Exam Project: Problem Description

Version 1

Hubert Baumeister

January 12, 2025

Contents

1 Exam Project **1**
1.1 Functionality 1
1.2 Authentication and Authorization 3
1.3 Architecture 3
1.4 Implementation 4
1.4.1 Customer-, merchant-, and manager facade 4
1.4.2 The use of databases 5
1.5 Process 5

2 Important: Mark your contributions **6**

3 Project Deliverables **6**

1 Exam Project

1.1 Functionality

DTU Pay is a company that offers a mobile payment option for shop owners (i.e. merchants) and customers. Both, customers and merchants, already have a bank account with the bank (the bank is supplied by me). To use the service, both, customers and merchants have to be registered with DTU Pay by providing their names and other personal informatioon plus their bank account numbers that have been obtained from the bank earlier. To take part in the mobile payment process, both, the customer and the merchant, have to have mobile devices.

In the payment process, the customer presents a unique token he has previously received from DTU Pay to the merchant, e.g. using RFID technology, as a proof, that the customer has agreed to the payment (cf. Fig. 1),

For the payment, the customer has to have at least one unused token in his posession. A token consists of a unique number/string (what is a good choice?). The tokens in the system should be unique and it should not be possible to guess an unused token. Furthermore, for privacy reasons, the tokens cannot contain any information about the customer to whom the token belongs. Only DTU Pay knows to which customer a token belongs.

The customer can request 1 to 5 tokens if he either has spent all tokens (or it is the first time he requests tokens) or has only **one** unused token left. Overall, a customer can only have at most 6 unused tokens.

If the user has more than 1 unused token and he requests again a set of tokens, his request will be denied.

A token can only be used for one payment. DTU Pay holds a record of all payments done via DTU Pay for each customer plus the tokens used in these payments.

The functionalities that should be implemented are:

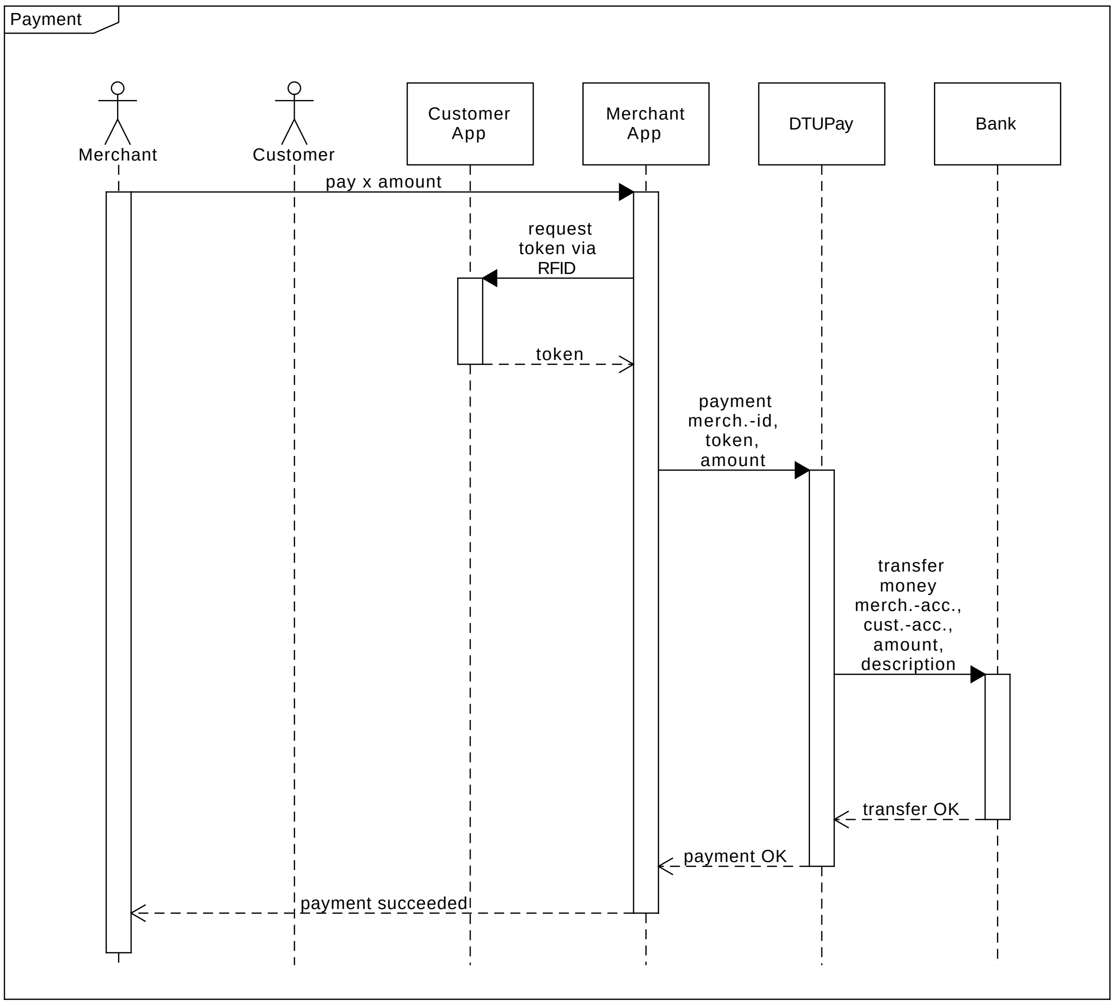

Figure 1: Only the actual payment is shown. It is assumed that the customer and the merchant are already registered with DTU Pay and that the customer has obtained at least one token.

- Pay at merchant, which includes transferring money from the account that the customer has in the bank to the account of the merchant in the bank. The account ids for the customer and merchants are provided when the customer and merchant register with DTU Pay. DTU Pay **does not** create bank accounts. In principle (though not implemented in the bank), each customer/merchant could have several bank accounts. Thus it is important to mention which bank account DTU Pay should use when registering a customer/merchant.
  - Note that, when registering a new customer and merchant, DTU Pay **does not** check if their bank accounts exist or that the bank account of the customer has enough money. The reason is, that registration can happen a long time before actually using DTU Pay to pay. Inbetween registration and first usage of the payment function, the bank account may not exist anymore or the necessary funds may not exist.
  - The money transfer service of the bank is responsible for making sure that the business logic for transfering money is observed, ie., that both accounts exist and that the debtor has sufficient funds. The money transfer functionservice will throww an exception when the money transfer call does not satisfy the business rules of the bank.
- Customers can obtain unique tokens (i.e. Token Management) according to the rules outlined above.
- Managing customer and merchant accounts with DTU Pay (i.e. Account Management)
  - The merchant and customer register themselves via their DTU Pay apps.
- A reporting function for the manager or DTU Pay to see all payments and a summary of all the money being transfered. For simplicity it is assumed that there is only one manager. That means, there is no need for managing manager accounts.
- A reporting function, that generates for a customer the list of his payments done with DTU Pay (amount of money transferred, with which merchant, and token used). DTU Pay has to self keep track of the payment transactions. It is **not possible** to use the transactions stored in the bank. These transactions may contain transactions not done through DTU Pay, and DTU Pay should not see those transactions.
- A reporting function, that generates for a merchant the list of the payments with DTU Pay he was involved with (amount of money transferred and token used). Note that the merchant should not know who the customer was. Thus the customer (only the token) should not appear in the list of payments that the merchant sees.
- ...

If you have questions regarding the functionality or if you run out of functionalities to implement, please contact me.

Note that the functionality of the terminal that reads the RFID token is not part of the project. The merchant "just knows" the token that the customer uses for payment.

1.2 Authentication and Authorization

Authentication and authorization is of course very important in a real application. However, in this course we disregard authentication and authorization. This means it is okay that everybody can use all endpoints without having to authenticate himself.

1.3 Architecture

Fig. 2 gives an overview over the architecture. The customer and the merchant both are using a mobile device to interact with DTU Pay. DTU Pay offers three distinct REST interfaces. One for the customer, one for the merchant, and one for the manager.

