MTAT.03.229 – Enterprise System Integration

Regular Exam (Sample)

Notes:
- The total duration of the exam is 3 hours and 30 minutes.
- Parts 1 and 2 of the exam are closed-book and closed-laptop. These two parts should be submitted on paper. Once you submit parts 1 and 2, you will be allowed to open your laptop to start part 3.
- Part 3 is open-laptop. Web browsing is allowed.
- You are not allowed to communicate with anyone during the exam in any way (except with the lecturer).

**Part 1. Theory (10 points)**

1) The conversion between DTOs and domain objects should be implemented in which of the following classes (multiple options are possible)
   a. A validator class
   b. A controller class
   c. A service class
   d. A repository class
   e. An assembler class

2) What is content negotiation? What does a client application need to include in an HTTP request to take advantage of content negotiation?

3) Consider an application to manage room bookings. One of the resources in this application is the Booking. The application exposes a container resource /bookings/ that provides access to all the bookings, and one item resource /bookings/{bookingid} for each booking. A booking consist of a reference to a room, a start date-time and an end date-time. Let us assume that you have to implement an operation to update a booking. Possible updates include changing the room, the start or the end time of the booking, or any combination of these three attributes. What would be the difference between using the HTTP PUT method to update a booking versus the HTTP PATCH method?

4) With reference to the example of the previous question, what status code could be returned when a request is made to change a booking that has already taken place in the past? (Such retroactive changes are not allowed).
   a. 400 Bad request
   b. 404 Not found
   c. 405 Method not allowed
   d. 409 Conflict
   e. 415 Unsupported Media Type

5) With reference to the example in the previous questions, a developer proposed to implement the operation to cancel a booking using the HTTP DELETE method. However, after a meeting with a customer to clarify the requirements for the cancelation feature, the development team realized that in some use
cases, it is necessary for some users to see the details of cancelled bookings. Would it be possible to implement this operation using the DELETE method (if so, how?) or should an alternative approach for cancelling meetings be considered (and if so, how would a meeting be cancelled via the REST API?).

6) Explain at least three advantages of a microservice architecture relative to a monolithic application? Each of these advantages should correspond to something that can be achieved with a microservice architecture and that cannot be achieved to the same extent with a monolithic architecture.

Part 2. RESTful API of FinInv’s invoice financing platform (20 points)

The FinInv platform is an invoice financing platform where sellers create invoices and sell these invoices to funders in order to get advance payment and thus better manage their cash flow. The users of the platform are sellers (who issue invoices and can then sell them), funders (who buy invoices and later get paid back by the buyers), and buyers (who receive invoices, approve them and eventually pay them).

The main process to be supported by the FinInv platform is the invoice financing process, which proceeds as follows. A seller creates an invoice addressed to a given buyer via the FinInv platform's API. The buyer can approve, reject or dispute the invoice. If they dispute the invoice, the seller can create a debit note or a credit note to correct the invoice. Note that the original invoice is never modified. Instead, a credit note or a debit note is added to it. When a credit or debit note is added to an invoice, the buyer can again accept, reject, or dispute the invoice. In the latter case, the seller might issue another credit or debit note. Sellers may also withdraw an unapproved invoice from the platform anytime. This is typically done in order to resolve invoice disputes outside the FinInv platform.

An invoice consists of a seller, a buyer, an issue date (assigned automatically when the invoice is submitted), a payment due date, a total amount and an invoice body. The invoice body has a relatively complex structure (consisting of line items) but we will treat it as a black-box in this exercise (e.g. you can treat it as a string here). Since detailed data about all buyers and sellers is already registered in FinInv, the buyer and the seller of an invoice can be referred to using their unique identifier in the FinInv platform.

A credit-debit note consists of a reference to an invoice, and an amount. The meaning is that the amount of the credit-debit note is added to the total amount of the invoice to which it is attached (i.e. if the invoice is for 1000 euros it has an attached credit-debit note with an amount of -200, then the effective amount of the invoice is 1000 + (-200) = 800.

Once an invoice is approved, it is offered to funders. Funders can view the invoices that are on offer and bid for an invoice. Each bid includes a "markup" (e.g. 2%, 3%). The markup is a percentage deducted from the invoice amount in order to compensate the funder. In addition, FinInv applies a 1% fee for each invoice sold via its platform. If an invoice is for 1000 euros, the markup is 3% and the fee is 1%, then the seller receives 960 euros for the invoice sale.

Bids for an invoice are collected during a period of 24 hours after an invoice is approved. At the end of this bidding period, the invoice is sold to the funder who offered the lowest
markup. The funds (minus markup) are then debited from the account of the winning funder (i.e. the account of this funder held at FinInv). FinInv retains its 1% fee and transfers the remaining amount to the seller. Sometimes an invoice is not sold (if it receives no bid).

The buyer of an approved invoice eventually pays the amount of the invoice (plus late fees if applicable) to FinInv. FinInv then transfers the proceeds to the funder who bought the invoice (if the invoice has been sold) or to the seller (if the invoice was not sold).

**Task 1 [8 points]**. Design a domain model for the FinInv platform based on the above description.

**Task 2 [4 points]**. Specify a state machine capturing the lifecycle of an invoice. Annotate the transitions in the state machine with the corresponding (REST) operations.

**Task 3 [8 points]** Specify an API of the RESTful interface that the FinInv platform will provide to its users. For each operation on a resource, you should specify the (relative) URL, the HTTP method, the state(s) in which the operation may be performed, and an example of the payload of the request (if applicable) and of the payload of the response (if applicable) in JSON format.
Part 3. REST API Implementation and Testing (20 points)

In the practice session of 2 April, we implemented a version of BuildIt’s information system in which a Plant Hire Request (PHR) is by default set into PENDING status when the Site Engineer creates it. Then, the Works Engineer is responsible for accepting or rejecting the PHR after analyzing the request.

Rentit’s management has decided that a PHR will be automatically accepted whenever the cost of the PHR is 1500 EUR or less. In all the other cases, the processing of the PHR should proceed as usual (i.e., the Works Engineer has to review the PHR and then accept or reject it).

**Task.** [Tests: 8 points + Business logic: 12 points]. You are required to implement the above change and to write at least three tests to ensure that the above requirement is met. Your tests should consider both “success” cases (the PHR is accepted automatically or manually), as well as test cases where an operation is performed on a resource in a state where this resource cannot accept this operation (e.g. trying to reject an already accepted PHR).

You must clone the git repository with the solution of the Practice Session of 2 April, which can be found in commit 3ac77c0 of the following repository

https://bitbucket.org/manuel_camargo/rentitf/src/master

You should then do the required changes to the code, add the tests, and commit it to a new bitbucket repo.

Your application does not need to have a frontend. It only needs to expose the required REST operations.

You should submit a TXT or PDF file with

- The link to your Bitbucket repository with the solution. If there are several commits in your repository, you must indicate which commit we should use for grading.
- An explanation of any changes you made to the API