CHAPTER 10 – INTRODUCING REACTOR

* **Imperative code** is a lot like that absurd hypothetical newspaper subscription. It’s a serial set of tasks, each running one at a time, each after the previous task. Data is processed in bulk and can’t be handed over to the next task until the previous task has completed its work on the bulk of data.
* **Reactive code** is a lot like a real newspaper subscription. A set of tasks is defined to process data, but those tasks can run in parallel. Each task can process subsets of the data, handing it off to the next task in line while it continues to work on another subset of the data.
* reactive programming is **functional and declarative** in nature. Rather than describe a set of steps that are to be performed sequentially, reactive programming **involves describing a pipeline or stream through which data flows**. Rather than requiring the data be available to be processed as a whole, a reactive stream **processes data as it becomes available**
* **Backpressure** is a means by which consumers of data can avoid being overwhelmed by an overly fast data source, by establishing limits on how much they’re willing to handle..
* **10.1.1 DEFINING REACTIVE STREAMS**
* The Reactive Streams specification can be summed up by **four interface** definitions: **Publisher, Subscriber, Subscription, and Processor**
* The **Publisher interface** declares a **single method**, **subscribe()**, through which a **Subscriber can subscribe** to the Publisher:

public interface Publisher<T> {

void subscribe(Subscriber<? super T> subscriber);

}

* Once a **Subscriber has subscribed**, it can **receive events from the Publisher**. Those events are **sent via methods on the Subscriber interface**:

public interface Subscriber<T> {

void onSubscribe(Subscription sub);

void onNext(T item);

void onError(Throwable ex);

void onComplete();

}

* When the **Publisher calls onSubscribe()**, it **passes a Subscription object to the Subscriber**. It’s **through the Subscription that the Subscriber can manage its subscription**:

public interface Subscription {

void request(long n);

void cancel();

}

* The **Subscriber** can call **request()** to **request that data be sent**, or it can call **cancel()** to indicate that it’s no longer interested in receiving data and is **canceling the subscription**. When **calling request()**, the Subscriber **passes in a long value** to indicate **how many data items it’s willing to accept**. This is where **backpressure** comes in, **preventing the Publisher from sending more data than the Subscriber is able to handle**. After the **Publisher** has **sent as many items as were requested**, the **Subscriber** can call **request() again to request more**.
* Once the **Subscriber** has **requested data**, the data starts flowing through the stream. For every item that’s published by the Publisher, the **onNext() method will be called to deliver the data to the Subscriber**. If there are **any errors, onError() is called**. If the Publisher has **no more data to send and isn’t going to produce any more data**, it will **call onComplete()** to tell the Subscriber that it’s out of business.
* As for the **Processor interface**, it’s a **combination of Subscriber and Publisher**, as shown here:

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}

* + **As a Subscriber**, a Processor will receive data and process it in some way. Then it will switch hats and act **as a Publisher** to publish the results to its Subscribers
* 10.2 GETTING STARTED WITH REACTOR
  + suppose you want to **take a person’s name, change all of its letters to uppercase, use it to create a greeting message, and then finally print it**. In **an imperative programming model**, the code would look something like this:

String name = "Craig";

String capitalName = name.toUpperCase();

String greeting = "Hello, " + capitalName + "!";

System.out.println(greeting);

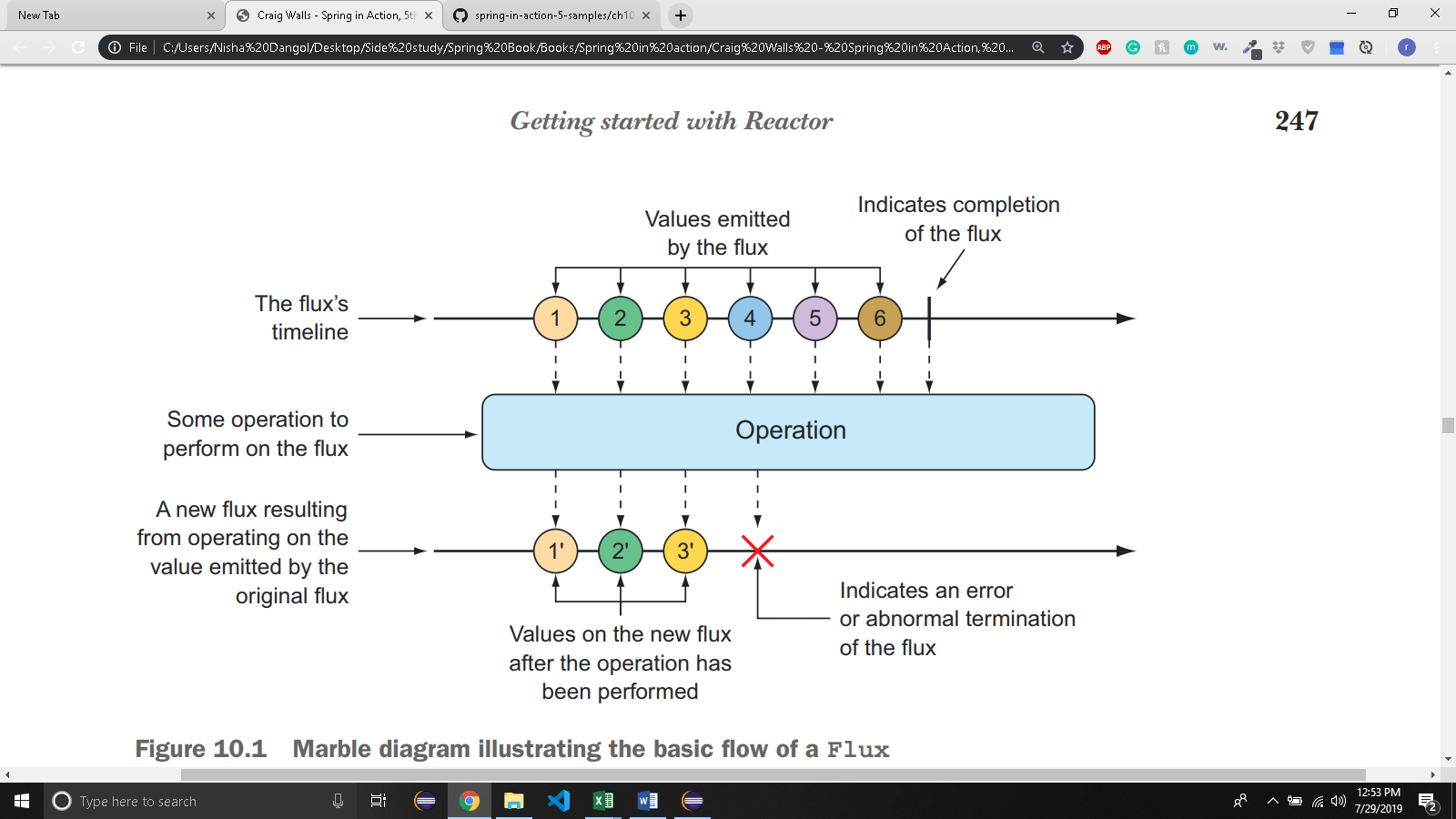
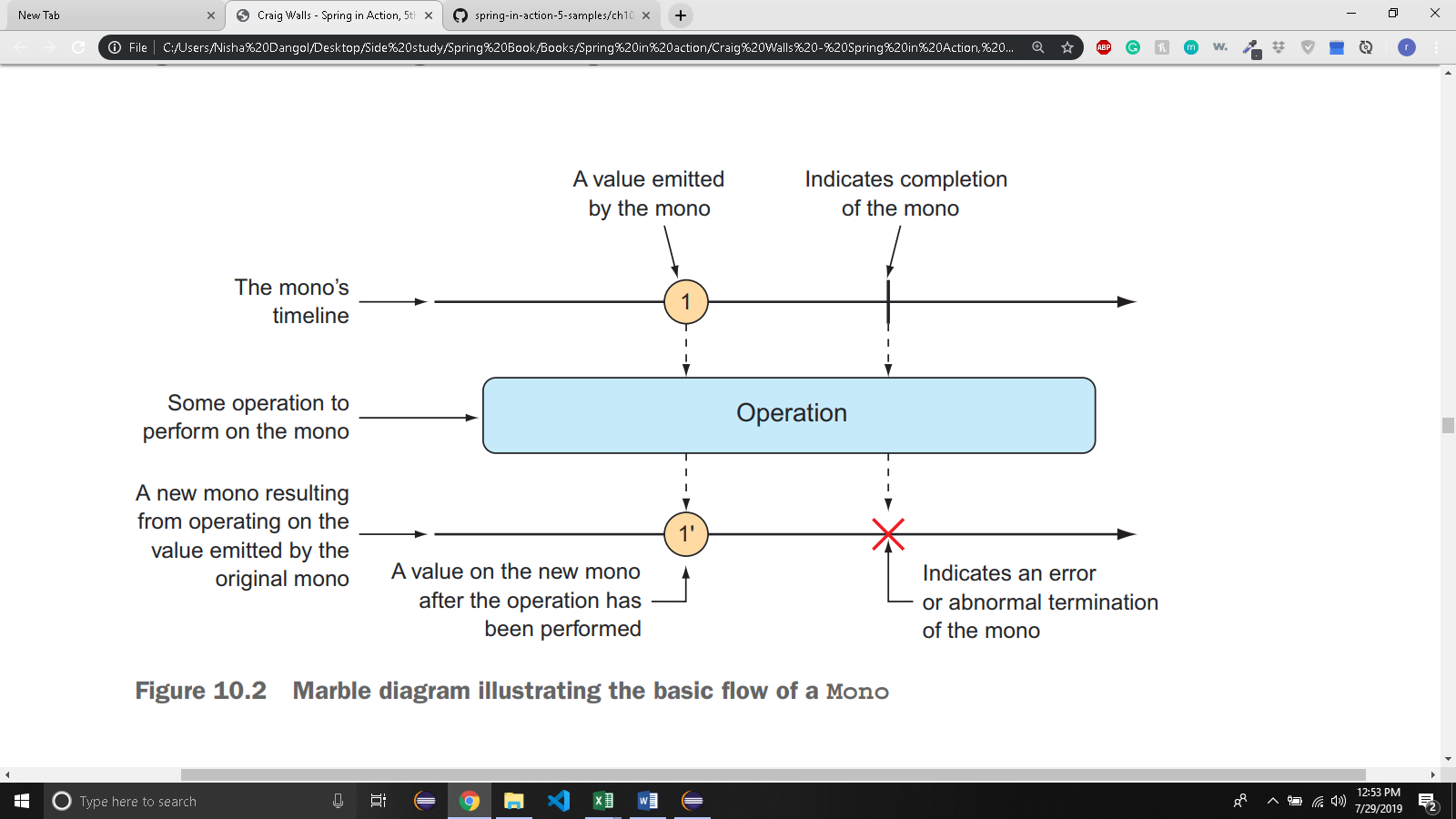
* + In contrast, functional, reactive code could achieve the same thing like this:

Mono.just("Craig")

.map(n -> n.toUpperCase())

.map(cn -> "Hello, " + cn + "!")

.subscribe(System.out::println);

* + A **Flux** represents a pipeline of zero, one, or many (potentially infinite) data items.
  + A **Mono** is a specialized reactive type that’s optimized for when the dataset is known to have no more than one data item.
  + There are actually **three Monos in the previous example**. The **just()** operation **creates the first one**. When the **Mono emits a value**, that value is **given to the map()** operation to be **capitalized and used to create another Mono**. When the **second Mono publishes its data**, it’s **given to the second map()** operation to **do some String concatenation**, the **results** of which are **used to create the third Mono**. Finally, the call to **subscribe() subscribes to the Mono, receives the data, and prints it**.
* 10.2.1 DIAGRAMMING REACTIVE FLOWS:
  + 
  + 
* 10.2.2 ADDING REACTOR DEPENDENCIES:
  + To get **started with Reactor**, add the following dependency to the project build

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-core</artifactId>

</dependency>

* + You’re going to write a lot of **tests** around your Reactor code, so you’ll definitely want to add this dependency to your build:

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-test</artifactId>

<scope>test</scope>

</dependency>

* 10.3 APPLYING COMMON REACTIVE OPERATIONS:
  + Most of the examples we’ll work with will involve Flux. Just know that Mono often has equivalent operations.
* 10.3.1 CREATING REACTIVE TYPES:
  + Creating from Objects:
    - If you have one or more **objects** that you’d like to **create a Flux or Mono from**, you can **use** the **static just() method** on Flux or Mono to create a reactive type whose data is driven by those objects.

