Messaging & Integration

Why do we need messaging?

- Classic Web Apps

- Implement a Photo Gallery
- Two parts:

Upload Picture

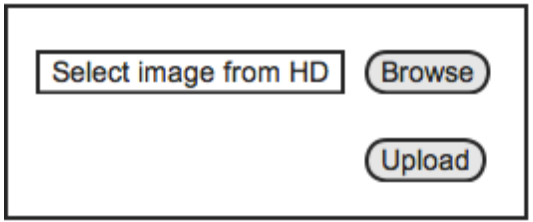
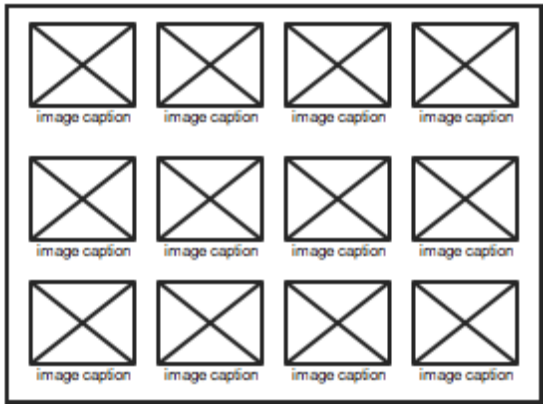


Image Gallery



New requirements arrive

- The Product Owner
  - Can we also notify the user friends when she uploads a new image?
- The Social Media Guru
  - We need to give points to users for each picture upload
  - and post uploads to Twitter
- The Sysadmin
  - Dumb! You're delivering full size images! The bandwidth bill has tripled!
- The Developer in the other team
  - I need to call your Java stuff but from Python
  - I need to call your PHP stuff but from Python
- The User
  - I don't want to wait till your app resizes my image!

Code evolution – Pseduo Code

- First Implementation

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  image_handler:do_upload(ReqData:get_file()),
  ok.
```

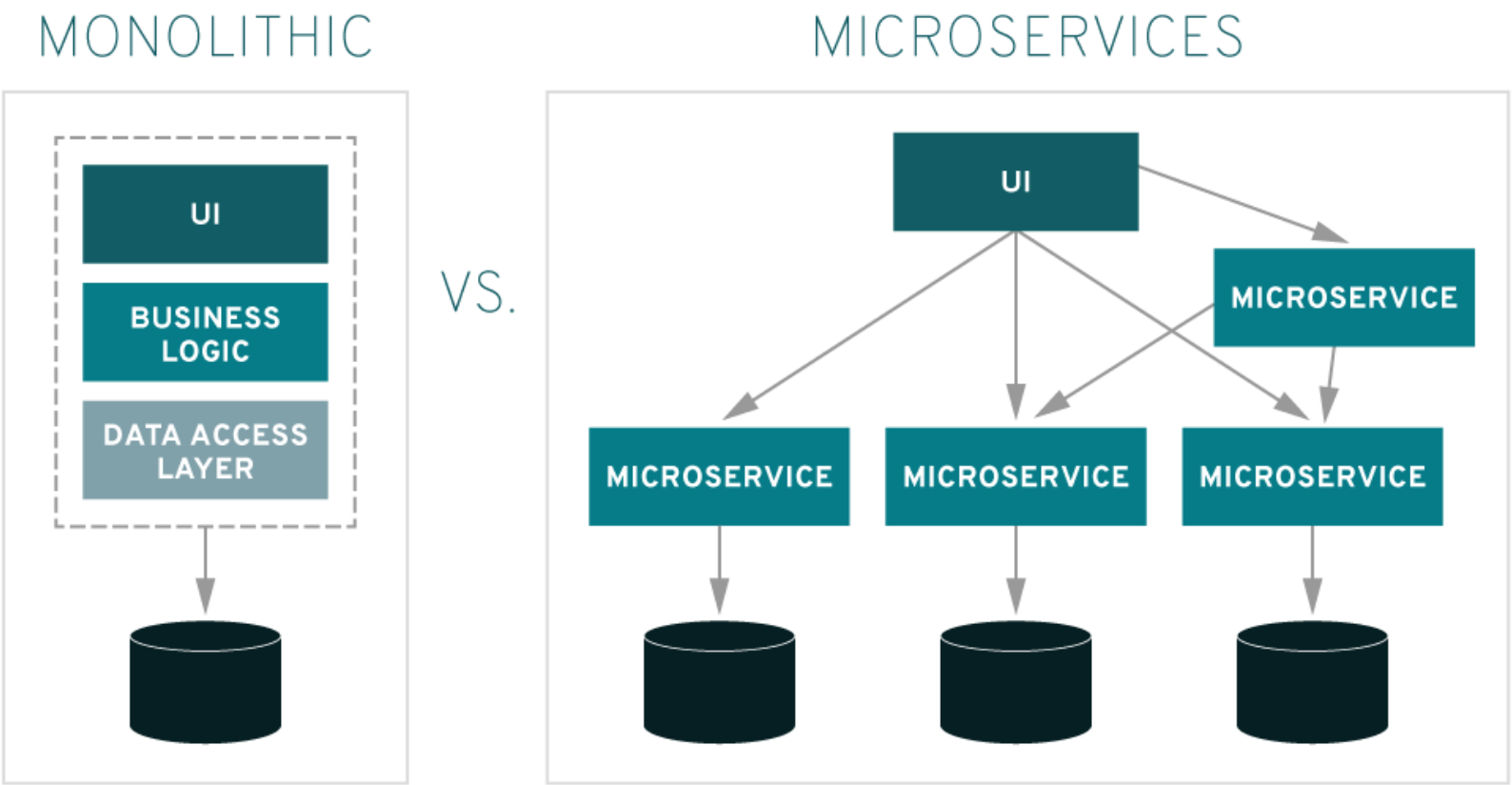
- More Implmentations

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  resize_image(Image),
  notify_friends(ReqData:get_user()),
  add_points_to_user(ReqData:get_user()),
  tweet_new_image(User, Image),
  ok.
```

