DDD en TypeScript: Comunicación entre servicios y aplicaciones

Codely Tv Pro

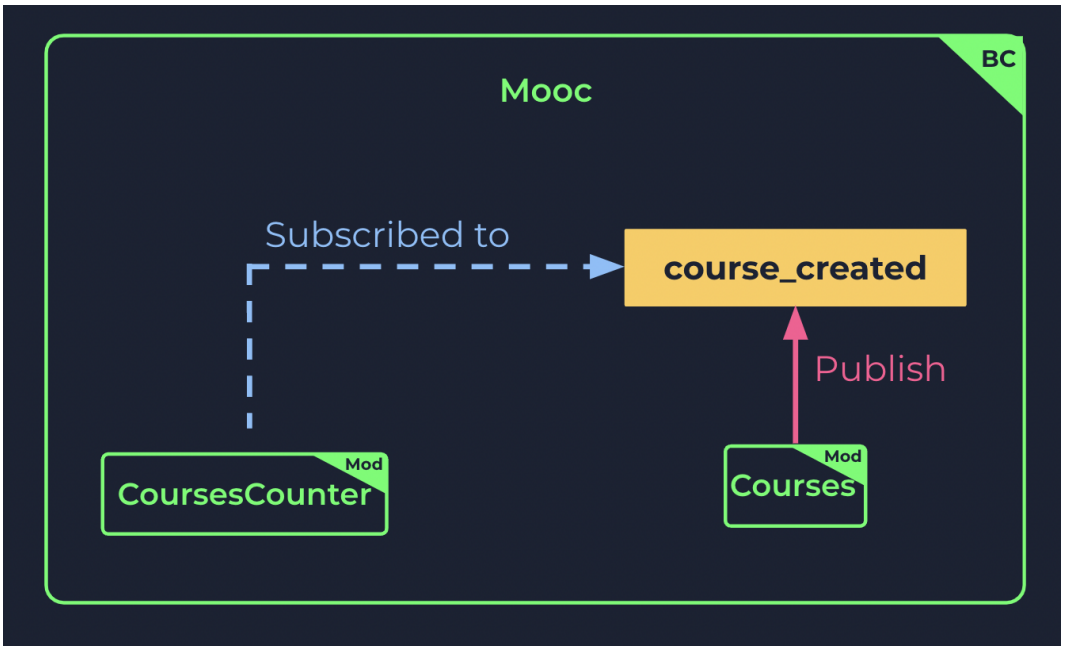
1. 🛰️ Comunica aplicaciones y servicios con TypeScript y DDD

1.1. ✉️ Comunicación con eventos de dominio en aplicaciones DDD

- En el anterior curso vimos el modelado y la arquitectura de la aplicación usando DDD y Arquitectura Hexagonal con TS.
  - Definimos el Bounded Context (BC) de *Mooc* con sus diferentes módulos que obedecen a la lógica de dominio (DDD).
    - Creamos el Módulo de Course que tiene el *Caso de Uso Crear Curso*
      - Tiene la Entidad Course.ts o también llamado agregado
      - Pero el Aggregate como tal, es un elemento Conceptual al que le definimos una única puerta de entrada para que se comuniquen entre todos los elementos que estén dentro de esa bolsita Aggregate que en este caso serán Elementos de DOMAIN pertenecientes a Course.
        - En este caso Course y todos sus Value Objects (VO)

- En este curso queremos implementar un *CoursesCounter*
  - Por lo tanto vamos a crear un Módulo que gestione esa Lógica de Dominio, y esto hacerlo de forma desacoplada al otro módulo de Courses y de forma más optimizada de lo que es la lectura de esos datos
    - Esto es 1 simple count que todo mundo podría pensar, para un simple count tira un count de DB y ya está.
    - Pero, piensa que si tienes 1 única Tabla de Courses, puedes llegar a tener un montón de operaciones a esa única tabla.
      - Como en Twitter, tienes todo el tiempo un montón de Operaciones de Escritura en esa tabla, y en Paralelo operaciones de lectura para 1 Simple Count.

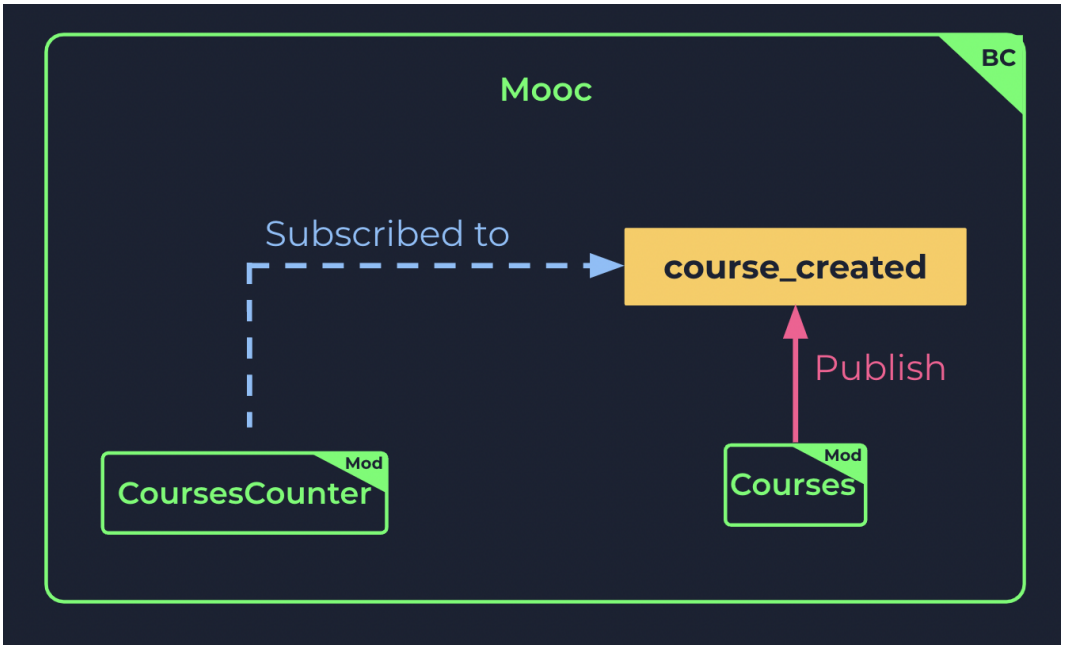
- Entonces, lo que se busca es Quitar esa operación de Lectura que afecta el rendimiento, y lo llevamos a una Proyección, a su única tabla que maneja ese valor de Counter.
  - Justo es esto, proyección mostrar una parte parcial de tus datos optimizada para toda esa cantidad de datos que vas a tener y No estar sobrecargando toda una DB central.
- Esto se va a conseguir con ***Eventos de Dominio***.



- **Eventos de dominio**
  - El Módulo de Courses publicará 1 Evento de Dominio (course_created)
    - Este módulo se despreocupa de la lógica de contar cursos.
  - El Módulo de CoursesCounter es quien se tiene que suscribir al Evento de Dominio *course_created* y aumenta ese Counter en su DB.

  - Este modelo de comunicación ***Publisher-Subscriber*** nos va a ayudar a Mantener un Tipo de Comunicación Desacoplada entre Todos los Módulos que se necesiten, e Incluso entre Otras Aplicaciones.

- En estas primeras lecciones estaremos viendo esta comunicación a nivel de Módulo, y luego iremos viendo: RabbitMQ, otro de equipos, etc.



- Resumen:

  En este curso vamos a dar un salto en complejidad respecto al anterior curso de DDD en Typescript.

  Partiremos de nuestro servicio de Mooc Backend, en el que solo teníamos un contexto con una aplicación, y pasaremos a simular lo que sería un caso de uso de lectura optimizada de datos, generando una proyección de los datos a leer en base a eventos de dominio.

  Vamos a basarnos en el caso de uso de mostrar el contador de cursos. En lugar de hacer un count de la base de datos cada vez que queramos mostrar el nº, vamos a mantener una proyección de ese contador sincronizada en base a los eventos de creación de cursos.

2. Comunicación entre módulos con EventBus asíncrono

2.1. 👇 Definición de eventos de dominio y subscribers

Comenzamos la lección de Eventos de Dominio y lo haremos viendo cómo podemos definir un ***event bus*** asíncrono en memoria utilizando las herramientas que nos da por defecto Node.js. Aunque este event bus no nos permite comunicarnos entre distintos servicios sí que nos ayudará a mantener el principio SOLID de SRP 🕺

Pero antes de empezar con el EventBus es importante revisar cómo hemos definido nuestras clases de dominio e interfaces ***DomainEvent*** , ***DomainEventSubscriber*** y ***EventBus*** , ya que son la base de toda la relación de componentes que existirá en el intercambio de eventos de dominio:

2.1.1. DomainEvent

Nuestro DomainEvent es una *clase abstracta*, ya que no tiene sentido que sea instanciado sin un contexto de qué evento representa. Además de las propiedades típicas ya vistas en otro cursos, disponemos de dos métodos que nos ayudarán a homogeneizar la de/serialización del evento: ***fromPrimitives()*** y ***toPrimitives()***

```
abstract class DomainEvent {
  static EVENT_NAME: string;
  static fromPrimitives: (params: {
    id: string;
    correlationId: string;
    occurredOn: Date;
    attributes: DomainEventAttributes;
  }) => DomainEvent;

  readonly aggregateId: string;
  readonly eventId: string;
  readonly occurredOn: Date;
  readonly eventName: string;

  constructor(eventName: string, aggregateId: string, eventId?: string, occurredOn?: Date) {
    this.aggregateId = aggregateId;
    this.eventId = eventId || Uuid.random().value;
    this.occurredOn = occurredOn || new Date();
    this.eventName = eventName;
  }

  abstract toPrimitives(): DomainEventAttributes;
}
```

