Міністерство освіти і науки України

Національний технічний університет України «Київський політехнічний

інститут імені Ігоря Сікорського"

Факультет інформатики та обчислювальної техніки

Катедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 2 з дисципліни

«Компоненти програмної інженерії. Частина 3.

Архітектура програмного забезпечення»

«Формування та документація технічних рішень додатку»

Виконали студенти ІП-11 Лесів Владислав, Прищепа Владислав

Перевірив Баришич Лука Маріянович

Київ 2023

**Лабораторна робота 2**

**Формування та документація технічних рішень додатку**

**(System/Software design)**

**Мета**: навчитися формувати та документувати базові технічні рішення з архітектури додатку, зокрема, основні компоненти (сервіси), зв’язки між компонентами, дизайн API.

**Завдання:**

1. Визначити нефункціональні вимоги;
2. Визначити основні компоненти системи. Визначити загальні зв’язки між компонентами та зобразити на діаграмі;
3. Визначити правила опису API (API design guidelines). Вибрати та розібрати існуючий guideline або розробити власний;
4. Описати API компонент, обов’язково використовуючи обраний guideline;
   1. За бажання реалізувати демонстраційний сервіс в обсязі Proof Of Concept;
   2. Надати Swagger для документації API;

**Хід роботи.**

1. Nonfunctional requirements:

* The system must have a simple and intuitive user interface that allows customers to search for products using search bar, browse product details, and complete purchases with minimal effort or confusion. The system must also ensure equal access for all customers.

|  |  |
| --- | --- |
| Usability scenario | |
| Source | Customer |
| Stimulus | Search for furniture to order |
| Artifact | Vlads` furniture store system, customer mode |
| Environment | The search bar was selected, some text is being entered |
| Response | A list of furniture, whose names contain the entered text |
| Response time | * The search is being processed – within 500 ms; * List of found items – within 3 s. |

* The system must have a minimum uptime of 99% over a 30-day period, with scheduled maintenance windows that do not exceed 1.68 hour per week. In the event of system downtime, the system must provide a meaningful error message and an estimated time for resolution.

|  |  |
| --- | --- |
| Availability scenario | |
| Source | Customer, software |
| Stimulus | Fault: customer can not use the system due to furniture database update |
| Artifact | customer mode, persistent storage |
| Environment | Customer`s operation on searching for furniture |
| Response | Prevent the fault from becoming a failure:   * Log the fault; * Make a search option temporarily unavailable while database is updating; * Operate in a degraded mode: customer can see his previous and current orders, but can not search for furniture or get details about furniture in his\her order. |
| Response time | * Time to detect a fault (closed access to the database) * “The database is updating, you can not use it now” – within 500 ms after detecting a fault; * Estimated time of repairing; * Every minute – update an estimated time in a customer`s mode. |

* The system should be able to handle a sustained load of 5000 concurrent users during peak shopping period and maintain a response time of under 5 seconds for 95% of all transactions.

|  |  |
| --- | --- |
| Performance scenario | |
| Source | Customer |
| Stimulus | Big amount of customers` operations in the system at a time |
| Artifact | Vlads` furniture store system, customer mode |
| Environment | Big amount of customers (nearly 5000). Peak load mode. |
| Response | Little delay of response on customer`s of under 5 seconds |
| Response measure | * Create an order of response for operations; * Divide it in parts (up to 250 operations); * Respond on each group within 250 ms. |

* The system must comply with PCI DSS and encrypt all customer and payment information in transit and at rest.

