CHAPTER 9 – INTEGRATING SPRING

* 9.1 DECLARING A SIMPLE INTEGRATION FLOW:
  + Spring Integration allows to create Integration flow through which your application can send or receive data to external sources.
  + Add spring integration in POM files:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-integration</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.integration</**groupId**>

<**artifactId**>spring-integration-file</**artifactId**>

</**dependency**>

* + - First dependency is Spring Integration. It is a must for developing Spring Integration flow no matter what the flow may integrate with.
    - Second dependency is spring Integration’s file endpoint module used to integrate with external system.
    - Lets see how your application can send data to the Integration flow so that it can write the data in the file.

@MessagingGateway(defaultRequestChannel="textInChannel")

public interface FileWriterGateway {

void writeToFile( @Header(FileHeaders.FILENAME) String filename, String data);

}

* + This is an interface.
  + @MessagingGateway tells spring to create an implementation of this interface at runtime.
  + defaultRequestChannel sets the channel through which the message will be sent when the writeToFile() method is called.
  + @Header specifies that the filename passed in the method is placed as a message Header.
  + The data parameter is carried in the message payload
* 9.1.2 CONFIGURING INTERGRATION FLOWS IN JAVA

@Configuration

public class FileWriterIntegrationConfig {

@Bean

@Transformer(inputChannel="textInChannel", outputChannel="fileWriterChannel")

public GenericTransformer upperCaseTransformer() {

return text -> text.toUpperCase();

}

@Bean

@ServiceActivator(inputChannel="fileWriterChannel")

public FileWritingMessageHandler fileWriter() {

FileWritingMessageHandler handler = new FileWritingMessageHandler(new File("/tmp/sia5/files"));

handler.setExpectReply(false);

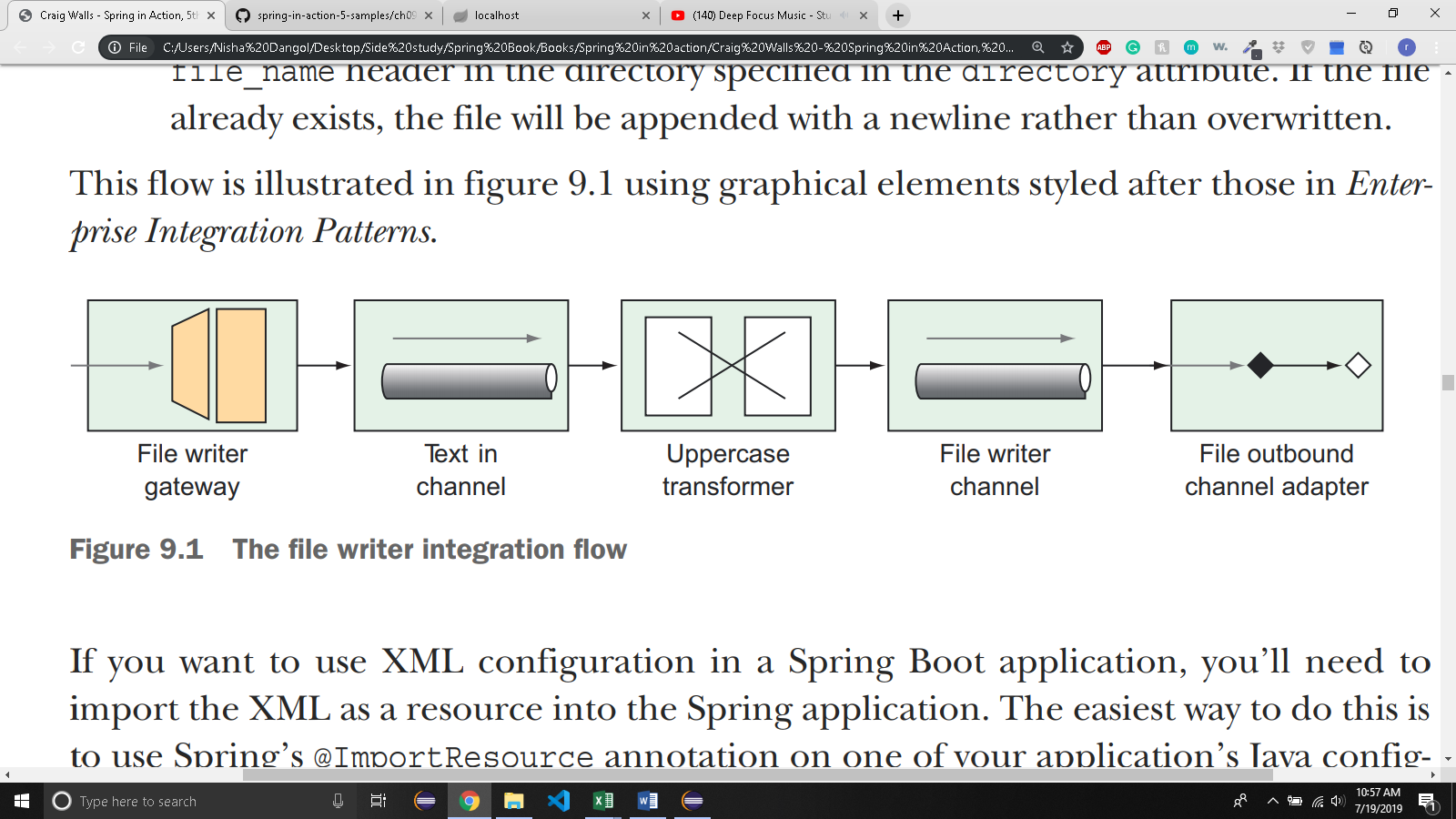
handler.setFileExistsMode(FileExistsMode.APPEND);

handler.setAppendNewLine(true);

return handler;

` }

}

* + - Generic transformer is a functional interface, you are able to provide its implementation as lambda which calls to toUpperCase() on the message text.
    - @Transformer specifies this bean as a transformer which receives message from ‘textInChannel’ and sends the transformed message to ‘fileWriterChannel’.
    - @ServiceActivator(inputChannel = ‘fileWriterChannel’ indicates that it will accept message from fileWriterChannel and hand the message to service defined by an instance of FileWritingMessageHandler.
    - FileWritingMessageHandler is message handler that writes message payload to a file in a specified directory by using filename specified in message’s file\_name header. If the file already exists, the file will be appended with a newline rather than overwritten.
    - setExpectReply(false) indicates that ServiceActivator shouldn’t expect a reply channel (a channel through which a value may be returned to upstream components in the flow).
    - You didn’t have to explicitely declare the channel. The textInChannel and fileWriterChannel will be created automatically if no beans with those names exists.
* 
* 9.1.2 USING SPRING INTEGRATION’S DSL CONFIGURATION

@Configuration

public class FileWriterIntegrationConfig {

@Bean

public IntegrationFlow fileWriterFlow() {

return IntegrationFlows

.from(MessageChannels.direct("textInChannel"))

.<String, String>transform(t -> t.toUpperCase())

.handle(Files

.outboundAdapter(new File("/tmp/sia5/files"))

.fileExistsMode(FileExistsMode.APPEND)

.appendNewLine(true))

.get();

}

}

* + The IntegrationFlows class initiates the builder API, from which you can declare the flow.
  + You start by receiving messages from textInChannel.
  + The message goes to transformer which changes the message’s payload to uppercase
  + The Files type provided by Spring Integrations file module is used to create the outbound channel adapter that handles the transformed message.
  + Finally, a call to get() builds the IntergrationFlow to be returned by the method.
  + You don’t need to explicitely declare the channels.
  + Furthermore, we didn’t even reference the FileWriterChannel that connects the transformer to the outbound channel adapter.
  + If you need to explicitly declare the channel you can do so as follows:

@Bean

public IntegrationFlow fileWriterFlow() {

return IntegrationFlows

.from(MessageChannels.direct("textInChannel"))

<String, String>transform(t -> t.toUpperCase())

.channel(MessageChannels.direct("fileWriterChannel"))

.handle(Files

.outboundAdapter(new File("/tmp/sia5/files"))

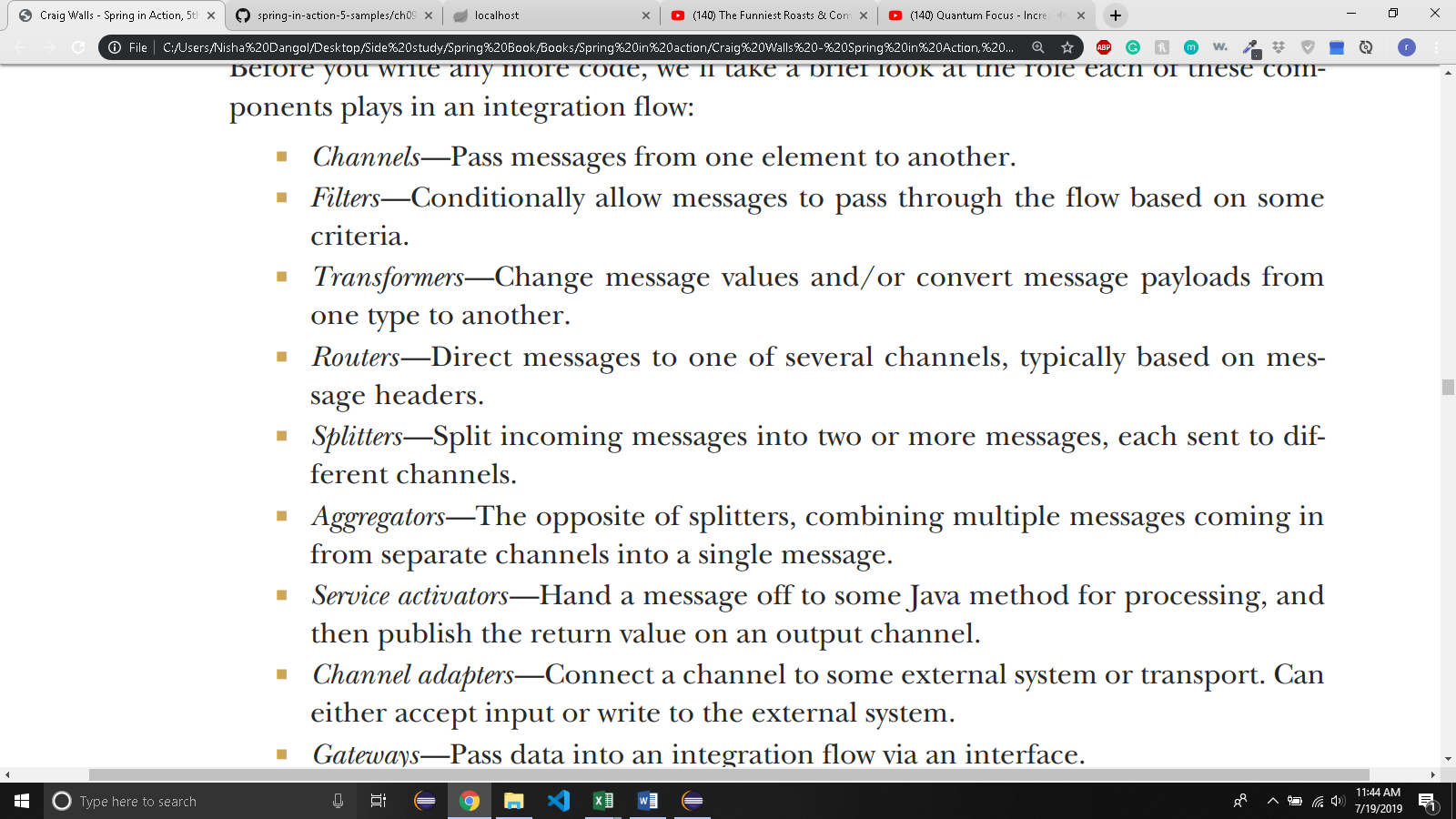
.fileExistsMode(FileExistsMode.APPEND)

