Chapter 8 – Sending Messages Asynchronouly

* Asynchronous messaging is a way of indirectly sending messages from one application to another without waiting for a response.
* 8.1 SENDING MESSAGES WITH JMS:
  + Before you can send or receive messages, you need a message broker that’s ready to relay those messages between producers and consumers.
  + 8.1.1 SETTING UP JMS
    - You need to first add broker (JMS starter) Apache ActiveMQ Artemis since it is the latest.

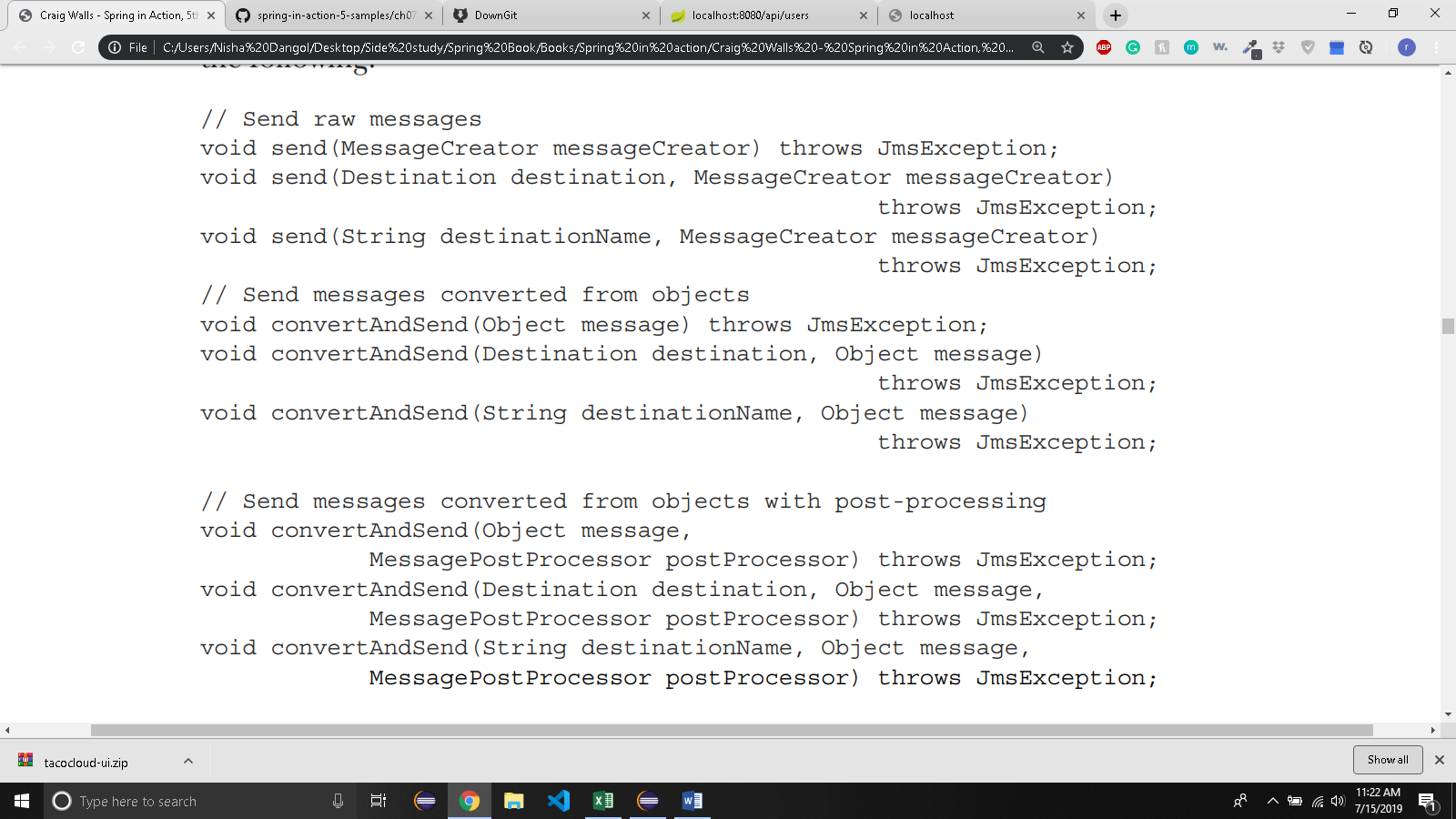
<dependency>

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

<artifactId>spring-boot-starter-artemis</artifactId>

</dependency>

* + - Default port for artemis is 61616
  + 8.1.2 SENDING MESSAGES WITH JMSTEMPLATE:
    - With Artemis, Spring boot autoconfigures JMSTemplate.
    - Without JMSTemplate, you’d need to write code to create a connection and session with the message broker, and more code to deal with any exceptions that might be thrown in the course of sending a message.
    - There are different methods for JMSTemplate:



* + - Three send() methods require a MessageCreator to manufacture a Message object.
    - Three convertAndSend() methods accept an object and automatically convert that object into a Message behind the scenes.
    - Three ConvertAndSend() methods automatically convert object to a Message, but also accept a MessagePostProcessor to allow for customization of the message before it’s sent.
    - Each of the these three methods have overriding methods:
      * One accepts no destination parameter and sends the message to a default destination.
      * One method accepts a Destination object that specifies the destination for the message
      * One method accepts a String that specifies the destination for the message by name.
    - You can set the default destination property in application.yml like this:

spring:

jms:

template:

default-destination: tacocloud.order.queue

* + - ConvertAndSend() automatically converts the object into message before sending.

package tacos.messaging.jms;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jms.core.JmsTemplate;

import org.springframework.stereotype.Service;

import tacos.Order;

*@Service*

public class JmsOrderMessagingService implements OrderMessagingService{

private JmsTemplate jms;

*@Autowired*

public JmsOrderMessagingService(JmsTemplate jms) {

this.jms = jms;

}

/\*

\* Converts the Order to message automatically to tacocloud.order.queue

\* destination.

\*/

*@Override*

public void sendOrder(Order order) {

jms.convertAndSend("tacocloud.order.queue",order);

}

}

* + - MessageConverter is a spring defined interface.
    - SimpleMessageConverter is the default implementation of MessageConverter but it requires that object being sent implements Serializable.
    - You may want to use another converter like MappingJackson2MessageConverter to avoid that restriction.

package tacos.messaging.jms;

import java.util.HashMap;

import java.util.Map;

import org.springframework.context.annotation.Bean;

import org.springframework.jms.support.converter.MappingJackson2MessageConverter;

import tacos.Order;

public class MessagingConfig {

*@Bean*

/\*

\* MappingJackson2MessageConverter uses the Jackson 2 JSON library to convert

\* messages to and from JSON. This bean declaration will enable

\* MappingJackson2MessageConverter to be used instead of SimpleMessageConverter.