The customer and merchant apps offer each an API to access the corresponding ports of DTU Pay. The presentation layer, as well as the business layer of the apps do not need to be implemented. Both, the customer and the merchant API's, are used by the automated system tests to test the functionality of the whole system.

Here is an example of the test scenario for successful payment in pseudo code (of course, you should write this as a Cucumber scenario). Here, CustomerAPI and MerchantAPI are classes inside the test client that represent the API's used in the merchant and customer apps to access the REST interface of DTU Pay.

```
customerAccount = bank.createAccount Name, CPR, balance
merchantAccount = bank.createAccount Name, CPR, balance
cid = customerAPI.register Name, customerAccount, ...
                  this will register the customer via the customer port of DTU Pay
  mid = merchantAPI.register Name, merchantAccount, ...
                  this will register the merchant via the merchant port of DTU Pay
tokens = customerAPI.getTokens 5, cid
                  this will get 5 tokens via the customer port of DTU Pay
token = select a token from tokens
merchantAPI.pay token, mid, amount
                  this will execute the payment via the merchant port of DTU Pay
                  and in DTU Pay, this will cause a call the money transfer service
                      of the bank
customerBalance = bank.getBalance customer account
merchantBalance = bank.getBalance merchant account
check that customerBalance and merchantBlance are correct
bank.retire customer account
bank.retire merchant account
```

Tip: To make it easier to run the system tests, the tests (Cucumber/JUnit), and both API's (customer and merachant API) should be implemented in one project, so that the tests can access the API of both the customer and the merchant at the same time directly. This is helpful to simulate the exchange of the token between customer application and merchant application using RFID.

The bank is provided by me and reachable at `http://fm-00.compute.dtu.dk`. You should use SOAP to access the bank.

1.4 Implementation

DTU Pay should be implemented using a microservice architecture using docker containers. The communication among the microservices should use message queues. The communication to the outside should use REST and SOAP (to the bank). The resulting application(s) should have associated tests with a high code coverage, and business logic should be tested using Cucumber scenarios. Note that it is not required to document what code coverage you have. However, the idea is that all your production code has associated tests. This should also be interpreted in the opposite direction. Don't include code for which you don't have a test, either an end-to-end test or a service level test.

1.4.1 Customer-, merchant-, and manager facade

The idea is that there are three facades that present DTU Pay to the outside while encapsulating the internal services. The customer facade, for the customer mobile application, the merchant facade, for the merchant mobile application, and the manager facade.

These facades can be offered either by one Microservice offering its services, e.g., on port 8080, where the three facades are distinguished by their path, e.g. "customers/. . . ", "merchants/. . . ", and "reports/. . . ".

Alternatively, there can be three Microservices, one for the customer, one for the merchant, and one for the manager running on, e.g., ports 8080, 8081, and 8082, respectively.

