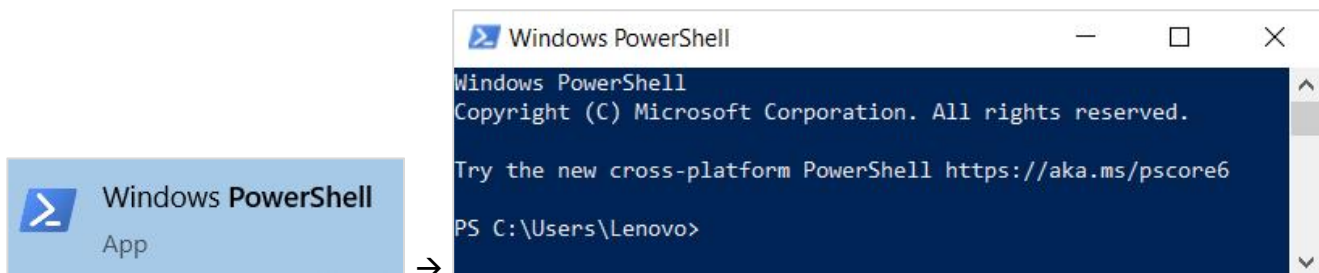
    // Dispose of the Appium Local Service
    appiumLocalService.Dispose();

    // Stop and dispose the local Web API server
    this.testEventuresApp.Dispose();
}
```

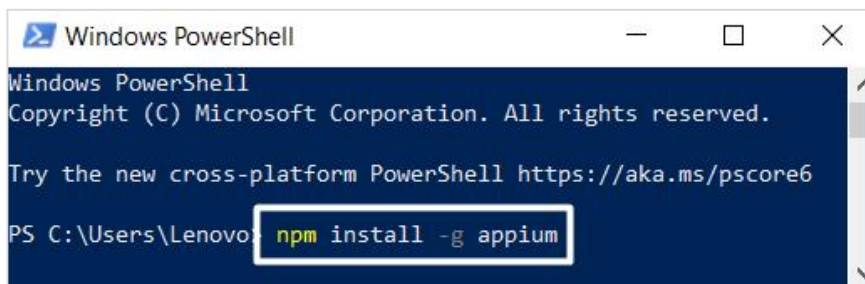
Let’s now write the tests.

## Install Appium Globally

In order to start the **Appium Local Service**, you will need to **install Appium globally**. On **Windows**, open a **Windows Powershell Command Prompt**:



Then, run the “**npm install -g appium**” command:



Now **Appium** should be **installed**, and the **Appium Local Service** should **work correctly**.

## Create the Desktop App Tests

Our aim now is to **fully test the “Eventures” Desktop App**. These are the **test scenarios** we should cover:



- **Connect** with an **empty URL** → the **"Connect to Eventures API"** window should appear again, and an **error message** should be displayed in the status box
- **Connect** with an **invalid URL** (with **wrong port number** for example) → the **"Connect to Eventures API"** window should appear again, and an **error message** should be displayed in the status box
- **Connect** with a **valid URL** → the **"Event Board"** window and a **success message** should appear
- **Register** with **valid data** → the **"Event Board"** window with **all events** and a **success message** should appear and **all buttons should be enabled**
- **Login** with **valid data** → the **"Event Board"** window with **all events** and a **success message** should appear and **all buttons should be enabled**
- **Reload** with an **authorized user** → the **"Event Board"** window with **all events** and a **success message** should appear
- **Create a new event:**
  - Insert **invalid data** at first → an **error window** should appear
  - Then insert **valid data** → the **"Event Board"** window with **all events (including the newly created one)** and a **success message** should appear

For the following test cases we need to **order them appropriately**, so that they **execute correctly**. It is important to **set the correct order** because, for example, the registration tests cannot be before the connection ones, as one cannot register without being connected to the Web API. That's why we should have the following order: **connection tests → registration tests → other tests (their order doesn't matter, as they only need to be after the registration, so that we have an authorized user)**. The **tests scenarios** above are given in **correct order** - we will use it. Let's write the tests.

First, our **"AppiumTests"** class should **inherit** the **"AppiumTestsBase"** class. Also, we will **create fields** for the data we will use multiple times:

```
public class AppiumTests : AppiumTestsBase
{
    private string username = "user" + DateTime.Now.Ticks.ToString().Substring(10);
    private string password = "pass" + DateTime.Now.Ticks.ToString().Substring(10);
    private const string EventBoardWindowName = "Event Board";
    private const string CreateEventWindowName = "Create a New Event";
    private const string RegisterAUserWindowName = "Register a New User";
}
```

Then, we shall write the **test methods**.

## Connect with an Empty URL

In our **first test scenario**, we should **locate** the **URL field** and **clear** it, so that it is **empty**:

```
[Test, Order(1)]
public void Test_Connect_WithEmptyUrl()
{
    // Locate the URL field, clear it and leave it with an empty URL
    var apiUrlField = driver.FindElementByAccessibilityId("textBoxApiUrl");
    apiUrlField.Clear();
}
```

Next, **locate** and **click** on the **[Connect]** button:

```
// Locate and click on the [Connect] button
var connectBtn = driver.FindElementByAccessibilityId("buttonConnect");
connectBtn.Click();
```

Then, assert the "Connect to Eventures API" window reappeared. Wait for it like this:

```
// Wait and assert the "Connect to Eventures API" window appeared again
var windowAppears = this.wait
    .Until(s => driver.PageSource.Contains("Connect to Eventures API"));
Assert.IsTrue(windowAppears);
```

Locate the status box and assert that it contains the "Error: Value cannot be null." error message. Use the explicit wait to wait until the driver loads the message:

```
// Wait until the driver loads the message
// and assert an error message is displayed in the status box
var statusTextBox = driver
    .FindElementByXPath("/Window/StatusBar/Text");

var messageAppears = this.wait
    .Until(s => statusTextBox.Text.Contains("Error: Value cannot be null."));
Assert.IsTrue(messageAppears);
}
```

## Connect with an Invalid URL

The second test is pretty similar to the previous one, so do it on your own. This time, fill in the API URL field with a URL address with an invalid port (an invalid port is any port, which is different from the one in the server base URL). The rest of the test is the same as the previous one - only the displayed error message should be this one: "Error: HTTP error `No connection`".

## Connect with a Valid URL

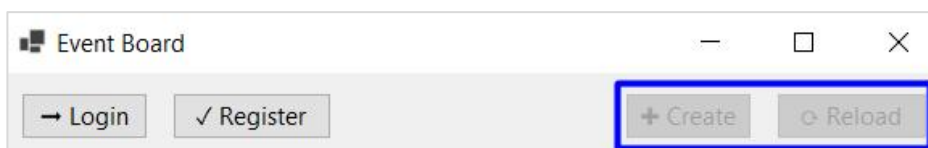
In this test, you should use the base URL from the server to connect correctly. You may notice that the base server URL connects to "127.0.0.1" - this is your loopback address, which is the same as "localhost". After successful connection, the "Event Board" window should appear, and the status box should contain a successful message "Connected to the Web API.". Write the test method yourself.

- Locate the text box URL and clear it.
- Send {baseUrl}/api to the text box.
- Locate the connect button and click it.
- Assert that the page source contains the EventBoardWindowName variable.
- Locate the status bar text wait until it equals "Connected to the Web API."
- Assert the above condition is true.

## Register with Valid Data

In this test, we should register a new user, which will be logged in the app by default.

First, locate and assert that the [Create] and [Reload] buttons on the "Event Board" window are disabled, as an unauthorized user should not be able to click on them:



Do it like this:

```
[Test, Order(5)]
public void Test_Register()
{
    // Assert the [Create] and [Reload] buttons are disabled
    var createBtn = driver.FindElementByAccessibilityId("buttonCreate");
    Assert.IsFalse(createBtn.Enabled);
    var reloadBtn = driver.FindElementByAccessibilityId("buttonReload");
    Assert.IsFalse(reloadBtn.Enabled);
}
```

Then, find the **[Register]** button by **id** and **click** on it to open the "Register a New User" window:

```
// Locate and click on the [Register] button
var registerBtn = driver.FindElementByAccessibilityId("buttonRegister");
registerBtn.Click();
```

Next, **fill in all the registration form fields** with **valid data** like this:

```
// Fill in valid data in the fields
var usernameField = driver.FindElementByAccessibilityId("textBoxUsername");
usernameField.Clear();
usernameField.SendKeys(this.username);

var emailField = driver.FindElementByAccessibilityId("textBoxEmail");
emailField.Clear();
emailField.SendKeys(this.username + "@mail.com");

var passwordField = driver.FindElementByAccessibilityId("textBoxPassword");
passwordField.Clear();
passwordField.SendKeys(this.password);

var confirmPasswordField = driver
    .FindElementByAccessibilityId("textBoxConfirmPassword");
confirmPasswordField.Clear();
confirmPasswordField.SendKeys(this.password);
var firstNameField = driver.FindElementByAccessibilityId("textBoxFirstName");
firstNameField.Clear();
firstNameField.SendKeys("Test");

var lastNameField = driver.FindElementByAccessibilityId("textBoxLastName");
lastNameField.Clear();
lastNameField.SendKeys("User");
```

Then, **click** on the **[Register]** button under the "Register" form:

```
// Click on the [Register] button under the "Register" form
var registerConfirmBtn = driver
    .FindElementByAccessibilityId("buttonRegisterConfirm");
registerConfirmBtn.Click();
```

At the end, **make the needed assertions**. Assert the "Event Board" window appears. Also, you need to **assert that events are loaded correctly**, and a **success message appears**. However, **extracting the event from the API** may be a



little bit slow. For this reason, **locate the status box** but **wait until it contains the "Load successful" message**. Then, **get the events from the database** and **assert the success message contains their count**. Do the assertions like this:

```
// Assert the "Event Board" windows appears
Assert.That(driver.PageSource.Contains("Event Board"), Is.True);

// Wait until the events are loaded
var statusTextBox = driver.FindElementByXPath("//div[contains(text(),'Load successful')]");

var messageAppears = this.WaitUntil(driver.PageSource.Contains("Load successful"));
Assert.IsTrue(messageAppears);

// Get the events count from the database
var eventsInDb = this.dbContext.Events.Count();

// Assert a success message is displayed in the status box
Assert.AreEqual($"Load successful: {eventsInDb} events loaded.", statusTextBox.Text);
```

Also, assert that the [Create] and [Reload] buttons are now enabled:

```
// Assert the [Create] and [Reload] buttons are enabled
Assert.IsTrue(createBtn.Enabled);
Assert.IsTrue(reloadBtn.Enabled);
}
```

## Login with Valid Data

In this test, you should **use the user credentials from the registration to login to the app**. Start by **clicking** on the [Login] button in the "Event Board" window. Then, **fill in** the "Login" form with **correct username** and **password** and **click** on the [Login] button under the "Login" form:

```
[Test]
0 references
public void Test_Login()
{
    // Locate and click on the [Login] button
    var loginBtn = driver.FindElementByAccessibilityId("buttonLogin");
    loginBtn.Click();

    // Fill in valid data in the fields
    var usernameField = driver.FindElementByAccessibilityId("textBoxUsername");
    usernameField.Clear();
    usernameField.SendKeys(this.username);

    var passwordField = driver.FindElementByAccessibilityId("textBoxPassword");
    passwordField.Clear();
    passwordField.SendKeys(this.password);

    // Click on the [Login] button under the "Login" form
    var loginConfirmBtn = driver.FindElementByAccessibilityId("buttonLoginConfirm");
    loginConfirmBtn.Click();
}
```

Next, **make the assertions**. They are absolutely the same as these from the registration test, so **make them on your own**.

## Create a New Event

In this test case, we shall **create a new valid event** and **assert that it is created correctly**. To start with, **get the events count from the database** before the creation of a new event:

```
[Test]
0 references
public void Test_CreateEvent()
{
    // Get the events count before
    var eventsCountBefore = this.dbContext.Events.Count();
```

Next, **locate and click** on the **[Create]** button in the **"Event Board" window** and **assert** that the **"Create a New Event" window appeared**:

```
// Locate and click on the [Create] button
var createBtn = driver.FindElementByAccessibilityId("buttonCreate");
createBtn.Click();

// Assert the "Create a New Event" windows appears
Assert.That(driver.PageSource.Contains(AppiumTests.CreateEventWindowName));
```

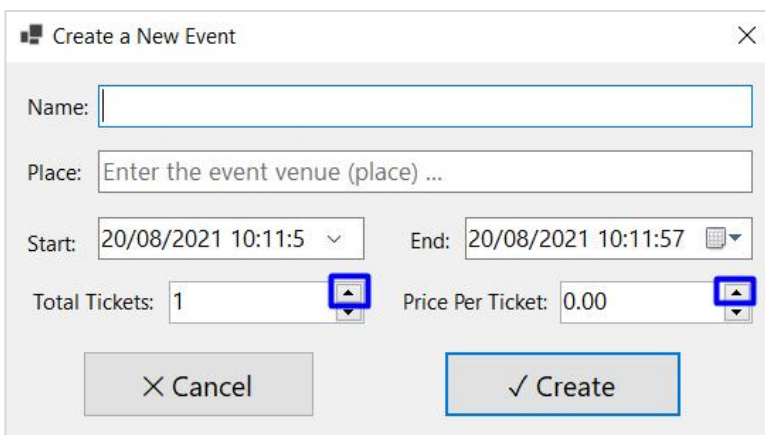
We shall now **add valid event name and valid event place**. Get the fields for **"name"** and **"place"** and **fill them in**:

```
// Fill in valid event name
var eventName = "Fun Event" + DateTime.Now.Ticks;
var nameField = driver.FindElementByAccessibilityId("textBoxName");
nameField.Clear();
nameField.SendKeys(eventName);

