Project report in IDATG22004

By Oddbjørn Borge Jensen, Nicholas Bodvin Sellevåg og Odd Hald Bliksås

Introduction

Links:

- Repository link
- SSH clone: git@git.gvk.idi.ntnu.no:course/idatg2204/idatg2204-2021-workspace/oddhb/idatg2204-prosjekt.git
- HTTPS clone: https://git.gvk.idi.ntnu.no/course/idatg2204/idatg2204-2021-workspace/oddhb/idatg2204-prosjekt.git
- URI: https://git.gvk.idi.ntnu.no/course/idatg2204/idatg2204-2021-workspace/oddhb/idatg2204-prosjekt

Repo structure

In the folder dokumenter located in the root of the project you will find all the documents and the prod database:

- **Manual.pdf** - Installation guide, same information as provided in this report

- **Endpoint_design.pdf** - Description of all the endpoints
- **Model_conceptua**l - The conceptual model
- **Model_logical.txt** - The logical model. Be aware that the logical model is from a previous version and can therefore be considered as outdated.
- **Model_physical.sql** - This is the physical model. This file is used as the production database.
- **Testcase_description.pdf** - A part of a milestone, more updated in this report.

Project production code is centralized into the folders **controller** and **db**. It is notewhile that the files **api.php** and **constants.php** also serves as production code.

The file **PDO.Testing.php** and the folder **tests** are related to testing.

The folder tests has many subdirectories, some om the most important subdirectories and files:

- **_data** - location of test database
- **Api** - contains all the api tests created
- **Unit** - contain all the unit test created
- **Api.suite.yml** - Defines libraries and environment variables for the API tester element
- **Unit.suite.yml** - Defines libraries and environment variables for the UNIT tester element

The product

Some of the code is inspired or directly copied from Rune Hjelsvold. Such instances of copied code are marked with Rune Hjelsvold as the author. The cookie check and implementation in the API test is directly copied from Rune Hjelsvold.

Database design

The models are located inside the “dokumenter” folder within our repository
Design and implementation of the database has been an iterative process and based on the project description given by Rune Hjelsvold. Moreover our group has created three models for the conceptual, logical and physical description of the database. The initial architectural design for our database does not match the final product, moreover feedback from the peer review induced a thorough redesign with improvements and corrections. For example, our first draft of the database proved to be lacking associations from orders and customer representatives.

Firstly, our group created a conceptual model to convey business logic and functionalities visually. Common understanding of the fundamental principles for our system became the first goal and was accomplished through design of the conceptual model. Furthermore, emphasis on design and documentation has from previous endeavors to be cost effective. “You can use an eraser on the drafting table or a sledgehammer on the construction site.” - Frank Lloyd Wright.

Secondly, we defined elements and data required to manifest our conceptual ideas into a logical model. Considering time constraints for the project, one could argue that our group should have bypassed developing the logical model and focused on the physical instead. Despite this, our group collectively agreed that the logical model provides an important foundation and structure of business data which the physical model can be created from.

Lastly, our group embodied the system case into a physical model. The physical model details how our system is to be implemented into our chosen database management system.

Overall application design

Application is designed to support the ski manufacturing and delivery process in the form of a RESTful API and a database. For example, the ski manufacturer and companies need to create and deliver skis based on their register of orders from customers. In addition, there are multiple types of employees from different departments that may interact with our application. With this in mind functionalities of our application are distributed over endpoints for each of our user types. At this point the application contains:

- Database for business data
- Endpoint for all user groups with affiliated functionalities.
    - Public
    - Storekeeper
    - Transporter
    - Customer
    - Customer representative

All transactions towards the RESTful API are expected to contain a token to identify the usertype and consequently their privilege to endpoints. In some cases endpoints require additional information for a successful transaction and may be provided inside the request body. A transaction will always result in one of three cases:

- Success
- API exception
- Business exception

If a request generates an error the transaction is regarded as unsuccessful. Errors are defined by the business logic or API design. For example, if a transaction tries to reach a non existing endpoint an API error is generated. On the other hand, if the transaction lacks data vital to the process it is a business exception. In short, all errors that are related to the business logic are defined as business errors and problems with the API generate an according exception. In contrast, any transaction that does not generate errors is considered successful. A complete transaction involves the following:

- Request resource from API
    - If connection to database is not establish generate API exception
- API forward request to endpoint
    - If endpoint does not exist generate API exception
    - If token does not match endpoint generate API exception

- Endpoint handles and responds to request
    - If information in request misses vital data generate Business exception
    - If interactions with database causes problems generate API exception

Repository layout strives to adhere to the application design. For example, all requests are initially taken to **api.php** which sends the request to the appropriate endpoint which handles the request further. Endpoints are located inside the **controller** folder as they “control” access to application functionalities. Moreover each endpoint verifies which resource that is trying to be accessed and controls information about the request, ultimately the request is sent to a model. Models are used to do data operations on a database and we have created a **db** folder that holds all models which supports our application functionalities.