\*/

public MappingJackson2MessageConverter messageConverter() {

MappingJackson2MessageConverter messageConverter = new MappingJackson2MessageConverter();

/\*

\* setTypeIdPropertyName enables receiver to know what type to convert an incoming message to. By

\* default \_typeId will contain full classname of the type being converted i.e Order.class. However,

\* that's inflexible as the receiver also needs the same full classname.

\*/

messageConverter.setTypeIdPropertyName("\_typeId");

Map<String, Class<?>> typeIdMappings = new HashMap<String, Class<?>>();

/\*

\* The value 'order' will be sent to the messages's "\_typeId" property instead

\* of full class name\*

\*/

typeIdMappings.put("order", Order.class);

/\* maps synthetic type name to the actual type by calling setTypeIdMappings() on the message converter\*/

messageConverter.setTypeIdMappings(typeIdMappings);

return messageConverter;

}

}

* + Post PROCESSING MESSAGES :
    - Suppose the customers can order in store as well as online.
    - Kitchen staff should be able to know from where did the order come from.
    - It would make sense to add new ‘source’ property in the Order class telling ‘STORE’ for orders from store and ‘WEB’ for online orders.
    - However, you will need to change both website’s Order class and the kitchen application’s Order class when, in reality, it is an information only required by Taco preparers.
    - An easier solution is to add custom header to message to carry the Order’s source.

/\* method to add an Order\_Source property to the message in the parameter \*/

private Message addOrderSource(Message message) throws JMSException{

message.setStringProperty("Order\_Source","WEB");

return message;

/\*

\* Converts the Order(Json object) to message automatically(Using Mapping2JacksonConverter),

\* uses addSource method to add a Order\_Source property to the converted message and sends

\* it to tacocloud.order.queue destination.T

\*/

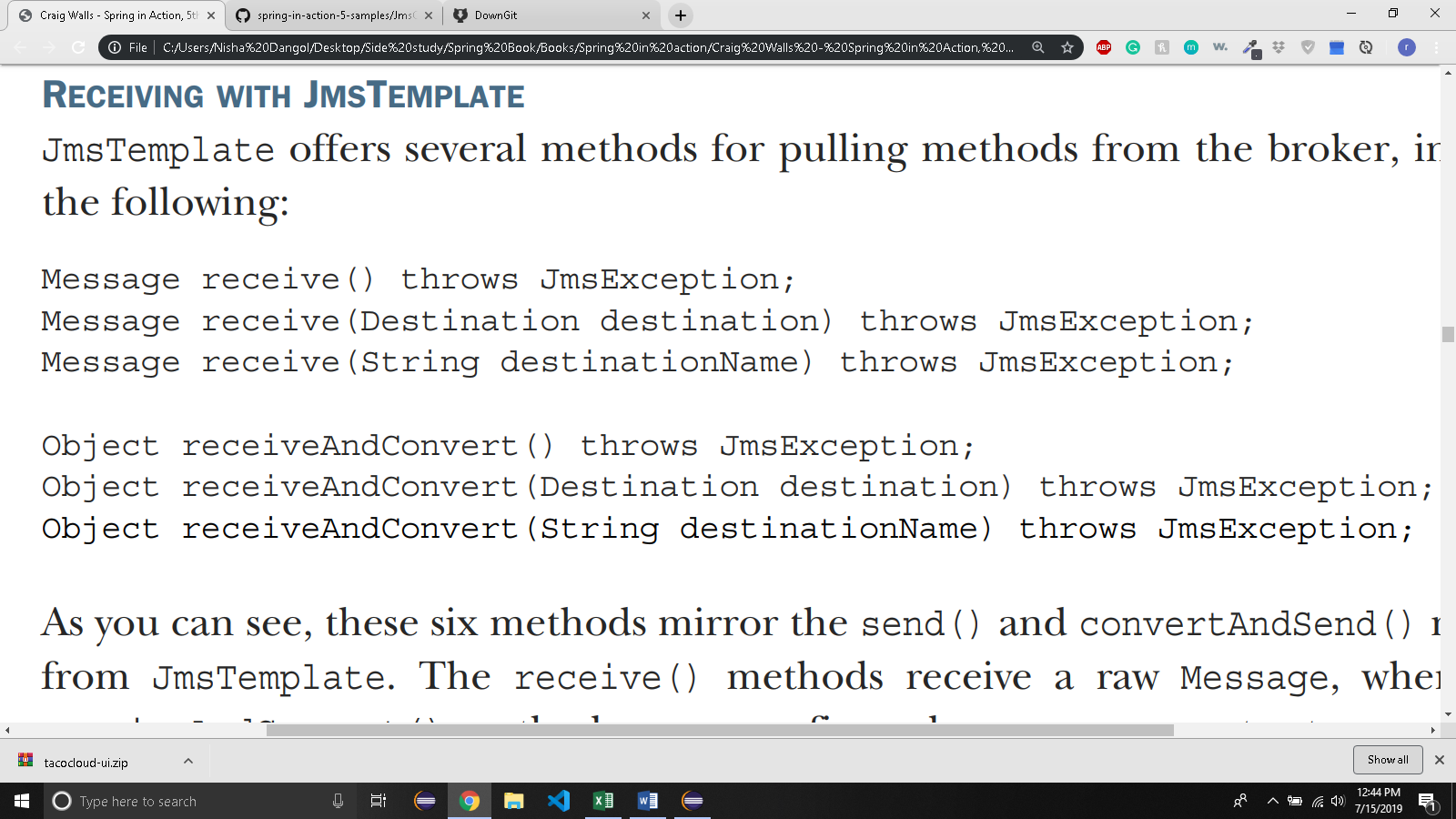
*@Override*

public void sendOrder(Order order) {

jms.convertAndSend("tacocloud.order.queue",order, this::addOrderSource);

}

* 8.1.3 RECEIVING JMS messages:
  + Pull model:
    - Code requests a message and waits until one arrives
  + Push Model:
    - Messages are handed to your code as they become available
  + Different methods for receive messages:



* + Receive() method receives a raw message.
  + receiveAndConvert() converts the raw message into domain types.
  + For each of these methods, you can either specify a Destination object, or string containing the destination name, or you can pull a message from the default destination.
  + Lets write some code that pulls an Order from tacocloud.order.queue destination.

