LAB 1:
PRIMER ON SPRING BOOT REST AND PERSISTENCE

The goal of this (and the next) lab session is to provide you with an introduction on how to build an enterprise application with a micro-service architecture in the Spring Boot development environment. We will not focus on the internal service architecture, but on the messaging and persistence of services. Internal service design will be the topic in the rest of the lab sessions.

In this lab session you will learn to create micro-services in Spring Boot and how to persist data. You will:

- Learn the basics of Spring Boot
- Use a relational database (SQL) and a MongoDB document store
- Implement a REST interface for micro-services

1 Our business: hospital enterprise software

Our running example will be the design of the enterprise software in a hospital. There are many business processes taking place in a hospital, e.g. managing patient records, checking in patients for a hospital stay, invoicing, catering, parking management, etc…

Obviously, we will not implement an entire enterprise application. We will focus on scenarios 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 three procedures must be completed when a 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, name of 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.

Figure 1 shows the microservice architecture that we will build throughout this and the next lab session. Notice how each micro-service corresponds to one subdomain of the hospital's business. Each service has its own domain model. These models are in this exercise relative simple and centered around a single concept, that is persisted to a database. The API Gateway is a separate micro-service that will serve as single point of access to the application. We consider a front-end UI that will talk over REST to this API Gateway.

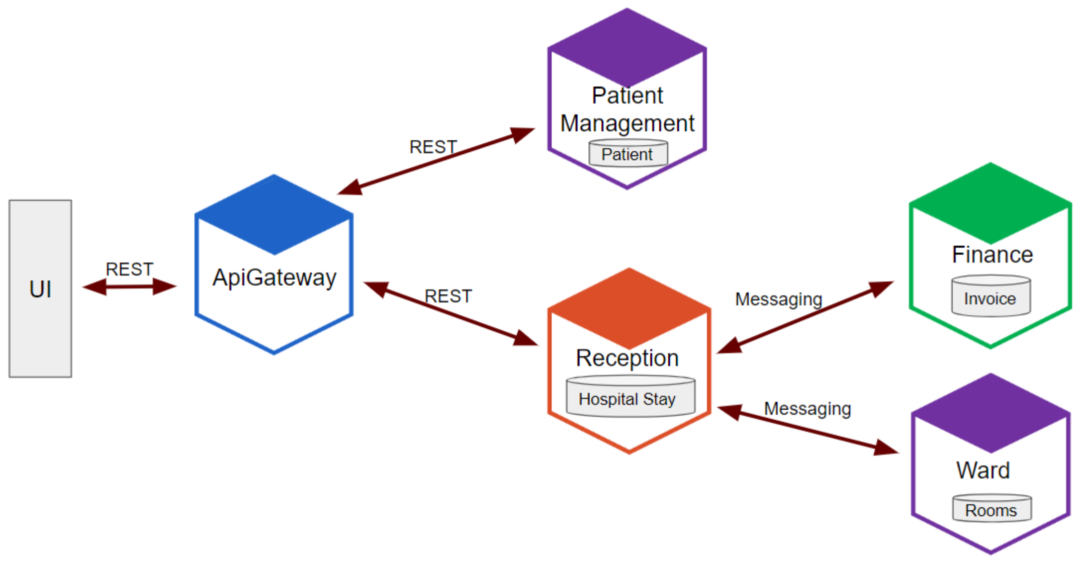

Figure 1: Micro-services you will develop during the first two lab sessions

Services only require knowledge about the interfaces of other services, they do not know about the inner workings. An update to one service should not affect the other services.

In this lab session, you will focus on creating the independent services, persistence, and REST interfaces for the Patient Management and Reception services.

2 Spring Boot

Although it is not uncommon that developer teams use separate technologies for each micro-service, we will implement all our micro-services in Spring Boot.

Spring is the de facto standard framework for developing Java-based enterprise applications. At its core, Spring is based on the concept of *dependency injection*. In a standard Java application, the application is decomposed into classes where each class often has explicit linkages to other classes in the application. The linkages are the invocation of a class constructor directly in the code. Once the code is compiled, these linkage points can’t be changed.