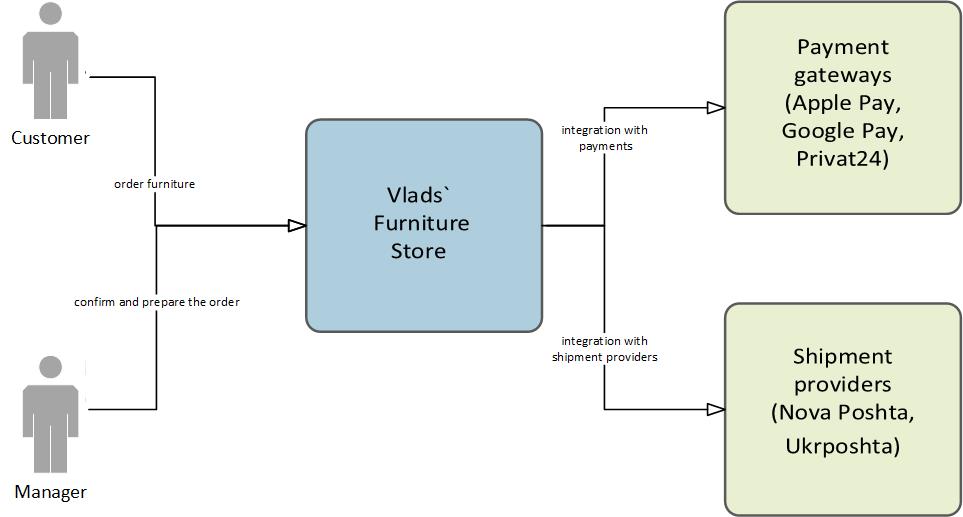
|  |  |
| --- | --- |
| Security scenario | |
| Source | Customer |
| Stimulus | Customer confirms the payment |
| Artifact | Vlads` furniture store system, customer mode |
| Environment | Customer has made a list of goods for order and entered the bank card data |
| Response | System sends order to a bank for needed sum |
| Response measure | * System encrypts customers and payment information according to PCI DSS standard; * System sends the crypt to the bank. |

1. The main components of the system:

* Customer`s and Manager`s versions of the mobile app to interact with;
* Auth with the use of SAML to ensure that each customer's data is protected reliably;
* Order Processing -> Customer`s order creation and Manager`s order preparation to implement the main functions of the application and provide a connection between the client part and the manager part;
* Payment to make a secure payment using an external payment system;
* Shipment to ensure that the delivery address is selected and the delivery process is carried out using an external shipment provider's system;
* Notification for users to quickly learn about important changes in order status.