.appendNewLine(true))

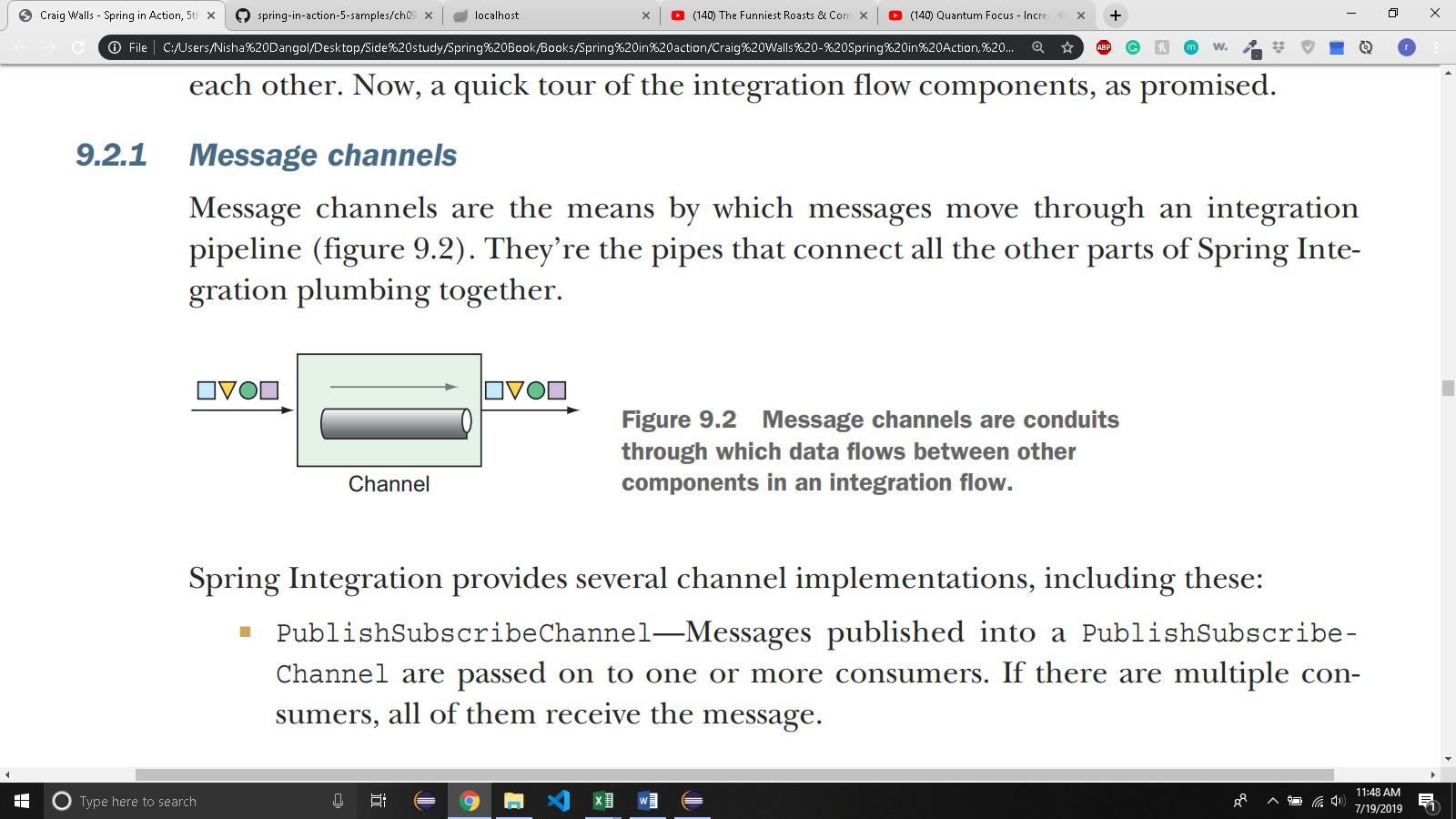
.get();

}

* 9.2 SURVEYING THE SPRING INTEGRATION LANDSCAPE:



* 9.2.1 MESSAGE CHANNELS:



* + There are several channel implementation:
  + 
  + Default input channel is DirectChannel. You can change that as follow:

@Bean

public MessageChannel orderChannel() {

return new PublishSubscribeChannel();

}

* + Then you reference this channel by name is Integraion flow. If a channel is consumed by service activator:

@ServiceActivator(inputChannel="orderChannel")

* + In Java DSL:

@Bean

public IntegrationFlow orderFlow() {

return IntegrationFlows

... .

channel("orderChannel")

...

.get();

}

* + If you are using queue channel, you must also declare pollers:

@Bean

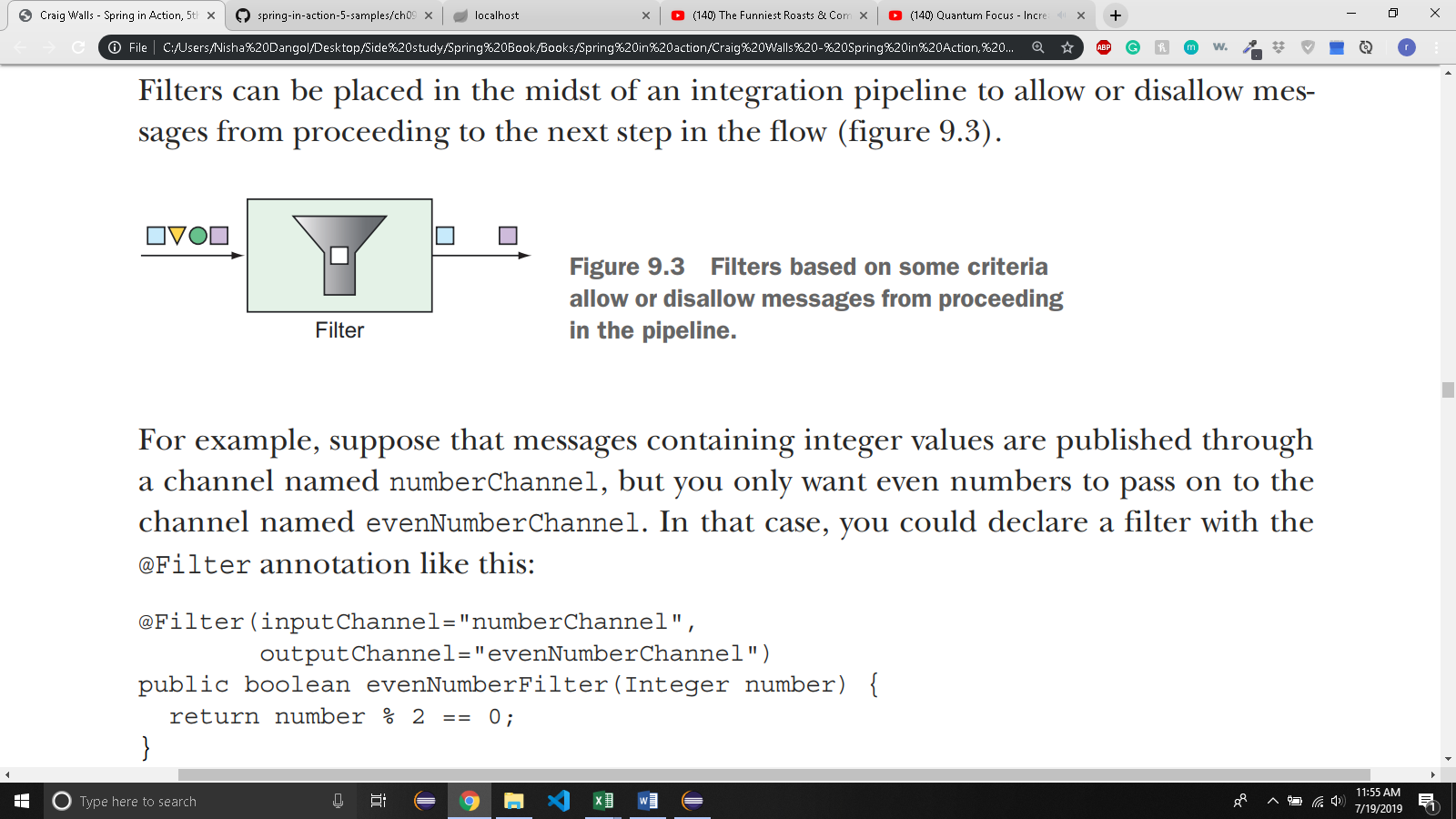
public MessageChannel orderChannel() {

return new QueueChannel();