*@Component*

public class JmsOrderReceiver implements OrderReceiver {

private JmsTemplate jms;

public JmsOrderReceiver(JmsTemplate jms) {

this.jms = jms;

}

*@Override*

public Order receiveOrder() {

return(Order) jms.receiveAndConvert("tacocloud.order.queue");

}

* }
  + Let’s take a look at push model:

*@Profile*("jms-listener")

*@Component*

public class OrderListener {

private KitchenUI ui;

*@Autowired*

public OrderListener(KitchenUI ui) {

this.ui = ui;

}

/\*

\* listens for messages at 'tacocoud.order.queue.

\* When messages arrive, the receiveOrder() method is invoked automatically with the messages's Order

\* Payload as a parameter.

\*/

*@JmsListener*(destination = "tacocloud.order.queue")

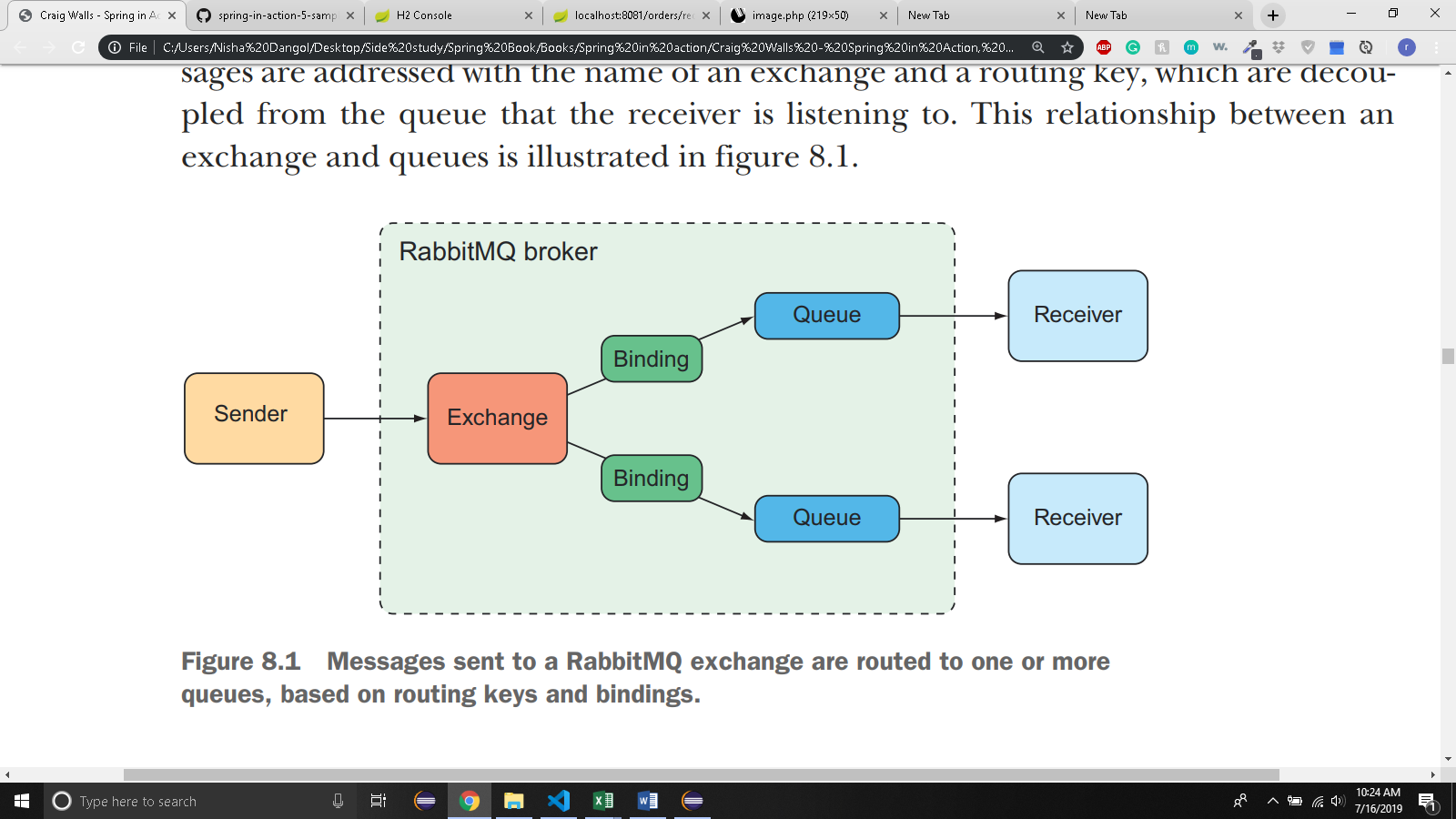
public void receiveOrder(Order order) {

ui.displayOrder(order);