Scale?

- What if
  - We need to speed up image conversion
  - User notifications sent by email
  - Stop tweeting about new images
  - Resize in different formats
  - Swap Language / Technology (No Down Time)

Microservices



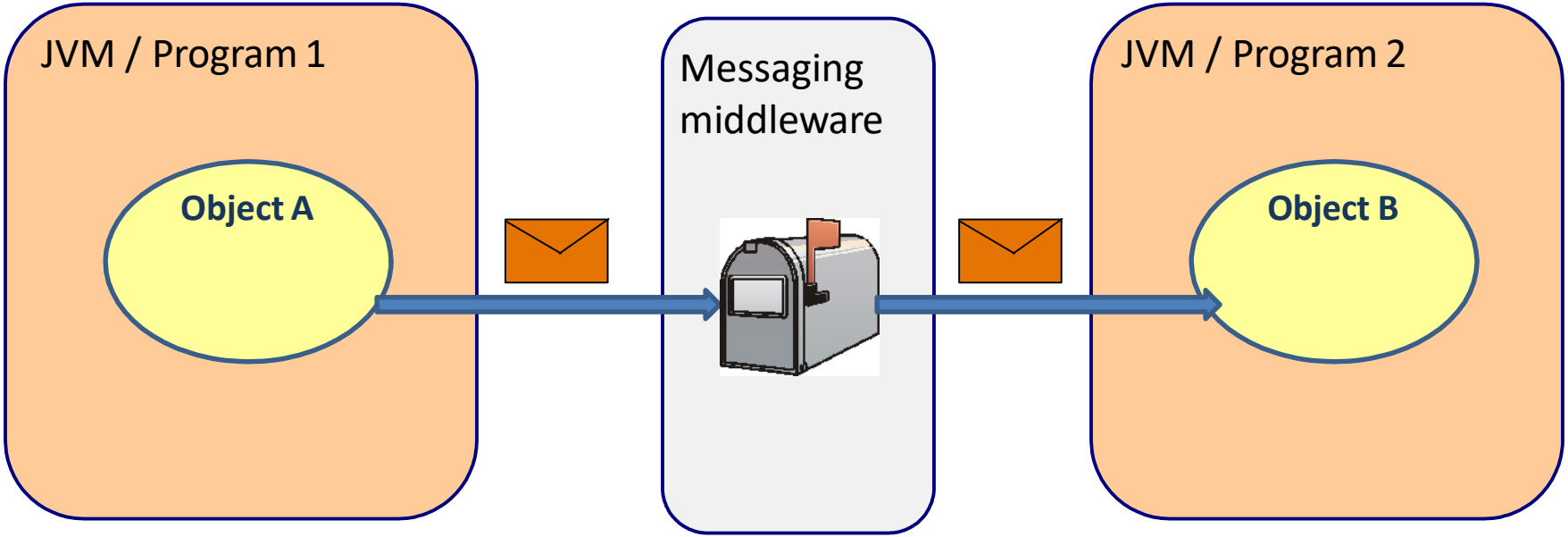
Microservices communication

- Remote invocation is synchronous
  - The caller waits for a response
  - Similar to a phone call

- **Messaging is asynchronous**
- Just send a message, no waiting for a reply
  - Similar to an email, message is stored in “middleware”
  - Can be picked up at any point