2.1.2. DomainEventSubscriber

En los subscribers buscamos que contengan la información de a qué evento se están suscribiendo, para ello añadimos un método *subscribedTo* que devolverá la clase del evento a la que se está suscribiendo. De esta manera, cuando el EventBus registra este subscriber, le basta con preguntarle en qué evento está interesado para avisarle cuando este llegue.
El código quedaría algo tal que así:

```
interface DomainEventSubscriber<T extends DomainEvent> {
  subscribedTo(): Array<DomainEvent>;
  on(domainEvent: T): Promise<void>;
}
```

Pero esto código tiene un problema, la firma del *subscribedTo* nos va a ofrecer las propiedades de métodos de la instancia de *DomainEvent*, y como podemos ver en la definición de la clase *DomainEvent* tenemos dos propiedades estáticas, es decir, dos propiedades de clase que son fundamentales para interactuar con el *EventuBus* : EVENT_NAME y fromPrimitives()

Para solucionar este problema y poder mantener este enfoque en los *DomainEventSubscribers* proponemos crear un nuevo *tipo* que nos ayude a indicar la forma correcta de la firma de este método, le hemos llamado ***DomainEventClass***

```
type DomainEventClass = {
  EVENT_NAME: string;
  fromPrimitives(params: {
    id: string;
    correlationId: string;
    occurredOn: Date;
    attributes: DomainEventAttributes;
  }): DomainEvent;
};
```

Utilizando este tipo, los DomainEventSubscriber quedaría tal que así:

```
interface DomainEventSubscriber<T extends DomainEvent> {
  subscribedTo(): Array<DomainEventClass>;

  on(domainEvent: T): Promise<void>;
}
```

2.1.3. EventBus

La interfaz que vamos a definir para nuestro EventBus es muy sencilla

```
interface EventBus {
  publish(events: Array<DomainEvent>): Promise<void>;
  addSubscribers(subscribers: Array<DomainEventSubscriber<DomainEvent>>): void;
}
```

Tendremos un método publish para publicar eventos, y un método *addSubscribers* que utilizaremos durante el setup del EventBus para añadir los *DomainEventSubscriber* que tengamos definidos. La mejor alternativa sería pasar estos subscribers en el constructor del EventBus, pero, como veremos en siguientes vídeos, esto nos generaría una dependencia circular que nuestro gestor de dependencias no es capaz de solventar.

Por último, añadimos una implementación del EventBus en memoria utilizando la clase ***EventEmitter*** de *Node.js*.

2.2. 🥳 Publicación de eventos de dominio CourseCreated

Vamos a ver cómo se guardan los Eventos de Dominio en el AggregateRoot, además de las implicaciones que tiene la publicación de eventos en nuestros tests y cómo podemos validarlo.

2.2.1. Eventos en el AggregateRoot

Nuestra clase abstracta *AggregateRoot* ahora dispone de un array donde se irán almacenando los ***eventos de dominio*** generados en base a modificaciones sobre nuestros agregados. La función pullDomainEvents será la encargada de devolver esos eventos y resetear el array para no generar eventos duplicados.

```
abstract class AggregateRoot {
  private domainEvents: Array<DomainEvent>;

  constructor() {
    this.domainEvents = [];
  }

  pullDomainEvents(): Array<DomainEvent> {
    const domainEvents = this.domainEvents.slice();
    this.domainEvents = [];

    return domainEvents;
  }

  record(event: DomainEvent): void {
    this.domainEvents.push(event);
  }
}
```

Con este cambio, nuestra clase Course dispone ahora de un método factoría que representa la creación de un nuevo curso, que previamente no existía.

```
static create(id: CourseId, name: CourseName, duration: CourseDuration): Course {
    const course = new Course(id, name, duration);

    course.record(
      new CourseCreatedDomainEvent({
        id: course.id.value,
        duration: course.duration.value,
        name: course.name.value
      })
    );

    return course;
  }
```

Dado que es un curso que no existía, debe generarse el *evento de dominio de* ***creación***, que se guarda dentro del mismo agregado.

2.2.2. Publicación de eventos y test unitario

Para añadir la funcionalidad de publicar el *evento de dominio*, previamente modificaremos el test unitario, de forma que este nos guíe a la hora de hacer los cambios:

```
beforeEach(() => {
  repository = new CourseRepositoryMock();
  eventBus = new EventBusMock();
  creator = new CourseCreator(repository, eventBus);
});

describe('CourseCreator', () => {
  it('should create a valid course', async () => {
    const request = CreateCourseRequestMother.random();
    const course = CourseMother.fromRequest(request);
    const domainEvent = CourseCreatedDomainEventMother.fromCourse(course);

    await creator.run(request);

    repository.assertSaveHaveBeenCalledWith(course);
    eventBus.assertLastPublishedEventIs(domainEvent);
  });
```

Añadimos un EventBusMock que pasamos como dependencia a la clase CourseCreator, y una vez ejecutada la acción, podremos preguntare a este mock si el evento esperado ha sido publicado.

Clase CourseCreator:

```
class CourseCreator {
  constructor(private repository: CourseRepository, private eventBus: EventBus) { }

  async run(request: CreateCourseRequest): Promise<void> {
    const course = Course.create(new CourseId(request.id), new CourseName(request.name), new
CourseDuration(request.duration));
    await this.repository.save(course);
    await this.eventBus.publish(course.pullDomainEvents());
  }
}
```

Estamos pasando como parámetro en el constructor el ***EventBus*** que añadiremos en nuestra definición de *inyección de dependencias* con la implementación del *InMemoryAsyncEventBus*

2.2.3. ✅ Test de aceptación tras añadir nuestro EventBus

En el test de aceptación para el caso de uso de crear un nuevo curso no tendremos ningún cambio tras haber añadido la publicación de eventos de dominio ya que, en nuestra opinión, lo único que debemos comprobar en este caso es que nos devuelva un 201, sin entrar en que se guarde correctamente en BD o que se publique el evento correspondiente

Repositorios de los ejemplos

🐙 typescript-ddd-course
🐙 typescript-ddd-example

2.3. 🤔 Suscripción de eventos y test de aceptación

En esta lección hemos añadido una nueva funcionalidad a través de un nuevo módulo dentro de nuestro contexto de Mooc. Esta funcionalidad consiste en llevar un conteo del número de cursos existentes en la plataforma, de manera que no sea necesario hacer un count en la base de datos de cursos cada vez que queramos mostrar un contador.

Este nuevo ***módulo*** se llama ***CoursesCounter*** y nos va a ayudar a implementar un caso de uso donde se necesita una *suscripción a un evento de dominio*, y veremos cómo hacer un *tests de aceptación* de la funcionalidad.

2.3.1. Subscripción a eventos de dominio

Empezaremos por la clase ***CourseCounterIncrementer***, que será la encargada de incrementar el contador cuando un nuevo curso sea creado:

```
class CoursesCounterIncrementer {
  constructor(private repository: CoursesCounterRepository, private bus: EventBus) {}

  async run(courseId: CourseId) {
    const counter = (await this.repository.search()) || this.initializeCounter();

    counter.increment(courseId);

    await this.repository.save(counter);
    await this.bus.publish(counter.pullDomainEvents());
  }

  private initializeCounter(): CoursesCounter {
    return CoursesCounter.initialize(CoursesCounterId.random());
  }
}
```

Para ejecutar este caso de uso cuando se crea un nuevo curso, creamos un *subscriber* que escuche el evento de *CourseCreatedDomainEvent*

```
class IncrementCoursesCounterOnCourseCreated implements
DomainEventSubscriber<CourseCreatedDomainEvent> {
  constructor(private incrementer: CoursesCounterIncrementer) {}

  subscribedTo(): DomainEventClass[] {
    return [CourseCreatedDomainEvent];
  }

  async on(domainEvent: CourseCreatedDomainEvent) {
    await this.incrementer.run(new CourseId(domainEvent.aggregateId));
  }
}
```

Este subscriber se registrará de forma automática en el ***EventBus*** una vez lo hayamos definido en nuestra inyección de dependencias, añadiéndole la etiqueta ***domainEventSubscriber***:

```
class:
../../../../../Contexts/Mooc/CoursesCounter/application/Increment/IncrementCoursesCounterOnCours
    arguments: ["@Mooc.CoursesCounter.CoursesCounterIncrementer"]
    tags:
      - { name: 'domainEventSubscriber' }
```

2.3.2. Test de aceptación

Primeramente vamos a definir cómo sería nuestro test de aceptación, para que este nos vaya guiando en los nuevos elementos que vamos a necesitar en la parte de testing para darle soporte, nuestro test quedaría tal que así:

- Como es un Test de Aceptación, se lo implementa con Gherkins y Cucumber

```
Scenario: With one course
    Given I send an event to the event bus:
    """
    {
      "data": {
        "id": "c77fa036-cbc7-4414-996b-c6a7a93cae09",
        "type": "course.created",
        "occurred_on": "2019-08-08T08:37:32+00:00",
        "aggregateId": "8c900b20-e04a-4777-9183-32faab6d2fb5",
        "attributes": {
          "name": "DDD en PHP!",
          "duration": "25 hours"
        },
        "meta" : {
          "host": "111.26.06.93"
        }
      }
    }
    """
    When I send a GET request to "/courses-counter"
    Then the response status code should be 200
    And the response content should be:
    """
    {
      "total": 1
    }
    """
```

En este test estamos enviando un *evento de dominio* a través del *EventBus*, este evento debe producir un cambio en el contador, que posteriormente validamos a través de una api rest.

Es importante resaltar que lo estamos definiendo en formato JSON, y ahora mismo nuestro ***InMemoryAsyncEventBus*** solo maneja instancias de DomainEvent. Por lo tanto, vamos a necesitar que nuestro código que da soporte a los tests de cucumber maneje esta conversión.