A dependency injection framework, such as Spring, allows you to more easily manage large Java projects by externalizing the relationship between objects within your application through convention (and annotation) rather than those objects having hard-coded knowledge about each other. Spring sits as an intermediary between the different Java classes of your application and manages their dependencies. It scans all the classes you define, interprets their annotations and creates components of it. You thus create the application as a set of components that are wired together by the framework.

Spring Boot provides a way to create production-ready Spring applications with minimal setup time. It comes with a set of familiar abstractions that are reasonable defaults for many projects. The documentation refers to this as an *opinionated* view of the Spring platform: the defaults show the community's opinion on how to build an application.

Each micro-service you design in this lab will be a different Spring Boot project. Each project will consist of multiple components (beans, controllers, database…) that are wired together by the framework.

Guides relating to Spring Boot can be found at https://spring.io/guides.

3 Set up

You have been provided with a new virtual machine.

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

For this practical you will be using the Spring Tool Suite (STS) IDE: https://spring.io/tools/sts/all. A shortcut for this has been added to the desktop.

4 The Patient Management Service

In this section we will write a `PatientManagement` application in Spring Boot and realize its REST-based interface. The `PatientManagement` service has only one responsibility: persisting and giving access to patient details. Therefore, there is no need for an advanced domain model in this service. The interface is a simple CRUD interface (Create-Read-Update- Delete) to operate on patient data that is internally stored in a relational database.

The design for this service is shown in Figure 2.

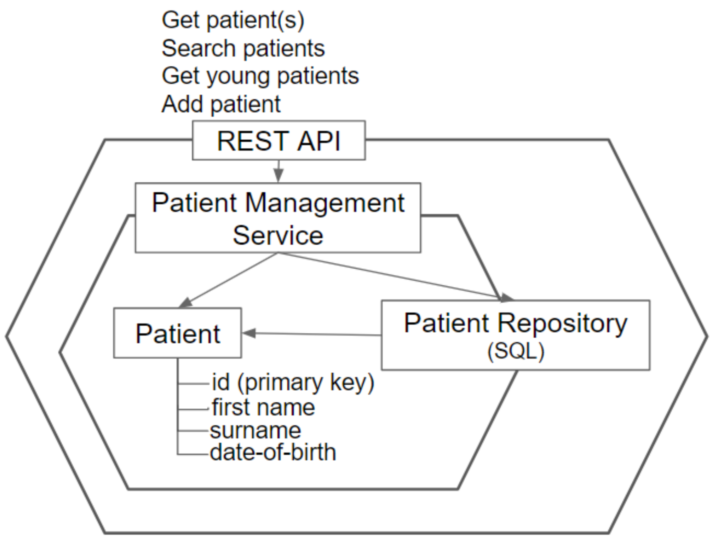

Figure 2: Design of the patient management service.

- Start the STS IDE
- Create a new “Spring Starter Project”, call it `hospital-app-patient-management`
    - Use a sensible name for the package (e.g. `be.ugent.student.patient_management`)
- Spring Starter Projects are built using Maven, so include a `pom.xml` file.
    - Take a look at what the default dependencies are.

Developers of Spring Boot have gathered related dependent projects into “starter” kits, so you don’t have to download everything and new versions of code can be released independently. In the `pom.xml` file you need to tell Maven what version of the projects you want to use.

A class containing a main method should have been generated for you. Have a look at this class. You will see the `@SpringBootApplication` annotation. This annotation is equivalent to using the three Spring annotations `@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan` with their default attributes. This is an example of “reasonable defaults” in Spring Boot.

- Check that your project runs.

Spring Boot will automatically scan child packages to find the components, controllers, beans, services, repositories it should manage.

4.1 Persistence using SQL

We will store and retrieve some patients from a relational database using an entity and a repository. The guide for this can be found at: https://spring.io/guides/gs/accessing-data-jpa/

- To be able to persist objects you will need to add a dependency for `spring-boot-starter-data-jpa`. We will use an in-memory SQL database, namely H2. When H2 is added as a dependency, Spring Boot will detect it and configure `javax.sql.DataSource` for you.
    - You can copy these two dependencies from the guide.
- Create a `Patient` entity in a `domain` sub-package, with the following fields: an id called ssn (Social Security Number), first name, last name and date of birth.

For the patient we use ssn as the ID, however for other entities you might want to auto-generate the ID. This can be done by annotating the ID field with: `@GeneratedValue(strategy=GenerationType.AUTO)`.