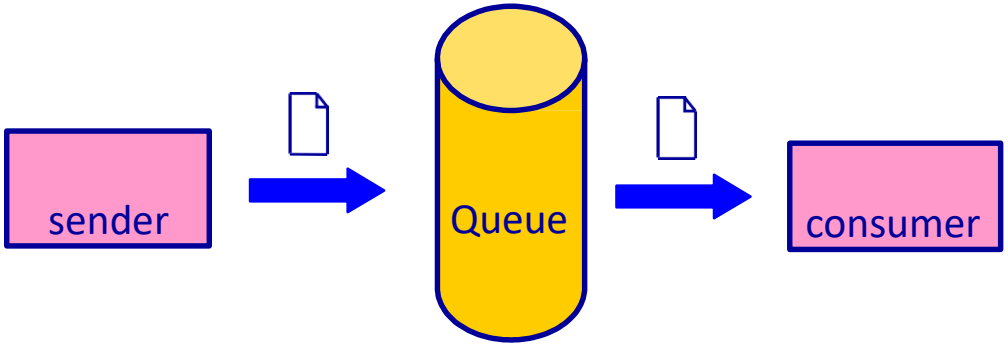
Visually

- 3 programs, on different host machines



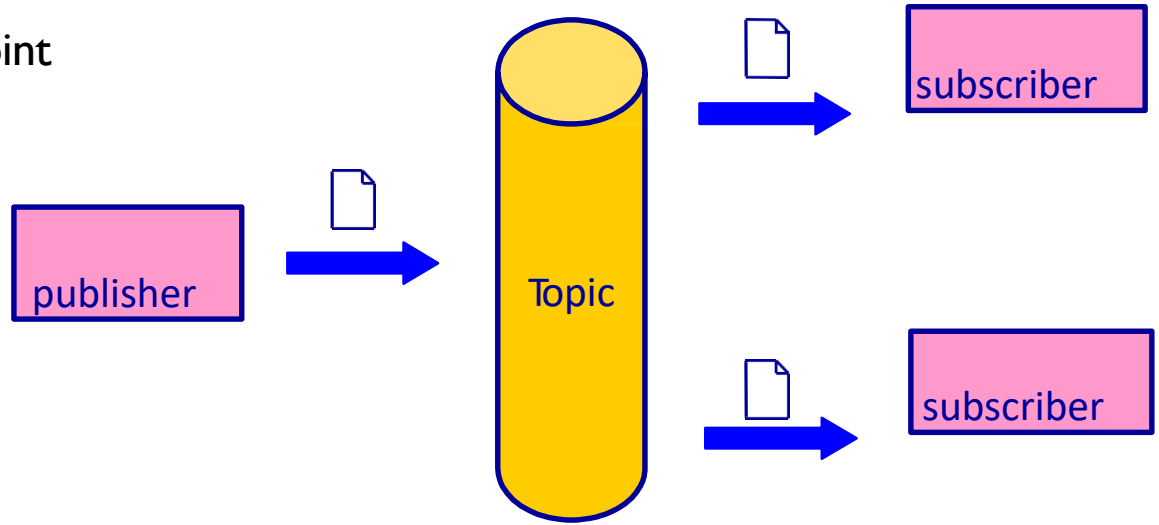
Model by Prof. Rene de Jong

PTP or Pub/Sub



Point-To-Point

Publish / Subscribe



Models by Prof. Rene de Jong

Messaging Implementation – Pseduo Code

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  Msg = #msg{user = ReqData:get_user(), image = Image},
  publish_message('new_image', Msg).

%% friends notifier
on('new_image', Msg) ->
  notify_friends(Msg.user, Msg.image).

%% points manager
on('new_image', Msg) ->
  add_points(Msg.user, 'new_image').