En realidad esta situación es una necesidad que también tendremos en futuras lecciones, cuando implementemos un ***EventBus*** que no funcione en memoria, si no en Infraestructura Externa (***RabbitMQ***, SQS...), pues esas implementaciones requerirán de una serialización del evento para viajar por el broker de mensajería, y una deserialización para poder ser procesado el evento. Así pues vamos a aprovechar la necesidad

de deserialización del evento que se nos presenta para implementar un mecanismo que nos sirva en futuras lecciones.

2.3.3. 🤔 Deserialización de evento de dominio

Lo que necesitamos es un elemento que, dado un EVENT_TYPE nos indique la clase que debe ser instanciada. Dado que no conocemos todos las clases de eventos de dominio que contiene nuestro código, la forma más sencilla que tenemos para descubrirlos es preguntarle a nuestros *DomainEventSubscriber* cuáles son los eventos que están Escuchando.

De esta manera construimos el ***DomainEventDeserializer*** que dispone de un constructor estático, el cual recibe todos los *DomainEventSubscribers* , y uno por uno va preguntándoles a qué evento están suscritos, y guarda la relación entre el type del evento, y su clase.

```
class DomainEventDeserializer extends Map<string, DomainEventClass> {
  static configure(subscribers: DomainEventSubscribers) {
    const mapping = new DomainEventDeserializer();
    subscribers.items.forEach(subscriber => {
      subscriber.subscribedTo().forEach(mapping.registerEvent.bind(mapping));
    });

    return mapping;
  }

  private registerEvent(domainEvent: DomainEventClass) {
    const eventName = domainEvent.EVENT_NAME;
    this.set(eventName, domainEvent);
  }

}
```

Una vez esto está listo, podemos añadirle a esta clase el método deserialize , el cual, dado un JSON en formato string:

1. Lo parsea
2. Extrae su type para poder recuperar la clase que instancia ese evento
3. Utiliza el método *fromPrimitives* de la clase para instanciar el evento de dominio

```
class DomainEventDeserializer extends Map<string, DomainEventClass> {
  static configure(subscribers: DomainEventSubscribers) {
    const mapping = new DomainEventDeserializer();
    subscribers.items.forEach(subscriber => {
      subscriber.subscribedTo().forEach(mapping.registerEvent.bind(mapping));
    });

    return mapping;
  }

  private registerEvent(domainEvent: DomainEventClass) {
    const eventName = domainEvent.EVENT_NAME;
    this.set(eventName, domainEvent);
  }

  deserialize(event: string) {
    const eventData = JSON.parse(event).data;
    const eventName = eventData.type;
    const eventClass = super.get(eventName);

    if (!eventClass) {
      return;
    }

    return eventClass.fromPrimitives({
      id: eventData.attributes.id,
      attributes: eventData.attributes,
      occurredOn: eventData.occurred_on,
      eventId: eventData.id
    });
  }
}
```

Con el deserializer ya listo, podemos volver a nuestro test de aceptación y añadir el código que implementa los steps de publicación:

```
Given('I send an event to the event bus:', async (event: any) => {
  const domainEvent = deserializer.deserialize(event);

  await eventBus.publish([domainEvent!]);
});

function buildDeserializer() {
  const subscribers = DomainEventSubscribers.from(container);

  return DomainEventDeserializer.configure(subscribers);
}
```

Este código recibe el JSON en formato string que definimos anteriormente, instancia la clase del evento utilizando el deserializer y lo publica en nuestro EventBus en Memoria.

2.4. 🚳 Idempotencia al consumir eventos

Cuando estamos trabajando con eventos de dominio debemos tener siempre en cuenta que los eventos pueden llegar por duplicado en cualquier momento, es por eso que nuestro código debe ser capaz de manejar esa duplicidad.

2.4.1. Definiendo el Caso de Uso

La *idempotencia* a la hora de procesar eventos de dominio debemos *manejarla en los casos de uso*, *ya que estamos hablando de eventos de dominio*, pero también podríamos recibir duplicidad en peticiones por una API Rest, o cualquier otro punto de entrada, y es la lógica de nuestro dominio la que debe manejar que un agregado no ejecute la misma acción dos veces, siendo la acción exactamente la misma.

```
class CoursesCounterIncrementer {
  constructor(private repository: CoursesCounterRepository, private bus: EventBus) {}

  async run(courseId: CourseId) {
    const counter = (await this.repository.search()) || this.initializeCounter();

    if (!counter.hasIncremented(courseId)) {
      counter.increment(courseId);

      await this.repository.save(counter);
      await this.bus.publish(counter.pullDomainEvents());
    }
  }

  private initializeCounter(): CoursesCounter {
    return CoursesCounter.initialize(CoursesCounterId.random());
  }
}
```

Lo primero que haremos cuando se llame al caso de uso será comprobar si ya tenemos un contador creado o si por el contrario debemos 'inicializarlo'. Este contador corresponde con el *agregado* ***CoursesCounter***

Una pieza importante dentro de este agregado es que *guardaremos en un atributo de tipo array los ids de los cursos que ya han sido Procesados*, lo cual nos permitirá ignorar aquellas peticiones que recibamos por eventos duplicados (Ojo! 👀 que no estaremos lanzando ningún error, simplemente no llevaremos a cabo la lógica del caso de uso). En el caso de no haber sido procesado llamaremos al método increment() del agregado, que será el encargado de incrementar internamente el valor del contador y añadir el courseId al array que comentábamos

- La opción de un array de ids es útil con un número reducido de elementos, no obstante, con un número elevado podría interesarnos más delegar esa gestión a un almacenamiento en *Redis*, por ejemplo

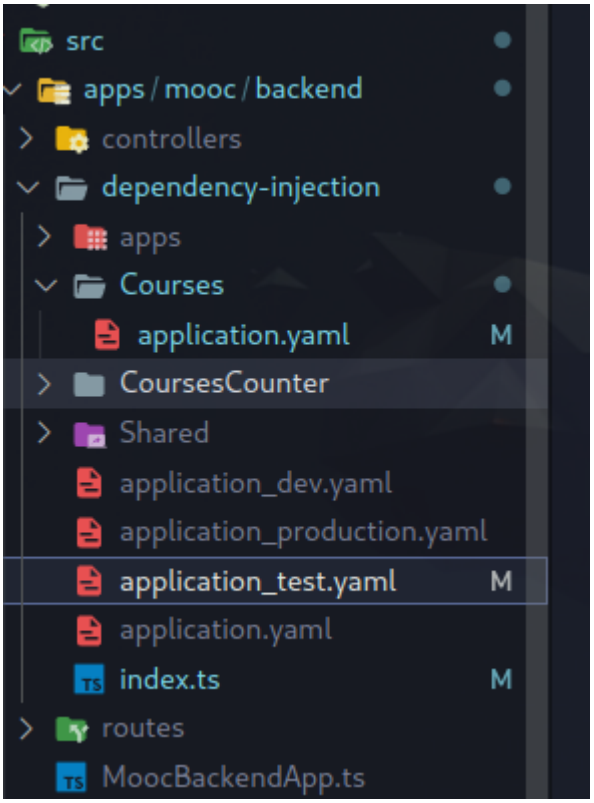
2.4.2. Tests unitarios

```
it('should increment an existing counter', async () => {
    const existingCounter = CoursesCounterMother.random();
    repository.returnOnSearch(existingCounter);
    const courseId = CourseIdMother.random();
    const expected = CoursesCounter.fromPrimitives(existingCounter.toPrimitives());
    expected.increment(courseId);
    const expectedEvent =
CoursesCounterIncrementedDomainEventMother.fromCourseCounter(expected);

    await incrementer.run(courseId);

    repository.assertLastCoursesCounterSaved(expected);
    eventBus.assertLastPublishedEventIs(expectedEvent);
  });

  it('should not increment an already incremented counter', async () => {
    const existingCounter = CoursesCounterMother.random();
    repository.returnOnSearch(existingCounter);
    const courseId = existingCounter.existingCourses[0];

    await incrementer.run(courseId);

    repository.assertNotSave();
  });
```

En los tests para este *caso de uso* debemos tener en cuenta que en la comunicación con los distintos colaboradores estaremos asumiendo que se están llamando con los parámetros que deberíamos.

El punto de partida de cada test será la generación del evento de dominio, y este evento *será lanzado directamente al subscriber* (sin que pase por el Bus), que lo inicializaremos en el setUp() de esta suite de tests.

TODO: Ver en q me da error, leccion 3 de esa seccion



3. 🐇 Comunicación entre Bounded Context con RabbitMQ

3.1. 👽 Publicar eventos de dominio en RabbitMQ

De cara a poder levantar más aplicaciones dentro de nuestro contexto, vamos a buscar un *EventBus* que salga de la ejecución en memoria, y se vaya a una infraestructura externa donde *múltiples Workers puedan Publicar y Consumir eventos*. El camino que hemos optado por tomar es implementar RabbitMQ, un Message Broker que sirve como sistema de colas.

- Es altamente recomendable en este punto volver a ver el contenido del curso de *Comunicación entre [Micro]servicios* para ver todos los conceptos relacionados con la comunicación con colas.

Siguiendo con nuestro caso de uso, el productor será el módulo de cursos, el cual producirá eventos que irán a un exchange de tipo topic.
Este exchange definirá el routing que tomarán los eventos, bindeando en este caso ***course_created*** con la cola de ***increment_courses_counter_on_course_created***. Cada cola se define por evento y suscriptor, por lo que todos los eventos que publique el productor se irán distribuyendo directamente por mediación del exchange.
Al final del flujo encontramos el consumidor, que en este caso será el subscriber *incrementCoursesCounter**On**CourseCreated*. Veremos más adelante cómo implementar la inserción nuevos productores y consumidores en nuestro sistema de colas.