By default Spring Data and the JPA framework provide basic CRUD methods for accessing a database. (see: https://docs.spring.io/spring-data/commons/docs/current/reference/html/#repositories.core-concepts).

If you want to build more advanced methods you need to add them to a repository interface. A repository is an interface which is managed by Spring; you should not attempt to write the method body. Spring will parse the name of your methods and automatically convert them to the correct SQL statements. You can also annotate methods with `@Query` and specify your own SQL query.

- Create a repository in a `persistence` (sub-)package, which enables you to persist patients.
    - Add a method to return a list of patients by name (you do not need any SQL for this query).
    - Using @Query, write a method that returns all patients whose name starts with the letter 'J'
    - Using `@Query`, write a method that returns all patients who are younger than 21. Hints:
        - The date of birth field should be annotated with: `@Temporal(TemporalType.DATE)`
        - `TIMESTAMPDIFF(…)` provides the difference between two times; and `NOW()` returns the current time. https://dev.mysql.com/doc/refman/5.6/en/date-and-time-functions.html

Spring Boot will detect methods annotated with `@Bean` and automatically runs them when the applications is started.

- Add a Bean method to the class containing your main method which adds several patients to the database. To test if your application works, in the same Bean method call your queries and print out the results. You should also try out the provided repository methods: print out all patients and a patient with a given SSN.
- Run your application.

4.2 Communication using REST

We will now implement the REST interface to enable other services (e.g. the API Gateway) to access the data. The guide can be found at: https://spring.io/guides/gs/rest-service/

- Add a dependency for `spring-boot-starter-web`, which brings in everything needed to build REST applications.
- The default port is 8080. Because we will also run our other services on the same host, each service we have to deviate from this default:
  - Inside `src/main/resources/application.properties` add: `server.port=2222`
- Create a REST controller called Patient`RestController` in an `adpters` sub-package.

Following the principles of a hexagonal architecture, the `PatientRestController` should not directly access the repository. This controller is an adapter that should call a service class, which is an inbound port to the domain logic. Similarly, the repository is an outbound port that (internally) calls an adapter to the actual databse. This way, we keep the REST and persistence code separated, and enable any application logic (in the service class) to be re-usable.

- Create a `PatientManagementService` in an `application_logic` package.
- Annotate it with `@Component.` This allows you to inject an instance of the class into the `PatientRestController.`

You will need to inject the `PatientRepository` into the `PatientManagementService,` and the `PatientManagementService` into the Patient`RestController.` The `@Autowired` annotation allows Spring to resolve and inject collaborating beans into your bean. You can read more about `@autowired` at http://www.baeldung.com/spring-autowire

Write the methods required to realise the following REST calls (we provide example curl commands and return data so you can easily test your application; there are also testing tools available in some browsers, e.g. Chrome):

`/patient` GET a list of all patients.

- `curl http://localhost:2222/patient`

```
[{"ssn":"0","firstName":"Jack","lastName":"Bauer","dateOfBirth":"1995-07-
25"},{"ssn":"1","firstName":"Chloe","lastName":"O'Brian","dateOfBirth":"2000-07-
25"},{"ssn":"2","firstName":"Kim","lastName":"Bauer","dateOfBirth":"2001-07-25"}]
```

`/patient/{id}` GET the details of the patient with the given ID.

- `curl http://localhost:2222/patient/1`

```
{"ssn":"1","firstName":"Chloe","lastName":"O'Brian","dateOfBirth":"2000-07-25"}
```

`/patient/search?name=<name>` GET patients with the given first name.

- `curl http://localhost:2222/patient/search -d "name=Kim"`

```
[{"ssn":"2","firstName":"Kim","lastName":"Bauer","dateOfBirth":"2001-07-25"}]
```

`/patient/young_patients` GET patients younger than 21

- `curl http://localhost:2222/young_patients`

```
[{"ssn":"1","firstName":"Chloe","lastName":"O'Brian","dateOfBirth":"2000-07-
25"},{"ssn":"2","firstName":"Kim","lastName":"Bauer","dateOfBirth":"2001-07-25"}]
```