%% resizer
on('new_image', Msg) ->
  resize_image(Msg.image).
```

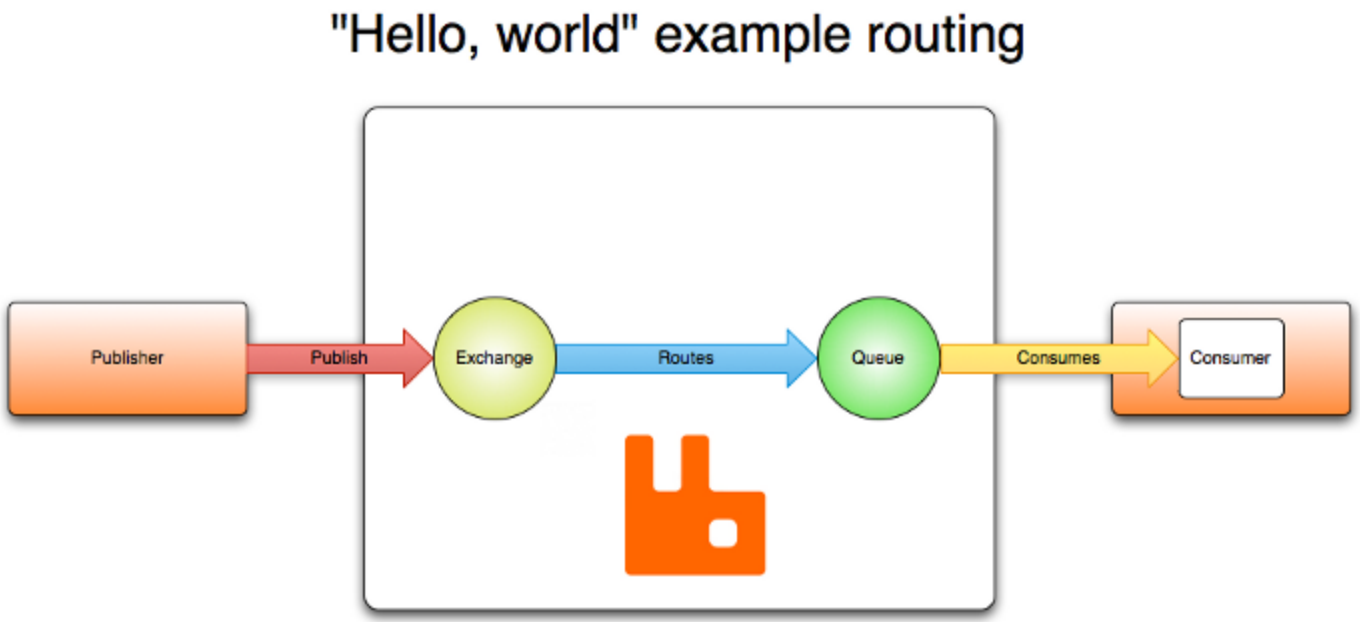
RabbitMQ

- **RabbitMQ** is a popular message-oriented middleware server using the **AMQP** protocol
  - Plugin support for STOMP, MQTT, and many others
  - Written in Erlang, client libraries for all major langs.
  - 2010 acquired by **SpringSource** (division of VMware)
  - Source code released under Mozilla Public License
  - User of RabbitMQ
    - Instagram, Indeed.com, MailBox App

AMQP Terminology

- **Producers**: send messages
- **Consumers**: receive messages
- **Broker**: the middleware server
- **Queue**: where messages are stored on the broker
- **Exchange**: what receives the messages on the broker and routes them to queues

Basic Flow


"Hello, world" example routing
Publisher
Publish
Exchange
Routes
Queue
Consumes
Consumer

Binding and Routing Key

- Bindings
  - Each Queue should bind with an Exchange with a *routing key*
  - a link between a queue and an exchange.

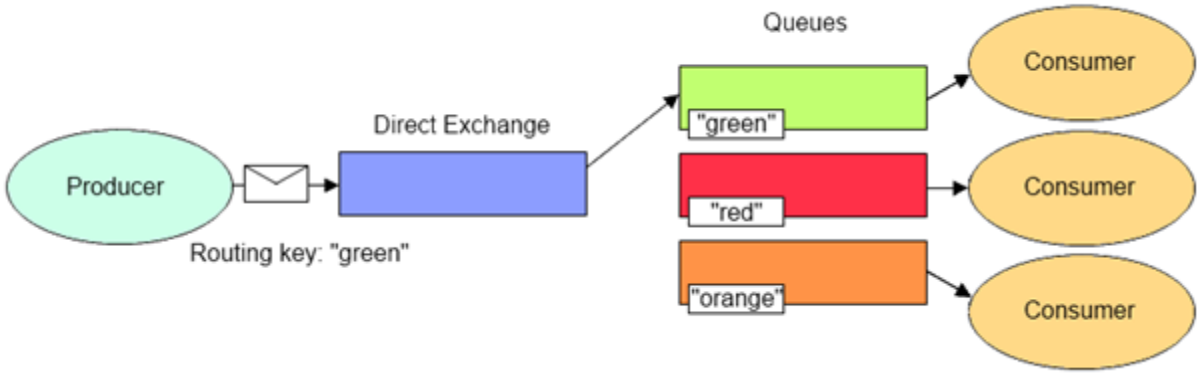
- Routing key
  - String of characters
  - a key that the exchange looks at to decide how to route the message to queues.
  - like an *address* for the message.

Exchange

- Default
  - Direct exchange with no name
  - Routing key equals QueueName
- Direct