}

@ServiceActivator(inputChannel="orderChannel", poller=@Poller(fixedRate="1000"))

* + The service activator polls from channel named ‘orderChannel’ every 1 second (1000 ms).
* 9.2.2 FILTERS:



* If you’re using Java DSL, you can do the following:

@Bean

public IntegrationFlow evenNumberFlow(AtomicInteger integerSource) {

return IntegrationFlows

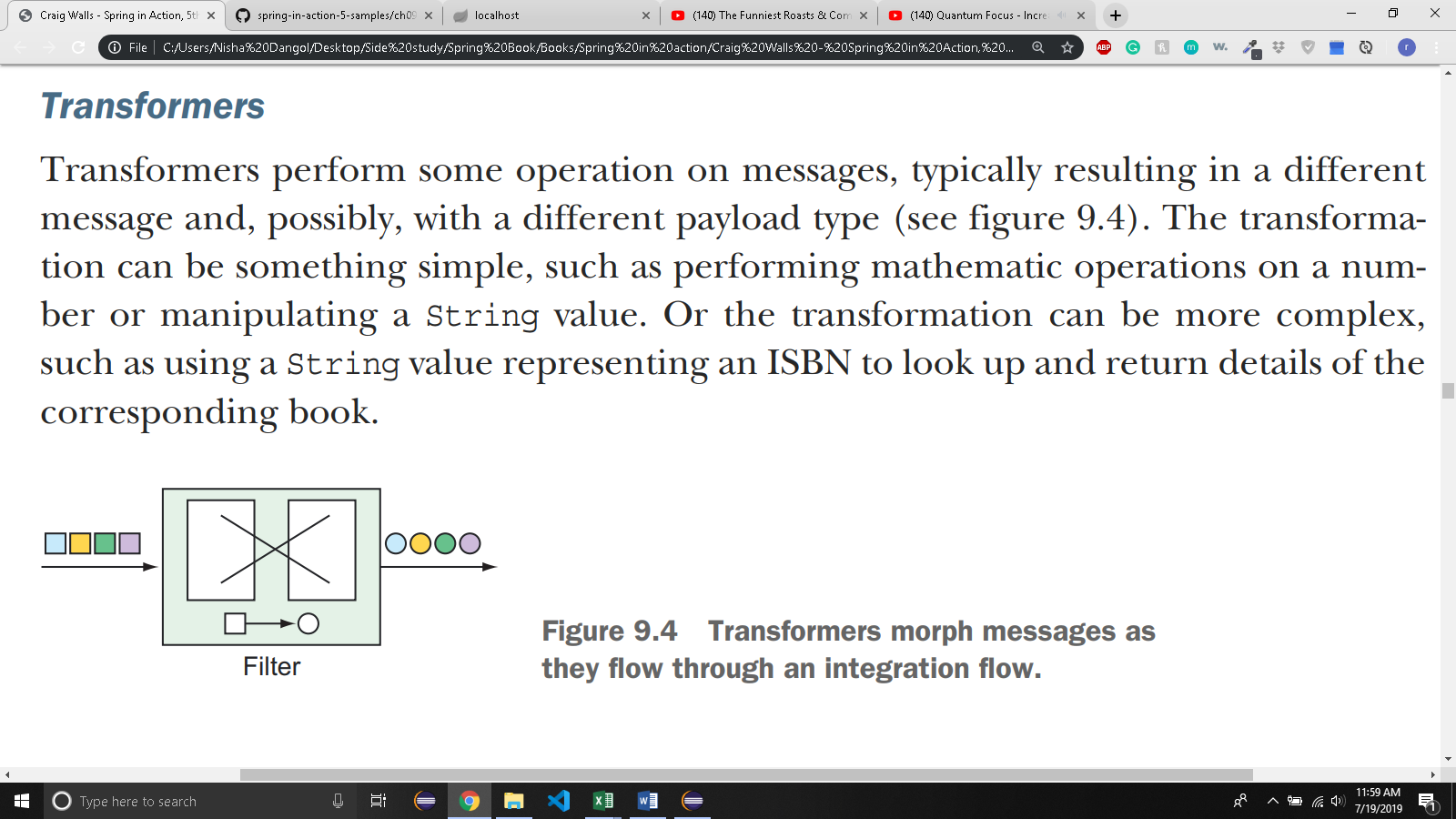
... .

<Integer>filter((p) -> p % 2 == 0)

.get();

}

* 9.2.3 Transformers



* + Suppose there are integers in a channel named ‘numberChanel’
  + You want to convert those numbers into strings containing the Roman numreral equivalent.

@Bean

@Transformer(inputChannel="numberChannel", outputChannel="romanNumberChannel")

public GenericTransformer<Integer, String> romanNumTransformer() {

return RomanNumbers::toRoman;

}

* Uses static method toRoman() to do the conversion(The toRoman() method is statically defined in class named RomanNumbers and referenced here with a method reference)
* You can do the same using Java DSL

@Bean

public IntegrationFlow transformerFlow() {

return IntegrationFlows

... .

.transform(RomanNumbers::toRoman)

... .

.get();

}

* + If a transformer is complex, you can inject it as a bean into the flow configuration and pass the reference to the transform() method>

@Bean

public RomanNumberTransformer romanNumberTransformer() {

return new RomanNumberTransformer();

}

@Bean

public IntegrationFlow transformerFlow( RomanNumberTransformer romanNumberTransformer) {

return IntegrationFlows

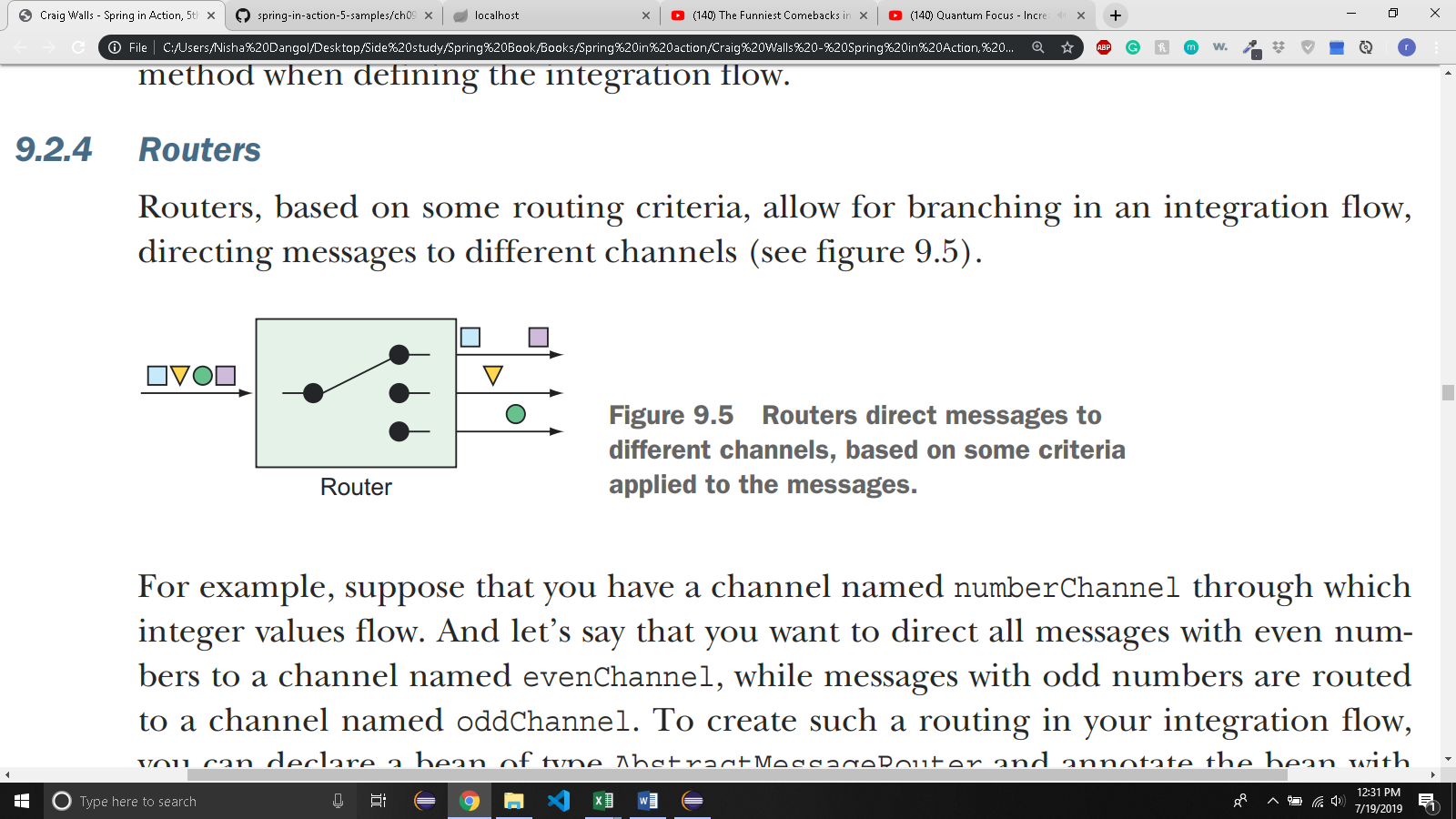
...

.transform(romanNumberTransformer)

...

.get();

}

* + You declare a bean of type RomanNumberTransformer, which is an implementation of Spring Integration’s Transformer or GenericTransformer interface.
  + The bean is injected into transformerFlow() method and passed to the transform() method when defining integration flow.
* 9.2.4 ROUTERS:
* 
  + Suppose you want to direct even number to even channel and odd numbers to oddChanel.
  + Declare a bean of type *AbstractMessageRouter* and annotate with @Router.