`/patient` PUT (add) a new patient

-
  ```
  curl http://localhost:2222/patient -X PUT -H "Content-
  Type:application/json" -d
  "{\"ssn\":\"3\",\"firstName\":\"David\",\"lastName\":\"Palmer\",\"dat
  eOfBirth\":\"1976-07-25\"}"
  ```

```
[{"ssn":"0","firstName":"Jack","lastName":"Bauer","dateOfBirth":"1995-07-
25"},{"ssn":"1","firstName":"Chloe","lastName":"O'Brian","dateOfBirth":"2000-07-
25"},{"ssn":"2","firstName":"Kim","lastName":"Bauer","dateOfBirth":"2001-07-
25"},{"ssn":"3","firstName":"David","lastName":"Palmer","dateOfBirth":"1976-07-
25"}]
```

Test your application.

5 Finance Service - MongoDB

It is the responsibility of the finance department to create and manage patient invoices. When the patient arrives at the hospital a new invoice is created, and during the patient's stay any expenses are added to the invoice. When a patient checks out of hospital, the invoice is closed and sent to the relevant parties (i.e. patient and insurer). The design of this service is shown in Figure 3.

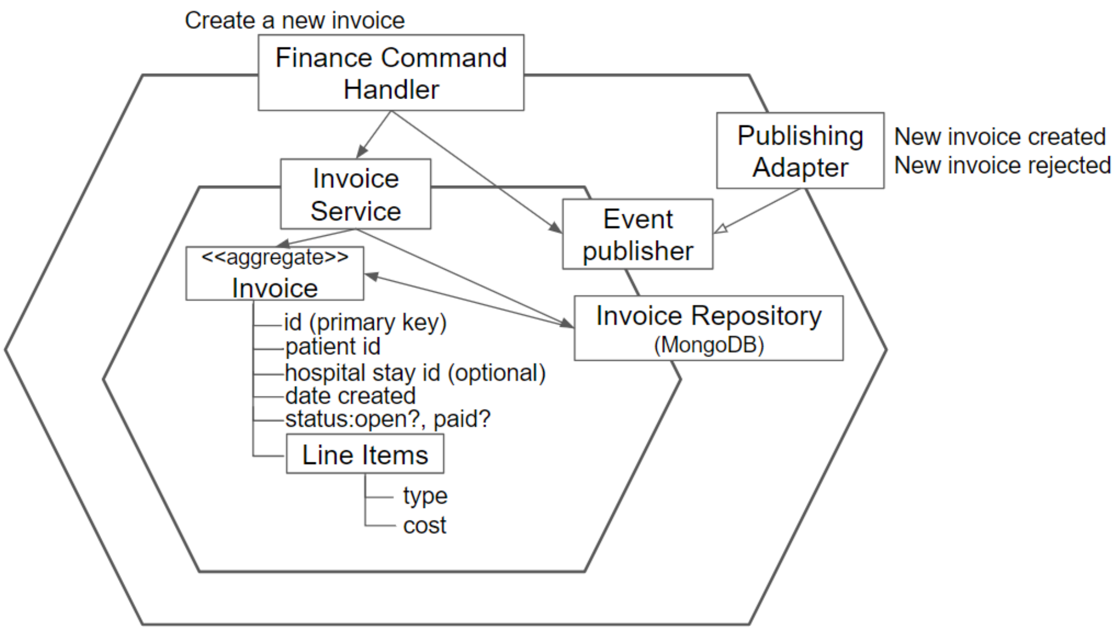

Figure 3: Design of the Finance service.

You will develop the data storage aspects of the Finance service, we will leave the messaging part for the next lab session.

Aggregates

In Domain-Driven-Design (DDD), an aggregate is a cluster of domain objects that is treated as a single unit. Only the aggregate root will be exposed to the outside so the aggregate can maintain its invariants. Invariants are conditions that must always be true for the aggregate.

In the `Finance` service, the `Invoice` is an aggregate, consisting of the `Invoice` class and one or more `LineItems`. Because people can only access an invoice via the `Invoice` class, we can enforce invariants such as preventing `LineItems` from being changed if the invoice has been closed. `LineItems` should only be accessed by the `Invoice` class.

MongoDB

Think about how invoice data would be persisted in a typical relational database and the operations that would be needed in order to query and update it.

Rather than using a table-based data model, we will use a document model. This allows for the number and type of fields to change from document to document (in particular: the line items).

increases flexibility and more easily maps onto objects in your code. We will use MongoDB that is a “document store” for JSON documents.