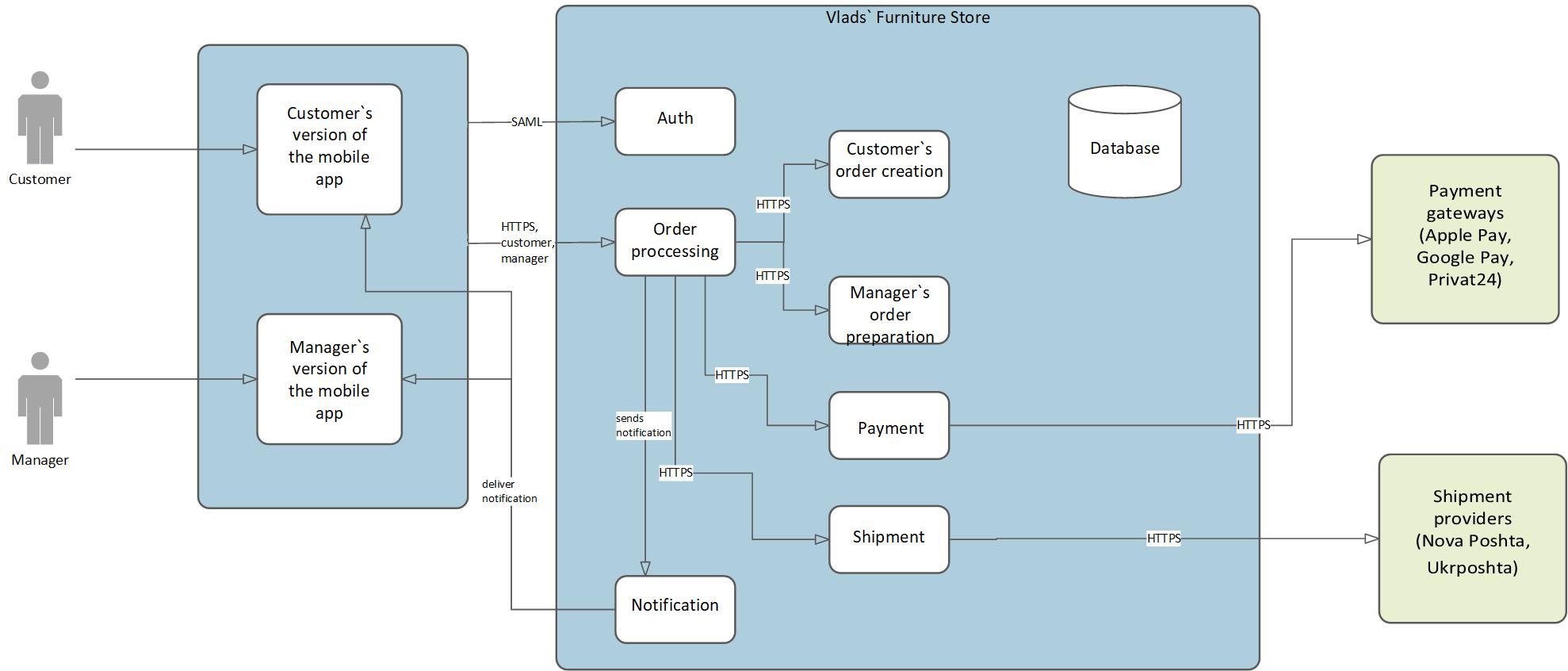
**C4 diagram, levels 1 and 2.**

**Level 1. Context diagram.**



There are two people categories: customers and managers. Customers use Vlads` Furniture Store system to order furniture; manager – to confirm and prepare the order. The Vlads` Furniture Store system integrates with payment systems (such as Google pay etc.) to allow customer to make payment; and shipment providers (Ukrposhta etc.) to allow manager to confirm shipment of an order.

**Level 2. Container diagram.**



Customer and manager use mobile phone apps to make operations in Vlads` Furniture Store system. Customer uses version of app for customers; manager – version for managers. Both customer and manager log in the system via SAML standart. Then users make operations for order processing in the system via HTTPS protocols. Customer can create order and do payment, manager can prepare order and confirm shipment. Order processing in the system can send notification both to customer and manager. After confirming a payment the system integrates with a chosen payment gateway system. After confirming a shipment the system integrates with a chosen shipment provider system.

1. API design guideline.

Set of general rules and recommendations that have to be followed along the entire API lifecycle of any API regardless of its type:

**Introduction**

* Everyone *must* follow the API First principle. A development of an API *must* always start with API design without any upfront coding activities. API design is the master of truth, not the API implementation. API implementation *must* always be compliant to particular API design which represents the contract between API, and it's consumer.
* Every API *must* be described using an API description format. The API description format used *must* be the OpenAPI Specification. The API description *must* be written in American English.

**Protocol of operation**

* Every API *must* support HTTP/1.1 and *must* adhere to its semantic. Every API *must* require secure connections with TLS 1.2. That is, an API using the HTTP protocol *must* use HTTPS. Any non-TLS requests *should* be ignored. In HTTP environments where this is not possible, a non-TLS request *should* result in the 403 Forbidden response.

**Data format**

* The *preferred* data format for the API is JSON. All API responses and request bodies *should* be formatted as valid JSON.

Any JSON-based message *must* conform to the following rules:

1. All JSON field names *must* follow the [Naming Conventions](https://adidas.gitbook.io/api-guidelines/rest-api-guidelines/evolution/naming-conventions) (camelCase, American English, etc.)
2. Field names *must* be ASCII alpha num characters, underscore (\_) or dollar sign ($)
3. Boolean fields *must* *not* be of null value
4. Fields with null value *should* be omitted
5. Empty arrays and objects *should* *not* be null (use [] or {} instead)
6. Array field names *should* be plural (e.g. "orders": [])

**API organization around resources.**

* Every API design *must* be resource-centric. That is an API design *MUST* revolve around Web-styled resources, relations between the resources and the actions the resources may. There *must* be a REST-like API with standard HTTP methods (GET, POST, PUT, DELETE) for data retrieval, creation, updating, and deletion.