  - Messages would only be routed when there is queue bonded with Routing Key
- Fanout
  - Messages would be routed to all queues bounded irrespective of routing key
- Topic
  - We can use regular expressions for routing key
- Header
  - Least used
  - Matches against header properties instead of routing key

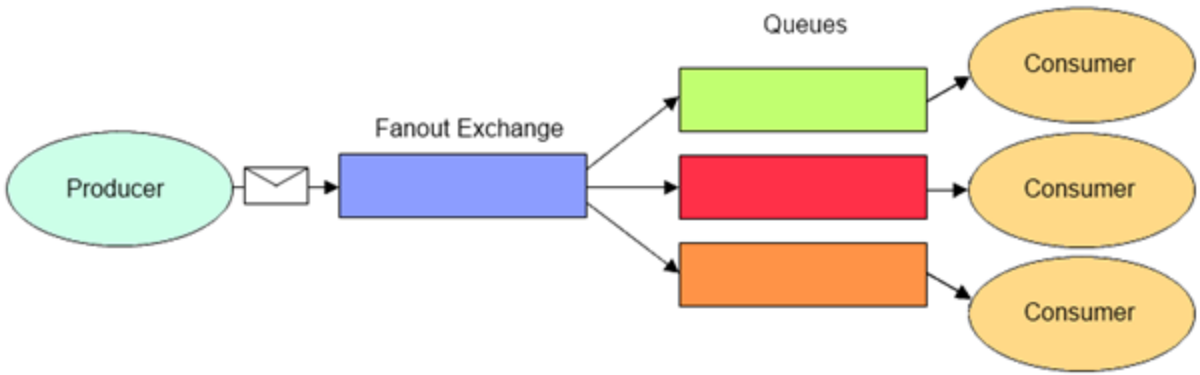
Direct Exchanges

- Routes messages with a routing key equal to the routing key declared by the binding queue.



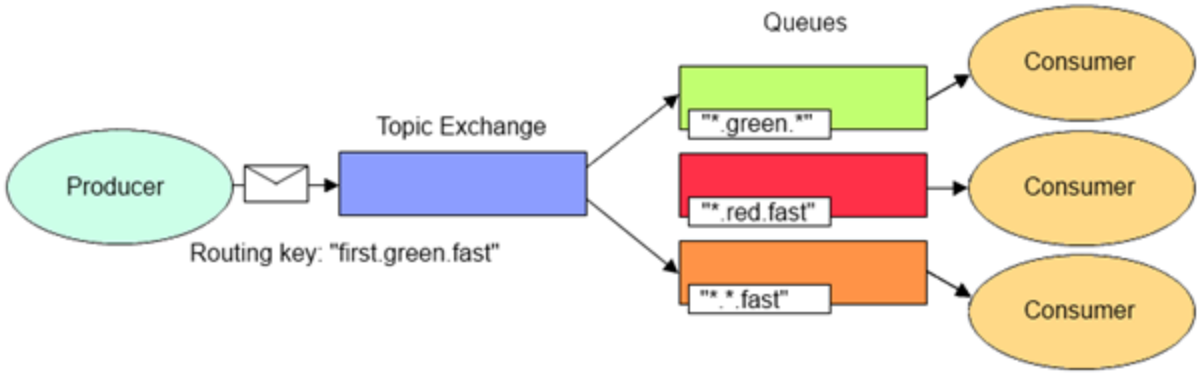
Fanout Exchanges

- Routes messages to all bound queues indiscriminately. If a routing key is provided, it will simply be ignored.



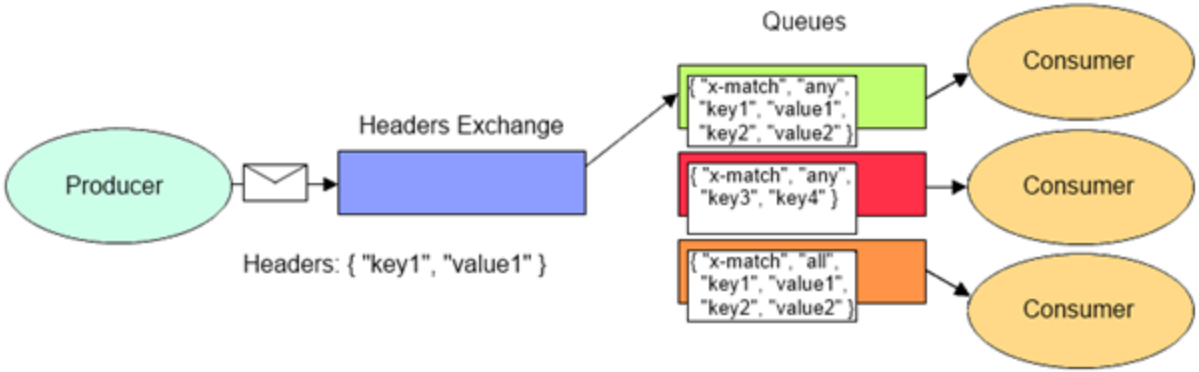
Topic Exchanges

- Routes messages to queues whose routing key matches all, or a portion of a routing key



Headers Exchanges

- Routes messages based upon a matching of message headers to the expected headers specified by the binding queue.



RabbitMQ config

- Exchanges and Queues
  - Created with the Broker API by producer/consumer
  - Configure on the Message Broker

- **Only need to create a queue**
  - There is a default Exchange (does direct delivery)
  - Give the name of the queue and it send it there

Hello World: Spring Boot Sender

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Spring boot automatically configures a RabbitTemplate bean for this dependency