The amount of line items on the invoice will vary. This kind of one-to-many relationship simply maps to a variable length array in Mongo whereas it would require a separate mapping table in SQL (which leads to expensive join operations slowing down the application).

The Spring Boot MongoDB tutorial: https://spring.io/guides/gs/accessing-data-mongodb/

5.1 Download and install MongoDB

MongoDB isn’t installed by default and unlike the H2 SQL database, Spring can’t start one for you automatically. Therefore, you will need to install and run an instance yourself.

- Install MongoDB. The instructions can be found on: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/
- Start MondoDB
    - `sudo service mongod start`

5.2 Create the Finance service

- Create a new Spring Boot application and include MongoDB in the dependencies. If done correctly you should see the following in your POM file:

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

- Change the port number the application runs on to 2225.
- Create an `Invoice` class and repository. The `Invoice` should contain the following fields:
    - Invoice ID (primary key. For Mongo to generate the ID it has to be a String.)
    - Patient ID
    - Hospital Stay ID
    - Date created
    - Status (Enum: Open, Closed, Paid)
    - List of charged Items (`ListItem`)
        - Create a `ListItem` class with fields for name and cost.

In order to let Spring know it should store the `Invoice` as a document you should annotate the class with the `@Document` annotation.

- Create a Bean method like you did for the Patient, populate the database with a few self-made invoices and read it back out to verify your solution works.

Next we will create some custom queries. Check the MongoDB documentation for how to write a query: https://docs.mongodb.com/manual/tutorial/query-documents/

6 Reception Service

When a patient checks in for a hospital stay the Reception service will look-up the patient's (preregistered) stay, request an invoice to be opened and requests a bed on a ward. If these steps are successful the service will emit a `Check-in Completed` event that can be consumed by any service requiring this knowledge (e.g. catering, the doctor). A design for this is shown in Figure 4.

The domain model of this service is again very simple. It consists of only one aggregate, that consists of only one entity: `HospitalStay`. In this session you should develop the `HospitalStay` entity, its repository and a basic REST controller for this design.

- Create the Hospital Stay service.
- Implement the `Hospital Stay` aggregate and repository. Include fields for: id, patientId, wardId, dateOfStay, bedId and status.
  - It is up to you which type of data model and storage technology you use (document store or relational database).
- Implement a REST controller with the following paths (you will also need to create a service class and add the relevant queries to the repository):
  - **`/reception/hospital_stays`** Returns a list of all hospital stays.

```
curl http://localhost:2223/reception/hospital_stays
```

```
[{"id":1,"patientId":"1","wardId":1,"bedId":null,"dateOfStay":"2018-08-
27","status":"BOOKED"},
{"id":2,"patientId":"2","wardId":2,"bedId":null,"dateOfStay":"2018-08-
27","status":"BOOKED"},
{"id":3,"patientId":"3","wardId":3,"bedId":null,"dateOfStay":"2018-08-
27","status":"BOOKED"}]
```

  - **`/reception/get_patient_booking?patientId=<id>`** Returns a single hospital stay for a patient with the status 'BOOKED'.

```
curl http://localhost:2223/reception/get_patient_booking?patientId=1
```

```
{"id":1,"patientId":"1","wardId":1,"bedId":null,"dateOfStay":"2018-08-
27","status":"BOOKED"}
```

- Change the port number so your application runs on port 2223.