3.1.1. Conociendo a nuestro productor 👨‍🌾

Clase ***RabbitMqEventBus***:

```
class RabbitMQEventBus implements EventBus {
  private connection: RabbitMQConnection;
  private exchange: string;

  constructor(params: { connection: RabbitMQConnection }) {
    this.connection = params.connection;
    this.exchange = 'amq.topic';
  }

  async publish(events: Array<DomainEvent>): Promise<void> {
    for (const event of events) {
      const routingKey = event.eventName;
      const content = this.serialize(event);
      const options = this.options(event);
      const exchange = this.exchange;

      await this.connection.publish({ routingKey, content, options, exchange });
    }
  }
```

```
  private options(event: DomainEvent) {
    return {
      messageId: event.eventId,
      contentType: 'application/json',
      contentEncoding: 'utf-8'
    };
  }

  private serialize(event: DomainEvent): Buffer {
    const eventPrimitives = {
      data: {
        id: event.eventId,
        type: event.eventName,
        occurred_on: event.occurredOn.toISOString(),
        attributes: event.toPrimitives()
      }
    };

    return Buffer.from(JSON.stringify(eventPrimitives));
  }
}
```

Un primer detalle importante de esta implementación es la ***RabbitMQConnection*** que recibe por *constructor*, esta conexión tira de la librería de bajo nivel **amqplib**

A la hora de publicar los eventos de dominio, lo haremos en el exchange (no sobre las colas directamente) que será quien conecte el evento (por la routingKey) a la cola (donde se define la bindingKey)
Por cada evento que nos llegue, lo que estaremos enviando será un json que contiene el evento serializado, la routingKey (nosotros hemos definido que será el nombre del evento) y el ***messageId*** que en nuestro caso corresponderá con el *propio id del evento*. Además será necesario especificar otros parámetros como el content type y el encoding de lo que estamos enviando.
Como veis la publicación de eventos en RabbitMQ no esconde mucha más magia, no obstante aún debemos trabajar en la gestión de errores en el momento de publicar ¡Y eso lo veremos en el próximo video!

El test publica 1 evento y aquí lo vemos

- username: guest
- password: guest

3.2. 🐰 Gestión de errores al publicar eventos de dominio

Un tema importante ahora que gestionamos las acciones “derivadas” a través de la publicación de eventos es que estos siempre se publiquen para que puedan ser posteriormente consumidos por sus *correspondientes casos de uso suscritos*. Pero al llevar a cabo este proceso mediante un sistema de colas como RabbitMQ nos encontramos con que no tenemos una garantía absoluta de que la publicación se lleve a cabo, por lo que se hace necesario contar con un sistema de Fallback para que, en caso de fallo, no perdamos dichos eventos

Lo que haremos será publicar en Mongo aquellos eventos que por alguna razón no hayan podido hacerlo mediante RabbitMQ, para esto vamos a añadir un colaborador a nuestra implementación de RabbitMQ, el cual se encargará de salvar estos eventos en mongo.

Clase RabbitMQEventBus:

```
class RabbitMQEventBus implements EventBus {
  private failoverPublisher: DomainEventFailoverPublisher;
  private connection: RabbitMQConnection;
  private exchange: string;

  constructor(params: {
    failoverPublisher: DomainEventFailoverPublisher;
    connection: RabbitMQConnection;
    exchange: string;
  }) {
    const { failoverPublisher, connection, exchange } = params;
    this.failoverPublisher = failoverPublisher;
    this.connection = connection;
    this.exchange = exchange;
  }

  async publish(events: Array<DomainEvent>): Promise<void> {
    for (const event of events) {
      try {
        const routingKey = event.eventName;
        const content = this.toBuffer(event);
        const options = this.options(event);
```

```
        await this.connection.publish({ exchange: this.exchange, routingKey, content, options
});
      } catch (error: any) {
        await this.failoverPublisher.publish(event);
      }
    }
  }

  private options(event: DomainEvent) {
    return {
      messageId: event.eventId,
      contentType: 'application/json',
      contentEncoding: 'utf-8'
    };
  }

  private toBuffer(event: DomainEvent): Buffer {
    const eventPrimitives = DomainEventJsonSerializer.serialize(event);

    return Buffer.from(eventPrimitives);
  }
}
```

El refactor del eventBus será bastante sencillo: englobaremos la publicación en el exchange dentro de un bloque try-catch de modo que si falla, lo publique desde el publisher de Mongo.

Clase ***DomainEventFailoverPublisher***

```
class DomainEventFailoverPublisher {
  static collectionName = 'DomainEvents';

  constructor(private _client: Promise<MongoClient>, private deserializer:
DomainEventDeserializer) {}

  protected async collection(): Promise<Collection> {
    return (await this._client).db().collection(DomainEventFailoverPublisher.collectionName);
  }

  async publish(event: DomainEvent): Promise<void> {
    const collection = await this.collection();

    const eventSerialized = DomainEventJsonSerializer.serialize(event);
    const options = { upsert: true };
    const update = { $set: { eventId: event.eventId, event: eventSerialized } };

    await collection.updateOne({ eventId: event.eventId }, update, options);
  }
```

```
  async consume(): Promise<Array<DomainEvent>> {
    const collection = await this.collection();
    const documents = await collection.find().limit(200).toArray();

    const events = documents.map(document => this.deserializer.deserialize(document.event));

    return events.filter(Boolean) as Array<DomainEvent>;
  }
}
```

El ***DomainEventFailoverPublisher*** no implementa ninguna interfaz de dominio, ya que es un *colaborador* que va a existir únicamente en Infraestructura, y por lo tanto no necesitamos definir un lenguaje de dominio que defina este componente.

Su método publish recibirá eventos de dominio, y este se encargará de transformarlos para almacenarlos de la forma que considere más óptima. Nosotros hemos optado por almacenar el evento serializado en string, pero dejando la propiedad ***eventId*** disponible para hacer búsquedas.

Con este componente ya listo, lo que haríamos sería definir un proceso que, de forma recurrente, llamase al método consume, el cual lee de base de datos los 200 primeros eventos *y los Deserializa a eventos de dominio*. Con esta información, dicho proceso podría publicar de nuevo estos eventos a través de RabbitMQ, en este punto podrían pasar dos cosas:

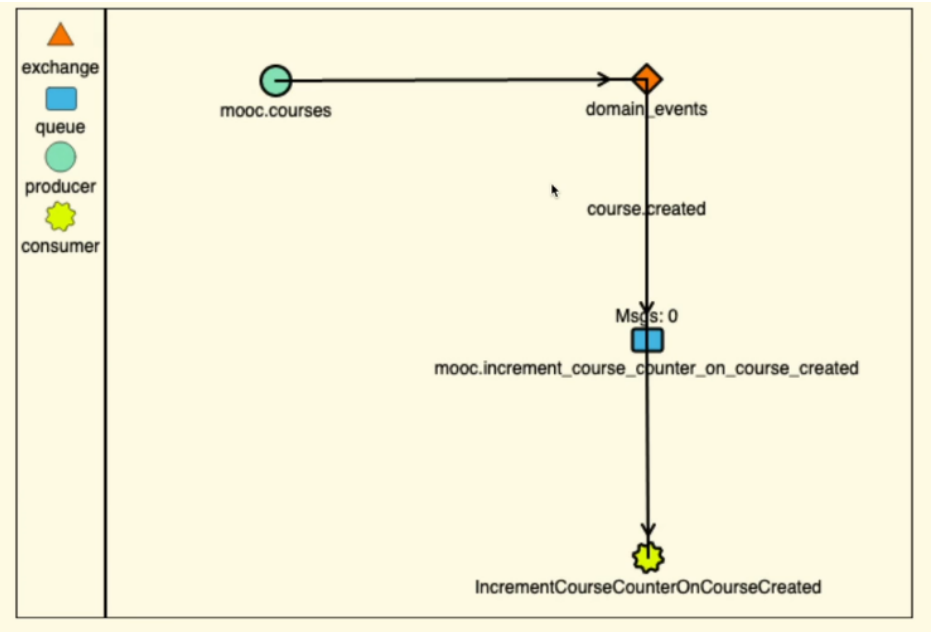
1. Que la publicación volviese a fallar: El RabbitMQ volvería a llamar a este failover publisher para almacenar el evento. Este evento todavía existe en la colección de mongo, pero dado que el método publish hace un *update a la colección*, no se vería modificada la colección añadiendo un nuevo elemento, y tampoco daría ningún error.
2. Que la publicación termine correctamente: En este caso, el proceso debería borrar de la colección de mongo los eventos publicados, utilizando como filtro de borrado la propiedad *eventId* de los eventos publicados.