Implementation

Setting up the environment

In this setup we are assuming that the composer is already installed.

Step 1 - Download git repo

git clone git@git.gvk.idi.ntnu.no:course/idatg2204/idatg2204-2021-workspace/oddhb/idatg2204-prosjekt.git

Step 2 - Install codeception in root

composer require codeception/codeception --dev
php vendor/bin/codecept bootstrap

Step 3 - Install dependencies

composer require codeception/module-rest --dev
composer require codeception/module-db --dev
composer require codeception/module-phpbrowser --dev

Step 4 - Setup dbCredentials

We have not yet implemented database users, so for now the connection from the database to the sql server is running with root permission.

1. From the root of the project directory navigate to the db folder and open the file DBCredentials.php
2. Edit the consts DB_HOSTS, DB_NAME, DB_USER, DB_PWD to match your system settings

a. By default dbCredentials will use the root user and an empty password. You may not have to edit it if you use myAdmin as root.

3. TEST constant toggles whether to use test or production db. Set to 1 for test and to 0 for prod.

Step 5 - Set up www environment

Create a new folder called **prosjekt3** in your www root folder
Copy **db/**, **controller/**, **api.php, constant.php** from your repo to the prosjekt3 folder
Create a .htaccess file and paste in the following:

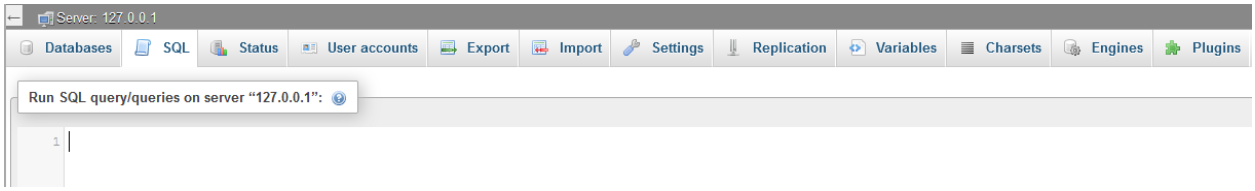
```
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule prosjekt3/(.*)$ prosjekt3/api.php?request=$1 [QSA,NC,L]
</IfModule>
```

Please note that what you specify in the .htaccess file will be the start of your URI. So in the example above the URI will be http://localhost/prosjekt3

Step 6 - Set up database

Production database:

1. Create a new database called ski_manufacturer
2. Copy everything in **dokumenter/model_physical.sql** (ctrl+A -> ctrl+C).
3. Navigate to the SQL tab in phpMyAdmin



4. Paste and run.

Test database:

1. Create new database called **testdb** in phpMyAdmin *(Mandatory)*
2. Import **tests/_data/testdb.sql** to **testdb** *(Optional)*

Step 7 - Tests

All tests for the project have been generated into one of two "suites". Namely the "unit" and "api" suite. Use following commands to run the according tests:

a. **ALL** tests: **php vendor/bin/codecept run**
b. **UNIT** tests: **php vendor/bin/codecept run unit**
c. **API** tests: **php vendor/bin/codecept run api**

NB! Before running any **API and UNIT** tests make sure that the appropriate database is chosen. For example, when running **API and UNIT** tests make sure that the variable DB_NAME in *dbCredentials.php* is:
const DB_NAME = 'testdb';

Update api.suite.yml og unit.suite.yml

Please change db information in the file api.suite.yml and unit.suite.yml located in the folder test with the proper db name and credentials if needed.

Authentication

We have implemented token based authentication on all APIs without the /public API. Please use the right token for the right department and API. The user is checked with a ACL for checking if the user is allowed to enter the specific endpoint. The check is based on what department a user is in.