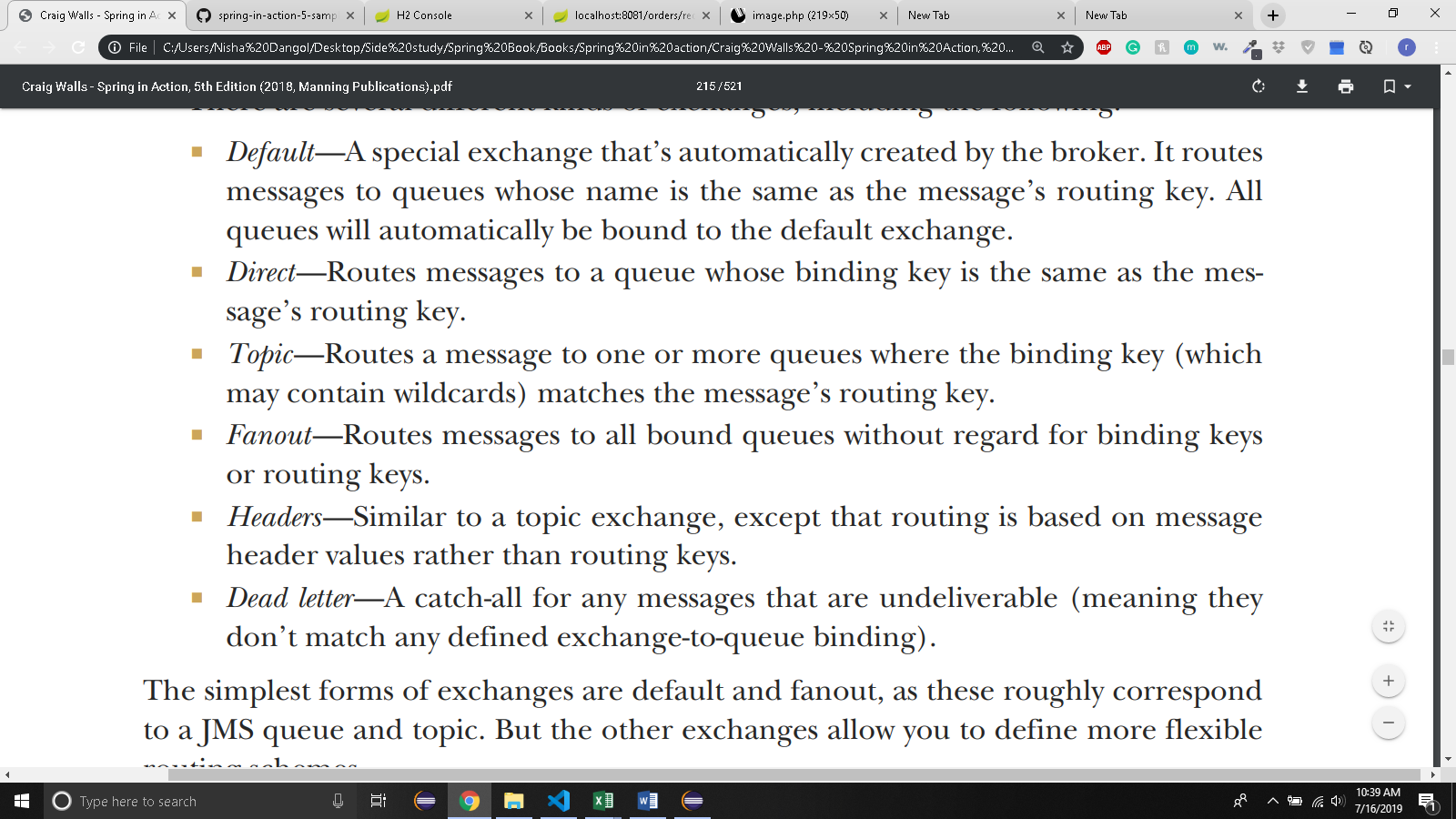
}

}

* + - JMS’s use is limited to only java application.
    - Newer messaging options such as RabbitMQ and kafka address these problems and are available for other languages and platforms beyond the JVM.
* 8.2 WORKING WITH RABBITMQ and AMQP
  + RabbitMQ is an implementation of AMQP
  + Advanced than JMS
  + AMQP messages are addressed with name of an exchange and a routing key which are decoupled from the queue that the receiver is listening to.



* + When a message arrives at RabbitMQ broker, the message goes to the exchange for which it is addressed.
  + The exchange routes the message to one or more queues depending on the type of exchange, the binding between the exchange and queues, and the value of the message’s routing key.
  + There are several kinds of exchanges



* + Messages are sent to exchanges with routing keys and they’re consumed from queues.
  + How they get from an exchange to a queue depends on the binding definitions and what best suits your use cases.
* 8.2.1 ADDING RABBITMQ to SPRING

<**dependency**>

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

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

</**dependency**>

* + This creates AMQP connection factory and RabbitTemplate beans and other supporting components for sendng and receiving messages.
  + Suppose that you move into production, your RabbitMQ broker is on a server named rabbit.tacocloud.com listening on port 5673,and require credentials. In prod profile, application.yml should be like this:

spring:

profiles: prod

rabbitmq:

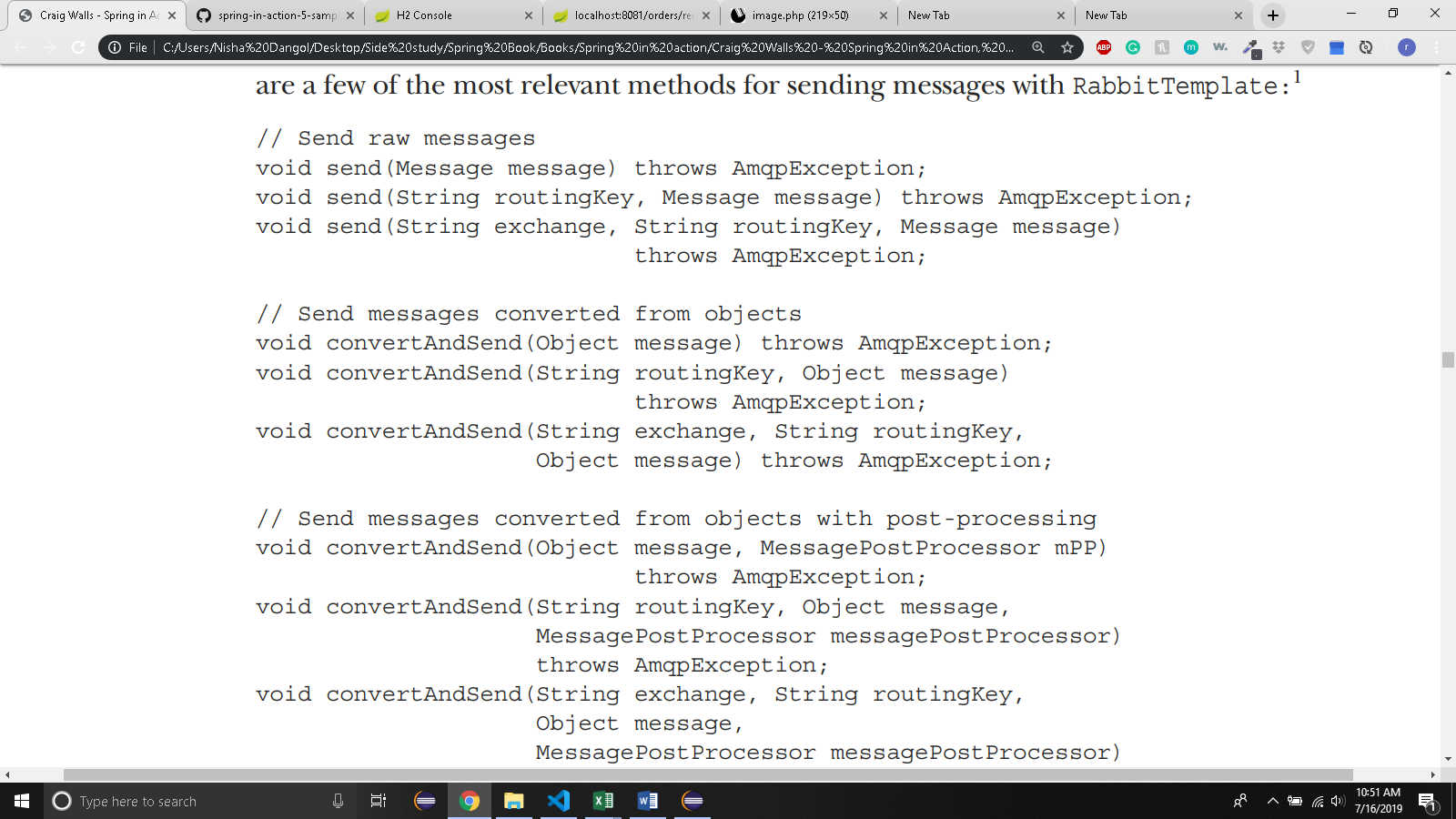
host:rabbit.tacocloud.com

port: 5673

username: tacoweb

password: letmein

* 8.2.2 SENDING MESSAGES WITH RABBITTEMPLATE
  + Here are few methods of RabbitTemplate:



* + The method that don’t take an exchange or routing key will have their messages sent to default exchange and routing key respectively.
  + The default exchange name is “ “(an empty String)
  + The default routing key is “ “(an empty String)
  + You can override these defaults by setting the yml properties:

spring:

rabbitmq:

template:

exchange: tacocloud.orders

routing-key: kitchens.central

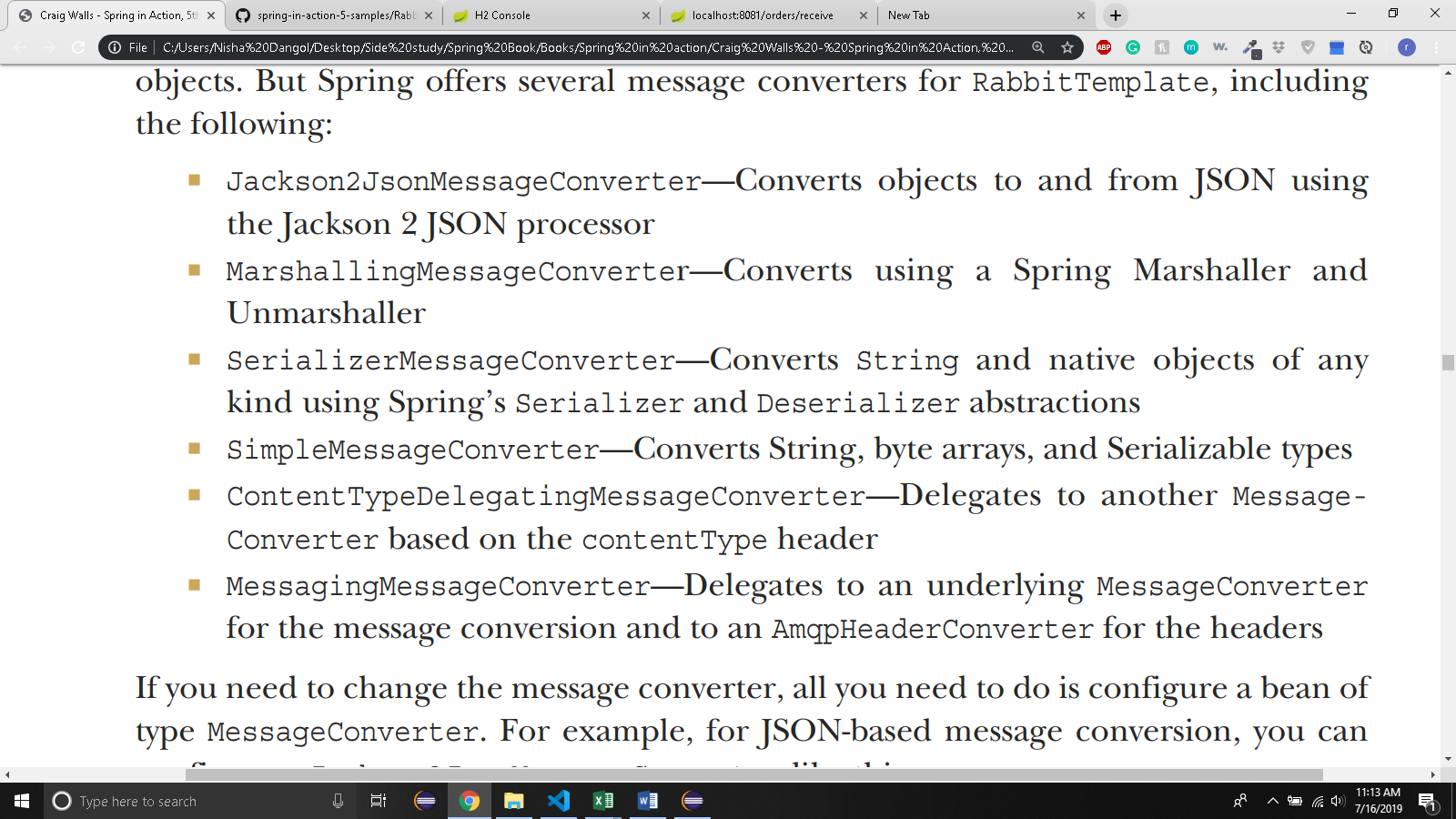
* + It’s easy to send with convertAndSend:

public void sendOrder(Order order) {

rabbit.convertAndSend("tacocloud.order", order);

}

* CONFIGURING A MESSAGE CONVERTER:
  + Default message conversion is done by SimpleMessageConverter which converts simple types(like String) and Seriablizable objects to message objects.
  + Spring offers several message Converter:



* + If you need to change converter, just declare a bean. Eg:

@Bean

public MessageConverter messageConverter() {

return new Jackson2JsonMessageConverter();

}

* + Spring boot autoconfiguration will discover this bean and inject it into RabbitTemplate in place of the default message converter.
* SETTING MESSAGE PROPERTIES:
  + You might want to add order \_source property to specify if the order came from web or store.
  + You can use MessagePostProcessor with convertAndSend()

*@Override*

public void sendOrder(Order order) {

rabbit.convertAndSend("tacocloud.order.queue",order,

new MessagePostProcessor() {

*@Override*

public Message postProcessMessage(Message message) throws AmqpException{

MessageProperties props = message.getMessageProperties();

