LAB 2: MESSAGING AND SAGAS

In this lab session you will continue developing the hospital application to learn about different communications mechanisms. You will:

- configure message channels to send and receive data between services using the abstractions of the Spring Cloud Stream framework
- use Apache Kafka as a messaging broker
- create adapters (gateways and command handlers) that convert technology-specific code to a generic business interface
- implement a saga
- create an REST API gateway, as a single point of access to your services, using the abstractions of the Spring Cloud Gateway framework

You can use the same virtual machine with the Spring Tool Suite (STS) IDE as last time.

- *Username*: student
- *Password*: student

1 Our business: hospital enterprise software

Remember that we are focusing on use cases that are related to the checking in of patients who arrive at the hospital for a pre-booked hospital stay (e.g. a planned surgery). The brainstorming with domain experts has taught us that a patient is considered to be “checked in” when the receptionist has completed the following three procedures once the patient presents himself at the reception desk:

- Checking if the administrative details of the patient need to be updated (e.g. change of telephone number, address, general practitioner).
- Opening an invoice to allow any expenses during a patient’s stay to be billed, e.g. medicines applied, room cost, medical procedures and consultations, equipment.
- Assigning the patient a bed on the ward they have booked their stay with.

Based on this analysis, we have identified the architecture and the interfaces shown in Figure 1.

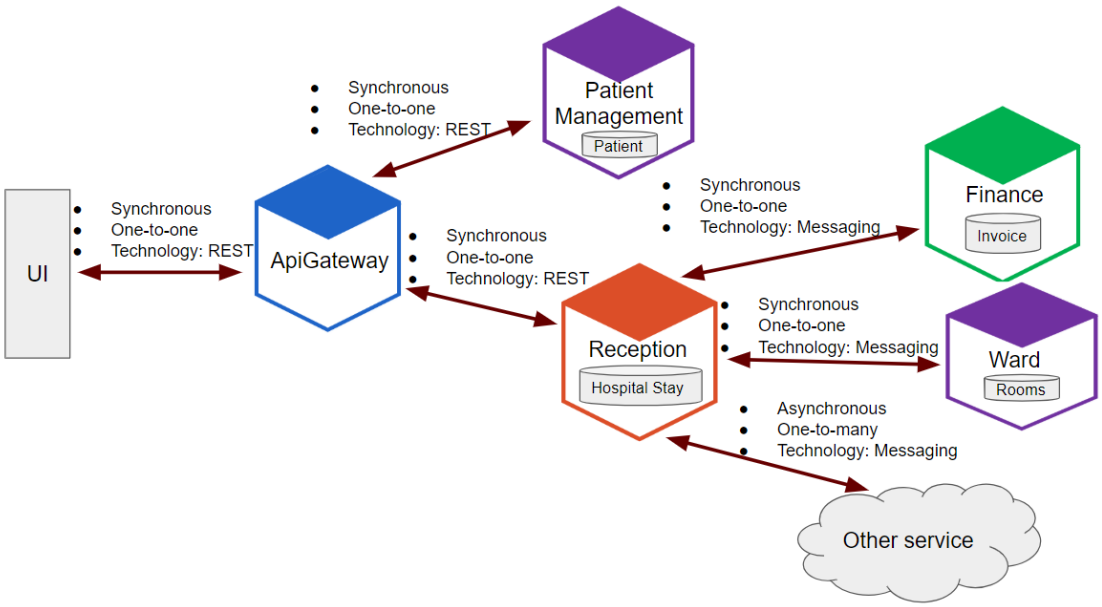

Figure 1 - Communication styles between micro-services. Notice the difference between (a)synchronicity at business level vs. (a)synchronicity at technological level.

In the theory course, we have seen that there are two dimensions to look at for communication between services:

- **synchronous or asynchronous**: this means if the client can proceed with its *business* logic with or without waiting for a reply. With synchronous communication, the reply is expected on the short term. Asynchronous communication is preferred.
- **one-to-one or one-to-many**: how many services receive the message

To implement these styles, you need to choose a particular technology, with REST and messaging the most popular choices.