Test users:

The users listed under is already in the database and can be used for accessing the specific endpoints

User	department	token
Sylvester Sølvtunge	customer-rep	839d6517ec104e2c70ce1da1d86b1d89c5f547b666adcdd824456c9756c7e261
Njalle Nøysom	production-planner	022224c9a11805494a77796d671bec4c5bae495af78e906694018dbbc39bf2cd
Didrik Disk	storekeeper	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Lars Monsen	customer	2927ebdf56c20cbb90fbd85cac5be30d60e3dfb9f9c9eda869d0fdce36043a85

ACL

Here is a list of what usertype /department is allowed to access the specified APIs:

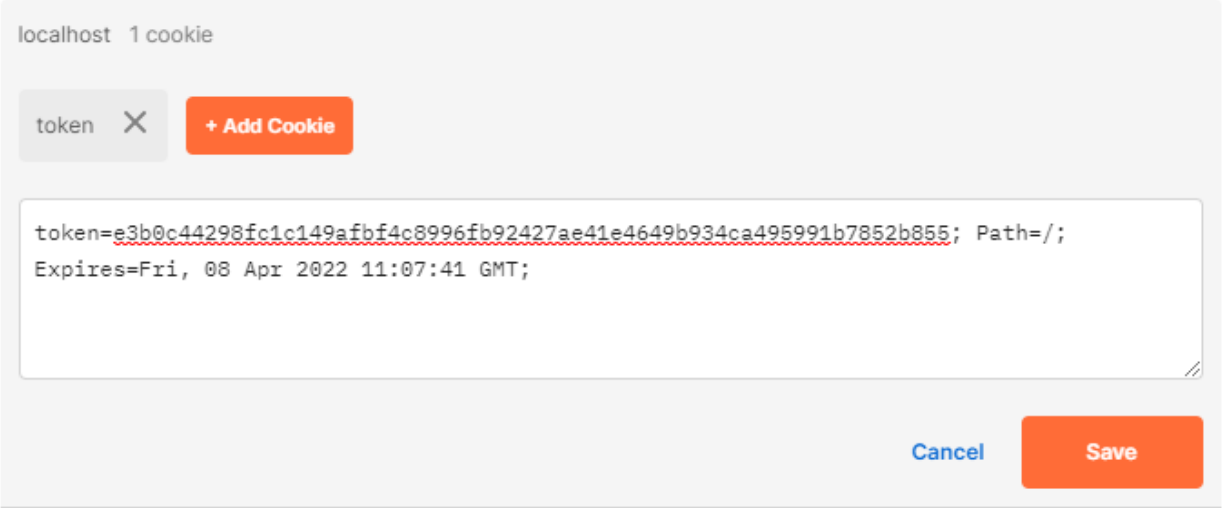
Api (start of the api uri)	Department / usertype allowed to acess
/orders	customer
/customer	customer
/shipment	storekeeper
/production-plans	production-plans
/public	Available for everyone
/storekeeper	storekeeper
/customer-rep	customer-rep

Accessing the API with cookie

For authentication go to postman and add a cookie.
Example:

1. In postman click **Cookies** (located under the **Send** button)
2. In the field “Type domain name” type **localhost** and click **add**
3. Select localhost and click **Add cookie**
4. Create a new cookie and paste in the following:

token=e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855; Path=/; Expires=Fri, 08 Apr 2022 09:27:14 GMT;



NB: This is the storekeeper token and only has access to the storekeeper endpoint.

5. Send a GET request to *http://localhost/prosjekt3/storekeeper/orders* to see if the token is working as it should.

What is implemented:

APIs/ endpoints:

Find and use the appropriate token for which endpoint you are testing. For example, use the tokens written in the “Test users” table in all requests to the API. Check the tabe ACL for what usertype that is allowed to access the endpoint .

We haven't had time to implement the functionality “create new production plans” under Planner endpoint.

Public:

There is implemented functionality to retrieve skis by model, grip or both.

Find all skies of a certain model:
Method: GET
/public/skis?model=<*model*>
Working example: public/skis?model=Redline

Find all skies of a certain grip:
Method: GET
public/skis?grip=<*grip*>
Working example: public/skis?grip=IntelliWax

Find all skis of a certain model and grip
Method: GET
/public/skis?model=<*model*>&grip=<*grip*>

Working example:
/public/skis?model=Redline&grip=IntelliWax

Storekeeper

Retrieve all orders
Method: GET

URI: /storekeeper/orders

Create new ski
Method: POST
URI: /storekeeper/ski

Example Body

```
{
    "ski_type": 2,
    "manufactured_date": ""
}
```

Create a transition record for the order when more skis are assigned to the order
Method: PUT
URI: /storekeeper/transitionrecord

Example body

```
{
    "orderNumber": 1,
    "serialNr": 20
}
```

Customer

Place order:
URI /customer/<customerId>/order
Method: POST
Example uri: /customer/<customerID>/order