@Bean

@Router(inputChannel="numberChannel")

public AbstractMessageRouter evenOddRouter() {

return new AbstractMessageRouter() {

@Override

protected Collection determineTargetChannels(Message message) {

Integer number = (Integer) message.getPayload();

if (number % 2 == 0) {

return Collections.singleton(evenChannel());

}

return Collections.singleton(oddChannel());

}

};

}

@Bean

public MessageChannel evenChannel() {

return new DirectChannel();

}

@Bean

public MessageChannel oddChannel() {

return new DirectChannel();

}

* + AbstractMessageRouter bean accepts message from ‘numberChannel’
  + The implementation, defined as an anonymous inner class, examines the message payload and returns the appropriate channel.
  + In JAVA DSL, routers are declared as route():

@Bean

public IntegrationFlow numberRoutingFlow(AtomicInteger source) {

return IntegrationFlows

...

.<Integer,String> route(n -> n%2==0 ? "EVEN":"ODD", mapping -> mapping

.subFlowMapping("EVEN", sf -> sf

.<Integer,Integer>transform(n -> n \* 10)

.handle((i,h) -> { ... })

)

.subFlowMapping("ODD", sf -> sf

.transform(RomanNumbers::toRoman)

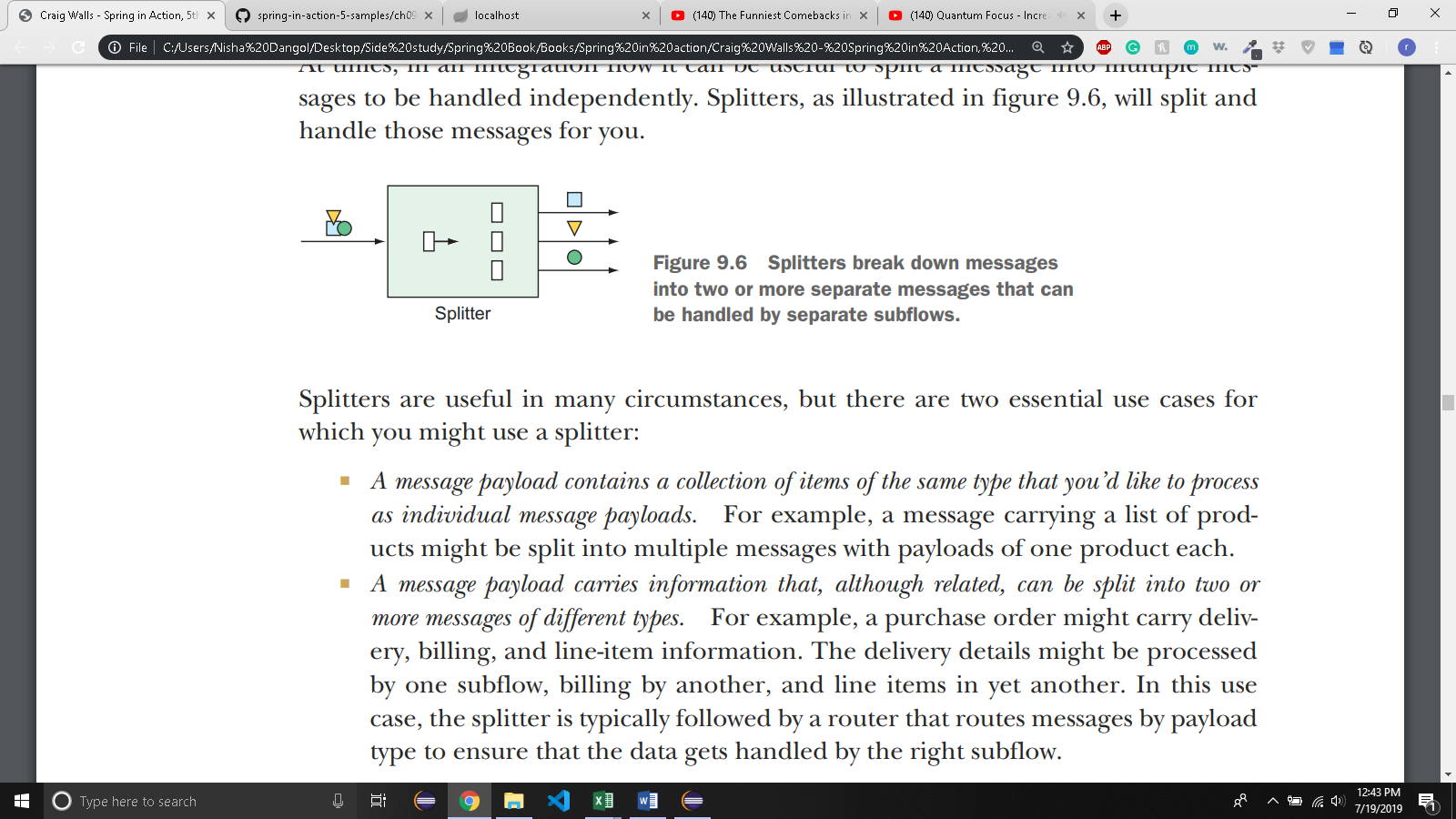
.handle((i,h) -> { ... }) )

)

.get();

}

* + If the number is even, then String value of EVEN is returned. If it’s odd, ODD is returned. These values are then used to determine which submapping will handle the message.
* 9.2.5 SPLITTERS



* + Suppose you want to split message carrying the purchase order into two messages: one carrying the billing information and the other carrying a list of line items.

public class OrderSplitter {

public Collection<Object> splitOrderIntoParts(PurchaseOrder po) {

ArrayList<Object> parts = new ArrayList<>();

parts.add(po.getBillingInfo());

parts.add(po.getLineItems());

return parts;

}

}

* + You can the declare the OrderSplitter bean as part of the Integration flow by annotating it with @Splitter like this:

@Bean

@Splitter(inputChannel="poChannel", outputChannel="splitOrderChannel")

public OrderSplitter orderSplitter() {

return new OrderSplitter();

}

* + Here, purchase order arrives in channel named poChannel and split by OrderSplitter.
  + The array returned by OrderSplitter has two elements. Those elements are published as a separate messages in the integration flow to a channel named splitOrderChannel.
  + Now, you can declare PayloadTypeRouter to route the billing information and the line items to their own subflows:

@Bean

@Router(inputChannel="splitOrderChannel")

public MessageRouter splitOrderRouter() {

PayloadTypeRouter router = new PayloadTypeRouter();

router.setChannelMapping(

BillingInfo.class.getName(), "billingInfoChannel");

router.setChannelMapping(

List.class.getName(), "lineItemsChannel");

return router;

}

* + Message whose payload is type BillingInfo are routed to chanel named billingInfoChannel.
  + Line items are in a java.util.List collection List<LineItem>. So, the payload of type List is routed to channel named lineItemsChannel.
  + What if you want to split the line line items as well?
  + Write a method (not bean) that’s annotated with @Splitter and returns a collection of LineItems:

@Splitter(inputChannel="lineItemsChannel", outputChannel="lineItemChannel")

public List<LineItem> lineItemSplitter(List<LineItem> lineItems) {

return lineItems;

}

* + When a message of lineitems arrive in lineItemsChannel, it is passed into lineItemSplitter() method.
  + The method returns list of items.
  + Each item is then passed to lineItemChannel as a different message.
  + You can use Java DSL:

return IntegrationFlows

...

.split(orderSplitter())

.<Object, String> route(

p -> {

if (p.getClass().isAssignableFrom(BillingInfo.class)) {

return "BILLING\_INFO";

} else {

return "LINE\_ITEMS";

}

}, mapping -> mapping

.subFlowMapping("BILLING\_INFO", sf -> sf

.<BillingInfo> handle((billingInfo, h) -> {

...

}))

.subFlowMapping("LINE\_ITEMS", sf -> sf

.split()

.<LineItem> handle((lineItem, h) -> {

...

}))

)

.get();

* + It uses the same OrderSplitter to split the objects and route by its type to separate subflows
* 9.2.6 SERVICE ACTIVATORS:



@Bean

@ServiceActivator(inputChannel="someChannel")

public MessageHandler sysoutHandler() {

return message -> {

System.out.println("Message payload: " + message.getPayload());

};

}

* + This bean is a service activator that handles message from channel named someChannel.
  + MessageHandler is implemented via lambda.When given a message, it emits the message payload to the standard output stream.
  + Here’s the service activator that saves the message before returning a new payload. You should use GenericHandler here.

@Bean

@ServiceActivator(inputChannel="orderChannel", outputChannel="completeOrder")

public GenericHandler<Order> orderHandler(OrderRepository orderRepo) {

return (payload, headers) -> {return orderRepo.save(payload);};

}

* + Here, service activator is generic handler that accepts message of type Order. When a message arrives, it is saved via a repository; the resulting saved order is returned to be sent to output channel named ‘completeOrder’.
  + You can also use Java DSL:
  + For message handler:

public IntegrationFlow someFlow() {

return IntegrationFlows

...

.handle(msg -> {

System.out.println("Message payload: " + msg.getPayload());

})

.get();

}

* + For generic handler:

public IntegrationFlow orderFlow(OrderRepository orderRepo) {

return IntegrationFlows

...