We will implement two external system commands that are sent from the graphical user interface of the receptionist:

1. a synchronous command to fetch patient details. This will be implemented as a GET request issued by the UI. The request will be routed by the API Gateway to the PatientManagement service.
2. a synchronous command to check-in a patient. This will be implemented as a POST request issued by the UI. The request will be routed by the API Gateway to the Reception service. The Reception service can only confirm that a patient is checked in after the Finance service has opened a bill and the Ward service has assigned a bed. Because this transaction spans multiple services, we will need to implement a **saga**. If those tasks are successfully other services requiring this knowledge will be informed (e.g. catering, the doctor), and the hospital stay information is returned to the client.

2 The Check-In Saga

A saga must be defined for each system command that needs to update data in multiple services. A Saga is a sequence of local transactions. Inside the Reception service you will define a check-in Saga orchestrator, which performs the following steps:

Step	Service	Transaction	Compensation transaction
1	Reception Service	setPatientArrived()	checkInFailed()
2	Finance Service	openInvoice()	checkInFailed()
3	Ward Service	assignBed()	-
4	Reception Service	approvePatientCheckIn()	-

The first 2 steps of the check-in Saga are termed compensatable transactions since they are followed by steps that can fail; the 3th step is a termed the saga's pivot transaction since it is followed by steps that never fail; and the last step is termed retriable transactions since they always succeed.

3 Asynchronous request handling in the Reception Service

Download the Reception and Ward services from Minerva. These contain a solution to sections 6 and 7 from the previous lab session. Familiarise yourself with the code.

The check-in saga is started when a `check_in_patient` command is received (over a HTTP GET request). To complete this saga, the Reception Service will send commands to assign the patient a bed and open an invoice to other services. To avoid keeping the thread blocked that serves the HTTP GET request, we will use asynchronous request handling based on callbacks and the `DeferredResult` class.

In addition to the solution from lab 1, the following lines of code have already been added to the REST controller:

```
Map<String, DeferredResult<HospitalStay>> deferredResults = new HashMap<String, DeferredResult<HospitalStay>>();

// http://localhost:2223/reception/check_in_patient?patientId=1
@RequestMapping(value="/reception/check_in_patient")
public DeferredResult<HospitalStay> handlePatientCheckIn(@RequestParam String patientId) {
	DeferredResult<HospitalStay> deferredResult = new DeferredResult<>(10000l);
	deferredResult.onTimeout(() -> deferredResult.setErrorResult("Request timeout occurred."));
```

```
	deferredResults.put(patientId, deferredResult); // we will use patientId as a
unique value

	/*try {
		//call your service class (saga) here
	} catch (BookedHospitalStayNotFoundException e) {
		deferredResults.remove(patientId);
		deferredResult.setErrorResult("Failed to check-in patient. " +
e.getMessage());
	}*/
	return deferredResult;
}

//Sends result to client after patient has been assigned a bed, and an invoice has
been opened.
public void performResponse(HospitalStay response) {
	DeferredResult<HospitalStay> deferredResult =
deferredResults.remove(response.getPatientId());
	if ((deferredResult != null) && !deferredResult.isSetOrExpired()) {
		deferredResult.setResult(response);
	}
}
```

4 Reception Service

We will start by defining the channels to send the `assign_bed` command and the `check_in_completed` event, see Figure 2. These channels are needed to realize step 3 and to inform other services about the completion of the saga (after step 4).

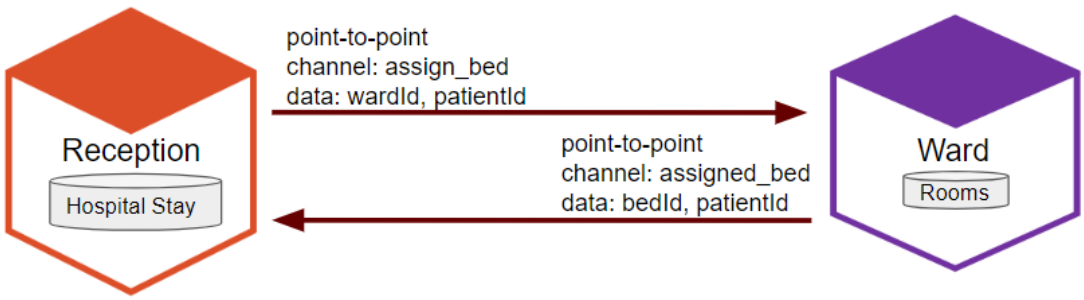

Figure 2: Communication between the Reception and Ward services.