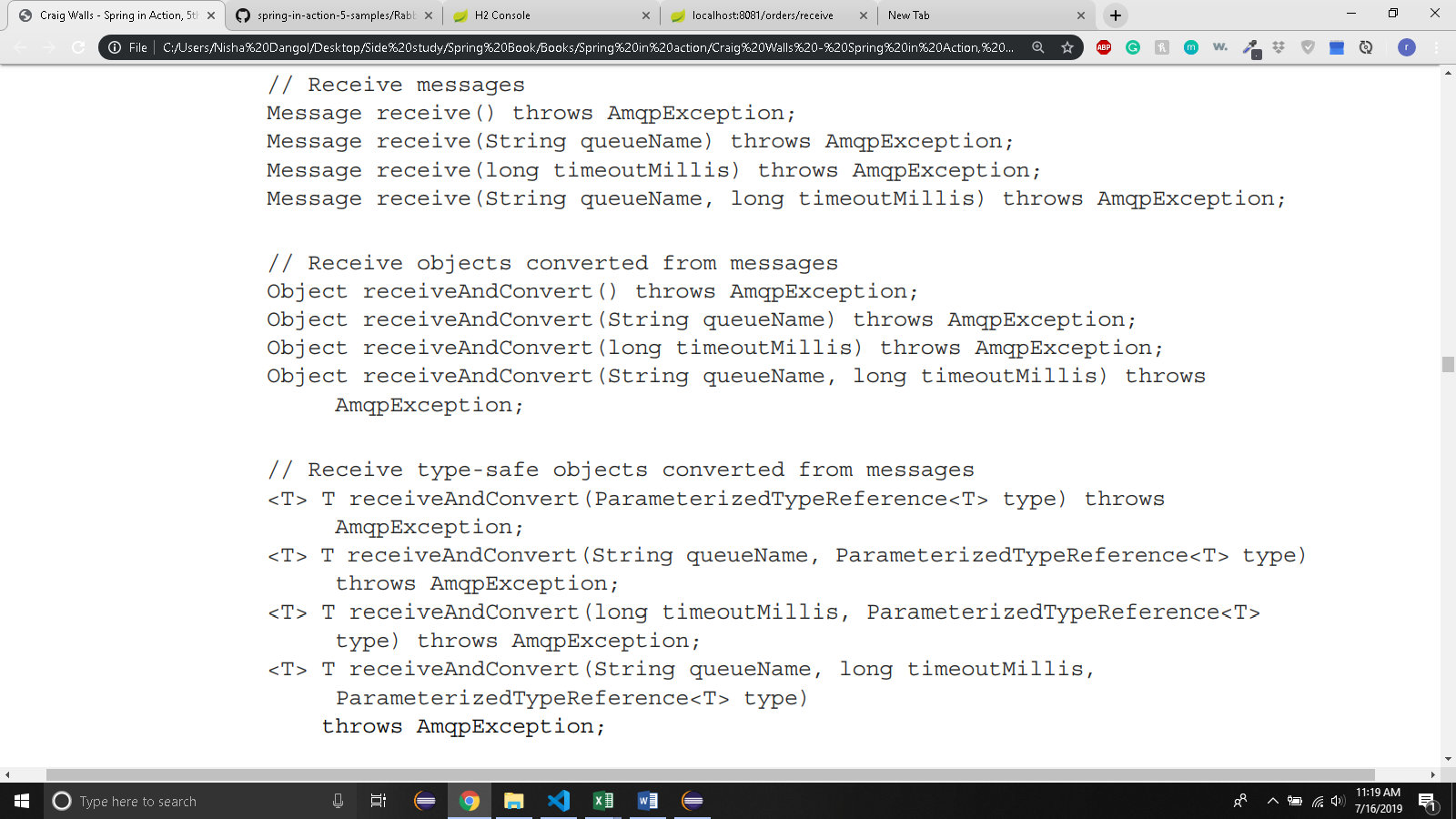
props.setHeader("ORDER\_SOURCE","WEB");

return message;

}

});

* 8.2.3 RECEIVING MESSAGE FROM RABBITMQ
  + Two choices:
    - Pulling messages from queue with RabbitTemplate
    - Having messages pushed to @RabbitListener annotated method
  + PULL METHOD:
    - There are different methods for pulling messages from queue:



* + - None of these methods take an exchange or routing key.
    - Consuming applications don’t need to concern themselves with exchanges or routing keys.
    - By default, the receive timeout is 0 Millisecond. That is, a call to receive() will return immediately, potentially with null value if no messages are available.
    - If you set the timeouts value, you can have the receive() and receiveAndConvert() methods to block until a message arrives or until the timout expires.
    - You can also set timeout via configuration:

spring:

rabbitmq:

template:

receive-timeout: 30000

* + - Here is RabbitOrderReceiver class:

*@Profile*("rabbitmq-template")

*@Component*("templateOrderReceiver")

public class RabbitOrderReceiver implements OrderReceiver {

private RabbitTemplate rabbit;

public RabbitOrderReceiver(RabbitTemplate rabbit) {

this.rabbit = rabbit;

}

*@Override*

public Order receiveOrder() {

// **TODO** Auto-generated method stub

return (Order) rabbit.receiveAndConvert("tacocloud.order.queue");

}

* + PUSH METHOD:
    - Annotate bean’s method with @RabbitListener

*@Profile*("rabbitmq-listener")

*@Component*

public class OrderListener {

private KitchenUI ui;

*@Autowired*

public OrderListener(KitchenUI ui) {

this.ui = ui;

}

*@RabbitListener*(queues = "tacocloud.order.queue")