.<Order>handle((payload, headers) -> {

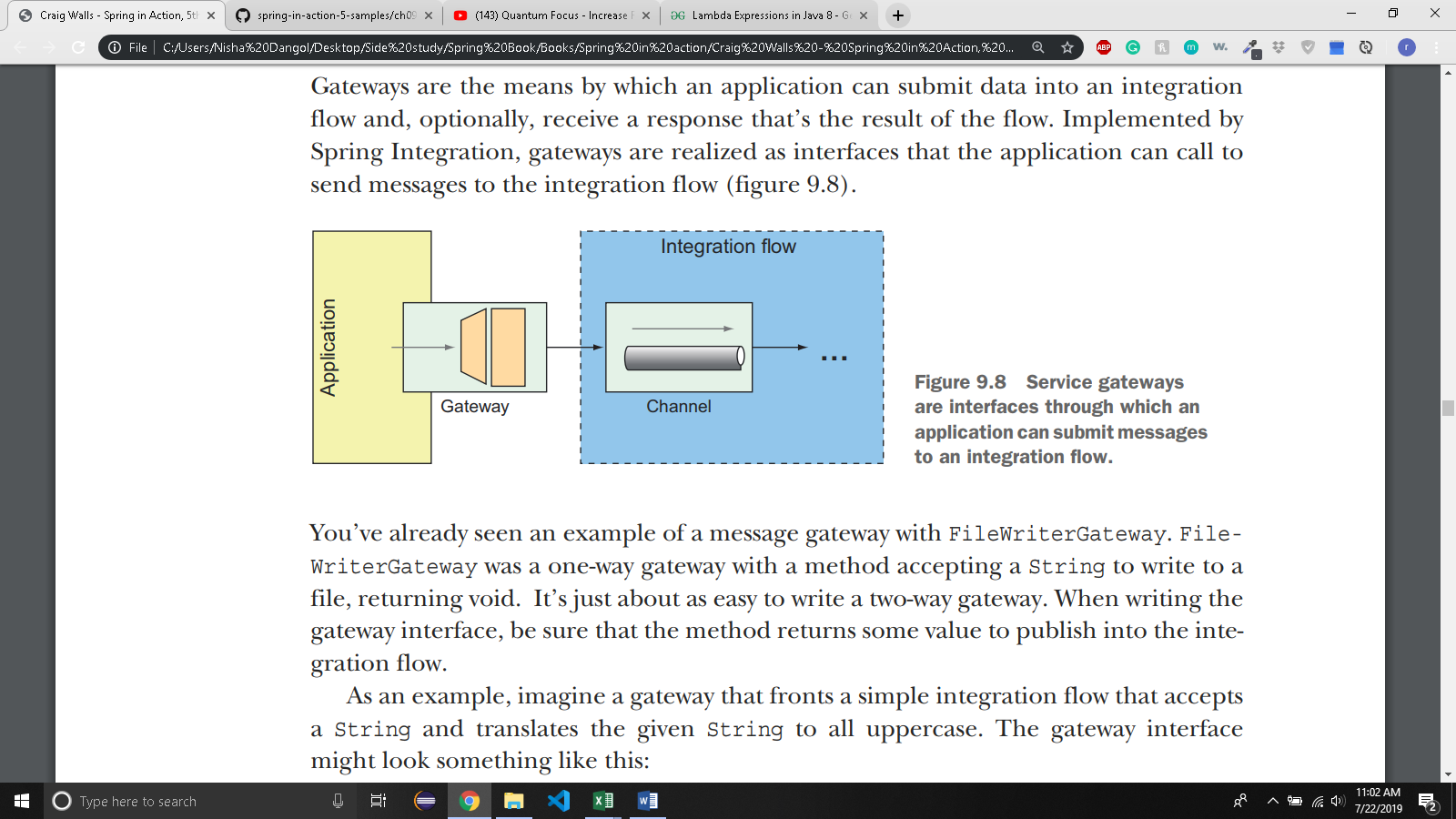
return orderRepo.save(payload);

})

...

.get();

}

* + For generic handler, at the end of the flow you’ll need to return null or else you’ll get errors indicating that there’s no output channel specified.
* 9.2.7 GATEWAYS
* 

@Component

@MessagingGateway(defaultRequestChannel="inChannel", defaultReplyChannel="outChannel")

public interface UpperCaseGateway {

String uppercase(String in);

}

* + Spring integration automatically provides implementation at runtime that sends and receives data through the specified channels.
  + When uppercase() is called, the given string is published in the integration flow into the channel named inChannel. When data arrives in the channel named outChannel, it is returned from the uppercase() method.
  + In java DSL,

@Bean

public IntegrationFlow uppercaseFlow() {

return IntegrationFlows

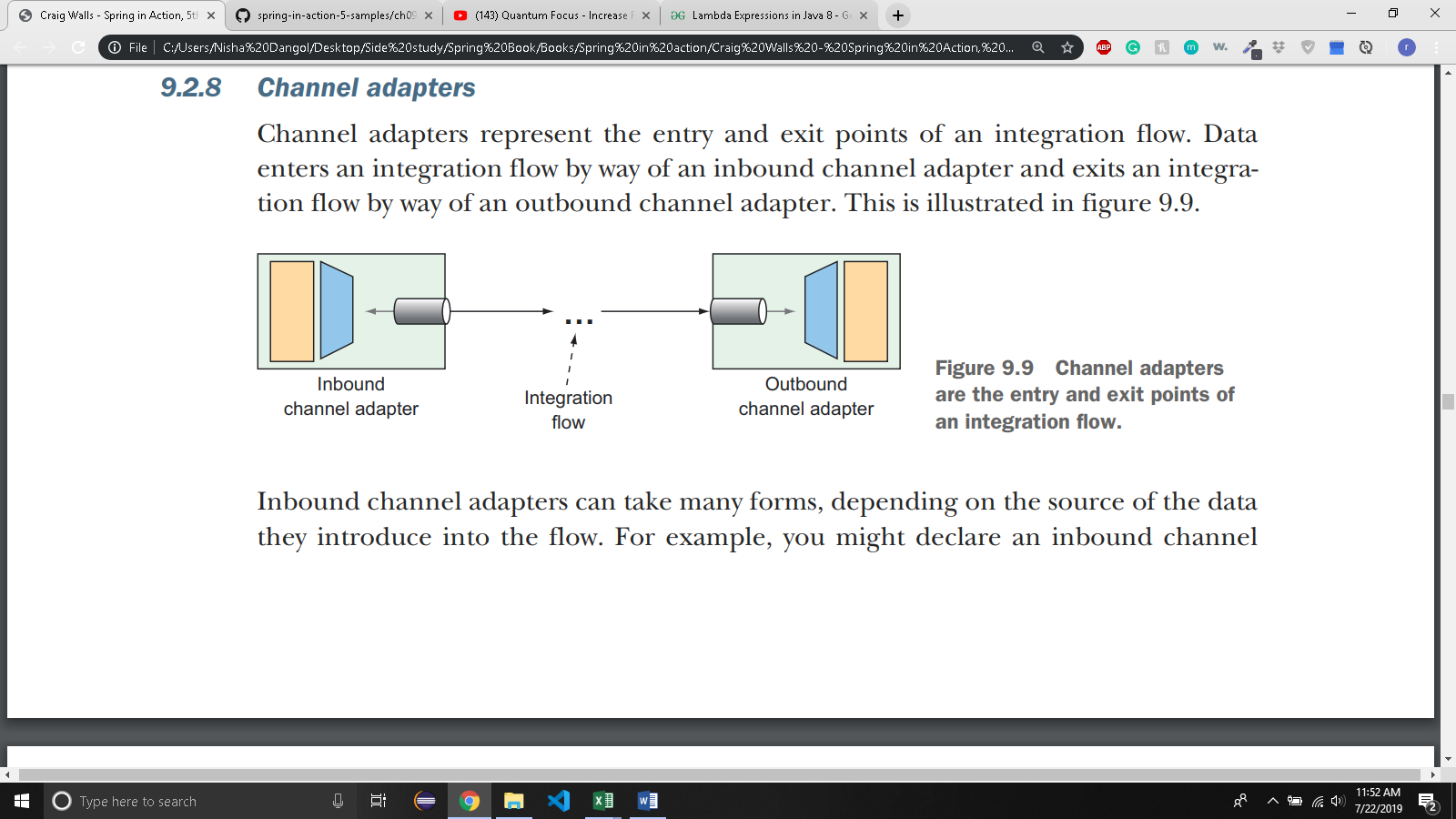
.from("inChannel")

.<String, String> transform(s -> s.toUpperCase())

.channel("outChannel")

.get();

}

* + The flow starts with data coming into the channel named inChannel.
  + The message payload is then transformed by the transformer, which is defined here as a lambda expression, to perform uppercase operation.
  + The resulting message is then published the the channel named outChannel which is what you’ve declared as the reply channel for the UpperCaseGateway interface.
* 9.2.8 CHANNEL ADAPTERS
* 
  + You might declare a inbound channel adapter that introduces incrementing numbers from an AtomicInteger into the flow. Using Java configuration, it might look like this:

@Bean

@InboundChannelAdapter(

poller=@Poller(fixedRate="1000"), channel="numberChannel")

public MessageSource<Integer> numberSource(AtomicInteger source) {

return () -> {

return new GenericMessage<>(source.getAndIncrement());

};

}

* + The bean submits the number from injected Atomic integer to the channel named numberChannel every 1 second(1000 ms).
  + In Java DSL, from() method indicates inbound channel adapter.

@Bean

public IntegrationFlow someFlow(AtomicInteger integerSource) {

return IntegrationFlows

.from(integerSource, "getAndIncrement",

c -> c.poller(Pollers.fixedRate(1000)))

...