4.1 Set up

In this lab session we will use a Kafka as the messaging broker.

- Start a local Kafka server.
  - Kafka has already been downloaded to the desktop of your virtual machine.

- Instructions on how to start the Kafka server can be found at: https://kafka.apache.org/quickstart
  - You will need two terminal windows/tabs: 1) for zookeeper and 2) for the Kafka server.

4.2 Spring Cloud Stream

Spring Cloud Stream is a framework that provides a uniform interface (i.e. Binder implementations) for message brokers (e.g. Kafka or Rabbit MQ). It enables message channels to be defined. Spring Cloud automatically detects the application properties and configures a `ConnectionFactory` for a broker. This makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the destinations (e.g., the Kafka topics or RabbitMQ exchanges) to which channels connect. Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and `application.yml` or `application.properties` files).

For more information see: https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#spring-cloud-stream-overview-introducing

4.3 Defining message channels and gateways

Let's define two channels: one for sending a command to open an invoice point-to-point (i.e. `open_invoice`); and another for broadcasting the hospital check in completed event (i.e. `check_in_completed`).

- Add the `spring-cloud-stream-binder-kafka` dependency to your Reception application.
- Create a `ProducerChannels` interface in a `be.ugent.student.adapters.messaging` package; and add the following code:

```
public interface ProducerChannels {
	String ASSIGN_BED = "assign_bed";
	String CHECK_IN_RESULT = "check_in_result";

	@Output(ASSIGN_BED)
	MessageChannel bedAssignment();

	@Output(CHECK_IN_RESULT)
	MessageChannel checkInResult();
}
```

Within the application code the `@Output` annotation identifies an output channel, through which published messages leave the application.

We need to give Spring Cloud Stream an idea of what to do with data sent into these channels, which is done using properties. Inside our `application.properties` file we can set the target destination of a channel on the bound middleware (e.g. Kafka topic). We set the Kafka topic name of "`assign_bed`" to "`assign_bed`"; and "`check_in_result`" to "`check_in_completed`".

- Modify your `application.properties` file

```
spring.cloud.stream.bindings.assign_bed.destination=assign_bed
spring.cloud.stream.bindings.assign_bed.contentType=application/json

spring.cloud.stream.bindings.check_in_result.destination=check_in_completed
spring.cloud.stream.bindings.check_in_result.contentType=application/json
```

- To activate Spring Cloud Streams, i.e. to trigger the creation of the bound channel, `@EnableBinding(ProducerChannels.class)` needs to be added to the class containing your main method.

We could now send messages over the channel by injecting it into a component (`@Autowired`) and calling it: `channel.send(MessageBuilder.withPayload(hospitalStay).build());`

However, we want to hide the messaging complexities from our business logic.

A message gateway, as a design pattern, is meant to hide the client from the messaging logic. From the client perspective, a gateway may seem like a regular object. This can be very convenient.

- Create a messaging gateway for the two channels.

```
@MessagingGateway
public interface MessageChannelGateway {

	@Gateway(requestChannel = ProducerChannels.ASSIGN_BED)
	void assignBed(HospitalStay hospitalStay);

	@Gateway(requestChannel = ProducerChannels.CHECK_IN_RESULT)
	void emitHospitalCheckInCompleted(HospitalStay hospitalStay);
}
```

Like Repositories, we can test message gateways using a bean method.

- Write a bean method to test your gateway.
  - Both the Repository and Gateway can be parameters
  - Find a HospitalStay; e.g. by using the `findByPatientId()` method
  - Call the Gateway's `assignBed()` method to send the message.
- In a terminal window start a Kafka consumer to check a message is sent to the `assign_bed` Kafka topic.
  - How to start a consumer can be found at: https://kafka.apache.org/quickstart

- Run your application

4.4 Implementing the application service and saga

When a check-in REST request is received the handler calls the application service. The application service should access the repository to find a hospital stay. If no hospital stay is found a `BookedHospitalStayNotFoundException` should be thrown; otherwise the check-in saga is started.