@Test

public void createAFlux\_just() {

Flux<String> fruitFlux = Flux

.just("Apple", "Orange", "Grape", "Banana", "Strawberry");

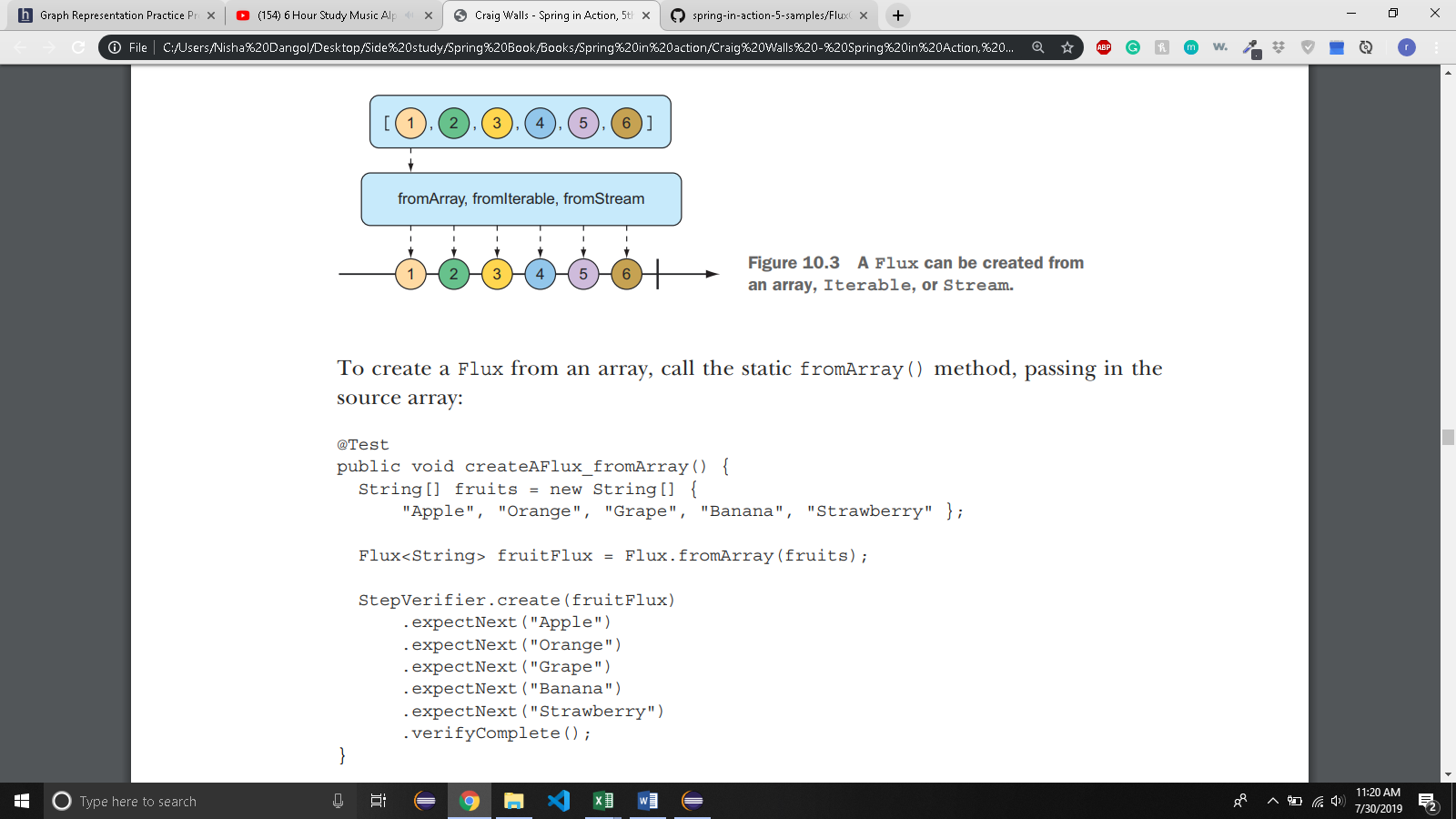
}

* + - At this point, the **Flux has been created**, but it has no subscribers. **Without** any **subscribers, data won’t flow**.

fruitFlux.subscribe(

f -> System.out.println("Here's some fruit: " + f)

);

* + - Upon **calling subscribe(), the data starts flowing**. In this example, there are no intermediate operations, so the **data flows directly from the Flux to the Subscriber.**
    - **better way to** actually **test a Flux or a Mono** is to use **Reactor’s StepVerifier**. Given a Flux or Mono, StepVerifier **subscribes to the reactive type** and then **applies assertions against the data as it flows through the stream**, finally **verifying that the stream completes as expected**.
    - StepVerifier **subscribes to the Flux** and then **asserts that each item matches the expected fruit name**. Finally, it **verifies** that **after Strawberry is produced** by the Flux, the **Flux is complete**
  + Creating from collections:
    - A Flux can also be **created from an array, Iterable, or Java Stream**.
* 
  + - If you need to **create a Flux from** a **java.util.List**, **java.util.Set**, or any other implementation of **java.lang.Iterable**, you can pass it into the static **fromIterable() method**:

@Test

public void createAFlux\_fromIterable() {

List<String> fruitList = new ArrayList<>();

fruitList.add("Apple");

fruitList.add("Orange");

fruitList.add("Grape");

fruitList.add("Banana");

fruitList.add("Strawberry");

Flux<String> fruitFlux = Flux.fromIterable(fruitList);

// ... verify steps

}

* + - if you happen to have a **Java Stream** that you’d like to use as the source for a Flux, **fromStream()** is the method you’ll use:

@Test

public void createAFlux\_fromStream() {

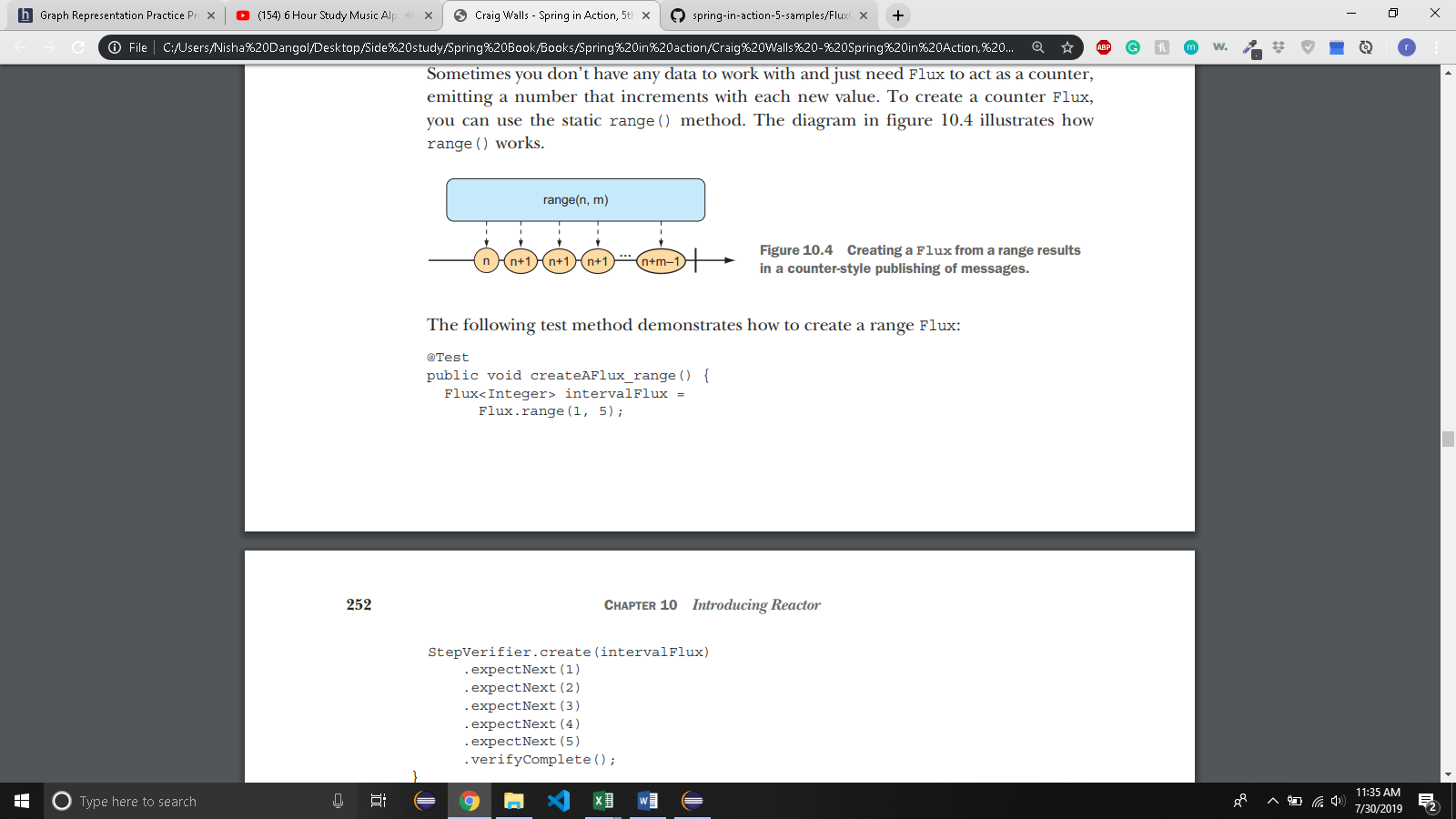
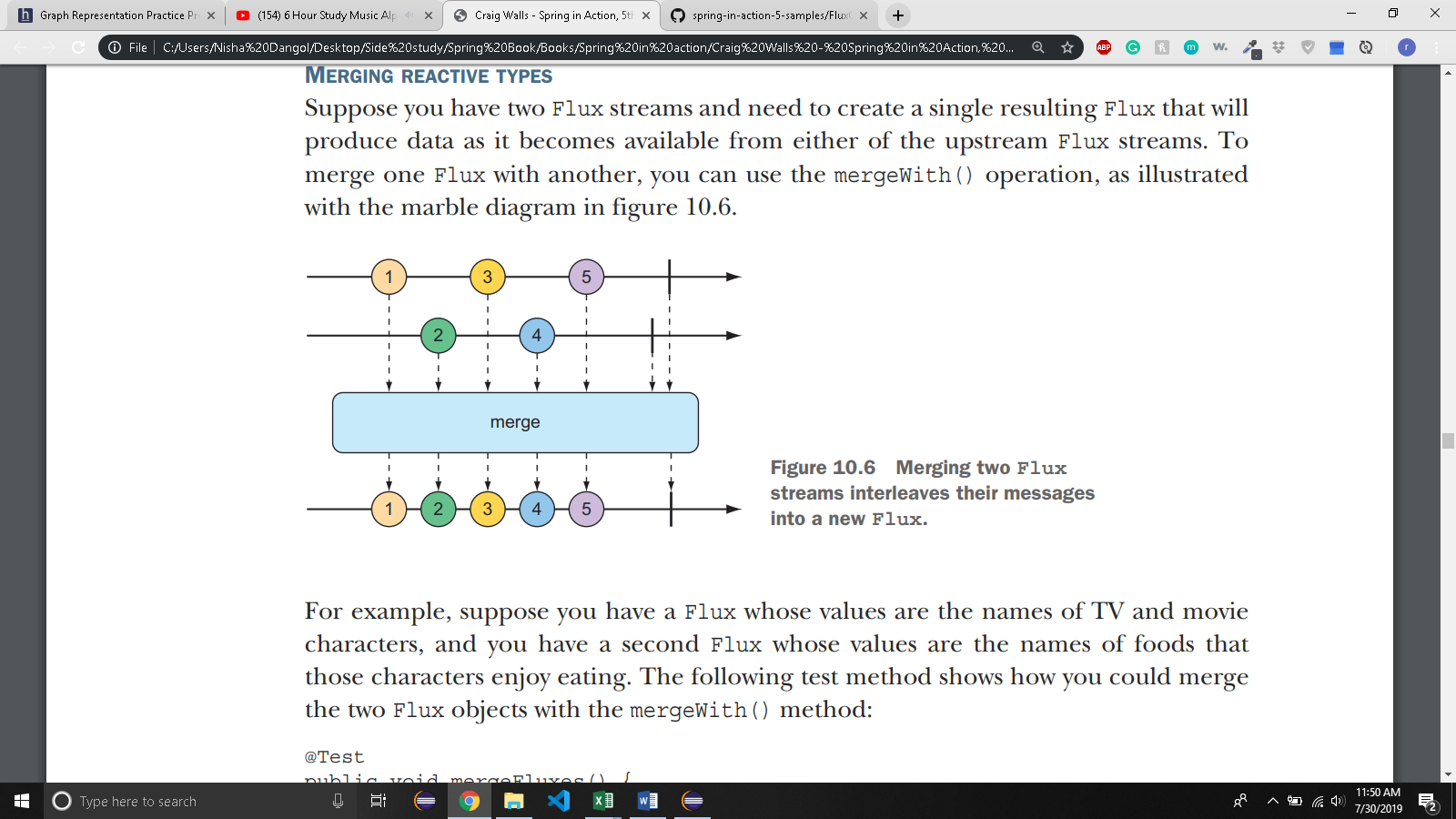
Stream<String> fruitStream =