```
package edu.mum.cs544.message;

import org.springframework.amqp.core.Queue;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {

  @Bean
  public Queue hello() {
    return new Queue("hello");
  }

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Only thing we need to add / configure is a Queue bean to be created on the broker

```
package edu.mum.cs544.message;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class Sender implements CommandLineRunner {
      @Autowired
      private RabbitTemplate template;

      @Override
      public void run(String... args) throws Exception {
            String queue = "hello";
            String msg = "Hello World!";
            template.convertAndSend(queue, msg);
            System.out.println("Sent: " + msg +" to: " + queue);
      }

}
```

Send to our queue

Hello World: Spring Boot Receiver

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Separate Receiver application has same dependency

```
package edu.mum.cs544.message;

import org.springframework.amqp.core.Queue;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {

 @Bean
 public Queue hello() {
  return new Queue("hello");
 }

 public static void main(String[] args) {
  SpringApplication.run(Application.class, args);
 }
}
```

Again just configure the queue

```
package edu.mum.cs544.message;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
@RabbitListener(queues = "hello")
public class Receiver {

        @RabbitHandler
        public void receive(String msg) {
                System.out.println("Received: " + msg);
        }
}
```

Our queue

@Rabbit annotations register our Listener and Handler

Optional application.properties

- These are the **default values**
  - Leaving them off gives the same result
  - Both for sender and receiver project
  - Only need to specify them if different values needed