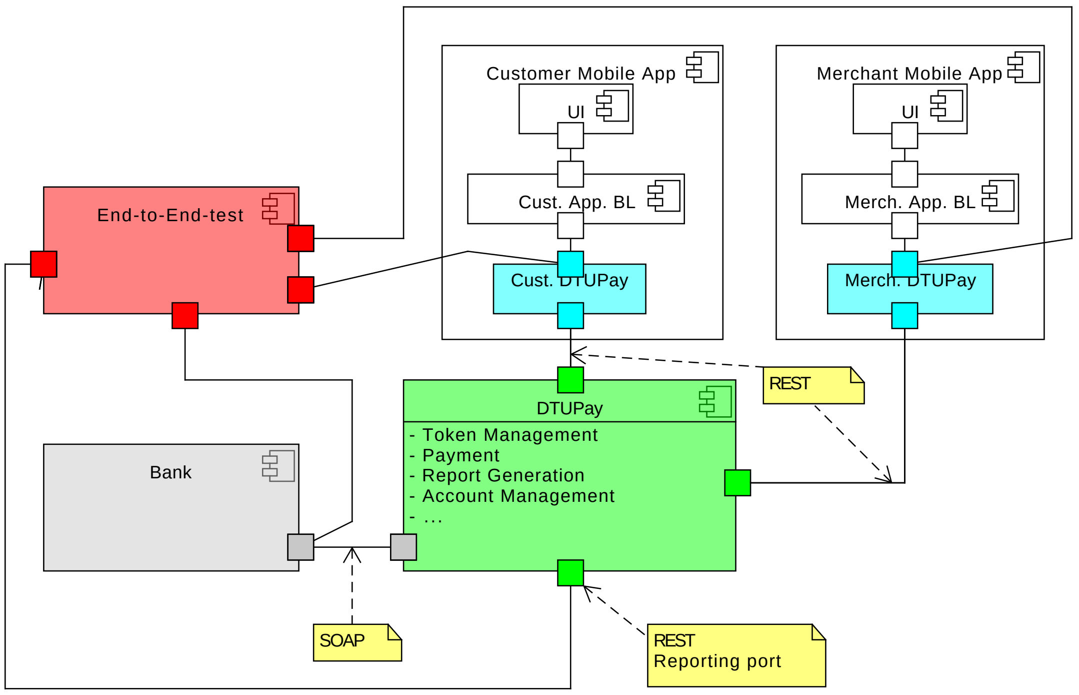

Figure 2: DTU Pay Overview. The green parts are the components that make up DTU Pay. The cyan parts are the customer and merchant APIs that the mobile apps use to talk to DTU Pay. The red parts belong to the end-to-end tests. The green ports at DTU Pay are the facades DTU Pay offers to the customer, the merchant, and for reporting, respectively.

The functions offered by these facades are for the customer: register, deregister, getTokens, and getReport (all payments the customer is involved in); for the merchant: register, deregister, pay, and getReports (all payments the merchant is involved in) and for the reports: getReports (all payments).

What should not be possible is that the customer API on the mobile phone directly accesses the internal services of DTU Pay, like the token manager and account manager services. It would break encapsulation if, e.g., the verify token functionality of the token manager would be visible to the mobile app.

1.4.2 The use of databases

You want to use the concepts of a repository from domain-driven development to store and retrieve data. However, it is not required to implement those repositories as databases. It is enough that those data is stored in memory in lists or hash maps.

1.5 Process

For the planning of your work, think about the resource triangle: time, functionality, and quality. The time for the project is fixed. Quality should be fixed too. Functionality is therefor variable. Grading will value quality over quantity. Quality here means, that your design and implementation applies the concepts taught in the course, e.g. using domain-driven design, hexagonal architecture, S.O.L.I.D principles, endpoint design according to the REST architecture principles, use of messaging, etc. When you choose a functionality to implement next, you should think about how good it will show that you have achieved specific learning objectives of the course.

You should look first at the successful payment process, then at token management, and then at reporting. After that you should also look at possible fail scenarios, e.g. when payment should not succeed (e.g. merchant is unknown, the customers token is not known, or invalid, or already used, etc.).

Important My suggestion is, to use an agile approach for implementing DTU Pay. An essential component of any agile approach is to focus on user stories that give value to the customer. Thus, my suggestion is to start with implementing the successful payment scenario.

Take the first step in the scenario and implement the step definition for the step. Use event storming to determine how the functions related to DTU Pay needs to be realised. Then implement the required microservices, but only as far as needed to implement the step definition. For example, in scenario in Sect~1.3, the first two steps create bank accounts for the customer and the merchant. Nothing has to be done from your side apart from calling the corresponding bank service in the step definition. The third step registers the customer with DTU Pay. Here, all the required code should be implemented to create a registration function in the customer facade and in DTU Pay. Note that this is not the full registration function yet. Only as much functionality is implemented as needed to execute the successful payment scenario. Basically, this means remembering the customer and his bank account number in DTU Pay. It is a good thing to remember, e.g., on a piece of paper, which code you may have wanted to implement, like error code checking. Once you are done with the scenario you are working on, you can make those notes into user stories and test and implement them in separate steps.

Make sure that with each of the implementation steps, you move from a working system to a working system and don't have a not working system for a long time.

Mob programming supports this development style.

The alternative approach, to first implement the separate microservices, before making sure that the successful payment scenario works, is discouraged. This violates the practice of continuous integration and has the danger that a lot of rework is needed to make the scenario work.