Stream.of("Apple", "Orange", "Grape", "Banana", "Strawberry");

Flux<String> fruitFlux = Flux.fromStream(fruitStream);

// ... verify steps

}

* + GENERATING FLUX DATA
* 
* 
  + - Notice that the value emitted by an interval Flux **starts with 0 and increments on each successive item**. Also, because interval() isn’t given a maximum value, it will potentially **run forever**. Therefore, you also **use the take() operation to limit the results to the first five entries.**
* 10.3.2 COMBINING REACTIVE TYPES
  + You may find yourself with two **reactive types** that you **need to somehow merge together**. Or, in other cases, you may need to **split a Flux into more than one reactive type**. In this section, we’ll **examine operations that combine and split Reactor’s Flux and Mono.**
  + MERGING REACTIVE TYPES:
* 

@Test

public void mergeFluxes() {

Flux<String> characterFlux = Flux

.just("Garfield", "Kojak", "Barbossa")

.delayElements(Duration.ofMillis(500));

Flux<String> foodFlux = Flux

.just("Lasagna", "Lollipops", "Apples")

.delaySubscription(Duration.ofMillis(250))

.delayElements(Duration.ofMillis(500));

Flux<String> mergedFlux = characterFlux.mergeWith(foodFlux);

StepVerifier.create(mergedFlux)

.expectNext("Garfield")

.expectNext("Lasagna")

.expectNext("Kojak")

.expectNext("Lollipops")

.expectNext("Barbossa")

.expectNext("Apples")

.verifyComplete();

}

* + - Normally, a Flux will publish data as quickly as it possibly can. Therefore, you use a **delayElements()** operation on both of the created Flux streams to **slow them down a little**—only **emitting an entry every 500 ms**. Furthermore, so that the **food Flux starts streaming after the character Flux**, you apply a **delaySubscription()** operation to the **food Flux** so that it **won’t emit any data until 250 ms have passed following a subscription**
    - When **StepVerifier** subscribes to the merged Flux, it will, in turn, **subscribe** to the **two source Flux streams**, starting the flow of data.
    - Because **mergeWith() can’t guarantee a perfect back and forth** between its sources, you may want to **consider the zip()** operation instead.
    - When **two Flux objects are zipped together**, it results in a new Flux that **produces a tuple of items**, where the tuple **contains one item from each source Flux**
* 

*@Test*

public void zipFluxes() {

Flux<String> characterFlux = Flux.*just*("Rohan","Shirish","Sunesh");

Flux<String> foodFlux = Flux.*just*("Apple","Orange","Grape");

Flux<Tuple2<String,String>> zippedFlux = Flux.*zip*(characterFlux, foodFlux);

StepVerifier.*create*(zippedFlux)

.expectNextMatches(p->

p.getT1().equals("Rohan")&&

p.getT2().equals("Apple")

)

.expectNextMatches(p->

p.getT1().equals("Shirish")&&

p.getT2().equals("Orange")

)