4. ⚡ Consume Eventos desde RabbitMQ

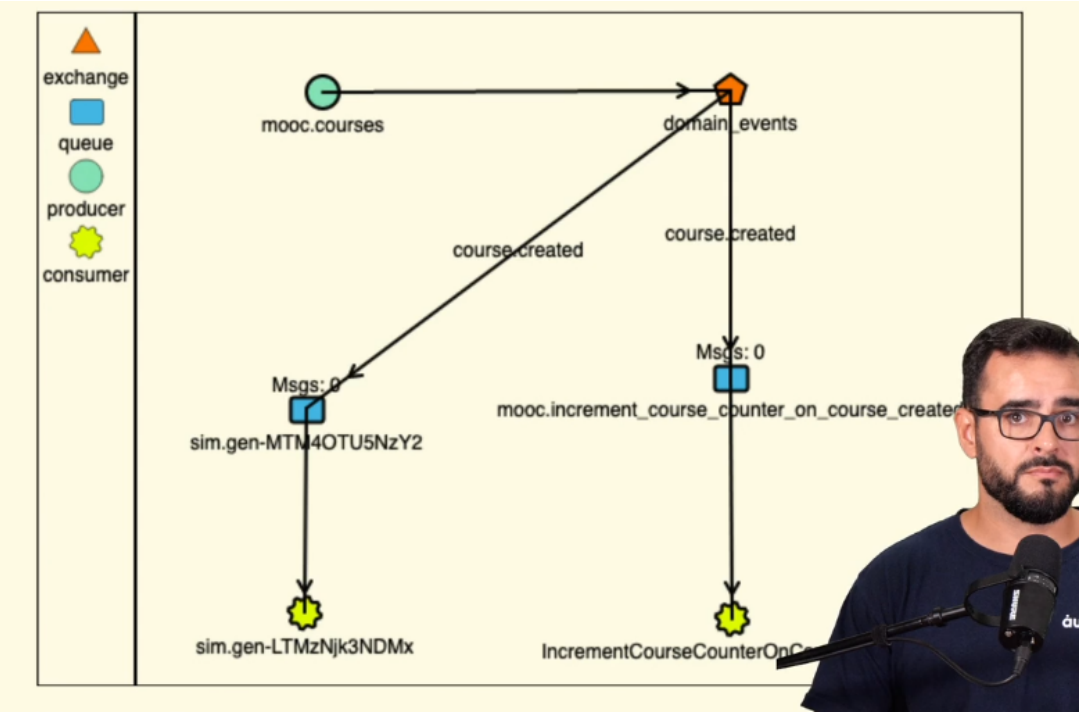
4.1. 🎱 Generar la configuración de RabbitMQ

Antes de decidir cómo suscribirnos a los mensajes que estamos enviando, debemos definir la Topología de nuestras Queue/Colas en RabbitMQ. Lo hacemos con el *RabbitMQ Simulator*

- Tenemos el Producer (mooc.courses) que está Emitiendo eventos de dominio.
  - Lo vamos a Emitir a 1 Exchange de Tipo Topic, que vamos a crear específico para nuestro Context/BC que se llama *domain_events*
    - Lo que vamos a hacer es que, por cada Subscriber que tengamos vamos a crear 1 Queue.
      - Así, cada Subscriber va a tener disponible en SU Queue, los mensajes que le interesan, de forma que Pueda Gestionar de forma Independiente cuando se hace ACK de ese mensaje, cuantas veces quiere consumirlo o cualquier cosa que necesite.
  - Entonces el mooc.courses es el Producer que Emite el Evento de *course_created*
    - A través de Exchange lo enrutamos a la Queue que es específica para el Consumidor/Subscriber que en este caso es el ***IncrementCourseCounterOnCourseCreated***
      - Es este *Subscriber* quien toma los eventos de la Queue y los procesa.



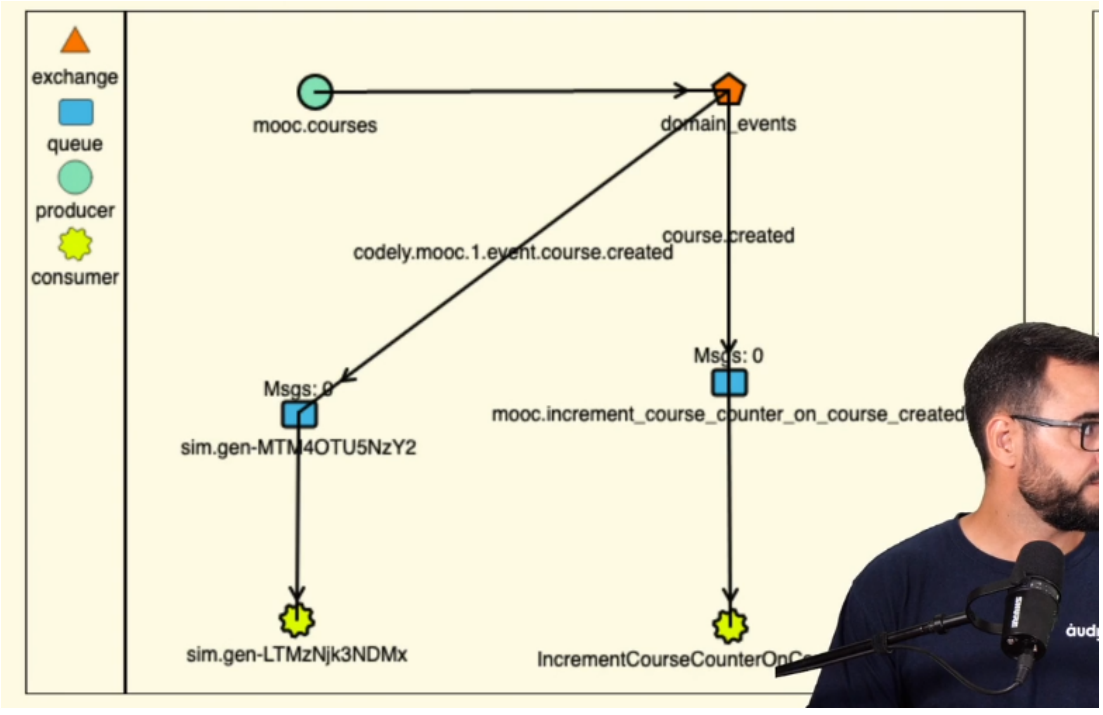
- En este punto, si viene Otro Subscriber de cualquier otro Contexto/BC que necesita escuchar ese Evento *course_created*
  - Lo que se haría es Generar 1 Nueva *Queue* para este Subscriber
    - En donde este nuevo Subscriber estaría Enlazado con su Propia Queue, y esta Queue tendría el *Mismo RoutingKey* que tiene la otra Queue del otro Subscriber porque les interesa el Mismo Evento.
      - Esto es interesante porque cada Consumidor trabaja con su propia Queue, así evitamos crear cruces de competición entre consumidores.
      - Y si hay que Reencolar en algún momento porque ha fallado esto es más fácil porque cada consumidor solo se hace cargo de los eventos que le interesan.
        - En este caso estamos diciendo que solo les interesan los Eventos que sean por *course_created*
    - Por poner 1 Ejemplo:
      - Podría haberse generado 1 error incrementando el Counter, por lo tanto devuelves ese Mensaje/Evento a la Queue para intentarlo de nuevo.
      - Pero el Módulo de Notificaciones SÍ que ha podido enviar la notificación, por lo tanto este Consumidor Elimina ese Mensaje (hace ACK) y NO se ve afectado por los reintentos que tenga el Contador de Cursos.



- A nivel de ***RuotingKey*** , vemos el *course_created* que es lo que en Code ponemos en los Eventos el EVENT_NAME, pues a la final nos estamos interesando Solo por estos Eventos
  - Este routingKey tiene ciertas carencias en cuanto a su nomenclatura porque limita a no saber qué otros eventos produce ese Producer de eventos que en este caso es el mooc.courses
    - Por eso tenemos otras Nomenclaturas como la siguiente:

      `codely.mooc.1.event.course.created`

      - 1ro va el Nombre de la Empresa, puede ser internos de la empresa o el nombre de otros colaboradores que pudieran estar lanzando eventos.
      - El Nombre del Context/BC
      - La Versión del Evento
      - El Tipo de Mensaje, en este caso Event, que también podría ser un comando u otra cosa.
      - La Entity/Agregado (donde se emite/registra el evento)
      - Finalmente, la Acción que se ha producido. (created)



  - Con esta nomenclatura tiene mayor flexibilidad de usar esos *wildCard* (*.course.*) para obtener sólo los eventos que me interesan. En este caso con *.course.*, solo me interesan los eventos del Module Course.

4.1.1. Configuración y Creación del Exchange 🔶

Vamos a ver cómo podemos automatizar el proceso de creación de nuestro exchange y las diferentes colas que nos permitan dirigir los eventos a todos sus subscriptores tal y como ya hemos visto

Configuración de RabbitMQ en el docker-compose:

```
rabbitmq:
  image: 'rabbitmq:3.8-management'
  ports:
    - 5672:5672
    - 15672:15672
```

En primer lugar vemos la configuración que utilizaremos para levantar RabbitMQ con su panel de administración web (para lo cual le estamos indicando el -management en la imagen), especificando los puertos para publicación y acceso al panel web.

Clase ***RabbitMqConfigurer*** (Resultado final):

```
class RabbitMQConfigurer {
  constructor(private connection: RabbitMqConnection, private queueNameFormatter:
RabbitMQqueueFormatter) {}

  async configure(params: { exchange: string; subscribers:
Array<DomainEventSubscriber<DomainEvent>> }): Promise<void> {
    await this.connection.exchange({ name: params.exchange });

    for (const subscriber of params.subscribers) {
      await this.addQueue(subscriber, params.exchange);
    }
  }

  private async addQueue(subscriber: DomainEventSubscriber<DomainEvent>, exchange: string) {
    const routingKeys = subscriber.subscribedTo().map(event => event.EVENT_NAME);
    const queue = this.queueNameFormatter.format(subscriber.constructor.name);

    await this.connection.queue({ routingKeys, name: queue, exchange });
  }
}
```

Desde el método configure lo que haremos simplemente será la declaración del exchange y de las colas. Es importante que el exchangeName lo reciba precisamente el configure() y no el constructor

porque esto nos permitirá utilizar la misma instancia y ejecutar este método por cada Bounded Context
Al declarar un nuevo exchange le estaremos indicando que sea de tipo Topic (como comentábamos en la lección anterior) y que además sea durable, es decir, no queremos que se borre salvo que nosotros se lo indiquemos. La ventaja que nos ofrece utilizar la librería de bajo nivel de AMQP es que si intentamos crear un exchange o una cola ya existentes no nos generará un error, simplemente no hará nada.

