# Just Trade It!

After opening your mailbox, you find an odd letter containing a cryptic message. After hours working on decrypting the letter, you are finally able to make sense of it. The letter states the following:

“Hi,

The world needs a platform to be able to trade valuable items among themselves. I want you to make a backend which supports this platform. Contact me by calling the reception at BC Corp and ask for the janitor.

Best regards,

J”

We have decided to contact this person and make the platform. It will be a tall task, but I think we are up for it.

## Assignment description

In this assignment we are going to build a microservice structure which consists of four services. Below each service will be added to a specific section with all its requirements.

## Web API (75%)

The web API should be written in **.NET** and the requirements for the API are the following:

### Authentication using JWT (10%)

- All endpoints should require authentication excluding: **(2%)**
  - **/api/account/register [POST]**
  - **/api/account/signin [POST]**
- The authentication scheme used is **JWT (8%)**
  - Configuration settings should be stored in **appsettings.json (1%)**
  - The authentication scheme should be setup using a middleware **(4%)**
  - The middleware should setup the **OnTokenValidated** event which checks whether the token is blacklisted or not **(2%)**
  - When the middleware has been implemented it should be registered within **Startup.cs (1%)**

### Endpoints (15%)

Each controller should make use of a corresponding service in order to retrieve data from the database, or other sources.

- AccountController **(5%)**
  - **/api/account/register [POST]** - Registers a user within the application, see **Models** section for reference
  - **/api/account/login [POST]** - Signs the user in by checking the credentials provided and issuing a JWT token in return, see **Models** section for reference
  - **/api/account/logout [GET]** - Logs the user out by voiding the provided JWT token using the id found within the claim
  - **/api/account/profile [GET]** - Gets the profile information associated with the authenticated user
  - **/api/account/profile [PUT]** - Updates the profile information associated with the authenticated user
- TradeController **(4%)**
  - **/api/trades [GET]** - Gets all trades associated with the authenticated user
  - **/api/trades [POST]** - Requests a trade to a particular user. Trade proposal always includes at least one item from each participant. Therefore if you want to acquire a certain item, you must offer some of your items which you believe are equally valuable as the desired item
  - **/api/trades/{identifier} [GET]** - Get a detailed version of a trade request

    - **/api/trades/{identifier} [PUT]** - Updates the status of a trade request. Only a participant of the trade offering can update the status of the trade request. Although if the trade request is in a finalized state it cannot be altered. The only non finalized state is the pending state.
- ItemController **(4%)**
    - **/api/items [GET]** - Gets all available items. The result is an envelope containing the results in pages.
    - **/api/items/{identifier} [GET]** - Gets a detailed version of an item by identifier
    - **/api/items [POST]** - Create a new item which will be associated with the authenticated user and other users will see the new item and can request a trade to acquire that item
    - **/api/items [DELETE]** - Delete an item from the inventory of the authenticated user. The item should only be soft deleted from the database. All trade requests which include the deleted item should be marked as cancelled
- UserController **(2%)**
    - **/api/users/{identifier} [GET]** - Get a user profile information
    - **/api/users/{identifier}/trades [GET]** - Get all successful trades associated with a user

## Service project (20%)

All service classes should make use of a corresponding repository class if it is fetching data from the database

- **AccountService (2%)**
    - CreateUser
        - Creates the user using the appropriate repository class
    - AuthenticateUser
        - Authenticates the user using the appropriate repository class
    - Logout
        - Voids the JWT token using the appropriate repository class
    - GetProfileInformation
        - Gets the profile information
    - UpdateProfile
        - Updates the profile information
- **ImageService (5%)**
    - UploadImageToBucket - Uploads an image to AWS bucket and retrieves the URL for the uploaded image and returns it. We will be using the AWS Free Tier (https://aws.amazon.com/free/). See AWS section for more details.
- **ItemService (2.5%)**
    - GetItems