.expectNextMatches(p->

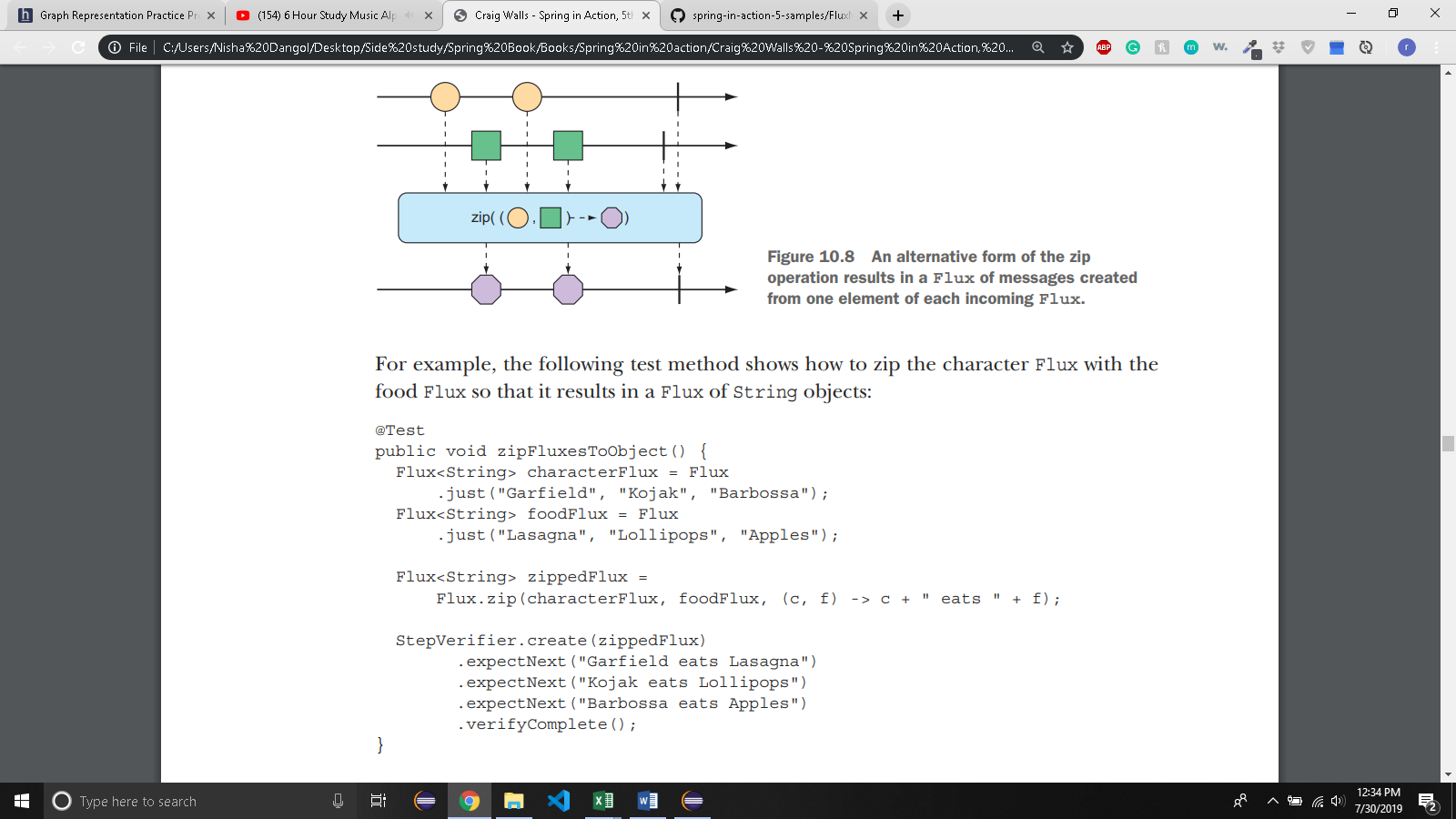
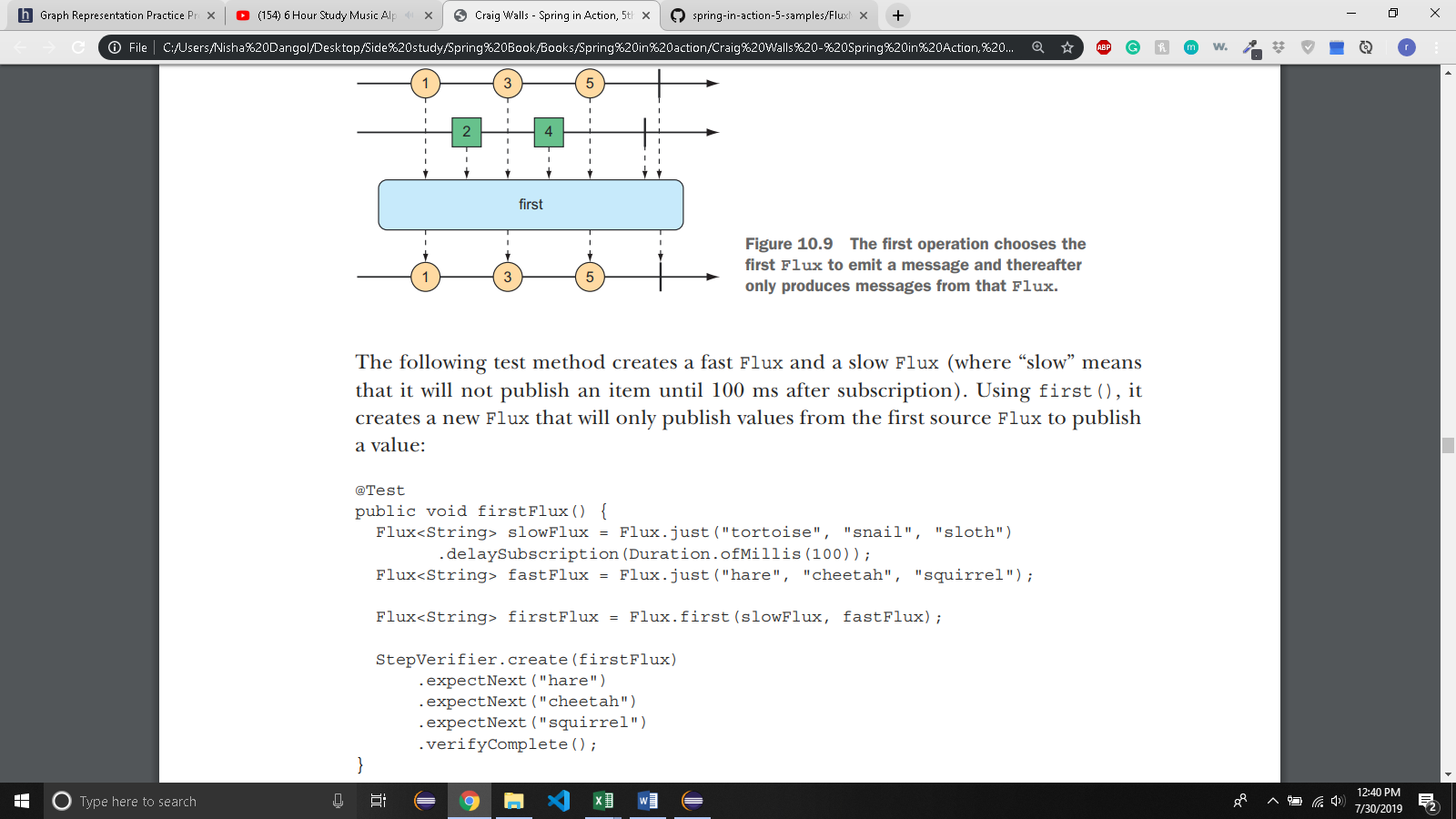
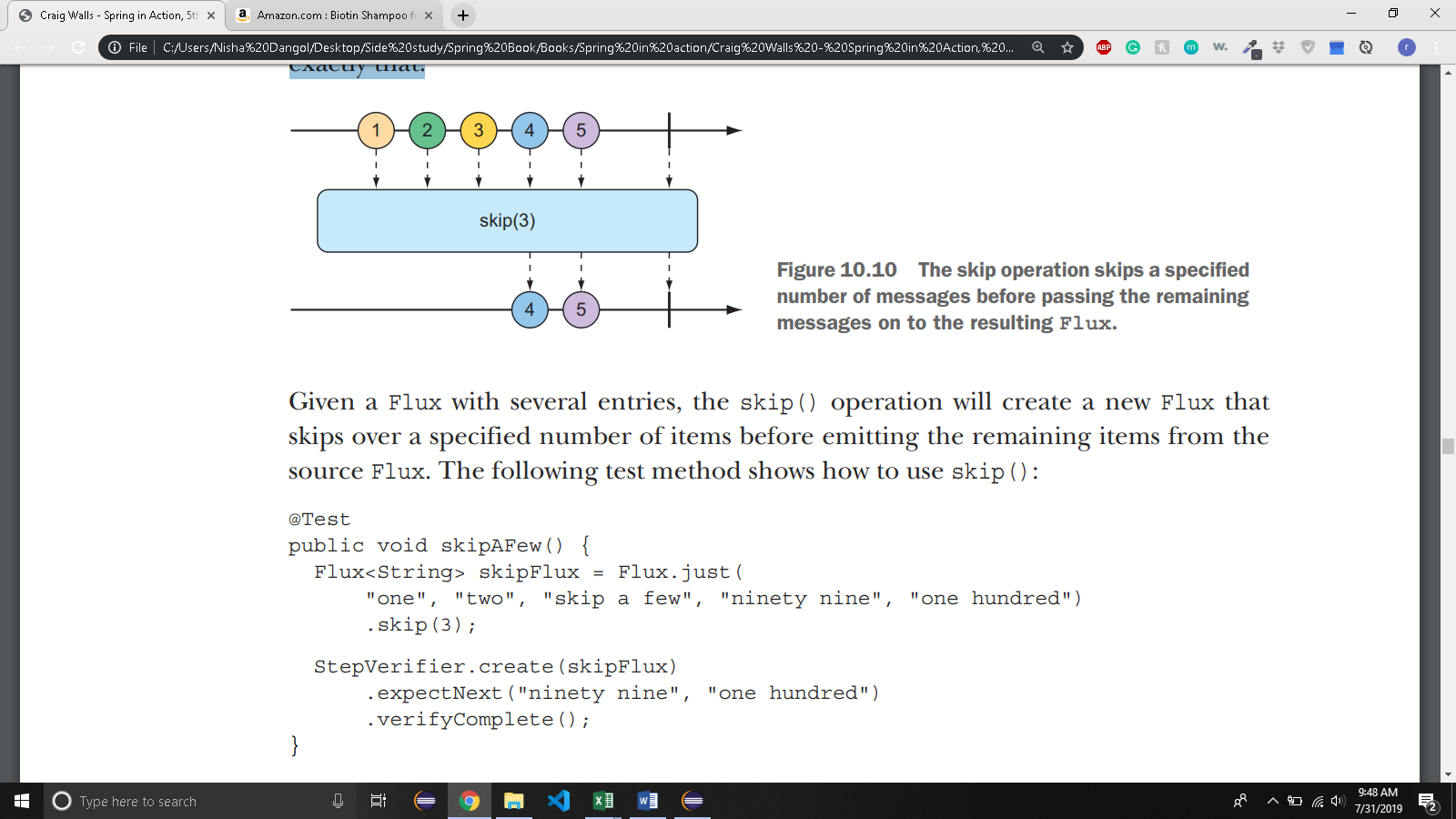
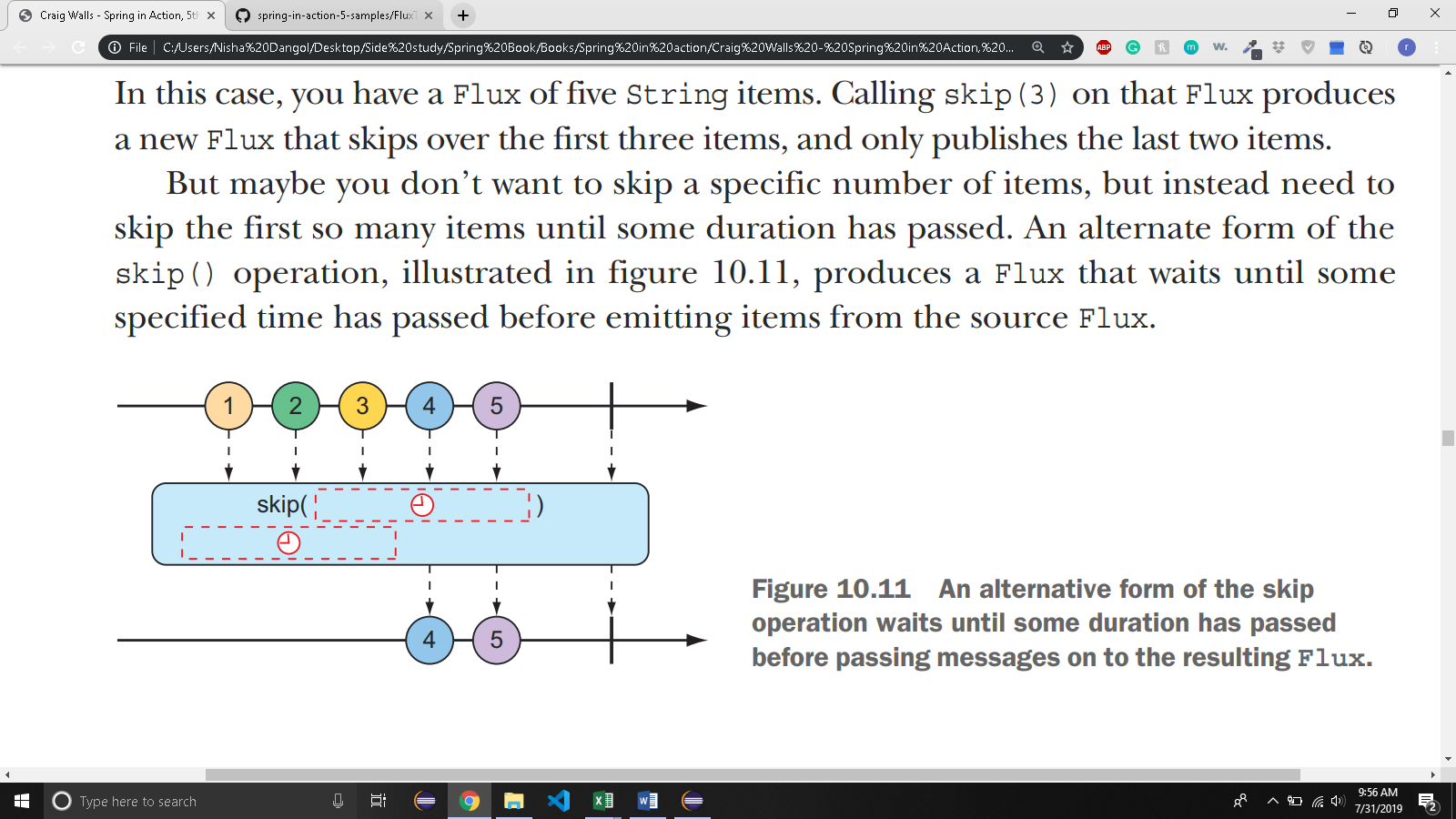
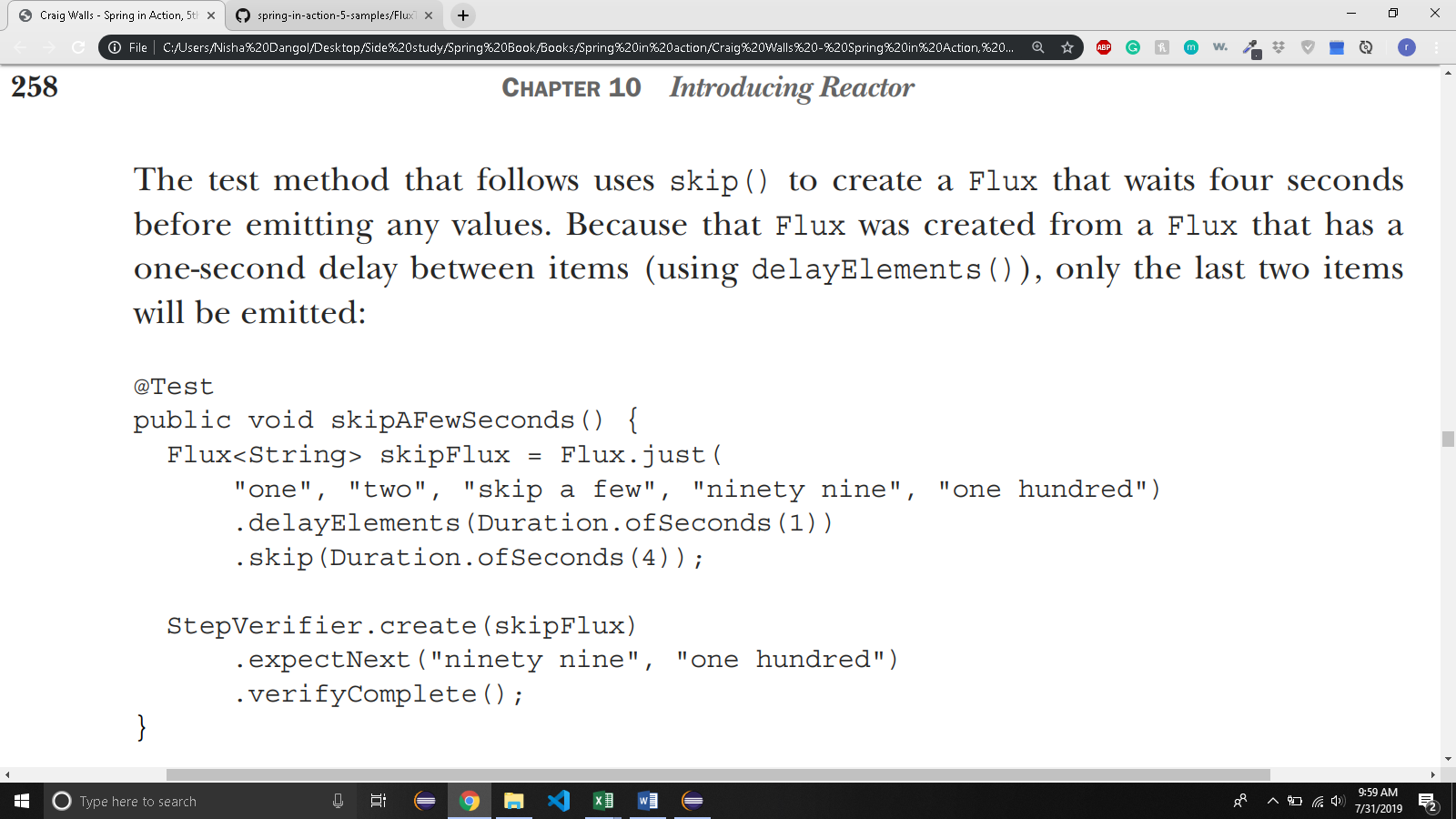
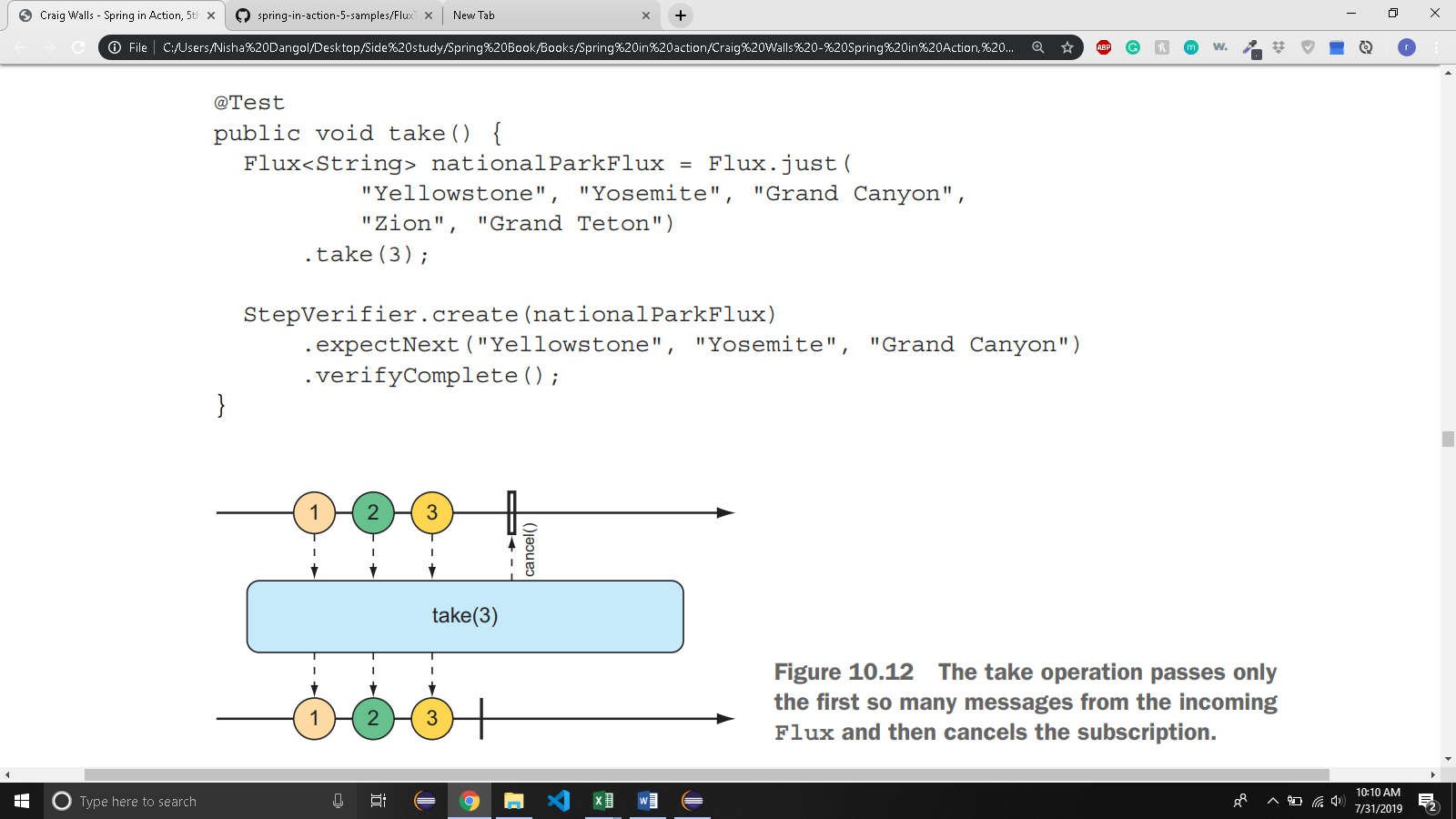
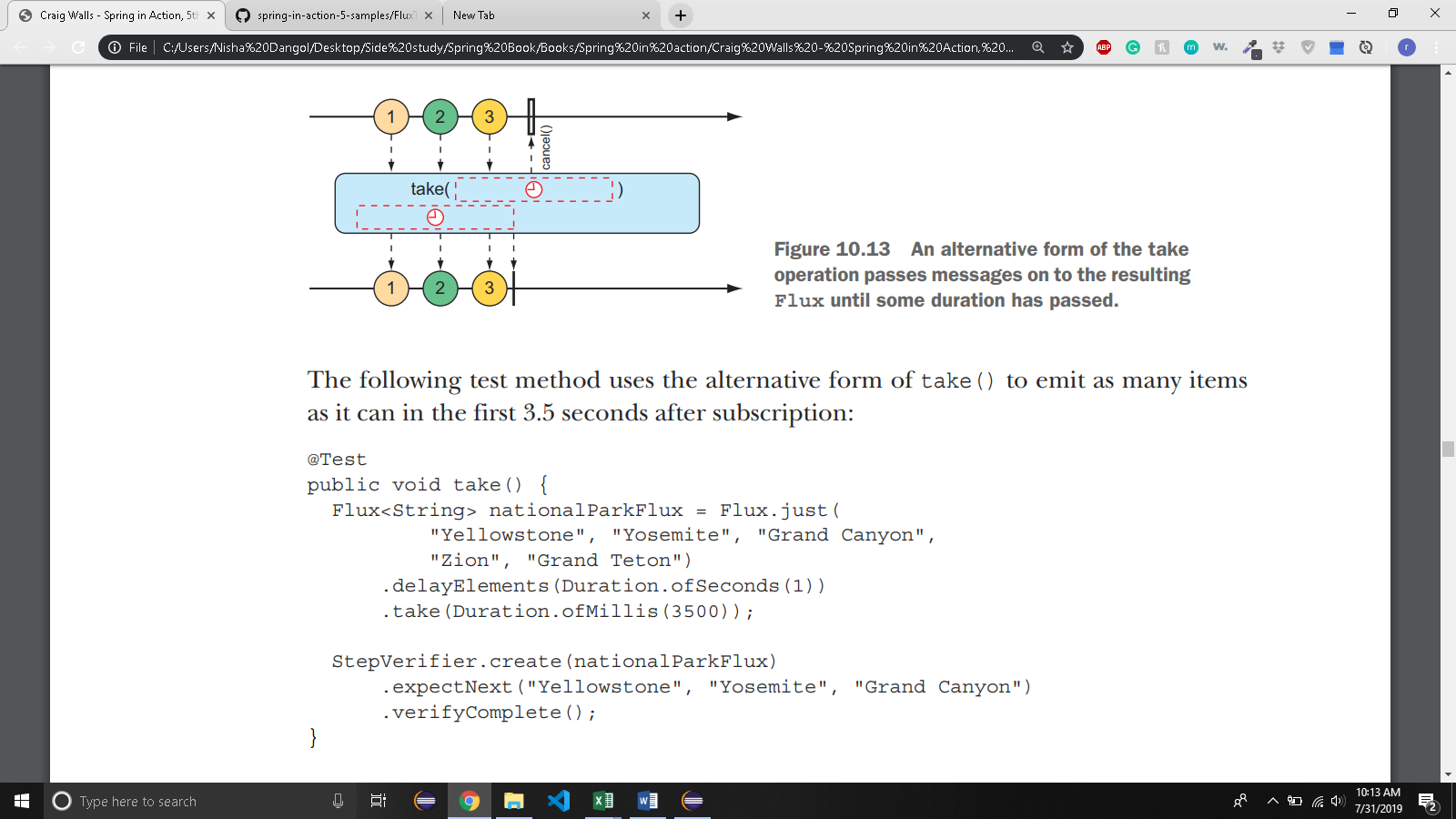
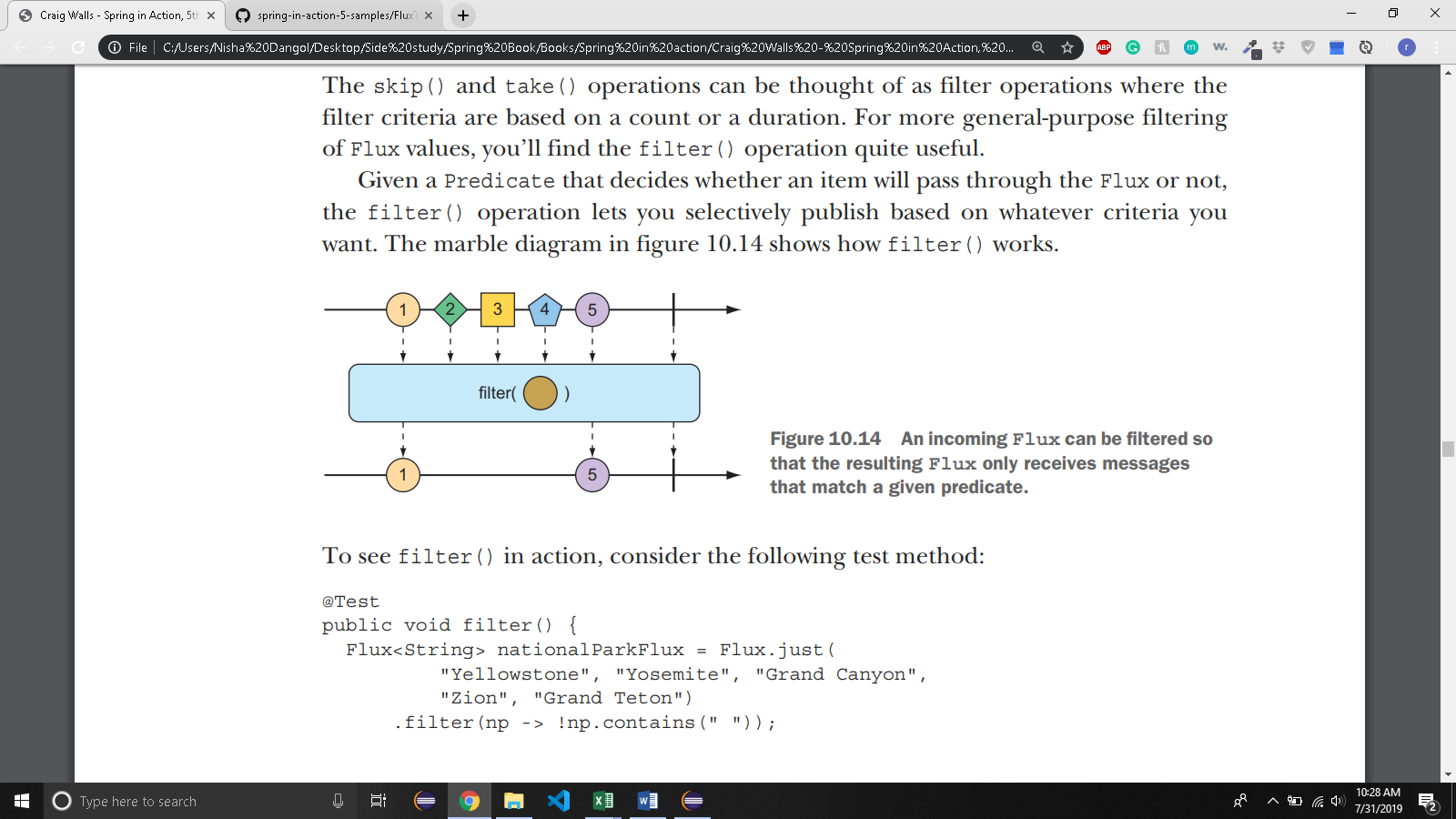
p.getT1().equals("Sunesh")&&