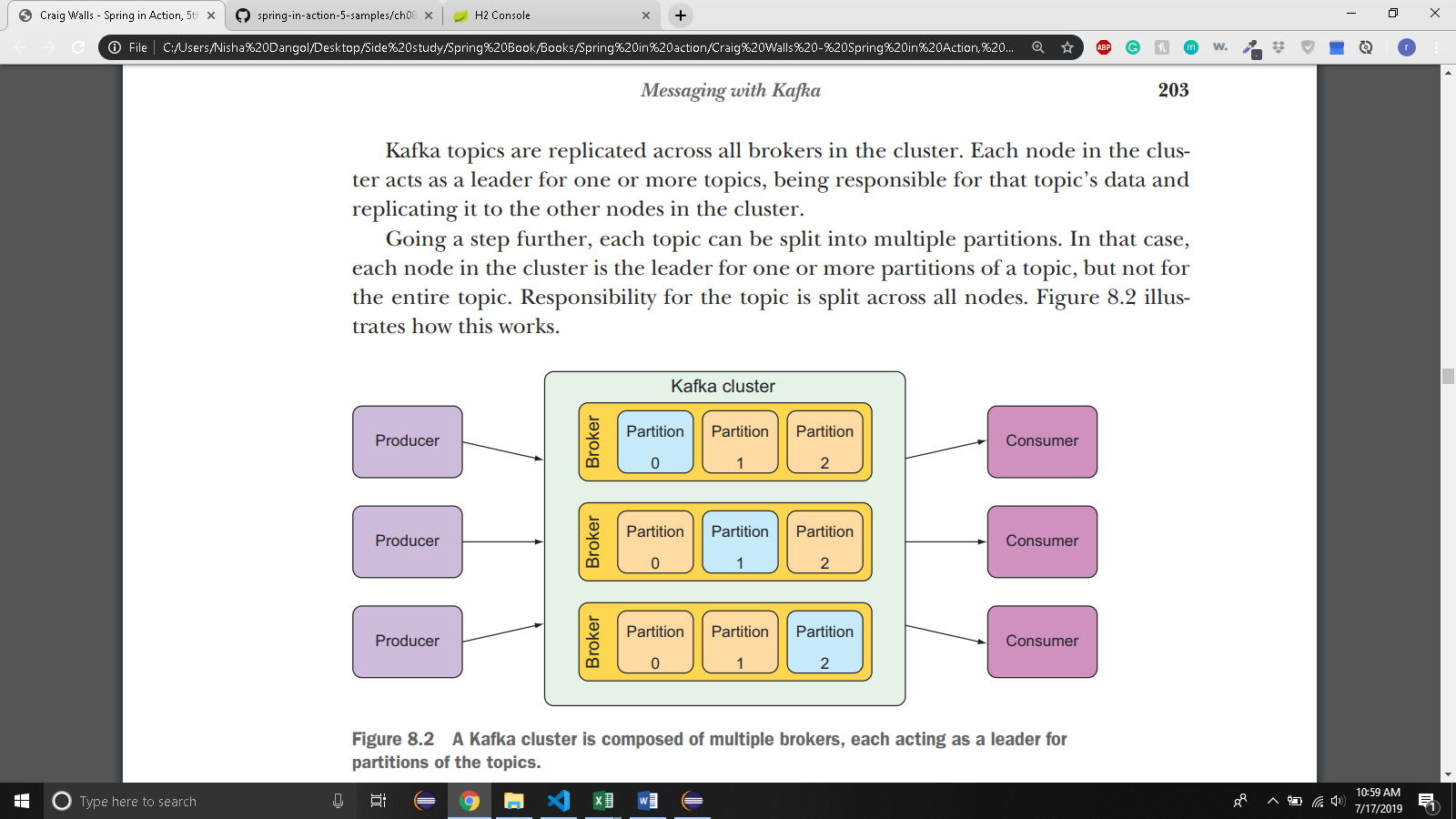
public void receiveOrder(Order order) {

ui.displayOrder(order);

}

}

* 8.3 MESSAGING WITH KAFKA
  + Newest messaging option
  + Designed to run in a cluster, affording great scalability. And by partitioning its topics across all instances in the cluster, it’s very resilient.



* + SETTING UP SPRING FOR KAFKA MESSAGING
    - Add dependency:

<**dependency**>

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

<**artifactId**>spring-kafka</**artifactId**>

</**dependency**>

* + - KafkaTemplate defaults to work with a Kafka broker on localhost, listening on port 9092.
    - In production, you need different host and port.
    - spring.kafka.bootstrap-servers sets the location of one or more kafka servers:

spring:

kafka:

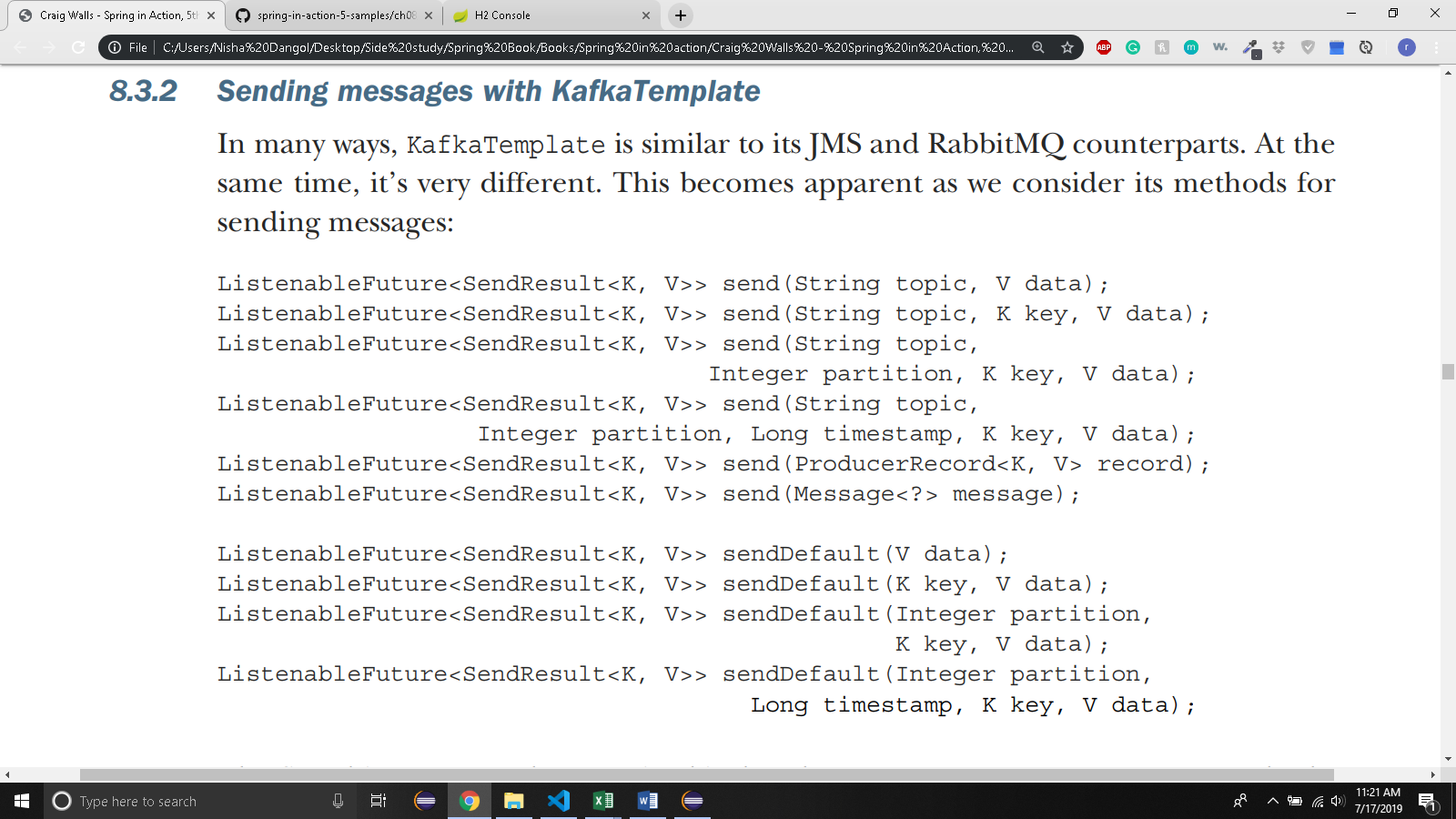
bootstrap-servers:

-kafka.tacocloud.com:9092

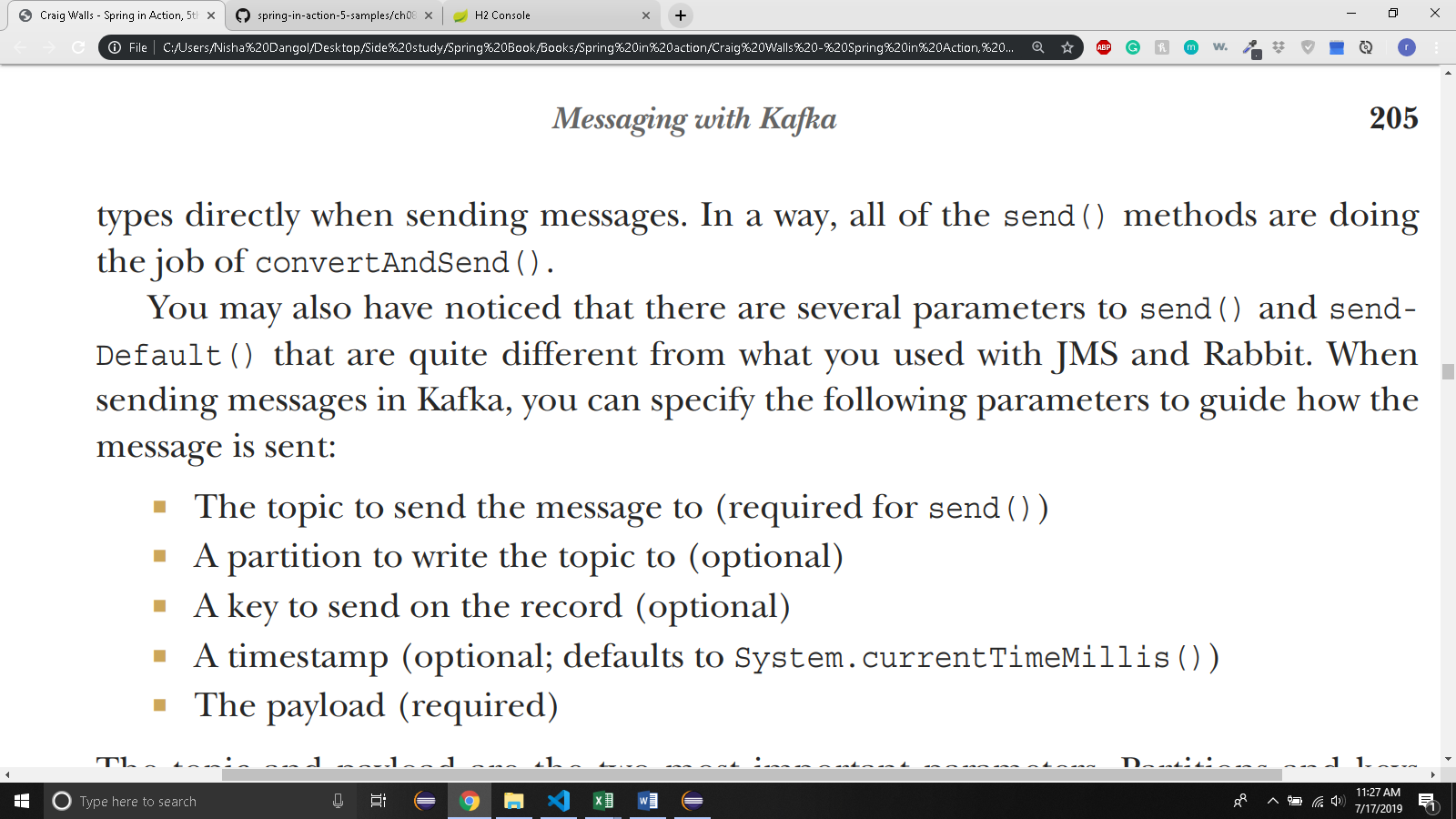
-kafka.tacocloud.com: 9093

-kafka.tacocloud.com: 9094

* + - Let’s start by sending Order objects to kafka using kafkaTemplate
* 8.3.2 SEDING MESSAGES WITH KAFKA TEMPLATE



* Kafka doesn’t have convertAndSend() because it can deal with domain types directly when sending messages.



*@Service*

public class KafkaOrderMessagingService implements OrderMessagingService {

private KafkaTemplate<String,Order> kafkaTemplate;

*@Autowired*

public KafkaOrderMessagingService(KafkaTemplate<String,Order> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

/\* sends order to the topic named 'tacocloud.orders.topic' \*/

*@Override*

public void sendOrder(Order order)

kafkaTemplate.send("tacocloud.orders.topic",order);

}

}

* + You can set a default topic as well:

spring:

kafka:

template:

default-topic: tacocloud.orders.topic

* + Then in sendOrder() method, you can use sendDefault()

*@Override*

public void sendOrder(Order order)

kafkaTemplate.sendDefault(order);

}

* 8.3.3 WRITING KAFKA LISTENERS:
  + Only way to consume messages from a kafka topic using Spring is to write a message listener.

*@Component*

public class OrderListener {

private KitchenUI ui;

public OrderListener(KitchenUI ui) {

this.ui = ui;

}

*@KafkaListener*(topics = "tacocloud.orders.topic")

public void handle(Order order) {

ui.displayOrder(order);

}

}

* + This method is invoked when message arrives in the topic names ‘tacocloud.orders.topic’.
  + You can also add ConsumerRecord to log the partition and timestamp of the message:

*@KafkaListener*(topics = "tacocloud.orders.topic")

public void handle(Order order, ConsumerRecord<String,Order> record) {

***log***.info("Received from partition{} with timestamp{}",record.partition(), record.timestamp());

ui.displayOrder(order);

}

* + Message payload is also available via ConsumerRecord.value()
  + This means you can ask for Order through ConsumerRecord objects instead of asking for it directly as a parameter to handle().