**Naming rules**

* Resource names *must* be plural nouns (e.g. /products, /orders). Field names *should* be in camelCase (e.g. productName). URLs *must* use underscores to separate words in multi-word resource names (e.g. /product\_categories).
* Verbs used in URL paths *must* match the HTTP method used (e.g. /products for GET, /products/{id} for PUT, DELETE)
* Query parameters *should* use lowercase letters and underscores (e.g. /products?category=furniture&sort=price)
* Field names *should* be descriptive and meaningful (e.g. product\_name instead of p\_name or name)

**API versioning rules**

* The API version number *must* always be included in the URL to allow for future changes without breaking existing clients.
* The version number format *must* be /api/v{version\_number}/resource, using semantic versioning (major.minor.patch) to indicate the significance of changes. For example: /api/v1/products, /api/v2/customers.
* The version number *must* be incremented when making changes to the API that could break existing clients.
* Documentation *must* be provided for each version of the API, including release notes and migration guides, and *must* be updated whenever a new version is released.
* Breaking changes to the API *should* be avoided too frequently. Instead, consider using deprecation warnings and a gradual transition period for deprecated features.

**HTTP methods usage**

* The standard HTTP methods (GET, POST, PUT, DELETE) *must* be used appropriately and consistently throughout the API.
* The GET method *must* be used to retrieve data from the API, and *must* *not* modify any resources on the server. This includes retrieving furniture items and user data. The POST method *must* be used to create new resources on the server, and *must* return the newly created resource. This includes creating new user accounts. The PUT method *must* be used to update existing resources on the server. This includes updating furniture item details and user account information. The DELETE method *must* be used to remove resources from the server. This includes removing user accounts.

**Custom methods usage**

* Custom methods refer to API methods besides the 5 standard methods. They *should* only be used for functionality that cannot be easily expressed via standard methods A custom method *can* be associated with a resource, a collection, or a service. It *may* take an arbitrary request and return an arbitrary response, and also supports streaming request and response. Custom method names *must* follow method naming conventions.
* The following guidelines *shall* be applied when choosing the HTTP mapping:
  + Custom methods *should* use HTTP POST verb since it has the most flexible semantics, except for methods serving as an alternative get or list which *may* use GET when possible.
  + The request message field(s) receiving the resource name of the resource or collection with which the custom method is associated *should* map to the URL path.
  + If the HTTP verb used for the custom method does not accept an HTTP request body (GET, DELETE), the HTTP configuration of such method *must not* use the body clause at all, and all remaining request message fields shall map to the URL query parameters.

**Request/response payload**

1. **GET**

Request Payload format:

* Use query parameters to specify filters and sorting criteria. Query parameters *should* be optional and their values *should* be URL encoded. The following query parameters are *recommended*:
  + `page`: Specifies the page number to retrieve when using pagination.
  + `limit`: Specifies the number of items to retrieve per page when using pagination. The default value *should* be reasonable and allow for efficient retrieval of results.
  + `sort`: Specifies the field to sort the results by and the order of sorting. Multiple fields *can* be specified using a comma-separated list.

Additional parameters can be added as needed to support specific use cases.

Response Payload format:

* The response *should* be a JSON object containing the retrieved items or an empty array if no items are found.
* When using pagination, the response *should* also contain metadata about the pagination:
  + page: The current page number.
  + total\_pages: The total number of pages.
  + total\_items: The total number of items.
* The response *should* include only the necessary fields for each item. Additional fields *can* be retrieved by using a separate API endpoint or specifying them in the query parameters.
* The API *should* return appropriate HTTP status codes and error messages when errors occur. For example, when no results are found, the API should return a 404 status code and an error message explaining that no results were found.

1. **POST**

Request Payload format:

* The request *should* be a JSON object containing the data to create a new item.
* The required fields *should* be specified in the request body and *should* adhere to the API's naming conventions.

Response Payload format:

* The response *should* be a JSON object containing the ID of the newly created item.
* The response *should* include only the necessary fields to identify the newly created item.