p.getT2().equals("Grape")

)

.verifyComplete();

}

* + Each item emitted from the zipped Flux is a **Tuple2 (a container object that carries two other objects)** containing items **from each source Flux**, in the order that they’re published.
  + you can **provide a Function to zip()** that **produces any object you’d like**
* 
  + SELECTING THE FIRST REACTIVE TYPE TO PUBLISH
    - Suppose you have **two Flux objects**, and rather than merge them together, you merely want to **create a new Flux** that emits the values **from the first Flux that produces a value**.
* 
  + - In this case, because the **slow Flux won’t publish any values until 100 ms after the fast Flux has started publishing**, the newly created Flux will simply **ignore the slow Flux and only publish values from the fast Flux.**
* 10.3.3 TRANSFORMING AND FILTERING REACTIVE STREAMS
  + Filtering Data from the Reactive types:
    - One of the most basic **ways of filtering data as it flows from a Flux** is to simply **disregard the first so many entries**. The **skip()** operation, illustrated in figure 10.10, **does exactly that**.
* 
* 
* 
  + - Whereas **skip() skips the first few items**, **take() only emits the first so many items**
* 
* 
  + - the **filter()** operation lets you **selectively publish based on whatever criteria you want**.
* 

*@Test*

public void filter() {

Flux<String> filterFlux = Flux.*just*("Yellowstone", "Yosemite", "Grand Canyon",

"Zion", "Grand Teton")

.filter(n->!n.contains(" "));