Write the first two steps of the check-in saga:

- Create a `CheckInSaga` component in the `domain` package
  - Use the `@Component` annotation
- Add a `startPatientCheckInSaga(HospitalStay)` method which:
  - Sets the `HospitalStay` status to pending
  - Calls the `assignBed(String patientId)` gateway method.
    - Message gateways can be inject using `@Autowired`

Write the application service (i.e. `ReceptionService` class) check-in method and call it from the REST controller. This method should:

- Retrieve the hospital stay from the repository
  - Throw a `BookedHospitalStayNotFoundException` exception if it is not found. You need to write this exception.
  - As the patient is arriving for a pre-booked hospital stay the `HospitalStay` repository has already been populated with hospital stays. In a real application, this would be done by listening for specific events emitted by other services, e.g. a service for handling reservations. In this lab session, you can populate the database with the bean method defined in the main application class.
- Start the saga
- Save the updated `HospitalStay` in the repository

Call your application service check-in method from the `ReceptionRestController` `handlePatientCheckIn(HospitalStay)` method we have provided you with; if the exception has been thrown set the `deferredResult` error message.

Run your application and visit: http://localhost:2223/check_in_patient?patientId=1 . You should receive a timeout error message, as we have not yet provided a response (i.e. called `performResponse()`). You will call this method after a message has been returned from the Finance and Ward services.

- In a terminal window start a Kafka consumer to check a message has been sent to the Kafka topic. https://kafka.apache.org/quickstart

5 Ward service

When a command to assign a bed is received, the Ward service checks if a bed is available and sends a message (event) to inform the Reception service what bed a patient has been assigned.

5.1.1 Consuming a message

On the receiver side, we want to accept delivery of a message, and perform any required processing on that message. The interface for input channels is very similar to defining output channels. The channel names at the consumer side do not have to line up with the producer side – only the destinations in the broker do (i.e. the Kafka topic).

- Add the `spring-cloud-stream-binder-kafka` dependency to your Ward application.
- Create a `ConsumerChannels` and add the following code:

```
public interface ConsumerChannels {
	String HOSPITAL_STAY_ARRIVAL = "hospital_stay_arrival";
	String HOSPITAL_STAY_CHECKED_IN = "hospital_stay_check_in_complete";

	@Input(HOSPITAL_STAY_ARRIVAL)
	SubscribableChannel hospitalStayArrival();

	@Input(HOSPITAL_STAY_CHECKED_IN)
	SubscribableChannel hospitalStayCheckInComplete();
}
```

- Add `@EnableBinding(ConsumerChannels.class)` to the main application class.

The `@Input` annotation identifies an input channel, through which received messages enter the application.

All bindings in Spring Cloud Streams are publish-subscribe by default. The publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important. When doing this, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message. Spring Cloud Stream models this behavior through the concept of a consumer group. Using a consumer group can achieve the effect of exclusivity and a direct connection.

Each consumer binding can use the `spring.cloud.stream.bindings.<channelName>.group` property to specify a group name. All groups which subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination.

Like when sending a message, we can set the target destination of a channel on the bound middleware (e.g. Kafka topic). The target destination must point to the same topic the messages are being sent to.

With all the above information, you should be able to understand the property definitions below. Set your channel properties in the Ward service as follows:

```
spring.cloud.stream.bindings.hospital_stay_arrival.destination=assign_bed
spring.cloud.stream.bindings.hospital_stay_arrival.contentType=application/j
son
spring.cloud.stream.bindings.hospital_stay_arrival.group=assign_bed_group

spring.cloud.stream.bindings.hospital_stay_check_in_complete.destination=
check_in_completed
spring.cloud.stream.bindings.hospital_stay_check_in_complete.contentType=app
lication/json
```

5.1.2 Defining a command handler

We will now write a command handler, which is a component that parses the incoming message and calls the appropriate methods of an application service.

You can add `@StreamListener` to a method to cause it to receive messages. When sending objects the incoming message will be deserialized into the method's argument.