4.1.2. Declarando colas

Clase ***RabbitMqConnection***:

```
async queue(params: { exchange: string; name: string; routingKeys: string[] }) {
    const durable = true;
    const exclusive = false;
    const autoDelete = false;

    await this.channel?.assertQueue(params.name, {
      exclusive,
      durable,
      autoDelete
    });
    for (const routingKey of params.routingKeys) {
      await this.channel!.bindQueue(params.name, params.exchange, routingKey);
    }
  }
```

En el caso de la declaración de colas, lo primero que haremos por cada una de ellas será definir el nombre utilizando la clase ***RabbitMqQueueFormatter*** que hemos creado para tener un "estándar" en la nomenclatura.

Al igual que en el exchange, declararemos que las colas sean durables (dado el uso que queremos darle a RabbitMQ, no nos interesa que sean elementos temporales que se borren lejos de nuestro control). Una vez declarada, debemos *bindear cada cola a los eventos de dominio*.

Una vez tenemos toda la configuración lista, podemos ejecutarlo lanzando un comando de npm desde la consola:

npm run command:mooc:rabbitmq

Añadiremos este comando en nuestra pipeline de despliegue para que se ejecute cada vez que hagamos deploy 🚀

4.1.3. Test

Test `RabbitMQEventBus.test.ts`:

```
it('should publish events to RabbitMQ', async () => {
      const eventBus = new RabbitMQEventBus({ failoverPublisher, connection, exchange });
      const event = DomainEventDummyMother.random();

      await configurer.configure({ exchange, subscribers: [dummySubscriber] });

      await eventBus.publish([event]);
    });
```

En el test de esta pieza de la infraestructura es donde lo que vamos a comprobar que jústamente la infraestructura real se está creando y podremos verlo directamente en el panel web de RabbitMQ (Por ello también borraremos estos valores antes de cada ejecución de los tests)

4.2. 🌬️ Consumir eventos desde RabbitMQ

A la hora de consumir eventos es muy importante tener en cuenta que un mismo evento puede ser publicado en múltiples colas, y que una misma aplicación puede estar subscrito a todas esas colas. Por ejemplo, si publicásemos el evento ***DummyEvent*** a dos colas distintas (queue_1 y queue_2, y nuestra aplicación *MoocBackend* estuviese suscrita a esas dos colas, el mismo evento llegaría a MoocBackend dos veces, una a través de queue_1 y otra a través de queue_2.

Es importante tener esto muy en cuenta, ya que los mensajes de distintas colas afectan a subscribers distintos, y estos deben ser procesados de forma independiente para:

1. No duplicar las acciones que puede ejecutar un subscriber a la hora de procesar un evento recibido.
2. Tener una gestión correcta de los errores a la hora de consumir.

4.2.1. Empezar a consumir

Lo 1º que haremos será informar a nuestro EventBus de cuales son los subscribers que debe tener en cuenta:

```
class RabbitMQEventBus implements EventBus {
  private failoverPublisher: DomainEventFailoverPublisher;
  private connection: RabbitMqConnection;
  private exchange: string;
  private queueNameFormatter: RabbitMQqueueFormatter;

  constructor(params: {
    failoverPublisher: DomainEventFailoverPublisher;
    connection: RabbitMqConnection;
    exchange: string;
    queueNameFormatter: RabbitMQqueueFormatter;
  }) {
    const { failoverPublisher, connection, exchange } = params;
    this.failoverPublisher = failoverPublisher;
    this.connection = connection;
    this.exchange = exchange;
    this.queueNameFormatter = params.queueNameFormatter;
  }
```

```
  async addSubscribers(subscribers: DomainEventSubscribers): Promise<void> {
    const deserializer = DomainEventDeserializer.configure(subscribers);

    for (const subscriber of subscribers.items) {
      const queueName = this.queueNameFormatter.format(subscriber);
      const rabbitMQConsumer = new RabbitMQConsumer(subscriber, deserializer);

      await this.connection.consume(queueName, rabbitMQConsumer);
    }
  }
```

Para ello implementamos la función addSubscribers que carecía de implementación hasta ahora. Esta función se encargará de configurar el DomainEventDeserializer con los subscribers correspondientes, y utilizar la conexión de RabbitMQ para empezar a consumir eventos.

```
class RabbitMqConnection {
  async consume(queue: string, consumer: RabbitMQConsumer) {
    await this.channel!.consume(queue, (message: ConsumeMessage | null) => {
      if (!message) {
        return;
      }

      const ack = this.getAck(message);
      const noAck = this.getNoAck(message);
      consumer.onMessage({ message, ack, noAck });
    });
  }

  private getAck(message: ConsumeMessage) {
    return () => {
      this.channel!.ack(message);
    };
  }
```

```
  private getNoAck(message: ConsumeMessage) {
    return () => {
      this.channel!.nack(message);
    };
  }
}
```

El método consume del elemento RabbitMqConnection es el encargado de establecer la conexión persistente a través de canal abierto con la instancia de rabbit. Cuando un mensaje es recibido por este canal, se entregará al RabbitMQConsumer que se especificó para la cola de la que se está consumiendo. Junto al mensaje, también le entregaremos los métodos ack y noAck que nos ayudarán a gestionar la persistencia de dicho mensaje en la instancia de RabbitMQ.

```
class RabbitMQConsumer {
  constructor(private subscriber: DomainEventSubscriber<DomainEvent>, private
deserializer: DomainEventDeserializer) { }

  async onMessage(params: { message: ConsumeMessage; ack: Function; noAck:
Function; }) {
    const content = params.message.content.toString();
    const domainEvent = this.deserializer.deserialize(content);

    try {
      await this.subscriber.on(domainEvent);
      params.ack();
    } catch (error) {
      params.noAck();
    }
  }
}
```

Por último, el RabbitMQConsumer será el encargado de coordinar las acciones de:

1. Deserializar el mensaje de RabbitMQ y convertirlo a un DomainEvent
2. Ejecutar el DomainEventSubscriber que corresponda
3. Confirmar el mensaje llamando a la función ack si todo ha ido bien, o devolver el mensaje a RabbitMQ en caso contrario llamando al la función noAck

4.2.2. Testando el consumidor de eventos ✅

Ahora que ya podemos consumir eventos de RabbitMQ, podemos modificar nuestro test que solo publicaba, para poder añadir aserciones que comprueben el ciclo completo:
Clase RabbitMQEventBus.test.ts:

```
it('should consume the events published to RabbitMQ', async () => {
      await configurer.configure({ exchange, subscribers: [dummySubscriber] });
      const eventBus = new RabbitMQEventBus({ failoverPublisher, connection,
exchange, queueNameFormatter });
      await eventBus.addSubscribers(subscribers);
      const event = DomainEventDummyMother.random();

      await eventBus.publish([event]);

      await dummySubscriber.assertConsumedEvents([event]);
    });
```

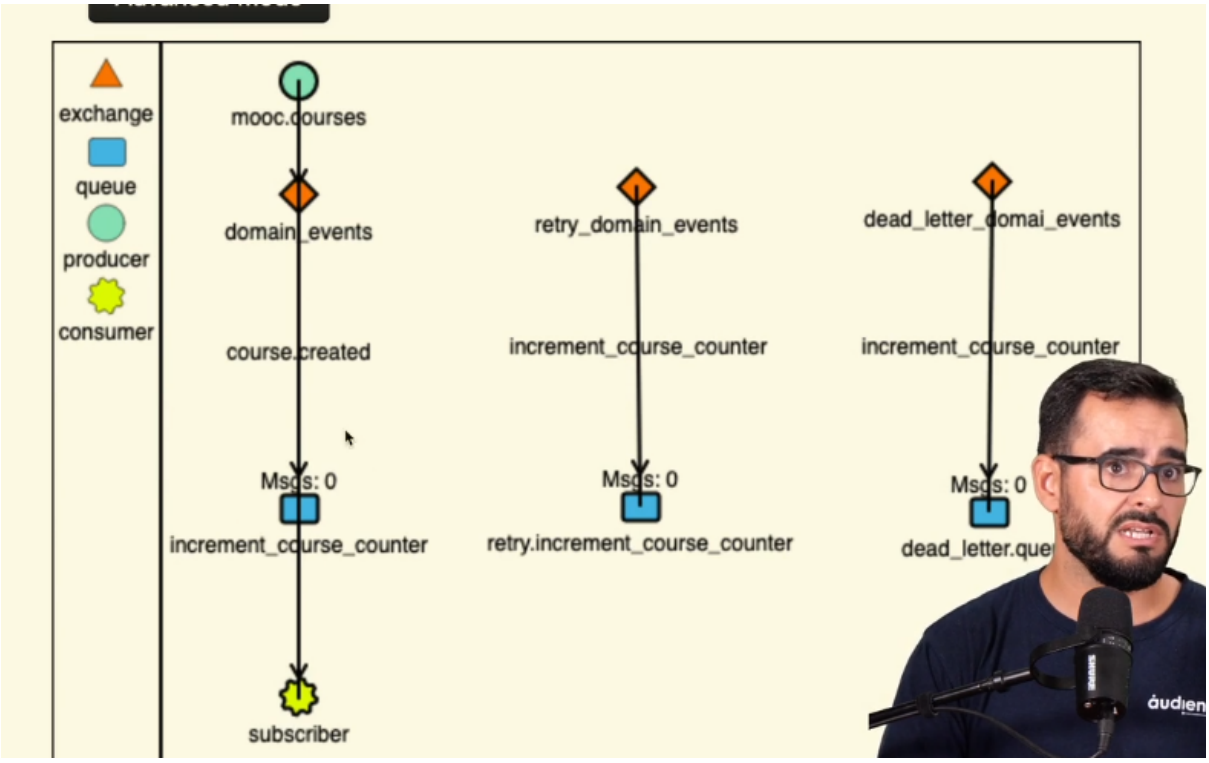
Hemos creado un subscriber dummy, al que le hemos añadido un método para poder preguntarle si ha recibido ciertos eventos. De esta forma no solo estamos comprobando que estamos recibiendo correctamente los mensajes en el subscriber que corresponde, si no que su deserialización a instancias de eventos de dominio de nuestro código funciona a la perfección, ya que la aserción como vemos en el código inferior, se hace comparando directamente entidades de DomainEvent.