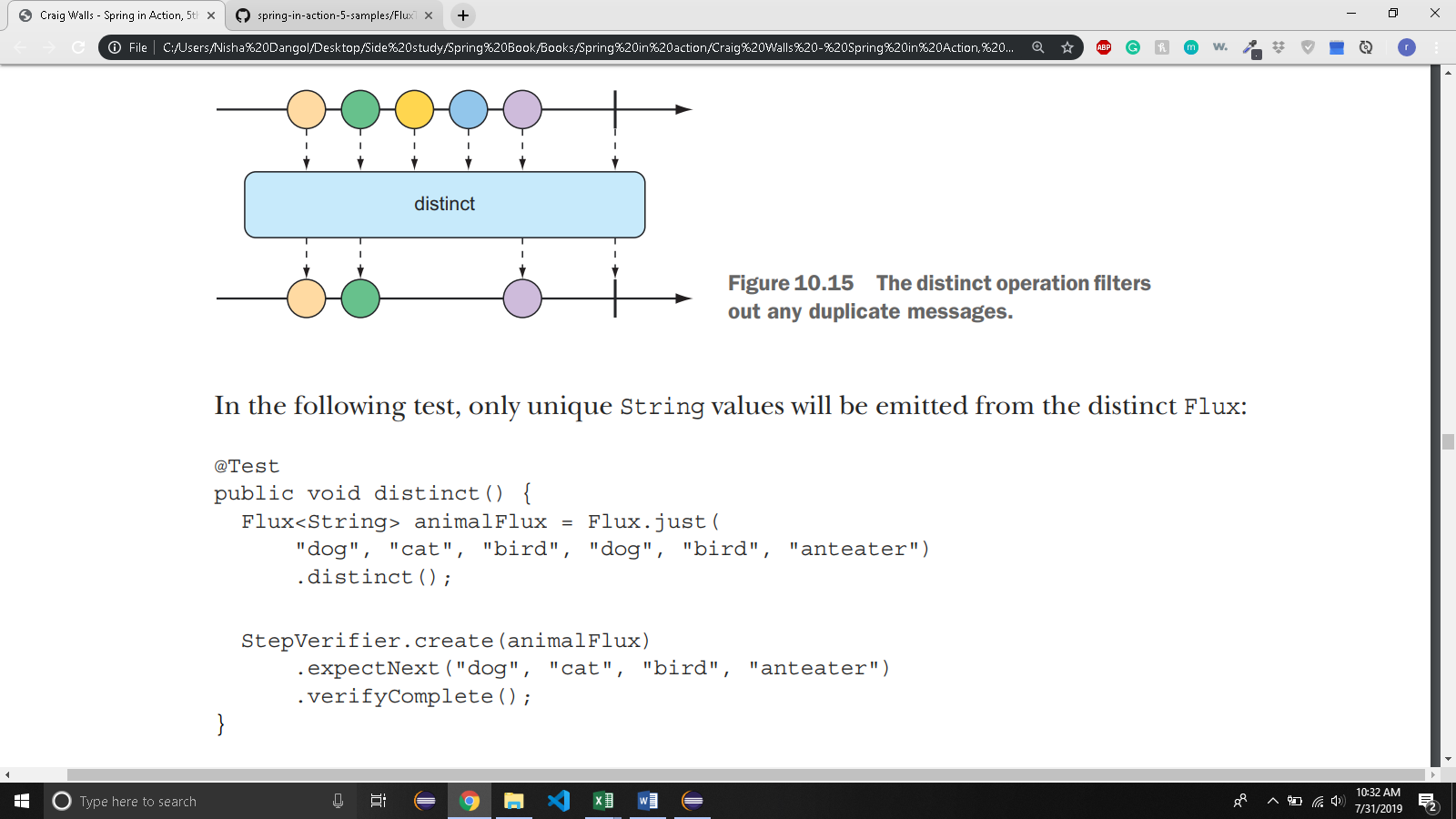
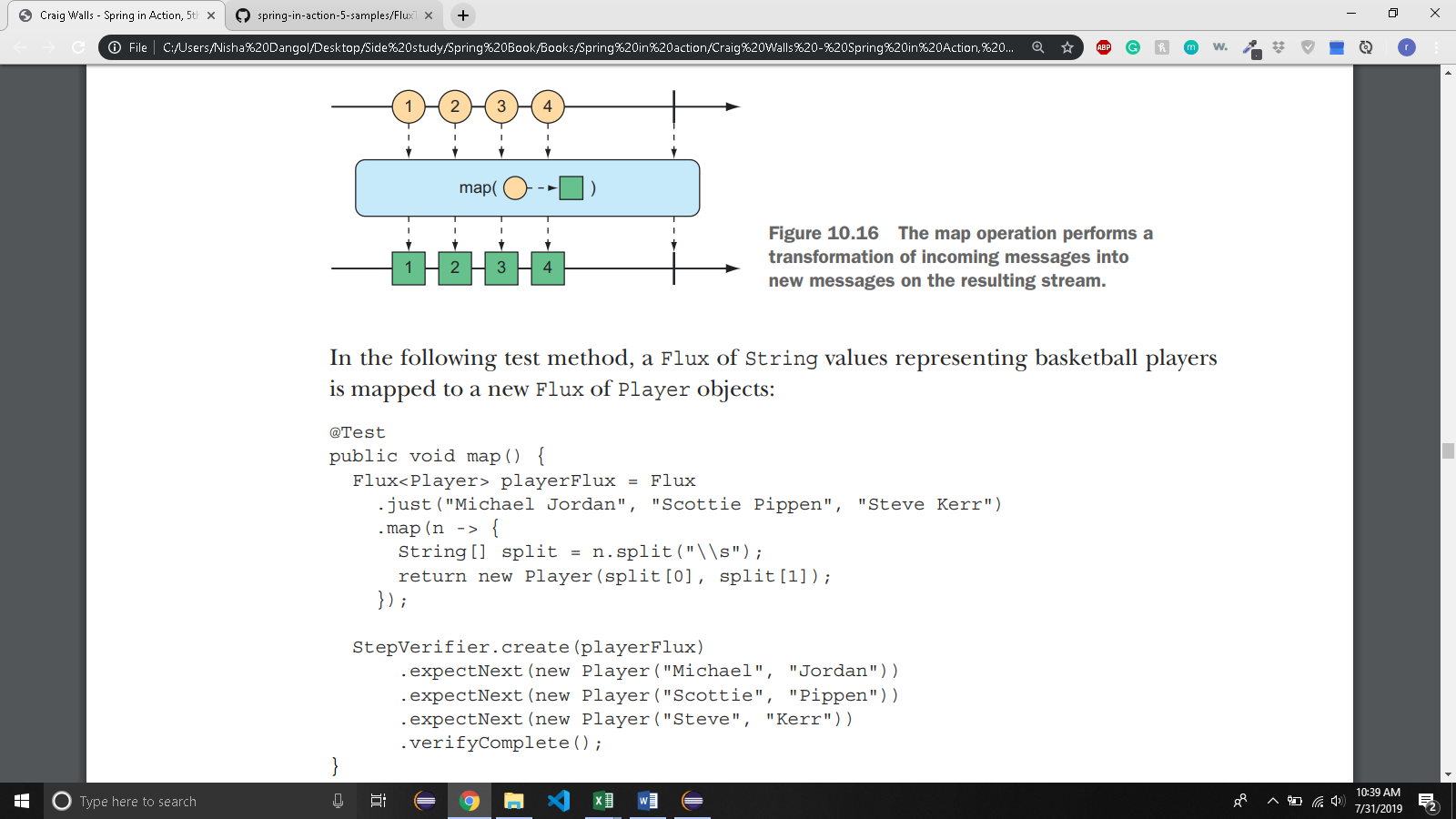
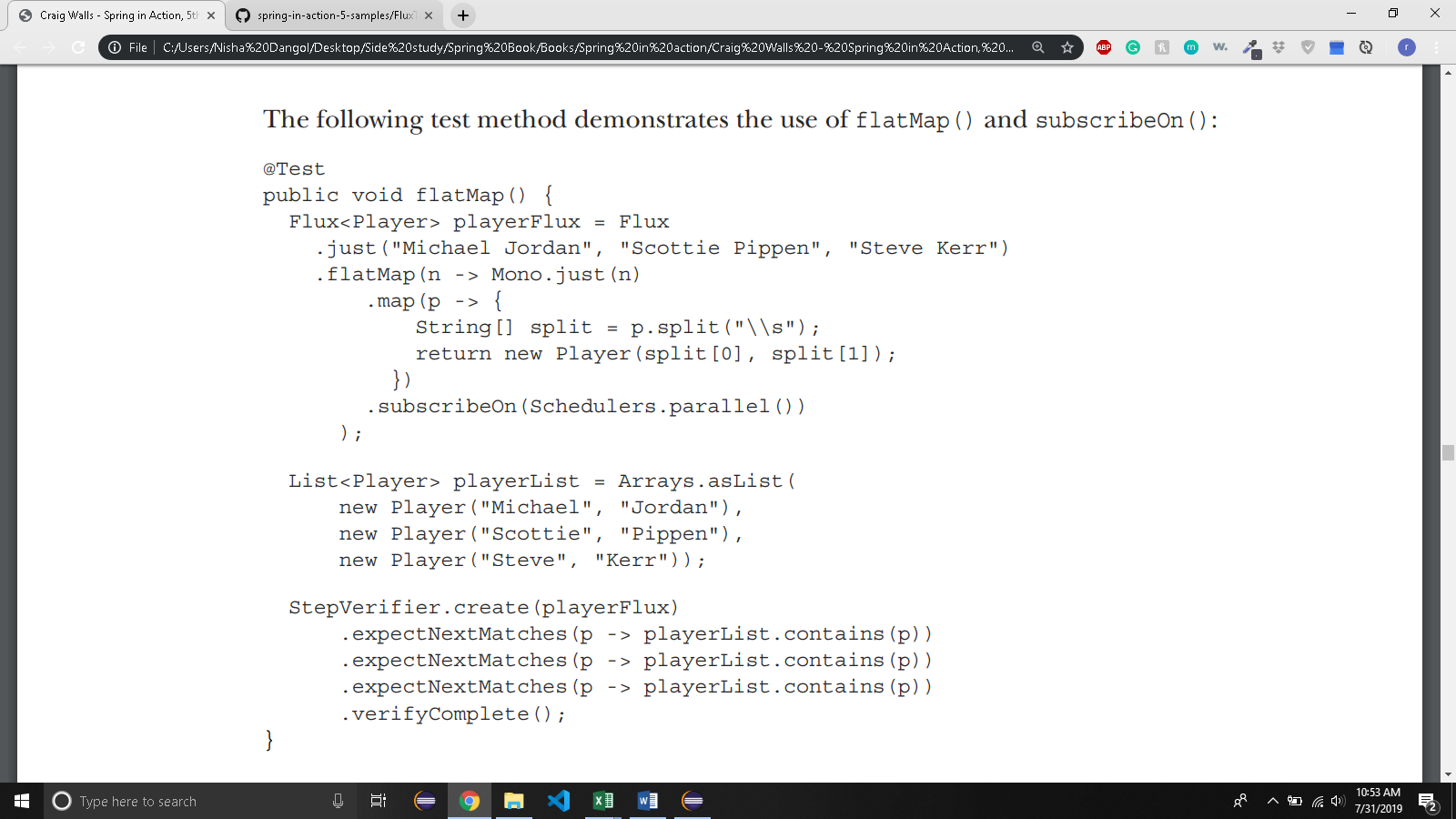
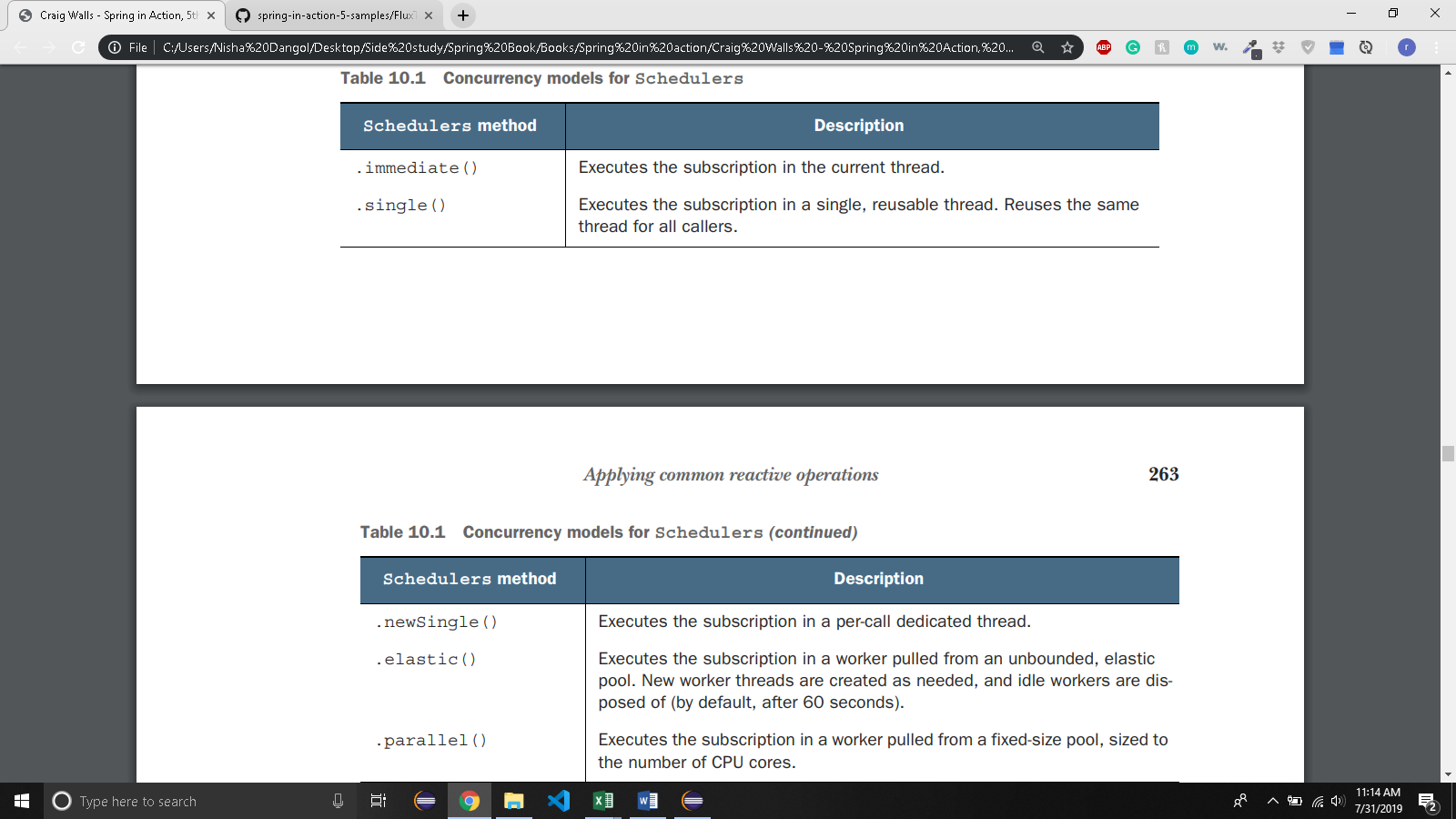
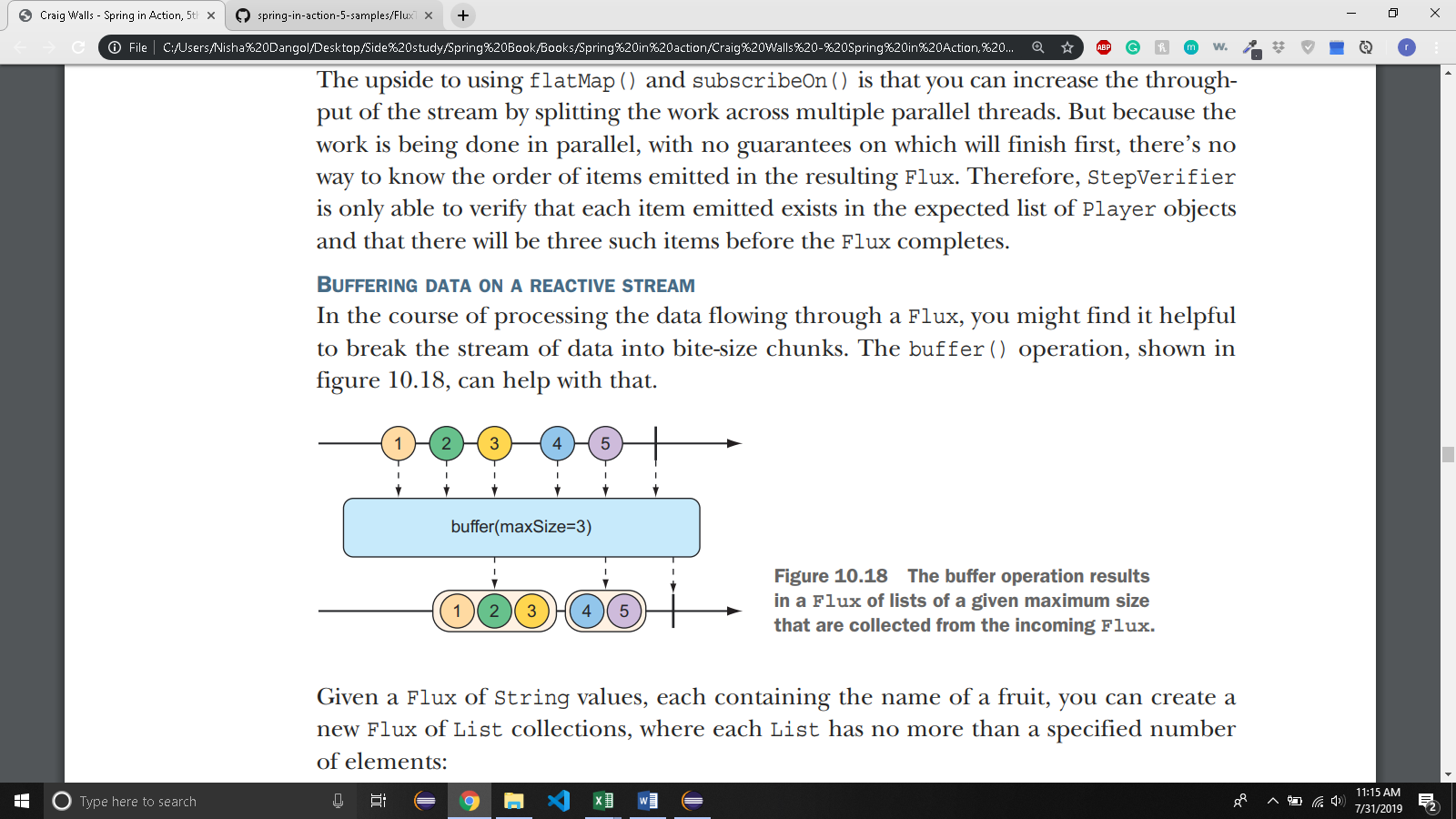
StepVerifier.*create*(filterFlux)