- Create a class to deserialize the incoming message. You can call this class `HospitalStay`, or pick another name such as `PatientBedRequest.`
    - Only include fields relevant to the Ward (e.g. wardId and patientId)
- Add the following class to the Ward application and fill in the missing code:

```
@Component
public class WardCommandHandler {
	@StreamListener(ConsumerChannels.HOSPITAL_STAY_ARRIVAL)
	public void assignBed(HospitalStay hospitalStay) {
		System.out.println("Ward Operations received: " +
hospitalStay.toString());
		// IMPLEMENT THIS:
		// wardController.assignBed(hospitalStay);
	}

	@StreamListener(ConsumerChannels.HOSPITAL_STAY_CHECKED_IN)
	public void checkInCompleted(HospitalStay hospitalStay) {
		System.out.println("Patient check in completed ");
		// We will simplify the system by just printing a message.
		// In a real system if the check-in has failed you should
		//    roll-back the bed assignment.
	}
}
```

- Write a service class which:

- finds an empty bed on the required ward and assigns the bed to the patient and returns the bed
  - Use an invariant (method) in the Ward class to prevent a bed that is already occupied being assigned to the patient.
- returns null if no bed is available

5.1.3 Responding to a message

The service returns what bed the patient has been assigned. The `WardCommandHandler` should send this data to an `assigned_bed` Kafka topic.

The return of a method can be sent to a message channel using `@SendTo(channelName).`

- Add an output channel to the Ward service (you don't need a gateway – why?).
  - The Kafka topic should be called `assigned_bed`
- Use the `@SendTo` annotation to send the assigned bed to your channel.
  - If you have issues with sending data with bidirectional relationships read: http://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion
  - If you return null, a message will not be sent. When no bed is assigned you should send a bed object which just contains the `patientId`; this will be used to inform the Reception service that the bed assignment failed.

Run your application and visit http://localhost:2223/check_in_patient?patientId=1. Use a Kafka consumer to check the result is being send. If you are using the service implementations that we provided on Minerva: patient 1 and 2 should be successfully assigned a bed, and the bed assignment should fail for patient 3.

5.2 Reception service: Receiving messages

The Reception service consumes the `assigned_bed` Kafka topic and triggers the next Saga step.

- Implement an input channel which consumes the `assigned_bed` Kafka topic
- Create a `ReceptionEventConsumer` class (component) and add a stream listener method.
  - This method should parse the JSON message received and call a corresponding `ReceptionService` method, i.e. either: `failedToCheckInPatient(patientId)` if the bed assignment failed or `patientAssignedBed(patientId, bedId)`if the bed was assigned.
- Write the `failedToCheckInPatient(patientId)` and `patientAssignedBed(patientId, bedId)` service methods, which should:
  - Find the hospital stay
  - Call a corresponding Saga method
  - Persist the hospital stay.

- Implement two methods in the Saga coordinator:
  - One to updates the hospital stay to `CHECKED_IN`, and the other to update it to `CHECK_IN_FAILED`
  - Both should: 1) call the `performResponse()` method in the REST controller and 2) emit the hospital stay check-in result.

Run both the Reception and Ward services. http://localhost:2223/check_in_patient?patientId=1 should now show the hospital stay JSON object with the updated status.

6 Communication with the Finance service

Use what you have learnt so far to implement the communication between the Finance and Reception services shown in Figure 3.

- The Reception requests an invoice over the `open_invoice` Kafka topic.
- The Finance service responds on the `opened_invoice` Kafka topic.
  - if the patient has any unpaid invoices, the command to open a new invoice should fail,
  - otherwise the invoice is opened.
- The Reception service listens for the opened invoice result.
  - You should complete the Saga steps listed in the table above; i.e. send the open invoice command in the `startPatientCheckInSaga(HospitalStay)` method, and call the `assignBed()` command (from the saga) after successfully opening an invoice.

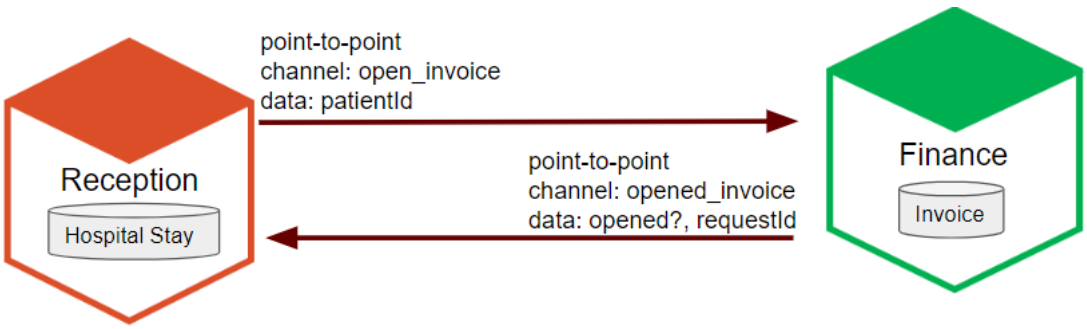

Figure 3: Communication between the Reception and Finance services.