4.3. 🙅🏾‍♂️ Gestión de errores al consumir: Colas de Retry y Dead Letter

- Partimos del Schema, y vemos que el 1ro es el Schema que ya teníamos, pero ahora tenemos los 2 de la derecha que representarán la gestión de los reintentos.
  - Cuando falle 1 Message, vamos a coger ese Message y publicarlo en el Exchange de retry
    - Donde modificaremos la cabecera para tener 1 contador de Reintentos y este Exchange retry tendrá 1 binding que utilizando el nombre de la Queue Original, en donde ese Binding irá a 1 Queue que tendrá el Mismo nombre de la Queue Origen con el *Prefijo Retry.*
    - Con esta implementación, ahora, por cada Subscriber ya tenemos 1 Queue para leer todos los mensajes de interés y otra adicional para el tema de los Reintentos
  - Para conseguir que se Reintente el Message utilizamos el mecanismo que tiene RabbitMQ, el del *TTL*
    - Podemos especificar que el Message puede estar en la cola del Retry durante 30s, a los 30s tiene que eliminarse y le configuramos como *Queue de dead_letter* dentro de RabbitMQ, a esa cola de retry, a la Queue Original
      - Por lo tanto, Si falla, ese mensaje terminará en la Queue de Retry gracias al Exchange de Retry (perteneciente a ese Subscriber retry….) y a los 30 segundos el propio RabbitMQ va a volver a Publicar el Mensaje en el Exchange de

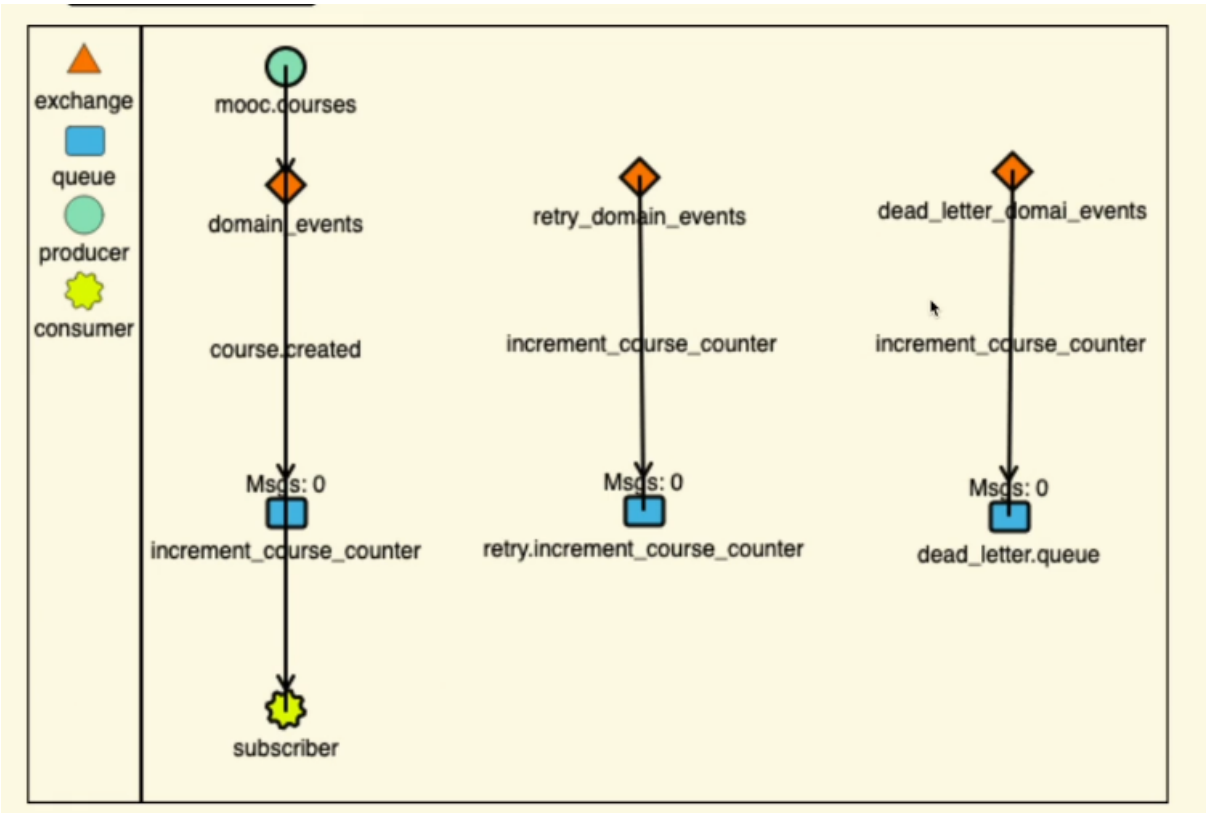
domain_events el mismo mensaje que cayó en el retry. Esto porque establecimos que el dead_letter de esta Queue de Retry, sea la Queue Original.

- Por lo tanto, con esta metodología acabamos volviendo a consumirlo en la Queue original durante X veces/try/intentos.



- Para recapitular, antes tenías el noAck que significa que NO han podido CONSUMIR el message.
  - Ahora, si que se ha podido consumir, lo que ha pasado es que ha Fallado el Subscriber
  - Cuando desde la Queue de Retry se vuelve a volcar sobre el exchange de domain_events , ya el RoutingKey no va a ser el del Evento, sino el de la Queue.
    - Entonces, esa Queue original va a tener 2 RoutingKeys, los Eventos en los que está interesado, y su propio nombre.
      - Su Propio Nombre para cuando le venga del Retry, así el Exchange se lo enviara solo a esa Queue que tenga el mismo name, porque la única que va a estar interesada en ese Evento porque es el Nombre de esa Queue.

- Ahora, si volvemos a consumir (después del try) y vuelve a fallar, incrementamos el contador de try, y llegamos al máximo de intentos permitidos. En este caso, Publicamos al Exchanger del dead_letter y hacemos ACK para eliminarlo. En el Exchanger del dead_letter en donde, de nuevo, tendremos una Queue que se llamará igual que la Queue Original (lo mismo que en el try).
  - Aquí podemos poner 1 Subscriber que nos envíe alertas como, oye, tengo mensajes en la Queue de dead_letter , no se va ha hacer nada con estos mensajes, NO se van a volver a reintentar, simplemente están aquí para que les eches un ojo.



Terminamos el bloque con la gestión de errores en la consumición de Errores (Recomendamos encarecidamente hacer un alto y echarle un ojo 👀 a los videos relacionados con la gestión de colas en el curso de Comunicación entre microservicios) viendo cómo podemos llevarla a cabo cuando trabajamos con RabbitMQ

4.3.1. Planteando la gestión de errores

En esta gestión entran en juego dos conceptos nuevos que van a facilitarnos la vida una vez implementados:

- Cola de Retry: A esta cola llegarán los eventos que por algún error no se han procesado en la cola 'padre' para volver a enviarlos a ésta pasado un tiempo de demora
- Cola de Dead Letter: Si tras N reintentos desde la cola de Retry no se ha procesado el evento, lo enviaremos a ésta cola, donde podremos configurar una alarma que nos avise para intentar ejecutar el proceso a mano y localizar dónde está el problema

El planteamiento será en primer lugar crear dos nuevos exchanges (retry y dead-letter) puesto que utilizaremos una binding key nueva que, en lugar de corresponder con el nombre del evento, lo hará con el nombre de la cola original para que cuando volvamos a re-encolar el evento lo hagamos en esa cola original y no a cada uno de los consumers que haya

Cada uno de estos dos exchanges tendrá a su vez las mismas colas que el exchange principal (pero con distinta binding key como explicábamos)
Con esta estructura, el procedimiento será, al producirse un error al consumir el evento, enviar éste a la cola de retry incrementando el contador de reintentos que tendremos en la metadata y seguidamente hacerle ack en la cola "padre"
Desde retry se volverá a lanzar el evento al exchange principal tantas veces como le especifiquemos (nosotros lo hemos acotado a 5 reintentos) tras lo cual, si sigue devolviéndonos un error, pasará a enviarse a la cola de 'dead letter' en la que se quedará sin hacer nada más

4.3.2. Creando los exchanges de retry y dead letter

Clase RabbitMqConfigurer (Declaración de los exchanges):

```
class RabbitMQConfigurer {

  constructor(
    private connection: RabbitMqConnection,
    private queueNameFormatter: RabbitMQqueueFormatter,
    private messageRetryTtl: number
  ) {}

  async configure(params: { exchange: string; subscribers:
Array<DomainEventSubscriber<DomainEvent>> }): Promise<void> {
    const retryExchange = RabbitMQExchangeNameFormatter.retry(params.exchange);
    const deadLetterExchange =
RabbitMQExchangeNameFormatter.deadLetter(params.exchange);

    await this.connection.exchange({ name: params.exchange });
    await this.connection.exchange({ name: retryExchange });
    await this.connection.exchange({ name: deadLetterExchange });
```

```
    for (const subscriber of params.subscribers) {
      await this.addQueue(subscriber, params.exchange);
    }
  }
}
```

Hemos modificado nuestra clase de configuración para que además de declarar el exchange padre, declare los exchanges de retry y dead letter, así como las colas para cada uno de ellos. Como se ve en el método configure(), llamamos las tres veces a exchange() pero previamente estaremos formateando el nombre de los exchange de retry y dead letter añadiéndoles el prefijo correspondiente.