.expectNext("Yellowstone")

.expectNext("Yosemite")

.expectNext("Zion")

.verifyComplete();

* + - }
    - The **distinct()** operation, as illustrated in figure 10.15, results in a Flux that only **publishes items from the source Flux that haven’t already been published**.
* 
  + **MAPPING REACTIVE DATA:**
    - The **map()** operation **creates a Flux** **that** simply **performs a transformation as prescribed by a given Function on each object it receives before republishing it**.
* 
* 
* 
  + - **flatMap()** is given a lambda Function that **transforms the incoming String into a Mono of type String**. A **map()** operation is then **applied to the Mono** to **transform the String to a Player**.
    - the **resulting Flux** would **carry Player objects**, produced **synchronously in the same order as with the map()** example.
    - call **subscribeOn()** to indicate that **each subscription should take place in a parallel thread.**
    - the **mapping operations for multiple incoming String objects** can be performed **asynchronously and in parallel**.
    - it’s through **subscribeOn()** that you can **specify the concurrency model**, using one of the static methods from **Schedulers**, that you want to use. In this example, you used **parallel()**, which **uses worker threads from a fixed pool** (sized to the number of CPU cores)
* 
* 

**@Test**

**public void buffer() {**

**Flux<String> fruitFlux = Flux.just(**

**"apple", "orange", "banana", "kiwi", "strawberry");**

**Flux<List<String>> bufferedFlux = fruitFlux.buffer(3);**

**StepVerifier**

**.create(bufferedFlux)**

**.expectNext(Arrays.asList("apple", "orange", "banana"))**

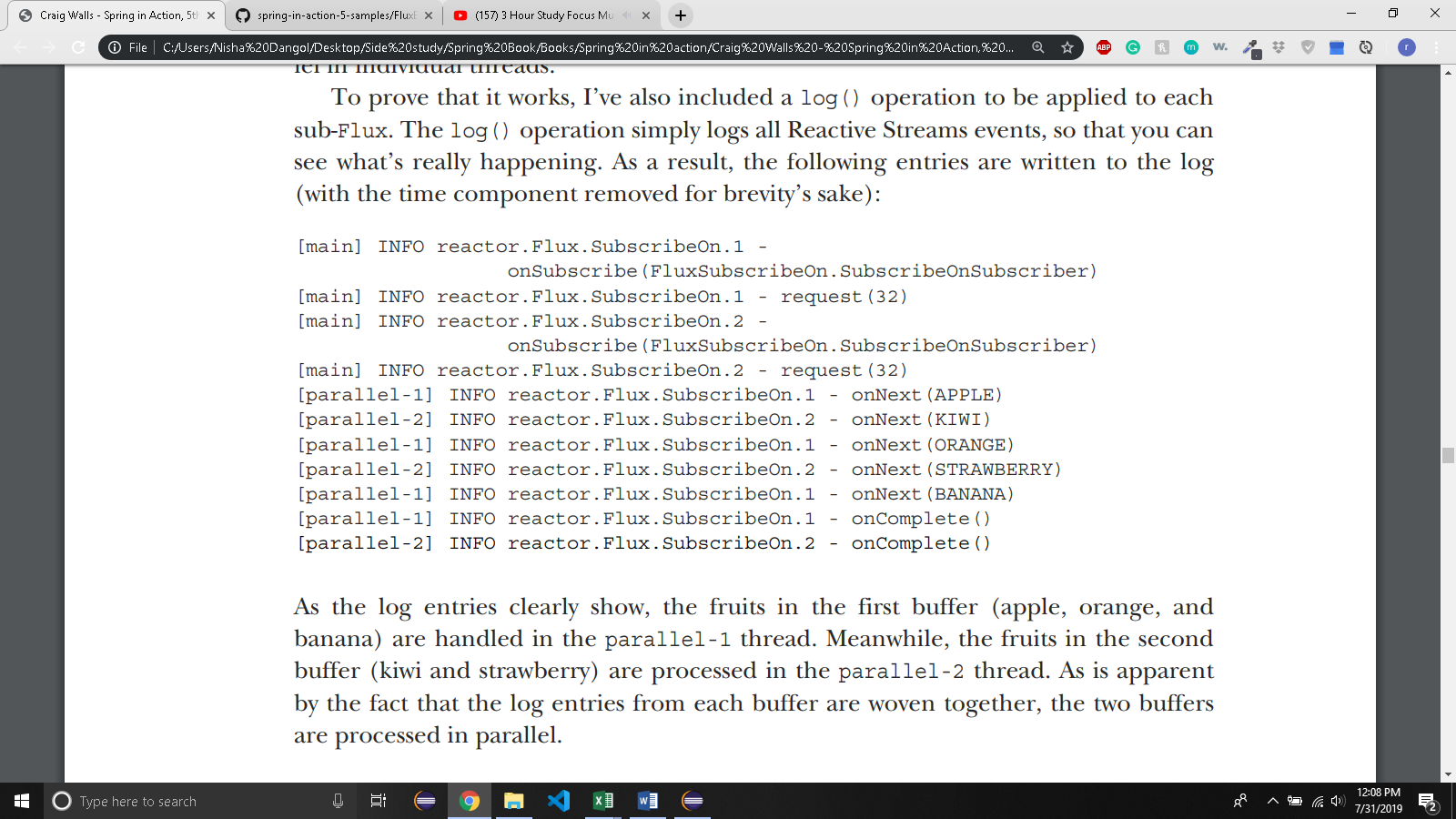
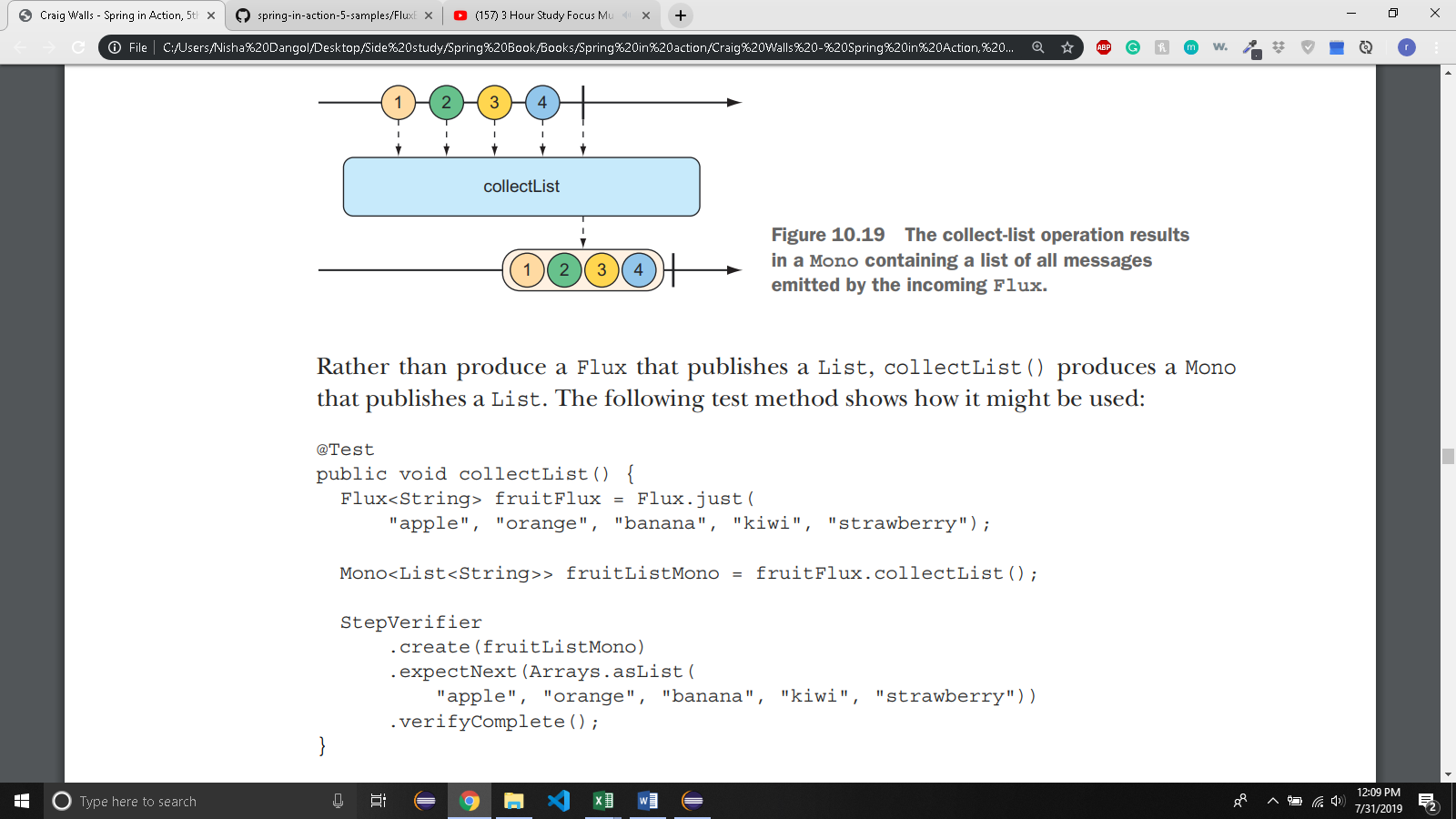
**.expectNext(Arrays.asList("kiwi", "strawberry"))**