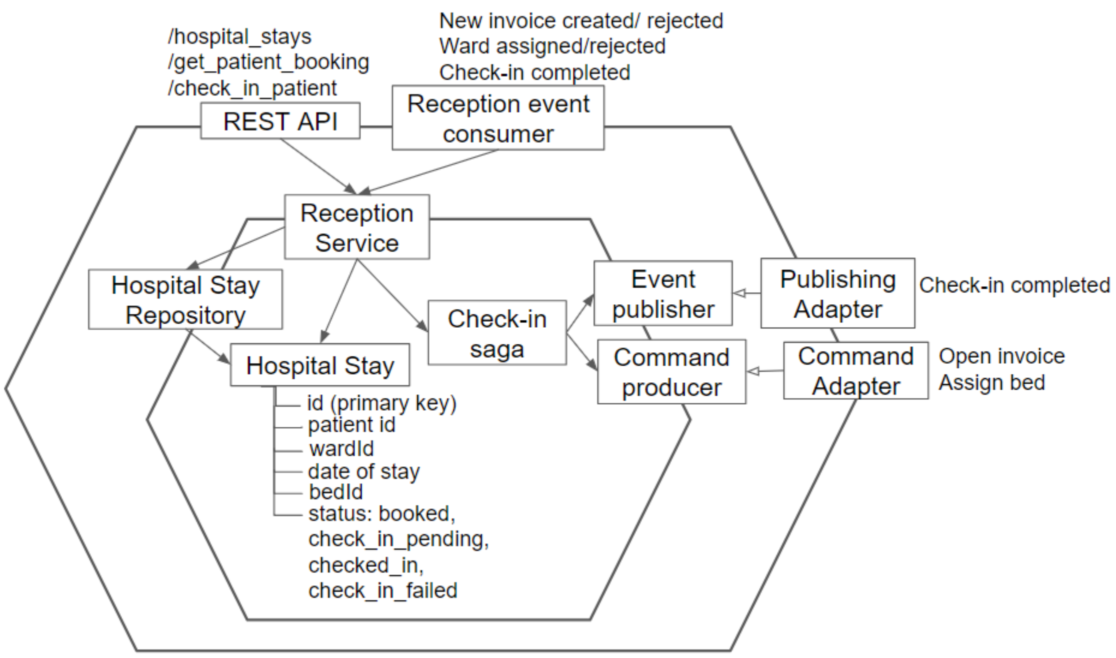

Figure 4: Design of the Reception service.

7 Ward Service

The Ward service keeps a record of which beds on each of the wards (maternity, pediatry, oncology, intensive care, etc…) are occupied by a patient. The `Ward` aggregate protects one invariant: a patient cannot be assigned a bed on a ward if there are no unoccupied beds available. The design is shown in Figure 5.

- Create the Ward service
- Implement the `Ward` aggregate and repository. Include fields for: id, and bed. A `Bed` should have an id and patientId.
  - It is up to you which type of data storage you use.
- Implement a query which returns the number of unoccupied beds (i.e. patientId is null) on a given ward (wardId should be a parameter). Test if it works using a `@Bean` method.

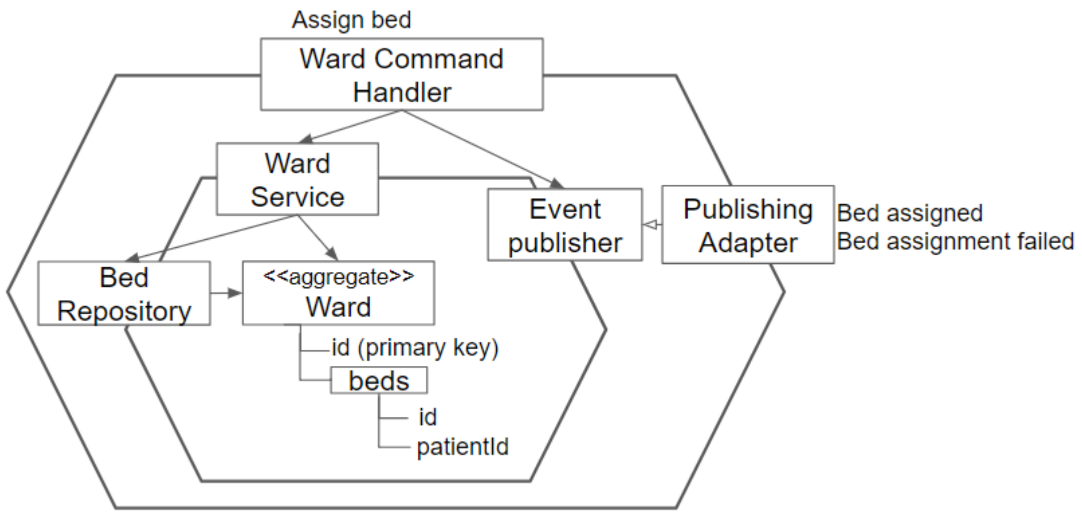

Figure 5: Design of the Ward service.

In the next lab session you will implement the messaging-based interfaces and the API gateway service.