.get();

}

* + Channel Adapters are ofthen provided by spring integration’s module
  + Suppose you need an inbound channel adapter that monitors a specified directory and submits any files written to that directory as messages to a channel named file-channel.
  + You can use FileReadingMessageSource from Spring Integration’s file endpoint module:

@Bean

@InboundChannelAdapter(channel="file-channel", poller=@Poller(fixedDelay="1000"))

public MessageSource<File> fileReadingMessageSource() {

FileReadingMessageSource sourceReader = new FileReadingMessageSource();

sourceReader.setDirectory(new File(INPUT\_DIR));

sourceReader.setFilter(new SimplePatternFileListFilter(FILE\_PATTERN));

return sourceReader;

}

* + In java dsl, inboundAdapter() method from Files class is same as file-reading inbound channel adapter.
  + An outbound channel adapter is the end of line for the integration flow, handing off the final message to the application or to some other system:

@Bean

public IntegrationFlow fileReaderFlow() {

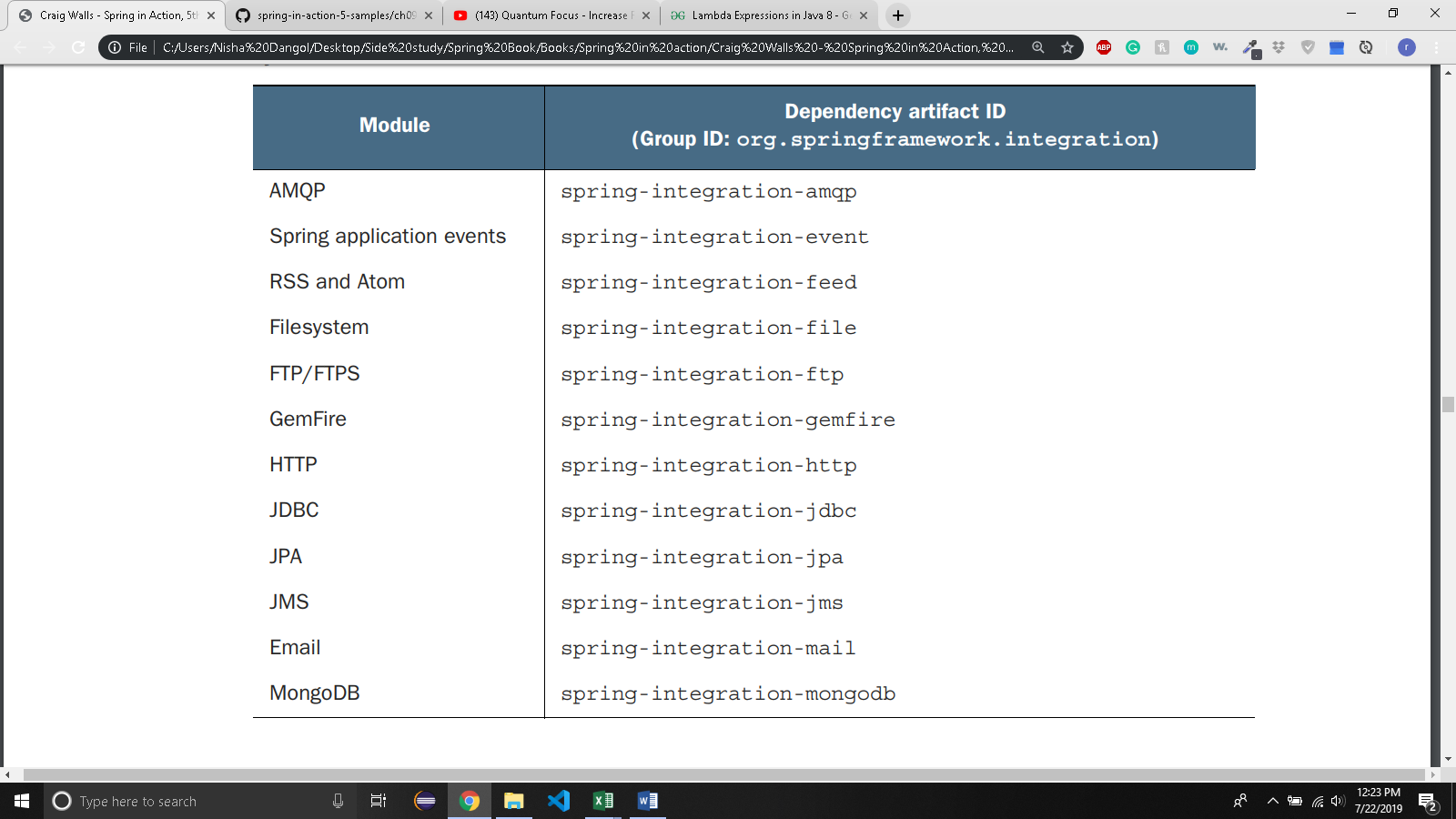
return IntegrationFlows

.from(Files.inboundAdapter(new File(INPUT\_DIR))

.patternFilter(FILE\_PATTERN))

.get();

}

* + Service activators implemented as message handlers often serve the purpose of an outbound channel adapter especially when data needs to be handed off to the application itself.
* 9.2.9 ENDPOINT MODULES:
  + 
  + 
* 9.3 CREATING AN EMAIL INTEGRATION FLOW
  + You will implement and integration flow that polls the TacoCloud inbox for the taco order email, parses the emails for order details, and submits the orders to Taco Cloud for handling.
  + Integration flow will use an inbound channel adapter from the email endpoint module to ingest emails from the Taco cloud inbox into the integration flow.
  + Next step is to parse the emails into order objects that are handed off to another handler to submit orders to Taco cloud’s REST API, where they’ll be processed the same as any order.

@Data

@ConfigurationProperties(prefix="tacocloud.email")

@Component

public class EmailProperties {

private String username;

private String password;

private String host;

private String mailbox;

private long pollRate = 30000;

public String getImapUrl() {

return String.format("imaps://%s:%s@%s/%s",

this.username, this.password, this.host, this.mailbox);

}

}

* + EmailProperties captures properties that are used to produce an IMAP URL.
  + The flow uses this URL to connect to the Taco cloud’s email server and poll for emails.
  + Properties captured are: user’s username and password, hostname of the IMAP server, the mailbox to poll, and the rate at which the mailbox is polled (which defaults to every 30 seconds).
  + EmailProperties class is annotated at the class level with @ConfigurationProperties with prefix attribute set to tacocloud.email. This means you can configure the details of consuming an email in the application.yml file like this.

tacocloud:

email:

host: imap.tacocloud.com

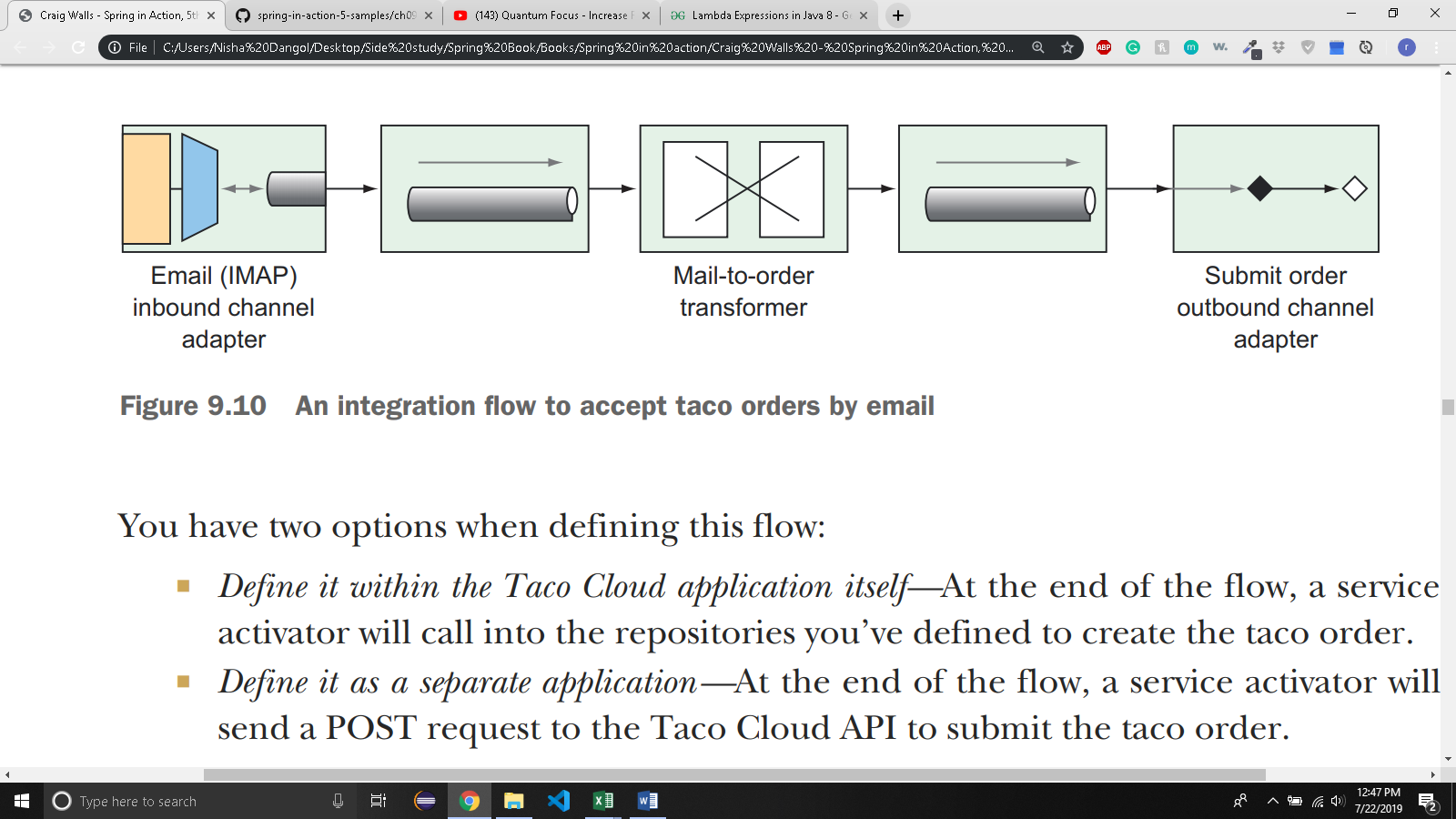
mailbox: INBOX

username: taco-in-flow

password: 1L0v3T4c0s

poll-rate: 10000

* + Now let’s use EmailProperties to configure the integration flow.



* + You are going to need some types that represents tacos, orders, and ingredients which are subtly different that those you’ve already defined in the main Tacocloud application.
  + So, you will define integration flow in a separate application to avoid any confusion with existing domain types.