        - Gets all items filtered by incoming arguments. The result should be paginated and wrapped within an envelope. Envelope class is provided within the template
    - GetItemByIdentifier
        - Gets a detailed representation of an item
    - AddNewItem
        - Adds a new item to the authenticated user inventory
    - RemoveItem
        - Removes an item from the authenticated user inventory
- **JwtTokenService (1%)**
    - IsTokenBlacklisted
        - Checks whether a token by id is blacklisted or not
- **QueueService (2.5%)**
    - PublishMessage - Publishes a message to the message broker (**RabbitMQ**) using a routing key and a body
        - Serialize the object to JSON
        - Publish the message using a channel created with the RabbitMQ client
- **TokenService (1%)**
    - GenerateJwtToken
        - Generates a JWT token including all the claims for the authenticated user
- **TradeService (4%)**
    - GetTrades
        - Gets all successful trades for a particular user
    - GetTradeByIdentifier
        - Gets a detailed representation of a trade
    - GetTradeRequests
        - Get all trade requests of the authenticated user
    - CreateTradeRequest
        - Create a new trade request
        - Publish a message to **RabbitMQ** with the routing key ‘new-trade-request’ and include the required data
    - UpdateTradeRequest
        - Update the status of the trade request. Trade requests can only be changed if not in a finalized state
        - Publish a message to **RabbitMQ** with the routing key ‘trade-update-request’ and include the required data
- **UserService (2%)**
    - GetUserInformation
        - Gets profile information on a specific user
    - GetUserTrades
        - Gets all successful trades associated with a specific user

## Repository project (20%)

All repository classes should make use of the **DbContext** in order to retrieve data from the database

- **UserRepository (5%)**
  - CreateUser
    - Check if user with same email exists within the database - if it does do not continue
    - Add a user to the database where the password has been hashed using the hashing function provided
    - Create a new token within the database
    - Return the user
  - AuthenticateUser
    - Check if user has provided the correct credentials by comparing the email and password - if it is not correct do not continue
    - Create a new token within the database
    - Return the user
  - UpdateProfile
    - Updates the profile of the user
  - GetProfileInformation
    - Retrieves the profile information of the authenticated user
  - GetUserInformation
    - Retrieves the profile information of a user by identifier
- **ItemRepository (4%)**
  - GetAllItems
    - Gets all items available as a paginated result using an envelope. The envelope class is provided within the template. The end user should be able to determine the page size, number and the sorting direction
  - GetItemByIdentifier
    - Retrieves an item by identifier containing detailed information regarding the item
  - AddNewItem
    - Adds a new item linked to the authenticated user
  - RemoveItem
    - Removes an item associated with the authenticated user by identifier. If the item which is trying to be removed is not linked to the authenticated user an exception should be thrown
- **TokenRepository (3%)**
  - CreateNewToken
    - Add a new token to the database
  - IsTokenBlacklisted
    - Check to see if the token is blacklisted within the database
  - VoidToken

    - Set the token to blacklisted within the database

- **TradeRepository (8%)**
    - GetTrades
        - Get all completed trades associated with the authenticated user. A trade is considered completed if it is in the ‘Accepted’ state
    - GetTradeByIdentifier
        - Get a trade by an identifier. This should return a detailed representation of the trade
    - GetTradeRequests
        - Get all trade requests associated with the authenticated user. A trade request is a trade, before it reaches an accepted state. Therefore all states excluding ‘Accepted’ are considered a trade request
    - CreateTradeRequest
        - Create a new trade request
        - When a user creates a trade request he can only put his own items as part of the trade request
        - The user which he is trying to trade with must also own the items proposed in the trade request
        - Both users must be valid users within the application
    - UpdateTradeRequest
        - Updates a trade request status. This is done to either cancel, decline or accept an active trade request
        - The only suitable trade request status prior to the update is the ‘Pending’ state. If the trade request is not in the ‘Pending’ state an exception should be thrown
        - If the user is the initiator of the trade, he can only cancel the trade request
        - If the user is the receiver in the trade, he can either accept or decline the trade request
    - GetUserTrades
        - Gets all completed trades by user identifier
        - The user can either be the initiator of the trade or the receiver

## Models (2.5%)

- Setup all **DTOs** (see Models for reference)
- Setup all **InputModels** (see Models for reference)
- Setup all **Entities** (see Database diagram for reference)

## Database (2.5%)

- Create a new **SQLite** database within the root of the entry application
- Add the connection string to **appsettings.json** in the API project
- Setup a **DbContext** for the newly created database in the **Repository** project
- Register the **DbContext** as dependency within the API project

- Create your first migration and update the database according to those migrations

## Dockerfile (5%)

- A Dockerfile should be created in order to run the application in **Docker**
- This file should be located in the root of the application

## Notification service (20%)

The notification service can be written in a programming language of your choice and should include two event handlers.