application.properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.rabbitmq.port=5672
```

Sending Objects

- In these slides we've only sent Strings
  - You can send Objects that implement **Serializable**
  - As long as it is the **exact same class** on both sides
  - Including it being in the same package
    - Fully Qualified Class Name should be the same

Messaging

- Messaging protocols are Asynchronous
- Receiving can be Synchronous / Asynchronous
- Messages can be routed in different ways
- Spring RabbitTemplate makes it easy to send and receive AMQP messages

CS544 EA

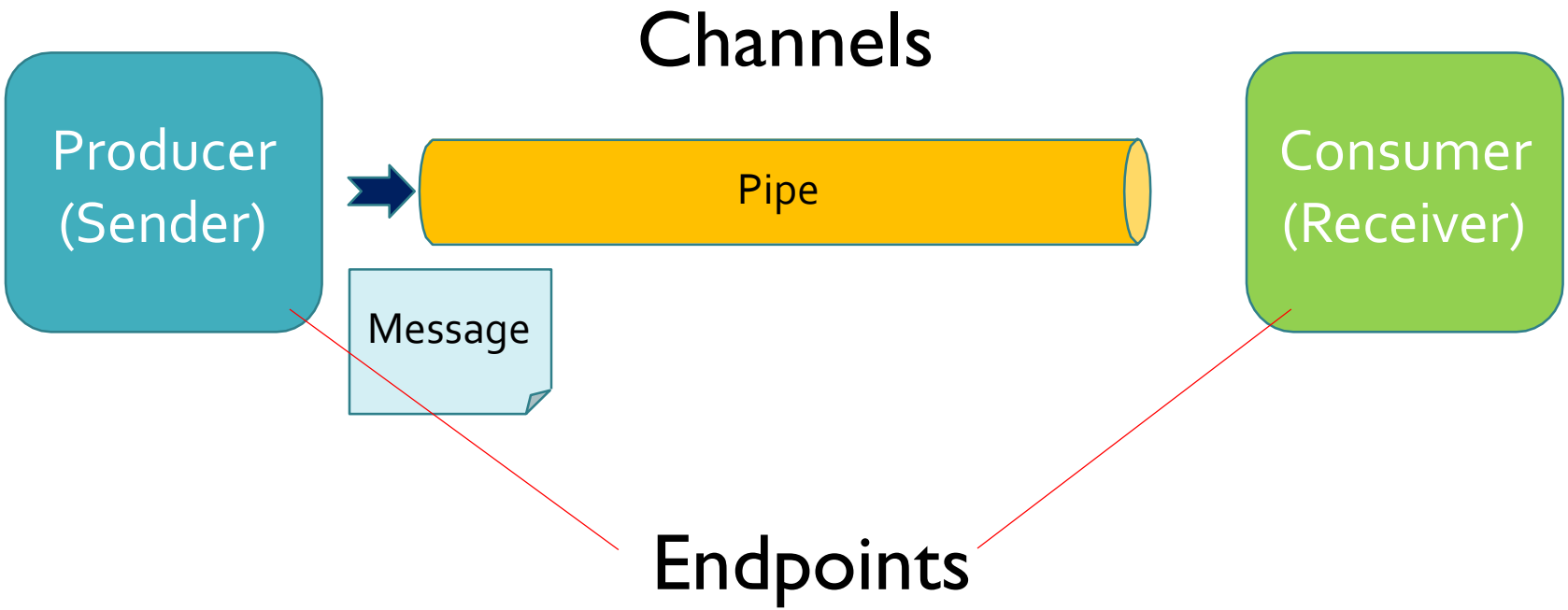
Integration

Spring Integration

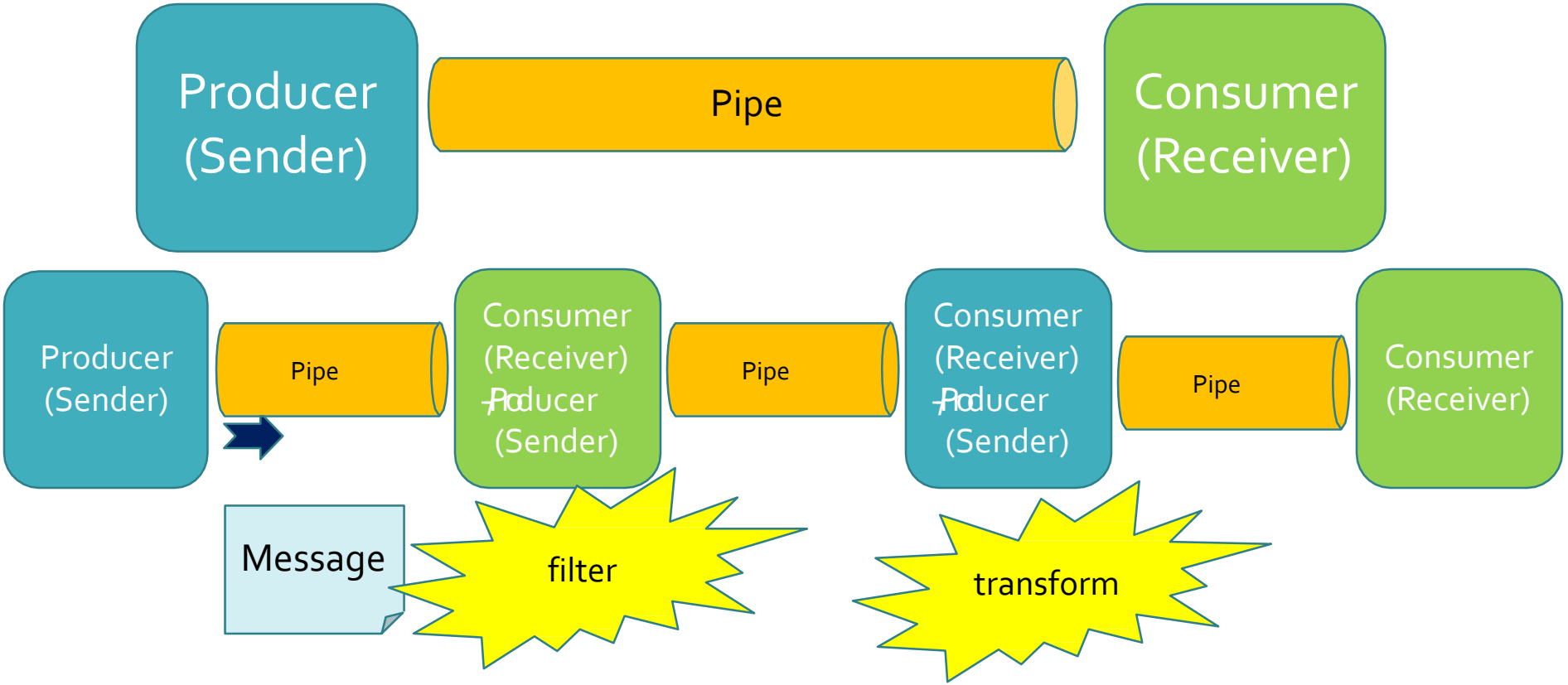
- The Spring Integration project is to Application Integration what Spring MVC is to HTTP.
  - **A thin layer interacting with service methods**

- But messages can come from through any of:
  - File system, shared DB, Remote Call, Messaging middleware
  - Can connect with all of these, and therefore is more abstract

Main Components

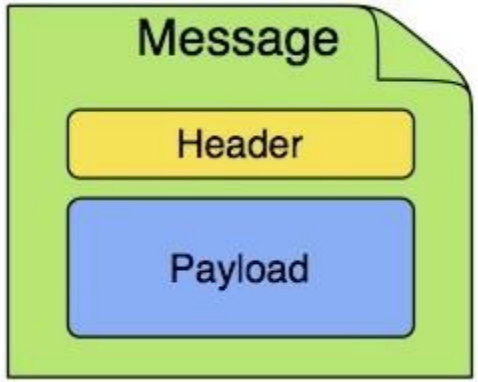


Spring Integration Applications



Message

- In Spring Integration a message is a generic wrapper for **any Java object combined with** metadata (key/value pairs known as headers).
  - Any object can be a message
  - Spring Integration will add headers
  - Unlike the messaging middleware it can come from / go to anywhere



Decoupling

- The whole purpose of messaging is to keep systems **as decoupled as possible, while still** integrating.
  - Each system should be able to work regardless of the protocols, formatting, or implementation details of the messages and the sender / receiver of them.

Message Endpoints

- There are **many types** of endpoints
- Adapters
- Filters
- Transformer
- Enricher
- Service activator
- Gateway

Message Channel

- Two general classifications of message channels
  - Pollable Channel
    - May buffer messages
    - Consumers actively poll to receive messages
    - Only one receiver of a message in the channel
  - Subscribable Channel
    - Messages are delivered to all registered subscribers on message arrival
    - Doesn’t buffer its messages

Service Activators and Adapters

- Service Activators do just what they say
  - Take an incoming message and based on that **make a call to a @Service** method

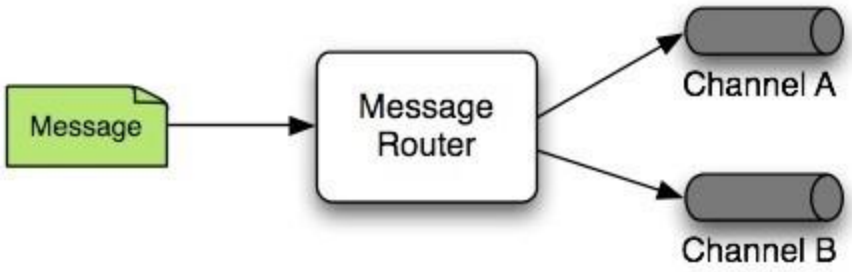
- Channel Adapters **connect to** some other system or transport
  - can be inbound or outbound

Transformer and Filter

- A Message Transformer transforms the message from **one format to another**
  - Example: from JSON to a Java Object
- Message Filters only let **certain messages** through. This can be based on the message payload or headers.
  - Example: email spam blocking uses a filter

Routers

- A router **decides** which channels (if any) a message goes to. Based on payload or headers



Basic Example