The Reception service emits a check-in completed event after the Invoice and Bed have been returned, or one of them fails. If one of the services fails we should roll-back any changes that were made in the previous Saga steps. We won't worry about writing the roll-back code during these lab sessions, just print out a message from the Finance and Ward services when the Hospital Stay check-in completed event is received.

7 API Gateway

The API gateway routes the requests to the correct services. This gateway is the single point of access to the infrastructure services and will contact all necessary services, which will return the data required. In a real application the API Gateway should also communicate with an Authentication services, however for our simplified application we will not deal with authentication.

For a tutorial and the required dependencies see: https://cloud.spring.io/spring-cloud-gateway/

- Create an API Gateway application which runs on port 8080.
- Add the spring-cloud-gateway dependency
- Add a bean to the class containing the main method, which uses Spring Cloud Gateway to route HTTP requests to the Patient and Reception services. The example code below routes all `http://localhost:8080/patient/**` requests to the patient service.

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
	return builder.routes()
		.route(r -> r.host("*").and().path("/patient/**").uri("http://localhost:2222"))
		// add your other routes here
	.build();
}
```

Run all of your services. You should now be able to access the Patient and Reception services via your API Gateway; e.g. by visiting: http://localhost:8080/patient/1 and http://localhost:8080/recpetion/check_in_patient?patientId=1

8 Appendix

Here you will find some additional information you may find useful while working on the worksheet.

8.1 Internals of Reception Service

Figure 4 shows the internal workings of the Reception Service. The *REST controller* and *Event handler* contain the interface/adapter code, which calls the application logic in the *Reception service class*. The *Reception service class* accesses the *Repository* and calls the *Check-in Saga coordinator*. The *Check-in Saga coordinator* contains business logic, which sets the hospital stay to the correct status and starts each business step (e.g. calls a gateway's method).

To save time in these Lab sessions we have put the `controller.performResponse()` call in the Saga, however in real applications, to keep the controller code isolated from the domain logic you should emit a message from the saga and call the controller from an event handler.

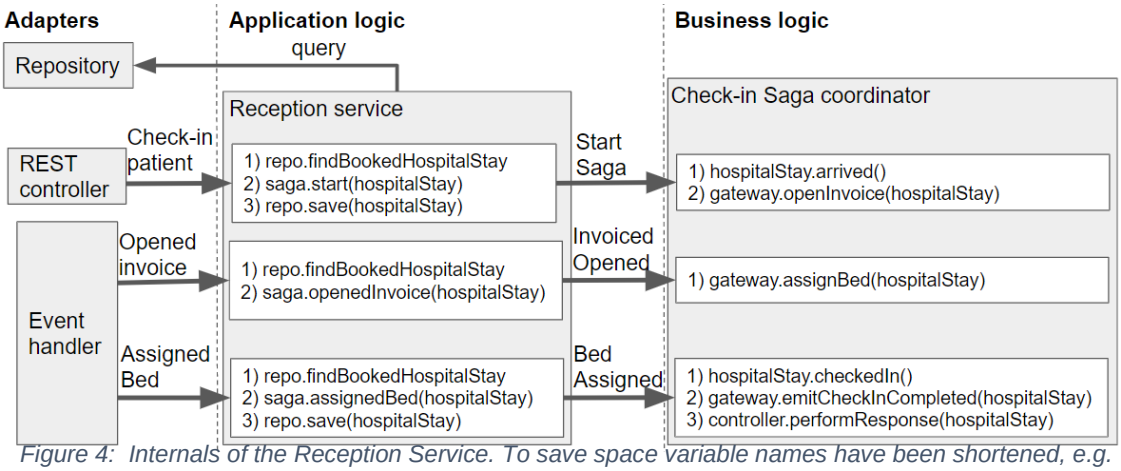

Figure 4: Internals of the Reception Service. To save space variable names have been shortened, e.g. receptionRepository has been shorted to repo.