The API *should* return appropriate HTTP status codes and error messages when errors occur. For example, when the request payload is invalid or incomplete, the API *should* return a 400 status code and an error message explaining the reason for the error.

1. **PUT**

Request Payload format:

* The request should be a JSON object containing the data to update an existing item.
* The request should include the ID of the item to update.
* The request should include only the necessary fields to update the item, and any fields not included in the request should remain unchanged.
* The API should validate the request payload and return appropriate HTTP status codes and error messages when errors occur.

Response Payload format:

* The response should be a JSON object containing the ID of the updated item.
* The response should include only the necessary fields to identify the updated item.

The API *should* return appropriate HTTP status codes and error messages when errors occur. For example, when the request payload is invalid or incomplete, the API *should* return a 400 status code and an error message explaining the reason for the error.

1. **DELETE**

Request Payload format:

* The DELETE request *should not* include a request payload. Instead, the item to delete *should* be identified by its ID in the URL.

Response Payload format:

* The response *should* be a JSON object containing the ID of the deleted item.
* The response *should* include only the necessary fields to identify the deleted item.
* The API *should* return appropriate HTTP status codes and error messages when errors occur. For example, when the requested item does not exist, the API *should* return a 404 status code and an error message explaining the reason for the error.

Note: Since DELETE requests do not require a request payload, the API should expect the ID of the item to delete to be included in the URL. The response *should* include only the necessary fields to identify the deleted item. Additional details about the deleted item *will no* longer be available in the API.

**Status Codes**

* Every API *must* use the appropriate HTTP Status Codes to communicate the result of a request operation.
* Every API designer, implementer and consumer *must* understand the semantic of the HTTP Status Code she is using.

Here are some rules for possible status codes and their explanations:

* 200 OK.

This status code indicates that the request was successful and that the response body contains the requested data.

* 201 Created.

This status code is returned when a new resource is successfully created.

The response body *should* contain the details of the newly created resource, including its ID.

* 204 No Content

This status code indicates that the request was successful, but there is no response body.

This is typically used for DELETE requests.

* 400 Bad Request

This status code is returned when the request is malformed or invalid.

The response body should contain a description of the error.

* 404 Not Found

This status code is returned when the requested resource cannot be found.

The response body should contain a description of the error and how to resolve it.

* 405 Method Not Allowed

This status code is returned when the requested HTTP method is not supported for the requested resource.

The response body should contain a description of the error and how to resolve it.

* 500 Internal Server Error

This status code is returned when an unexpected error occurs on the server.

The response body should contain a description of the error and how to resolve it.

Note: The status codes and their explanations *should* be consistent across all endpoints of the API, and *should* follow industry standards wherever possible.

1. Description of API components. The description is presented in Swagger format. We have two services:
2. Customer service, which includes such resources as furniture, stores and customers.
3. Manager service, which includes such resources as post\_invoices and managers.

Depending on the resource, we can take appropriate actions. With furniture, we can add a new item, update existing furniture, find an item by id, update it with a certain data, upload an image to represent it and delete a furniture.

As for stores, we can place an order, find it by id and delete it. Customers: we can create it, do a login, logout, get, update and delete customer by its email.

Looking on post\_invoices, we can create one, update it, find it by id and delete it. Lastly, we can create a manager, log them in, log them out and update them.

The code for the Swagger is provided in file that is attached with the laboratory work.

**Висновок.**

Отже, у цій роботі ми отримали навички формування та документації технічних рішень додатку. Ми навчились формувати та документувати базові технічні рішення з архітектури додатку, зокрема, основні компоненти (сервіси), зв’язки між компонентами, дизайн API. У результаті лабораторної роботи відповідно до проєкту для розробки - Платформи е-комерції для магазину меблів - ми визначили нефункціональні вимоги, основні компоненти системи, правила опису API design guidelines та описали основні компоненти API у форматі Swagger. Використовуючи засоби специфікування й програмний засіб MS Visio для побудови діаграм, Swagger Editor для дизайну API у форматі Swagger, отримуємо коректний результат.