### New trade request (7.5%)

When a new trade request occurs, the receiver of the trade request should be notified via email and the notification is responsible for listening to the event and send the email to the user

- AMQP listener **(2.5%)**
  - Sets up a queue called **new-trade-queue** which is bound to the **new-trade-request** routing key
- Send an email using **Mailgun** stating that a new trade request has been delivered to his account **(5%)**
  - The email should be setup in a proper manner using HTML structure **(2.5%)**
  - The following information should be part of the email: **(2.5%)**
    - The email of user to be notified

### Trade request update (7.5%)

When a new trade request occurs, the receiver of the trade request should be notified via email and the notification is responsible for listening to the event and send the email to the user

- AMQP listener **(2.5%)**
  - Sets up a queue called **trade-update-queue** which is bound to the **trade-update-request** routing key
- Send an email using **Mailgun** stating that one of his trade requests has been updated **(5%)**
  - The email should be setup in a proper manner using HTML structure **(2.5%)**
  - The following information should be part of the email: **(2.5%)**
    - The email of user to be notified
    - A list of receiving items
    - A list of offering items
    - Information on the other user in the trade
    - Date of issue
    - Date of modification
    - Status of the trade

### Dockerfile (5%)

A Dockerfile should be at the root of the application, containing all the requirements to run the notification service in a Docker container.

## RabbitMQ (5%)

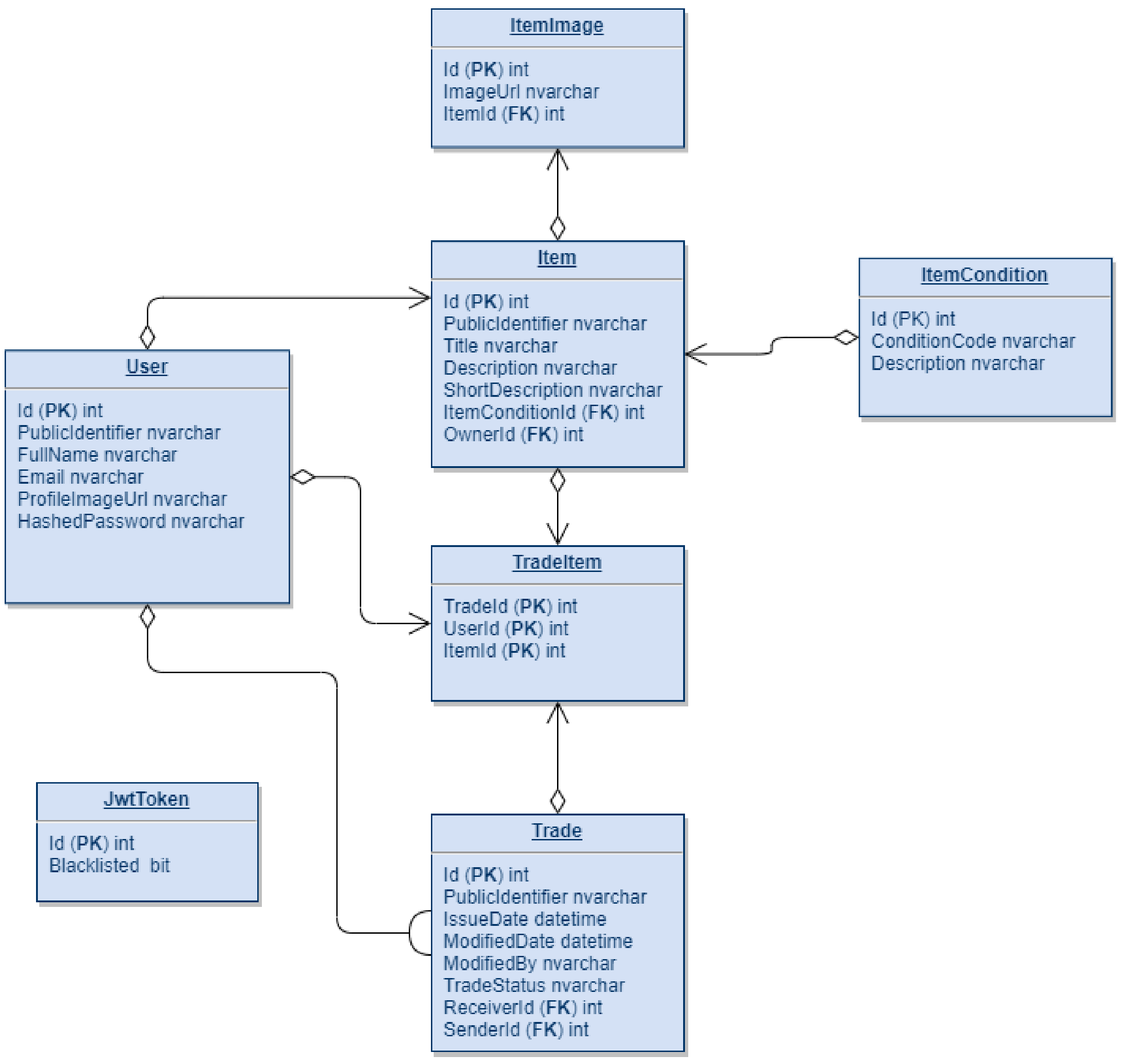
This service works as a glue between the other services to communicate with each other via AMQP.