**.verifyComplete();**

**}**

* + the **original Flux that emits five String values** will be **converted to a Flux that emits two List collections**, **one containing three fruits and the other with two fruits.**
  + when you **combine buffer() with flatMap()**, it enables **each of the List collections** to be **processed in parallel**:

|  |
| --- |
|  |
| @Test |
|  | public void bufferAndFlatMap() throws Exception { |
|  | Flux.just( |
|  | "apple", "orange", "banana", "kiwi", "strawberry") |
|  | .buffer(3) |
|  | .flatMap(x -> |
|  | Flux.fromIterable(x) |
|  | .map(y -> y.toUpperCase()) |
|  | .subscribeOn(Schedulers.parallel()) |
|  | .log() |
|  | ).subscribe(); |
|  | } |
|  |  |

* + you apply **flatMap() to** that **Flux of List collections**.
  + This **takes each List buffer** and **creates a new Flux from its elements**, and then **applies a map() operation** on it.
  + each **buffered List** is further **processed in parallel in individual threads**.
  + The **log()** operation simply **logs all Reactive Streams events,** so that you can see what’s really happening.
* 
  + If, for some reason, you need to **collect everything that a Flux emits into a List**, you can call buffer() with no arguments:
    - Flux<List> bufferedFlux = fruitFlux.buffer();
  + You can achieve the same thing with the **collectList()** operation,
* 
* 

**@Test**

**public void collectMap() {**

**Flux<String> animalFlux = Flux.just(**

**"aardvark", "elephant", "koala", "eagle", "kangaroo");**

**Mono<Map<Character, String>> animalMapMono =**

**animalFlux.collectMap(a -> a.charAt(0));**

**StepVerifier**

**.create(animalMapMono)**

**.expectNextMatches(map -> {**

**return**

**map.size() == 3 &&**

**map.get('a').equals("aardvark") &&**

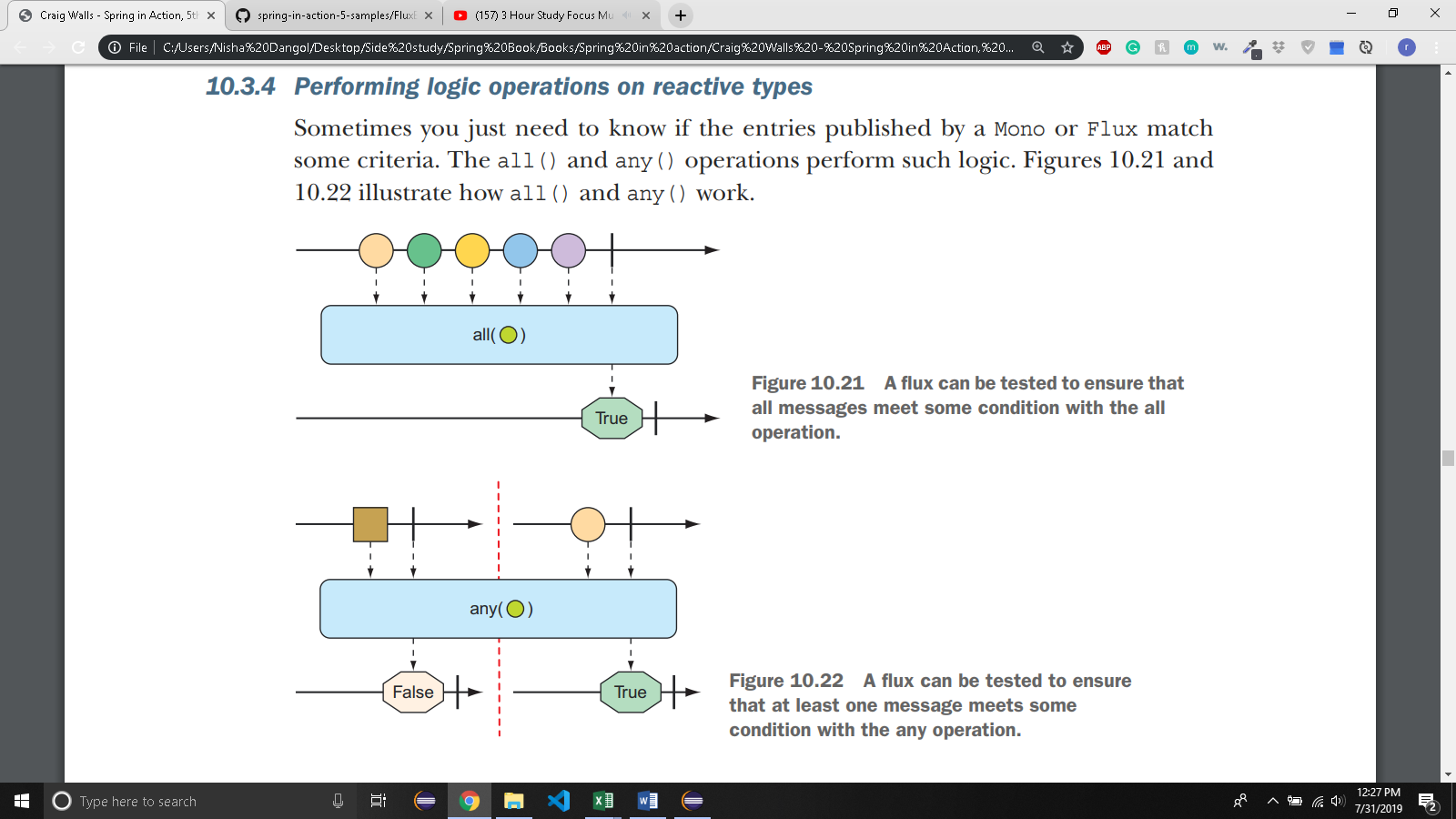
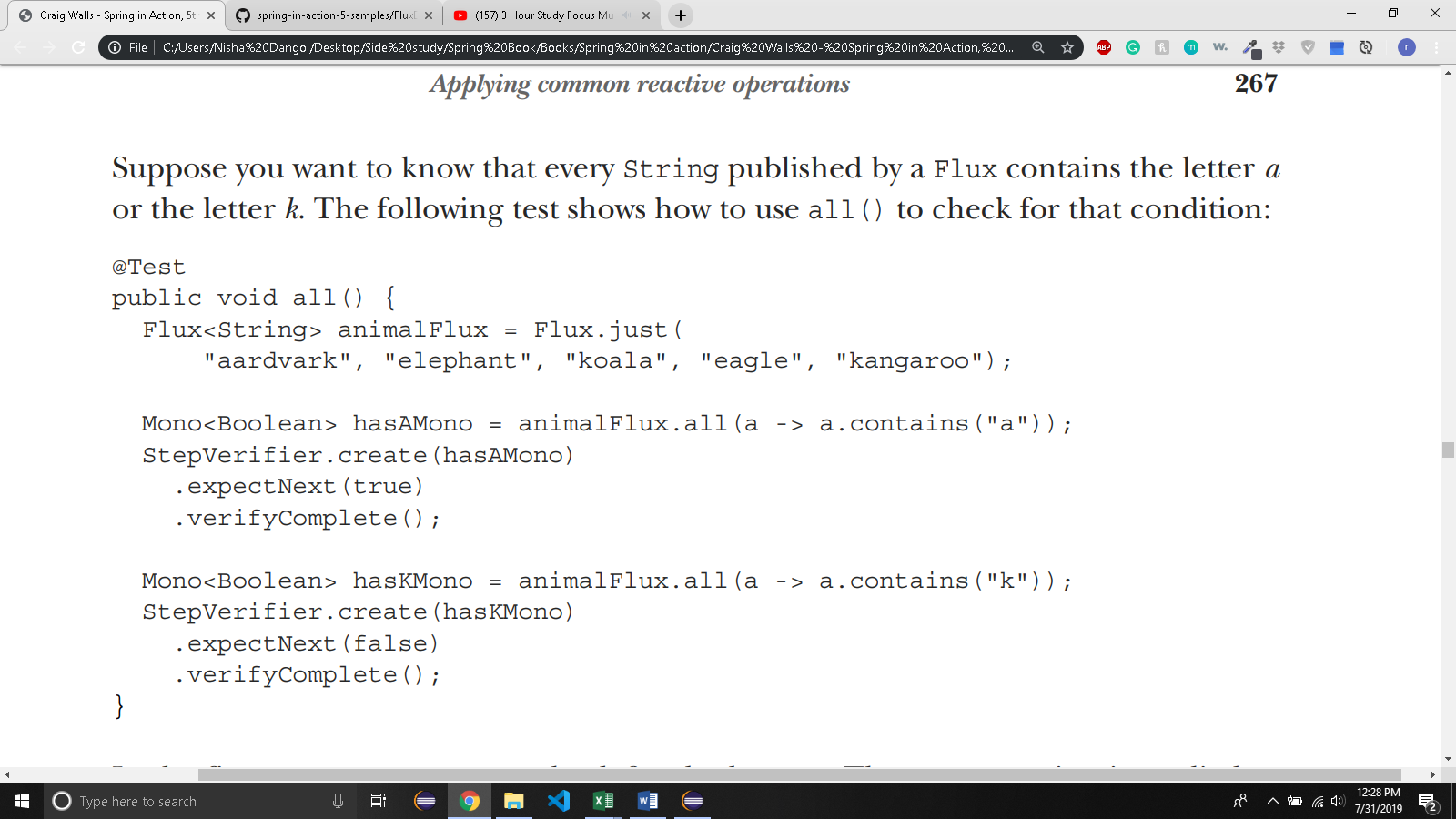
**map.get('e').equals("eagle") &&**

**map.get('k').equals("kangaroo");**

**})**

**.verifyComplete();**

**}**

* + The **source Flux emits** the **names of a handful of animals**. From that Flux, you **use collectMap() to create a new Mono that emits a Map**, where the **key value** is determined by the **first letter of the animal name** and the **value** is the **animal name itself**. In the event that **two animal names start with the same letter** (as with elephant and eagle or koala and kangaroo), the **last entry** flowing through the stream **overrides any earlier entries**.
* 10.3.4 PERFORMING LOGIC OPERATIONS ON REACTIVE TYPES:
* 
* 
* 