Example Body

```
{
    "customer": 2,
    "skis": [
        {
            "type": 2,
            "quantity": 4
        },
        {
            "type": 3,
```

```
            "quantity": 5
        }
    ]
}
```

Delete a record of order:
URI /customer/<customerId>/order/<orderId>
Method: DELETE
Working example: /customer/2/order/2

Get an order:
URI: /customer/<customerId>/order/<orderId>
Method: GET
Working example: /customer/2/order/6

Get an order since date:
URI: /customer/<customerId>/orderSince?since=date
Method: GET
Working example: /customer/2/orderSince?since=”2021-03-22”

Split an order
URI: /customer/<customerID>/split/<orderID>
METHOD: POST
Working example: /customer/1/split/7

Retrieve a four week production plan summary

URI: /customer/plansummary

METHOD: GET

Customer-rep

Retrieve orders with status filter
URI: /customer-rep/orders?status=<status>
Method: GET

Example orders with status filter new:
Working example: /customer-rep/orders?status=new

Example orders with status filter skis-available:
URI: /customer-rep/orders?status=skis-available

Examples orders with multiple filter status:

/customer-rep/orders?status=new,skis-available

Change the order state from new to open for an unassigned order
URI: /customer-rep/state
Method: PUT
Format:

```
{
   "orderNumber": int,
   "status": "open"
}
```

Working example:

```
{
   "orderNumber": 1,
   "status": "open"
}
```

Transporter - shipment endpoint

Update a record of shipment:
URI /shipment/<order_nr>
Method: Post
Working example: /shipment/4

```
{
   "shipping_address" : "tesaddresse",
   "scheduled_pickup" : "2021-04-05",
   "transporter": "Flyttegutta A/S",
   "driver_id": 1
}
```

Tests

We have implemented various tests based on API and UNIT tests. However there are a lot more API tests than UNIT tests.
One of the reasons that we have implemented more API than UNIT tests is because of the application type. The application we have created is a backend application with APIs. We have designed the test as what customers may or may not do with houer application.

But we should have implemented more Unit tests. The unit tests is a faster way to implement and we can test the code logic in a better and derailed way.
The main reason for not implementing more UNIT test (as we should) is the timeframe.

API

Authentication

AuthenticationTestCest

This api test checks if we can access the APIs with a valid token.

Public endpoints

publicEndpointCest.php

This api checks if we find a ski with a certain grip, find a skis with a certain model and skis with both model and grip.

Customer rep endpoints

customerRepCest.php

Checks that the filter for finding orders with status ne, status skis-avaiable, thath we can compine to retrieve both skis with status new and ski-avaible. We have also a function thath test thath we get a proper error if we dosent find any matches.
The function changeOrderStateFromNewToOpen cheks that we successfully can change a order from state new to open. The last function changeOrderStateIlllleagel trys to change the order to a illegal state.

Storekeeper

storekeeperTransitionCest.php

This API test is testing that we can set a transitionrecord on skis produced. The first function test to set a transitonRecordSucess, we have also a testfunction thath test to set the transitionrecord to non existing ordernumber (This function is called transitionRecordNotexistingOrderNumber)
The last testfunction in this test try to set a transition record on nonexistent skis.

CustomerCest.php

Test has the purpose of trying all functionalities for the customer endpoint.

Checks CustomerModel's ability to:

- Retrieve an order
- Create an order
- Delete an order

UpdateShipmentCest.php

Update order with status of "skis-available" to shipped, update history of order, lastly create shipment record in the shipments table.

UNIT

ShipmentTableTest.php

Test has the purpose of checking that there is a record inside the shipments table with **shipment_nr** = 1 which has a **state** of picked-up.

Storekeeper

StorekeeperUnitTest.php

Test has the purpose of checking that there is a record inside the orders table with
order_nr = 3
ski_type = 3
ski_quantity of 30
price = 32175
State = open
Customer_id of 3
Date_placed = 2021-03-19

Customer-rep

customerRepTest-php

This unit test tests that we can change an order state successfully (changeOrderState) and what happens if we try to change an order on a nonexistent order number. (testChangeOrderStateIllegal)

Discussion

Database design assessment

Production Plans

Our initial idea for implementing production plans was to store the type and amount of ski to be made every single day within the four week period, with a PK of (day, ski_type). This was fine as an idea, but actually implementing it for a test db proved very impractical. It would probably have been better - for this project specifically - to implement the production plans on a period by period basis, as opposed to day by day. This way we could still represent the amount to make per day by simply dividing the amount to make within a certain period by the amount of days the period consists of - all while requiring a lot less work to test.