// Fill in a valid event place
var placeField = driver.FindElementByAccessibilityId("textBoxPlace");
var eventPlace = "Beach";
placeField.SendKeys(eventPlace);
```

Look at the **"start"** and **"end"** **datetime fields** - they have **future dates set by default**. In this case, we may not change the default values, as they are already **valid**.

Now we want to **locate the up-arrow buttons [^]** for the event **"tickets"** and **"price"** fields:



The screenshot shows a 'Create a New Event' dialog box with the following fields and controls:

- Name:** A text input field.
- Place:** A text input field with placeholder text 'Enter the event venue (place) ...'.
- Start:** A datetime picker showing '20/08/2021 10:11:5'.
- End:** A datetime picker showing '20/08/2021 10:11:57'.
- Total Tickets:** A numeric input field with the value '1' and an up-arrow button.
- Price Per Ticket:** A numeric input field with the value '0.00' and an up-arrow button.
- Buttons:** 'Cancel' and 'Create' buttons at the bottom.

Then, we will **click on them to increase the fields values**. However, they **don't have ids**, so we will get **both of them together**, as they have **the same name**. Do it like this:

```
// Locate the up arrow buttons
var upBtns = driver.FindElementsByName("Up");

// Click the second up arrow button to increase the event tickets field value
var ticketsUpBtn = upBtns[1];
ticketsUpBtn.Click();

// Click the first up arrow button to increase the event price field value
var priceUpBtn = upBtns[0];
priceUpBtn.Click();
priceUpBtn.Click();
```

Click on the [Create] button in the "Create a New Event" window. From there we assert the **DB events count**, the **status message** and the **status of the windows that are active**:

```
// Click on the [Create] button under the "Create" form
var createConfirmationBtn = driver
    .FindElementByAccessibilityId("buttonCreateConfirm");
createConfirmationBtn.Click();

// Wait until in the database the events count is increased by 1
this.wait.Until(
    s => this.testDb.CreateDbContext().Events.Count() == eventsCountBefore + 1);

// Assert a success message is displayed in the status bar
var loadSuccessfulMsgAppered =
    this.wait.Until(s => driver.FindElementByXPath("/Window/StatusBar/Text")
        .Text.Contains($"Load successful: {eventsCountBefore + 1} events loaded"));
Assert.IsTrue(loadSuccessfulMsgAppered);

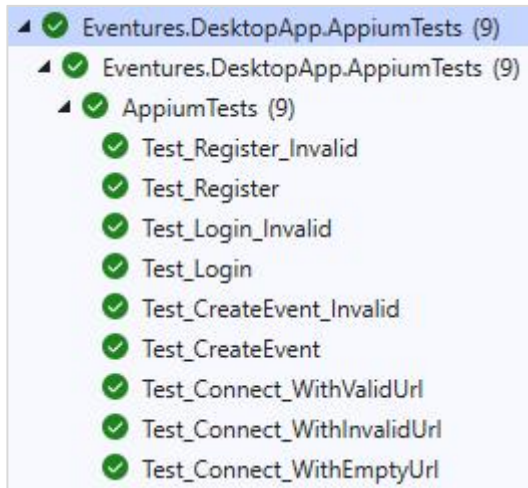
// Assert the "Create a New Event" windows disappears
string pageSource = driver.PageSource;
Assert.That(!pageSource.Contains(AppiumTests.CreateEventWindowName));

// Assert the "Event Board" windows appears
Assert.That(pageSource.Contains(AppiumTests.EventBoardWindowName));

// Assert the new event is displayed correctly
Assert.That(pageSource.Contains(eventName));
Assert.That(pageSource.Contains(eventPlace));
Assert.That(pageSource.Contains(this.username));
}
```

At the end, **run all tests**. They should be **successful**:





## Additional Tests

As you may have noticed, we **haven't written tests** for the **"Register"** and **"Login"** functionalities when **invalid user data is entered**. **Write tests on your own**. Think about the **test order** and what the new tests place in it should be, so that they are **executed properly**.

### Register with Invalid Data

Now do the same thing but this time insert invalid data into one of the fields.

- Assert that an error box appears such as: **"Email field must have a valid email address."**
- Click on the **[Ok]** button.
- Click on the **[Cancel]** button.
- Assert the **"Register a New User"** window **disappears**.
- Assert the **"Event Board"** window **appears**.

### Login with Invalid Data

- Locate and click on the **[Login]** button.
- Fill in invalid data in the username field, e.g., empty string.
- Click on the **[Login]** button under the **"Login"** form.
- Assert the **"Event Board"** windows appears.
- Assert a failure message is displayed and contains the error -> **"Error: HTTP error `BadRequest`"**.

### Create an Event with Invalid Data

To create an invalid event, start the same way by clicking on the **[Create]** button and asserting that the **"Create a New Event"** window appears. Fill in a valid event name. From there:

- Fill in invalid event place, e.g., empty string.
- Click on the **[Create]** button under the **"Create"** form.
- Assert an error window appears -> **"Place field is required."**
- Click on the **[Ok]** button to close the window.
- Click on the **[Cancel]** button to close the window.
- Assert the **"Create a New Event"** windows disappears.
- Assert the **"Event Board"** windows appears