```
@SpringBootApplication
@EnableIntegration
public class App {
    public static void main(String[] args) {
        new SpringApplicationBuilder(App.class)
            .web(WebApplicationType.NONE).run(args);
    }

    @Bean
    @InboundChannelAdapter(value = "fileInputChannel", poller = @Poller(fixedDelay = "1000"))
    public MessageSource<File> fileReadingMessageSource() {
        FileReadingMessageSource source = new FileReadingMessageSource();
        source.setDirectory(new File("/tmp"));
        CompositeFileListFilter<File> comp = new CompositeFileListFilter<>();
        comp.addFilter(new SimplePatternFileListFilter("*.person.txt"));
        comp.addFilter(new AcceptOnceFileListFilter<File>());
        source.setFilter(comp);
        return source;
    }

    @Bean
    @Transformer(inputChannel = "fileInputChannel", outputChannel = "processFileChannel")
    public FileToStringTransformer fileToStringTransformer() {
        return new FileToStringTransformer();
    }

    @Bean
    @Transformer(inputChannel = "processFileChannel", outputChannel = "personChannel")
    public JsonToObjectTransformer jsonToObjectTransformer(){
        return new JsonToObjectTransformer(Person.class);
    }
}
```

Boot app without web and no need for application.properties

Beans are automatically made for channels

InboundChannelAdapter reads from file system every 1000 milliseconds

In /tmp directory

Filters for files ending in .person.txt

Filters for files not seen before

Transforms File to String from fileInputChannel to processFileChannel

Transforms JSON to Person from processFileChannel to personChannel

Basic Example

- After the file is filtered and transformed the personChannel ends at a @ServiceActivator

```
package edu.mum.cs544;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.stereotype.Component;

@Component
public class FsEndpoint {
    @Autowired
    private PersonService personService;

    @ServiceActivator(inputChannel = "personChannel")
    public void getPerson(Person p) {
        if (p.getId() == null) {
            personService.add(p);
        } else {
            personService.update(p);
        }
        System.out.println(personService.getAll());
    }
}
```