Orders and suborders

When the project description described a relationship between orders and larger orders, our first interpretation was to have orders only contain a single type of ski, and providing the ability to place several orders under a larger order - referred to as an “order aggregate”. The semantics of only allowing a single type of ski per order didn’t sit well with us, and over time we started leaning towards having orders serve the purpose of the aggregates, and rather having them consist of several “suborders” that only contained the type and quantity of ski to buy.

This also allowed us to store information about the customer and total price in a single place, reducing the redundancy our initial implementation created. Forcing the order - suborder relationship even when there is only one suborder also means we always know suborders are part of an order, and that all orders contain at least one suborder. This was not the case for aggregates, as orders did not necessarily have to be part of one.

Customer specializations

We chose to implement the customer specializations as a general customers table which we could join with specialized franchise, store, and team skier tables for data that doesn’t apply to all customer types. This is not the customary solution for {mandatory, or} specializations, but we found it to be more readable than having a single customers table with every possible field and a bunch of NULL values. Practically speaking there is very little difference, so we chose to go with the method we preferred. It’s technically possible to create a customer with no specialization in the database, but policy can still demand that this should never be the case.

Security assessment

In the project we have had the security in our mind when developing the application but we have some points we need to improve on next time.

If we first take a look at the security functionality used in our code and after that reflect on some improvements. If we look at the code in general there is use of prepared statements and bound values in all of the queries to the database, this is used for decreasing the probability of SQL injection. With the bind and prepare statement the PDO verifies the input before it is validated and executed in the database. In the code where we have multiple inserts in the database we have used transactions. If some of the query's failed running on the database we can then do a rollback. An example is in the customer rep model in the function changeOrderState.

The project uses authentication and authorization for all the APIs beside the public. Each user has a unique token. When a user try to access a API we take the uri and the token and compare it with a couple of things. First based on the token we find what type of user (example customer, storekeeper etc) if the token is not set we are only allowing the request to the public API. After we have the type of user we check in a ACL list wherever the usertype is allowed to access the resource, if the usertype is in the ACL list the request is passed and if the usertype is not allowed to access the resource we stop the request to be processed. (The ACL list is just arrays with user types and can be found in the controller folder AuthenticationEndpoint.php under the function aclList.

Peer review feedback on security topics

From the peer review feedbacks on the security we got some feedbacks which can be summarised:

- Some functions were prone for sql injection
- There shouldn't be a unique token for each user. Just for each usertype
- No error handling

The first feedback of some of the functions prone to sql injection was taken seriously and was fixed by implementing and prepared statements and bound values.

The feedback based on “There shouldn't be a unique token for each user” was discussed inside the group and we reached the conclusion that we’d prefer to keep the unique tokens, as it is a good way to distinguish the users from each other.

The last part about error handling was taken into account and fixed following the feedback. It was already on the todo list, but we couldn’t implement it in time for the peer review.

Product assessment - strengths and weaknesses

Strengths

- Well formatted and commented code
- Use of exceptions
- SQL security (transactions, prepared statements, bound values)

Weaknesses

- Runs as root
- Manual deployment
- Lacking error handling

We have spotted some weaknesses in ouer code. The most important weakness is the SQL user running as root. This is the first thing we would improve if we had more time. Another weakness is the lack of automatic deploy

Future improvements

We should have created an automatic deployment of the project code (but this is a topic none of the project attendants have learned).

In the test topic we should have implemented more UNIT test for more details tests on the code. We should have been better at updating the logical model when we updated the conceptual and physical model. In the security topics we should have created more sql users based on the principle “least privilege” and assigned those users to the specific endpoints, as currently all functionality runs as root - which is not exactly great. In the authorization and authentication part we should create the ACL list in the database. The token should be generated with a salt algorithm for less predictable hashing.

Due to admittedly poor planning on our part design-wise, the production plans have not been properly implemented beyond existing as a table in the database. Our implementation proved too unwieldy to develop in a short amount of time, and would definitely be tackled differently if we were to do it again. Rethinking the design and implementing the functionality of the production plans would be a high priority task for future development.

As the system exists at the moment, you do not need the appropriate customer token to access a specific customer’s orders. That is, any customer can access the orders of any other customer. An obvious future improvement would be to check the database for whether the customer actually has the *correct* customer token, and not just *a* customer token.

Another improvement would be to get rid of the awkward manual change from test db to prod db, though finding a practical *and* secure solution is not a headache we’re tackling right now.