## Docker compose (5%)

A single docker-compose.yml should be part of this structure in order to start and stop the microservice structure at will and as a whole.

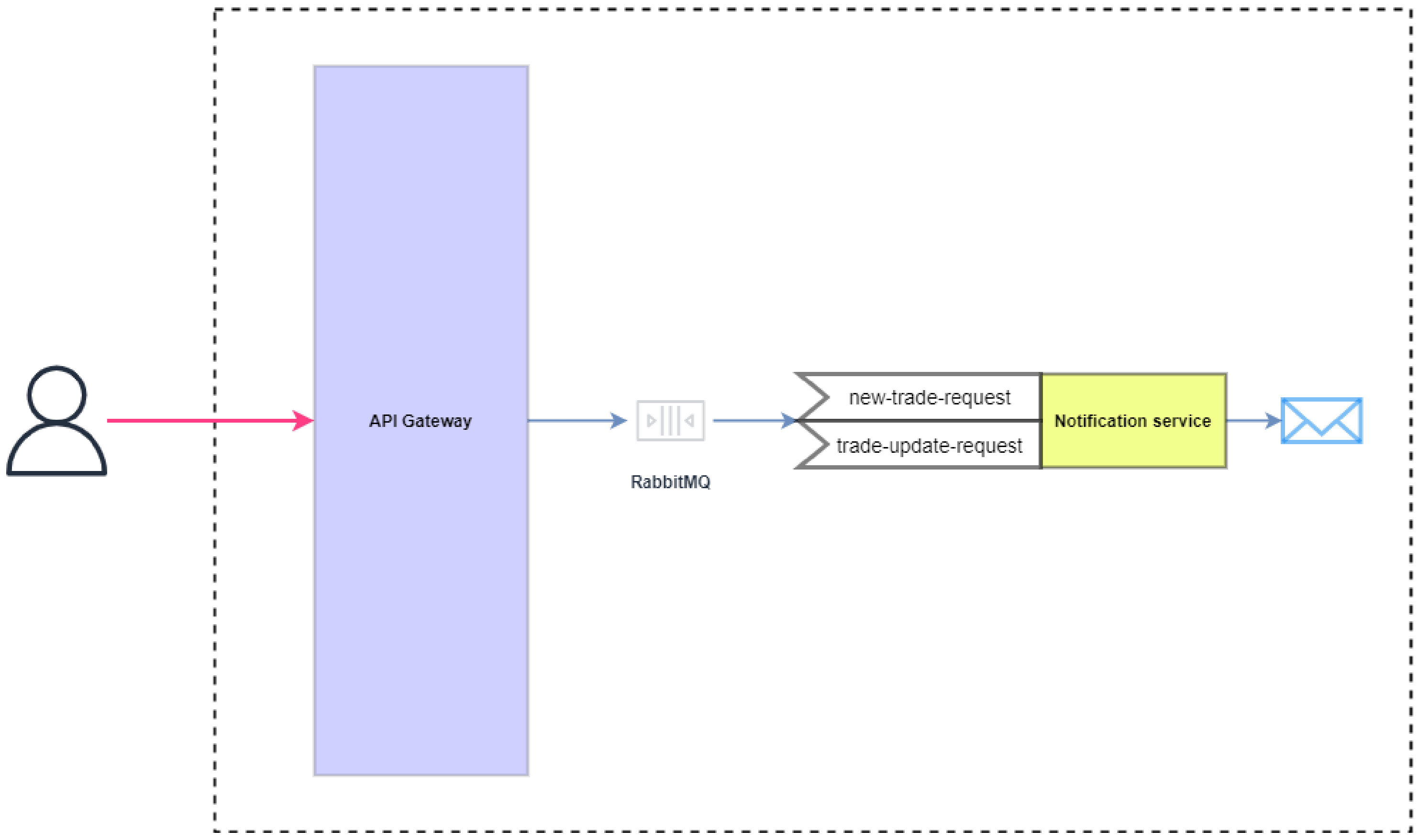
## Database diagram

Here below is a diagram of how the database should look like and you can derive from the diagram on how your entity models should be set up.



## Microservice structure

Here below is an overview of how the microservice structure should function as a whole.