2 Important: Mark your contributions

DTU has the requirement of individual grading. This poses a challenge in group work. The solution is, that you put the names of the main responsible for a piece of text/code in the corresponding documents, i.e. in the PDF files and as comments in the source code (e.g. using the @author) tag.

If several have worked on the same text/code (which I expect to be the default case), then only one main contributor should be mentioned. If all of you contribute to everything equally, then you can choose any name of those who contributed to put as the contributor. What you are not allowed to do is, to mention someone as contributor, who didn't contribute to the text/code. **In addition, it is a good idea to explain in the project description document how you have worked together.**

Grading is based on how you have achieved all the learning goals mentioned in the learning objectives section of the course description. Therefore, make sure that all of you contribute to all aspects of the text/code. For example, everybody should have written Cucumber scenarios, implemented them, created a REST endpoint etc.

3 Project Deliverables

The deliverables are to be put into **one** Zip file that is to be uploaded. The Zip file needs to contain the following:

- Software:
  - all the Maven projects that you have used as top level directories in such a way that I can import the projects as Maven projects into Eclipse
    - run "`mvn clean`" for each project before putting it in the zip file to keep the zip files small
  - include source code with all JUnit & Cucumber Tests
  - any other files needed to deploy and run your application (cf. Installation Guide)
  - It should be possible to build, deploy, and run all the tests locally on your computer. There should be no dependency on any services running on the Linux virtual machine.

  - Your code needs to be able to build, deploy, and test on any Linux computer with Java, Docker, docker-compose, and Maven installed.
    * It is a good idea to do a fresh install to check that this is the case
  - Jenkins can build, deploy, and test your project (Andon board is green).
    * To this purpose, I need an account with name `huba` on your Jenkins server (see below)

- Swagger/OpenAPI files for each REST API implemented
- Project Description (PDF, at the root of the Zip file)
  - including a list of the team members with name and study number
  - show the result of the event storming process:
    * Show only the final event storming diagram and not the intermediate steps
      · Also provide a legend of which color scheme you have used, events = orange, etc. This can be done in a form of a small diagram.
    * UML class diagrams showing the classes implementing the business logic.
  - description of the architecture including drawing(s),
    * e.g. what are the microservices
    * a description of the different REST interfaces used. Use tables for this (like those used in the lecture).
      · Explain your design decisions for the resources, URL's for resources, and operations.
    * how do the microservices communicate (REST, Messages)? What are the messages being exchanged? How many message queues are there and how are the configured (e.g. topics used)?
  - description of the major features/scenarios
    * What are these features/scenarios, what do these features/scenarios do, and how are they realized within your architecture? For example, how is the feature initiated, what messages are exchanged between which Microservices, etc.
    * You need to show, how a given feature/scenario is realized in your architecture. The idea is, that, in principle, from that description, a reader would be able to implement the feature him/herself.
  - provide a short description of how you have worked together
- Installation Guide (PDF, at the root of the Zip file)
  - I need to be able to install your microservices and run the tests from the command line on any Linux computer, without the use of Jenkins and without requiring access to your virtual machine.
    * make sure that the shell scripts follow the Unix convention for endlines and not the DOS/Windows convention
    * make sure that the shell scripts are executable for the user
  - On the top level (or in the system test project) provide a shell script that builds, deploys, and runs all the tests.
  - URL(s) of the git repository(ies) which I can access.
    * For GitHub and GitLab, please make sure that I have access to your source code (basically, that I am allowed to clone it). Some guest roles do not have sufficient permissions. You should give me a role corresponding to a developer or similar.
    * On `github.com`, my account is `hubertbau`. On `gitlab.gbar.dtu.dk` my account is `huba`. On `gitlab.comm` my account is `huba63`.
    * Public read access for everybody enabled will also work. I don't need to change your repository.

  - detailed installation guide, which explains how to build, run, and test your version of DTU Pay
  - precise information on software tools used for development (Link for downloads, precise versions, how to install, ...)

- Misc:
  - Please also make sure that I still can login as root on your virtual machine using the SSH keys I have provided to you.
  - Create a user `huba` on your Jenkins server and provide me with the password in the installation guide.