Clase RabbitMqConfigurer (Declaración de las colas):

```
private async addQueue(subscriber: DomainEventSubscriber<DomainEvent>,
exchange: string) {
    const retryExchange = RabbitMQExchangeNameFormatter.retry(exchange);
    const deadLetterExchange =
RabbitMQExchangeNameFormatter.deadLetter(exchange);

    const routingKeys = subscriber.subscribedTo().map(event =>
event.EVENT_NAME);
    const queue = this.queueNameFormatter.format(subscriber);
    routingKeys.push(queue);
    const deadLetterQueue =
this.queueNameFormatter.formatDeadLetter(subscriber);
    const retryQueue = this.queueNameFormatter.formatRetry(subscriber);

    await this.connection.queue({ routingKeys, name: queue, exchange });
    await this.connection.queue({
      routingKeys: [queue],
      name: retryQueue,
      exchange: retryExchange,
      messageTtl: this.messageRetryTtl,
      deadLetterExchange: exchange,
      deadLetterQueue: queue
    });
    await this.connection.queue({ routingKeys: [queue], name: deadLetterQueue,
exchange: deadLetterExchange });
  }
```

Class RabbitMQConnection (declaración de colas):

```
async queue(params: {
    exchange: string;
    name: string;
    routingKeys: string[];
    deadLetterExchange?: string;
    deadLetterQueue?: string;
    messageTtl?: Number;
  }) {
    const durable = true;
    const exclusive = false;
    const autoDelete = false;
    const args = this.getQueueArguments(params);

    await this.channel?.assertQueue(params.name, {
      exclusive,
      durable,
      autoDelete,
      arguments: args
    });
```

```
    for (const routingKey of params.routingKeys) {
      await this.channel!.bindQueue(params.name, params.exchange, routingKey);
    }
  }

  private getQueueArguments(params: {
    exchange: string;
    name: string;
    routingKeys: string[];
    deadLetterExchange?: string;
    deadLetterQueue?: string;
    messageTtl?: Number;
  }) {
    let args: any = {};
    if (params.deadLetterExchange) {
      args = { ...args, 'x-dead-letter-exchange': params.deadLetterExchange };
    }
    if (params.deadLetterQueue) {
      args = { ...args, '0': params.deadLetterQueue };
    }
    if (params.messageTtl) {
      args = { ...args, 'x-message-ttl': params.messageTtl };
    }

    return args;
  }
```

A la hora de configurar las colas, no solo declararemos las correspondientes a cada exchange sino que también las estamos bindeando con los eventos de dominio y el nombre de la cola (que nos permitirá devolver como hemos visto los eventos a la cola del exchange padre). En el caso de la cola de retry, añadiremos además los parámetros necesarios para indicarle que republique los eventos que le lleguen pasado 1 segundo en el exchange principal.

4.4. 🚑 Implementación de la gestión de errores al consumir

Con la nueva ***topología de colas*** ya creada en RabbitMQ, ahora necesitamos gestionar los errores en nuestros consumidores.

4.4.1. Manejando errores al consumir eventos

Clase RabbitMQConsumer:

```
class RabbitMQConsumer {
  private subscriber: DomainEventSubscriber<DomainEvent>;
  private deserializer: DomainEventDeserializer;
  private maxRetries: Number;

  constructor(params: {
    subscriber: DomainEventSubscriber<DomainEvent>;
    deserializer: DomainEventDeserializer;
    maxRetries: Number;
  }) {
    this.subscriber = params.subscriber;
    this.deserializer = params.deserializer;
    this.maxRetries = params.maxRetries;
  }

  async onMessage(params: { message: ConsumeMessage; ack: Function; retry:
Function; deadLetter: Function }) {
    const { message, ack, retry, deadLetter } = params;
    const content = message.content.toString();
    const domainEvent = this.deserializer.deserialize(content);
```

```
    try {
      await this.subscriber.on(domainEvent);
    } catch (error) {
      this.hasBeenRedeliveredTooMuch(message) ? deadLetter() : retry();
    } finally {
      ack();
    }
  }

  private hasBeenRedeliveredTooMuch(message: ConsumeMessage) {
    if (this.hasBeenRedelivered(message)) {
      const count = parseInt(message.properties.headers['redelivery_count']);
      return count >= this.maxRetries;
    }

    return false;
  }
```

```
  private hasBeenRedelivered(message: ConsumeMessage) {
    return message.properties.headers['redelivery_count'] !== undefined;
  }
}
```

Ahora podemos modificar el consumidor de modo que cuando capturemos un error, evaluemos si queremos reintentar el evento, o por lo contrario ha llegado el momento de enviarlo a la cola de dead_letter. Para ver si se ha enviado demasiadas veces a retry lo que haremos será consultar el contador que, como habíamos visto anteriormente, iría en el header del envoltorio que RabbitMQ coloca sobre el Json del evento. Es importante tener en cuenta que cada vez que reenviamos el evento lo enviaremos con los mismos datos de la cabecera pero incrementando el contador redelivery_count

4.4.2. Tests

Test RabbitMQEventBus.test:

```
it('should retry failed domain events', async () => {
  dummySubscriber = DomainEventSubscriberDummy.failsFirstTime();
  subscribers = new DomainEventSubscribers([dummySubscriber]);
  await configurer.configure({ exchange, subscribers: [dummySubscriber] });
  const eventBus = new RabbitMQEventBus({
    failoverPublisher,
    connection,
    exchange,
    queueNameFormatter,
    maxRetries: 3
  });
  await eventBus.addSubscribers(subscribers);
  const event = DomainEventDummyMother.random();

  await eventBus.publish([event]);

  await dummySubscriber.assertConsumedEvents([event]);
});
```

```
it('it should send events to dead letter after retry failed', async () => {
  dummySubscriber = DomainEventSubscriberDummy.alwaysFails();
  subscribers = new DomainEventSubscribers([dummySubscriber]);
  await configurer.configure({ exchange, subscribers: [dummySubscriber] });
  const eventBus = new RabbitMQEventBus({
    failoverPublisher,
    connection,
    exchange,
    queueNameFormatter,
    maxRetries: 3
  });
  await eventBus.addSubscribers(subscribers);
  const event = DomainEventDummyMother.random();

  await eventBus.publish([event]);

  await dummySubscriber.assertConsumedEvents([]);
  assertDeadLetter([event]);
});
```

El primer cambio que hemos añadido para estos tests es que ahora debemos asegurarnos de eliminar no sólo las colas de la rama principal sino también las de retry y dead_letter antes de cada prueba

Para comprobar que realiza el retry correctamente lo que haremos será forzar un primer intento fallido (pasándole un consumer que falla la 1º vez) y esperando un segundo (tiempo en que la cola de retry manda el evento a la cola principal) antes de volver a tratar de consumir el mismo evento

En el caso de la cola de dead letter el proceso será forzar de nuevo tantos intentos fallidos de consumir el evento como hayamos definido de máximo para, posteriormente, revisar que efectivamente encontramos ese mismo evento en la cola de dead letter.

Estas modificaciones no nos repercutirán en el diseño de los tests de aceptación puesto que tal como explicábamos al inicio del curso, éstos se ejecutarán atacando al Bus local, priorizando que estos se ejecuten más rápido y sean los tests unitarios los que nos aseguren que todo va fino en la implementación de infraestructura

- g
  - s
- g
  - s
    - n

- g
  - s
    - n

- g
  - s
    - n

4.5. j

- g
  - s
- g
  - s
    - n
- g

4.6. j

- g
  - s
- g
  - s
    - n
- g

4.7. j

- g
  - s
- g
  - s
    - n
- g

4.8. j

- g
  - s
- g
  - s
    - n
- g

4.9. j

- g
  - s
- g
  - s
    - n
- g

5. s

5.1. j

- g
  - s
- g
  - s
    - n
- g

5.2. j

- g
  - s
- g
  - s
    - n
- g

5.3. j

- g
  - s
- g
  - s
    - n
- g

5.4. j

- g
  - s

- g
  - s
    - n
- g

5.5. j

- g
  - s
- g
  - s
    - n
- g

5.6. j

- g
  - s
- g
  - s
    - n
- g

5.7. j

- g
  - s
- g
  - s
    - n
- g

5.8. j

- g
  - s
- g
  - s
    - n
- g

5.9. j

- g
  - s
- g
  - s
    - n
- g

6. g

6.1. j

- g
  - s
- g
  - s
    - n
- g

6.2. j

- g
  - s
- g

    - s
        - n
- g

6.3. j

- g
    - s
- g
    - s
        - n
- g

6.4. j

- g
    - s
- g
    - s
        - n
- g

6.5. j

- g
    - s
- g
    - s
        - n
- g

6.6. j

- g
  - s
- g
  - s
    - n
- g

6.7. j

- g
  - s
- g
  - s
    - n
- g

6.8. j

- g
  - s
- g
  - s
    - n
- g

6.9. j

- g
  - s

- g
  - s
    - n
- g

7. g

7.1. j

- g
  - s
- g
  - s
    - n
- g

7.2. j

- g
  - s
- g
  - s
    - n
- g

7.3. j

- g
  - s
- g
  - s
    - n
- g

7.4. j

- g
  - s
- g
  - s
    - n
- g

7.5. j

- g
  - s
- g
  - s
    - n
- g

7.6. j

- g
  - s
- g
  - s
    - n
- g

7.7. j

- g
  - s

- g
  - s
    - n
- g

7.8. j

- g
  - s
- g
  - s
    - n
- g

7.9. j

- g
  - s
- g
  - s
    - n
- g

7.10. j

- g
  - s
- g
  - s
    - n
- g

7.11. j

- g
  - s
- g
  - s
    - n
- g

7.12. j

- g
  - s
- g
  - s
    - n
- g

7.13. j

- g
  - s
- g
  - s
    - n
- g

7.14. j

- g
  - s
- g

    - s
        - n
- g

7.15. j

- g
    - s
- g
    - s
        - n
- g