@Configuration

public class TacoOrderEmailIntegrationConfig {

@Bean

public IntegrationFlow tacoOrderEmailFlow(

EmailProperties emailProps,

EmailToOrderTransformer emailToOrderTransformer,

OrderSubmitMessageHandler orderSubmitHandler) {

return IntegrationFlows

.from(Mail.imapInboundAdapter(emailProps.getImapUrl()),

e -> e.poller(

Pollers.fixedDelay(emailProps.getPollRate())))

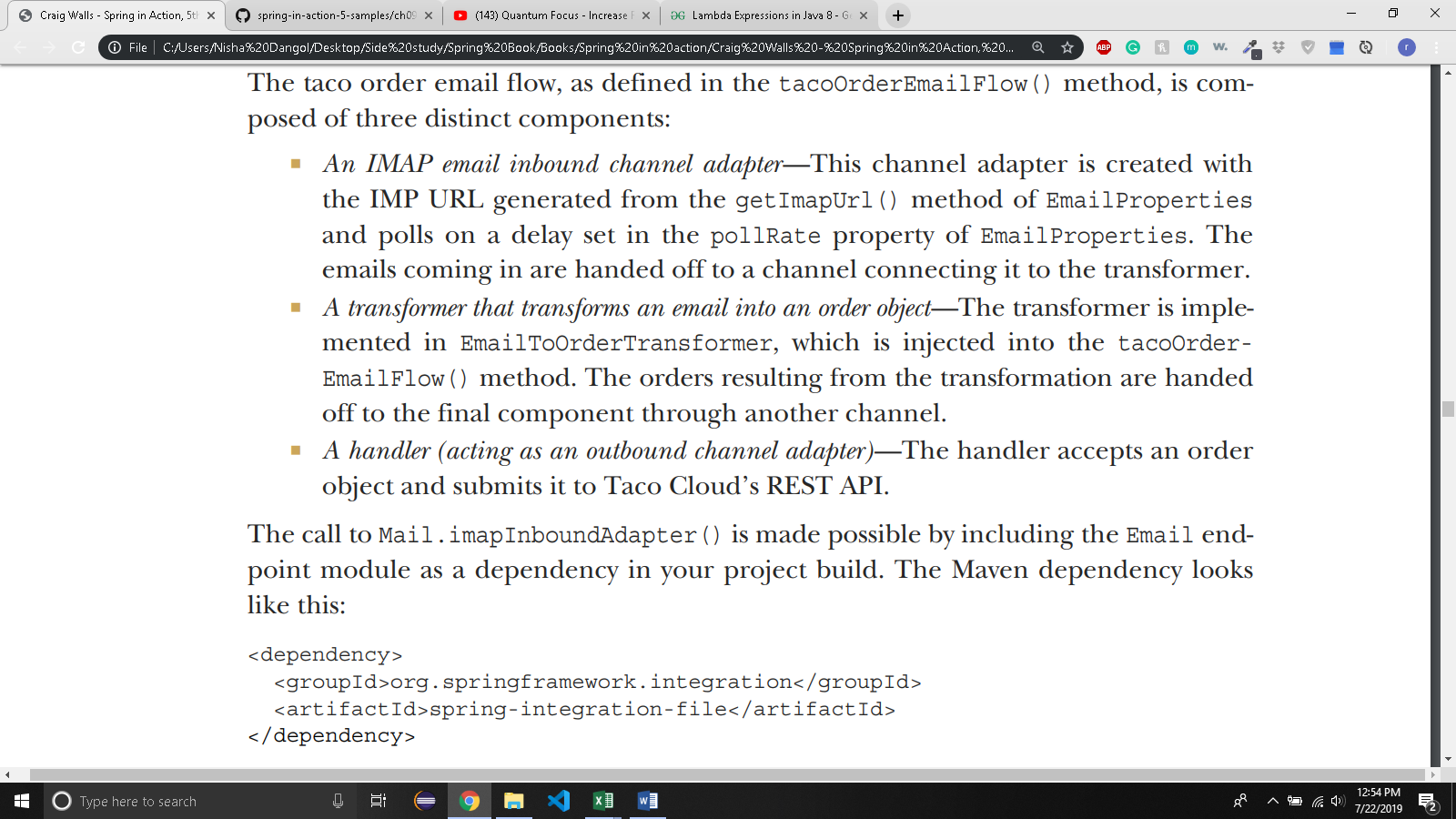
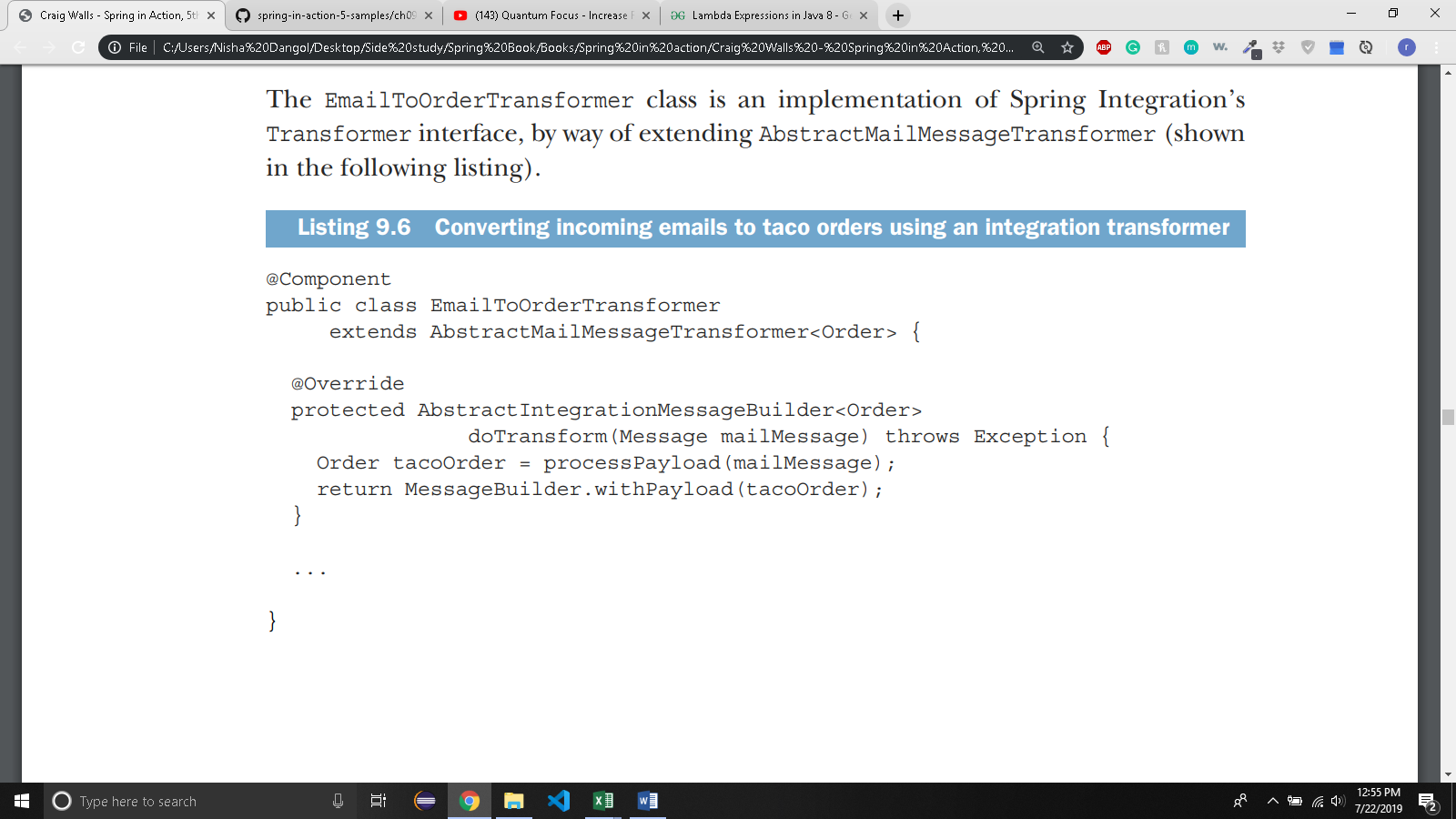
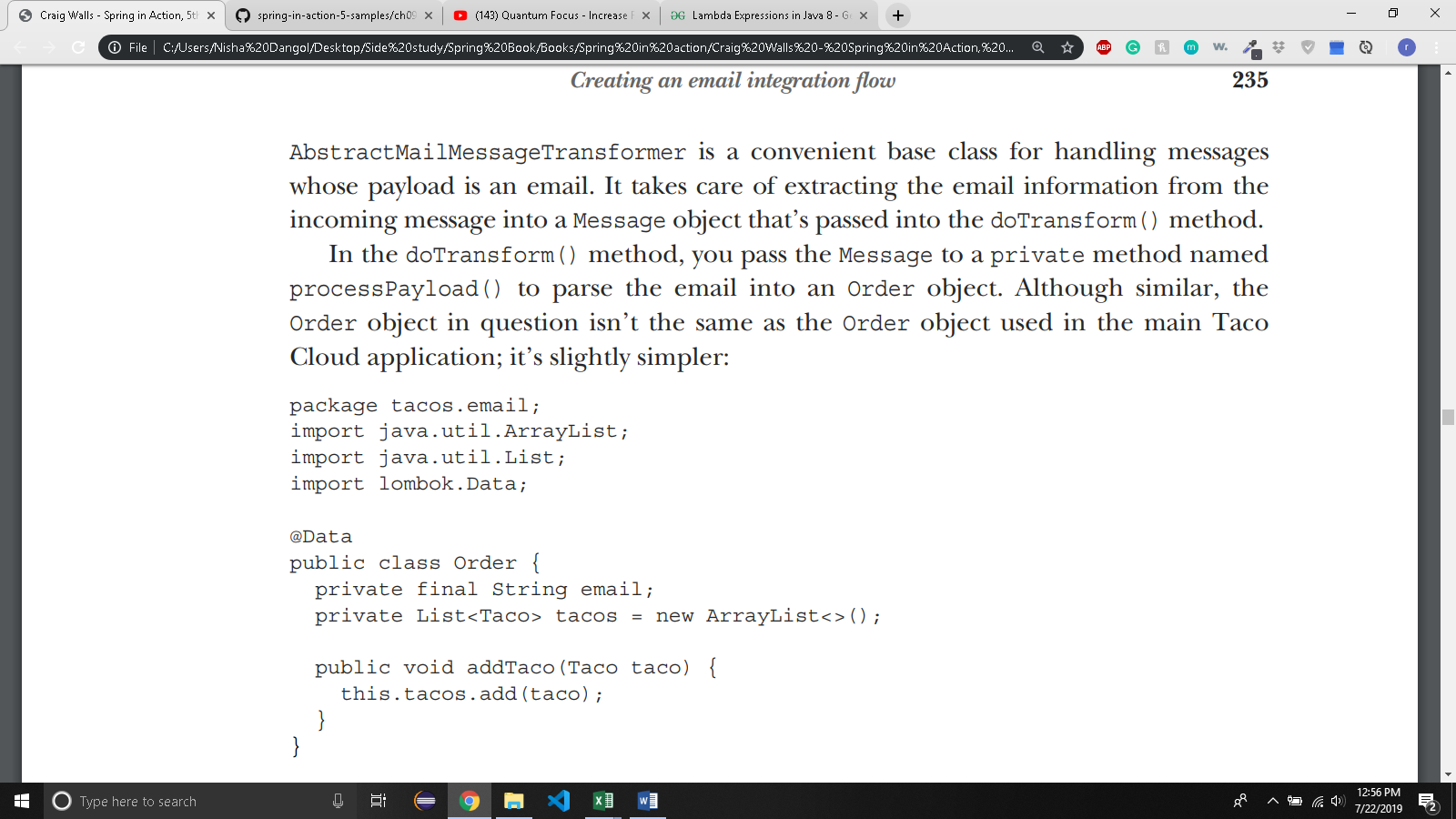
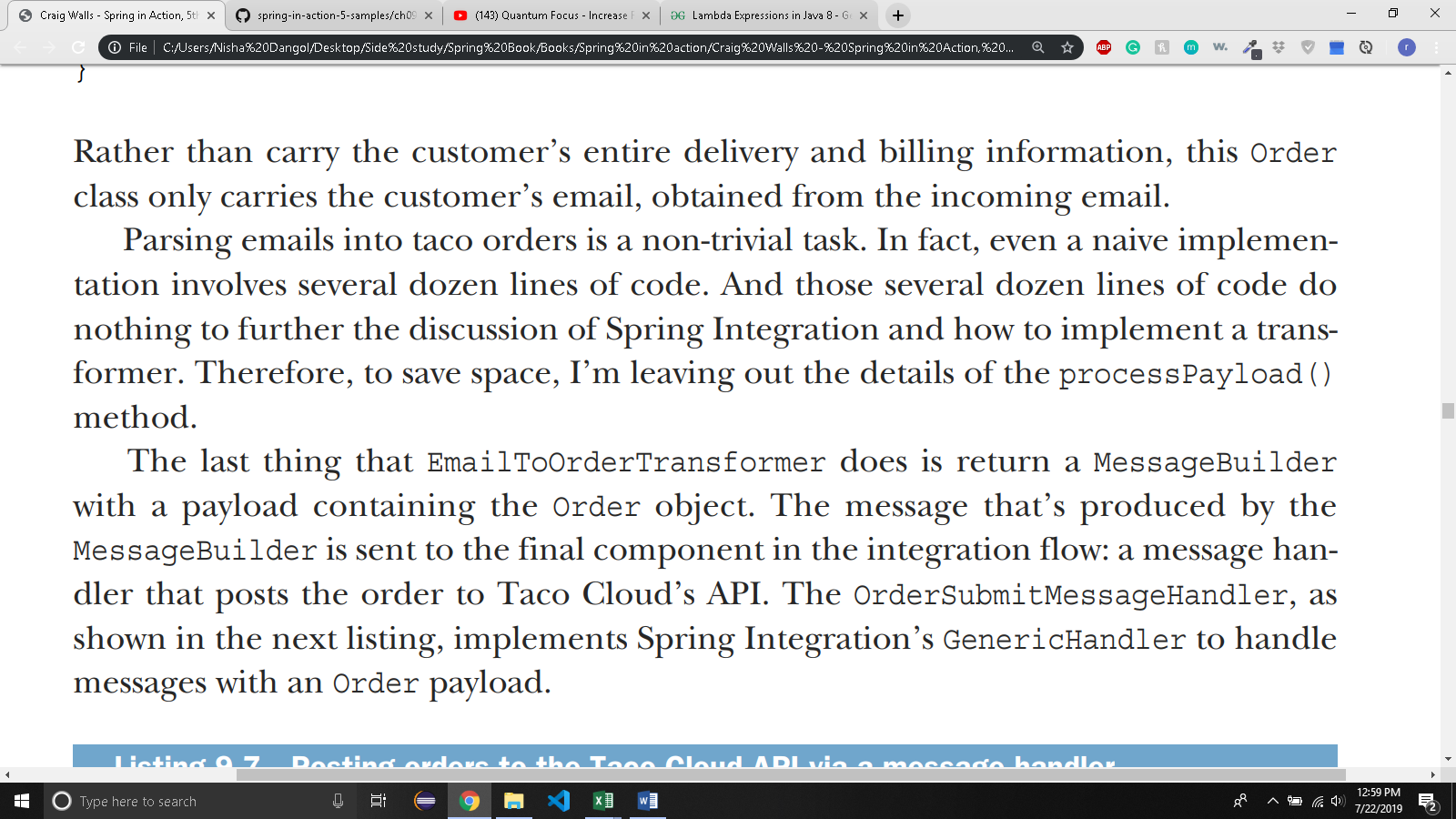
.transform(emailToOrderTransformer)