## Models

Below you can see the model structure for each model within the application, this includes: **Dtos**, **InputModels** and **Enums** (*Enums are not required but can help make the code more readable*). Entity models are excluded because they can be derived from the database diagram.

### Data Transfer Object (DTOs)

- **ImageDto**
  - Id (int)
  - ImageUrl (string)
- **ItemDetailsDto**
  - Identifier (string)
  - Title (string)
  - Description (string)
  - Images (IEnumerable<ImageDto>)
  - NumberOfActiveTradeRequests (int)
  - Condition (string)
  - Owner (UserDto)
- **ItemDto**
  - Identifier (string)
  - Title (string)
  - ShortDescription (string)
  - Owner (UserDto)
- **TradeDetailsDto**
  - Identifier (string)
  - ReceivingItems (IEnumerable<ItemDto>)
  - OfferingItems (IEnumerable<ItemDto>)
  - Receiver (UserDto)
  - Sender (UserDto)
  - ReceivedDate (DateTime?)
  - IssuedDate (DateTime)
  - ModifiedDate (DateTime)
  - ModifiedBy (string)
  - Status (TradeStatus / string)
- **TradeDto**
  - Identifier (string)
  - IssuedDate (DateTime)
  - ModifiedDate (DateTime)
  - ModifiedBy (string)
  - Status (TradeStatus / string)
- **UserDto**
  - Identifier (string)
  - FullName (string)

    - Email (string)
    - ProfileImageUrl (string)
    - TokenId (int)
        - Can be JSON ignored, so it won't turn out in response to end-user

## Input models

In this section * means it is a required property.

- **ItemInputModel**
    - Title* (string)
    - ShortDescription* (string)
    - Description* (string)
    - ConditionCode* (string)
        - Must be a valid condition code: MINT, GOOD, USED, BAD or DAMAGED
    - ItemImages (IEnumerable<string>)
        - Should be a list of URLs of images
- **LoginInputModel**
    - Email* (string)
        - Should be a valid email
    - Password* (string)
        - Minimum length of 8
- **ProfileInputModel**
    - FullName (string)
        - Minimum length of 3
    - ProfileImage (IFormFile)
        - An image file (See Uploading an image for reference)
- **RegisterInputModel**
    - Email* (string)
        - Should be a valid email
    - FullName* (string)
        - Minimum length of 3
    - Password* (string)
        - Minimum length of 8
    - PasswordConfirmation* (string)
        - Minimum length of 8
        - Must be equal to **Password** field within same class
- **TradeInputModel**
    - ReceiverIdentifier* (string)
    - ItemsInTrade* (IEnumerable<ItemDto>)
        - All items included in the trade request, for both receiver and initiator of the trade

## Enums

- **TradeStatus**
    - Pending
    - Accepted
    - Declined
    - Cancelled
    - TimedOut

## AWS

A part of this project is uploading an image to **AWS S3 Bucket**, so it will become available for everyone who wishes to see this particular image. We will go through the steps of setting up our free AWS tier.

### Setting up your S3 bucket (and possibly free tier account)

1. Navigate to https://aws.amazon.com/free/
2. From there press the **Create an AWS Account** button in the top right corner and go through all the steps to create a new account
3. If everything was successful you should arrive at the **AWS Management Console**. From there search for **S3** and click on the **S3 - Scalable Storage in the Cloud**
4. From there click **Create bucket**
   a. Give your bucket a name, e.g. **tradebucket** or something similar
   b. Select the EU (Ireland) eu-west-1
   c. Uncheck the **Block all public access**
   d. All other options should remain the default
   e. Press the **Create bucket** button on the bottom of the page
5. Now you should be able to select the newly created bucket in the table within the dashboard view
6. Navigate to the **Permissions** tab
   a. Scroll down to **Bucket policy**, press **Edit** and paste the following code in there, but replace the **{name-of-bucket}** with the name of your bucket. Then save your changes

```
{
    "Version": "2008-10-17",
    "Statement": [
        {
            "Sid": "AllowPublicRead",
            "Effect": "Allow",
            "Principal": {
                "AWS": "*"
            },
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::{name-of-bucket}/*"
        }
    ]
}
```

## Fetching your API key

In order to upload an image to your **AWS S3 Bucket** you must authenticate yourself. One of the options for authentication is providing access keys (*access key ID and secret access key*)

1. Within the **AWS Management Console** you can press your username in the top right corner and select **My Security Credentials**
2. You get multiple options for authentication but within the accordion you should select **Access keys (access key ID and secret access key)** and from there **Create New Access Key**
3. This will generate an access key and will only show it once. You can press **Show Access Key** and copy both the ID and secret and store it within your **appsettings.json** file