.handle(orderSubmitHandler)

.get();

}

}

* + 
  + 
  + 
  + Parsing emails to taco orders is difficult. So, the author left the details for processPayload().
  + 

@Component

public class OrderSubmitMessageHandler implements GenericHandler<Order> {

private RestTemplate rest;

private ApiProperties apiProps;

public OrderSubmitMessageHandler( ApiProperties apiProps, RestTemplate rest) {

this.apiProps = apiProps;

this.rest = rest;

}

@Override

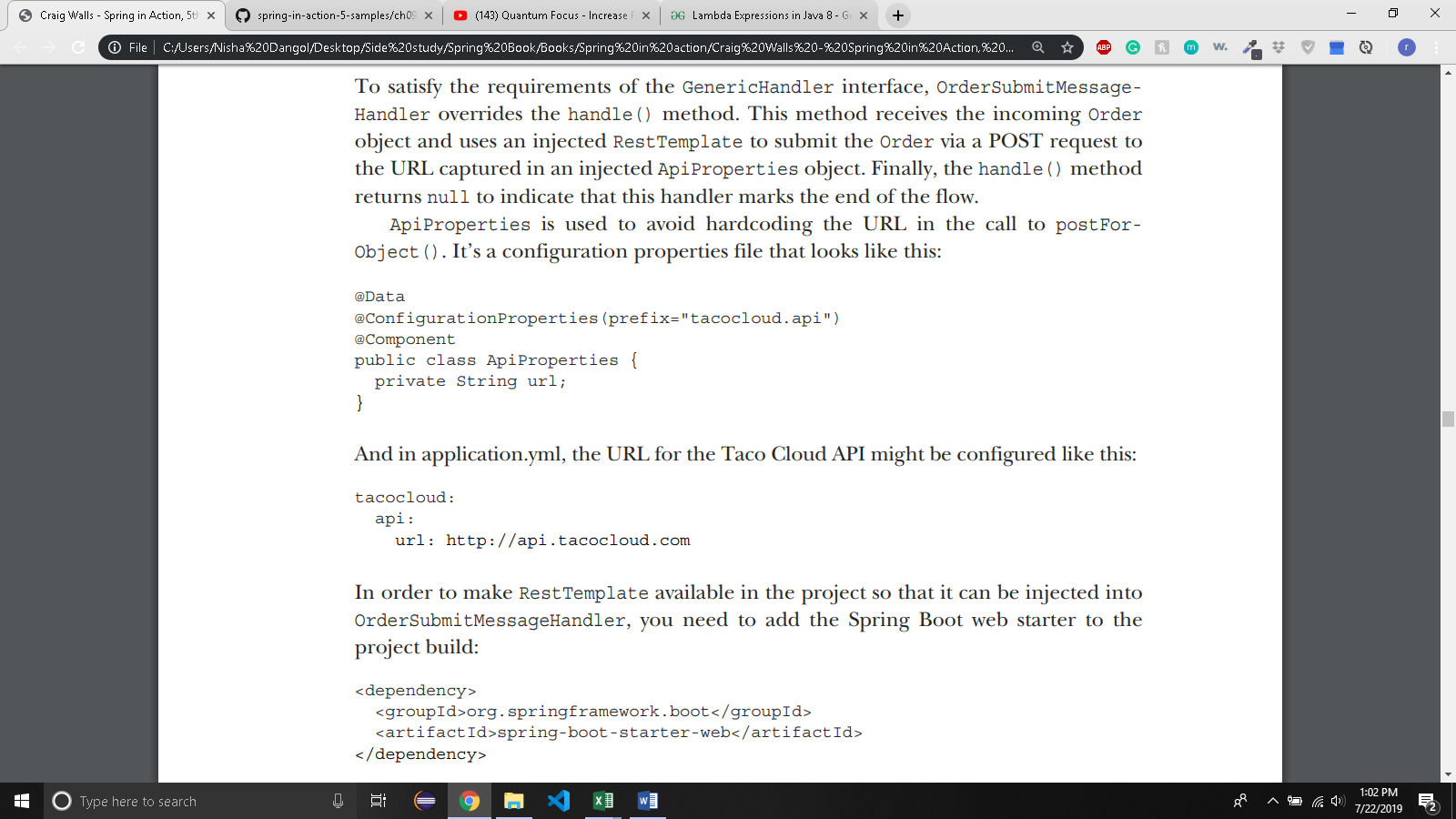
public Object handle(Order order, Map<String, Object> headers) {

rest.postForObject(apiProps.getUrl(), order, String.class);

return null;

}

}

* + 
  + 